MADlib Design Document

July 9, 2019

Contents

1 Abstraction Layers

Author Florian Schoppmann

History

- **v0.6** Replaced UML figure [Rahul Iyer]
- **v0.5** Initial revision of design document
- **v0.4** Support for function pointers and sparse-vectors
- **v0.3** C++ abstraction layer rewritten as a template library, switched to Eigen [36] as linear-algebra library
- **v0.2** Initial revision of C++ abstraction layer, incorporated Armadillo [63] as linear-algebra library

1.1 The C++ Abstraction Layer

There are a number of complexities involved in writing C or C++-based user-defined functions over a legacy DBMS like PostgreSQL, all of which can get in the way of maintainable, portable application logic. This complexity can be especially frustrating for routines whose pseudocode amounts to a short linear-algebra expression that *should* result in a compact implementation.

MADlib provides a C++ abstraction layer both to ease the burden of writing high-performance UDFs, and to encapsulate DBMS-specific logic inside the abstraction layer, rather than spreading the cost of maintenance and porting across all the UDFs in the library. In brief, the MADlib C++ abstraction currently provides five classes of functionality: type bridging, math-library integration, resource-management shims, high-level types, and templates for modular fold/reduce components.

1.1.1 Overview of Functionality

Type Bridging The basic responsibility for the C++ abstraction layer is to bridge database types to native C++ types. For a DBMS, a user-defined function implemented in a compiled language is typically nothing more but a symbol (i.e., an address) in a shared library. As such, DBMS APIs specify that UDFs must have a fixed signature, and arguments are passed as an array of pointers along with additional meta data. Hand-written C code would therefore often consist of long boilerplate code that is very specific to the underlying DBMS: Making sure that the passed data is of the correct type, copying immutable data before doing modifications, verifying array lengths, etc. The C++ abstraction layer encapsulates all this within the recursive `AnyType` class that can contain either a primitive type (like, e.g., `int` or `double`) or multiple other values of type `AnyType` (for representing a composite type). This encapsulation works both for passing data from the DBMS to the C++ function, as well as returning values back from C++. To give an example: A simple, portable, and completely type-safe (though arguably not very useful) function that adds two numbers could be implemented with essentially as little code as in a high-level scripting language:

```
1: AnyType
2: sum_two_doubles::run(AnyType& args) {
3:     return args[0].getAs<double>()
```

```
4:             + args[1].getAs<double>();
5:  }
```

Math-Library Integration and Performance SQL comes without any native support for vector and matrix operations. This presents challenges at two scales. At a macroscopic level, matrices must be intelligently partitioned into chunks that can fit in memory on a single node. At a microscopic scale, the database engine must invoke efficient linear-algebra routines on the pieces of data it gets in core. To this end, the C++ abstraction layer incorporates the very performant linear-algebra library Eigen [36]. Most importantly, it provides additional type bridges that do not involve memory copying and thus are very efficient: For instance, double-precision arrays in the DBMS are the canonic way to represent real-valued vectors. Therefore, the C++ abstraction layer not just provides an array-to-array bridge but also maps DBMS arrays to Eigen vectors. The bridged types can be used with all of the very sophisticated vector and matrix operations provided by Eigen.

Incorporating proven third-party libraries moreover makes it easy for MADlib developers to write correct and performant code: For instance, the Eigen linear-algebra library contains well-tested and well-tuned code that makes use of the SIMD instruction sets (like SSE) found in today's CPUs. Recent versions of Eigen even allow coupling with proprietary high-performance mathematical routines like the Intel Math Kernel Library.

Likewise, the C++ abstraction layer itself has been tuned for efficient value marshaling. Some examples include: All type bridges are aware of mutable and immutable objects and avoid making copies whenever possible. DBMS-catalogue lookups occur only once per query and are then minimized by caching. Moreover, the C++ abstraction layer is written as a template library and with the goal of reducing the runtime and abstraction overhead to a minimum. In particular, it takes extra steps to avoid memory allocation whenever possible.

Resource-Management Shims Another aspect of the C++ abstraction layer is to provide a safe and robust runtime environment with a standard interface. For instance, PostgreSQL maintains a hierarchy of memory contexts: When a query is started, a new memory context is created and all transient memory allocations are supposed to occur within this context. When the query ends, disposing of the query context provides a simple and effective way of garbage collection. The C++ abstraction layer makes sure that such modi operandi are followed. On the other hand, the C++ abstraction layer also facilitates writing C++ code with a well-defined interface. This is particularly necessary if (as is typically the case) a DBMS only provides a C plugin interface: In that case it is important that exceptions, signals, etc. do not cross runtime boundaries.

High-level types A second responsibility of the abstraction layer is to help compensating for SQL's lack of higher-order logic: For instance, an `AnyType` object can contain a `FunctionHandle`, which points to a user-defined function. With the syntactic sugar possible in C++, this essentially makes in-database functions first-class objects like they commonly are in modern programming languages. Internally, the abstraction layer maps UDFs to their object ID in the database, and it takes care of looking up the function in the database catalog, verifying argument lists, ensuring type-safety, etc.

Likewise, there is the need to pass internal data structures from one UDF to another (i.e., possibly through the DBMS) in a performant and portable way. While DBMSs like PostgreSQL support user-defined composite types, converting into them (or even using them in internal C++ code) is slow, creates dependencies on MADlib-specific type-bridging classes, and hinders code reuse

in/from other projects than MADlib. The C++ abstraction layer therefore contains the recursive `DynamicStruct` class that provides a C++ struct/class interface around a stream of bytes. This class is very generic and can easily be used with any contiguous blocks of memory. Compared to the alternative of using existing serialization and deserialization classes, we expect `DynamicStruct` to be far better performing, as it modifies constant-length elements directly in the byte stream.

Modular Fold/Reduce Components The most basic building block in the macro-programming of MADlib is the use of user-defined aggregates (UDAs). In general, aggregates—and the related window functions—are the natural way in SQL to implement mathematical functions that take as input the values of an arbitrary number of rows. Unfortunately, concrete extension interfaces for user-defined aggregates vary widely across vendors and open-source systems. Nonetheless, the aggregation paradigm (or in functional programming terms, "fold and reduce") is natural and ubiquitous, and in most widely-used DBMSs (e.g., in PostgreSQL, MySQL, Greenplum, Oracle, SQL Server, Teradata) a user-defined aggregate consists of a well-known pattern of two or three user-defined functions:

i) A *transition function* that takes the current transition state and a new data point. It combines both into into a new transition state. The transition function is equivalent to the "combining" function passed to linear *left-fold* functions in functional-programming languages.

ii) An optional *merge function* that takes two transition states and computes a new combined transition state. This function is only needed for parallel execution. In functional-programming terms, a merge operation is a tree-like fold.

iii) A *final function* that takes a transition state and transforms it into the output value.

Clearly, a user-defined aggregate is inherently data-parallel if the transition function is associative and the merge function returns the same result as if the transition function was called repeatedly for every individual element in the second state.

Since the fold/reduce computational model is so ubiquitous and we anticipate the need to share code with other projects, fold/reduce code should not be interwoven with the DBMS interface. That is, fold/reduce-components should be implemented as independent C++ classes (possibly as generic template classes), without dependencies on MADlib-specific type-bridging classes like `AnyType`. However, fold/reduce components need to store their state as objects that the backend can understand—for maximum portability, all state information must even reside in a single contiguous block of memory. The C++ abstraction layer therefore provides a recursive class `DynamicStruct` that can contain objects both of primitive data types as well as objects of variable length, including other objects of `DynamicStruct`. This solution is more performant than serialization and deserialization, because it allows fixed-length datums to be modified directly in the block of memory.

1.1.2 Type Bridging

1.1.2.1 Class `AnyType`

`AnyType` is a container type for all values that are passed between the DBMS and C++ code. This also includes values passed to or returned from UDFs invoked as `FunctionHandle`. An `AnyType` object represents one of three kinds of values:

i) NULL

ii) A simple value. E.g., this may be a value of a primitive data type like `int` or `double`, or a value of some abstraction-layer type like `FunctionHandle`.

iii) A composite value (i.e., a tuple). In this case, the `AnyType` object contains a list of other `AnyType` objects. Tuple elements are not named but instead accessed by index.

`AnyType` objects should be explicitly instantiated only for returning values to the backend or for invoking a `FunctionHandle`. Implementations may choose to have different internal representations for values received from the backend and values from UDF code. The two constructors below only pertain to instantiations within UDF code. Constructors used by internal abstraction-layer code are implementation-specific.

Member functions

- `AnyType()`

  Default constructor. Initializes this object as NULL. This constructor must also be used for initializing a composite object. After construction, `operator<<()` can be used to append values to the composite object.

- `template <class T> AnyType(const T& inValue)`

  Template constructor (will not be used as copy constructor). This constructor will be invoked when initializing this object with an arbitrary value (excluding composite types).

- `template <class T> T getAs()`

  Convert this object to the type specified as template argument.

- `AnyType operator[](uint16_t inID) const`

  If this object is a composite value, return the element with index `inID`. To the user, `AnyType` is a fully recursive type: An `AnyType` object may contain a composite value, in which case it is composed of a number of other AnyType objects.

  This method will raise an error if this object does not contain a composite value.

- `uint16_t numFields() const`

  Return the number of elements in the tuple. If this object contains NULL, return 0; if it contains a simple value, return 1.

- `bool isNull() const`

  Return if this object is NULL.

- `bool isComposite() const`

  Return if this object is a composite value.

- `AnyType& operator<<(const AnyType& inValue)`

  If this object is NULL, or a composite value that has previously been constructed with the default constructor, add an element to this object. Otherwise, raise an error.

Non-Member Functions

- `AnyType Null()`

  Return an AnyType object representing NULL.

- ```
  template <class T> T AnyType_cast(const AnyType& inValue)
  template <class T> T AnyType_cast(const T& inValue)
  template <class T> T AnyType_cast(T& inValue)
  ```

  Explicit cast that converts `AnyType` objects to a target type `T`, but leaves values that already are of type `T` unaffected.

  Sometimes it is desirable to write generic code that works on both an `AnyType` object as well as a value with a concrete type. For instance, a `FunctionHandle` always returns an `AnyType` object. In generic code, however, a `FunctionHandle` might as well be replaced by just a call to a "normal" functor or a C++ function pointer, both of which typically return concrete types (e.g., `double`).

  In generic code, we could write `AnyType_cast<double>(func())` so that if the template type of `func` is a `FunctionHandle`, we have an explicit conversion to `double`, whereas if `func` is just a function pointer, the return value of `func()` passes unchanged.

1.1.3 Math-Library Integration

1.1.3.1 Class `HandleMap`

Requirements

- `EigenType` The Eigen type that this `HandleMap` will wrap. Examples are `Eigen::VectorXd` or `Eigen::MatrixXd`.
- `Handle` Type conforming to `ContiguousDataHandle` concept. The two types `EigenType::Scalar` and `Handle::element_type` must coincide.
- `MapOptions` Passed as template parameter `MapOptions` to `Eigen::Map`.

Types

- `Index`: `EigenType::Index`

Member functions

- ```
  // constructors
  HandleMap(const Handle& inHandle) // (1)
  HandleMap(const Eigen::MapBase<Derived>& inMappedData) // (2)
  HandleMap(const Handle &inHandle, Index inNumElem) // (3)
  HandleMap(const Handle &inHandle, Index inNumRows, Index inNumCols) // (4)
  ```

  Constructor (1) constructs an empty `HandleMap` that points to NULL. Constructor (2) constructs a `HandleMap` that is backed by a contiguous memory block within an existing Eigen matrix (or vector). Note that not all of Eigen's block operations return contiguous memory blocks. E.g., while the `col()` method returns contiguous blocks if column-oriented storage is used, the `row()` method does not! Constructor (3) and (4) construct a `HandleMap` that is backed by a `Handle`.

- `HandleMap& operator=(const HandleMap& other)`

  Assign another `HandleMap` to this object. This does not change any references or pointers, but copies all elements, i.e., the behavior is identical to other Eigen objects.

- ```
  HandleMap& rebind(const Handle& inHandle);
  HandleMap& rebind(const Handle& inHandle, Index inSize);
  HandleMap& rebind(const Handle& inHandle, Index inRows, Index inCols);
  HandleMap& rebind(Index inSize);
  HandleMap& rebind(Index inRows, Index inCols);
  ```

  Change the handle that is backing this object. All except the first form may also be used to change the dimensions of the matrix (or vector) represented by this object.

Concrete Types

- `MappedMatrix`: `HandleMap<const Matrix, TransparentHandle<double> >`
- `MutableMappedMatrix`: `HandleMap<Matrix, TransparentHandle<double, Mutable> >`
- `NativeMatrix`: `HandleMap<const Matrix, ArrayHandle<double> >`
- `MutableNativeMatrix`: `HandleMap<Matrix, MutableArrayHandle<double> >`
- The corresponding four definitions for `ColumnVector` instead of `Matrix`.

1.1.4 Resource-Management Shims

1.1.4.1 Class `Allocator`

Member functions

1.1.4.2 Class `NativeRandomNumberGenerator`

Member functions

1.1.5 High-Level Types

1.1.5.1 Class `FunctionHandle`

A `FunctionHandle` is a function "pointer" to a UDF. A compliant implementation might just store the object ID of the UDF in the database, and upon invocation look up the function in the database catalog, do argument verification, retrieve a function pointer in memory and finally invoke this function pointer.

There are several options for implementations to improve performance. First, if function `funcPtr()` returns a non-NULL value, the UDF is implemented using the C++ abstraction layer and can be called directly, without going through the usual backend functions. Second, the `invoke` function is *not* `const`, so implementations can cache metadata (and therby modify the `FunctionHandle` object).

Types

- `udf_ptr`: `AnyType (*)(AnyType&)`
  ```

Member functions

- `udf_ptr funcPtr();`

  If the UDF is a function written using the C++ abstraction layer, implementations may return a pointer to the C++ function. Alternatively, an implementation may always return NULL. Callers must not rely on `funcPtr` returning non-NULL values.

- `FunctionHandle& setFunctionCallOptions(uint32_t inFlags)`
  `FunctionHandle& unsetFunctionCallOptions(uint32_t inFlags)`
  `uint32_t getFunctionCallOptions() const`

  Set or get the current function call options. Options are a bit field of properties as defined above.

- `AnyType invoke(AnyType& args)`

  Invoke the UDF with the given arguments. Note that `args` has to be a composite value.

- `AnyType operator()()`
  `AnyType operator()(AnyType& arg1, ..., AnyType& argn)`

  Convenience method. Call `invoke` with the given arguments combined into one composite value.

1.1.5.2 Concept `ContiguousDataHandle`

A `ContiguousDataHandle` is an opaque pointer to contiguous data that may be augmented by metadata. For instance, a `ContiguousDataHandle` may wrap a DBMS-native array that contains both a pointer to the array data, as well as information about the array's size. Since elements are stored contiguously, they can also be accessed using offsets on regular pointers to elements.

Requirements

- `T` Element type

Types

- `element_type`: `T`

Member Functions

- `const element_type* ptr() const`

  Return a pointer to the first element.

- `const element_type& operator[](size_t inIndex) const`

  Return an element.

Specialized Concepts The concept `MutableContiguousDataHandle` contains also the following non-const member functions:

- `element_type* ptr()`
- `element_type& operator[](size_t inIndex)`

The concept `SizedContiguousDataHandle` contains also the following member function:

- `size_t size() const`

  Return the number of elements in this object.

The concept `MutableSizedContiguousDataHandle` combines both `MutableContiguousDataHandle` and `SizedContiguousDataHandle`.

1.1.5.3 Class `Ref`

`Ref` objects are conceptually equivalent to normal C++ references. However, they allow *rebinding* to a different target.

Requirements

- `T` Target type
- `IsMutable` Boolean parameter indicating if objects of this type can be used to modify the target

Types

- `value_type`: `T`

Member Functions

- `Ref& rebind(val_type* inPtr)`

  Rebind this reference to a different target.

- `operator const val_type&() const`

  Return a const-reference to the target.

- `const val_type* ptr() const`

  Return a const-pointer to the target.

- `bool isNull() const`

  Return if this reference has been bound to a target.

If `IsMutable == true`, then `Ref` also contains the following non-const member functions:

- `operator val_type&()`

- `val_type* ptr()`

Moreover it contains:

- `Ref& operator=(Ref& inRef)`
  `Ref& operator=(const val_type& inValue)`

  Assign the target value of `inRef` or `inValue` to the target of this object.

  It is important to define the first assignment operator because C++ will otherwise perform an assignment as a bit-by-bit copy. Note that this default `operator=` would be used even though there is a conversion path through `dest.operator=(orig.operator const val_type&())`.

1.1.5.4 Class `ByteStream`

`ByteStream` objects are similar to `std::istream` objects in that they are used to *bind* (as opposed to *read* in the case of `std::istream`) references to positions in byte sequences. `operator>>()` functions are provided for users of `ByteStream` objects. Each `ByteStream` object controls a `ByteStreamHandleBuf`, which in turn controls a block of memory (the storage/buffer) and has a current position.

A `ByteStream` object can be in *dry-run* mode, in which case `operator>>` invocations move the current position, but no rebinding takes place. Dry-run mode is used, e.g., to determine the storage size needed to hold a `DynamicStruct`.

Member Functions

- `template <size_t Alignment>`
  `size_t seek(std::ptrdiff_t inOffset, std::ios_base::seekdir inDir) // (1)`
  `size_t seek(size_t inPos) // (2)`
  `size_t seek(std::ptrdiff_t inOffset, std::ios_base::seekdir inDir) // (3)`

  Move the current position in the stream. Variant (1) rounds the new position up to the next multiple of `Alignment`.

- `size_t available() const`

  Return the number of characters between the current position and the end of the stream.

- `const char_type* ptr() const`

  Return a pointer to the beginning of the buffer.

- `size_t size() const`

  Return the size of the buffer.

- `size_t tell() const`

- `std::ios_base::iostate rdstate() const`
  `bool eof() const`

  Return status information about the stream, in a fashion similar to `std::istream`.

- `bool isInDryRun() const`

  Return if the stream is in dry-run mode.

- `template <class T> const T* read(size_t inCount = 1)`

  Advance the current position in the buffer to the next address suitable to read a value of type `T` and return that address.

Non-Member Functions

- `template <class Reference>`
  `ByteStream& operator>>(ByteStream& inStream, Reference& inReference)`

  Bind a reference to the next suitable address in the buffer. Internally, this function calls `read<typename Reference::val_type>(inReference.size())`.

1.1.5.5 Class `ByteStreamHandleBuf`

`ByteStreamHandleBuf` objects are similar to `std::streambuf` objects in that they are in charge of providing reading functionality from certain types of byte sequences. Unlike `std::streambuf`, however, reading refers to *binding* a reference to the current position in the byte sequence.

`ByteStreamHandleBuf` objects are associated with a *storage* objects, which is of a class conforming to the `ContiguousDataHandle` concept.

Types

- `Storage_type`: Type conforming to `ContiguousDataHandle` concept.

Constants

- `isMutable`: `Storage_type::isMutable`, i.e., `true` if `Storage_type` also conforms to the `MutableContiguousDataHandle` concept, and `false` if not.

Member Functions

- `ByteStreamHandleBuf(size_t inSize) // (1)`
  `ByteStreamHandleBuf(const Storage_type& inStorage) // (2)`

  Constructor (1) constructs an empty buffer initialized with `inSize` zero characters. Constructor (2) initializes a buffer using existing storage.

- `size_t seek(size_t inPos)`
  `const char_type* ptr() const;`
  `size_t size() const`
  `size_t tell() const`

  Change the current position in the buffer, return the start of the buffer, return the size of of the buffer, and return the current position in the buffer.

The following member functions are only present if `isMutable == true`.

- `void resize(size_t inSize, size_t inPivot)`

  Change the size of the buffer, and preserve the old buffer in the following way: Denote by s the old size of the buffer, by n the new size `inSize`, and by p the pivot `inPivot`. Then bytes $[0, p)$ will remain unchanged, bytes $[p, p + n - s)$ will be initialized with 0, and bytes $[p + (n - s), s + (n - s))$ will contain the old byte range $[p, s)$.

1.1.5.6 Concept `DynamicStructContainer`

A `DynamicStructContainer` may contain member variables of type `DynamicStruct`. In order for automatic inclusion in the byte stream of a `DynamicStruct`, the `DynamicStructContainer` has to be provided has the `Container` template parameter of the `DynamicStruct`.

Types

- `RootContainer_type`: Type conforming to `DynamicStructContainer` concept
- `Storage_type`: Type conforming to `ContiguousDataHandle` concept.
- `ByteStream_type`: A `ByteStream` class

Constants

- `isMutable`: `Storage_type::isMutable`, i.e., `true` if `Storage_type` also conforms to the `MutableContiguousDataHandle` concept, and `false` if not.

Member functions

- `void initialize()`

  Initialize the object. The default implementation does nothing.

- `const RootContainer_type& rootContainer() const`

  Return the root-level container.

- `const Storage_type& storage() const`

  Return the storage object.

- `const ByteStream_type& byteStream() const`

  Return the stream object.

Specialized Concepts The concept `MutableDynamicStructContainer` also has the non-const member functions:

- `RootContainer_type& rootContainer()`
- `RootStorage_type& storage()`
- `ByteStream_type& byteStream()`

- ```
  template <class SubStruct>
  void setSize(SubStruct &inSubStruct, size_t inSize)
  ```

Moreover, the types `RootContainer_type` and `Storage_type` have to conform to the respective `Mutable` concepts, and `ByteStream_type` has to be a mutable `ByteStream` class.

1.1.5.7 Class `DynamicStruct`

A `DynamicStruct` gives a C++ struct/class interface to a byte stream. Modifying member variables directly modifies the byte stream, without any need for serialization and deserialization. Member variables may have variable length that may be changed even after object creation. Moreover, `DynamicStruct` is a recursive type in it also allows member variables of type `DynamicStruct`.

Requirements

- `Derived` Type of the derived class. Used for static polymorphism.
- `Container` Type conforming to `DynamicStructContainer` concept.

Types

- `Init_type`: `Storage_type` if the `Container_type` is a root-level container, and `Container_type` otherwise.

  This is a convenience definition: While in general a `ContiguousDataHandle` would be initialized with its container, a top-level `DynamicStruct` is initialized directly with a `ContiguousDataHandle` (without having to instantiate a root-level container first).
- `Storage_type`: Type conforming to `ContiguousDataHandle` concept
- `Container_type`: Type conforming to `DynamicStructContainer` concept.
- `ByteStream_type`: `Container_type::ByteStream_type`, i.e., a `ByteStream` class as defined by the container

Member Functions

- ```
  DynamicStruct(Init_type& inInitialization) // constructor
  ```

- ```
  template <class OtherDerived>
  DynamicStruct& copy(
      const DynamicStruct<
          OtherDerived,
          typename OtherDerived::Container_type
      >& inOtherStruct)
  ```

  Copy the value of `inOtherStruct` into this object. Copying will be performed bitwise, and all member variables will be rebound afterwards.

The following member functions are only present if `isMutable == true`.

- ```
  void setSize(size_t inSize)
  ```

  Set the size of this object.

Non-Member Functions

- `ByteStream_type& operator>>(ByteStream_type& inStream, Derived& inStruct)`

  Bind `inStruct` to the byte stream at the current position.

1.1.6 Modular Fold/Reduce Components

1.1.6.1 Concept `Accumulator`

A class implementing the `Accumulator` concept derives from `DynamicStruct` and stores a transition state. It contains methods for adding a new tuple to the transition state (equivalent to the *transition function*) as well as adding a new transition state (equivalent to the *merge function*).

Requirements

- `Container` Type conforming to `DynamicStructContainer` concept.

Types Inherited from `DynamicStruct`:

- `Init_type`: Type passed to constructor. It is only needed to pass through the constructor argument to the `DynamicStruct` base class.
- `ByteStream_type`: `Container::ByteStream_type`, i.e., the concrete `ByteStream` type as defined by `Container`. A reference of this type is passed to the `bind()` function.

Member Functions

- `Accumulator(Init_type& inInitialization) // constructor`

  The constructor is expected to call `DynamicStruct::initialize()`, which eventually will call the `bind()` method.[1]

- `void bind(ByteStream_type& inStream)`

  Bind all elements of the state to the data in the stream.

  Implementations bind a member variable `x` to the current position in the stream by running `inStream >> x`. Note that even after running `operator>>()` on a member variable, there is no guarantee yet that the variable can indeed be accessed. Instead, if the end of `inStream` has been reached, it would still be uninitialized. It is crucial to first check this.

  Provided that this methods correctly lists all member variables, all other methods can, however, rely on the fact that all variables are correctly initialized and accessible.

- `Accumulator& operator<<(const tuple_type& inTuple)`

  Add a new tuple to this transition-state object.

- `template <class OtherContainer>`
  `Accumulator& operator<<(const Accumulator<OtherContainer>& inOther)`

[1] Unfortunately, the need for defining an `initialize()` member cannot be removed. No super class can safely call `initialize()` because the `Accumulator` object has not been completely constructed at that time, yet.

Add a new transition state to this transition-state object.

`OtherContainer` must conform to the `DynamicStructContainer` concept.

- `template <class OtherContainer>`
  `Accumulator& operator=(const Accumulator<OtherContainer>& inOther)`

  The assignment operator must be implemented, because the implicit assignment operator is not enough: Whenever the length of the `Accumulator` changes, member variables have to be rebound. The `copy` method in `DynamicStruct` takes care of this and should be explicitly called instead.

  `OtherContainer` must conform to the `DynamicStructContainer` concept.

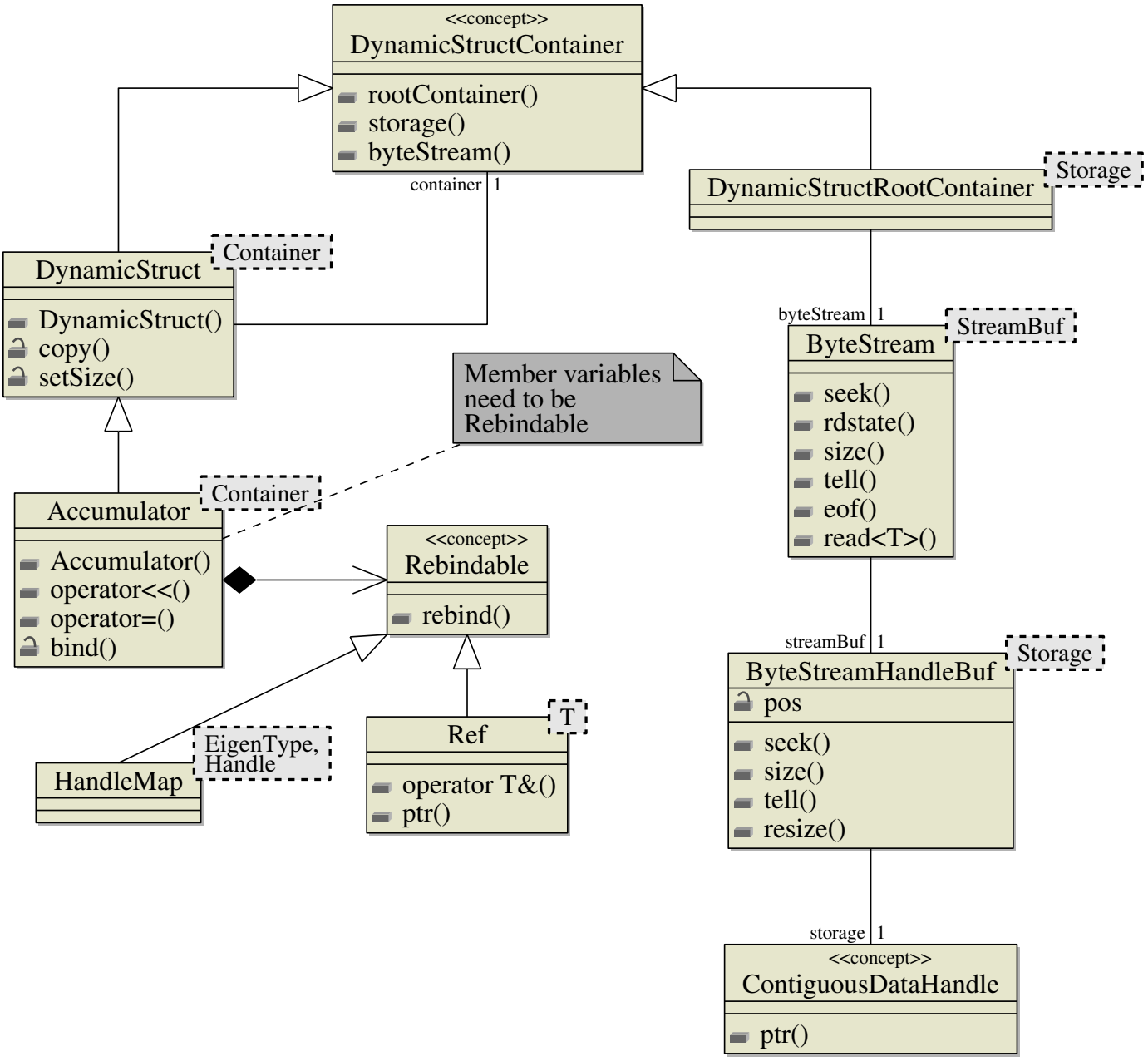

Figure 1.1: Class diagram for modular fold/reduce

2 Sampling

Author Florian Schoppmann

History **v0.5** Initial revision

2.1 Sampling without Replacement

Given a list of known size n through that we can iterate with arbitrary increments, sampling m elements without replacement can be implemented in time $O(m)$, i.e., proportional to the sample size and independent of the list size [73]. Even if the list size n is unknown in advance, sampling can still be implemented in time $O(m(1 + \log \frac{n}{m}))$ [72].

While being able to iterate through a list with arbitrary increments might seem like a very modest requirement, it is still not always an option in real-world databases (e.g., in PostgreSQL). It is therefore important to also consider more constrained algorithms.

2.1.1 Probabilistic Sampling

Probabilistic sampling selects each element in the list with probability p. Hence, the sample size is a random variable with Binomial distribution $B(n, p)$ and expectation np. The standard deviation is $\sqrt{np(1-p)}$, i.e., approximately $\sqrt{np}$ if p is small. In many applications a fixed sample size m is needed, however. In this case, we could choose p slightly larger than m/n, so that with high probability at least m items are selected. Items in excess of m are then discarded.

2.1.1.1 Formal Description

In the following, we discuss how to choose p so that with high probability at least m elements are sampled, but also not "much" more than m (in fact, only $O(\sqrt{m})$ more in expectation).

In mathematical terms: What is a lower bound on the probability p so that for a random variable $X \sim B(n, p)$ we have that $\Pr[X < m] \leq \epsilon$? We use the Chernoff bound for a fairly good estimate. It says

$$\Pr[X < (1-\delta) \cdot \mu] \leq \exp\left(\frac{-\delta^2}{2} \cdot \mu\right),$$

where $\mu = np$ is the expectation of X, and $\delta \geq 0$. We set $m = (1-\delta) \cdot \mu$, or equivalently $\delta = \frac{\mu - m}{\mu}$. This yields

$$\Pr[X < m] \leq \exp\left(\frac{-(\mu - m)^2}{2\mu}\right). \tag{2.1.1}$$

We want the right-hand side of (2.1.1) to be bounded by ϵ from above. Rearranging this gives

$$\mu \geq m - \ln(\epsilon) + \sqrt{\ln^2(\epsilon) - 2m\ln(\epsilon)}\,.$$

Since $p = \mu/n$, this immediately translates into a lower bound for p. For instance, suppose we require $\epsilon = 10^{-6}$, i.e., we want the probability of our sample being too small to be less than one in a million. $\ln(10^{-6}) \approx -13.8$, so we could choose

$$p \geq \frac{m + 14 + \sqrt{196 + 28m}}{n}.$$

Note that the bound on μ does not depend on n. So in expectation, only $O(m + \sqrt{m})$ items are selected. At the same time, at least m items are selected with very high probability.

2.1.1.2 Implementation in SQL

In real-world DBMSs, probabilistic sampling has the advantage that it is trivially data-parallel. Discarding excessive items can be done using the well-known `ORDER BY random() LIMIT` idiom. Tests show that PostgreSQL is very efficient in doing the sorting (today's CPUs can easily sort 1 million numbers in less than a couple hundred milliseconds). In fact, the sorting cost is almost not measurable if the sample size is only at the scale of several million or less. Since `ORDER BY random() LIMIT` is an often-used idiom, there is also hope that advanced optimizers might give it special treatment. Put together, in order to sample m random rows uniformly at random, we write:

```
1: SELECT * FROM list WHERE random() < p ORDER BY random() LIMIT m
```

If necessary, checks can be added that indeed m rows have been selected.

2.1.2 Generating a Random Variate According to a Discrete Probability Distribution

In practice, probability distributions are often induced by weights (that are not necessarily normalized to add up to 1). The following algorithm is a special case of the "unequal probability sampling plan" proposed by Chao [20]. Its idea is very similar to reservoir sampling [53].

2.1.2.1 Formal Description

Algorithm `WeightedSample`(A, w)
Input: Finite set A, Mapping w of each element $a \in A$ to its weight $w[a] \geq 0$
Output: Random element $Sample \in A$ sampled according to distribution induced by w

1: $W \leftarrow 0$
2: **for** $a \in A$ **do**
3: $\quad W \leftarrow W + w[a]$
4: $\quad$ **with** probability $\frac{w[a]}{W}$ **do**
5: $\quad\quad Sample \leftarrow a$

Runtime $O(n)$, single-pass streaming algorithm

Space $O(1)$, constant memory

Correctness Let $a_1, \ldots, a_n$ be the order in which the algorithm processes the elements. Denote by $Sample_t$ the value of $Sample$ at the end of iteration $t \in [n]$. We prove by induction over t that it holds for all $i \in [t]$ that $\Pr[Sample_t = a_i] = \frac{w[a_i]}{W_t}$ where $W_t := \sum_{j=1}^{t} w[a_j]$.

The base case $t = 1$ holds immediately by lines 4–5. To see the induction step $t - 1 \to t$, note that $\Pr[\mathit{Sample}_t = a_t] = \frac{w[a_t]}{W_t}$ (again by lines 4–5) and that for all $i \in [t-1]$

$$\Pr[\mathit{Sample}_t = a_i] = \Pr[\mathit{Sample}_t \neq a_t] \cdot \Pr[\mathit{Sample}_{t-1} = a_i] \overset{\text{IH}}{=} \left(1 - \frac{w[a_t]}{W_t}\right) \cdot \frac{w[a_i]}{W_{t-1}} = \frac{w[a_i]}{W_t}\,.$$

Scalability The algorithm can easily be transformed into a divide-and-conquer algorithm, as shown in the following.

Algorithm `RecursiveWeightedSample`(A_1, A_2, w)
Input: Disjoint finite sets A_1, A_2, Mapping w of each element $a \in A_1 \cup A_2$ to its weight $w[a] \geq 0$
Output: Random element $\mathit{Sample} \in A_1 \cup A_2$ sampled according to distribution induced by w

1: $\tilde{A} \leftarrow \emptyset$
2: **for** $i = 1, 2$ **do**
3: $\quad \mathit{Sample}_i \leftarrow$ `WeightedSample`(A_i, w)
4: $\quad \tilde{A} \leftarrow \tilde{A} \cup \{\mathit{Sample}_i\}$
5: $\quad \tilde{w}[\mathit{Sample}_i] \leftarrow \sum_{a \in A_i} w[a]$
6: $\mathit{Sample} \leftarrow$ `WeightedSample`$(\tilde{A}, \tilde{w})$

Correctness Define $W_i := \sum_{a \in A_i} w[a]$. Let $a \in A_i$ be arbitrary. Then $\Pr[\mathit{Sample} = a] = \Pr[\mathit{Sample}_i = a] \cdot \Pr[\mathit{Sample} \in A_i] = \frac{w[a]}{W_i} \cdot \frac{W_i}{W} = \frac{w[a]}{W}$.

Numerical Considerations

- When Algorithm `WeightedSample` is used for large sets A, line 3 will eventually add two numbers that are very different in size. Compensated summation should be used [56].

2.1.2.2 Implementation as User-Defined Aggregate

Algorithm `WeightedSample` is implemented as the user-defined aggregate `weighted_sample`. Data-parallelism is implemented as in Algorithm `RecursiveWeightedSample`.

Input The aggregate expects the following arguments:

Column	Description	Type
`value`	Row identifier, each row corresponds to an $a \in A$. There is no need to enforce uniqueness. If a value occurs multiple times, the probability of sampling this value is proportional to the sum of its weights.	*generic*
`weight`	weight for row, corresponds to $w[a]$	floating-point

While it would be desirable to define a user-defined aggregate with a first argument of generic type, this would require a generic transition type (see below). Unfortunately, this is currently not supported in PostgreSQL and Greenplum. However, the internal C++ accumulator type is a generic template class, i.e., only the SQL interface contains redundancies.

Output Value of column `id` in row that was selected.

Components

- Transition State:

Field Name	Description	Type
`weight_sum`	corresponds to W in Algorithm `WeightedSample`	floating-point
`sample`	corresponds to *Sample* in Algorithm `WeightedSample`, takes value of column `value`	*generic*

- Transition Function (`state, id, weight`): Lines 3–5 of Algorithm `WeightedSample`
- Merge Function (`state1, state2`): It is enough to call the transition function with arguments (`state1, state2.sample_id, state2.weight_sum`)

Tool Set While the user-defined aggregate is simple enough to be implemented in plain SQL, we choose a C++ implementation for performance. **In the future, we want to use compensated summation. (Not documented yet.)**

2.2 Sampling with Replacement

In Postgres, with the help of the `row_number()` window function, we could achive sampling with replacement by joining the input table with a list of randomly-generated numbers using row numbers.

2.2.1 Assign Row Numbers for Input Table

```
SELECT (row_number() OVER ())::integer AS rn, * FROM $input_table;
```

2.2.2 Generate A Row Number Sample Randomly

```
SELECT ceiling((1 - random()) * $size)::integer AS rn FROM generate_series(1, $size) s;
```

2.2.3 Join to Generate Sample

```
SELECT *
FROM
(
    SELECT (row_number() OVER ())::integer AS rn, * FROM $input_table
) src
JOIN
(
    SELECT ceiling((1 - random()) * $size)::integer AS rn FROM generate_series(1, $size) s
) sample_rn
USING (rn);
```

3 Matrix Operations

Author Florian Schoppmann

History **v0.5** Initial revision

While dense and sparse matrices are native objects for MADlib, they are not part of the SQL standard. It is therefore essential to provide bridges between SQL types and MADlib types, as well provide a ground set of primitive functions that can be used in SQL.

3.1 Constructing Matrices

3.1.1 Construct a matrix from columns stored as tuples

Let $X = (x_1, \ldots, x_n) \subset \mathbb{R}^m$. `matrix_agg`$(X)$ returns the matrix $(x_1 \ldots x_n) \in \mathbb{R}^{m \times n}$.

3.1.1.1 Implementation as User-Defined Aggregate

	Name	Description	Type
In	`x`	Vector $x_i \in \mathbb{R}^m$	floating-point vector
Out		Matrix $M = (x_1 \ldots x_n) \in \mathbb{R}^{m \times n}$	floating-point matrix

3.2 Norms and Distances

3.2.1 Column in a matrix that is closest to a given vector

Let $M \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^m$, and *dist* be a metric. `closest_column`$(M, x, dist)$ returns a tuple (i, d) so that $d = dist(x, M_i) = \min_{j \in [n]} dist(x, M_j)$ where M_j denotes the j-th column of M.

3.2.1.1 Implementation as User-Defined Function

	Name	Description	Type
In	`M`	Matrix $M \in \mathbb{R}^{m \times n}$	floating-point matrix
In	`x`	Vector $x \in \mathbb{R}^m$	floating-point vector
In	`dist`	Metric to use	function
Out	`column_id`	index i of the column of M that is closest to x	integer
Out	`distance`	$dist(x, M_i)$	floating-point

4 Linear Systems

Authors Srikrishna Sridhar

History **v1.0** Initial version

4.1 Introduction

In this document, we describe solution methods for systems of a consistent linear equations.

$$Ax = b \tag{4.1.1}$$

where $x \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. We assume that all rows of A are non-zero. We denote the rows of A and b by a_i^T and b_i, respectively. This can be written as

$$A = \begin{bmatrix} a_1^T & a_2^T & \dots & a_m^T \end{bmatrix} \qquad b = \begin{bmatrix} b_1 & b_2 & \dots & b_m \end{bmatrix} \tag{4.1.2}$$

The algorithms discussed in this document are suitable for large sparse linear systems which are expensive for ordinary elimination. Amongst the many methods for iteratively solving linear systems, algorithms like the Jacobi and over-relaxation methods not as effective as methods like conjugate gradient. The preconditioned conjugate gradient (CG) method is one of the most commonly used algorithms for solving large sparse systems for symmetric A. The textbook CG algorithm has been modified to be applicable even when A is not symmetric. The disadvantage of CG is that in each iteration, it must perform a matrix-vector product. To avoid this computation, some applications implement a new algorithm called the randomized Kaczmarz (RK) algorithm. It is a popular choice for extremely large scale applications. The algorithm is known to have a linear convergence rate and each iteration requires an $O(n)$ operation. In some applications, it outperforms CG. In general, it is be difficult to predict which one of CG or RK is preferable for a given linear system.

We now discuss three different approaches to solve linear systems. Direct method, Conjugate Gradient and Randomized Kaczmarz. Each method has its own advantages and disadvantages which will be highlighted.

4.2 Direct Methods

Direct methods are suitable for solving small linear systems that fit completely in memory. The workhorse of several commercial codes is the LU decomposition. The LU decomposition factors a matrix as the product of a lower triangular matrix (L) and an upper triangular matrix (U) such that

$$PA = LU$$

where P is a permutation matrix which re-orders the rows of A. Such an LU-decomposition can be used to solve (4.1.1) using

$$Ax = b \Rightarrow LUx = Pb$$

Now, we can solve for x using the following steps

i) Solve for y in the equation $Ly = Pb$

ii) Solve for x in the equation $Ux = b$

Since both L and U are triangular matrices, we can efficiently solve both the equations directly using forward and backward substitutions.

The main advantage of solving linear systems with direct methods is that direct methods are independent of the conditioning of the system. Solve time depends purely on the sparsity and size of the matrices. The major disadvantage is that the LU-decomposition has large memory requirements. Even when the matrix A is sparse, the L and U factors might be dense. The large memory requirements make it unsuitable for solving large or very sparse linear systems.

4.3 Iterative Methods

In solving (4.1.1) a convergent iterative method starts with an initial estimate x_0 of the solution and generates a sequence of iterates x_k that are successively closer to the solution x^*. Iterative methods are often useful even for linear problems involving a large number of variables. Amongst the many iterative methods, we will review the two most popular methods; the conjugate gradient method (CG) and the randomized Kaczmarz (RK) method.

4.3.1 Conjugate Gradient (CG)

The linear conjugate gradient method (not to be confused with the non-linear conjugate gradient method) to solve large sparse linear systems with a *symmetric positive definite* A matrix. Such a system can be stated as:

$$Ax = b \tag{4.3.1}$$

where $x \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times n}$ is symmetric and positive definite and $b \in \mathbb{R}^n$.

Unlike direct methods, the time taken to solve a linear system using CG depends on the distribution of the eigenvalues of the A matrix. In some applications, the A matrix is appropriately scaled by a process called pre-conditioning to generate an equivalent system with a more favorable distribution of eigenvalues.

The system (4.3.1) can be restated as the quadratic minimization

$$\min \phi(x) := \frac{1}{2} x^T A^T A x - b^T x$$

which allows us to interpret CG as an algorithm to minimize convex quadratic functions. In the rest of this section, we will refer to the gradient $\nabla\phi(x)$ as the residual of the linear system:

$$\nabla\phi(x) := r(x) := Ax - b$$

The linear conjugate gradient method generates a sequence of directions $p_0, p_1 \dots p_l$ that satisfy an important property called conjugacy which implies that the method can minimize the function $\phi(x)$ in exactly n steps. We refer the reader to the textbook by Nocedal and Wright [55] for details on the theoretical and algorithmic aspects of CG.

We now discuss an efficient implementation of the linear CG algorithm. In each, iteration (k), we keep track of the direction vector p_k, the residual vector r_k and the solution vector x_k. The computational bottleneck is in a matrix-vector multiplication between A and p.

Algorithm 4.3.1
Input: Symmetric matrix $A \in \mathbb{R}m \times n$, $b \in \mathbb{R}m$
Output: Solution to $Ax = b$

Choose $x_0 \in \mathbb{R}^n$, $r_0 \leftarrow Ax_0$, $p_0 \leftarrow -r_0$, $k \leftarrow 0$
while $\|r_k\|_2 \leq \epsilon$ **do**
    $z_k \leftarrow Ap_k$
    $\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T z_k}$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $r_{k+1} \leftarrow r_k + \alpha_k z_k$
    $\beta_k \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
    $p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k$
    $k = k + 1$

The conjugate gradient method is suitable for large sparse linear systems where direct methods can often run into memory bottlenecks. This is mainly because, the only memory requirements of the CG method is to store the latest copies of the vectors p_k, r_k and x_k. The majority of the computational efforts are spent in the step $z_k \leftarrow Ap_k$. Hence, CG tends to perform better in sparse linear systems.

Conjugate gradient Least Squares (CGLS)

In this section, we will extend the CG algorithm to be numerically suited to any linear system of the form (4.1.1). The naive extension of CG to (4.3.1) solves $A^T Ax = A^T b$. In addition to requiring an expensive matrix-matrix multiplication algorithm, it has it use of vectors of the form $A^T Ap$. An algorithm with better numerical properties was developed by Hestenes et. al [39].

Algorithm 4.3.2
Input: Matrix $A \in \mathbb{R}m \times n$, $b \in \mathbb{R}m$
Output: Solution to $Ax = b$

Choose $x_0 \in \mathbb{R}^n$, $r_0 \leftarrow b$, $s_0 \leftarrow A^T b$, $p_0 \leftarrow s_0$, $\gamma_0 = \|s_0\|_2^2$, $k \leftarrow 0$
while $\|r_k\|_2 \leq \epsilon$ **do**
    $z_k \leftarrow Ap_k$
    $\alpha_k \leftarrow \frac{\gamma_k}{z_k^T z_k}$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $r_{k+1} \leftarrow r_k - \alpha_k z_k$
    $s_{k+1} \leftarrow A^T r_{k+1}$
    $\gamma_{k+1} \leftarrow s_{k+1}^T s_{k+1}$
    $\beta_{k+1} \leftarrow \frac{\gamma_{k+1}}{\gamma_k}$
    $p_{k+1} \leftarrow s_{k+1} + \beta_{k+1} p_k$
    $k = k + 1$

Paige et. al [57] developed an algorithm called LSQR which has similar performance to CGLS. We might consider implementing LSQR in case CG performs poorly on linear systems.

4.3.2 Randomized Kaczmarz (RK)

As discussed earlier, the randomized Kaczmarz (RK) algorithm, is a popular algorithm for solving (4.1.1). Each iteration requires an $O(n)$ storage and computational effort. During each iteration,

RK picks a row a_i of the matrix A, and does an orthogonal projection of the current solution vector x_k to the hyperplane $a_i^T x = b$. The update step is given by

$$x_{k+1} = x_k - \frac{(a_i^T x_k - b_i)}{\|a_i\|_2} a_i$$

An alternate interpretation of RK is that the algorithm is identical to the stochastic gradient descent algorithm on the problem

$$\min \phi(x) := \frac{1}{2} x^T A^T A x - b^T x$$

The algorithm performs best when a row i is chosen randomly but proportional to $\|a_i\|_2$. Since sequential scans are preferred for in-database algorithms, a common pre-processing procedure for RK is to rescale the system so that each equation $a_i^T x = b$ has the same norm.

Algorithm 4.3.3
Input: Matrix $A \in \mathbb{R}m \times n$, $b \in \mathbb{R}m$
Output: Solution to $Ax = b$

1: Choose $x_0 \in \mathbb{R}^n$, $k \leftarrow 0$
2: **while** $\|Ax - b\|_2 \leq \epsilon$ **do**
3: $\quad x_{k+1} \leftarrow x_k - \frac{(a_i^T x_k - b_i)}{\|a_i\|_2}$
4: $\quad k = k + 1$

The termination criterion of the algorithm is implemented by computing the residual $\|Ax - b\|_2$ extremely infrequently. Typically, this computation is performed every K epochs where an epoch is defined as one whole pass of the data which in the case of RK is m iterations.

5 Singular Value Decomposition

Author Rahul Iyer

History **v0.1** Initial version

In linear algebra, the singular value decomposition (SVD) is a factorization of a real or complex matrix, with many useful applications in signal processing and statistics.

Let A be an $m \times n$ matrix, where $m \geq n$. Then A can be decomposed as follows:

$$A = U\Sigma V^T,$$

where U is a $m \times n$ orthonormal matrix, Σ is a $n \times n$ diagonal matrix, and V is an $n \times n$ orthonormal matrix. The diagonal elements of Σ are called the *singular values*.

It is possible to formulate the problem of computing the singular triplets (σ_i, u_i, v_i) of A as an eigenvalue problem involving a Hermitian matrix related to A. There are two possible ways of achieving this:

i) With the cross product matrix, A^TA and AA^T

ii) With the cyclic matrix $H(A) = \begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}$

The singular values are the nonnegative square roots of the eigenvalues of the cross product matrix. This approach may imply a severe loss of accuracy in the smallest singular values. The cyclic matrix approach is an alternative that avoids this problem, but at the expense of significantly increasing the cost of the computation.

Computing the cross product matrix explicitly is not recommended, especially in the case of sparse A. Bidiagonalization was proposed by Golub and Kahan [**golub1965**] as a way of tridiagonalizing the cross product matrix without forming it explicitly.

Consider the following decomposition

$$A = PBQ^T,$$

where P and Q are unitary matrices and B is an $m \times n$ upper bidiagonal matrix. Then the tridiagonal matrix BB^T is unitarily similar to AA^T. Additionally, specific methods exist that compute the singular values of B without forming BB^T. Therefore, after computing the SVD of B,

$$B = X\Sigma Y^T,$$

it only remains to compute the SVD of the original matrix with $U = PX$ and $V = QY$.

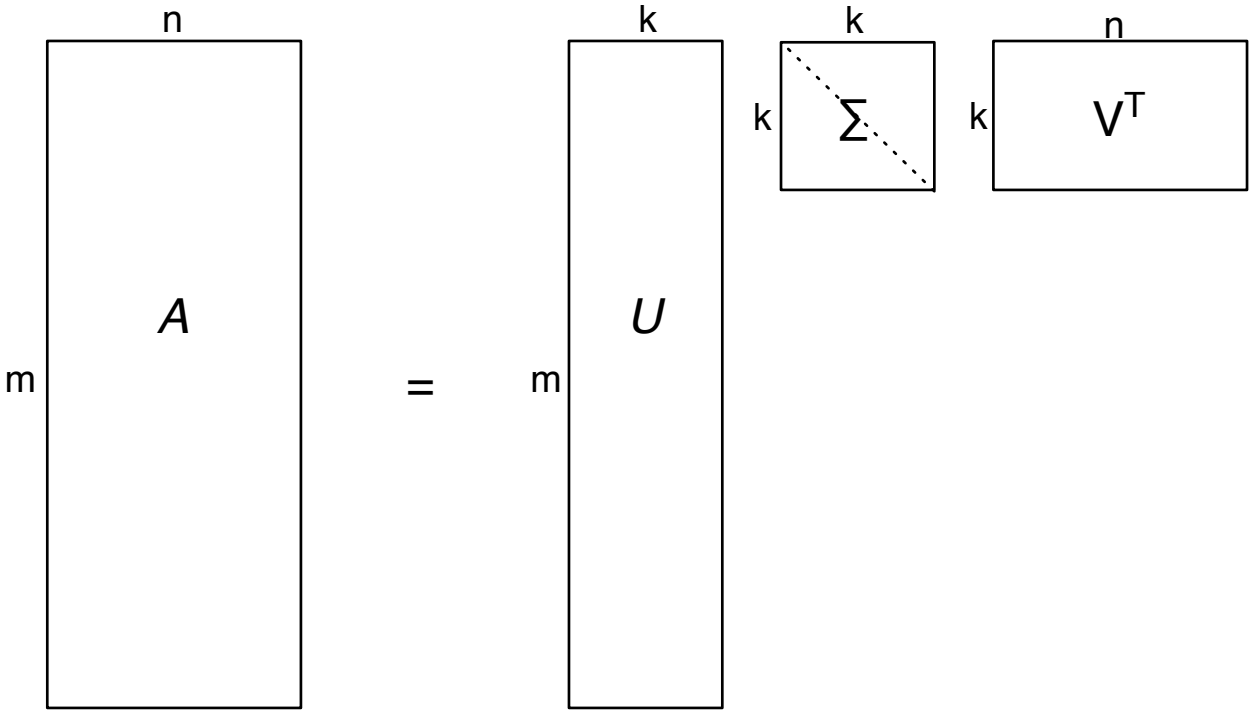

Figure 5.1: Singular Value Decomposition

5.1 Lanczos Bidiagonalization

The Lanczos algorithm is an iterative algorithm devised by Cornelius Lanczos that is an adaptation of power methods to find eigenvalues and eigenvectors of a square matrix or the singular value decomposition of a rectangular matrix. It is particularly useful for finding decompositions of very large sparse matrices.

For a rectangular matrix A, the Lanczos bidiagonalization method computes a sequence of Lanczos vectors $p_j \in \mathbb{R}^m$ and $q_j \in \mathbb{R}^n$ and scalars α_j and β_j for $j = 1, 2, \ldots, k$ as follows:

Algorithm `Lanczos Bidiagonalization Algorithm`

1: Choose a unit-norm vector q_1 and let $\beta_0 = 0$ and $p_0 = 0$
2: **for** $j = 1, 2, \ldots, k$ **do**
3: $\quad p_j \leftarrow Aq_j - \beta_{j-1}p_{j-1}$
4: $\quad \alpha_j \leftarrow ||p_j||_2$
5: $\quad p_j \leftarrow p_j/\alpha_j$
6: $\quad q_{j+1} \leftarrow A^T p_j - \alpha_j q_j$
7: $\quad \beta_j \leftarrow ||q_{j+1}||_2$
8: $\quad q_{j+1} \leftarrow q_{j+1}/\beta_j$

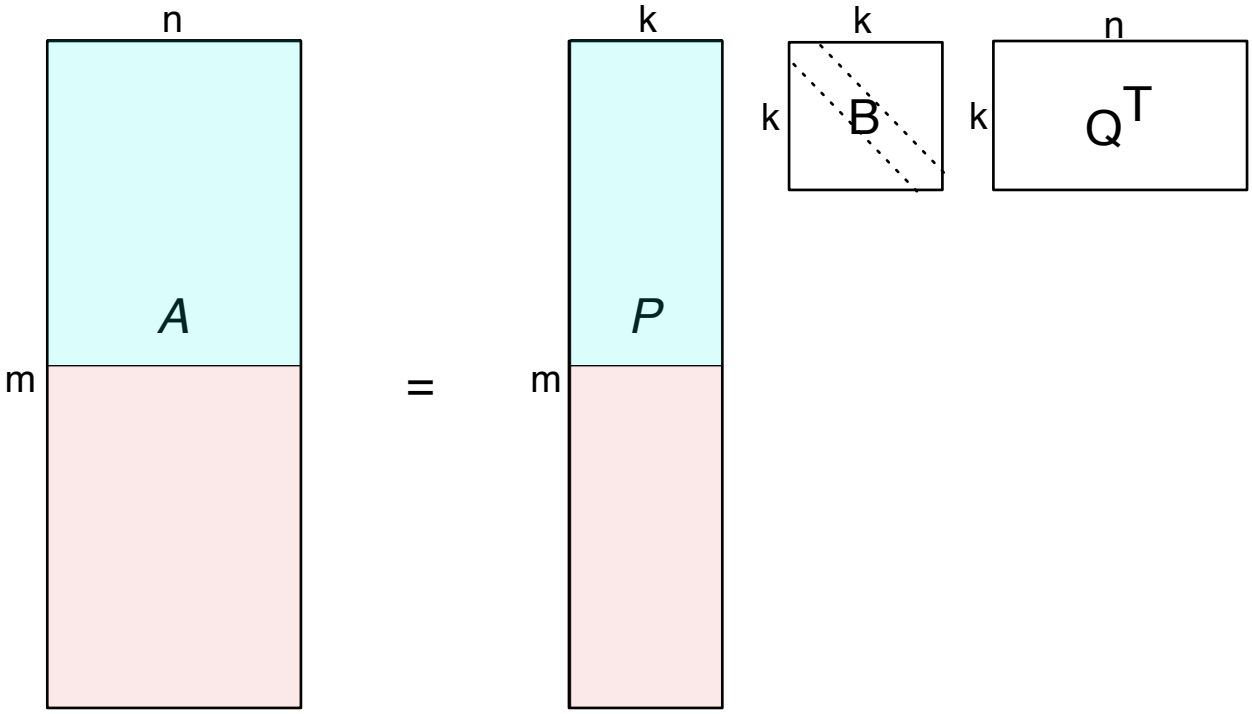

Figure 5.2: Lanczos bidiagonalization of A

After k steps, we can generate the bidiagonal matrix B_k as follows,

$$\begin{bmatrix} \alpha_1 & \beta_1 & & & & \\ & \alpha_2 & \beta_2 & & & \\ & & \alpha_3 & \beta_3 & & \\ & & & \ddots & \ddots & \\ & & & & \alpha_{k-1} & \beta_{k-1} \\ & & & & & \alpha_k \end{bmatrix}$$

In exact arithmetic the Lanczos vectors are orthonormal such that,

$$P_{k+1} = (p_1, p_2, \ldots, p_{k+1}) \in \mathbb{R}^{m\times(k+1)}, P_{k+1}^T P_{k+1} = I_{k+1}$$
$$Q_{k+1} = (q_1, q_2, \ldots, q_{k+1}) \in \mathbb{R}^{n\times(k+1)}, Q_{k+1}^T Q_{k+1} = I_{k+1}.$$

After k Lanczos steps, the Ritz values $\tilde{\sigma_i}$ (approximate singular values of A) are equal to the computed singular values of B_k, and the Ritz vectors are

$$\tilde{u_i} = P_k x_i, \qquad \tilde{v_i} = Q_k y_i$$

5.2 Dealing with Loss of Orthogonality

After a sufficient number of steps the Lanczos vectors start to lose their mutual orthogonality, and this happens together with the appearance in the spectrum of B_j of repeated and spurious Ritz values. The simplest cure for this loss of orthogonality is full orthogonalization. In Lanczos bidiagonalization, two sets of Lanczos vectors are computed, so full orthogonalization amounts to

orthogonalizing vector p_j explicitly with respect to all the previously computed left Lanczos vectors, and orthogonalizing vector q_{j+1} explicitly with respect to all the previously computed right Lanczos vectors.

Algorithm `Lanczos Bidiagonalization with Partial Reorthogonalization`

1: Choose a unit-norm vector q_1 and let $\beta_0 = 0$ and $p_0 = 0$
2: **for** $j = 1, 2, \ldots, k$ **do**
3: $\quad p_j \leftarrow Aq_j - \beta_{j-1}p_{j-1}$
4: $\quad \alpha_j \leftarrow ||p_j||_2$
5: $\quad p_j \leftarrow p_j/\alpha_j$
6: $\quad q_{j+1} \leftarrow A^T p_j - \alpha_j q_j$
7: $\quad$ Orthogonalize q_{j+1} with respect to Q_j
8: $\quad \beta_j \leftarrow ||q_{j+1}||_2$
9: $\quad q_{j+1} \leftarrow q_{j+1}/\beta_j$

There is a variation of this orthogonalization that maintains the effectiveness of full reorthogonalization but with a considerably reduced cost. This technique was proposed by Simon and Zha [66]. The idea comes from the observation that, in the Lanczos bidiagonalization procedure without reorthogonalization, the level of orthogonality of left and right Lanczos vectors go hand in hand. This observation led to Simon and Zha to propose what they called the one-sided version shown in Algorithm 2.

5.3 Enhancements for Distributed Efficiency

Algorithm `Distributed version of Lanczos BPRO`

1: Choose a unit-norm vector q_1 and let $\beta_0 = 0$ and $p_0 = 0$
2: **for** $j = 1, 2, \ldots, k$ **do**
Transition step
3: $\quad p_j \leftarrow Aq_j - \beta_{j-1}p_{j-1}$
4: $\quad \alpha_j \leftarrow ||p_j||_2^2$ ▷ Delayed normalization
5: $\quad q_{j+1} \leftarrow A^T p_j - \alpha_j q_j$
Merge step
6: $\quad$ Concatenate p_j across parallel segments
7: $\quad$ Sum q_{j+1} across parallel segments
Final Step
8: $\quad \alpha_j \leftarrow \sqrt{\alpha_j}$
9: $\quad p_j \leftarrow p_j/\alpha_j$
10: $\quad q_j \leftarrow q_j/\alpha_j$
11: $\quad$ Orthogonalize q_{j+1} with respect to Q_j
12: $\quad \beta_j \leftarrow ||q_{j+1}||_2$
13: $\quad q_{j+1} \leftarrow q_{j+1}/\beta_j$

6 Regression

Authors Rahul Iyer and Hai Qian

History
- **v0.3** Added section on Clustered Sandwich Estimators
- **v0.2** Added section on Marginal Effects
- **v0.1** Initial version, including background of regularization

Regression analysis is a statistical tool for the investigation of relationships between variables. Usually, the investigator seeks to ascertain the causal effect of one variable upon another - the effect of a price increase upon demand, for example, or the effect of changes in the money supply upon the inflation rate. More specifically, regression analysis helps one understand how the typical value of the dependent variable changes when any one of the independent variables is varied, while the other independent variables are held fixed.

Regression models involve the following variables:

i) The unknown parameters, denoted as β, which may represent a scalar or a vector.

ii) The independent variables, x

iii) The dependent variables, y

6.1 Multinomial Logistic Regression

Multinomial logistic regression is a widely used regression analysis tool that models the outcomes of categorical dependent random variables. *Generalized linear models* identify key ideas shared by a broad class of distributions thereby extending techniques used in linear regression, into the field of logistic regression.

This document provides an overview of the theory of multinomial logistic regression models followed by a design specification of how the model can be estimated using maximum likelihood estimators. In the final section, we outline a specific implementation of the algorithm that estimates multinomial logistic regression models on large datasets.

6.1.1 Problem Description

In this section, we setup the notation for a generic multinomial regression problem. Let us consider an N-dimensional multinomial random variable $\boldsymbol{Z}$ that can take values in J different categories, where $J \geq 2$. As input to the problem, we are given an $N \times J$ matrix of observations $\boldsymbol{y}$. Here $y_{i,j}$ denotes the observed value for the j^{th} category of the random variable Z_i. Analogous to the observed values, we define a set of parameters $\boldsymbol{\pi}$ as an $N \times J$ matrix with each entry $\pi_{i,j}$ denoting the probability of observing the random variable Z_i to fall in the j^{th} category. In logistic regression, we assume that the random variables $\boldsymbol{Z}$ are explained by a *design matrix* of independent random variables $\boldsymbol{X}$ which contains N rows and $(K+1)$ columns. We define a regression coefficient $\boldsymbol{\beta}$

as a matrix with $K+1$ rows and J columns such that $\beta_{k,j}$ corresponds to the importance while predicting the j^{th} value of the k^{th} explanatory variable.

For the multinomial regression model, we assume the observations $\boldsymbol{y}$ are realizations of random variables $\boldsymbol{Z}$ which are explained using random variables $\boldsymbol{X}$ and parameters $\boldsymbol{\beta}$. More specifically, if we consider the J^{th} category to be the 'pivot' or 'baseline' category, then the log of the odds of an observation compared to the J^{th} observation can be predicted using a linear function of variables $\boldsymbol{X}$ and parameters $\boldsymbol{\beta}$.

$$\log\Big(\frac{\pi_{i,j}}{\pi_{i,J}}\Big) = \log\Big(\frac{\pi_{i,j}}{\sum_{j=1}^{J-1}\pi_{i,j}}\Big) = \sum_{k=0}^{K} x_{i,k}\beta_{k,j} \tag{6.1.1}$$

Solving for $\pi_{i,j}$, we have

$$\pi_{i,j} = \frac{\exp\left(\sum_{k=0}^{K} x_{i,k}\beta_{k,j}\right)}{1+\sum_{j=1}^{J-1}\exp\left(\sum_{k=0}^{K} x_{i,k}\beta_{k,j}\right)} \qquad \forall j < J \tag{6.1.2a}$$

$$\pi_{i,J} = \frac{1}{1+\sum_{j=1}^{J-1}\exp\left(\sum_{k=0}^{K} x_{i,k}\beta_{k,j}\right)} \tag{6.1.2b}$$

In a sentence, the goal of multinomial regression is to use observations $\boldsymbol{y}$ to estimate parameters $\boldsymbol{\beta}$ that can predict random variables $\boldsymbol{Z}$ using explanatory variables $\boldsymbol{X}$.

6.1.2 Parameter Estimation

We evaluate the parameters $\boldsymbol{\beta}$ using a maximum-likelihood estimator (MLE) which maximizes the likelihood that a certain set of parameters predict the given set of observations. For this, we define the following likelihood function:

$$L(\boldsymbol{\beta}|\boldsymbol{y}) \simeq \prod_{i=1}^{N}\prod_{j=1}^{J} {\pi_{i,j}}^{y_{i,j}} \tag{6.1.3}$$

Substituting (6.1.2) in the above expression, we have

$$= \prod_{i=1}^{N}\prod_{j=1}^{J-1}\Big(1+\sum_{j=1}^{J-1} e^{\sum_{k=0}^{K} x_{i,k}\beta_{k,j}}\Big) e^{y_{i,j}\sum_{k=0}^{K} x_{i,k}\beta_{k,j}} \tag{6.1.4}$$

Taking the natural logarithm of $L(\boldsymbol{\beta}|\boldsymbol{y})$ defined above, we derive the following expression for the log-likelihood function, $l(\boldsymbol{\beta})$ as:

$$l(\boldsymbol{\beta}) = \sum_{i=1}^{N}\sum_{j=1}^{N}\Big(y_{i,j}\sum_{k=0}^{K} x_{i,k}\beta_{k,j}\Big) - \log\Big(1+\sum_{j=1}^{J-1}\sum_{k=0}^{K} x_{i,k}\beta_{k,j}\Big) \tag{6.1.5}$$

The maximum likelihood estimator tries maximize the log-likelihood function as defined in Equation (6.1.5). Unlike linear regression, the MLE has to be obtained numerically. Since we plan to implement derivative based algorithms to solve $\max_{\beta} l(\boldsymbol{\beta})$, we first derive expressions for the first and second derivatives of the log-likelihood function.

We differentiate (6.1.5) with respect to each parameter $\beta_{k,j}$

$$\frac{\partial l(\boldsymbol{\beta})}{\partial \beta_{k,j}} = \sum_{i=1}^{N} y_{i,j} x_{i,k} - \pi_{i,j} x_{i,k} \qquad \forall k \; \forall j \tag{6.1.6}$$

We evaluate the extreme point of the function $l(\boldsymbol{\beta})$ by setting each equation of (6.1.6) to zero. We proceed on similar lines to derive the second order derivative of the $l(\boldsymbol{\beta})$ with respect to two parameters β_{k_1,j_1} and β_{k_2,j_2}

$$\frac{\partial^2 l(\boldsymbol{\beta})}{\partial \beta_{k_2,j_2} \partial \beta_{k_1,j_1}} = \sum_{i=1}^{N} -\pi_{i,j_2} x_{i,k_2} (1 - \pi_{i,j_1}) x_{i,k_1} \qquad j_1 = j_2 \tag{6.1.7a}$$

$$= \sum_{i=1}^{N} \pi_{i,j_2} x_{i,k_2} \pi_{i,j_1} x_{i,k_1} \qquad j_1 \neq j_2 \tag{6.1.7b}$$

6.1.3 Algorithms

Newton's method is a numerical recipe for root finding of non-linear functions. We apply this method to solve all nonlinear equations produced by setting (6.1.6) to zero. Newton's method begins with an initial guess for the solution after which it uses the Taylor series approximation of function at the current iterate to produce another estimate that might be closer to the true solution. This iterative procedure has one of the fastest theoretical rates of convergence in terms of number of iterations, but requires second derivative information to be computed at each iteration which is a lot more work than other light-weight derivative based methods.

Newton method can be compactly described using the update step. Let us assume $\boldsymbol{\beta^0}$ to be the initial guess for the MLE (denoted by MLE). If the $\boldsymbol{\beta^I}$ is the 'guess' for MLE at the I^{th} iteration, then we can evaluate $\boldsymbol{\beta^{I+1}}$ using

$$\boldsymbol{\beta^{I+1}} = \boldsymbol{\beta^I} - [l''(\boldsymbol{\beta^I})]^{-1} \; l'(\boldsymbol{\beta^I}) \tag{6.1.8}$$

$$= \boldsymbol{\beta^I} - [l''(\boldsymbol{\beta^I})]^{-1} \boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{\pi}) \tag{6.1.9}$$

where $\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{\pi})$ is matrix notation for the first order derivative. The newton method might have proven advantage in terms of number of iterations on small problem sizes but it might not scale well to larger sizes because it requires an expensive step for the matrix inversion of the second order derivative.

As an aside, we observe that Equation (6.1.6) and (6.1.7a) refers to the first derivative in matrix form and the second derivative in tensor form respectively. In the implementation phase, we work with 'vectorized' versions of $\boldsymbol{\beta}, \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{\pi}$ denoted by $\boldsymbol{\beta}, \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{\pi}$ respectively where the matrix are stacked up together in row major format.

Using this notation, we can rewrite the first derivative in (6.1.6) as:

$$\frac{\partial l(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{\pi}) \tag{6.1.10}$$

Similarly, we can rewrite the second derivative in (6.1.7a) as:

$$\frac{\partial^2 l(\boldsymbol{\beta})}{\partial^2 \boldsymbol{\beta}} = \boldsymbol{X}^T W \boldsymbol{X} \tag{6.1.11}$$

where W is a diagonal matrix of dimension $(K+1) \times J$ where the diagonal elements are set $\pi_{i,j_1}\pi_{i,j_2}$ if $j_1 \neq j_2$ or $\pi_{i,j_1}(1-\pi_{i,j_2})$ otherwise. Note that (6.1.11) is merely a compact way to write (6.1.7a).

The Newton method procedure is illustrated in Algorithm 6.1.1.

Algorithm 6.1.1
Input: $\boldsymbol{X}, \boldsymbol{Y}$ and an initial guess for parameters $\boldsymbol{\beta^0}$
Output: The maximum likelihood estimator β_{MLE}

1: $I \leftarrow 0$
2: **repeat**
3: Diagonal Weight matrix W: $w_{j_1,j_2} \leftarrow \pi_{i,j_1}\pi_{i,j_2}$ if $j_1 \neq j_2$ or $\pi_{i,j_1}(1-\pi_{i,j_2})$ otherwise
4: Compute $\boldsymbol{\beta}^{I+1}$ using:
5: $\boldsymbol{\beta}^{I+1} = \boldsymbol{\beta}^I - (\boldsymbol{X}^T W \boldsymbol{X})^{-1}\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{\pi})$
6: **until** $\boldsymbol{\beta}^{I+1}$ converges

6.1.4 Common Statistics

Irrespective of the solver that we choose to implement, we would need to calculate the standard errors and p-value.

Asymptotics tells that the MLE is an asymptotically efficient estimator. This means that it reaches the following Cramer-Rao lower bound:

$$\sqrt{n}(\beta_{MLE} - \beta) \rightarrow \mathcal{N}(0, I^{-1}) \tag{6.1.12}$$

where I^{-1} is the Fisher information matrix. Hence, we evaluate the standard errors using the asymptotic variance of i^{th} parameter (in vector form) of the MLE as:

$$se(\boldsymbol{\beta}_i) = (\boldsymbol{X}^T W \boldsymbol{X})_i^{-1} \tag{6.1.13}$$

The Wald statistic is used to assess the significance of coefficients β. In the Wald test, the MLE is compared with the null hypothesis while assuming that the difference between them is approximately normally distributed. The Wald p-value for a coefficient provides us with the probability of seeing a value as extreme as the one observed, when null hypothesis is true. We evaluate the Wald p-value as:

$$p_i = \Pr\Big(|Z| \geq |\frac{\boldsymbol{\beta}_i}{se(\boldsymbol{\beta}_i)}|\Big) = 2\Big[1 - F(se(\boldsymbol{\beta}_i))\Big] \tag{6.1.14}$$

where Z is a normally distributed random variable and F represents the cdf of Z.

6.2 Implementation

For the implementation, we plan to mimic the framework used for the current implementation of the logistic regression. In this framework, the regression is broken up into 3 steps, denoted as the *transition* step, the *merge states* step, and the *final* step. Much of the computation is done in parallel, and each transition step operates on a small number of rows in the data matrix to compute a portion of the calculation for $\boldsymbol{X}^T W \boldsymbol{X}$, as well as the calculation for $\boldsymbol{X}^T W \boldsymbol{\alpha}$. This is used in the Hessian calculation in equation 6.1.11.

The merge step aggregates these transition steps together. This consists of summing the $X^T W X$ calculations, summing the $\boldsymbol{X}^T W \boldsymbol{\alpha}$ calculations, and updating the bookkeeping. The final step computes the current solution $\boldsymbol{\beta}$, and returns it and its associated statistics.

The three steps in the framework communicate by passing around *state* objects. Each state object carries with it several fields:

coef This is the column vector containing the current solution. This is the β part of $\beta^T X$ with the exception of β_0.

beta The scalar constant term β_0 in $\beta^T X$.

widthOfX The number of features in the data matrix.

numRows The number of data points being operated on.

dir The direction of the gradient it the kth step. Not sure about this one.

grad The gradient vector in the kth step.

gradNew The gradient vector in the $x + 1$th step.

X_transp_AX The current sum of $\boldsymbol{X}^T W \boldsymbol{X}$.

X_transp_Az The current sum of $\boldsymbol{X}^T W \boldsymbol{\alpha}$.

logLikelihood The log likelihood of the solution.

Each transition step is given an initial state, and returns a state with the updated X_transp_AX and X_transp_Az fields. These states are iteratively passed to a merge step, which combines them two at a time. The final product is then passed to the final step. We expect to use the same system, or something similar.

We can formalize the API's for these three steps as:

```
multilogregr_irls_step_transition(AnyType *Args)
multilogregr_irls_step_merge(AnyType *Args)
multilogregr_irls_step_final(AnyType *Args)
```

The `AnyType` object is a generic type used in madlib to retrieve and return values from the backend. Among other things, an `AnyType` object can be a `NULL` value, a scalar, or a state object.

The first step, `multilogregr_cg_step_transition`, will expect `*Args` to contains the following items in the following order: a state object for the current state, a vector x containing a row of the design matrix, a vector y containing the values of $\boldsymbol{Z}$ for this row, and a state object for the previous state. The return value for this function will be another state object.

The second step `multilogregr_cg_step_merge` will expect `*Args` to contain two state objects, and will return a state object expressing the merging of the two input objects. The final step `multilogregr_cg_step_final` expects a single state object, and returns an `AnyType` object containing the solution's coefficients, the standard error, and the solution's p values.

6.3 Regularization

Usually, y is the result of measurements contaminated by small errors (noise). Frequently, ill-conditioned or singular systems also arise in the iterative solution of nonlinear systems or optimization problems. In all such situations, the vector $x = A^{-1}y$ (or in the full rank overdetermined case $A^+ y$, with the pseudo inverse $A^+ = (A^T A)^{-1} A^T X$), if it exists at all, is usually a meaningless bad approximation to x.

Regularization techniques are needed to obtain meaningful solution estimates for such ill-posed problems, and in particular when the number of parameters is larger than the number of available measurements, so that standard least squares techniques break down.

6.3.1 Linear Ridge Regression

Ridge regression is the most commonly used method of regularization of ill-posed problems. Mathematically, it seeks to minimize

$$Q\left(\boldsymbol{w}, w_0; \lambda\right) \equiv \min_{\boldsymbol{w}, w_0} \left[\frac{1}{2N} \sum_{i=1}^{N} \left(y_i - w_0 - \boldsymbol{w} \cdot \boldsymbol{x}_i\right)^2 + \frac{\lambda}{2} \|\boldsymbol{w}\|_2^2 \right] , \tag{6.3.1}$$

for a given value of λ, where $\boldsymbol{w}$ and w_0 are the fitting coefficients, and λ is a non-negative regularization parameter. $\boldsymbol{w}$ is a vector in d dimensional space, and

$$\|\boldsymbol{w}\|_2^2 = \sum_{j=1}^{d} w_j^2 = \boldsymbol{w}^T \boldsymbol{w} \ . \tag{6.3.2}$$

When $\lambda = 0$, Q is the mean squared error of the fitting.

The intercept term w_0 is not regularized, because this term is fully decided by the mean values of y_i and $\boldsymbol{x}_i$ and the values of $\boldsymbol{w}$, and does not affect the model's complexity.

$Q\left(\boldsymbol{w}, w_0; \lambda\right)$ is a quadratic function of $\boldsymbol{w}$ and w_0, and thus can be solved analytically

$$\boldsymbol{w}_{ridge} = \left(\lambda \boldsymbol{I}_d + \boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y} \ . \tag{6.3.3}$$

By using the available Newton method (Sec. 6.2.4), the above quantity can be easily calculated from one single step of the Newton method.

Many packages for Ridge regularization actually regularize the fitting coefficients not for the fitting model for the original data but for the data that has be scaled. MADlib also provides this option. When the normalization parameter is set to be True, which is by default False, the data will be first converted to the following before applying the Ridge regularization.

$$x_i' \leftarrow \frac{x_i - \langle x_i \rangle}{\langle (x_i - \langle x_i \rangle)^2 \rangle} \ , \tag{6.3.4}$$

$$y_i \leftarrow y_i - \langle y_i \rangle \ , \tag{6.3.5}$$

where $\langle \cdot \rangle = \sum_{i=1}^{N} \cdot / N$.

Note that Ridge regressions for scaled data and un-scaled data are not equivalent.

6.3.2 Elastic Net Regularization

As a continuous shrinkage method, ridge regression achieves its better prediction performance through a bias-variance trade-off. However, ridge regression cannot produce a parsimonious model, for it always keeps all the predictors in the model [79]. Best subset selection in contrast produces a sparse model, but it is extremely variable because of its inherent discreteness.

A promising technique called the lasso was proposed by Tibshirani (1996). The lasso is a penalized least squares method imposing an L1-penalty on the regression coefficients. Owing to the nature of the L1-penalty, the lasso does both continuous shrinkage and automatic variable selection simultaneously.

Although the lasso has shown success in many situations, it has some limitations. Consider the following three scenarios:

i) In the 'Number of features' (p) » 'Number of observations' (n) case, the lasso selects at most n variables before it saturates, because of the nature of the convex optimization problem. This seems to be a limiting feature for a variable selection method. Moreover, the lasso is not well defined unless the bound on the L_1-norm of the coefficients is smaller than a certain value.

ii) If there is a group of variables among which the pairwise correlations are very high, then the lasso tends to select only one variable from the group and does not care which one is selected.

iii) For usual n>p situations, if there are high correlations between predictors, it has been empirically observed that the prediction performance of the lasso is dominated by ridge regression.

These scenarios make lasso an inappropriate variable selection method in some situations.

Hui Zou and Trevor Hastie [42] introduce a new regularization technique called the 'elastic net'. Similar to the lasso, the elastic net simultaneously does automatic variable selection and continuous shrinkage, and it can select groups of correlated variables. It is like a stretchable fishing net that retains 'all the big fish'.

The elastic net regularization minimizes the following target function

$$\min_{\boldsymbol{w}\in R^N} L(\boldsymbol{w}) + \lambda \left[\frac{1-\alpha}{2} \|\boldsymbol{w}\|_2^2 + \lambda\alpha \|\boldsymbol{w}\|_1 \right] \ , \tag{6.3.6}$$

where $\|\boldsymbol{w}\|_1 = \sum_{i=1}^{N} |w_i|$ and N is the number of features.

For the elastic net regularization on linear models,

$$L(\boldsymbol{w}) = \frac{1}{2M} \sum_{m=1}^{M} \left(y_m - w_0 - \boldsymbol{w} \cdot \boldsymbol{x}_m \right)^2 \ , \tag{6.3.7}$$

where the sum is over all observations and M is the total number of observation.

For the elastic net regularization on logistic models,

$$L(\boldsymbol{w}) = \sum_{m=1}^{M} \left[y_m \log\left(1 + e^{-(w_0 + \boldsymbol{w}\cdot\boldsymbol{x}_m)}\right) + (1-y_m) \log\left(1 + e^{w_0 + \boldsymbol{w}\cdot\boldsymbol{x}_m}\right) \right] \ , \tag{6.3.8}$$

where $y_m \in \{0, 1\}$.

6.3.2.1 Optimizer Algorithms

Right now, we support two algorithms for optimizer. The default one is FISTA, and the other is IGD.

FISTA Fast Iterative Shrinkage Thresholding Algorithm (FISTA) with **backtracking step size** [4]:

Step 0: Choose $\delta > 0$ and $\eta > 1$, and $\boldsymbol{w}^{(0)} \in R^N$. Set $\boldsymbol{v}^{(1)} = \boldsymbol{w}^{(0)}$ and $t_1 = 1$.

Step k: ($k \geq 1$) Find the smallest nonnegative integers i_k such that with $\bar{\delta} = \delta_{k-1}/\eta^{i_k - 1}$

$$F(p_{\bar{\delta}}(\boldsymbol{v}^{(k)})) \leq Q_{\bar{\delta}}(p_{\bar{\delta}}(\boldsymbol{v}^{(k)}), \boldsymbol{v}^k) \ . \tag{6.3.9}$$

Set $\delta_k = \delta_{k-1}/\eta^{i_k-1}$ and compute

$$\boldsymbol{w}^{(k)} = p_{\delta_k}\left(\boldsymbol{v}^{(k)}\right) \ , \tag{6.3.10}$$

$$t_{k+1} = \frac{1+\sqrt{1+4t_k^2}}{2} \ , \tag{6.3.11}$$

$$\boldsymbol{v}^{(k+1)} = \boldsymbol{w}^{(k)} + \frac{t_k - 1}{t_{k+1}}\left(\boldsymbol{w}^{(k)} - \boldsymbol{w}^{(k-1)}\right) \ . \tag{6.3.12}$$

Here,

$$F(\boldsymbol{w}) = f(\boldsymbol{w}) + g(\boldsymbol{w}) \ , \tag{6.3.13}$$

where $f(\boldsymbol{w})$ is the differentiable part of Eq. (6.3.6) and $g(\boldsymbol{w})$ is the non-differentiable part. For linear models,

$$f(\boldsymbol{w}) = \frac{1}{2M}\sum_{m=1}^{M}\left(y_m - w_0 - \boldsymbol{w}\cdot\boldsymbol{x}_m\right)^2 + \frac{\lambda(1-\alpha)}{2}\|\boldsymbol{w}\|_2^2 \ , \tag{6.3.14}$$

and for logistic models,

$$f(\boldsymbol{w}) = \sum_{m=1}^{M}\left[y_m \log\left(1+e^{-(w_0+\boldsymbol{w}\cdot\boldsymbol{x}_m)}\right) + (1-y_m)\log\left(1+e^{w_0+\boldsymbol{w}\cdot\boldsymbol{x}_m}\right)\right] + \frac{\lambda(1-\alpha)}{2}\|\boldsymbol{w}\|_2^2 \ . \tag{6.3.15}$$

And for both types of models,

$$g(\boldsymbol{w}) = \lambda\alpha\sum_{i=1}^{N}|w_i| \ . \tag{6.3.16}$$

And

$$Q_\delta(\boldsymbol{a},\boldsymbol{b}) := f(\boldsymbol{b}) + \langle \boldsymbol{a}-\boldsymbol{b}, \nabla f(\boldsymbol{b})\rangle + \frac{1}{2\delta}\|\boldsymbol{a}-\boldsymbol{b}\|^2 + g(\boldsymbol{a}) \ , \tag{6.3.17}$$

where $\langle\cdot\rangle$ is just the usual vector dot product.

And the proxy function is defined as

$$p_\delta(\boldsymbol{v}) := \arg\min_{\boldsymbol{w}}\left[g(\boldsymbol{w}) + \frac{1}{2\delta}\left\|\boldsymbol{w} - (\boldsymbol{v} - \delta\,\nabla f(\boldsymbol{v}))\right\|^2\right] \tag{6.3.18}$$

For our case, where $g(\boldsymbol{w}) = \lambda\alpha\sum_{i=1}^{N}|w_i|$, the proxy function is simply equal to the soft-thresholding function

$$p_\delta(v_i) = \left\{\begin{array}{ll} v_i - \lambda\alpha\delta u_i \ , & \text{if } v_i > \lambda\alpha\delta u_i \\ 0 \ , & \text{otherwise} \\ v_i + \lambda\alpha\delta u_i \ , & \text{if } v_i < -\lambda\alpha\delta u_i \end{array}\right\} \tag{6.3.19}$$

where

$$\boldsymbol{u} = \boldsymbol{v} - \delta\,\nabla f(\boldsymbol{v}) \ . \tag{6.3.20}$$

Active set method is used in our implementation for FISTA to speed up the computation. Considerable speedup is obtained by organizing the iterations around the active set of features - those with nonzero coefficients. After a complete cycle through all the variables, we iterate on only the active set till convergence. If another complete cycle does not change the active set, we are done, otherwise the process is repeated.

Warm-up method is also used to speed up the computation. When the option parameter *warmup* is set to be *True*, a series of lambda values, which is strictly descent and ends at the lambda value that the user wants to calculate, will be used. The larger lambda gives very sparse solution, and the sparse solution again is used as the initial guess for the next lambda's solution, which will speed up the computation for the next lambda. For larger data sets, this can sometimes accelerate the whole computation and might be faster than computation on only one lambda value.

Note: Our implementation is a little bit different from the original FISTA. In the original FISTA, during the backtracking procedure, the algorithm is searching for a non-negative integer i_k and the new step size is $\delta_k = \delta_{k-1}/\eta^{i_k}$. Thus the step size is non-increasing. Here, we allow the step size to increase by using $\delta_k = \delta_{k-1}/\eta^{i_k-1}$ so that larger step sizes can be tried by the algorithm. Tests show that this is actually faster.

IGD The Incremental Gradient Descent (IGD) algorithm is a stochastic algorithm by its nature. So it is difficult to get sparse solutions. What we implemented is Stochastic Mirror Descent Algorithm made Sparse (SMIDAS). The inverse p-form link function is used

$$h_j^{-1}(\boldsymbol{\theta}) = \frac{\text{sign}(\theta_j)|\theta_j|^{p-1}}{\|\theta\|_p^{p-2}} \ , \tag{6.3.21}$$

where

$$\|\theta\|_p = \left(\sum_j |\theta|^p\right)^{1/p} \ , \tag{6.3.22}$$

and $\text{sign}(0) = 0$.

Choose step size $\delta > 0$.
Let $p = 2\log d$ and let h^{-1} be as in Eq. (6.3.21)
Let $\boldsymbol{\theta} = \mathbf{0}$
for $k = 1, 2, \ldots$
  $\boldsymbol{w} = h^{-1}(\boldsymbol{\theta})$
  $\boldsymbol{v} = \nabla f(\boldsymbol{w})$
  $\tilde{\boldsymbol{\theta}} = \theta - \delta\, \boldsymbol{v}$
  $\forall j, \theta_j = \text{sign}(\tilde{\theta}_j)\max(0, |\tilde{\theta}_j| - \lambda\alpha\delta)$

The resulting fitting coefficients of this algorithm is not really sparse, but the values are very small (usually $< 10^{15}$), which can be safely set to be zero after filtering with a threshold.

This is done as the following: (1) multiply each coefficient with the standard deviation of the corresponding feature (2) compute the average of absolute values of re-scaled coefficients (3) divide each rescaled coefficients with the average, and if the resulting absolute value is smaller than *threshold*, set the original coefficient to be zero.

IGD is in nature a sequential algorithm, and when running in a distributed way, each segment of the data runs its own SGD model, and the models are averaged to get a model for each iteration. This average might slow down the convergence speed, although we acquire the ability to process large data set on multiple machines. So this algorithm provides an option *parallel* to let the user choose whether to do parallel computation.

IGD also implements the **warm-up method**.

Stopping Criteria Both **FISTA** and **IGD** compute the average difference between the coefficients of two consecutive iterations, and if the difference is smaller than *tolerance* or the iteration number is larger than *max_iter*, the computation stops.

The resulting fitting coefficients of this algorithm is not really sparse, but the values are very small (usually $< 10^{15}$), which can be safely set to be zero after filtering with a threshold.

This is done as the following: (1) multiply each coefficient with the standard deviation of the corresponding feature (2) compute the average of absolute values of re-scaled coefficients (3) divide each rescaled coefficients with the average, and if the resulting absolute value is smaller than threshold, set the original coefficient to be zero.

IGD is in nature a sequential algorithm, and when running in a distributed way, each segment of the data runs its own SGD model, and the models are averaged to get a model for each iteration. This average might slow down the convergence speed, although we acquire the ability to process large data set on multiple machines. So this algorithm provides an option parallel to let the user choose whether to do parallel computation.

IGD also implements the **warm-up method**.

Stopping Criteria Both **FISTA** and **IGD** compute the average difference between the coefficients of two consecutive iterations, and if the difference is smaller than tolerance or the iteration number is larger than max_iter, the computation stops.

We note that the Hessian and derivative both depend on β, so the variance estimates will also depend on β. Rather than allow the user to specify a β value, the implementation computes the optimal β by running the appropriate regression before computing the robust variance. In cases where the regression has parameters (regression tolerance, max iterations), the interface allows the user to specify those parameters.

6.4 Robust Variance via Huber-White Sandwich Estimators

Given N data points, where the ith point is defined by a feature vector $x_i \in \mathbb{R}^M$ and a category scalar y_i, where $y_i \in \mathbb{R}, y_i \in \{0, 1\}$, and $y_i \in \{1, \ldots, L\}$ for linear, logistic, and multi-logistic regression respectively. We assume that y_i is drawn from an independent and identically distributed (i.i.d.) distribution determined by a K-dimensional parameter vector β (which is $K \times L$ dimensional for multi-logistic regression).

Generally, we are interested in finding the values of β that best predict y_i from x_i, with *best* being defined as the values that maximize some log-likelihood function $l(y, x, \beta)$. The maximization is typically solved using the derivative of the likelihood ψ and the Hessian H. More formally, ψ is defined as

$$\psi(y, x, \beta) = \frac{\partial l(x, y, \beta)}{\partial \beta} \tag{6.4.1}$$

and H is defined as

$$H(y, x, \beta) = \frac{\partial^2 l(x, y, \beta)}{\partial \beta^2}. \tag{6.4.2}$$

Using these derivatives, one can solve the logistic or linear regression for the optimal solution, and compute the variance and other statistics of the regression.

6.4.1 Sandwich Operators

However, we may believe that the underlying distribution is not i.i.d., (in particular, the variance is not independent of x_i), in which case, our variance estimates will be incorrect. The Huber sandwich

estimator is used to get a robust estimate of the variance even if the i.i.d. assumption is wrong. The estimator is known as a sandwich estimator because the robust covariance matrix $S(\beta)$ of β can be expressed in a *sandwich formulation*, of the form

$$S(\beta) = B(\beta)M(\beta)B(\beta). \tag{6.4.3}$$

The $B(\beta)$ matrix is commonly called the *bread*, whereas the $M(\beta)$ matrix is the *meat*.

The Bread Computing B is relatively straightforward,

$$B(\beta) = N\left(\sum_{i}^{N} -H(y_i, x_i, \beta)\right)^{-1} \tag{6.4.4}$$

The Meat There are several choices for the M matrix, each with different robustness properties. In the Huber-White estimator, the matrix M is defined as

$$M_H = \frac{1}{N}\sum_{i}^{N} \psi(y_i, x_i, \beta)^T \psi(y_i, x_i, \beta). \tag{6.4.5}$$

6.4.2 Implementation

The Huber-White sandwich estimators implemented for linear, logistic, and multinomial-logistic regression mimic the same framework as the linear/logistic regression implementations. In these implementations, the gradient and Hessian are computed in parallel, and a final step operates on the aggregate result.

This framework breaks the computation into three steps: *transition* step, the *merge states* step, and the *final* step. The transition step computes the gradient and Hessian contribution from each row in the data matrix. To compute this step, we need to define the derivatives for each regression.

6.4.2.1 Linear Regression Derivatives

For linear regression, the derivative is

$$\frac{\partial l(x, y, \beta)}{\partial \beta} = X^T(y - X\beta) \tag{6.4.6}$$

and the Hessian is

$$\frac{\partial^2 l(x, y, \beta)}{\partial^2 \beta} = -X^T X. \tag{6.4.7}$$

6.4.2.2 Logistic Regression Derivatives

For logistic, the derivative is

$$\frac{\partial l(x, y, \beta)}{\partial \beta} = \sum_{i=1}^{N} \frac{-1^{y_i} \cdot \beta}{1 + \exp(-1^{y_i} \cdot \beta^T x)} \exp(-1^{y_i} \cdot \beta^T x) \tag{6.4.8}$$

and the Hessian is

$$\frac{\partial^2 l(x, y, \beta)}{\partial^2 \beta} = -X^T A X = -\sum_{i=1}^{N} A_{ii} x_i^T x_i \tag{6.4.9}$$

where A is a diagonal matrix with

$$A_{ii} = \left([1 + \exp(c^T x_i))(1 + \exp(-c^T x_i)]\right)^{-1}. \tag{6.4.10}$$

6.4.2.3 Multi-Logistic Regression Derivatives

For multi-logistic regression, we replace y_i with a vector $Y_i \in \{0, 1\}^L$, where all entries of Y_i are zero expect the y_ith entry, which is set to 1. In addition, we choose a *baseline* category, for which the odds of all the categories are measured against. Let $J \in \{1, \ldots, L\}$ be the baseline category.

We define the variables

$$\pi_{i,j} = \frac{\exp\left(\sum_{k=1}^{K} x_{i,k}\beta_{k,j}\right)}{1 + \sum_{j \neq J} \exp\left(\sum_{k=1}^{K} x_{i,k}\beta_{k,j}\right)}, \quad \forall j \neq J \tag{6.4.11}$$

$$\pi_{i,J} = \frac{1}{1 + \sum_{j \neq J} \exp\left(\sum_{k=1}^{K} x_{i,k}\beta_{k,j}\right)}. \tag{6.4.12}$$

The derivatives are then

$$\frac{\partial l}{\partial \beta_{k,j}} = \sum_{i=1}^{N} Y_{i,j} x_{i,k} - \pi_{i,j} x_{i,k} \quad \forall k \; \forall j \tag{6.4.13}$$

The Hessian is then

$$\frac{\partial^2 l(\beta)}{\partial \beta_{k_2,j_2} \partial \beta_{k_1,j_1}} = \sum_{i=1}^{N} -\pi_{i,j_2} x_{i,k_2} (1 - \pi_{i,j_1}) x_{i,k_1} \qquad j_1 = j_2 \tag{6.4.14}$$

$$= \sum_{i=1}^{N} \pi_{i,j_2} x_{i,k_2} \pi_{i,j_1} x_{i,k_1} \qquad j_1 \neq j_2 \tag{6.4.15}$$

6.4.2.4 Merge Step and Final Step

For the logistic and multi-logistic, the derivative and Hessian are sums in which the terms can be computed in parallel, and thus in a distributed manner. Each transition step computes a single term in the sum. The merge step sums two or more terms computed by the transition steps, and the final step computes the Hessian inverse, and the matrix product between the bread and meat matrices.

6.5 Marginal Effects

Most of the notes below are based on [25]

A *marginal effect* (ME) or partial effect measures the effect on the conditional mean of y of a change in one of the regressors, say x_k [18]. In the linear regression model (without any interaction terms), the ME equals the relevant slope coefficient, greatly simplifying analysis. For nonlinear models, this is no longer the case, leading to remarkably many different methods for calculating MEs.

Let $E(y_i|x_i)$ represent the expected value of a dependent variable y_i given a vector of independent variables x_i. In the case where y is a linear function of $(x_1, \ldots, x_M) = \boldsymbol{x}$ and y is a continuous

variable, a linear regression model (without any interaction effects) can be stated as follows:

$$y = \boldsymbol{x}^T\boldsymbol{\beta}$$
or
$$y = \beta_1 x_1 + \ldots + \beta_M x_M.$$

From the above equation it is straightforward to see that the marginal effect of any variable x_k on the dependent variable is $\partial y/\partial x_k = \beta_k$. However, this is true only if there exists no interaction between the variables. If the output expression contains interaction terms, the model would be

$$y = \beta_1 f_1 + \ldots + \beta_N f_N.$$

where f_i is a function of the base variables $x_1, x_2, \ldots, x_M$ and describes the interaction between the base variables. In the simple (non-interaction) case, $f_i = x_i$ and $M = N$.

The standard approach to modeling dichotomous/binary variables (so $y \in 0, 1$) is to estimate a generalized linear model under the assumption that y follows some form of Bernoulli distribution. Thus the expected value of y becomes,

$$y = G(\boldsymbol{x}^T\boldsymbol{\beta}),$$

where G is the specified binomial distribution. Here we assume to use logistic regression and use g to refer to the inverse logit function. The same approach is applied when y is a discrete, multi-valued variable.

6.5.1 Discrete change effect

Along with marginal effects we can also compute the following discrete change effects.

i) Unit change effect, which should not be confused with the marginal effect:

$$\begin{aligned}\frac{\partial y}{\partial x_k} &= P(y = 1|X, x_k + 1) - P(y = 1|X, x_k) \\ &= g(\beta_1 x_1 + \cdots + \beta_k(x_k + 1) + \beta_l x_l) \\ &\quad - g(\beta_1 x_1 + \cdots + \beta_k x_k + \beta_l x_l)\end{aligned}$$

ii) Centered unit change:

$$\frac{\partial y}{\partial x_k} = P(y = 1|X, x_k + 0.5) - P(y = 1|X, x_k - 0.5)$$

iii) Standard deviation change:

$$\frac{\partial y}{\partial x_k} = P(y = 1|X, x_k + 0.5\delta_k) - P(y = 1|X, x_k - 0.5\delta_k)$$

iv) Min-max change:

$$\frac{\partial y}{\partial x_k} = P(y = 1|X, x_k = x_k^{max}) - P(y = 1|X, x_k = x_k^{min})$$

6.5.2 Categorical variables

Categorical variables are converted to dummy variables to use in the regression. The regression method treats the dummy variable as a continuous variable and returns a coefficient for each dummy variable. To compute marginal effect for each such dummy variable, similar to the regression, we can treat it as a continuous variable, and compute the partial derivative of the response variable with respect to this variable. However, since these variables are discrete, using the partial derivative is inappropriate.

An alternative method is to compute the discrete change for each dummy variable with respect to the reference level of the categorical variable. If x_k is a dummy variable corresponding to the value v of a categorical variable and let x_l represent another dummy variable representing value w of the same categorical variable. The discrete difference with respect to x_k is defined as:

$$\Delta_{x_k} y = y_k^{set} - y_k^{unset}$$

where,

$$y_k^{set} = \beta_1 f_1 + \ldots + \beta_k f_k(x_k = 1) + \ldots + \beta_l f_l(x_l = 0) + \ldots,$$
$$\text{and}$$
$$y_k^{unset} = \beta_1 f_1 + \ldots + \beta_k f_k(x_k = 0) + \ldots + \beta_l f_l(x_l = 0) + \ldots.$$

If the y expression contains dummy variable (x_p) corresponding to the reference level of the categorical variable (represented as r), then that variable will be unset ($x_p = 0$) in y^{set} and set ($x_p = 1$) in y^{unset}. Note that in many cases, the dummy variable for the reference level does not appear in the regression model and unsetting the variable for value w when $w \neq r$ is enough in the second term (y^{unset}).

In MADlib, we only support the discrete difference method for dummy variables. Let's walk through the computation of marginal effect for `color_blue` when the input for the regression is

```
array[1, color_blue, color_green, degree_college, degree_college * color_blue,
      degree_college * color_green, gpa, gpa^2, degree_college * gpa,
      degree_college * gpa^2, weight]
```

for a specific row of the data. Here `color_blue` and `color_green` belong to the same categorical variable, with `color_red` being the reference variable for it (not included in the regressor list).

i) The value of `color_blue` would be set equal to 1

ii) The value of `color_blue` in its interaction terms would also be set to 1
(i.e. `degree_college * color_blue = degree_college`)

iii) The value of `color_green` would be set equal to 0

iv) The value of `color_green` in its interaction terms would also be set to 0
(i.e. `degree_college * color_green = 0`)

v) Compute resulting predicted values of response variable (y^{set})

vi) Set value of `color_blue` equal to 0

vii) Set value of `color_blue` in its interaction terms to 0
(i.e. `degree_college * color_blue = 0`)

viii) Set value of `color_green` equal to 0 and also in its interaction terms (i.e. `degree_college * color_green = 0`)

ix) Compute resulting predicted values of response variable (y^{unset})

x) Compute ($y^{set} - y^{unset}$)

6.5.3 AME vs MEM

There are two main methods of calculating the marginal effects for dependent variables.

i) The first uses the average of the marginal effects at every sample observation (AME).

ii) The second approach calculates the marginal effect for x_k by taking predicted probability calculated when all regressors are held at their mean value from the same formulation with the exception of variable under consideration (MEM).

It is generally viewed to be problematic to evaluate marginal effects at means (MEM) of dummy variables since means of dummies refer to nonexisting observations. In MADlib, we currently only provide the option to compute the average marginal effects (AME). In future versions, we aim to provide various options including marginal effects at means and marginal effects at a representative value. We currently do provide the option to compute the margial effect on a separate dataset, which could be used as a workaround by including just the mean value or the representative value.

6.5.4 Marginal effects for regression methods

Below we derive the marginal effects for the various regression methods in MADlib. We assume a general formulation of the output y as

$$y = g(\beta_1 f_1 + \beta_2 f_2 + \ldots + \beta_N f_N).$$

As stated above, each f_i is a function of all the base variables $x_1, x_2, \ldots, x_M$ and describes the interaction between the base variables. In the simple (non-interaction) case, $f_i = x_i$.

Let's represent the vector of coefficients as

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_N \end{bmatrix},$$

and the partial derivative of the x terms in the output with respect to each variable x_i (Jacobian matrix) as

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}, & \frac{\partial f_1}{\partial x_2}, & \frac{\partial f_1}{\partial x_3}, & \ldots, & \frac{\partial f_1}{\partial x_M} \\ \frac{\partial f_2}{\partial x_1}, & \frac{\partial f_2}{\partial x_2}, & \frac{\partial f_2}{\partial x_3}, & \ldots, & \frac{\partial f_2}{\partial x_M} \\ \frac{\partial f_3}{\partial x_1}, & \frac{\partial f_3}{\partial x_2}, & \frac{\partial f_3}{\partial x_3}, & \ldots, & \frac{\partial f_3}{\partial x_M} \\ & & \vdots & & \\ \frac{\partial f_N}{\partial x_1}, & \frac{\partial f_N}{\partial x_2}, & \frac{\partial f_N}{\partial x_3}, & \ldots, & \frac{\partial f_N}{\partial x_M} \end{bmatrix}$$

In the above equation, for each element $J_{n,m}$ the column corresponds to a base varnable (x_m) and the row corresponds to a term in the output expression (f_n). Let ∇f_n denote the n-th row of J (i.e.

gradient of f_n), and let $J(m) = \frac{\partial \boldsymbol{f}}{\partial x_m}$ denote the m-th column of J, where $\boldsymbol{f} = (f_1, \dots f_N)^T$. If x_k is a categorical variable and if discrete differences are required for it, then the column corresponding to x_k can be replaced by,

$$\begin{aligned}\frac{\partial \boldsymbol{f}}{\partial x_k} &= \boldsymbol{f^{set_k}} - \boldsymbol{f^{unset_k}} \\ &= \left[f_0^{set_k}, \quad f_1^{set_k}, \quad f_2^{set_k}, \quad \dots, \quad f_{N-1}^{set_k}\right]^T - \left[f_0^{unset_k}, \quad f_1^{unset_k}, \quad f_2^{unset_k}, \quad \dots, \quad f_{N-1}^{unset_k}\right]^T \\ &= \left[f_0^{set_k} - f_0^{unset_k}, \quad f_1^{set_k} - f_1^{unset_k}, \quad f_2^{set_k} - f_2^{unset_k}, \quad \dots, \quad f_{N-1}^{set_k} - f_{N-1}^{unset_k}\right]^T,\end{aligned}$$

where

$$\begin{aligned}&f_i^{set_k} = f_i(\dots, x_k = 1, x_l = 0, x_r = 0) \\ &\text{and} \\ &f_i^{unset_k} = f_i(\dots, x_k = 0, x_l = 0, x_r = 1),\end{aligned}$$

$\forall x_l \in$ (set of dummy variables related to x_k excluding the reference variable) and $x_r =$ reference variable of x_k (if present). The response probability corresponding to f^{set_k} is denoted as P^{set_k}.

6.5.4.1 Linear regression

For the linear regression equation of the form $y = \boldsymbol{f}^T\boldsymbol{\beta}$, the marginal effect for each variable (x_i) is the same as the coefficient for that variable (β_i). When interaction effects are present, the marginal effect will be,

$$ME = J^T\boldsymbol{\beta}$$

6.5.4.2 Logistic regression

In logistic regression:

$$\begin{aligned}P &= \frac{1}{1 + e^{-\boldsymbol{f}^T\boldsymbol{\beta}}} \\ &= \frac{1}{1 + e^{-z}}\end{aligned}$$

$$\implies \frac{\partial P}{\partial X_k} = P \cdot (1 - P) \cdot \frac{\partial z}{\partial x_k},$$

where the partial derivative in the last equation equals to β_k if there is no interaction terms. Thus the marginal effect for all variables presented as a vector will be,

$$ME = P \cdot (1 - P) \cdot J^T\boldsymbol{\beta}$$

For categorical variables, we compute the discrete difference as described in 6.5.2.
For variable x_k:

$$ME_k = P^{set} - P^{unset}$$

6.5.4.3 Multilogistic regression

The probabilities of different outcomes for multilogistic regression are expressed as,

$$P^l = P(y = l|\boldsymbol{x}) = \frac{e^{\boldsymbol{f}^T\boldsymbol{\beta^l}}}{1 + \sum_{q=1}^{L-1} e^{\boldsymbol{f}^T\boldsymbol{\beta^q}}} = \frac{e^{\boldsymbol{f}^T\boldsymbol{\beta^l}}}{\sum_{q=1}^{L} e^{\boldsymbol{f}^T\boldsymbol{\beta^q}}},$$

where $\boldsymbol{\beta^l}$ represents the coefficient vector for category l, with L being the total number of categories. The coefficients are set to zero for one of the outcomes, called the "base outcome" or the "reference category". Here, without loss of generality, we let $\boldsymbol{\beta^L} = \mathbf{0}$.

Thus,

$$\begin{aligned}\frac{\partial P^l}{\partial x_m} &= \frac{1}{\sum_{q=1}^{L} e^{\boldsymbol{f}^T\boldsymbol{\beta^q}}} \frac{\partial e^{\boldsymbol{f}^T\boldsymbol{\beta^l}}}{\partial x_m} - \frac{e^{\boldsymbol{f}^T\boldsymbol{\beta^l}}}{\left(\sum_{p=1}^{L} e^{\boldsymbol{f}^T\boldsymbol{\beta^p}}\right)^2} \cdot \frac{\partial}{\partial x_m} \sum_{q=1}^{L} e^{\boldsymbol{f}^T\boldsymbol{\beta^q}} \\ &= \frac{e^{\boldsymbol{f}^T\boldsymbol{\beta^l}}}{\sum_{q=1}^{L} e^{\boldsymbol{f}^T\boldsymbol{\beta^q}}} \frac{\partial \boldsymbol{f}^T\boldsymbol{\beta^l}}{\partial x_m} - \frac{e^{\boldsymbol{f}^T\boldsymbol{\beta^l}}}{\sum_{p=1}^{L} e^{\boldsymbol{f}^T\boldsymbol{\beta^p}}} \sum_{q=1}^{L} \frac{e^{\boldsymbol{f}^T\boldsymbol{\beta^q}}}{\sum_{p=1}^{L} e^{\boldsymbol{f}^T\boldsymbol{\beta^p}}} \frac{\partial \boldsymbol{f}^T\boldsymbol{\beta^q}}{\partial x_m}\end{aligned}$$

Hence, for every m-th variable x_m, we have the marginal effect for category l as,

$$ME^l_m = P^l \left(J(m)^T\boldsymbol{\beta^l} - \sum_{q=1}^{L-1} P^q \cdot J(m)^T\boldsymbol{\beta^q} \right).$$

Vectorizing the above equation, we get

$$ME^l = P^l \left(J^T\boldsymbol{\beta^l} - J^T B\boldsymbol{p} \right),$$

where $\boldsymbol{p} = (P^1, \ldots, P^{L-1})^T$ is a column vector and $B = (\boldsymbol{\beta}^1, \ldots, \boldsymbol{\beta}^{L-1})$ is a $N \times (L-1)$ matrix. Finally, we can simplify the computation of the marginal effects matrix as

$$ME = J^T B\, diag(\boldsymbol{p}) - J^T B\boldsymbol{p}\boldsymbol{p}^T.$$

Once again, for categorical variables, we compute the discrete difference as described in 6.5.2. For categorical variable x_k, the k-th row of ME will be,

$$ME_k = \left(\boldsymbol{p}^{\boldsymbol{set_k}} - \boldsymbol{p}^{\boldsymbol{unset_k}} \right)^T$$

6.5.5 Standard Errors

The delta method is a popular way to estimate standard errors of non-linear functions of model parameters. While it is straightforward to calculate the variance of a linear function of a random variable, it is not for a nonlinear function. The delta method therefore relies on finding a linear approximation of the function by using a first-order Taylor expansion.

We can approximate a function $g(x)$ about a value a as,

$$g(x) \approx g(a) + (x-a)g'(a)$$

Taking the variance and setting $a = \mu_x$,

$$Var(g(X)) \approx \left[g'(\mu_x)\right]^2 Var(X)$$

Linear Regression

Using this technique, to compute the variance of the marginal effects at a given observation value in *linear regression*, we obtain the standard error by first computing the marginal effect's derivative over the coefficients, which is a $M \times N$ matrix $S_{mn} = \frac{\partial ME_m}{\partial \beta_n}$

$$\begin{aligned} \boldsymbol{S} &= \frac{\partial J^T \boldsymbol{\beta}}{\partial \boldsymbol{\beta}} \\ &= J^T \end{aligned}$$

Using the delta method we can then compute the variance of the marginal effects as,

$$Var(ME) = S \cdot Var(\beta) \cdot S^T$$

where $Var(\beta)$ is a $N \times N$ matrix, S is a $M \times N$ matrix, M is the number of base variables, and N is the total number of terms in independent variable list (i.e. length of $\boldsymbol{\beta}$).

Note: The $Var(\boldsymbol{\beta})$ is computed using the training data employed by the underlying regression, but not the data used to compute the marginal effects. The delta matrix ($\boldsymbol{S}$) is computed over the data for which marginal effects is desired (averaged over the data for AME).

Logistic Regression

Similar to linear regression, we obtain the variance matrix by first computing the delta matrix, $\boldsymbol{S}$:

$$\begin{aligned} S_{mn} &= \frac{\partial}{\partial \beta_n}\left[P(1-P)\cdot\frac{\partial z}{\partial x_m}\right] \\ &= P(1-P)\cdot\frac{\partial}{\partial \beta_n}\left(\frac{\partial z}{\partial x_m}\right) + \frac{\partial\left[P(1-P)\right]}{\partial \beta_n}\cdot\frac{\partial z}{\partial x_m} \\ &= P(1-P)\cdot\frac{\partial^2 z}{\partial x_m \partial \beta_n} + P(1-P)(1-2P)\cdot\frac{\partial z}{\partial \beta_n}\cdot\frac{\partial z}{\partial x_m}, \\ \text{where } P &= \frac{1}{1+e^{-z}} \\ \text{and } z &= \boldsymbol{f}^T\boldsymbol{\beta}. \end{aligned}$$

Using the definition of z, we can simplify S a little bit

$$S_{mn} = P(1-P)\left(\frac{\partial f_n}{\partial x_m} + (1-2P)\cdot f_n \cdot J(m)^T\boldsymbol{\beta}\right)$$

Thus, n-th column of S

$$c_n(S) = P(1-P)\left[\nabla f_n^T + (1-2P)\cdot f_n \cdot J^T\boldsymbol{\beta}\right].$$

Vectorizing this equation to express for the complete matrix,

$$\boldsymbol{S} = P(1-P)\left(J^T + (1-2P)\cdot(J^T\boldsymbol{\beta})\boldsymbol{f}^T\right)$$

And for categorical variables, we replace $P(1-P)\cdot(\partial z/\partial x_m)$ with $\Delta_{x_m}P$ in S_{mn}, we get

$$\begin{aligned} S_{mn} &= \frac{\partial(P^{set} - P^{unset})}{\partial\beta_n} \\ &= P^{set}(1-P^{set})\cdot f_n^{set} - P^{unset}(1-P^{unset})\cdot f_n^{unset} \end{aligned}$$

Similar to linear regression, we can then compute the variance of the marginal effects as,

$$Var(ME) = S\cdot Var(\boldsymbol{\beta})\cdot S^T$$

where $Var(\boldsymbol{\beta})$ is computed on the original training dataset, while $\boldsymbol{S}$ is averaged over the dataset on which marginal effect is desired (could be just a single datapoint).

Multinomial Logistic Regression

For multinomial logistic regression, the coefficients $\boldsymbol{\beta}$ form a matrix of dimension $N\times(L-1)$ where L is the number of categories and N is the number of features (including interaction terms). In order to compute the standard errors on the marginal effects of category l for independent variable x_m, we need to compute the term $\frac{\partial ME_m^l}{\partial\beta_n^{l'}}$ for each $l'\in\{1\dots(L-1)\}$ and $n\in\{1\dots N\}$. Note that m here is restricted to be in the set $\{1\dots M\}$. The result is a column vector of length $(L-1)\times N$ denoted by $\frac{\partial ME_m^l}{\partial\boldsymbol{\beta}}$. Hence, for each category $l\in\{1\dots(L-1)\}$ and independent variable $m\in\{1\dots M\}$, we perform the following computation

$$Var(ME_m^l) = \frac{\partial ME_m^l}{\partial\boldsymbol{\beta}}^T V\frac{\partial ME_m^l}{\partial\boldsymbol{\beta}}, \tag{6.5.1}$$

where V is the variance-covariance matrix of the multinomial logistic regression vectorized in the **same order** in which the partial derivative is vectorized.

From our earlier derivation, we know that the marginal effect for multinomial logistic regression for the m-th index of data vector $\boldsymbol{x}$ is given as:

$$ME_m^l = P^l\left[J(m)^T\boldsymbol{\beta^l} - \sum_{q=1}^{L-1}P^qJ(m)^T\boldsymbol{\beta^q}\right]$$

where

$$P^l = P(y=l|\boldsymbol{x}) = \frac{e^{\boldsymbol{f\beta^l}}}{\sum_{q=1}^{L}e^{\boldsymbol{f\beta^q}}} \qquad \forall l\in\{1\dots(L-1)\}.$$

We now compute the term $\frac{\partial ME_m^l}{\partial\boldsymbol{\beta}}$. First, we define the indicator function, δ, as:

$$\delta_{i,j} = \begin{cases} 1 & \text{if } i=j \\ 0 & \text{otherwise} \end{cases}.$$

We can show that for each $l' \in \{1 \dots (L-1)\}$ and $n \in \{1 \dots N\}$, the partial derivative will be

$$\frac{\partial ME_m^l}{\partial \beta_n^{l'}} = \frac{\partial P^l}{\partial \beta_n^{l'}} \left[J(m)^T \boldsymbol{\beta^l} - \sum_{q=1}^{L-1} P^q J(m)^T \boldsymbol{\beta^q} \right] + P^l \left[\frac{\partial}{\partial \beta_n^{l'}} (J(m)^T \boldsymbol{\beta^l}) - \frac{\partial}{\partial \beta_n^{l'}} \left(\sum_{q=1}^{L-1} P^q J(m)^T \beta^q \right) \right]$$

where

$$\frac{\partial P^l}{\partial \beta_n^{l'}} = P^l f_n (\delta_{l,l'} - P^{l'})$$

The expression above can be simplified to obtain

$$\begin{aligned}
\frac{\partial ME_m^l}{\partial \beta_n^{l'}} &= P^l f_n (\delta_{l,l'} - P^{l'}) \left[J(m)^T \boldsymbol{\beta^l} - \sum_{q=1}^{L-1} P^q J(m)^T \boldsymbol{\beta^q} \right] + \\
&\quad P^l \left[\delta_{l,l'} \frac{\partial f_n}{\partial x_m} - P^{l'} f_n J(m)^T \boldsymbol{\beta^{l'}} + P^{l'} f_n \sum_{q=1}^{L-1} P^q J(m)^T \boldsymbol{\beta^q} - P^{l'} \frac{\partial f_n}{\partial x_m} \right] \\
&= f_n (\delta_{l,l'} - P^{l'}) \cdot P^l \left[J(m)^T \boldsymbol{\beta^l} - \sum_{q=1}^{L-1} P^q J(m)^T \boldsymbol{\beta^q} \right] + \\
&\quad P^l \left[\delta_{l,l'} \frac{\partial f_n}{\partial x_m} - f_n \cdot P^{l'} \left(J(m)^T \boldsymbol{\beta^{l'}} - \sum_{q=1}^{L-1} P^q J(m)^T \boldsymbol{\beta^q} \right) - P^{l'} \frac{\partial f_n}{\partial x_m} \right] \\
&= f_n (\delta_{l,l'} - P^{l'}) ME_m^l + P^l \left[\delta_{l,l'} \frac{\partial f_n}{\partial x_m} - f_n ME_m^{l'} - P^{l'} \frac{\partial f_n}{\partial x_m} \right].
\end{aligned}$$

Again, the above computation is performed for every $l \in \{1 \dots (L-1)\}$ (base outcome is skipped) and every $m \in \{1 \dots M\}$, with each computation returning a column vector of size $(L-1) \times N$.

For categorical variables, we use the discrete difference value of ME_m^l to compute the standard error as,

$$\begin{aligned}
\frac{\partial ME_m^l}{\partial \beta_n^{l'}} &= \frac{\partial P^{set_m,l}}{\partial \beta_n^{l'}} - \frac{\partial P^{unset_m,l}}{\partial \beta_n^{l'}} \\
&= P^{set_m,l} f_n^{set_m} (\delta_{l,l'} - P^{set_m,l'}) - P^{unset_m,l} f_n^{unset_m} (\delta_{l,l'} - P^{unset_m,l'}) \\
&= -\left(P^{set_m,l} f_n^{set_m} P^{set_m,l'} - P^{unset_m,l} f_n^{unset_m} P^{unset_m,l'} \right) + \\
&\quad \delta_{l,l'} \left(P^{set_m,l} f_n^{set_m} - P^{unset_m,l} f_n^{unset_m} \right),
\end{aligned}$$

where

$$P^{set_m,l} = \frac{e^{\boldsymbol{f^{set_m} \beta^l}}}{\sum_{q=1}^{L} e^{\boldsymbol{f^{set_m} \beta^q}}}$$

and

$$P^{unset_m,l} = \frac{e^{\boldsymbol{f^{unset_m} \beta^l}}}{\sum_{q=1}^{L} e^{\boldsymbol{f^{unset_m} \beta^q}}} \qquad \forall l \in \{1 \dots (L-1)\}.$$

6.6 Clustered Standard Errors

Adjusting standard errors for clustering can be important. For example, replicating a dataset 100 times should not increase the precision of parameter estimates. However, performing this procedure with the IID assumption will actually do this. Another example is in economics of education research, it is reasonable to expect that the error terms for children in the same class are not independent. Clustering standard errors can correct for this.

6.6.1 Overview of Clustered Standard Errors

Assume that the data can be separated into m clusters. Usually this can be down by grouping the data table according to one or multiple columns.

The estimator has a similar form to the usual sandwich estimator

$$S(\boldsymbol{\beta}) = B(\boldsymbol{\beta})M(\boldsymbol{\beta})B(\boldsymbol{\beta}) \tag{6.6.1}$$

The bread part is the same as sandwich estimator

$$B(\boldsymbol{\beta}) = \left(-\sum_{i=1}^{n} H(y_i, \boldsymbol{x}_i, \boldsymbol{\beta})\right)^{-1} \tag{6.6.2}$$

$$= \left(-\sum_{i=1}^{n} \frac{\partial^2 l(y_i, \boldsymbol{x}_i, \boldsymbol{\beta})}{\partial\beta_\alpha \partial\beta_\beta}\right)^{-1} \tag{6.6.3}$$

where H is the hessian matrix, which is the second derivative of the target function

$$L(\boldsymbol{\beta}) = \sum_{i=1}^{n} l(y_i, \boldsymbol{x}_i, \boldsymbol{\beta}) \ . \tag{6.6.4}$$

The meat part is different

$$M(\boldsymbol{\beta}) = A^T A \tag{6.6.5}$$

where the m-th row of A is

$$A_m = \sum_{i \in G_m} \frac{\partial l(y_i, \boldsymbol{x}_i, \boldsymbol{\beta})}{\partial \boldsymbol{\beta}} \tag{6.6.6}$$

where G_m is the set of rows that belong to the same cluster.

6.6.2 Implementation

We can compute the quantities of B and A for each cluster during one scan through the data table in an aggregate function. Then sum over all clusters to the full B and A in the outside of the aggregate function. At last, the matrix mulplitications are done in a separate function on master.

7 Clustering (k-Means et al.)

Author Florian Schoppmann (version 0.5 only)

History

- **v0.5** Initial revision of design document, complete rewrite of module, arbitrary user-specifyable distance and recentering functions
- **v0.3** Multiple seedings methods (kmeans++, random, user-specified list of centroids), multiple distance functions (corresponding recentering function hard-coded), simplified silhouette coefficient as goodness-of-fit measure
- **v0.1** Initial version, always use kmeans++ for seeding

Clustering refers to the problem of partitioning a set of objects into homogeneous subsets, i.e., such that objects in the same group have *similar* properties. Arguably the best-known clustering problem is *k-means*. Here, one is given n points $x_1, \ldots, x_n \in \mathbb{R}^d$, and the goal is to position k centroids $c_1, \ldots, c_k \in \mathbb{R}^d$ so that the sum of squared Euclidean distances between each point and its closest centroid is minimized. A cluster is identified by its centroid and consists of all points for which this centroid is closest. Formally, we wish to minimize the following objective function:

$$(c_1, \ldots, c_k) \mapsto \sum_{i=1}^{n} \min_{j=1}^{k} \mathit{dist}(x_i, c_j)\,,$$

where $\mathit{dist}(x, y) := \|x - y\|_2^2$. A straightforward generalization of the above minimization problem is to choose a different metric *dist*. Strictly speaking, this is no longer k-means; for instance, choosing $\mathit{dist}(x, y) = \|x - y\|_1$ yields the k-median problem instead.

Despite certain reservations, we follow the practice of many authors and use "k-means" also to refer to a particular algorithm (as opposed to an optimization problem), as discussed below.

7.1 Overview of Algorithms

The k-means problem is NP-hard in general Euclidean space (even for just two clusters) [2] and for a general number of clusters (even in the plane) [49]. However, the local-search heuristic proposed by Lloyd [48] performs reasonably well in practice. In fact, it is so ubiquitous today that it is often referred to as the standard algorithm or even just the k-means algorithm. At a high level, it works as follows:

i) Seeding phase: Find initial positions for k centroids $c_1, \ldots, c_k$.

ii) Assign each point $x_1, \ldots, x_n$ to its closest centroid.

iii) Move each centroid c to a position that minimizes the sum of distances between c and each point assigned to c. (Note that if "distance" refers to the squared Euclidean distance, then this position is the barycenter/mean of all points assigned to c.)

iv) If convergence has been reached, stop. Otherwise, goto (ii).

Since the value of the objective function decreases in every step, and there are only finitely many clusterings, the algorithm is guaranteed to converge to a local minimum [50, Section 16.4]. While it is known that there are instances for which Lloyd's heuristic takes exponentially many steps [70], it has been shown that the algorithm has polynomial smoothed complexity [4]—thus giving some theoretical explanation for good results in practice. With a clever seeding technique, Lloyd's heuristic is moreover $O(\log k)$-competitive [3].

7.1.1 Algorithm Variants

Seeding The quality of k-means is highly influenced by the choice of the seeding [3]. The following is a non-exhaustive list of options:

i) Manual: User-specified list of initial centroid positions.

ii) Uniformly at random: Choose the k centroids uniformly at random among the point set

iii) `k-means++`: Perform seeding so that the objective function is minimized in expectation [3]

iv) Use a different clustering algorithm for determining the seeding [52]

v) Run k-means with multiple seedings and choose the clustering with lowest cost

Repositioning Most k-means formulations in textbooks do not detail the case where a centroid has no points assigned to it. It is an easy observation that moving a stray centroid in this case can only decrease the objective function. This can be done in a simple way (move onto a random point) or more carefully (e.g., move so that the objective function is minimized in expectation).

Convergence Criterion There are several reasonable convergence criteria. E.g., stop when:

i) The number of repositioned centroids is below a given threshold

ii) The change in the objective drops below a given threshold

iii) The maximum number of iterations has been reached

iv) See, e.g., Manning et al. [50, Section 16.4] for more options.

Variable Number of Clusters The number of clusters k could be determined by the seeding algorithm (instead of being a parameter) [52]. Strictly speaking, however, the algorithm should not be called k-means in this case.

7.2 Seeding Algorithms

In the following, we describe the seeding methods to be implemented for MADlib.

7.2.1 Uniform-at-random Sampling

Uniform-at-random sampling just uses the algorithms described in Section 2.1.

7.2.2 k-means++

`k-means++` seeding [3] starts with a single centroid chosen randomly among the input points. It then iteratively chooses new centroids from the input points until there are a total of k centroids. The probability for picking a particular point is proportional to its minimum distance to any existing centroid. Intuitively, `k-means++` favors seedings where centroids are spread out over the whole range of the input points, while at the same time not being too susceptible to outliers.

7.2.2.1 Formal Description

Algorithm k-`means++`$(k, P, dist, C)$
Input: Number of desired centroids k, set P of points in $\mathbb{R}^d$, metric *dist*, set C of initial centroids
Output: Set of centroids C

1: **if** $C = \emptyset$ **then**
2: $\quad C \leftarrow \{$initial centroid chosen uniformly at random from $P\}$
3: **while** $|C| < k$ **do**
4: $\quad C \leftarrow C \cup \{$random $p \in P$ with probability proportional to $\min_{c \in C} dist(p, c)\}$

Runtime A naive implementation needs $\Theta(k^2 n)$ distance calculations, where $n = |P|$. A single distance calculation takes $O(d)$ time.

Space Store k centroids.

Subproblems The existing `weighted_sample` subroutine can be used for:

- Line 2: Sample uniformly at random
- Line 4: Sample according to a discrete probability distribution.

The number of distance calculations could be reduced by a factor of k if we store, for each point $p \in P$, the distance to its closest centroid. Then, each iteration only needs n distance calculations (i.e., only between the most recently added centroid and all points). In total, these are $\Theta(kn)$ distance calculations. Making this idea explicit leads to the following algorithm.

Algorithm k-`means++-ext`$(k, P, dist)$
Input: Number of desired centroids k, set P of points in $\mathbb{R}^d$, metric *dist*, set C of initial centroids
Output: Set of centroids C
Initialization/Precondition: For all $p \in P : \delta[p] = \min_{c \in C} dist(p, c)$ *(or* $\delta[p] = \infty$ *if* $C = \emptyset$*)*

1: **while** $|C| < k$ **do**
2: $\quad lastCentroid \leftarrow$ `weighted_sample`(P, δ) $\quad \triangleright$ δ denotes the mapping $p \mapsto \delta[p]$
3: $\quad C \leftarrow C \cup \{lastCentroid\}$
4: $\quad$ **for** $p \in P$ **do**
5: $\quad\quad$ **if** $dist(p, lastCentroid) < \delta[p]$ **then**
6: $\quad\quad\quad \delta[p] \leftarrow dist(p, lastCentroid)$

Tuning The inner for-loop in line 4 and `weighted_sample` in line 2 could be combined. With this improvement, only one pass over P is necessary.

Runtime $O(dkn)$ as explained before.

Space Store k centroids and n distances.

Scalability The outer while-loop is inherently sequential because the random variates in each iteration depend on all previous iterations. The inner loop, however, can be executed with data parallelism.

7.2.2.2 Implementation as User-Defined Function

In general, the performance benefit of explicitly storing points (i.e., choosing k-`means++-ext` over k-`means++`) depends on the DBMS, the data, and the operating environment. The pattern of updating temporary state is made a bit more awkward in PostgreSQL due to its legacy of versioned storage. PostgreSQL performs an update by first inserting a new row and then marking the old row as invisible [69, Section 23.1.2]. As a result, for updates that touch many rows it is typically faster to copy the updated data into a new table (i.e., `CREATE TABLE AS SELECT` and `DROP TABLE`) rather than issue an `UPDATE`. Given these constraints, we currently choose to only implement algorithm k-`means++` (but not k-`means++-ext`) as the user-defined function `kmeanspp_seeding`.

In- and Output The UDF expects the following arguments, and returns the following values:

Name	Description	Type	
In	`rel_source`	Relation containing the points as rows	relation
In	`expr_point`	Point coordinates, i.e., the point p	expression (floating-point vector)
In	`k`	Number of centroids	integer
In	`fn_dist`	Function returning the distance between two vectors	function
In	`initial_centroids`	Matrix containing the initial centroids as columns. This argument may be omitted (corresponding to an empty set C of initial centroids).	floating-point matrix
Out		Matrix containing the k centroids as columns	floating-point matrix

Components The set of centroids C is stored as a dense floating-point matrix that contains the centroids as columns vectors. Algoritm k-`means++` can be (roughly) translated into SQL as follow. We assume here that all function arguments are available as constants, and the matrix containing the centroids as columns is available as `centroids`. Line 2 becomes:

```
1: SELECT ARRAY[weighted_sample($expr_point, 1)]
2: FROM $rel_source
```

Line 4 is implemented using essentially the following SQL.

```
1: SELECT centroids || weighted_sample(
2:     $expr_point, (
3:         closest_column(
4:             centroids,
5:             $expr_point,
6:             fn_dist
```

```
7:       )).distance
8:   ) FROM $rel_source
```

See Section 3.2.1 for a description of `closest_column`.

7.2.2.3 Historical Implementations

Implementation details and big-data heuristics that were used in previous versions of MADlib are documented here for completeness.

v0.2.1beta and earlier In lines 2 and 4 of Algorithm k-`means++` use a random sample $P' \subsetneq P$.

Here P' will be a new random sample in each iteration. Under the a-priori assumption that a random point belongs to any of the k (unknown) clusters with equal probability, sample enough points so that with high probability (e.g., $p = 0.999$) there is a point from each of the k clusters.

This is the classical occupancy problem (also called balls-into-bins model) [28]: Throwing r balls into k bins, what is the probability that no bin is empty? The exact value is

$$u(r,k) = k^{-r} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^r \,.$$

For $r, k \to \infty$ so that $r/k = O(1)$ we have the limiting form $u(r,k) \to (1 - e^{-r/k})^k =: \widetilde{u}(r,k)$. Rearranging $\widetilde{u}(r,k) > p$ gives $-\log(1 - \sqrt[k]{p}) \cdot k < r$. The smallest r satisfying this inequality is chosen as the size of the sample set.

7.3 Standard algorithm for k-means clustering

The standard algorithm has been outlined in Section 7.1. The formal description and our implementation are given below.

7.3.1 Formal Description

Algorithm k-`means`$(k, P, dist)$
Input: Set of initial centroids C, set P of points, seeding strategy *Seeding*, metric *dist*, centroid function *centroid*, convergence strategy *Convergence*
Output: Set C of final means
Initialization/Precondition: $i = 0$

1: **repeat**
2: $\quad i \leftarrow i + 1$
3: $\quad C_{\text{old}} \leftarrow C$
4: $\quad C \leftarrow \bigcup_{c \in C} \{centroid\{p \in P \mid \arg\min_{c' \in C} dist(p, c') = c\}\}$
5: $\quad C \leftarrow Seeding(k, P, dist, C)$ ▷ Reseed "lost" centroids (if any)
6: **until** $Convergence(C, C_{\text{old}}, P, i)$

Runtime See discussion in Section 7.1.

Space Store $2k$ centroids (both sets C and C_{old})

Scalability The outer loop is inherently sequential. The recentering in line 4 is data-parallel (provided that the sets C and C_{old} are available on all computation nodes). Likewise, the convergence check in line 6 is data-parallel if it only relies on distances between points p and the set of centroids C, or the number of iterations.

7.3.2 Implementation as User-Defined Function

Algorithm *k*-`means` is implemented as the user-defined function `kmeans`. We choose to not make the convergence criterion a function argument but instead settle for parameters for the most typical criteria. Should the need arise, we might revoke that decision in the future. Moreover, the seeding strategy is currently not an argument, but `kmeanspp_seeding` is always used in line 5.

In- and Output The UDF expects the following arguments, and returns the following values:

Name	Description	Type	
In	`rel_source`	Relation containing the points as tuples	relation
In	`expr_point`	Point coordinates, i.e., the point p	expression (floating-point vector)
In	`initial_centroids`	Matrix containing the initial centroids as columns	floating-point matrix
In	`fn_dist`	Function returning the distance between two vectors	function
In	`agg_centroid`	Aggregate returning the centroid for a set of points	function
In	`max_num_iterations`	Convergence criterion: Maximum number of iterations	integer
In	`min_frac_reassigned`	Convergence criterion: Convergence is reached if the fraction of points being reassigned to another centroid drops below `conv_level`	floating-point
Out		Matrix containing the k centroids as columns	floating-point matrix

Components The set of centroids C is stored as a dense floating-point matrix that contains the centroids as columns vectors. Algoritm *k*-`means` can be (roughly) translated into SQL as follow. We assume here that all function arguments are available as constants, and the matrix containing the centroids is available as `centroids`. (These variables, which are unbound in the SQL statement, are shown in italic font.) Line 4 becomes:

```
1: SELECT matrix_agg(_centroid)
2: FROM (
3:     SELECT $agg_centroid(_point) AS _centroid
4:     FROM (
5:         SELECT
6:             $expr_point AS _point,
```

```
 7:             (closest_column(
 8:                 centroids,
 9:                 $expr_point,
10:                 fn_dist
11:             )).column_id AS _new_centroid_id
12:         FROM $rel_source
13:     ) AS _points_with_assignments
14:     GROUP BY _new_centroid_id
15: ) AS _new_centroids
```

See Section 3.1.1 for a description of `matrix_agg`.

It is a good idea to also compute the number of reassigned centroids, so that both line 4 and the convergence check in line 6 can be computed with one pass over the data. To that end, we extend the inner-most query to also compute the previous closest centroid (i.e., we do a second `closest_column` call where we pass matrix `old_centroids` as first argument). During the aggregations in the two outer queries, we can then count (or sum up, respectively) the number of points that have been reassigned.

A caveat during testing whether a point has been reassigned is that centroid IDs are not constant over iterations: `closest_column` returns a column index in the matrix `centroids`, and this matrix is the result of the `matrix_agg` aggregate—hence, the order of the columns is non-deterministic. We therefore cannot directly compare a column index from iteration i to a column index from iteration $i-1$, but instead need to translate the "new" index into an "old" index first. In order to do that, we extend the outermost query and also build up an array `old_centroid_ids`, where position i will contain the column index that centroid i had in the previous iteration. **A crucial assumption on the DBMS backend here is that the two aggregates `array_agg` and `matrix_agg` see all tuples in the same order.** Putting everything together, the query becomes:

```
 1: SELECT
 2:     matrix_agg(_centroid),         -- New value for: centroids
 3:     array_agg(_new_centroid_id),   -- New value for: old_centroid_ids
 4:     sum(_objective_fn),            -- New value for: objective_fn
 5:     CAST(sum(_num_reassigned) AS DOUBLE PRECISION) / sum(_num_points)
 6:                                    -- New value for: frac_reassigned
 7: FROM (
 8:     SELECT
 9:         (_new_centroid).column_id AS _new_centroid_id,
10:         sum((_new_centroid).distance) AS _objective_fn,
11:         count(*) AS _num_points,
12:         sum(
13:             CAST(
14:                 old_centroid_ids[(_new_centroid).column_id + 1] != _old_centroid_id
15:                 AS INTEGER
16:             )
17:         ) AS _num_reassigned,
18:         $agg_centroid(_point) AS _centroid
19:     FROM (
20:         SELECT
21:             $expr_point AS _point,
22:             closest_column(
23:                 centroids,
24:                 $expr_point,
25:                 fn_dist
26:             ) AS _new_centroid,
27:             (closest_column(
28:                 old_centroids,
29:                 $expr_point,
```

```
30:                     fn_dist
31:                 )).column_id AS _old_centroid_id
32:             FROM $rel_source
33:         ) AS _points_with_assignments
34:         GROUP BY (_new_centroid).column_id
35:     ) AS _new_centroids
```

Finally, line 5 is simply implemented by calling function `kmeanspp_seeding`. For a slight performance benefit, this function call should be guarded by a check if the number of centroids is lower than k.

8 Convex Programming Framework

Author Xixuan (Aaron) Feng

History **v0.5** Initial revision

The nature of MADlib drives itself to support many different kinds of data modeling modules, such as logistic regression, support vector machine, matrix factorization, etc. However, keeping up with the state of the art and experimenting with individual data modeling modules require significant development and quality-assurance effort. Therefore, to lower the bar of adding and maintaining new modules, it is crucial to identify the invariants among many important modules, in turn, abstract and encapsulate them as reusable components.

Bismarck [29] is such a unified framework that links many useful statistical modeling modules and the relational DBMS, by introducing a well-studied formulation, convex programming, in between. Incremental Gradient Descent (IGD) has also been shown as a very effective algorithm to solve convex programs in the relational DBMS environment. But it is natural that IGD does not always fit the need of MADlib users who are applying convex statistical modeling to various domains. Driven by this, convex programming framework in MADlib also implements other algorithms that solves convex programs, such as Newton's method and conjugate gradient methods.

8.1 Introduction

This section is to first explain, formally, the type of problems that we consider in the MADlib convex programming framework, and then give a few example modules.

8.1.1 Formulation

We support numerical optimization problems with an objective function that is a sum of many component functions [7], such as

$$\min_{w \in \mathbb{R}^N} \sum_{m=1}^{M} f_{z_m}(w),$$

where $z_m \in \mathcal{O}, m = 1, ..., M$, are observations, and $f_{z_m} : \mathbb{R}^N \to \mathbb{R}$ are convex functions. For simplicity, let $z_{1:M}$ denote $\{z_m \in \mathcal{O} | m = 1, ..., M\}$. Note: given $z_{1:M}$, let $F(w) = \sum_{m=1}^{M} f_{z_m}(w)$, and $F : \mathbb{R}^N \to \mathbb{R}$ is also convex.

8.1.2 Examples

Many popular models can be formulated in the above form, with f_{z_m} being properly specified.

Logistic Regression. The component function is given by

$$f_{(x_m, y_m)}(w) = \log(1 + e^{-y_m w^T x_m}),$$

where $x_m \in \mathbb{R}^N$ are values of independent variables, and $y_m \in \{-1, 1\}$ are values of the dependent variable, $m = 1, ..., M$.

Linear SVM with hinge loss. The component function is given by

$$f_{(x_m,y_m)}(w) = \max(0, 1 - y_m w^T x_m),$$

where $x_m \in \mathbb{R}^N$ are values of features, and $y_m \in \{-1, 1\}$ are values of the label, $m = 1, ..., M$. Bertsekas [7] gives many other examples across application domains.

8.2 Algorithms

Gradient Descent. A most-frequently-mentioned algorithm that solves convex programs is gradient descent. This is an iterative algorithm and the iteration is given by

$$w_{k+1} = w_k - \alpha_k \nabla F(w_k),$$

where, given $z_{1:M}$, $F(w) = \sum_{m=1}^{M} f_{z_m}(w)$, and α_k is a positive scalar, called stepsize (or step length). Gradient descent algorithm is simple but usually recognized as a slow algorithm with linear convergence rate, while other algorithms like conjugate gradient methods and Newton's method has super-linear convergence rates [55].

Line Search: A Class of Algorithms. Convex programming has been well studied in the past few decades, and two main classes of algorithms are widely considered: line search and trust region ([55], section 2.2). Because line search is more commonly deployed and discussed, we focus on line search in MADlib, although some of the algorithms we discuss in this section can also easily be formulated as trust region strategy. All algorithms of line search strategies have the iteration given by

$$w_{k+1} = w_k + \alpha_k p_k,$$

where $p_k \in \mathbb{R}^N$ is search direction, and stepsize α_k [55]. Specifiedly, for gradient descent, p_k is the steepest descent direction $-\nabla \sum_{m=1}^{M} f_{z_m}(w_k)$.

8.2.1 Formal Description of Line Search

Algorithm `line-search`$(z_{1:M})$
Input: Observation set $z_{1:M}$,
convergence criterion *Convergence*(),
start strategy *Start*(),
initialization strategy *Initialization*(),
transition strategy *Transition*(),
finalization strategy *Finalization*()
Output: Coefficients $w \in \mathbb{R}^N$
Initialization/Precondition: $iteration = 0, k = 0$

1: $w_{\text{new}} \leftarrow Start(z_{1:M})$
2: **repeat**
3: $\quad w_{\text{old}} \leftarrow w_{\text{new}}$
4: $\quad state \leftarrow Initialization(w_{\text{new}})$
5: $\quad$ **for** $m \in 1..M$ **do** ▷ Single entry in the observation set
6: $\quad\quad state \leftarrow Transition(state, z_m)$ ▷ Usually computing derivative
7: $\quad w_{\text{new}} \leftarrow Finalization(state)$
8: **until** $Convergence(w_{\text{old}}, w_{\text{new}}, iteration)$
9: **return** w_{new}

Programming Model. We above give the algorithm of generic line search strategy, in the fashion of the selected programming model supported by MADlib (mainly user-defined aggregate).

Parallelism. The outer loop is inherently sequential. We require the inner loop is data-parallel. Simple component-wise addition or model averaging [26] are used to merge two states. A merge function is not explicitly added to the pseudocode for simplicity. A separate discussion will be made when necessary.

Convergence criterion. Usually, following conditions are combined by AND, OR, or NOT.

i) The change in the objective drops below a given threshold (E.g., negative log-likelihood, root-mean-square error).

ii) The value of the objective matches some pre-computed value.

iii) The maximum number of iterations has been reached.

iv) There could be more.

In MADlib implementation, the computation of objective is paired up with line-search to share data I/O.

Start strategy. In most cases, zeros are used unless otherwise specified.

Transition and finalization strategies. The coefficients update code ($w_{k+1} \leftarrow w_k + \alpha_k p_k$) is put into either *Transition*() or *Finalization*(). These two functions contain most of the computation logic, for computing the search direction p_k. We discuss details of individual algorithms in the following sections. For simplicity, global iterator k is read and updated in place by these functions without specifed as an additional argument.

8.2.2 Incremental Gradient Descent (IGD)

A main challenge arises when we are handling large amount of data, $M \gg 1$, where the computation of $\nabla(\sum_{m=1}^{M} f_{z_m})$ requires a whole pass of the observation data which is usually expensive. What distinguishes IGD from other algorithms is that it approximates $\nabla(\sum_{m=1}^{M} f_{z_m}) = \sum_{m=1}^{M}(\nabla f_{z_m})$ by the gradient of a single component function ∇f_{z_m} [1]. The reflection of this to the pseudocode makes the coefficients update code ($w_{k+1} \leftarrow w_k + \alpha_k p_k$) in *Transition*() instead of in *Finalization*().

8.2.2.1 Initialization Strategy

Algorithm `initialization-igd`(w)
Input: Coefficients $w \in \mathbb{R}^N$
Output: Transition state *state*

1: $state.w_k \leftarrow w$
2: **return** *state*

[1] z_m is usually selected in a stochastic fashion. Therefore, IGD is also referred to as stochastic gradient descent. The convergence and convergence rate of IGD are well developed [7], and IGD is often considered to be very effective with M being very large [11].

8.2.2.2 Transition Strategy

Algorithm `transition-igd`$(state, z_m)$
Input: Transition state *state*,
observation entry z_m,
stepsize $\alpha \in \mathbb{R}^+$,
gradient function *Gradient*()
Output: Transition state *state*

1: $p_k \leftarrow -Gradient(state.w_k, z_m)$ ▷ Previously mentioned as $p_k = -\nabla f_{z_m}$
2: $state.w_{k+1} \leftarrow state.w_k + \alpha p_k$
3: $k \leftarrow k + 1$ ▷ In-place update of the global iterator
4: **return** *state*

Stepsize. In MADlib, we support only constant stepsize for simplicity. Although IGD with constant stepsizes does not even have convergence guarantee [7], but it works reasonably well in practice so far [29] with some proper tuning. Commonly-used algorithms to tune stepsize ([8], appendix C) are mostly heuristics and do not have strong guarantees on convergence rate. More importantly, these algorithms require many evaluations of the objective function, which is usually very costly in use cases of MADlib.

Gradient function. A function where IGD accepts computational logic of specified modules. In MADlib convex programming framework, currently, there is no support of objective functions that does not have gradient or subgradient. Those objective functions that is not linearly separable is not currently supported by the convex programming framework, such as Cox proportional hazards models [22].

8.2.2.3 Finalization Strategy

Algorithm `finalization-igd`$(state)$
Input: Transition state *state*
Output: Coefficients $w \in \mathbb{R}^N$

1: **return** $state.w_k$ ▷ Trivially return w_k

8.2.3 Conjugate Gradient Methods

Conjugate gradient methods that solve convex programs are usually refered to as nonlinear conjugate gradient mthods. The key difference between conjugate gradient methods and gradient descent is that conjuagte gradient methods perform adjustment of the search direction p_k by considering gradient directions of previous iterations in some intriguing way. We skip the formal desciption of conjugate gradient methods that can be found in the references (such as Nocedal & Wright [55], section 5.2).

8.2.3.1 Initialization Strategy

Algorithm `initialization-cg`(w)
Input: Coefficients $w \in \mathbb{R}^N$,
gradient value $g \in \mathbb{R}^N$ (i.e., $\sum_{m=1}^{M} \nabla f_{z_m}(w_{k-1})$),
previous search direction $p \in \mathbb{R}^N$
Output: Transition state *state*

1: $state.p_{k-1} \leftarrow p$
2: $state.g_{k-1} \leftarrow g$
3: $state.w_k \leftarrow w$
4: $state.g_k \leftarrow 0$
5: **return** *state*

8.2.3.2 Transition Strategy

Algorithm `transition-cg`($state, z_m$)
Input: Transition state *state*,
observation entry z_m,
gradient function *Gradient*()
Output: Transition state *state*

1: $state.g_k \leftarrow state.g_k + Gradient(state.w_k, z_m)$
2: **return** *state*

8.2.3.3 Finalization Strategy

Algorithm `finalization-cg`(*state*)
Input: Transition state *state*,
stepsize $\alpha \in \mathbb{R}^+$,
update parameter strategy *Beta*()
Output: Coefficients $w \in \mathbb{R}^N$,
gradient value $g \in \mathbb{R}^N$ (i.e., $\sum_{m=1}^{M} \nabla f_{z_m}(w_{k-1})$),
previous search direction $p \in \mathbb{R}^N$

1: **if** k = 0 **then**
2: $state.p_k \leftarrow -state.g_k$
3: **else**
4: $\beta_k \leftarrow Beta(state)$
5: $p_k \leftarrow -state.g_k + \beta_k p_{k-1}$
6: $state.w_{k+1} \leftarrow state.w_k + \alpha p_k$
7: $k \leftarrow k + 1$
8: $p \leftarrow p_{k-1}$ ▷ Implicitly returning
9: $g \leftarrow state.g_{k-1}$ ▷ Implicitly returning again
10: **return** $state.w_k$

Update parameter strategy. For cases that F is strongly convex quadratic (e.g., ordinary least squares), β_k can be computed in closed form, having p_k be in conjugate direction of $p_0, ..., p_{k-1}$. For more general objective functions, many different choices of update parameter are proposed [37, 55], such as

$$\beta_k^{FR} = \frac{\|g_k\|^2}{\|g_{k-1}\|^2},$$

$$\beta_k^{HS} = \frac{g_k^T(g_k - g_{k-1})}{p_{k-1}^T(g_k - g_{k-1})},$$

$$\beta_k^{PR} = \frac{g_k^T(g_k - g_{k-1})}{\|g_{k-1}\|^2},$$

$$\beta_k^{DY} = \frac{\|g_k\|^2}{p_{k-1}^T(g_k - g_{k-1})},$$

where $g_k = \sum_{m=1}^{M} \nabla f_{z_m}(w_k)$, and $p_k = -g_k + \beta_k p_{k-1}$. We choose the strategy proposed by Dai and Yuan due to lack of mechanism for stepsize tuning in MADlib, which is required for other alternatives to guarantee convergence rate. (See Theorem 4.1 in Hager and Zhang [37]). In fact, lack of sufficient stepsize tuning for each iteration individually could make conjugate gradient methods have similar or even worse convergence rate than gradient descent. This should be fixed in the future.

8.2.4 Newton's Method

Newton's method uses a search direction other than the steepest descent direction – *Newton direction*. The Newton direction is very effective in the cases that the objective function is not too different from a quadratic approximation, and it gives quadratic convergence rate by considering Taylor's theorem. Formally, the Newton direction is given by

$$p_k = -(\nabla^2 F(w_k))^{-1} \nabla F(w_k),$$

where, given $z_{1:M}$, $F(w) = \sum_{m=1}^{M} f_{z_m}(w)$, and $H_k = \nabla^2 F(w_k)$ is called the Hessian matrix.

8.2.4.1 Initialization Strategy

Algorithm `initialization-newton`(w)
Input: Coefficients $w \in \mathbb{R}^N$
Output: Transition state *state*

1: $state.w_k \leftarrow w$
2: $state.g_k \leftarrow 0$
3: $state.H_k \leftarrow 0$
4: **return** *state*

8.2.4.2 Transition Strategy

Algorithm `transition-newton`($state, z_m$)
Input: Transition state *state*,
observation entry z_m,
gradient function *Gradient*(),
Hessian matrix function *Hessian*()
Output: Transition state *state*

1: $state.g_k \leftarrow state.g_k + Gradient(state.w_k, z_m)$
2: $state.H_k \leftarrow state.H_k + Hessian(state.w_k, z_m)$
3: **return** *state*

8.2.4.3 Finalization Strategy

Algorithm `finalization-newton`($state$)
Input: Transition state *state*
Output: Coefficients $w \in \mathbb{R}^N$

1: $p_k \leftarrow -(state.H_k)^{-1} state.g_k$
2: $state.w_{k+1} \leftarrow state.w_k + p_k$

3: $k \leftarrow k + 1$
4: **return** *state.*w_k

Hessian Matrix Function. A function where Newton's method accepts another computational logic of specified modules. See also gradient function.

Inverse of the Hessian Matrix. The inverse of Hessian matrix may not always exist if the Hessian is not guaranteed to be positive definite ($\nabla^2 F = 0$ when F is linear). We currently only support Newton's method for objetcive functions that is strongly convex. This may sometimes mean an objective function that is not globally strongly convex but Newton's method works well with a good starting point as long as the objective function is strongly convex in a convex set that contains the given starting point and the minimum. A few techniques that modify the Newton's method to adapt objective functions that are not strongly convex can be found in the references [8, 55].

Feed a Good Start Point. Since Newton's method is sensitive to the start point w_0, we provide a start strategy *Start*() to accept a start point that may not be zeros. It may come from results of other algorithms, e.g., IGD.

8.3 Implemented Machine Learning Algorithms

We have implemented several machine learning algorithms under the framework of convex optimization.

8.3.1 Linear Ridge Regression

Ridge regression is the most commonly used method of regularization of ill-posed problems. Mathematically, it seeks to minimize

$$Q\left(\boldsymbol{w}, w_0; \lambda\right) \equiv \min_{\boldsymbol{w}, w_0} \left[\frac{1}{2N} \sum_{i=1}^{N} \left(y_i - w_0 - \boldsymbol{w} \cdot \boldsymbol{x}_i\right)^2 + \frac{\lambda}{2} \|\boldsymbol{w}\|_2^2 \right] , \tag{8.3.1}$$

for a given value of λ, where $\boldsymbol{w}$ and w_0 are the fitting coefficients, and λ is a non-negative regularization parameter. $\boldsymbol{w}$ is a vector in d dimensional space, and

$$\|\boldsymbol{w}\|_2^2 = \sum_{j=1}^{d} w_j^2 = \boldsymbol{w}^T \boldsymbol{w} \ . \tag{8.3.2}$$

When $\lambda = 0$, Q is the mean squared error of the fitting.

The intercept term w_0 is not regularized, because this term is fully decided by the mean values of y_i and $\boldsymbol{x}_i$ and the values of $\boldsymbol{w}$, and does not affect the model's complexity.

$Q\left(\boldsymbol{w}, w_0; \lambda\right)$ is a quadratic function of $\boldsymbol{w}$ and w_0, and thus can be solved analytically

$$\boldsymbol{w}_{ridge} = \left(\lambda \boldsymbol{I}_d + \boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y} \ . \tag{8.3.3}$$

By using the available Newton method (Sec. 6.2.4), the above quantity can be easily calculated from one single step of the Newton method.

Many packages for Ridge regularization actually regularize the fitting coefficients not for the fitting model for the original data but for the data that has be scaled. MADlib also provides this option. When the normalization parameter is set to be True, which is by default False, the data will be first converted to the following before applying the Ridge regularization.

$$x_i' \leftarrow \frac{x_i - \langle x_i \rangle}{\langle (x_i - \langle x_i \rangle)^2 \rangle} \ , \tag{8.3.4}$$

$$y_i \leftarrow y_i - \langle y_i \rangle \ , \tag{8.3.5}$$

where $\langle \cdot \rangle = \sum_{i=1}^{N} \cdot / N$.

Note that Ridge regressions for scaled data and un-scaled data are not equivalent.

9 Low-rank Matrix Factorization

Author Xixuan (Aaron) Feng (version 0.5 only)

History
- **v0.5** Initial revision of design document, Implementation based on incremental gradient descent
- **v0.1** Initial revision (somewhat misleadingly called SVD matrix factorization at that time)

This module implements "factor model" for representing an incomplete matrix using a low-rank approximation [67]. Mathematically, this model seeks to find matrices U and V (also referred as factors) that, for any given incomplete matrix A, minimizes:

$$\|\boldsymbol{A} - \boldsymbol{U}V^T\|_2$$

subject to $rank(\boldsymbol{U}V^T) \leq r$, where $\|\cdot\|_2$ denotes the Frobenius norm. Let A be a $m \times n$ matrix, then U will be $m \times r$ and V will be $n \times r$, in dimension, and $1 \leq r \ll \min(m, n)$. This model is not intended to do the full decomposition, or to be used as part of inverse procedure. This model has been widely used in recommendation systems (e.g., Netflix [6]) and feature selection (e.g., image processing [77]).

9.1 Incremental Gradient Descent

9.1.1 Solving as a Convex Program

Recent work [19, 61] has demonstrated that the low-rank matrix factorization can be solved as a convex programming problem. This body of work enables us to solve the problem by using gradient-based line search algorithms. Among many of these algorithms, incremental gradient descent algorithm is a popular choice, especially for really large input matrices [29, 33].

9.1.2 Formal Description

Algorithm `lmf-igd`(r, A, α)
Input: Sparse matrix A,
step size α,
low-rank constraint r,
convergence criterion *Convergence*,
random factor generator *GenerateFactor*
Output: Factors U $(m \times r)$ and V $(n \times r)$
Initialization/Precondition: $iteration = 0$

1: $U \leftarrow GenerateFactor(m, r)$
2: $V \leftarrow GenerateFactor(n, r)$
3: **repeat**
4: $\quad iteration \leftarrow iteration + 1$

5: $U_{\text{old}} \leftarrow U$
6: $V_{\text{old}} \leftarrow V$
7: **for** $(i, j, y) \in A$ **do** ▷ Single entry in sparse matrix A
8: $e \leftarrow U_i \cdot V_j - y$
9: $temp \leftarrow U_i - \alpha e V_j$
10: $V_j \leftarrow V_j - \alpha e U_i$ ▷ In-place update of V
11: $U_i \leftarrow temp$ ▷ In-place update of U
12: **until** $Convergence(U_{\text{old}}, V_{\text{old}}, U, V, iteration)$

Runtime $O(N_A(m+n)r + mnr)$ for one iteration, where N_A is the number of nonzero elements in A.

Space Store the *temp*, an r-floating-point vector.

Parallelism The outer loop is inherently sequential. The inner loop is data-parallel and model averaging [26] is used.

Factor initialization The author of this document is not aware that significant differences are caused if random factors are initialized by different distributions. But zero values should be avoided. And entries in factors should not be initialized as the same value; otherwise, factors will always be rank 1.

Convergence criterion Usually, following conditions are combined by AND, OR, or NOT.

i) The change in the objective drops below a given threshold (E.g., RMSE).

ii) The value of the objective matches some pre-computed value.

iii) The maximum number of iterations has been reached.

iv) There could be more.

10 Latent Dirichlet Allocation (LDA)

Author Shengwen Yang (version 0.6 only)

History
- **v0.6** Initial revision of design document, complete rewrite of module, standard parallel implementation (support for local model update tuple by tuple and global model update iteration by iteration)
- **v0.1** Initial version (An approximated implementation which has memory problem for big datasets)

LDA is a very popular technique for topic modeling. This module implements a parallel Gibbs sampling algorithm for LDA inference.

10.1 Overview of LDA

LDA[9] is a very popular technique for discovering the main themes or topics from a large collection of unstructured documents and has been widely applied to various fields, including text mining, computer vision, finance, bioinformatics, cognitive science, music, and social sciences.

With LDA, a document can be represented as a random mixture of latent topics, where each topic can be characterized by a probability distribution over a vocabulary of words. Given a large text corpus, LDA will be able to infer a set of latent topics from the corpus, each represented with a multinomial distribution over words, denoted as $P(w|z)$, and infer the topic distribution for each document, represented as a multinomial distribution over topics, denoted as $P(z|d)$.

Several methods have been proposed for the inference of LDA, including variational Bayesian, expectation propagation, and Gibbs sampling[35]. Among of these methods, Gibbs sampling is the most widely used one because it is simple, fast, and has very few adjustable parameters. Besides, Gibbs sampling is easy to parallelize and easy to scale up, which allows us to utilize a cluster of machines to deal with very big datasets.

10.2 Gibbs Sampling for LDA

10.2.1 Overview

Althouth the derivation of Gibbs sampling for LDA is complicated, the results are very simple. The following equation tells us how to sample a new topic for a word in a corpus:

$$P(z_i = k|\mathbf{z}_{-i}, \mathbf{w}) \propto \frac{nwz_{-i,k}^{(w_i)} + \beta}{nz_{-i,k} + W\beta} \times (ndz_{-i,k}^{(d_i)} + \alpha) \tag{10.2.1}$$

where:

- i - index of word in the corpus
- d_i - docid of the i_{th} word

- w_i - wordid of the i_{th} word
- k - the k_{th} topic, where $1 <= k <= T$, and T is the number of topics
- z_i - topic assignment for the i_{th} word
- $\mathbf{z}_{-i}$ - topic assignments for other words excluding the i_{th} word
- $\mathbf{w}$ - all words in the corpus
- ndz - per-document topic counts
- nwz - per-word topic counts
- nz - corpus-level topic counts

According to this equation, we can update the topic assignment to each word sequnetially. This process can be iterated enough times until the conditional distribution reachs a stable state.

10.2.2 Parallization

The parallization of the above algirhtm is very straightforward. The basic idea is to distribute a large set of documents to a cluster of segment nodes and allow each segment node to do Gibbs sampling on a subset of documents locally. Note that at the end of each iteration, the local models generated on each segment node will be merged to generate a global model, which will be distributed to each segment node at the begining of next iteration.

Refer to [74] for a similar parallel implementation based on MPI and MapReduce.

10.2.3 Formal Description

Algorithm `gibbs-lda`(D, T, α, β)
Input:
Dataset D,
topic number T,
prior on per-document topic distribution α,
prior on per-word topic distribution β
Output:
Per-document topic distribution $P(z|d)$,
per-word topic distribution $P(z|w)$

1: $ndz \leftarrow 0$
2: $nwz \leftarrow 0$
3: $nz \leftarrow 0$
4: **for** $d \in D$ **do**
5: **for** $w \in W_d$ **do**
6: $z \leftarrow \text{random}(T)$
7: $ndz[d, z] \leftarrow ndz[d, z] + 1$
8: $nwz[w, z] \leftarrow nwz[w, z] + 1$
9: $nz[z] \leftarrow nz[z] + 1$
10: $Z[d, w] \leftarrow z$
11: **repeat**
12: **for** $d \in D$ **do**

for $w \in W_d$ **do**
  $z_{old} \leftarrow Z[d, w]$
  $z_{new} \leftarrow$ gibbs-sample$(z_{old}, ndz, nwz[w], nz, \alpha, \beta)$
  $Z[d, w] \leftarrow z_{new}$
  $ndz[d, z_{old}] \leftarrow ndz[d, z_{old}] - 1$
  $nwz[w, z_{old}] \leftarrow nwz[w, z_{old}] - 1$
  $nz[z_{old}] \leftarrow nz[z_{old}] - 1$
  $ndz[d, z_{new}] \leftarrow ndz[d, z_{new}] + 1$
  $nwz[w, z_{new}] \leftarrow nwz[w, z_{new}] + 1$
  $nz[z_{new}] \leftarrow nz[z_{new}] + 1$
until Stop condition is satisfied
$P(z|d) \leftarrow$ normalize(ndz, α)
$P(z|w) \leftarrow$ normalize(nwz, β)

Parallelism The inner loop is sequential. The outer loop is data-parallel and model averaging is used.

10.2.4 Implementation as User-Defined Function

Algorithm`gibbs-lda` is implemented as the user-defined function `lda_train`.

Name	Description	Type	
In	`data_table`	Table containing the training dataset	Relation
In	`voc_size`	Size of vocabulary	Integer
In	`topic_num`	Number of topics	Integer
In	`iter_num`	Number of iterations	Integer
In	`alpha`	Prior on per-document topic distribution	Double
In	`beta`	Prior on per-word topic distribution	Double
Out	`model_table`	Table containing the model information	Relation
Out	`output_data_table`	Table containing the per-document topic counts and topic assignments	Relation

Internally, two work tables are used alternately in the iterative Gibbs sampling process, one as input, another as output. The key part of an iteration is implemented essentially using the following SQL:

```
INSERT INTO work_table_out
SELECT
    distid,
    docid,
    wordcount,
    words,
    counts,
    madlib.__newplda_gibbs_sample(
        words,
        counts,
        doc_topic,
        model,
```

```
13:         alpha,
14:         beta,
15:         voc_size,
16:         topic_num)
17: FROM
18: (
19:     SELECT
20:         distid,
21:         docid,
22:         wordcount,
23:         words,
24:         counts,
25:         doc_topic,
26:         model
27:     FROM
28:     (
29:         SELECT
30:             madlib.__newplda_count_topic_agg(
31:                 words,
32:                 counts,
33:                 doc_topic[topic_num + 1:array_upper(doc_topic, 1)]
34:                                 AS topic_assignment,
35:                 voc_size,
36:                 topic_num) model
37:         FROM
38:             work_table_in
39:     ) t1
40:     JOIN
41:     work_table_in
42: ) t2
```

Note that within the `madlib.__newplda_gibbs_sample` function, the `model` parameter will be read in the first invocation and stored in the memory. In the incoming invocations within the same query, the parameter will be ignored. In this way, the model can be updated by an invocation and the updated model can be transferred to the next invocation.

The above SQL can be further rewritten to eliminate the data redundancy and reduce the overhead of joining operation, and thus improve the overall performance. This is very useful when the product of $voc\_size \times topic\_num$ is very large. See below for the rewritten SQL:

```
1: INSERT INTO work_table_out
2: SELECT
3:     distid,
4:     docid,
5:     wordcount,
6:     words,
7:     counts,
8:     madlib.__newplda_gibbs_sample(
9:         words,
10:        counts,
11:        doc_topic,
12:        model,
13:        alpha,
14:        beta,
15:        voc_size,
16:        topic_num)
17: FROM
18: (
19:     SELECT
```

```
20:         dcz.distid,
21:         dcz.docid,
22:         dcz.wordcount,
23:         dcz.words,
24:         dcz.counts,
25:         dcz.doc_topic,
26:         chunk.model
27:     FROM
28:     (
29:         SELECT
30:             distid, docid, model
31:         FROM
32:         (
33:             SELECT
34:                 madlib.__newplda_count_topic_agg(
35:                     words,
36:                     counts,
37:                     doc_topic[topic_num + 1:array_upper(doc_topic, 1)]
38:                             AS topic_assignment,
39:                     voc_size,
40:                     topic_num) model
41:             FROM
42:                 work_table_in
43:         ) t1,
44:         (
45:             SELECT
46:                 distid,
47:                 min(docid) docid
48:             FROM
49:                 work_table_in
50:             GROUP BY distid
51:         ) t2 -- One row per-segment
52:     ) chunk -- Ideally only one row per-segment
53:     RIGHT JOIN work_table_in dcz
54:     ON (dcz.distid = chunk.distid AND dcz.docid = chunk.docid)
55:     ORDER BY distid, docid -- Local data manipulation, no data redistribution
56: ) joined -- Ideally only one row per-segment has the fully joined data
```

11 Linear-chain Conditional Random Field

Conditional random field(CRF) [46] is a type of discriminative undirected probabilistic graphical model. Linear-chain CRFs are special CRFs which assume that the next state depends only on the current state. Linear-chain CRFs achieve state of the art accuracy in some real world natural language processing tasks such as part of inforamtion extraction[21], speech tagging(POS) and named entity resolution(NER).

11.1 Linear-chain CRF Learning

11.1.1 Mathematical Notations

- $p(\boldsymbol{Y}|\boldsymbol{X})$: conditional probability distributions of label sequence $\boldsymbol{Y}$ given input sequence $\boldsymbol{X}$.
- M: total number of unique features.
- I: the position of last token in a sentence.
- N: number of sentences in the training data set.
- λ: the coefficients (feature weights).
- ℓ_λ: log-likelihood summed over all training sentences.
- $\nabla\ell_\lambda$: gradient vector summed over all training sentences.
- ℓ'_λ: adjusted log-likelihood to avoid overfitting using spherical Gaussian weight prior.
- $\nabla\ell'_\lambda$: adjusted gradient vector to avoid overfitting using spherical Gaussian weight prior.

11.1.2 Formulation

A linear-chain CRF [64] is a distribution

$$p(\boldsymbol{Y}|\boldsymbol{X}) = \frac{\exp\sum_{m=1}^{M}\sum_{i=0}^{I}\lambda_m f_m(y_i, y_{i-1}, x_i)}{Z(X)},$$

where Z(X) is an instance specific normalizer

$$Z(X) = \sum_{y}\exp\sum_{m=1}^{M}\sum_{i=0}^{I}\lambda_m f_m(y_i, y_{i-1}, x_i).$$

Train a CRF by maximizing the log-likelihood of a given training set $T = \{(\boldsymbol{x^{(k)}}, \boldsymbol{y^{(k)}})\}_{k=1}^{N}$. Seek the zero of the gradient.

$$\ell_\lambda = \sum_k \log p_\lambda(\boldsymbol{y^{(k)}}|\boldsymbol{x^{(k)}}) = \sum_k[\lambda F(\boldsymbol{y^{(k)}}, \boldsymbol{x^{(k)}}) - \log Z_\lambda(\boldsymbol{x^{(k)}})]$$

$$\nabla \ell_\lambda = \sum_k [F(\boldsymbol{y^{(k)}}, \boldsymbol{x^{(k)}}) - E_{p_\lambda(Y|\boldsymbol{x^{(k)}})} F(Y, \boldsymbol{x^{(k)}})]$$

To avoid overfitting, we penalize the likelihood with a spherical Gaussian weight prior:

$$\ell'_\lambda = \sum_k [\lambda F(\boldsymbol{y^{(k)}}, \boldsymbol{x^{(k)}}) - \log Z_\lambda(\boldsymbol{x^{(k)}})] - \frac{\|\lambda\|^2}{2\sigma^2}$$

$$\nabla \ell'_\lambda = \sum_k [F(\boldsymbol{y^{(k)}}, \boldsymbol{x^{(k)}}) - E_{p_\lambda(Y|\boldsymbol{x^{(k)}})} F(Y, \boldsymbol{x^{(k)}})] - \frac{\lambda}{\sigma^2}$$

Note:We hard code σ as 100 in the implementation which follows other CRF packages in the literature.

11.1.3 Forward-backward Algorithm

$E_{p_\lambda(Y|x)} F(Y, x)$ is computed using a variant of the forward-backward algorithm:

$$E_{p_\lambda(Y|x)} F(Y, x) = \sum_y p_\lambda(y|x) F(y, x) = \sum_i \frac{\alpha_{i-1}(f_i * M_i)\beta_i^T}{Z_\lambda(x)}$$

$$Z_\lambda(x) = \alpha_I . 1^T$$

where α_i and β_i the forward and backward state cost vectors defined by

$$\alpha_i = \begin{cases} \alpha_{i-1} M_i, & 0 < i <= n \\ 1, & i = 0 \end{cases}, \beta_i^T = \begin{cases} M_{i+1} \lambda_{i+1}^T, & 1 <= i < I \\ 1, & i = n \end{cases}$$

11.1.4 L-BFGS Convex Solver

The limited-memory BFGS(L-BFGS) [54] is the limited memory variation of the Broyden-Fletcher-Goldfarb-Shanno(BFGS) algorithm which is the state of art of large scale non constraint convex optimization method. We translate the in-memory Java implementation to C++ in-database implementation using Eigen support. Eigen vector and Eigen matrix are used instead of the plain one dimentional and two dimentional arrays. In the Java in- memory implementation, it defines many static variables defined and shared between the interations. However, in the MADlib implementation, we define these variables in the state object. Before each iteration of L-BFGS optimization, we need to initialize the L-BFGS with the current state object. At the end of each iteration, we need to dump the updated variables to the database state for next iteration.

11.1.5 Parallel CRF Training

Algorithm `CRF training`$(z_{1:M})$
Input: Observation set $z_{1:M}$,
convergence criterion *Convergence*(),
start strategy *Start*(),
initialization strategy *Initialization*(),
transition strategy *Transition*(),
finalization strategy *Finalization*()
Output: Coefficients $w \in \mathbb{R}^N$
Initialization/Precondition: $iteration = 0, diag = \mathbf{1}$

1: $w_{\text{new}} \leftarrow Start(z_{1:M})$
2: **repeat**
3: $\quad w_{\text{old}} \leftarrow w_{\text{new}}$
4: $\quad state \leftarrow Initialization(w_{\text{new}})$
5: $\quad$ **for** $m \in 1..M$ **do** ▷ Single entry in the observation set
6: $\quad\quad state \leftarrow Transition(state, z_m)$ ▷ Computing gradient and log-likelihood.
7: $\quad w_{\text{new}} \leftarrow Finalization(state)$ ▷ Mainly invoke L-BFGS convex solver
8: **until** $Convergence(w_{\text{new}}, g_{\text{new}}, iteration)$
9: **return** w_{new}

Programming Model. We provide above the algorithm of parallel CRF training strategy, in the fashion of the selected programming model supported by MADlib (mainly user-defined aggregate).

Parallelism. The outer loop is inherently sequential over multiple iterations. The iteration $n+1$ takes the output of iteration n as input, so on so forth until the stop criterion is satisfied. The inner loop which calculates the gradient and log-likelihood for each document is data-parallel. Simple model averaging are used to merge two states. A merge function is not explicitly added to the pseudocode for simplicity. The finalization function invokes the L-BFGS convex solver to get a new solution. L-BFGS is sequential, but very fast. Experiments show that the speed-up ration approaches the number of segments configured in the Greenplum database.

Convergence criterion. Usually, the following conditions are combined by AND, OR, or NOT.

i) The norm of gradient divided by the norm of coefficient drops below a given threshold.

ii) The maximum number of iterations is reached.

iii) There could be more.

Start strategy. In most cases, zeros are used unless otherwise specified.

Transition strategies. This function contains the logic of computing the gradient and log-likelihood for each tuple using the forward-backward algorithm. The algorithms will be discussed in the following sections.

Algorithm `transition-lbfgs`$(state, z_m)$
Input: Transition state *state*,
observation entry z_m,
gradient function *Gradient*()
Output: Transition state *state*

1: $\{state.g, state.loglikelihood\} \leftarrow Gradient(state, z_m)$ ▷ using forward-backward algorithm to calculate gradient and loglikelihood
2: $state.num\_rows \leftarrow state.num\_rows + 1$
3: **return** *state*

Merge strategies. The merge function simply sums the gradient and log-likelihood over all training documents

Algorithm `merge-lbfgs`($state_1, state_2$)
Input: Transition state $state_1$,
Transition state $state_2$
Output: Transition state $state_{new}$

1: $state_{new}.g \leftarrow state_1.g + state_2.g$
2: $state_{new}.loglikelihood \leftarrow state_1.loglikelihood + state_2.loglikelihood$
3: **return** $state_{new}$

Finalization strategy. The finalization function invokes the L-BFGS convex solver to get a new coefficent vector.

Algorithm `finalization-lbfgs`($state$)
Input: Transition state $state$,
LBFGS $lbfgs()$
Output: Transition state $state$

1: $\{state.g, state.loglikelihood\} \leftarrow penalty(state.g, state.loglikelihood)$ ▷ To avoid overfitting, add penalization
2: $\{state.g, state.loglikelihood\} \leftarrow -\{state.g, state.loglikelihood\}$ ▷ negation for maximization
3: LBFGS instance($state$) ▷ initialize the L-BFGS instance with previous state
4: instance.$lbfgs()$ ▷ invoke the L-BFGS convex solver
5: instance.save_state($state$) ▷ save updated variables to the state for next iteration
6: **return** $state$

Feeding with current solution, gradient, log-likelihood, etc., the L-BFGS will output a new solution. To avoid overfitting, a penalization function is needed. We choose to penalize the log-likelihood with a spherical Gaussian weight prior. Also, L-BFGS is to maximum objective, so we need to negate the gradient vector and log-likelihood to fit our needs in order minimize the log-likehood.

11.2 Linear-chain CRF Applications

Linear-chain CRF can be used in various applications such as part of speech tagging and named entity resolution. All the following sections assume that the application is part of speech tagging. It can be fitted to named entity resolution with minimal effort.

11.2.1 Part of Speech Tagging

Part-of-speech tagging, also called grammatical tagging or word-category disambiguation [24], is the process of assigning a part of speech to each word in a sentence. POS has been widely used in information retrieval and text to speech. There are two distinct methods for POS task: rule-based and stochastic. In rule-based method, large collection of rules are defined to indentify the tag. Stochastic method is based on probabilistic graphic models such as hidden markov models and conditional random fields. In practice, conditional random fields are approved to achieve the state of art accuracy.

11.2.2 Tag Set

There are various tag set used in the literature. The Pennsylvania Treebank tag-set [51] is the commonly used tag set and contains 45 tags. The following table shows part of tags in the tag set.

Table 11.1: Pen Treebank III Tag Set

Tag	Description	Example	Tag	Description	Example
CC	Coordin,Conjunction	and,but,or	SYM	Symbol	+,%,&
CD	Cardinal number	one,two,three	TO	'to'	to
DT	Determiner	a,the	UH	Interjection	ah,oops
EX	Existential	there	VB	Verb,base form	eat
...	...	...	...	...	...
RBR	Adverb,comparative	faster	.	Sentence-final	(.!?)
RBS	Adverb,superlative	fastest	:	Mid-sentence punc	(:;...-)
RP	Particle	up,off			

11.2.3 Regular Expression Table

Regex feature captures the relationship between the morphology of a token and it's corresponding tag. For example, a token ending with 's' is mostly likely to be a plural noun whereas a token ending with 'ly' is more likely to be an adverb. One can define his/her own regular expressions to capture the intrinsic characteristics of the given training data.

Table 11.2: Regular expression table

pattern	name	pattern	name
$\wedge[A-Z][a-z]+\$$	InitCapital	$\wedge[A-Z]+\$$	isAllCapital
$\wedge.*[0-9]+.*\$$	containsDigit	$\wedge.+[.]\$$	endsWithDot
$\wedge.+[,]\$$	endsWithComma	$\wedge.+er\$$	endsWithEr
$\wedge.+est\$$	endsWithEst	$\wedge.+ed\$$	endsWithEd
$\wedge.+s\$$	endsWithS	$\wedge.+ing\$$	endsWithIng
$\wedge.+ly\$$	endsWithly	$\wedge.+-.+\$$	isDashSeparatedWords
$\wedge.*@.*\$$	isEmailId		

11.3 Feature Extraction

The Feature Extraction module provides functionality for basic text-analysis tasks such as part-of-speech (POS) tagging and named-entity resolution. At present, six feature types are implemented.

- Edge Feature: transition feature that encodes the transition feature weight from current label to next label.
- Start Feature: fired when the current token is the first token in a sentence.
- End Feature: fired when the current token is the last token in a sentence.

- Word Feature: fired when the current token is observed in the trained dictionary.
- Unknown Feature: fired when the current token is not observed in the trained dictionary for at least certain times.
- Regex Feature: fired when the current token can be matched by the regular expression.

Advantages of extracting features using SQL statements:

⋆ Decoupling the feature extraction and other code.

⋆ Compared with procedure language, SQL is much more easier to understand.

⋆ Storing all the features in tables avoids recomputing of features over iterations. It also boosts the performance.

⋆ SQL is naivelly paralleled.

11.3.1 Column Names Convention and Table Schema

11.3.1.1 Column Names Convention

The following column names are commonly used in the tables of the following sections.

- doc_id: Unique integer indentifier of a document.
- start_pos: Position of the token in the document starting from 0.
- seg_text: Text token itself.
- prev_label: Label of previous token.
- label: Label of current token.
- max_pos: End positon of the document.
- weight: Feature weight associated with certain feature
- f_index: Unique integer identifier of a feature
- f_name: Feature name.

11.3.1.2 Training and Testing Data Schema

The text data has to been tokenized before it can be stored in the database. One of the commonly used tokenization program for part-of-speech tagging is the Treebank tokenization script. The following table dipicts how the training/testing data is stored in the database table.

Table 11.3: Training or Testing data

start_pos	doc_id	seg_text	label	start_pos	doc_id	seg_text	label
0	1	'confidence'	11	1	1	'in'	5
2	1	'the'	2	3	1	'pound'	11
4	1	'is'	31	5	1	'widely'	19
6	1	'expected'	29	7	1	'to'	24
8	1	'take'	26	9	1	'another'	2
10	1	'sharp'	6	11	1	'dive'	11
12	1	'if'	5	13	1	'trade'	11
14	1	'figures'	12	15	1	'for'	5
16	1	'september'	13	17	1	','	42
18	1	'due'	6	19	1	'for'	5

11.3.2 Design Challanges and Work-arounds

As far as I know, the MADlib C++ abstraction layer doesn't support array of self-defined composite data types or multi-dimentional arrays. But we do have the need of these complex data types in the implemenation of this module. For example, the *viterbi_mtbl* table is indeed a two dimentional arrays. Due to the limitations of current C++ abstraction layer, we have to convert the matrix to an array and later index the data with $M[i * n + j]$ instead of the normal way $M[i][j]$. Another example is the data types to represent the features. A single feature cannot be represented by a single DOUBLE varialbe but by a unit of struct : $[prev\_label, label, f\_index, start\_pos, exist]$ But there is no arrays of struct type, we had to represent it with one dimentional array. Also we have to store the features for a document using array of doubles instead of array of struct.

11.3.3 Training Data Feature Extraction

Given training data, SQLs are writen to extract all the features. It happened that any type of features mentioned above can be extracted out by one single SQL clause which makes the code succinct. We illustrate the training data feature extraction by SQL clause examples.

Sample Feature Extraction SQLs for edge features and regex features

- SQL_1:

```
1: SELECT doc2.start_pos, doc2.doc_id, 'E.', ARRAY[doc1.label, doc2.label]
2: FROM   segmenttbl doc1, segmenttbl doc2
3: WHERE  doc1.doc_id = doc2.doc_id AND doc1.start_pos+1 = doc2.start_pos
```

- SQL_2:

```
1: SELECT start_pos, doc_id, 'R_' || name, ARRAY[-1, label]
2: FROM  regextbl, segmenttbl
3: WHERE seg_text ~ pattern
```

Build the feature dictionary and assign each feature with a unique feature id

- SQL_3

```
1:  INSERT INTO tmp_featureset(f_name, feature)
2:  SELECT DISTINCT f_name, feature
3:  FROM   tmp1_feature;
4:  INSERT INTO featureset(f_index, f_name, feature)
5:  SELECT nextval('seq')-1, f_name, feature
6:  FROM   tmp_featureset;
```

Generate sparse_r table

- SQL_3

```
1:  INSERT INTO rtbl(start_pos,doc_id,feature)
2:  SELECT start_pos, doc_id, array_cat(fset.feature,
3:          ARRAY[f_index,start_pos,
4:          CASE WHEN tmp1_feature.feature = fset.feature THEN 1
5:          ELSE 0 END] )
6:  FROM   tmp1_feature, featureset fset
7:  WHERE  tmp1_feature.f_name = fset.f_name AND fset.f_name <> 'E.';
```

The final input table schema which contains all the feature data for the crf learning algorithm is as follows:

doc_id	sparse_r	dense_m	sparse_m

- sparse r feature(single state feature):(prev_label, label, f_index, start_pos, exist)

label	Description
prev_label	the label of previous token, it is always 0 in r table.
label	the label of the single state feature
f_index	the index of the feature in the feature table
start_pos	the postion of the token(starting from 0)
exist	indicate whether the token exists or not in the acutal training data set

- dense m feature: (prev_label, label, f_index, start_pos, exist)

label	Description
prev_label	the label of previous token.
label	the label of current token
f_index	the index of the feature in the feature table
start_pos	the postion of the token in a sentence(starting from 0)
exist	indicate whether the token exists or not in the acutal training data set

- sparse m feature:(f_index, prev_label, label)

label	Description
f_index	the index of the feature in the feature table
prev_label	the label of previous token
label	the label of current token

For performance consideraton, we split the m feature to *dense_m* feature and *sparse_m* feature The acutal *spare_r* table is array union of individal r features ordered by the start positon of tokens. So the function to compute the gradient vector and log-likelihood can scan the feature arrays from beginning to end.

11.3.4 Learned Model

The CRF learning algorithm will generate two tables: feature table and dictionary table Feature table stores all the features and their corresponding feature weight. The dictionary constains all the tokens and the number of times they appear in the training data.

Table 11.4: Feature table

f_index	f_name	prev_label	label	weight	f_index	f_name	prev_label	label	weight
0	'U'	-1	6	2.037	1	'E.'	2	11	2.746
2	'W_exchequer'	-1	13	1.821	3	'W_is'	-1	31	1.802
4	'E.'	11	31	2.469	5	'W_in'	-1	5	3.252
6	'E.'	11	12	1.305	7	'U'	-1	2	-0.385
8	'E.'	31	29	1.958	9	'U'	-1	29	1.422
10	'R_endsWithIng'	-1	11	1.061	11	'W_of'	-1	5	3.652
12	'S.'	-1	13	1.829	13	'E.'	24	26	3.282
14	'W_helped'	-1	29	1.214	15	'E.'	11	24	1.556

Table 11.5: Dictionary table

token	total	token	total	token	total	token	total
'freefall'	1	'policy'	2	'measures'	1	'commitment'	1
'new'	1	'speech'	1	"'s'	2	'reckon'	1
'underlying'	1	'week'	1	'prevent'	1	'has'	2
'failure'	1	'restated'	1	'announce'	1	'thursday'	1
'but'	1	'lawson'	1	'last'	1	'firm'	1
'exchequer'	1	'helped'	1	'sterling'	2	...	...

11.3.5 Testing Data Feature Extraction

This component extracts features from the testing data based on the learned models. It will produce two factor tables *viterbi_mtbl* and *viterbi_rtbl*. The *viterbi_mtbl* table and a *viterbi_rtbl* table are used to calculate the best label sequence for each sentence.

Sample Feature Extraction SQLs

- SQL_1: Extracting unique tokens:

```
1:  INSERT INTO segment_hashtbl
2:  SELECT DISTINCT seg_text
3:  FROM   segmenttbl
```

- SQL_2: Summerize over all single state features with respect to specific tokens and labels :

```
1:  INSERT INTO viterbi_rtbl
2:  SELECT seg_text, label, SUM(value)
3:  FROM   rtbl
4:  GROUP BY seg_text,label
```

doc_id	*viterbi_mtbl*	*viterbi_rtbl*

- *viterbi_mtbl* table encodes the edge features which are solely dependent on upon current label and previous y value. The m table has three columns which are prev_label, label, and value respectively. If the number of labels is n, then the m factor table will have n^2 rows. Each row encodes the transition feature weight value from the previous label to the current label.

  startFeature is considered as a special edge feature which is from the beginning to the first token. Likewise, endFeature can be considered as a special edge feature which is from the last token to the very end. So m table encodes the edgeFeature, startFeature, and endFeature. If the total number of labels in the label space is 45 from 0 to 44, then the m factor array is as follows:

- *viterbi_r* table is related to specific tokens. It encodes the single state features, e.g., wordFeature, RegexFeature for all tokens. The r table is represented as shown in the table.

Table 11.6: viterbi_mtbl table

token	0	1	2	3	...	43	44
-1	2.1	1.1	1.0	1.1	1.1	2.1	1.1
0	1.1	3.9	1.2	2.1	2.8	1.8	0.8
1	0.7	1.7	2.9	3.8	0.6	3.2	0.2
2	0.2	3.2	3.8	2.9	0.2	0.1	0.2
3	1.2	6.9	7.8	8.0	0.1	1.9	1.7
...	...	...	...	...	...	...	...
44	8.2	1.8	3.7	2.1	7.2	1.3	7.2
45	1.8	7.8	5.6	9.8	2.3	9.4	1.1

Table 11.7: viterbi_rtbl table

token	0	1	2	3	...	43	44
madlib	0.2	4.1	0.0	2.1	0.1	2.5	1.2
is	1.3	3.0	0.2	3.1	0.8	1.9	0.9
an	0.9	1.1	1.9	3.8	0.7	3.8	0.7
open-source	0.8	0.2	1.8	2.7	0.5	0.8	0.1
library	1.8	1.9	1.8	8.7	0.2	1.8	1.1
...	...	...	...	...	...	...	...

11.4 Linear-chain CRF Inference

The Viterbi algorithm [71] is the popular algorithm to find the top-k most likely labelings of a document for CRF models. For the tasks in natural language processing domain, it is sufficient to only generate the best label sequence. We chose to use a SQL clause to drive the inference over all documents. In Greenplum, Viterbi can be run in parallel over different subsets of the document on a multi-core machine.

11.4.1 Parallel CRF Inference

The *vcrf_top_label* is implemented sequentially and each function call will finish labeling of one document. The inference is paralledl in the level of document. We use a SQL statment to drive the inference of all documents. So, the CRF inference is naivelly parallel.

```
1:      SELECT doc_id, vcrf_top1_label(mfactors.score, rfactors.score)
2:      FROM   mfactors,rfactors
```

11.4.2 Viterbi Inference Algorithm

$$V(i,y) = \begin{cases} \max_{y'}(V(i-1,y')) + \sum_{k=1}^{K} \lambda_k f_k(y,y',x_i), & \text{if } i \geq 0 \\ 0, & \text{if } i = -1. \end{cases}$$

11.4.3 Viterbi Inference output

The final inference output will produce the the best label sequence for each document and also the conditional probability given the observed input sequence.

Table 11.8: Viterbi inference output

doc_id	start_pos	token	label	probability
1	0	madlib	proper noun, singular	0.6
1	1	is	Verb, base form	0.6
1	2	an	determiner	0.6
1	2	open-source	adjective	0.6
1	4	library	noun	0.6
1	5	for	preposition	0.6
1	6	scalable	adjective	0.6
1	7	in-dababase	adverb	0.6
1	8	analytics	noun, singular	0.6
1	9	.	sentence-final punc	0.6
2	0	it	personal pronoun	0.4
2	1	provides	verb, base form	0.4
2	2	data-parallel	noun	0.4
2	2	...	...	0.4

12 ARIMA

Authors Mark Wellons

History **v0.1** Initial version

12.1 Introduction

An ARIMA model is an *a*uto-*r*egressive *i*ntegrated *m*oving *a*verage model. An ARIMA model is typically expressed in the form

$$(1 - \phi(B))Y_t = (1 + \theta(B))Z_t, \tag{12.1.1}$$

where B is the backshift operator. The time t is from 1 to N.

ARIMA models involve the following variables:

i) The lag difference Y_t, where $Y_t = (1 - B)^d(X_t - \mu)$.

ii) The values of the time series X_t.

iii) p, q, and d are the parameters of the ARIMA model. d is the differencing order, p is the order of the AR operator, and q is the order of the MA operator.

iv) The AR operator $\phi(B)$.

v) The MA operator $\theta(B)$.

vi) The mean value μ, which is always set to be zero for $d > 0$ or need to be estimated.

vii) The error terms Z_t.

12.1.1 AR & MA Operators

The auto regression operator models the prediction for the next observation as some linear combination of the previous observations. More formally, an AR operator of order p is defined as

$$\phi(B)Y_t = \phi_1 Y_{t-1} + \cdots + \phi_p Y_{t-p} \tag{12.1.2}$$

The moving average operator is similar, and it models the prediction for the next observation as a linear combination of the errors in the previous prediction errors. More formally, the MA operator of order q is defined as

$$\theta(B)Z_t = \theta_1 Z_{t-1} + \cdots + \theta_q Z_{t-q}. \tag{12.1.3}$$

12.2 Solving for the model parameters

12.2.1 Least Squares

We assume that

$$\Pr(Z_t) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-Z_t^2/2\sigma^2}, \quad t > 0 \tag{12.2.1}$$

and that $Z_{-q+1} = Z_{-q+2} = \cdots = Z_0 = Z_1 = \cdots = Z_p = 0$. The initial values of $Y_t = X_t - \mu$ for $t = -p+1, -p+2, \ldots, 0$ can be solved from the following linear equations

$$\begin{aligned}
\phi_1 Y_0 + \phi_2 Y_{-1} + \cdots + \phi_p Y_{-p+1} &= Y_1 \\
\phi_2 Y_0 + \cdots + \phi_p Y_{-p+2} &= Y_2 - \phi_1 Y_1 \\
&\vdots \\
\phi_{p-1} Y_0 + \phi_p Y_{-1} &= Y_{p-1} - \phi_1 Y_{p-2} - \cdots - \phi_{p-2} Y_1 \\
\phi_p Y_0 &= Y_p - \phi_1 Y_{p-1} - \cdots - \phi_{p-1} Y_1
\end{aligned} \tag{12.2.2}$$

The likelihood function L for N values of Z_t is then

$$L(\phi, \theta) = \prod_{t=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-Z_t^2/2\sigma^2} \tag{12.2.3}$$

so the log likelihood function l is

$$\begin{aligned}
l(\phi, \theta) &= \sum_{t=1}^{N} \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-Z_t^2/2\sigma^2}\right) \\
&= \sum_{t=1}^{N} -\ln\left(\sqrt{2\pi\sigma^2}\right) - \frac{Z_t^2}{2\sigma^2} \\
&= -\frac{N}{2}\ln\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{t=1}^{N} Z_t^2 \ .
\end{aligned} \tag{12.2.4}$$

Thus, finding the maximum likelihood is equivalent to solving the optimization problem (known as the conditional least squares formation)

$$\min_{\theta,\phi} \sum_{t=1}^{N} Z_t^2. \tag{12.2.5}$$

The error term Z_t can be computed iteratively as follows:

$$Z_t = X_t - F_t(\phi, \theta, \mu) \tag{12.2.6}$$

where

$$F_t(\phi, \theta, \mu) = \mu + \sum_{i=1}^{p} \phi_i (X_{t-i} - \mu) + \sum_{i=1}^{q} \theta_i Z_{t-i} \tag{12.2.7}$$

12.2.1.1 Levenberg-Marquardt Algorithm

In mathematics and computing, the Levenberg-Marquardt algorithm (LMA), also known as the damped least-squares (DLS) method, provides a numerical solution to the problem of minimizing a function, generally nonlinear, over a space of parameters of the function. These minimization problems arise especially in least squares curve fitting and nonlinear programming.

To understand the Levenberg-Marquardt algorithm, it helps to know the gradient descent method and the Gauss-Newton method. On many "reasonable" functions, the gradient descent method takes large steps when the current iterate is distant from the true solution, but is slow to converge an the current iterate nears the true solution. The Gauss-Newton method is much faster for converging when the current iterate is in the neighborhood of the true solution. The Levenberg-Marquardt algorithm tries to get the best of best worlds, and combine the gradient descent step with Gauss-Newton step in a weighted average. For iterates far from the true solution, the step favors the gradient descent step, but as the iterate approaches the true solution, the Gauss-Newton step dominates.

Like other numeric minimization algorithms, LMA is an iterative procedure. To start a minimization, the user has to provide an initial guess for the parameter vector, p, as well as some tuning parameters $\tau, \epsilon_1, \epsilon_2, \epsilon_3$, and k_{max}. Let $Z(p)$ be the vector of calculated errors (Z_t's) for the parameter vector p, and let $J = (J_1, J_2, \ldots, J_N)^T$ be a Jacobian matrix.

A proposed implementation is as follows:

Algorithm 12.2.1

Input: An initial guess for parameters $\boldsymbol{\phi}_0, \boldsymbol{\theta}_0, \mu_0$

Output: The parameters that maximize the likelihood $\boldsymbol{\phi}^*, \boldsymbol{\theta}^*, \mu^*$

$k \leftarrow 0$ ▷ Iteration counter
$v \leftarrow 2$ ▷ The change in the weighting factor.
$(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu) \leftarrow (\boldsymbol{\phi}_0, \boldsymbol{\theta}_0, \mu_0)$ ▷ Initialize parameter vector
Calculate $Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)$ with equation 12.2.6. ▷ Vector of errors
$A \leftarrow J^T J$ ▷ The Gauss-Newton Hessian approximation
$u \leftarrow \tau * \max_i(A_{ii})$ ▷ Weight of the gradient-descent step
$g \leftarrow J^T Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)$ ▷ The gradient descent step.
stop $\leftarrow (\|g\|_\infty \leq \epsilon_1)$ ▷ Termination Variable
while (not stop) and $(k < k_{max})$ **do**
    $k \leftarrow k + 1$
    repeat
        $\delta \leftarrow (A + u \times \mathrm{diag}(A))^{-1} g$ ▷ Calculate step direction
        if $\|\delta\| \leq \epsilon_2 \|(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)\|$ **then** ▷ Change in the parameters is too small to continue.
            stop $\leftarrow$ true
        else
            $(\boldsymbol{\phi}_{new}, \boldsymbol{\theta}_{new}, \mu_{new}) \leftarrow (\boldsymbol{\phi}, \boldsymbol{\theta}, \mu) + \delta$ ▷ Take a trial step in the new direction
            $\rho \leftarrow (\|Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)\|^2 - \|Z(\boldsymbol{\phi}_{new}, \boldsymbol{\theta}_{new}, \mu_{new})\|^2)/(\delta^T(u\delta + g))$ ▷ Calculate improvement of trial step
            if $\rho > 0$ **then** ▷ Trial step was good, proceed to next iteration
                $(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu) \leftarrow (\boldsymbol{\phi}_{new}, \boldsymbol{\theta}_{new}, \mu_{new})$ ▷ Update variables
                Calculate $Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)$ with equation 12.2.6.
                $A \leftarrow J^T J$
                $g \leftarrow J^T Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)$
                stop $\leftarrow (\|g\|_\infty \leq \epsilon_1)$ or $(\|Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)^2\| \leq \epsilon_3)$ ▷ Terminate if we are close to the solution.

$v \leftarrow 2$
$u \rightarrow u * \max(1/3, 1 - (2\rho - 1)^3)$
else ▷ Trial step was bad, change weighting on the gradient decent step
$v \leftarrow 2v$
$u \leftarrow uv$
until (stop) or $(\rho > 0)$
$(\boldsymbol{\phi}^*, \boldsymbol{\theta}^*, \mu^*) \leftarrow (\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)$

Suggested values for the tuning parameters are $\epsilon_1 = \epsilon_2 = \epsilon_3 = 10^{-15}, \tau = 10^{-3}$ and $k_{max} = 100$.

12.2.1.2 Partial Derivatives

The Jacobian matrix $J = (J_1, J_2, \ldots, J_N)^T$ requires the partial derivatives, which are

$$J_t = (J_{t,\phi_1}, \ldots, J_{t,\phi_p}, J_{t,\theta_1}, \ldots, J_{t,\theta_q}, J_{t,\mu})^T \ . \tag{12.2.8}$$

Here the last term is present only when `include_mean` is `True`. The iteration relations for J are

$$J_{t,\phi_i} = \frac{\partial F_t(\phi,\theta)}{\partial \phi_i} = -\frac{\partial Z_t}{\partial \phi_i} = X_{t-i} - \mu + \sum_{j=1}^{q} \theta_j \frac{\partial Z_{t-j}}{\partial \phi_i} = X_{t-i} - \mu - \sum_{j=1}^{q} \theta_j J_{t-j,\phi_i}, \tag{12.2.9}$$

$$J_{t,\theta_i} = \frac{\partial F_t(\phi,\theta)}{\partial \theta_i} = -\frac{\partial Z_t}{\partial \theta_i} = Z_{t-i} + \sum_{j=1}^{q} \theta_j \frac{\partial Z_{t-j}}{\partial \theta_i} = Z_{t-i} - \sum_{j=1}^{q} \theta_j J_{t-j,\theta_i}, \tag{12.2.10}$$

$$J_{t,\mu} = \frac{\partial F_t(\phi,\theta)}{\partial \mu} = -\frac{\partial Z_t}{\partial \mu} = 1 - \sum_{j=1}^{p} \phi_j - \sum_{j=1}^{q} \theta_j \frac{\partial Z_{t-j}}{\partial \mu} = 1 - \sum_{j=1}^{p} \phi_j - \sum_{j=1}^{q} \theta_j J_{t-j,\mu}. \tag{12.2.11}$$

Note that the mean value μ is considered separately in the above formulations. When `include_mean` is set to `False`, μ will be simply set to 0. Otherwise, μ will also be estimated together with $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$. The initial conditions for the above equations are

$$J_{t,\phi_i} = J_{t,\theta_j} = J_{t,\mu} = 0 \quad \text{for } t \le p, \text{ and } i = 1, \ldots, p; j = 1, \ldots, q \ , \tag{12.2.12}$$

because we have fixed Z_t for $t \le p$ to be a constant 0 in the initial condition. Note that J is zero not only for $t \le 0$ but also for $t \le p$.

12.2.2 Estimates of Other Quantities

Finally the variance of the residuals is

$$\sigma^2 = \frac{1}{N-p} \sum_{t=1}^{N} Z_t^2 \ . \tag{12.2.13}$$

The estimate for the maximized log-likelihood is

$$l = -\frac{N}{2} \left[1 + \log(2\pi\sigma^2) \right] \ , \tag{12.2.14}$$

where σ^2 uses the value in Eq. (12.2.13). Actually if you put Eq. (12.2.13) into Eq. (12.2.4), you will get a result slightly different from Eq. (12.2.14). However, Eq. (12.2.14) is what R uses for the method `"CSS"`.

The standard error for coefficient a, where $a = \phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_q, \mu$, is

$$\text{error}_a = \sqrt{(H^{-1})_{aa}} \ . \tag{12.2.15}$$

The Hessian matrix is

$$H_{ab} = \frac{\partial^2}{\partial a \partial b}\left(\frac{1}{2\sigma^2}\sum_{t=1}^{N} Z_t^2\right) = \frac{1}{\sigma^2}\sum_{t=1}^{N}\left(J_{t,a}J_{t,b} - Z_t K_{t,ab}\right) = \frac{1}{\sigma^2}\left(A - \sum_{t=1}^{N} Z_t K_{t,ab}\right) \ , \tag{12.2.16}$$

where $a, b = \phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_q, \mu$, σ^2 is given by Eq. (12.2.13), $A = J^T J$ and

$$K_{t,ab} = \frac{\partial J_{t,a}}{\partial b} = -\frac{\partial^2 Z_t}{\partial a \partial b} \ . \tag{12.2.17}$$

And

$$\begin{aligned}
K_{t,\phi_i\phi_j} &= -\sum_{k=1}^{q}\theta_k K_{t-k,\phi_i\phi_j} = 0 \\
K_{t,\phi_i\theta_j} &= -J_{t-j,\phi_i} - \sum_{k=1}^{q}\theta_k K_{t-k,\phi_i\theta_j} \\
K_{t,\phi_i\mu} &= -1 - \sum_{k=1}^{q}\theta_k K_{t-k,\phi_i\mu} \\
K_{t,\theta_i\phi_j} &= -J_{t-i,\phi_j} - \sum_{k=1}^{q}\theta_k K_{t-k,\theta_i\phi_j} \\
K_{t,\theta_i\theta_j} &= -J_{t-i,\theta_j} - J_{t-j,\theta_i} - \sum_{k=1}^{q}\theta_k K_{t-k,\theta_i\theta_j} \\
K_{t,\theta_i\mu} &= -J_{t-i,\mu} - \sum_{k=1}^{q}\theta_k K_{t-k,\theta_i\mu} \\
K_{t,\mu\phi_j} &= -1 - \sum_{k=1}^{q}\theta_k K_{t-k,\mu\phi_j} \\
K_{t,\mu\theta_j} &= -J_{t-j,\mu} - \sum_{k=1}^{q}\theta_k K_{t-k,\mu\theta_j} \\
K_{t,\mu\mu} &= -\sum_{k=1}^{q}\theta_k K_{t-k,\mu\mu} = 0 \ ,
\end{aligned} \tag{12.2.18}$$

where the initial conditions are

$$K_{t,ab} = 0 \quad \text{for } t \le p, \text{ and } a, b = \phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_q, \mu \ . \tag{12.2.19}$$

According to Eqs. (12.2.18,12.2.19), $K_{t,\phi_i\phi_j}$ and $K_{t,\mu\mu}$ are always 0.

The iteration equations Eq. (12.2.18) are quite complicated. Maybe an easier way to compute K is to numerically differentiate the function

$$f(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu) = \frac{1}{2\sigma^2}\sum_{t=1}^{N} Z_t^2 \ , \tag{12.2.20}$$

where Z_t values are computed using given coefficients of $\boldsymbol{\phi}$, $\boldsymbol{\theta}$ and μ. For example,

$$H_{ab} = \frac{1}{\Delta}\left[\frac{f(a+\Delta/2,b+\Delta/2)-f(a-\Delta/2,b+\Delta/2)}{\Delta} - \frac{f(a+\Delta/2,b-\Delta/2)-f(a-\Delta/2,b-\Delta/2)}{\Delta}\right] \text{ for } a \neq b\ , \tag{12.2.21}$$

$$H_{aa} = \frac{f(a+\Delta)-2f(a)+f(a-\Delta)}{\Delta^2} \tag{12.2.22}$$

where Δ is a small number and the coefficients other than a and b are ignored from the arguments of $f(\cdot)$ for simplicity.

12.2.2.1 Implementation

The key of LMA is to compute Jacobian matrix J in each iteration. In order to compute J, we need to compute Z_t for $t = 1, \ldots, N$ in a recursive manner. The difficulty here is how to leverage the parallel capability of MPP databases. By carefully distributing the dataset to the segment nodes - distribute time servies data by the time range, the recursive computation can be done in parallel via approximation. It's also necessary here to utilize the window function for the recursive computation.

12.2.3 Exact Maximum Likelihood Calculation

12.3 Solving for the optimal model

12.3.1 Auto-Correlation Function

Note that there several common definitions of the auto-correlation function. This implementation uses the normalized form.

The auto-correlation function is a cross-correlation of a function (or time-series) with itself, and is typically used to find periodic behavior in a time-series. For a real, discrete time series, the auto-correlation $R(k)$ for lag k of a time series X with N data points is

$$R_X(k) = \sum_{t=k+1}^{N} \frac{(x_t-\mu)(x_{t-k}-\mu)}{N\sigma^2}. \tag{12.3.1}$$

where σ^2 and μ are the variance and mean of the time series respectively. For this implementation, the range of desired k values will be small ($\approx 10\log(N)$), and the auto-correlation function for the range can be computed naively with equation 12.3.1.

12.3.2 Partial Auto-Correlation Function

The partial auto-correlation function is a conceptually simple extension to the auto-correlation function, but greatly increases the complexity of the calculations. The partial auto-correlation is the correlation for lag k after all correlations from lags $< k$ have been accounted for.

Let

$$R_{(k)} \equiv [R_X(1), R_X(2), \ldots, R_X(k)]^T \tag{12.3.2}$$

and let

$$P_k = \begin{bmatrix} 1 & R_X(1) & \dots & R_X(k-1) \\ R_X(1) & 1 & \dots & R_X(k-2) \\ \vdots & \vdots & \ddots & \vdots \\ R_X(k-1) & R_X(k-2) & \dots & 1 \end{bmatrix} \tag{12.3.3}$$

Then the partial auto-correlation function $\Phi(k)$ for lag k is

$$\Phi(k) = \frac{\det P_k^*}{\det P_k} \tag{12.3.4}$$

where P_k^* is equal to the matrix P_k, except the kth column is replaced with $R_{(k)}$.

12.3.3 Automatic Model Selection

12.4 Seasonal Models

13 Cox Proportional-Hazards

13.1 Introduction

Proportional-Hazard models enable comparison of *survival models*. Survival models are functions describing the probability of an one-item event (prototypically, this event is death) with respect to time. The interval of time before death occurs is the *survival time*. Let T be a random variable representing the survival time, with a cumulative probability function $P(t)$. Informally, $P(t)$ represents the probability that death has happened before time t.

An equivalent formation is the *survival function* $S(t)$, defined as $S(t) \equiv 1 - P(t)$. Informally, this is the probability that death hasn't happened by time t. The *hazard function* $h(t)$ which assesses the instantaneous risk of demise at time t, conditional on survival upto time t.

$$h(t) = \lim_{\Delta t \to 0} \frac{p\left(t \leq T < t + \Delta t | T \geq t\right)}{\Delta t} \tag{13.1.1}$$

$$= \lim_{\Delta t \to 0} \frac{1}{\Delta t} \frac{p(T < t + \Delta t) - p(T < t)}{P(T \geq t)} \tag{13.1.2}$$

The relationship between $h(t)$ and $S(t)$, using $S(t) = 1 - p(T < t)$ is

$$h(t) = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \frac{(S(t + \Delta_t) - S(t))}{-S(t)} \tag{13.1.3}$$

$$= \frac{-S'(t)}{S(t)} \tag{13.1.4}$$

where

$$S'(t) = \lim_{\Delta t \to 0} \frac{S(t + \Delta t) - S(t)}{\Delta t}$$

denotes the derivative of $S(t)$.

In the simplest case, the Cox model assumes that $h(t)$ is

$$h(t) = e^{\alpha(t)} \tag{13.1.5}$$

where $\exp(\alpha(t))$ is the *baseline function*, which depends only on t. However, in many applications, the probability of death may depend on more than just t. Other covariates, such as age or weight, may be important. Let x_i denote the observed value of the ith covariate, then the Cox model is written as

$$h(t) = e^{\alpha(t)} e^{\beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m} = e^{\alpha(t)} e^{\beta^{\mathbf{T}} \mathbf{x}} \tag{13.1.6}$$

where β_i is the coefficient associated with the ith covariate.

Many applications take values from multiple observations, measuring the values of x_i for each observation.

In the *proportional-hazard model*, the hazard functions of two observations j and k are compared. The ratio of the two is

$$\frac{h_j(t)}{h_k(t)} = \frac{e^{\alpha(t)} e^{\beta^{\mathbf{T}} \mathbf{x_j}}}{e^{\alpha(t)} e^{\beta^{\mathbf{T}} \mathbf{x_k}}} = \frac{e^{\beta^{\mathbf{T}} \mathbf{x_j}}}{e^{\beta^{\mathbf{T}} \mathbf{x_k}}} \tag{13.1.7}$$

The critical idea here is that the ratio of the two hazard functions is completely independent of the baseline function. This allows meaningful comparisons between samples without knowing the baseline function, which may be difficult to measure or specify.

13.2 Applications

Generally, applications start with a list of n observations, each with m covariates and a time of death. From this $n \times (m+1)$ matrix, we would like to derive the correlation between the covariates and the hazard function. This amounts to finding the values of β.

The values of β can be estimated with the method of *partial likelihood*. This method begins by sorting the observations by time of death into a list $[t_1, t_2, \ldots, t_n]$ such that $t_i \leq t_j : i < j \ \forall i, j$. For any time t_i, let $R(t_i)$ be the set of observations still alive at time t_i.

Given that there was a death at time t_i and observation k was alive prior to t_i, the probability that the death happened to observation k is

$$\Pr(T_k = t_i | R(t_i)) = \frac{e^{\beta^{\mathbf{T}} \mathbf{x_k}}}{\sum_{j \in R(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}}}. \tag{13.2.1}$$

The *partial likelihood function* can now be generated as the product of conditional probabilities. More formally,

$$\mathcal{L} = \prod_{i=1}^{n} \left(\frac{e^{\beta^{\mathbf{T}} \mathbf{x_i}}}{\sum_{j \in R(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}}} \right). \tag{13.2.2}$$

The log-likelihood form of this equation is

$$L = \sum_{i=1}^{n} \left[\beta^{\mathbf{T}} \mathbf{x_i} - \log \left(\sum_{j \in R(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}} \right) \right]. \tag{13.2.3}$$

An estimation of β can be found by simply maximizing this log-likelihood. To maximize the likelihood, it helps to have the derivative of equation 13.2.3, which is

$$\frac{\partial L}{\partial \beta_k} = \sum_{i=1}^{n} \left(x_{ik} - \frac{\sum_{j \in R(t_i)} x_{jk} e^{\beta^{\mathbf{T}} \mathbf{x_j}}}{\sum_{j \in R(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}}} \right). \tag{13.2.4}$$

It follows that the second derivative is

$$\frac{\partial^2 L}{\partial \beta_k \beta_l} = \sum_{i=1}^{n} \left(\frac{\left(\sum_{j \in R(t_i)} x_{jk} e^{\beta^{\mathbf{T}} \mathbf{x_j}} \right) \left(\sum_{j \in R(t_i)} x_{jl} e^{\beta^{\mathbf{T}} \mathbf{x_j}} \right)}{\left(\sum_{j \in R(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}} \right)^2} - \frac{\sum_{j \in R(t_i)} x_{jk} x_{jl} e^{\beta^{\mathbf{T}} \mathbf{x_j}}}{\sum_{j \in R(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}}} \right). \tag{13.2.5}$$

13.2.1 Incomplete Data

Frequently, not every observation will have an associated time of death. Typically, this arises when the period of observation terminates before the entire population being studied has died. This is known as *censoring* the data. To account for this, an additional indicator variable is introduced δ_i, which is set to 1 if the ith observation has an associated time of death, and 0 otherwise.

Incorporating this indicator variable into equation 13.2.2 gives

$$\mathcal{L} = \prod_{i=1}^{n} \left(\frac{e^{\beta^{\mathbf{T}} \mathbf{x_i}}}{\sum_{j \in R(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}}} \right)^{\delta_i}. \tag{13.2.6}$$

The appropriate changes to the LLH function and its derivatives are trivial.

13.2.2 Partition and aggregation of the data to speed up

In order to speed up the computation, we first partition the data and aggregate each piece of the data into one big row. During the computation, the whole big row is loaded into the memory for processing, which speeds up the computation.

When we partition the data, we want to (1) keep the sorted descending order of the time column, (2) make sure that each piece has approximately the same amount of data so that the work load is even, and (3) do this as fast as possible.

Our solution to to first sample a certain amount of the data, and then compute the approximate break points using the sampled data. The sampled data should be small enough to load into the memory, abd also large enough so that the break points can be computed relatively accurate.

After we have partitioned the data, we aggregate each partition into one big row. The order of the data should be kept during the aggregation.

Then we use the sequential algorithm described in the next to process the new data row by row in the order of the time column. Since each big row contains lots of original rows, and we deal with them all in the memory, this can speed up the computation.

13.2.3 Implementation of Newton's Method

Newton's method is the most common choice for estimating β by minimizing 13.2.2 using the following update rule:

$$\beta_k = \beta_k - \alpha_k \left(\nabla^2 L^{-1} \nabla L \right) \tag{13.2.7}$$

where α_k is a positive scalar denoting the step length in the newton direction $\nabla^2 L^{-1} \nabla L$ determined using the first and second derivative information. We would like to emphasize that the problems we are designing this system for are those with many records and few features i.e. $n \gg m$, thereby keeping the inverse operation on the hessian matrix relatively cheap.

The gradient and Hessian matrix may be hard to parallelize therefore reducing an advantage for large number of observations. To elaborate, consider equations 13.2.4 and 13.2.5 which are sums with independent terms. One might think it is natural to reduce the computational by parallelization. Efficient parallelization may be achieved if each term could be computed independently in parallel by a set of worker tasks and a master task could collect the output from each worker node sum them together. However, this might not work well in this setting. To see why, consider parallelizing equation 13.2.5. Each worker task is given one term in the sum, which looks like

$$\frac{\left(\sum_{j \in R(t_i)} x_{jk} e^{\beta^{\mathbf{T}} \mathbf{x_j}} \right) \left(\sum_{j \in R(t_i)} x_{jl} e^{\beta^{\mathbf{T}} \mathbf{x_j}} \right)}{\left(\sum_{j \in R(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}} \right)^2} - \frac{\sum_{j \in R(t_i)} x_{jk} x_{jl} e^{\beta^{\mathbf{T}} \mathbf{x_j}}}{\sum_{j \in R(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}}}. \tag{13.2.8}$$

Note that the sums in the numerator are summing over all the data points in the data matrix. A similar such issue is encountered while computing the first derivative terms as defined in 13.2.4. However, we note that this sum has a structure that allows it to be computed in linear time (with respect to the number of data points) using the following quantities.

$$H_i = \sum_{j \in R(t_i)} x_j e^{\beta^{\mathbf{T}} \mathbf{x_j}} \tag{13.2.9}$$

$$S_i = \sum_{j \in R(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}} \tag{13.2.10}$$

$$V_i = \sum_{j \in R(t_i)} x_j x_j^T e^{\beta^{\mathbf{T}} \mathbf{x_j}} \tag{13.2.11}$$

Note that H_i is a column vector with m elements ($H_i \in \mathbb{R}^m$), S_i is a scalar and and V_i is an $m \times m$ matrix. We can now write the first derivative of the maximum likelihood estimator, defined in Equation 13.2.4 as

$$\frac{\partial L}{\partial \beta_k} = \sum_{i=1}^{n} \left(x_i - \frac{H_i}{S_i} \right) \tag{13.2.12}$$

while the second derivative, defined in Equation 13.2.5, can be reduced to

$$\frac{\partial^2 L}{\partial \beta_k \beta_l} = \sum_{i=1}^{n} \left(\frac{H_i H_i^T}{S_i^2} - \frac{V_i}{S_i} \right) \tag{13.2.13}$$

Since we assume that the data points are sorted in increasing order i.e $R(t_i) = \{i, i+1 \dots n\}$, we can calculate the above summation as

$$H_i = H_{i+1} + x_i e^{\beta \mathbf{T} \mathbf{x_i}} \tag{13.2.14}$$

$$S_i = S_{i+1} + e^{\beta \mathbf{T} \mathbf{x_i}} \tag{13.2.15}$$

$$V_i = V_{i+1} + \frac{H_i H_i^T}{S_i^2} - \frac{V_i}{S_i}. \tag{13.2.16}$$

With this recurrence relationship, the hessian matrix and the gradient direction can be computed in linear time.

13.3 Stratification Support

A crucial component of the Cox Proportional Hazards model is the proportional hazards assumption: The hazard for a given individual is a fixed proportion of the hazard for any other individual in the same stratum, and the ratio of the hazards is constant across time.

In actual use cases, the proportional hazard assumption may not be satisfied if we use all independent variables as covariates. A stratified Cox regression model may then be useful. It offers a way such that we can choose a subset of independent variables as covariates while are still taking the remaining independent variables into account. The stratified Cox regression is available in both R [32] and Stata [59].

Stratification is used as shorthand for building a Cox model that allows for more than one stratum, and hence, allows for more than one baseline hazard function. Stratification provides two pieces of key, flexible functionality for the end user of Cox models:

i) Allows a categorical variable Z to be appropriately accounted for in the model without estimating its predictive/associated impact on the response variable (i.e. without estimating Z's "coefficient").

ii) Categorical variable Z is predictive/associated with the response variable, but Z may not satisfy the proportional hazards assumption.

To explicitly clarify how stratification differentiates from grouping support:

- Grouping by a categorical column would build a completely separate Cox model for each value of the categorical column, where the baseline hazards for each value of the categorical column would be different and the estimated coefficient values for each explanatory variable would be different for each value of the categorical column.

- Stratifying by a categorical column would build a single common Cox model, where the baseline hazards for each value of the categorical column would be different, but the estimated coefficient values for each explanatory variable would be the same for each value of the stratum.

It is valuable to emphasize that all strata share all coefficients, and that the only difference between strata is the baseline hazard. In other words, coefficients for all non-strata explanatory variables are identical across the different strata.

13.3.1 Estimating A Stratified Cox Model

The parameter estimation is done by maximizing the product of likelihoods, each from a stratum [10].

Given n observations, each with m covariates and each in one of K strata [1], let ST_k denote the set of observations in the k-th stratum.

Because, as an objective function, the sum of log-likelihoods is equivalent to the product of likelihoods, according to Equation 13.2.3, we have the log-likelihood associated with the k-th stratum,

$$L_k = \sum_{i \in ST_k} \left[\beta^{\mathbf{T}} \mathbf{x_i} - \log \left(\sum_{j \in R_k(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}} \right) \right], \tag{13.3.1}$$

where $R_k(t_i)$ the set of observations in ST_k and still alive at time t_i.

Therefore, the objective function of stratified cox regression can be expressed as

$$L^{stratified} = \sum_{k=1}^{K} L_k = \sum_{k=1}^{K} \sum_{i \in ST_k} \left[\beta^{\mathbf{T}} \mathbf{x_i} - \log \left(\sum_{j \in R_k(t_i)} e^{\beta^{\mathbf{T}} \mathbf{x_j}} \right) \right]. \tag{13.3.2}$$

The appropriate changes to gradient, Hessian and censoring are trivial.

13.3.2 When We Need A Stratified Cox Model?

General practice in standard statistical packages (i.e. R, SAS) is to make use of the Schoenfeld residuals to gauge whether or not the proportional hazards assumption has been violated.

The Schoenfeld residuals are centered on zero and should be independent of time if PHA (proportional hazard assumption) is true. Deviations from this, i.e. residuals that exhibit some trend in time, indicate that the PHA is violated.

The Schoenfeld residuals, at the time when a failure or death were to occur, are defined by the difference between the observed and expected covariate values at that time. Also note that the Schoenfeld residuals are zeroes for censored observations.

$$\hat{\boldsymbol{r}}_i = \boldsymbol{x}_i - E[\boldsymbol{x}_i] \ , \text{ only for } \delta_i = 1 \ , \tag{13.3.3}$$

To compute the expected values at time t_i, we use the probability distribution given by Equation 13.2.1.

$$\begin{aligned} E[\boldsymbol{x}_i] &= \sum_{k \in \mathcal{R}(t_i)} \boldsymbol{x}_k p(k \text{ dies at } t_i) \\ &= \frac{\sum_{k \in \mathcal{R}(t_i)} \boldsymbol{x}_k e^{\boldsymbol{\beta} \boldsymbol{x}_k}}{\sum_{j \in \mathcal{R}(t_i)} e^{\boldsymbol{\beta} \boldsymbol{x}_j}}, \end{aligned} \tag{13.3.4}$$

[1] Note that this does not mean that we have K variables other than the m covariates, but K groups classified by strata ID variables.

where the values of $\boldsymbol{\beta}$ are the fitted coefficients, and $\mathcal{R}(t)$ is the set of individuals that are still alive at time t.

13.3.2.1 Scaled Schoenfeld Residuals

Suggested by Grambsch and Therneau [34] and also followed by statistical softwares [17], scaling the Schoenfeld residuals by an estimator of its variance yields greater diagnostic power. The scaled Schoenfeld residuals is

$$\begin{aligned} \hat{\boldsymbol{r}}_i^* &= [\mathit{Var}(\hat{\boldsymbol{r}}_i)]^{-1}\hat{\boldsymbol{r}}_i && (13.3.5)\\ &\approx m\,\mathit{Var}(\hat{\boldsymbol{\beta}})\hat{\boldsymbol{r}}_i, && (13.3.6)\end{aligned}$$

where *Var* denotes a covariance matrix, and m is the number of uncensored survival times. $\hat{\boldsymbol{r}}_i$ is a length-n vector, and $\mathit{Var}(\hat{\boldsymbol{\beta}})$ is a $n \times n$ matrix, where m is the number of uncensored data points and n is the number of features.

13.3.2.2 Transformation of Time

Transformation of the time values often helps the analysis of the correlation between the scaled Schoenfeld residuals and time. Common transformation methods include ranking, computing log, and Kaplan-Meier's method. (We don't fully understand the last one yet.)
[2]

13.3.2.3 p-values

The process for computing the p-values of the correlation in R's `survival` package is given in the following.

Let m be the number of uncensored data points, n be the number of features, $\hat{\boldsymbol{t}}$ be the transformed survival time, which is a length-m vector, $\mathit{Var}(\hat{\boldsymbol{\beta}})$ be the variance matrix for the fitted coefficients which is a $n \times n$ matrix, $\hat{\boldsymbol{R}}$ is the unscaled Schoenfeld residuals, which is a $m \times n$ matrix.

$$\begin{aligned} \boldsymbol{w} &= \hat{\boldsymbol{t}} - \bar{\hat{t}}\ ,\ \text{a length-}m\ \text{vector}\ , && (13.3.7)\\ \boldsymbol{v} &= m\,\boldsymbol{w}^T\hat{\boldsymbol{R}}\mathit{Var}(\hat{\boldsymbol{\beta}})\ ,\ \text{a length-}n\ \text{vector} && (13.3.8)\\ z_i &= \frac{1}{m\,\boldsymbol{w}^T\boldsymbol{w}} \cdot \frac{v_i^2}{\left[\mathit{Var}(\hat{\boldsymbol{\beta}})\right]_{ii}}\ ,\ \text{for } i = 1, \ldots, n\ , && (13.3.9)\\ p_i &= 1 - \chi_1^2(z_i),\ \text{for } i = 1, \ldots, n\ . && (13.3.10)\end{aligned}$$

Here $\bar{\hat{t}}$ is the average of the transformed survival times. z_i is the z-stats for i-th coefficient. p_i is the p-value for i-th coefficient. We need a separate function to compute $\boldsymbol{w}$, but both $\boldsymbol{v}$ and $\boldsymbol{w}^T\boldsymbol{w}$ can be computed in an aggregate function. z_i and p_i can be computed in the final function of that aggregate function. $\chi_1^2(\cdot)$ is the chi-square function with the degree of freedom being 1.

[2] The `cox.zph()` function in R allows different options for transformation of times: `"km"`, `"rank"`, `"log"` and `"identity"`. `"km"` stands for Kaplan-Meier's method. `"rank"` takes the ranks of the times instead of times. `"log"` uses the logarithm of times. And `"identity"` does nothing and directly uses times.[68]

13.3.3 Implementation

We can use the iteration equations Eqs. (13.2.14, 13.2.15, 13.2.16) to compute the residuals and the hessian, which is needed by the variance matrix.

The current implementation uses an ordered aggregate on the data to compute these quantities, which is not in parallel. To enable a distributed solution we can use the "GROUP BY" functionality provided by SQL to enable the independent computation of the log-likelihood function in each distributed segment corresponding to each strata ('transition' function). These values can then be added across the segments ('merge' function), with the gradient for the parameter computed on the final sum ('final' function).

13.3.4 Resolving Ties

In "coxph_train", Breslow method is used to resolve ties, which uses as the partial likelihood [40]

$$L = \sum_{i=1}^{m} \left[\boldsymbol{\beta}^T \boldsymbol{x}_{i+} - d_i \log \left(\sum_{j \in R(t_i)} e^{\boldsymbol{\beta}^T \boldsymbol{x}_j} \right) \right] \tag{13.3.11}$$

$$= \sum_{i=1}^{m} \left[\boldsymbol{\beta}^T \boldsymbol{x}_i - \log \left(\sum_{j \in R(t_i)} e^{\boldsymbol{\beta}^T \boldsymbol{x}_j} \right) \right]_+ \tag{13.3.12}$$

where d_i is the number of observations that have the same t_i, and x_{i+} is the sum of the covariants over all observations that have the same t_i. m is the number of distinct survival times. Here $[\cdot]_+$ means sum over all observations with the same survival time.

13.3.5 Robust Variance Estimators

In MADlib, we implement robust variance estimator devised by Lin and Wei [47]. With our notations above, let

$$\begin{aligned}
\boldsymbol{W}_i &= \delta_i \cdot \left[\boldsymbol{x_i} - \frac{\left(\sum_{l:t_l \geq t_i} e^{\boldsymbol{\beta}^T \boldsymbol{x_l}} \boldsymbol{x_l} \right)}{\sum_{l:t_l \geq t_i} e^{\boldsymbol{\beta}^T \boldsymbol{x_l}}} \right] - \sum_{j:t_j \leq t_i} \left\{ \delta_j \cdot \frac{e^{\boldsymbol{\beta}^T \boldsymbol{x_i}}}{\sum_{k:t_k \geq t_j} e^{\boldsymbol{\beta}^T \boldsymbol{x_k}}} \cdot \left[\boldsymbol{x_i} - \frac{\left(\sum_{k:t_k \geq t_j} e^{\boldsymbol{\beta}^T \boldsymbol{x_k}} \boldsymbol{x_k} \right)}{\sum_{k:t_k \geq t_j} e^{\boldsymbol{\beta}^T \boldsymbol{x_k}}} \right] \right\}. \\
&= \delta_i \cdot \left[\boldsymbol{x_i} - \frac{\boldsymbol{H}_i}{S_i} \right] - \sum_{j:t_j \leq t_i} \left\{ \delta_j \cdot \frac{e^{\boldsymbol{\beta}^T \boldsymbol{x_i}}}{S_j} \cdot \left[\boldsymbol{x_i} - \frac{\boldsymbol{H}_j}{S_j} \right] \right\}. \\
&= \delta_i \cdot \left[\boldsymbol{x_i} - \frac{\boldsymbol{H}_i}{S_i} \right] - \sum_{j:t_j \leq t_i} \delta_j \cdot \frac{e^{\boldsymbol{\beta}^T \boldsymbol{x_i}}}{S_j} \boldsymbol{x_i} + \sum_{j:t_j \leq t_i} \delta_j \cdot \frac{e^{\boldsymbol{\beta}^T \boldsymbol{x_i}}}{S_j} \frac{\boldsymbol{H}_j}{S_j}.
\end{aligned}$$

where

$$\begin{aligned}
H_i &= \sum_{l:t_l \geq t_i} e^{\boldsymbol{\beta}^T \boldsymbol{x_l}} \boldsymbol{x_l} \\
S_i &= \sum_{l:t_l \geq t_i} e^{\boldsymbol{\beta}^T \boldsymbol{x_l}}
\end{aligned}$$

Let

$$\begin{aligned} A_i &= \sum_{j:t_j \le t_i} \frac{\delta_j}{S_j}, \\ \boldsymbol{B}_i &= \sum_{j:t_j \le t_i} \frac{\delta_j \boldsymbol{H}_j}{S_j^2}, \end{aligned}$$

we have

$$\boldsymbol{W}_i = \delta_i \cdot \left[\boldsymbol{x_i} - \frac{\boldsymbol{H}_i}{S_i}\right] - e^{\boldsymbol{\beta}^T \boldsymbol{x_i}} A_i \boldsymbol{x_i} + e^{\boldsymbol{\beta}^T \boldsymbol{x_i}} \boldsymbol{B}_i.$$

And the recursions of A_i and $\boldsymbol{B}_i$ are given if we assume time t_i is sorted ascending order

$$\begin{aligned} A_i &= A_{i-1} + \frac{\delta_i}{S_i} \\ \boldsymbol{B}_i &= \boldsymbol{B}_{i-1} + \frac{\delta_i \boldsymbol{H}_i}{S_i^2} \end{aligned}$$

The meat part of the sandwich estimator is

$$\mathbb{M} = \sum_{i=1}^{n} \boldsymbol{W}_i \boldsymbol{W}_i^T \ . \tag{13.3.13}$$

13.3.6 Clustered Variance Estimation

The data has multiple clusters $m = 1, \ldots, K$, and the meat part is

$$\mathbb{M} = \mathbb{A}^T \mathbb{A} \ , \tag{13.3.14}$$

where the matrix $\mathbb{A}$'s m-th row is given by

$$\mathbb{A}_m = \sum_{i \in G_m} \boldsymbol{W}_i \ . \tag{13.3.15}$$

Here G_m is the set of rows of that belong to the m-th cluster.

With stratification, we take into account of the strata only when we compute $\boldsymbol{H}_i$, S_i, A_i, $\boldsymbol{B}_i$ and $\boldsymbol{W}_i$. The calculations of $\mathbb{A}_m$ and $\mathbb{M}$ only need to group by the clustered variables and the strata variables are irrelavent.

13.4 How to prevent under/over -flow errors ?

A problem that is not mentioned above but appears in the real applications of the CoxPH training model is the under-flow or over-flow errors. We have exponential functions in the computation, and it is very easy to get under-flow or over-flow errors if the coefficients become too small or large at certain steps.

We use the same method as R's "survival" package to deal with the possible under-flow/over-flow errors. This methods contains three parts, which makes the algorithm described above even more complicated:

(1) center and scale the independent variables.

$$x_i \to \frac{x_i - E[x]}{E[|x_i - E[x]|]} \tag{13.4.1}$$

(2) Estimate the maximum possible value of all the coefficients using the coefficients computed from the first iteration.

$$\beta_k^{(max)} = 20 \sqrt{\frac{h_{kk}}{\sum_i \delta_i}} \ , \tag{13.4.2}$$

where $\beta_k^{(max)}$ is the estimate of the possible maximum value of the coefficient β_k, $h_{kk} = \partial^2 L / \partial \beta_k^2$ is the diagonal elements of the Hessian matrix, and $\delta_i = 0, 1$ is the censoring status of the records.

During the computation, whenever the coefficient $|\beta_k| > \beta_k^{(max)}$, we set $\beta_k \to \text{sign}(\beta_k)\beta_k^{(max)}$.

The authors of "survival" package explains in http://stat.ethz.ch/R-manual/R-devel/RHOME/library/survival/d why they use such an estimate for the coefficients:

"We use a cutpoint of $\beta * \text{std}(x) < 23$ where the standard deviation is the average standard deviation of x within a risk set. The rationale is that e^{23} is greater than the current world population, so such a coefficient corresponds to a between subject relative risk that is larger than any imaginable."

In their implementation, they also use $1/h_{kk}$ as an approximation to $\text{std}(x)$. And besides 23, they also use 20 in the estimate.

(3) Although (1) and (2) stabalize the computation, it is still not enough. Step-halving method is used. Whenever the current iteration's log-likelihood is smaller than that of previous iteration, we accept the coefficients as

$$\beta_k = \frac{1}{2}(\beta_k^{new} + \beta_k^{old}) \tag{13.4.3}$$

(4) The stopping threshold is

$$1 - \frac{L_{new}}{L_{old}} < \text{threshold} \ . \tag{13.4.4}$$

13.5 Marginal Effects

See 6.5 for an introduction to marginal effects (all notations below are the same as those defined in 6.5). We implement the default algorithm used by Stata 13.1 for computing the marginal effects. Note that older versions of Stata may use a different default algorithm to compute values different from MADlib.

13.5.1 Basic Formulation

The relative hazard ratio for i-th record is given by

$$h_i = h_0 \exp\left(\boldsymbol{\beta} \cdot \boldsymbol{f}\right) \ , \tag{13.5.1}$$

where h_0 is the baseline and both $\boldsymbol{\beta}$ and $\boldsymbol{f}$ are vectors. Here we use the indices i or j for the data records, and a or b for the indices of covariate terms in $\boldsymbol{\beta} \cdot \boldsymbol{f}(\boldsymbol{x}_i)$. And we will use u or v to denote the indices of $\boldsymbol{x}$.

The value of the baseline h_0 is arbitrary and difficult to compute. Strata ignores the baseline hazard value for the computation of marginal effect. For MADlib, we follow the same principle and ignore the baseline (i.e. set baseline as 1) to compute hazard value.

Thus the marginal effect corresponding to variable x_k is computed as,

$$\begin{aligned} ME_k &= \frac{\partial h_i}{\partial x_k} \\ &= \frac{\partial e^{\boldsymbol{\beta f}}}{\partial x_k} \\ &= e^{\boldsymbol{\beta f}} \boldsymbol{\beta} \frac{\partial f}{\partial x_k}. \end{aligned}$$

Vectorizing the above equation (similar to 6.5.4.2) gives

$$ME = e^{\boldsymbol{\beta f}} J^T \boldsymbol{\beta},$$

where J is defined in 6.5.4.

Censoring status and stratification are irrelavant to the marginal effect calculation and can be ignored.

13.5.2 Categorical variables

For categorical variables, we compute the discrete difference as described in 6.5.2. The discrete difference with respect to x_k is given as

$$\begin{aligned} ME_k &= h^{set} - h^{unset} \\ &= e^{\boldsymbol{\beta f^{set}}} - e^{\boldsymbol{\beta f^{unset}}} \end{aligned}$$

13.5.3 Standard Error

As has already been described in 6.5.5, the method to compute the standard errors is

$$\operatorname{Var}(ME) = \mathbb{S} \operatorname{Var}(\boldsymbol{\beta}) \mathbb{S}^T, \tag{13.5.2}$$

where the matrix $\mathbb{S}$, computed as the partial derivative of the marginal effect over all the coefficients, is a $M \times N$ matrix $S_{mn} = \frac{\partial ME_m}{\partial \beta_n}$

$$\begin{aligned} S_{mn} &= \frac{\partial ME_m}{\partial \beta_n} \\ &= \frac{\partial}{\partial \beta_n} \left(e^{\boldsymbol{\beta f}} \boldsymbol{\beta} \frac{\partial \boldsymbol{f}}{\partial x_m} \right) \\ &= e^{\boldsymbol{\beta f}} \frac{\partial}{\partial \beta_n} \left(\boldsymbol{\beta} \frac{\partial \boldsymbol{f}}{\partial x_m} \right) + \frac{\partial e^{\boldsymbol{\beta f}}}{\partial \beta_n} \boldsymbol{\beta} \frac{\partial \boldsymbol{f}}{\partial x_m} \\ &= e^{\boldsymbol{\beta f}} \frac{\partial f_n}{\partial x_m} + e^{\boldsymbol{\beta f}} \cdot f_n \cdot \boldsymbol{\beta} \frac{\partial \boldsymbol{f}}{\partial x_m} \\ &= e^{\boldsymbol{\beta f}} \left(\frac{\partial f_n}{\partial x_m} + f_n \cdot \boldsymbol{\beta} \frac{\partial \boldsymbol{f}}{\partial x_m} \right). \end{aligned}$$

Vectorizing this equation to express for the complete matrix,

$$\boldsymbol{S} = e^{\boldsymbol{\beta f}} \left(J^T + (J^T \boldsymbol{\beta}) \boldsymbol{f}^T \right).$$

14 Sandwich Estimators

14.1 Introduction

Given a regression setup of n data points, each defined by a feature vector x_i and a category y_i, we assume that y_i is controlled by a k-dimensional parameter vector θ. Generally, we are interested in finding the values of θ that best predict y_i from x_i, with *best* being defined as the values that maximize some likelihood function $L(y, x, \theta)$. The maximization is typically solved using the derivative of the likelihood ψ and the Hessian H. More formally, ψ is defined as

$$\psi(y_i, x_i, \theta) = \frac{\partial l(x_i, y_i, \theta)}{\partial \theta} \tag{14.1.1}$$

and H is defined as

$$H(y, x, \theta) = \frac{\partial^2 L(x, y, \theta)}{\partial \theta^2}. \tag{14.1.2}$$

In addition to the values of θ, we may also be interested in the covariance matrix $S(\theta)$ of θ. This can be expressed in a *sandwich formulation*, of the form

$$S(\theta) = B(\theta)M(\theta)B(\theta). \tag{14.1.3}$$

The $B(\theta)$ matrix is commonly called the *bread*, whereas the $M(\theta)$ matrix is the *meat*.

14.1.1 The Bread

Computing B is relatively straightforward,

$$B(\theta) = n\left(\sum_i^n -H(y_i, x_i, \theta)\right)^{-1} \tag{14.1.4}$$

14.1.2 The Meat

There are several choices for the M matrix, each with different robustness properties. The estimators we are interested in for this implementation are the Huber/White estimator, and the clustered estimator.

In the Huber/White estimator, the matrix M is defined as

$$M_H = \sum_i^n \psi(y_i, x_i, \theta)^T \psi(y_i, x_i, \theta). \tag{14.1.5}$$

15 Generalized Linear Models

Author Liquan Pei

Author Lei Huang

History **v0.1** Initial version

v0.2 Extention to multivariate response and ordinal response case

15.1 Introduction

Linear regression model assumes that the dependent variable Y is equal to a linear combination $\boldsymbol{X}^\top \boldsymbol{\beta}$ and a normally distributed error term

$$Y = \boldsymbol{X}^\top \boldsymbol{\beta} + \epsilon$$

where $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_m)^\top$ is a vector of unknown parameters and $\boldsymbol{X} = (X_1, \ldots, X_m)^\top$ is a vector of independent variables.

In a generalized linear model (GLM), the distribution of dependent variable Y is a member from the *exponential family* and the mean $\mu = \mathbf{E}(Y)$ depends on the independent variables $\boldsymbol{X}$ through

$$\mu = \mathbf{E}(Y) = g^{-1}(\eta) = g^{-1}(\boldsymbol{X}^\top \beta)$$

where $\eta = \boldsymbol{X}^\top \beta$ is the *linear predictor* and g is the *link function*.

In what follows, we denote $G(\eta) = g^{-1}(\eta)$ as the inverse link function.

15.1.1 Exponential Family

A random variable Y is a member from the exponantial family if its probability function or its density function has the form

$$f(y, \theta, \psi) = \exp\left\{\frac{y\theta - b(\theta)}{a(\psi)} + c(y, \psi)\right\}$$

where θ is the canonical parameter. The mean and variance of the exponential family are

- $\mathbf{E}(Y) = \mu = b'(\theta)$
- $\mathbf{Var}(Y) = V(\mu)a(\psi) = b''(\theta)a(\psi)$

15.1.2 Linear Predictor

The linear predictor η incorporates the information about the independent variables into the model. It is related to the expected value of the data through link functions.

15.1.3 Link Function

The link function provides the relationship between η, the linear predictor and μ, the mean of the distribution function. There are many commonly used link functions, and their choice can be somewhat arbitrary. It makes sense to try to match the domain of the link function to the range of the distribution function's mean.

For canonical parameter θ, the canonical link function is the function that expresses θ in terms of $\eta = g(\mu)$. In what follows we treat $\theta = \theta(\eta) = h(g(\mu))$. If we choose h to be an identical function, then $\theta = \eta$ and $\mu = G(\eta) = \mu(\theta)$.

15.2 Parameter Estimation

We estimate unknown parameters β by maximizing the log-likelihood of a GLM. Given the examples $\boldsymbol{Y} = (Y_1, \ldots, Y_n)^\top$ and denote their mean as $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_n)^\top$, the log-likelihood for a GLM is

$$\begin{aligned} l(\boldsymbol{Y}, \boldsymbol{\mu}, \psi) &= \sum_{i=1}^{n} \log f(Y_i, \theta_i, \psi) \\ &= \sum_{i=1}^{n} \left\{ \frac{Y_i \theta_i - b\left(\theta_i\right)}{a\left(\psi\right)} - c(Y_i, \psi) \right\} \end{aligned}$$

where $\theta_i = \theta(\eta_i) = \theta(x_i^\top \beta)$

Note that $a(\psi)$ and $c(Y_i, \psi)$ dose not depend on β, we then maximize

$$\tilde{l}(\boldsymbol{Y}, \boldsymbol{\mu}) = \sum_{i=1}^{n} \{Y_i \theta_i - b(\theta_i)\} \tag{15.2.1}$$

with respect to $\boldsymbol{\beta}$. In what follows, we denote $\boldsymbol{x_i}$ to be the vector of values of independent variables for Y_i.

15.2.1 Iterative Reweighted Least Squares Algorithm

We use iterative reweighted least squares (IRLS) algorithm to find $\boldsymbol{\beta}$ that maximize $\tilde{l}(\boldsymbol{Y}, \boldsymbol{\mu})$. Specifically, we use Fisher scoring algorithm which updates $\boldsymbol{\beta}$ at step $k+1$ using

$$\boldsymbol{\beta}^{k+1} = \boldsymbol{\beta}^k + \left\{ \mathbf{E}[H(\beta^k)] \right\}^{-1} \nabla_{\boldsymbol{\beta}} \tilde{l}(\boldsymbol{\beta}^k)$$

where $\mathbf{E}[H]$ is the mean of Hessian over examples $\boldsymbol{Y}$ and $\nabla_{\boldsymbol{\beta}} \tilde{l}$ is the gradient. For GLM, the gradient is

$$\nabla_{\boldsymbol{\beta}} \tilde{l} = \sum_{i=1}^{n} \{Y_i - b'(\theta_i)\} \nabla_{\boldsymbol{\beta}} \theta_i$$

Note that $\mu_i = G(\eta_i) = G(\boldsymbol{x_i}^\top \boldsymbol{\beta}) = b'(\theta_i)$, we have

$$\nabla_{\boldsymbol{\beta}} \theta_i = \frac{G'(\eta_i)}{V(\mu_i)} \boldsymbol{x_i}$$

then

$$\nabla_{\boldsymbol{\beta}} \tilde{l} = \sum_{i=1}^{n} \{Y_i - \mu_i\} \frac{G'(\eta_i)}{V(\mu_i)} \boldsymbol{x_i}$$

The Hessian is

$$\begin{aligned} H(\beta) &= \sum_{i=1}^{n} \left\{ -b''(\theta_i) \nabla_{\boldsymbol{\beta}} \theta_i \nabla_{\boldsymbol{\beta}} \theta_i{}^{\top} - \{Y_i - b'(\theta_i)\} \nabla^2_{\boldsymbol{\beta}} \theta_i \right\} \\ &= \sum_{i=1}^{n} \left\{ \frac{G'(\eta_i)^2}{V(\mu_i)} - \{Y_i - \mu_i\} \nabla^2_{\boldsymbol{\beta}} \theta_i \right\} \boldsymbol{x_i} \boldsymbol{x_i}{}^{\top} \end{aligned}$$

Note that $\mathbf{E}[Y_i] = \mu_i$, we have

$$\mathbf{E}[H(\beta)] = \sum_{i=1}^{n} \left\{ \frac{G'(\eta_i)^2}{V(\mu_i)} \right\} \boldsymbol{x_i} \boldsymbol{x_i}{}^{\top}$$

Define the weight matrix

$$\boldsymbol{W} = \operatorname{diag}\left(\frac{G'(\eta_1)^2}{V(\mu_1)}, \ldots, \frac{G'(\eta_n)^2}{V(\mu_n)} \right)$$

and define

$$\tilde{\boldsymbol{Y}} = \left(\frac{Y_1 - \mu_1}{G'(\eta_1)}, \ldots, \frac{Y_n - \mu_n}{G'(\eta_n)} \right)^{\top}$$

and the design matrix

$$\boldsymbol{X}^{\top} = (\boldsymbol{x_1}, \ldots, \boldsymbol{x_n})$$

Finally, the update rule for GLM is

$$\begin{aligned} \boldsymbol{\beta}^{k+1} &= \boldsymbol{\beta}^k + (\boldsymbol{X}^{\top} \boldsymbol{W} \boldsymbol{X})^{-1} \boldsymbol{X}^{\top} \boldsymbol{W} \tilde{\boldsymbol{Y}} \\ &= (\boldsymbol{X}^{\top} \boldsymbol{W} \boldsymbol{X})^{-1} \boldsymbol{X}^{\top} \boldsymbol{W} \boldsymbol{Z} \end{aligned}$$

where $\boldsymbol{Z} = (Z_1, \ldots, Z_n)$ is a vector of *adjusted dependent variables*

$$Z_i = \boldsymbol{x_i}{}^{\top} \boldsymbol{\beta}^k + \frac{Y_i - \mu_i}{G'(\eta_i)}$$

Note that each step is the result of a weighted least square regression on the adjusted variables Z_i on x_i and this the reason that this algorithm is called iterative reweighted least squares.

The IRLS algorithm for GLM is as follows

Algorithm 15.2.1

Input: $\boldsymbol{X}$, $\boldsymbol{Y}$, inverse link function $G(\eta)$, dispersion function $V(\mu)$ and initial values $\boldsymbol{\beta}^0$

Output: $\boldsymbol{\beta}$ that maximize $\tilde{l}(\boldsymbol{Y}, \boldsymbol{\mu})$

1: $k \leftarrow 0$
2: **repeat**
3: Compute $\boldsymbol{\mu}$ where $\mu_i = G(\eta_i) = G(\boldsymbol{x_i}{}^{\top} \boldsymbol{\beta}^k)$
4: Compute $\boldsymbol{Z}$ where $Z_i = \boldsymbol{x_i}{}^{\top} \boldsymbol{\beta}^k + \frac{Y_i - \mu_i}{G'(\eta_i)}$
5: Compute $\boldsymbol{W}$ where $W_{ii} = \frac{G'(\eta_i)^2}{V(\mu_i)}$
6: $\boldsymbol{\beta}^{k+1} = (\boldsymbol{X}^{\top} \boldsymbol{W} \boldsymbol{X})^{-1} \boldsymbol{X}^{\top} \boldsymbol{W} \boldsymbol{Z}$
7: **until** $\boldsymbol{\beta}^{k+1}$ converges

15.2.2 Functions for contructing the exponential families

Table 15.1 [31] provides functions $a(), b()$ and $c()$ to contruction the exponential families,

Family	$a(\psi)$	$b(\theta)$	(y, ψ)
Gaussian	ψ	$\theta^2/2$	$-\frac{1}{2}\left[y^2/\psi + \log_e(2\pi\psi)\right]$
Binomial	1/n	$\log_e(1+e^\theta)$	$\log_e C_{ny}^n$
Poisson	1	e^θ	$-log_e y!$
Gamma	ψ	$-\log_e(-\theta)$	$\psi^{-1}\log_e(y/\psi) - \log_e y - \log_e \Gamma(\psi^{-1})$
Inverse-Gaussian	$-\psi$	$\sqrt{2\theta}$	$-\frac{1}{2}\left[\log_e(\pi\psi y^3)^3 + 1/(\psi y)\right]$

Table 15.1: Functions for constructing the exponential families

15.2.3 Multivariate response family

Instead of a single scalar number, some response variable follows a multivariate distribution (i.e. the response variable is a vector instead of a scalar number). One example is multinomial GLM where the response variable is an indictor vector containing zeros and ones to dummy code the corresponding categories. For illustration purpose, in this section, we are discussing multinomial GLM. However, other distributions can be easily extended.

Let J denote the number of categories, $y_i = (y_{i1}, y_{i2}, ..., y_{i(J-1)})^T$ be the indicator vector for ith subject where each y_{ij} is the binary indicator whether subject i is in categories j, μ_{ij} will be the probabilty subject i is in category j. Therefore, we can have the log likelihood as below,

$$\begin{aligned} l &= \sum_{i=1}^{I}\left(\sum_{j=1}^{J-1} y_{ij}\log\frac{\mu_{ij}}{1-\sum_{j=1}^{J}\mu_{ij}} + \log(1-\sum_{j=1}^{J-1})\right) \\ &= \sum_{i=1}^{I}\left(\sum_{j=1}^{J-1} y_{ij}\theta_{ij} - \log(1+\sum_{j=1}^{J-1}\exp\theta_{ij})\right) \end{aligned}$$

Define $b(\theta_i) = \log(1+\sum_{j=1}^{J-1}\exp\theta_{ij})$, then it can be showed $\triangledown b(\theta_i) = \mu_i$ and

$$\triangledown\triangledown^T b(\theta_i) = \begin{pmatrix} \mu_{i1}(1-\mu_{i1}) & -\mu_{i1}\mu_{i2} & ... & -\mu_{i1}\mu_{i(J-1)} \\ -\mu_{i2}\mu_{i1} & \mu_{i2}(1-\mu_{i2}) & ... & -\mu_{i2}\mu_{i(J-1)} \\ \vdots & \vdots & \vdots & \vdots \\ -\mu_{i(J-1)}\mu_{i1} & -\mu_{i(J-1)}\mu_{i2} & ... & \mu_{i(J-1)}\mu_{i(J-1)} \end{pmatrix}$$

We set $V = \triangledown\triangledown^T b(\theta_i)$

Let $\eta_{ij} = g_j(\mu_i)$ and $g() = (g_1(), g_2(), ..., g_{J-1}())^T$ be the link map, which is $\Re^{J-1}$ to $\Re^{J-1}$. Also we have $\mu_i = G(\eta_i)$ be its inverse map. We define the derivative of G to be $G' = \left(\frac{\partial\mu_i}{\partial\eta_{i1}}, \frac{\partial\mu_i}{\partial\eta_{i2}}, ..., \frac{\partial\mu_i}{\partial\eta_{i(J-1)}}\right)$. For example, in multinomial logistic regression, $\eta_{ij} = \theta_{ij}$, then $G' = V$.

Denote the coefficient to be β_{kj} where k stands for the kth predictor and j stands for the jth category. Then we have

$$\begin{aligned} \frac{\partial l_i}{\partial\beta_{kj}} &= (y_i - \triangledown b(\theta_i))^T \frac{\partial\theta_i}{\partial\mu_i}\frac{\partial\mu_i}{\partial\eta_{ij}}\frac{\partial\eta_{ij}}{\partial\beta_{kj}} \\ &= (y_i-\mu_i)^T V^{-1} G_j' x_{ik} \end{aligned}$$

where G'_j is the jth column of G'.

$$\begin{aligned}\frac{\partial^2 l_i}{\partial\beta_{kj}\partial\beta_{lh}} &= -x_{il}(G'_h)^T V^{-1}\nabla\nabla^T b(\theta_i)V^{-1}G'_j x_{ik}\\ &= -x_{il}(G'_h)^T V^{-1}G'_j x_{ik}\end{aligned}$$

As a entire vector β,

$$\begin{aligned}\nabla_\beta l_i &= \Big((y_i-\mu_i)^T V^{-1}G'(\mu_i)\Big)^T \otimes X_i\\ E\left[\nabla\nabla^T_\beta l_i\right] &= -\Big([G'(\mu_i)]^T V^{-1}[G'(\mu_i)]\Big)\otimes(X_i X_i^T)\end{aligned}$$

where $X_i = (x_{i1}, x_{i2}, ..., x_{ip})^T$.

Finally, we can use the below update equation for Newton-Raphson method,

$$\beta^{k+1} = \beta^k + \left\{\sum_{i=1}^{I}\Big([G'(\mu_i)]^T V^{-1}[G'(\mu_i)]\Big)\otimes(X_i X_i^T)\right\}^{-1}\sum_{i=1}^{I}\left\{\Big((y_i-\mu_i)^T V^{-1}G'(\mu_i)\Big)^T\otimes X_i\right\}$$

15.2.4 Ordinal logistic regression

In statistics, the ordered logit model (also ordered logistic regression or proportional odds model), is a regression model for ordinal dependent variables. For example, if one question on a survey is to be answered by a choice among "poor", "fair", "good", "very good", and "excellent", and the purpose of the analysis is to see how well that response can be predicted by the responses to other questions, some of which may be quantitative, then ordered logistic regression may be used. It can be thought of as an extension of the logistic regression model that applies to dichotomous dependent variables, allowing for more than two (ordered) response categories.

The model we implement here only applies to data that meet the proportional odds assumption, the meaning of which can be exemplified as follows.

$$\log\left(\frac{\Pr(Y_i\le j)}{1-\Pr(Y_i\le j)}\right) = \alpha_j - \beta_1 x_{i1} - \beta_2 x_{i2} - ... - \beta_p x_{ip}$$

where $\Pr(Y_i \le j)$ is the cumulative probability that the ith subject belongs to the first jth categories; α_j is category-specific intercept and β_k is feature-specific coefficient. Using the notation in the subsection 15.2.3, the link function is

$$g\left([\mu_1,\mu_2,...,\mu_{J-1}]^T\right) = \left[\log\left(\frac{\mu_1}{1-\mu_1}\right), \log\left(\frac{\mu_1+\mu_2}{1-\mu_1-\mu_2}\right), ..., \log\left(\frac{\mu_1+\mu_2+...+\mu_{J-1}}{1-\mu_1-\mu_2-...-\mu_{J-1}}\right)\right]^T$$

Then the inverse of link function G is,

$$\begin{aligned}G\left([\eta_1,\eta_2,...,\eta_{J-1}]^T\right) = \Bigg[&\frac{\exp(\eta_1)}{1+\exp(\eta_1)}, \frac{\exp(\eta_2)}{1+\exp(\eta_2)} - \frac{\exp(\eta_1)}{1+\exp(\eta_1)},\\ &..., \frac{\exp(\eta_{J-1})}{1+\exp(\eta_{J-1})} - \frac{\exp(\eta_{J-2})}{1+\exp(\eta_{J-2})}\Bigg]^T\end{aligned}$$

Its derivative matrix G' is

$$G' = \left[\frac{\partial\mu}{\partial\eta_1}, \frac{\partial\mu}{\partial\eta_2}, ..., \frac{\partial\mu}{\partial\eta_{J-1}}\right]$$
$$= \begin{bmatrix} \frac{\exp(\eta_1)}{(1+\exp(\eta_1))^2} & 0 & 0 & ... & 0 \\ -\frac{\exp(\eta_1)}{(1+\exp(\eta_1))^2} & \frac{\exp(\eta_2)}{(1+\exp(\eta_2))^2} & 0 & ... & 0 \\ 0 & -\frac{\exp(\eta_2)}{(1+\exp(\eta_2))^2} & \frac{\exp(\eta_3)}{(1+\exp(\eta_3))^2} & ... & 0 \\ \vdots & \vdots & \vdots & ... & \vdots \\ 0 & 0 & 0 & ... & \frac{\exp(\eta_{J-1})}{(1+\exp(\eta_{J-1}))^2} \end{bmatrix}$$

Define $h_i = \sum_{j=1}^{J-1} \frac{\partial\mu_i}{\partial\eta_{ij}}$, then as a entire coefficient vector $\gamma = [\alpha_1, \alpha_2, ..., \alpha_{J-1}, \beta_1, \beta_2, ..., \beta_p]^T$,

$$\nabla_\gamma l_i = \begin{bmatrix} \left\{(y_i - \mu_i)^T V^{-1} G_i'\right\}^T \\ -(y_i - \mu_i)^T V^{-1} h_i X_i \end{bmatrix}$$
$$E\left[\nabla\nabla_\gamma^T l_i\right] = -\begin{bmatrix} (G_i')^T V^{-1} G_i' & -(G_i')^T V^{-1} h_i X_i^T \\ -X_i h_i^T V^{-1} G_i & h_i^T V^{-1} h_i X_i X_i^T \end{bmatrix}$$

where $X_i = (x_{i1}, x_{i2}, ..., x_{ip})^T$, $G_i' = G'(\eta_{i1}, \eta_{i2}, ..., \eta_{iJ-1})$. We can then implement Newton-Raphson method using above gradient and Hessian matrix.

15.2.5 Ordinal probit model

In the ordinal probit model, instead of logistic link function used in ordinal logistic regression, the probit function $\Phi(x)$ is used. Therefore the model becomes,

$$\Phi^{-1}\left(\Pr(Y_i \le j)\right) = \alpha_j - \beta_1 x_{i1} - \beta_2 x_{i2} - ... - \beta_p x_{ip}$$

and the link function is

$$g\left([\mu_1, \mu_2, ..., \mu_{J-1}]^T\right) = \left[\Phi^{-1}(\mu_1), \Phi^{-1}(\mu_1 + \mu_2), ..., \Phi^{-1}(\mu_1 + \mu_2 + ... + \mu_{J-1})\right]^T$$

Then the inverse of link function G is,

$$G\left([\eta_1, \eta_2, ..., \eta_{J-1}]^T\right) = [\Phi(\eta_1), \Phi(\eta_2) - \Phi(\eta_1), ..., \Phi(\eta_{J-1}) - \Phi(\eta_{J-2})]^T$$

Its derivative matrix G' is

$$G' = \left[\frac{\partial\mu}{\partial\eta_1}, \frac{\partial\mu}{\partial\eta_2}, ..., \frac{\partial\mu}{\partial\eta_{J-1}}\right]$$
$$= \begin{bmatrix} \phi(\eta_1) & 0 & 0 & ... & 0 \\ -\phi(\eta_1) & \phi(\eta_2) & 0 & ... & 0 \\ 0 & -\phi(\eta_2) & \phi(\eta_3) & ... & 0 \\ \vdots & \vdots & \vdots & ... & \vdots \\ 0 & 0 & 0 & ... & \phi(\eta_{J-1}) \end{bmatrix}$$

where $\Phi(x)$ and $\phi(x)$ are the cumulative probability and density probability of standard normal distribution.

16 Decision Trees: Classification and Regression

Authors Rahul Iyer and Liquan Pei

History **v0.2** Parallelism

v0.1 Initial version: introduction, theory, and interface

16.1 Introduction

Notes, examples and figures in this section are borrowed from [38] and [43].

Tree-based methods for regression and classification involve stratifying or segmenting the predictor space into a number of simple regions. In order to make a prediction for a given observation, we typically use the mean or the mode of the training observations in the region to which it belongs. Since the set of splitting rules used to segment the predictor space can be summarized in a tree, these types of approaches are known as decision tree methods. Tree-based methods are simple and useful for interpretation. However, they typically are not competitive with the best supervised learning approaches, in terms of prediction accuracy. The results from multiple decision tree are usually combined to yield a single consensus prediction often resulting in dramatic improvements in prediction accuracy, at the expense of some loss in interpretation.

16.1.1 Basics of Decision Trees

In order to motivate tree methods, we begin with a simple example.

Let's consider a regression problem with continuous response Y and inputs X_1 and X_2, each taking values in the unit interval. The top left panel of Figure 16.1 shows a partition of the feature space by lines that are parallel to the coordinate axes. In each partition element we can model Y with a different constant. However, there is a problem: although each partitioning line has a simple description like $X_1 = c$, some of the resulting regions are complicated to describe.

To simplify matters, we restrict attention to recursive binary partitions like that in the top right panel of Figure 16.1. We first split the space into two regions, and model the response by the mean of Y in each region. We choose the variable and split-point to achieve the best fit. Then one or both of these regions are split into two more regions, and this process is continued, until some stopping rule is applied. For example, in the top right panel of Figure 16.1, we first split at $X_1 = t1$. Then the region $X1 \leq t1$ is split at $X2 = t2$ and the region $X1 > t1$ is split at $X1 = t3$. Finally,the region $X1 > t3$ is split at $X2 = t4$. The result of this process is a partition into the five regions $R_1, R_2, \ldots, R_5$ shown in the figure. The corresponding regression model predicts Y with a constant c_m in region R_m, that is,

$$f(X) = \sum_{m=1}^{5} c_m I(X_1, X_2) \in R_m.$$

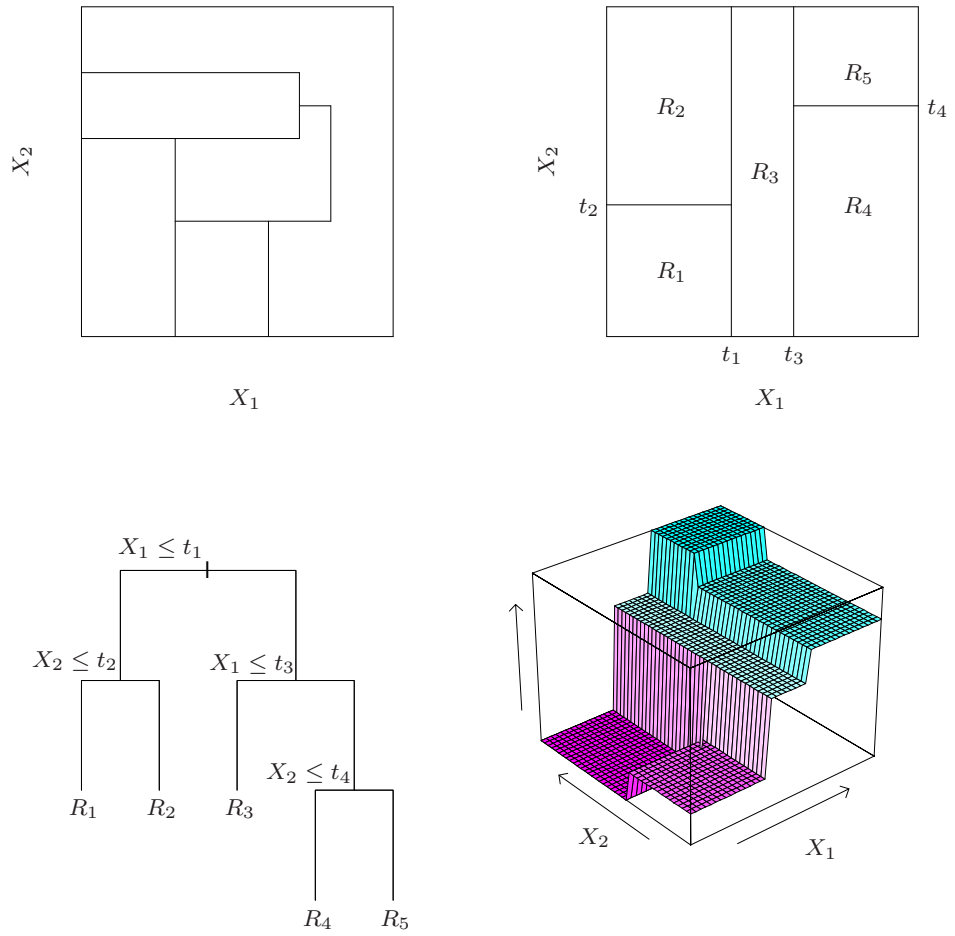

Figure 16.1: (Figure from [38].) Top right panel shows a partition of a two-dimensional feature space by recursive binary splitting, applied to some arbitrary data. Top left panel shows a general partition that cannot be obtained from recursive binary splitting. Bottom left panel shows the tree corresponding to the partition in the top right panel, and a perspective plot of the prediction surface appears in the bottom right panel.

This same model can be represented by the binary tree in the bottom left panel of Figure 16.1. The full dataset sits at the top of the tree. Observations satisfying the condition at each junction are assigned to the left branch, and the others to the right branch. The terminal nodes or leaves of the tree correspond to the regions $R_1, R_2, \ldots, R_5$. The bottom right panel is a perspective plot of the regression surface from this model.

Prediction via Stratification of the Feature Space: We now discuss the process of building a regression tree. Roughly speaking, there are two steps.

i) We divide the predictor space - that is, the set of possible values for $X1, X2, \ldots, Xp$ into J distinct and non-overlapping regions, $R_1, R_2, \ldots, R_J$.

ii) For every observation that falls into the region R_j, we make the same prediction, which is simply the mean of the response values for the training observations in R_j.

Advantages of using tree-based models:

i) Trees are very easy to explain to people.

ii) Some believe that decision trees more closely mirror human decision-making.

iii) Trees can be displayed graphically, and are easily interpreted even by a non-expert.

iv) Trees can easily handle qualitative predictors without the need to create dummy variables.

16.1.2 Trees Versus Linear Models

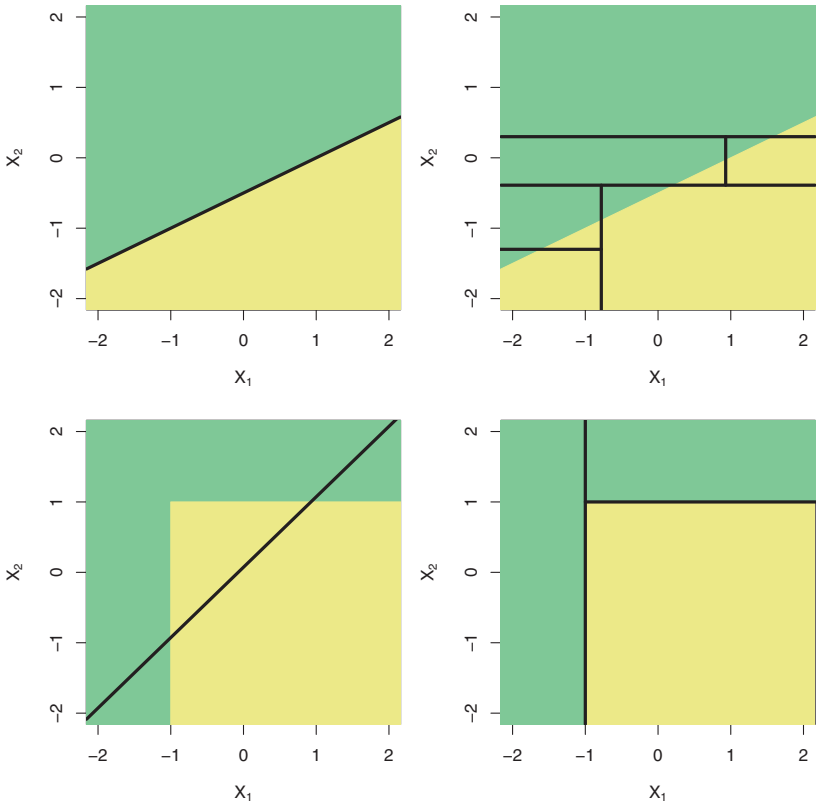

Figure 16.2: (Figure from [43].) Top Row: A two-dimensional classification example in which the true decision boundary is linear, and is indicated by the shaded regions. A classical approach that assumes a linear boundary (left) will outperform a decision tree that performs splits parallel to the axes (right). Bottom Row: Here the true decision boundary is non-linear. Here a linear model is unable to capture the true decision boundary (left), whereas a decision tree is successful (right).

Regression and classification trees have a very different flavor from the more classical approaches for regression presented in 6. In particular, linear regression assumes a model of the form

$$f(X) = \beta_0 + \sum_{j=1}^{p} X_j \beta_j,$$

whereas regression trees assume a model of the form

$$f(X) = \sum_{m=1}^{M} c_m \cdot 1_{X \in R_m}$$

where $R_1, \ldots, R_M$ represent a partition of feature space. The efficacy of the two models depend on the problem at hand, as illustrated by Figure 16.2

16.2 Interface

The interface of the decision tree functionality includes two functions, `tree_train` and `tree_predict`, described below.

16.2.1 Training

```
SELECT tree_train(
        training_table_name,
        output_table_name,
        id_col_name,
        dependent_variable,
        list_of_features,
        list_of_features_to_exclude,
        split_criterion,
        grouping_cols,
        weights,
        max_depth,
        min_split,
        min_bucket,
        surrogate_params,   -- not implemented
        pruning_params,
        verbose
)
```

Arguments:

- *training_table_name*: Name of the table containing data.
- *output_table_name*: Name of the table to output the model.
- *id_col_name*: Name of column containing the id information in training data.
- *dependent_variable*: Name of the column that contains the output for training. Boolean, integer and text are considered classification outputs, while float values are considered regression outputs.
- *list_of_features*: List of column names (comma-separated string) to use as predictors. Can also be a '*' implying all columns are to be used as predictors (except the ones included in the next argument). Boolean, integer, and text columns are considered categorical columns.

- *list_of_features_to_exclude*: OPTIONAL. List of column names (comma-separated string) to exlude from the predictors list.
- *split_criterion*: OPTIONAL (Default = 'gini'). Various options to compute the feature to split a node. Available options are 'gini', 'cross-entropy', and 'misclassification' for classification. For regression tree, the input for argument is ignored and the split_criterion is always set to 'mse' (mean-squared error).
- *grouping_cols*: OPTIONAL. List of column names (comma-separated string) to group the data by. This will lead to creating multiple decision trees, one for each group.
- *weights*: OPTIONAL. Column name containing weights for each observation.
- *max_depth*: OPTIONAL (Default = 10). Set the maximum depth of any node of the final tree, with the root node counted as depth 0.
- *min_split*: OPTIONAL (Default = 20). Minimum number of observations that must exist in a node for a split to be attempted.
- *min_bucket*: OPTIONAL (Default = minsplit/3). Minimum number of observations in any terminal node. If only one of minbucket or minsplit is specified, minsplit is set to minbucket*3 or minbucket to minsplit/3, as appropriate.
- *n_bins*: OPTIONAL (Default = 100) Number of bins to use during binning. Continuous-valued features are binned into discrete bins (per the quantile values) to compute split boundaries. This global parameter is used to compute the resolution of the bins. Higher number of bins will lead to higher processing time.
- *surrogate_params*: (Not implemented) OPTIONAL (String with multiple key-value parameters, default = NULL)
  i) *max_surrogate*: OPTIONAL (Default = 0) The number of surrogate splits retained in the output. If this is set to zero the compute time will be reduced by half.
- *pruning_params*: OPTIONAL (String with multiple key-value parameters, default = NULL)
  i) *cp*: OPTIONAL (Default = 0) A complexity parameter that determines that a split is attempted only if it decreases the overall lack of fit by a factor of 'cp'.
  ii) *n_folds*: OPTIONAL (Default = 0; no cross-validation) Number of cross-validation folds
- *verbose*: OPTIONAL (Default = False) Prints status information on the splits performed and any other information useful for debugging.

16.2.2 Prediction

```
SELECT tree_predict(
        tree_model,
        new_data_table,
        output_table,
        type
)
```

Arguments:

- *tree_model*: Name of the table containing the decision tree model.
- *new_data_table*: Name of table containing prediction data.
- *output_table*: Name of table to output prediction results.
- *type*: OPTIONAL (Default = 'response'). For regression trees, 'response', implies output is the predicted value. For classification trees, this can be 'response', giving the classification prediction as output, or 'prob', giving the class probabilities as output (for two classes, only a single probability value is output that corresponds to the first class when the two classes are sorted by name; in case of more than two classes, an array of class probabilities (a probability of each class) is output).

16.3 CART

CART stands for Classification and Regression Trees ([13]). It is characterized by the fact that it constructs binary trees, namely each internal node has exactly two outgoing edges. The splits can be selected using any of the impurity metrics criteria and the obtained tree is pruned by cost-complexity pruning.

16.3.1 Impurity metrics

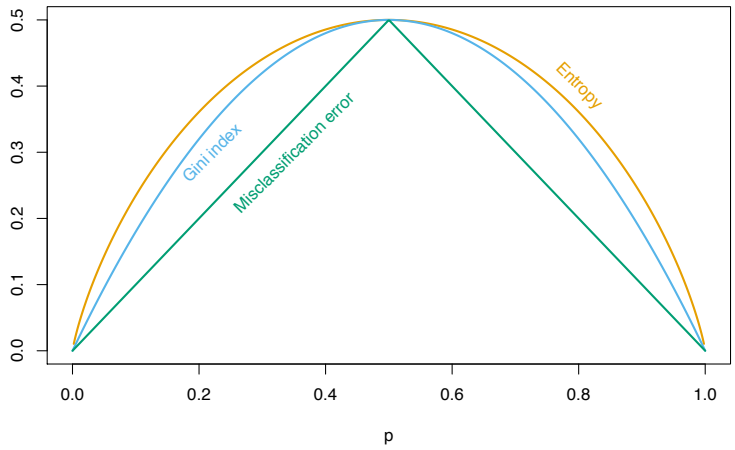

Figure 16.3: (Figure obtained from [38]) Node impurity measures for two-class classification, as a function of the proportion p in class 2. Cross-entropy has been scaled to pass through (0.5, 0.5).

The measures developed for selecting the best split are often based on the degree of impurity of the child nodes. The smaller the degree of impurity, the more skewed the class distribution. For example, a node with class distribution $(0, 1)$ has zero impurity, whereas a node with uniform class distribution $(0.5, 0.5)$ has the highest impurity. Examples of impurity measures include

- Cross-entropy: $-\sum_{i=1}^{C} p(i|A) \log_2 p(i|A)$

- Gini: $1 - \sum_{i=1}^{C} (p(i|A))^2$
- Classification error: $1 - \max_{i \in 1 \ldots C} p(i|A)$,

where C is the number of classes and $p(i|A)$ denotes the proportion of records belonging to class i at a given node A.

Figure 16.3 compares the values of the impurity measures for binary classification problems. p refers to the fraction of records that belong to one of the two classes. Observe that all three measures attain their maximum value when the class distribution is uniform (i.e., when p = 0.5). The minimum values for the measures are attained when all the records belong to the same class (i.e., when p equals 0 or 1).

A split for a node is computed by how much 'impurity' is reduced by that split. For example, if we use the Gini index for selecting the split, we obtain the best binary split (D_1 and D_2) that maximizes the Gini gain as,

$$G_{\text{gini}}(D; D_1, D_2) = g(D) - \left(\frac{|D_1|}{|D|} g(D_1) + \frac{|D_2|}{|D|} g(D_2) \right),$$

where $g(D)$ is the Gini impurity metric.

For ordered, continuous attributes, we create splits of the form $X_i < v$, where X_i is an attribute and v is a value in the domain of X_i, allowing possibly infinite such splits. To reduce the computational load, we compute quantile boundaries of the attribute and test for splits on these boundary values.

For categorical attributes, we create splits of the form $X_i \in \{v_1, v_2, v_3, \ldots\}$, where $\{v_1, v_2, v_3, \ldots\}$ is a subset of all possible values of the categorical attribute. For ordered, categorical attributes (integer input), the split points are obtained by sorting the categorical values and each increasing subsequence is then used as a possible subset for the split. For unordered, categorical attribute, the values are sorted by the entropy of the target variable. In the case of binary classification, this implies that values are sorted by the proportion of labels of the primary class. For multinomial classification, we sort by the entropy of each label i.e. we compute $-\sum_{i=1}^{C} p(i|X_i = v) \log_2 p(i|X_i = v)$ for each value (v) of the attribute (X_i) and sort in increasing order. The splits are then obtained by evaluating each increasing subsequence of this sorted list.

16.3.2 Stopping Criteria

The growing phase continues until a stopping criterion is triggered. The following conditions are some of the stopping rules used:

- All instances in the training set belong to a single value of target variable.
- The maximum tree depth has been reached (*max_depth*).
- The number of cases in the terminal node is less than the minimum number of cases for parent nodes (*min_split*).
- If the node were split, the number of cases in one or more child nodes would be less than the minimum number of cases for child nodes (*min_bucket*).

16.3.3 Missing data

(Note: Not implemented in the initial version.)

Suppose our data has some missing predictor values in some or all of the variables. We might discard any observation with some missing values, but this could lead to serious depletion of the training set. Alternatively we might try to fill in (impute) the missing values, with say the mean of that predictor over the nonmissing observations. For tree-based models, there are two better approaches. The first is applicable to categorical predictors: we simply make a new category for "missing". From this we might discover that observations with missing values for some measurement behave differently than those with nonmissing values. The second more general approach is the construction of surrogate variables. When considering a predictor for a split, we use only the observations for which that predictor is not missing. Having chosen the best (primary) predictor and split point, we form a list of surrogate predictors and split points. The first surrogate is the predictor and corresponding split point that best mimics the split of the training data achieved by the primary split. The second surrogate is the predictor and corresponding split point that does second best, and so on. When sending observations down the tree either in the training phase or during prediction, we use the surrogate splits in order, if the primary splitting predictor is missing. Surrogate splits exploit correlations between predictors to try and alleviate the effect of missing data. The higher the correlation between the missing predictor and the other predictors, the smaller the loss of information due to the missing value.

The resemblance between two binary splits over sample S is formally defined as:

$$sim(a_i, dom_1(a_i), dom_2(a_i), a_j, dom_1(a_j), dom_2(a_j), S) =$$
$$|\sigma_{a_i \in dom_1(a_i) and a_j \in dom_1(a_j)} S| + |\sigma_{a_i \in dom_2(a_i) and a_j \in dom_2(a_j)} S|$$

16.3.3.1 Implementation considerations

For implementing surrogates the following should be considered:

i) Training surrogates: There are two options to consider for training of surrogates for each split:
   a) Compute surrogates after the whole tree has trained. Advantage: the increase in training time will be equivalent to increasing number of iterations by one. Disadvantage: we won't have surrogates for a level while training the sub-levels. Hence, all rows with NULL will be ignored during the training.
   b) Compute surrogates splits for a node while that node is being trained. Advantage: during training, we have surrogate splits for all nodes above the current trained level, implying that features with NULL rows are ignored only while training nodes that use that feature - all nodes below will still get to use that row. Disadvantage: the training doubles, since every iteration will have to first train the nodes in level, and then compute the surrogates for that node.

ii) Storage for surrogates: We store an array of surrogate variables and their corresponding split thresholds for each node. In abstract, all the surrogates would be stored as two matrices: one for each surrogate feature index and another for corresponding threshold.

iii) Update to search using surrogates: During searching the tree for a incoming tuple we go through the tree as usual. If for a node, the feature to check is NULL in the tuple, we go through the surrogate splits. If at least one of the surrogate split features has a non-NULL

value, we use it search further. If an observation is missing all the surrogates we use the majority branch to search further.

iv) Displaying surrogate splits: we can display the surrogate splits for each split in a separate table - this can implemented as a function call for the user. The table would have a row for each node, with a column for each surrogate split variable. Along with the surrogate split we also display the aggrement measure that indicates how similar the two splits are.

It's important to note that, similar to the 'rpart' package, no surrogate that does worse than "go with the majority" is printed or used. The "go with the majority" rule, also referred to as the "blind" rule computes the agreement measure when we always go with the branch that has majority of the tuples. Computing this blind rule measure is easy since we already have the statistics of tuples going left and right from our previous training of the node.

16.3.4 Pruning

See [38] for details on various theoretical explanations in this section.

We use the same method used by R's rpart package to prune the tree. The method is a little different from what is normally described in textbooks.

It is often observed that a decision tree perfect on the training set, will have a worse generalization ability than a tree which is not-so-good on the training set; this is called **overfitting** which may be caused by the fact that some peculiarities of the training data, such as those caused by noise in collecting training examples, are misleadingly recognized by the learner as the underlying truth. To reduce the risk of overfitting, a general strategy is to employ pruning to cut off some tree branches caused by noise or peculiar- ities of the training set. **Pre-pruning** tries to prune branches when the tree is being grown, while **post-pruning** re-examines fully grown trees to decide which branches should be removed. When a validation set is available, the tree can be pruned according to the validation error: for pre-pruning, a branch will not be grown if the validation error will increase by growing the branch; for post-pruning, a branch will be removed if the removal will decrease the validation error.

To perform pruning, we define a misclassification (resubstitution) error of the tree, which is the number of misclassified entries for a classification tree and the mean-squared error for a regression tree. We can estimate this error or risk (R^*) for classification tree as,

$$R^*(T) = \sum_{t \in \tilde{T}} r(t)p(t), \tag{16.3.1}$$

where $p(t)$ is the proportion of points in terminal node t (member of terminal nodes set $\tilde{T}$) and $r(t)$ is the probability of making wrong classification for points in node t. For a point in a given leaf node t, the estimated probability of misclassification is 1 minus the probability of the majority class in node t based on the training data.

For a regression tree this would be estimated as

$$R^*(t) = \sum_{t \in \tilde{T}} Var(t),$$

where $Var(t)$ is the variance of the dataset assigned to node t.

It is easy to prove that the resubstitution error rate $R(T)$ is biased downward, i.e. it tends to produce bigger trees. Hence, we need to add a complexity penalty to this resubstitution error rate. The penalty term favors smaller trees, and hence balances with $R(T)$.

For any subtree $T < T_{\max}$, we will define its complexity as $|\tilde{T}|$, the number of terminal or leaf nodes in T. Let $\alpha \geq 0$ be a real number called the 'complexity parameter' and define the cost-complexity measure $R_\alpha(T)$ as

$$R_\alpha(T) = R(T) + \alpha|\tilde{T}| \cdot R(T_1), \tag{16.3.2}$$

where $R(T_1)$ is the risk computed for the tree with only one root node. Note that textbooks usually do not have $R(T_1)$ in the definition of the cost-complexity. According to the R package 'rpart', the above scaled version is unitless and thus much more user friendly than the original CART formular where there is no $R(T_1)$.

Because we are using this scaled version, we can simplify the computation of risk in the case of classification Eq. (16.3.1). In Eq. (16.3.1), we have $r = n_{mis}/n$ and $p = n/N$, where n_{mis} is the number of mis-classified records, n is the number records classified into the current node and N is the total number of records. Clearly in Eq. (16.3.2), we can directly use n_{mis} instead of $r \times p$.

When there is a weight for each record, we should use weight in the calculation. For example, n_{mis} should be replaced by the sum of weights of the mis-classified records.

There is some logic in rpart's code that is not described anywhere. We try to give more detailed explanation here.

(1) For a node i, we compute the risk of this node r_i, and the total risk s_i of all nodes in the subtree where the node i is the root node and not included in the subtree. The number of splits in subtree-i is $m_i = m_{i,left} + m_{i,right} + 1$. Here the subscript "left" and "right" denote the left and right child of node i. Another important quatity is the cost-complexity itself c_i.

(2) Instead of directly applying Eq. (16.3.2), we use the following logic: For a node, we compute the average improvement per split in the subtree of node i.

$$\bar{t} = \frac{r_i - s_{i,left} - s_{i,right}}{m_i}. \tag{16.3.3}$$

If $\bar{t} > c_{i,right} > c_{i,left}$, the improvement at this split is more important than the splits in the sub-tree, since we already keep the nodes in the sub-tree, we must keep the current split. Thus, we re-compute the average improvement as the following

$$\bar{t} = \frac{r_i - r_{i,left} - s_{i,right}}{m_{i,right} + 1}. \tag{16.3.4}$$

And then if $\bar{t} > c_{i,right}$, we need to do the same thing and update

$$\bar{t} = r_i - r_{i,left} - r_{i,right}. \tag{16.3.5}$$

And when $\bar{t} > \alpha R(T_1)$, we keep the split.

We use a similar method if instead $\bar{t} > c_{i,left} > c_{i,right}$, then we first deal with the right child, and then deal with the left child.

(3) We recursively call the function of pruning. The question is when we stop. We stop when the current node under examination is a leaf node, or the risk is smaller than the threshold $\alpha \times R(T_1)$ (no change can make the improvement larger than this threshold). However, rpart uses a fairly complicated method to estimate the risk that is used to compare with the threshold. See the code for more details.

At the end, the cost complexity measure comes as a penalized version of the resubstitution error rate. This is the function to be minimized when pruning the tree. In general, given a pre-selected α, we can find the subtree $T(\alpha)$ that minimizes $R_\alpha(T)$. Since there are at most a finite number of subtrees of $T_{\max}$, $R_\alpha(T(\alpha))$ yields different values for only finitely many α's. T(α) continues to be the minimizing tree when α increases until a jump point is reached.

16.3.5 Cross-validation with the cost-complexity

The starting point for the pruning is not $T_{\max}$, but rather $T_1 = T(0)$, which is the smallest subtree of $T_{\max}$ satisfying $R(T_1) = R(T_{\max})$.

First, look at the biggest tree, $T_{\max}$, and for any two terminal nodes descended from the same parent, for instance t_L and t_R, if they yield the same resubstitution error rate as the parent node t, prune off these two terminal nodes, that is, if $R(t) = R(t_L) + R(t_R)$, prune off t_L and t_R. This process is applied recursively. After we have pruned one pair of terminal nodes, the tree shrinks a little bit. Then based on the smaller tree, we do the same thing until we cannot find any pair of terminal nodes satisfying this equality. The resulting tree at this point is T_1.

We now find the next α using the following method. The new α will result in a different optimal subtree.

For any node $t \in T_1$, we can set $R_\alpha(t) = R(t) + \alpha$, and for the subtree starting at t, we can define $R_\alpha(T_t) = R(T_t) + \alpha|\tilde{T}_t|$. When $\alpha = 0, R_0(T_t) < R_0(t)$, but when α increases there is a point at which $R_\alpha(T_t) = R_\alpha(t)$. We compute this exact alpha value by minimizing the function

$$g_1(t) = \begin{cases} \frac{R(t)-R(T_t)}{|\tilde{T}_t|-1}, & \text{if } t \notin \tilde{T}_1 \\ \infty, & \text{if } t \in \tilde{T}_1. \end{cases}$$

We find the weakest link $\bar{t_1} \in T_1$ that achives the minimum of $g_1(t)$ and set the new $\alpha_2 = g_1(t)$. To get the optimal subtree T_2, simply remove the branch growing out of $\bar{t_1}$. If there are several nodes that simultaneously achieve the minimum $g_1(t)$, we remove the branch grown out of each of these nodes. We repeat this process till we end up with only the root node i.e. a single-split tree.

To find which of the multiple optimal subtrees is the best one, we can run cross-validation for the optimal subtree of each α to obtain an average test resubstitution error for each optimal subtree and pick the one with the lowest error.

16.4 Parallelism

Let $X = \{X_1, X_2, \ldots, X_N\}$ be a set of attributes with domains $D_{X_1}, D_{X_2}, \ldots, D_{X_N}$. Let Y be an output with domain D_Y. Given a dataset $D^* = \{(x_i, y_i) | x_i \in D_{X_1} \times D_{X_2} \times \ldots D_{X_N}, y_i \in D_Y\}$, the goal is to learn a tree model that best approximates the true distribution of D^*. However, finding the optimal tree is a NP-Hard problem, thus most decision tree construction algorithms use a greedy top-down approach as described in Algorithm `BuildSubtree`.

Algorithm `BuildSubtree`(n, D)
Input: Node n, Data $D \in D^*$
Output: A Decision Tree Model

```
1: (n → split, D_L, D_R) = FindBestSplit(D)
2: if StoppingCriteria(D_L) then
3:     n → left_prediction = FindPrediction(D_L)
4: else
5:     BuildSubtree(n → left, D_L)
6: if StoppingCriteria(D_R) then
7:     n → right_prediction = FindPrediction(D_R)
8: else
9:     BuildSubtree(n → right, D_R)
```

Note that the each call of Algorithm `BuildSubtree` requires at lease one pass over the dataset D^*, which means that the number of passes needed for training a depth l tree is $O(2^l)$. The node-wise training incurs IO cost that is exponential to tree depth which does not scale to large dataset.

To scale to large dataset, in our algorithm, we perform level-wise training strategy as proposed in [58]. With each pass over the training dataset, we find best splits for all nodes at the same level. Moreover, to reduce search space in finding best splits, we construct equal-depth binning for each attributes and only consider bin boundaries as split candidates. In what follows, we will describe details of each step in level-wise decision tree training algorithm. The following assumptions hold in the level-wise training algorithm.

1. The tree model can fit in memory
2. Dataset is distributed over multiple machines

16.4.1 Initialization

In the initialization step, we find split candidates by equal-depth binning on each attribute. For continuous attributes, on large distributed datasets, we perform a quantile calculation over a sampled fraction of the data to get an approximation set of split candidates. The ordered splits create bins and the maximum number of such bins can be specified by user. Note that the number of bins cannot be greater than the number of training examples. The tree algorithm automatically reduces the number of bins if the condition is not satisfied.

For M categorical attributes and binary classification, the number of split candidates can be reduced to $M-1$ by ordering the categorical attribute values by the proportion of labels falling in one of the two classes. For example, for a binary classification problem with one categorical feature with three categories A, B and C with corresponding proportion of label 1 as 0.2, 0.6 and 0.4,the categorical features are ordered as A followed by C followed by B. The two split candidates are $A|C,B$ and $A,C|B$ where $|$ denotes the split. For categorical variables in multiclass classification, each bin is a category. The bins are sorted and they are ordered by calculating the impurity of their corresponding labels.

Algorithm `findSplitBins` describes binning for CART.

Algorithm `findSplitBins`(D^*, MB)
Input: Training dataset D^*, max bins for each feature MB
Output: A equal-depth binning for each attributes in D

```
1: if MB < |D*| then
2:     numBins = MB
3: else
4:     numBins = |D*|
5: if numBins * numBins < |D*| then
6:     Sample fraction f = numBins * numBins/|D*|
7: else
8:     Sample fraction f = 1.0
9: Randomly select f|D*| records R from D* and load R into memory
10: for all attr in D* do
11:     if attr is continuous then
12:         sort R by attr, perform quantile computation
13:     else
14:         if binary classification then
```

```
15:             compute centroid of labels for each value in attr
16:             sort attr by centroid
17:         else
18:             compute impurity for each label for each value in attr
19:             sort attr by impurity
```

16.4.2 Find best splits

For a particular split node, we want to find an attribute that maximizes purity. In classification, we use Gini impurity and entropy as our impurity measure and in regression, we use variance as the impurity measure.

For classification, we need to compute the frequency f_i for label i at each node to compute Gini impurity and entropy.

For regression, we need to compute

$$\text{Var}(D) = \frac{1}{n}\sum_i y_i^2 - \left(\frac{1}{n}\sum_i y_i\right)$$

where y_i is value for attribute Y and n is the number of examples in D. One important observation is that both label frequency and variance can be computed from sufficient statistics.

For label frequency, the sufficient statistics we aggregate when passing over the training examples are

- The number of examples with label i, $N_i = \sum_i I(i)$

For variance, the sufficient statistics we aggregate when passing over the training examples are

- The number of training $N = \sum_i 1$
- The sum of values $S = \sum_i y_i$
- The sum of square values $Q = \sum_i y_i^2$

Note that the computations of the above sufficient statistics are associative, we then have the following high level algorithm for parallel decision tree construction. Assume that we are training nodes at level l, in each segment, we store the aggregated values for sufficient statistics that we need to compute purity. As we pass over each record r in each segment, we find the corresponding bin for each attributes in r and add to the aggregated statistics. When all segments finish data processing, the aggregated sufficient statistics will be merged. Once we have the overall aggregated sufficient statistics, we find the split that maximizes impurity for each node at the level l.

Algorithm `findBestSplits`(D^*, M, n_l, B)

Input: Training dataset D^*,
Tree model M,
Set of nodes at level l, n_l, Binning info B

Output: split that maximizes impurity for each node in n_l

```
1: On each segment, perform the following                    ▷ Transition function
2: for d ∈ D* do
3:     for attr_value ∈ d do
4:         Find the bin index in B that attr_value belongs to
5:         Add sufficient statistics s to aggregate
```

6: Merge aggregated statistics on different segments ▷ Merge function
7: Find splits that maximize impurity for each node in n_l ▷ Final function

For Algorithm `findBestSplits` to handle weights for each training example, when we add sufficient statistics s to aggregate, we instead add $w \times s$ to aggregage, where w is the weight for the traning example.

17 Random Forests: Classification and Regression

Author Preethi Jayaram

Author Feng, Xixuan (Aaron)

History **v0.1** Initial version: introduction, theory, and interface

17.1 Introduction

Tree-based methods provide a simple and human-interpretable model for classifying data. Decision Trees, for example, use splitting rules to segment the predictor space into simple regions that are then summarized in a tree form. However, they tend to over-fit the data, resulting in poor prediction accuracies. One way to mitigate this problem is by building an ensemble of classifiers, each of which produces a tree model on some combination of the input data. The results of these models are then combined to yield a single prediction, which, although at the expense of some loss in interpretation, have been found to be highly accurate. Such methods of using multiple decision trees to make predictions are called random forest methods.

17.1.1 Basics of Random Forests

Random forests operate by constructing many trees at training time using bootstrapped samples from the training data. During prediction, they output the class that is the mean (regression) or mode (classification) of the predictions output by the individual trees.

Each tree is grown as follows:

i) If the number of training samples is N, we sample, with replacement, N cases at random. This forms the training set for the current tree.

ii) Each time a split is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors such that $m \ll p$. Typically, we choose $m \approx \sqrt{p}$. One of the m predictors is then used to split the node.

iii) The tree is grown to the maximum size without pruning.

[10] proves that the generalization error, or the misclassification rate, has a limiting value and that, random forests do not over-fit the data.

17.1.2 Out-of-bag error (oob) error estimate

The Out-of-bag error is an unbiased estimate of the test set error. While growing each tree during training, about one-third of the samples are left out. These are used to get a test-set classification for each sample in about one-third of the trees. The average prediction c for each sample is calculated as the average of the predictions output by individual trees. The number of times c is not equal

to the true class of the sample, averaged over all the samples, is the oob error estimate. Such an unbiased error estimate removes the need for cross-validation or a separate test set for calculating test set error.

17.1.3 Variable importance

Random forests can be used to rank the importance of variables in the dataset. For a given dataset $D_n = \{(X_i, Y_i)\}_{i=1}^n$, first, a random forest is fit to the data. To measure the importance of the j-th feature after training, the values of the j-th feature are permuted among the training data, and the oob error is calculated on this perturbed dataset. The importance of the j-th feature is calculated by averaging the difference between the oob error estimate before and after permuting the j-th feature. The average is normalized by the standard deviations of the differences.

17.1.4 Proximities

Proximity matrices are a useful tool in clustering and outlier detection. Random forests can be used to calculate pair-wise proximities between input samples. If there are N training samples, the proximity matrix is created as an NxN matrix. After a tree is grown, both the training and oob data are sent down the tree. If two different samples end up in the same leaf node, their proximity is increased by one. After repeating this process for all samples, the proximity values are normalized by diving by the number of trees.

Advantages of Random forests:

i) It is one of the most accurate learning algorithms available. For many data sets, it produces a highly accurate classifier.

ii) It runs efficiently on large databases.

iii) It can handle thousands of input variables without variable deletion.

iv) It gives estimates of what variables are important in the classification.

v) It generates an internal unbiased estimate of the generalization error as the forest building progresses.

vi) It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing.

vii) It computes proximities between pairs of cases that can be used in clustering, locating outliers, or (by scaling) give interesting views of the data.

viii) The capabilities of the above can be extended to unlabeled data, leading to unsupervised clustering and outlier detection. Proximity matrix calculated from the random forest for unlabeled data can be used for clustering and outlier detection.

17.2 Interface

The interface to the random forest functionality includes forest_train and forest_predict as the training and prediction functions, and forest_display as the visualization function.

17.2.1 Training

```
SELECT forest_train(
        training_table_name,
        output_table_name,
        id_col_name,
        dependent_variable,
        list_of_features,
        list_of_features_to_exclude,
        grouping_cols,
        num_max_trees,
        num_random_features,
        max_depth,
        min_split,
        min_bucket,
        verbose
)
```

Arguments: The descriptions of the arguments are the same as found in Decision Trees with the exception of the following:

- *num_max_trees*: Maximum number of trees to grow while building the random forest. Default: 500
- *num_random_features*: Number of features to randomly select at each split. Default=$\sqrt{p}$ for classification, and p/3 for regression, where p is the total number of features.
- There will be no parameters for cross-validation and pruning as random forests build complete trees without pruning, and the technique of cross-validation is inherent.

17.2.2 Prediction

```
SELECT forest_predict(
        random_forest_model,
        new_data_table,
        output_table,
        type
)
```

The descriptions of all the arguments are the same as found in Decision Trees.

17.2.3 Training Output

The function forest_train produces three output tables, one each for the model, summary and groups, as described below:

Model table: The structure of the model table remains mostly similar to the one constructed by Decision trees. For Random forests, we provide two additional fields: 'sample_id' and 'group_id'. sample_id refers to the id of the bootstrap sample, which is a number ranging from 1 to num_trees, where num_trees is the number of trees that were actually grown. group_id is an integer assigned to each unique combination of values of grouping columns. As one can see, a tree can uniquely be identified by using the sample_id as well as particular values for the grouping columns. To make it easier for the user to retrieve a tree, we assign a group_id instead of the user having to list all grouping columns and their values. The total number of trees in the model table will be equal to the number of groups multiplied by the number of bootstrap samples.

- *group_id*: If grouping columns were provided, this refers to the Id of the group for which this tree was generated. The specific grouping columns and values can be obtained from the 'Groups' table.
- *sample_id*: Id of the (bootstrap) sample for which this tree was generated.
- *tree*: Trained decision tree model stored in bytea8 format.
- *cat_levels_in_text*: Ordered levels of categorical variables.
- *cat_n_levels*: Number of levels for each categorical variable.
- *tree_depth*: Depth of the tree.

Summary table: This table stores details that are common across all random forest models created by the training function.

- *method_name*: Name of training method, 'forest_train' in this case.
- *is_classification*: Indicates if the model is for classification/regression.
- *training_table_name*: Name of table containing training data.
- *output_table_name*: Name of table containing the model.
- *id_col_name*: The Id column name.
- *dependent_variable*: The dependent variable.
- *features*: The independent variables.
- *cat_features_str*: The list of categorical feature names as a comma-separated string.
- *con_features_str*: The list of continuous feature names as a comma-separated string.
- *grouping_cols_str*: Names of grouping columns.
- *n_groups*: Number of groups in training.
- *failed_groups*: Number of failed groups in training.
- *n_rows*: Total number of rows processed across all groups.
- *n_rows_skipped*: Total number of rows skipped across groups.
- *dep_list_str*: Distinct levels of dependent variable in the case of classification.
- *dep_type*: Type of dependent variable.

Groups table: This table stores statistics on a per-group basis. It has one row for each group (which in turn corresponds to the random forest model for that group).

- *grouping_cols*: Zero or more grouping columns depending upon user input.
- *group_id*: If grouping columns were provided, this refers to the Id of the group that this row of statistics corresponds to.
- *num_trees*: Actual number of trees constructed by the model. This should usually be the same as num_max_trees, but we could optimize the implementation to limit the number of trees to grow based on the observed oob error during construction of the random forest model for each group.
- *oob_error*: Out-of-bag error estimate for the model.
- *variable_importance*: A matrix containing the permutation-based importance measures. The rows correspond to the different predictors, and the columns correspond to various importance measures. Importance measures include Mean Decrease in MSE (for regression) and Mean Decrease in Classification Accuracy for classification. Additionally, we may be able to add Mean Decrease in Gini criterion for classification.
- *proximity*: A matrix containing proximity measures among the input, based on the number of times pairs of input points end up in the same terminal node. Note that this feature being Priority 2, will not be available in the first iteration.

17.2.4 Prediction Output

The function forest_predict produces one output table containing a list of predictions, which are either class probabilities, or classification prediction outputs. In the case of classification with multiple classes, an array of class probabilities are produced as output. The output table is as described below:

- *id*: Id of the data point.
- *estimated_<column>*: Contains prediction for this data point. Can be one (regression) or more columns(classification).

17.2.5 Other functions

17.2.5.1 Display

```
1: SELECT forest_display(
2:       random_forest_model,
3:       group_id,
4:       sample_id,
5:       is_dot_format)
```

- *random_forest_model*: Table containing the random forest model.
- *group_id*: Id of the group that the tree to be displayed is a part of.
- *sample_id*: Id of the sample within the group.
- *is_dot_format*: True for dot format, False for text format.

The output of the display function to output individual trees will follow the same format as used by Decision Trees.

17.3 Implementation

Let $X = \{X_1, X_2, \ldots, X_N\}$ be a set of attributes with domains $D_{X_1}, D_{X_2}, \ldots, D_{X_N}$. Let Y be an output with domain D_Y. Given a dataset $D^* = \{(x_i, y_i) | x_i \in D_{X_1} \times D_{X_2} \times \ldots D_{X_N}, y_i \in D_Y\}$, the goal is to learn a random forest model that best approximates the true distribution of D^*. The random forest will be composed of individual trees each of which is learnt using a greedy top-down approach, as described in Decision Trees.

In the below sections, we will only expand item[1] and item[5], which are specific to building random forests. The others have already been covered in Decision Trees. Any differences will be noted along in the process.

The following are the major steps for building a random forest model:

1. Bootstrap, which includes sampling N samples from the training set.
2. Initialize, which includes binning and finding split candidates.
3. Finding best split for each node.
4. Finding prediction for leaf nodes in the individual decision trees.
5. Calculate random forest statistics.

17.3.1 Bootstrapping

Given the dataset $D^* = \{(x_i, y_i) | x_i \in D_{X_1} \times D_{X_2} \times \ldots D_{X_N}, y_i \in D_Y\}$ we will generate B bootstrap samples $B_1, B_2, \ldots B_b$ where each bootstrap sample is obtained by sampling N times, with replacement, from the same dataset. Each bootstrapped sample B_i is treated as the training set for `BuildRandomForest`. We thought about a couple of approaches for achieving this:

17.3.1.1 Using Row Identifiers

The first method generates row ids for each row of the input table, which ranges from 1..N. We then generate N random numbers between 1 and N, and store the results in a one-column table. A bootstrapped sample can now be obtained by joining this table with the input dataset. See 2.2 for implementing this in sql. The problem, however, is that, the step which generates row ids for the input table would involve pulling in data from all segments to the master in order to be able to generate a sequential gap-less set of ids. This is an expensive operation which is overcome by the second method.

17.3.1.2 Using Poisson sampling

The process of bootstrapping can be thought of as sampling from a multinomial distribution where the probability of selecting any data point is uniform over the entire data set. While this requires scanning the entire data set, the second approach approximates the multinomial distribution by sampling from an identical Poisson distribution on each data point independently. Thus, we are able to generate counts indicative of the number of times the particular data point should be included in the bootstrap sample.

It has been decided that we will be using Poisson sampling for our bootstrapping step. We will use a Poisson distribution with $\lambda = 1$ where λ is the average fraction of input data that we want in each tree. In our case, this fraction = 1 as we want N samples in each tree from the input where N is also the size of the data set.

Algorithm `Bootstrapping`(D^*)
Input: Training dataset D^*
Output: Bootstrap sample B

1: **for** $d \in D^*$ **do**
2: count = Poisson(1)
3: id = user provided id for d
4: Add $(count, id)$ to C where C is a two-column table
5: Output $Join(D^*, C)$

17.3.2 Variable Importance

"Subtract the number of votes for the correct class in the variable-m-permuted oob data from the number of votes for the correct class in the untouched oob data." [14] According to the source code of R package randomForest [15], we define the normalization of the subtraction,

$$D_{m,g,t,p} = \frac{\left(\sum_{n \in oob_{t,g}} predict(n) == y_n\right) - \left(\sum_{n \in oob_{t,g}} predict_p^m(n) == y_n\right)}{oobsize_{t,g}}.$$

For regression,

$$D_{m,g,t,p} = \frac{\left[\sum_{n \in oob_{t,g}} \left(predict_p^m(n) - y_n\right)^2\right] - \left[\sum_{n \in oob_{t,g}} \left(predict(n) - y_n\right)^2\right]}{oobsize_{t,g}}.$$

"The average of this number over all trees in the forest is the raw importance score for variable m." [14] Considering multiple permutations, we have

$$importance_{m,g} = \sum_{p=1}^{P} \sum_{t=1}^{T} \frac{D_{m,g,t,p}}{TP}$$

for each variable m per group g.

Note: R package randomForest [15] computes the above importance without considering grouping support and aggregating p prior to t.

In order to rank the importance of features in the input dataset, random forests use a mechanism of calculating the average increase in misclassification rate when the values of a particular feature are randomly permuted in the input set, and comparing the classification accuracy against the original data set. The algorithm is as follows:

Algorithm `calculateVariableImportance`(R, F^*)
Input: Random Forest model R
Input: Features F^*
Output: Variable importance for each feature in F

1: **for** $subtreeT_i \in R$ **do**
2: O_i = out-of-bag samples of T_i

3: size = size of O_i
4: **for** $f \in F$ **do**
5: Create one-column table C with values of f randomly selected *size* times
6: Join (O_i, C) replacing column f with values from C
7: **for** $o \in O_i$ **do**
8: Find prediction for o, and increment count for (f, o, prediction)
9: Find majority prediction for each input point per feature
10: Calculate misclassification rate per feature

17.3.3 Proximities

Proximities help identify clusters within the input set. This is depicted as an NxN matrix where N is the size of the data set. The algorithm is as follows:

Algorithm `calculateProximity`(R, D^*)
Input: Random Forest model R
Output: Proximity for data points in D^*

1: **for** $subtreeT_i \in R$ **do**
2: **for** $d \in D^*$ **do**
3: Find prediction p for d in T_i. Increment pairwise counts of d with all other data points with prediction p.
4: Normalize proximity by dividing by number of trees.

17.3.4 Feature sampling

Unlike in decision trees where all the features are used to grow the tree, random forests will at each split, select at random, a subset of the features to find the best split candidate. This would require some changes in the current Decision tree implementation to both support and be able to use the reduced feature set for an optimized implementation, by, for example, only having to aggregate statistics for the feature subsample.

17.3.5 Random forest and Decision Tree building algorithms

Algorithm `BuildRandomForest`(D^*)
Input: Training dataset D^*
Output: A random forest model fit to the dataset D

1: Generate bootstrap samples $B_1, B_2, \ldots B_r$ using `Bootstrapping`
2: **for** $i \in B_i$ **do**
3: Generate binning info b_i using `findSplitBins`
4: Build subtree using `buildSubtreeRandomForest`

Algorithm `buildSubtreeRandomForest`(D^*, M, n_l, B)
Input: Bootstrapped training dataset B_i,
Binning info b
Output: Subtree built from B_i

1: Select m features at random from the set of p features
2: On each segment, perform the following ▷ Transition function
3: **for** $d \in B_i$ **do**
4: **for** $attr \in m$ in d **do**

5: Find the bin index in b that *attr_value* belongs to
6: Add sufficient statistics s to aggregate
7: Merge aggregated statistics on different segments ▷ Merge function
8: Find split in m that maximizes impurity using `findBestSplits` ▷ Final function

17.3.6 Grouping Support

Like other algorithms in MADlib, Random Forests will also support grouping. A lot of the functionality that exists in Decision Trees to support grouping can be leveraged. Two major steps to support grouping in random forests would be:

- Generate bootstrap samples for a particular group before training a random forest on that group. See 2.2 for implementation details of sampling with replacement involving groups, in the database.
- Aggregate necessary results on a per-group basis in order to output final statistics such as variable importance, proximity etc.

17.4 Data Handling

In order to calculate metrics such as the oob error, variable importance and proximities, data from all the trees need to aggregated. Three different alternatives were experimented with, w.r.t storing and retrieving data.

i) In the first approach, we construct two tables, one that stores the bootstrap samples of the current working set, i.e., the data needed to construct the current tree, and another table which accumulates both the training and oob samples for all trees. Metrics are then calculated at the end of training the random forest model by running aggregation queries on the table with all of the data. The query to accumulate all data simply appends the current sample to the aggregated sample.

ii) In the second approach, we store one big table, which is used for both training the current tree, as well as accumulating samples for all trees. The overall amount of data stored is reduced.

iii) In the final approach, we construct one table to store the current working set like in approach (1), and additionally, we store one other table to store various partially aggregated statistics from training each tree, such as the predictions for oob samples for each tree. These will be aggregated at the end of training to produce the final results.

iv) Based on preliminary tests on HAWQ (with half a rack), a table with 1M rows using approach (3) takes only half as much time as required for approaches (1) and (2)

17.5 Design Considerations

- Building a random forest can potentially be made embarrassingly parallel as each tree in the model can be built independently of the rest. This, however, requires that each segment has all the data necessary for training a single tree, and would involve copying large amounts of data over to the segments, which is a lot of overhead. It has therefore been decided that

individual trees will be built in sequence, with each tree itself being built in parallel, as discussed under Decision Trees.

- Random forests are usually built without pruning, which could lead to trees that are very deep. That could potentially have an impact on performance. The current design proposes not to use pruning, but based on performance tests and accuracy results, this could be changed to use pruning and/or limit the maximum depth of the tree.
- num_trees is currently an input parameter to the rf_train method. If, however, we want to be able to determine the number of trees to use, one approach is to use a number that is typically used, such as few hundreds of trees. Another way might be to observe the out-of-bag error rate on each training sample on the current model, and stop constructing more trees when the improvement in the error rate falls below a certain threshold.

18 SVM

Authors

History **v0.1** Initial version

Support Vector Machines (SVMs) are a commonly used technique for approaching regression and classification problems. SVM models have two particularly desirable features: robustness in the presence of noisy data, and applicability to a variety of data schemes. At its core, a *linear* SVM model is a hyperplane separating the two distinct classes of data (in the case of classification problems), in such a way that the *margin* is maximized. Using kernels, one can approximate a large variety of decision boundaries.

18.1 Linear SVM

Suppose we have a classification task with training data $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R}^d \times \{0, 1\}$. A hyperplane in $\mathbb{R}^d$ is determined by $w \in R^d$ and $b \in R$: $x \in \mathbb{R}^d$ is on the plane if $\langle w, x \rangle + b = 0$. To solve the classification problem, we seek a hyperplane which separates the data, and therefore coefficients $(w, b) \in \mathbb{R}^d \times \mathbb{R}$ such that for all $i \leq n$, $y_i = \text{sgn}(\langle w, x_i \rangle + b)$.

The *margin* determined by a hyperplane and the training data is the smallest distance from a data point to the hyperplane. For convenience, the length of w can be modified so that the margin is equal to $\frac{1}{||w||}$. In addition to merely separating the data, it is desirable to find a hyperplane which maximizes the margin; inuitively, a larger margin will lead to better predictions on future data.

The *support vectors* are the members of the training data whose distance to hyperplane is exactly the margin; that is, the training points which are closest to the hyperplane. Learning with SVM lends itself to large datasets in part because the model in depends only on these support vectors; all other training data do not influence the final model. When assigning values to new data, one needs only to check it against the support vectors.

Thus, we are left with the following convex programming problem:

$$\begin{aligned} \underset{w,b}{\text{Minimize}} \ \frac{1}{2}||w||^2, w \in \mathbb{R}^d, \\ \text{subject to } y_i(\langle w, x_i \rangle + b) \geq 1 \text{ for } i = 1 \ldots n. \end{aligned}$$

If the data are not linearly separable, *slack variables* ξ_i are introduced to allow for some points to be classified incorrectly. We do not want to sacrifice too much to capture outliers, so we introduce an extra term in the objective function to control the error. The optimization problem now becomes

$$\begin{aligned} \underset{w,\boldsymbol{\xi},b}{\text{Minimize}}\ & \frac{1}{2}||w||^2 + \frac{C}{n}\sum_{i=1}^{n}\xi_i \\ \text{subject to}\ & y_i(\langle w, x_i\rangle + b) \geq 1 - \xi_i, , \\ & \xi_i \geq 0. \end{aligned}$$

The parameter C is free, and should be chosen to optimize the model. One can use cross-validation, for example, to optimize the choice of C.

18.1.1 Unconstrained optimization

One can formulate the same problem without constraints:

$$\underset{w}{\text{Minimize}}\ \lambda||w||^2 + \frac{1}{n}\sum_{i=1}^{n}\ell(y_i, f(x_i)) \tag{18.1.1}$$

where $\ell(y, f(x)) = \max(0, 1 - yf(x))$ is the *hinge loss*. Also, in the case of linear SVM, $f(x) = \langle w, x\rangle + b$ (Note: there are a couple ways to handle the variable b; one can include it in the gradient calculations, or add extra feature to the data space.) Also note that this loss function is not smooth. However, one can solve this unconstrained convex problem using the techniques outlined in the "Convex Programming Framework" section of this document. In particular, the subgradient of this loss function which is used in our gradient-descent method is:

$$v(y, f(x)) = \begin{cases} 0 & 1 - yf(x) \leq 0 \\ -y \cdot x & 1 - yf(x) > 0 \end{cases}$$

See [65] (extended version) for details.

18.1.2 ϵ-Regression

SVM can also be used to predict the values of an affine function $f(x) = \langle w, x\rangle + b$, given sample input-output pairs $(x_1, y_1), \ldots, (x_n, y_n)$. If we allow ourselves an error bound of $\epsilon > 0$, and some error controlled by the slack variables ξ^*, it is a matter of simply modifying the above convex problem. By demanding that our function is relatively "flat," and that it approximates the true f reasonably, the relevant optimization problem is:

$$\begin{aligned} \underset{w,\boldsymbol{\xi},\boldsymbol{\xi^*},b}{\text{Minimize}}\ & \frac{1}{2}||w||^2 + \frac{C}{n}\sum_{i=1}^{n}\xi_i + \xi_i^* \\ \text{subject to}\ & y_i - \langle w, x_i\rangle - b \leq \epsilon + \xi_i, \\ & \langle w, x_i\rangle + b - y_i \leq \epsilon + \xi_i^*, \\ & \xi_i, \xi_i^* \geq 0 \end{aligned}$$

One can also formulate ϵ-regression as an unconstrained optimization problem just as we did with classification. In fact, the objective function remains the same, except rather than hinge loss we would use the loss function

$$\ell(y, f(x)) = \max(0, |y - f(x)| - \epsilon)$$

whose subgradient is

$$v(y, f(x)) = \begin{cases} x & f(x) - y > \epsilon \\ -x & y - f(x) > \epsilon \\ 0 & \text{otherwise} \end{cases}$$

18.2 Nonlinear SVM

One can generalize the above optimization problems using *kernels*. A kernel is nothing more than a continuous, symmetric, positive-definite function $k : X \times X \to \mathbb{R}$. This replacement, known as the *kernel trick*, allows one to compute decision boundaries which are more general than hyperplanes.

Two commonly used kernels are polynomial kernels, of the form $k(x, y) = (\langle x, y \rangle + q)^r$, and Gaussian kernels, $k(x, y) = e^{-\gamma||x-y||^2}$, γ a free parameter.

The approach we take to learning with these kernels, following [60], is to approximate them with linear kernels and then apply the above gradient descent based algorithm for learning linear kernels. The approximation takes the following form: we embed the data into a finite dimensional feature space, $z : \mathbb{R}^d \to \mathbb{R}^D$, so that the inner product between transformed points approximates the kernel:

$$k(x, y) \approx \langle z(x), z(y) \rangle.$$

This map z will be a randomly generated (in a manner depending on the kernel) by methods outlined in the papers [44, 60]. We then use z to embed both the training data and the test data into $\mathbb{R}^D$ and run linear SVM on the embedded data.

18.2.1 Gaussian kernels

A kernel k is said to be shift-invariant if, for all x, y, z, $k(x, y) = k(x - z, y - z)$. In the paper [60], the authors show that for any shift- invariant kernel k, there exists a probability measure $p(w)$ on its domain such that

$$k(x, y) = k(x - y, 0) = \int_X p(w) e^{i\langle w, x-y \rangle} dw = E_w[e^{i\langle w,x \rangle} e^{-i\langle w,y \rangle}]$$

where the expectation above is taken with respect to the measure p. In the particular case of $k(x, y) = \exp(-\gamma||x - y||^2)$,

$$p(w) = \left(\frac{1}{\sqrt{4\pi\gamma}}\right)^d \exp\left(\frac{-||w||^2}{4\gamma}\right)$$

where d is the dimension of the sample data. In other words, $p(w)$ is a Gaussian distribution with mean zero and standard deviation $\sigma = \sqrt{2\gamma}$.

Since both the kernel and the probability distribution are real-valued in the above integral, the complex exponential can be replaced with $\cos(\langle w, x - y \rangle)$. Define a random function $z : x \mapsto$

$\sqrt{2}\cos(\langle w, x\rangle + b)$, where w is drawn from the distribution p and b uniformly from $[0, 2\pi]$. Then $\langle z(x), z(y)\rangle$ is an unbiased estimator of $k(x, y) = k(x - y, 0)$. To see this, use the sum of angles formula:

$$\begin{aligned} z(x)z(y) &= 2\cos(\langle w, x\rangle + b)\cos(\langle w, y\rangle + b) \\ &= 2\cos(\langle w, x\rangle)\cos(\langle w, y\rangle)\cos^2(b) \\ &+ \sin(\langle w, x\rangle)\sin(\langle w, y\rangle)\sin^2(b) - \sin(\langle w, x\rangle + \langle w, y\rangle)\cos(b)\sin(b) \end{aligned}$$

Since w, b are chosen independently,

$$\begin{aligned} E[2\cos(\langle w, x\rangle)\cos(\langle w, y\rangle)\cos^2(b)] &= 2E[2\cos(\langle w, x\rangle)\cos(\langle w, y\rangle)]E[\cos^2(b)] \\ &= 2E[\cos(\langle w, x\rangle)\cos(\langle w, y\rangle)] \cdot \frac{1}{2} \\ &= E[\cos(\langle w, x\rangle)\cos(\langle w, y\rangle)] \end{aligned}$$

and similarly with the other terms. Finally,

$$\begin{aligned} E_{w,b}[z(x)z(y)] &= E[\cos(\langle w, x\rangle)\cos(\langle w, y\rangle) + \sin(\langle w, x\rangle)\sin(\langle w, y\rangle)] \\ &= E[\cos(\langle w, x - y\rangle)]. \end{aligned}$$

To lower the variance of our estimate, we sample several (w, b) pairs and average the resulting values of z.

18.2.1.1 Formal Description

Algorithm `Random Fourier Features`

Input: Training data X, an $n \times d$ matrix representing n data points in dimension d,
γ parameter of Gaussian distribution $e^{-\gamma||x||^2}$
dimension of range space, D,
random normal generator seed

Output: X', an $n \times D$ dimension matrix of data in feature space to be sent to linear solver, as well as the Ω and

1: $\Omega \leftarrow d \times D$ matrix of samples drawn from the standard normal distribution *stdnormal*, then scaled by a factor of $\sqrt{2\gamma}$ to simulate a Gaussian with $\sigma = \sqrt{2\gamma}$ (see fit function from RBF-sampler class of scikit-learn)
2: $\boldsymbol{b} \leftarrow$ vector of length D, each entry a uniform random sample from $[0, 2\pi]$
3: $X' \leftarrow X \cdot \Omega$
4: $X' \leftarrow X' + \boldsymbol{b}$ ($\boldsymbol{b}$ is added to each row)
5: $X' \leftarrow \cos(X')$ (take the cosine of each entry)
6: $X' \leftarrow \sqrt{\frac{2}{D}} \cdot X'$
7: return $X', \Omega, \boldsymbol{b}$

18.2.1.2 Parallelization

Since the random cosine features are generated independently, each coordinate could be computed independently in parallel, as long as each node has access to the distribution p.

18.2.2 Dot product kernels

As in the case of Gaussian kernels, we can use random feature maps to approximate polynomial kernels, $k(x, y) = (\langle x, y\rangle + q)^r$. Again, the embedding takes the following simple form:

$$z : \mathbb{R}^d \to \mathbb{R}^D$$
$$x \mapsto \frac{1}{\sqrt{D}}(z_1(x), \ldots, z_D(x)).$$

The idea here is to choose each random features z_i such that it satisfies $E[z_i(x)z_i(y)] = k(x, y)$. Then the concentration of measure phenomenon, e.g., as D goes to infinity, ensures $|k(x, y) - z(x)^T z(y)|$ to be small with high probability. We describe the process of generating z_i below. Note that it applies to any positive definite kernel $k : (x, y) \mapsto f(\langle x, y\rangle)$, where f admits a Maclaurin expansion, i.e., $f(x) = \sum_{N=0}^{\infty} a_N x^N$, where $a_N = f^{(N)}(0)/N!$. For example, in the case of polynomial kernels $k(x, y) = (\langle x, y\rangle + q)^r$, $f(x)$ will be $(x + q)^r$ and a_N will be the N-th derivative of $(x + q)^r$ divided by $N!$.

i) Randomly pick a number $N \in \mathbb{N} \cup \{0\}$ with $\mathcal{P}[N] = \frac{1}{p^{N+1}}$. Here p is a positive number larger than 1 and in practice $p = 2$ is often a good choice since it establishes a normalized measure over $\mathbb{N} \cup \{0\}$ [44].

ii) Pick N independent Rademacher vector $w_1, ..., w_N$, i.e., each component of Rademacher vector is chosen independently using a fair coin toss from the set $\{-1, 1\}$.

iii) Compute $z_i(x)$ as follows,

$$z_i(x) = \begin{cases} \sqrt{a_0 p}, & \text{if } N = 0, \\ \sqrt{a_N p^{N+1}} \prod_{j=1}^{N} w_j^T x, & \text{otherwise.} \end{cases} \tag{18.2.1}$$

The reason why we pick N from $\mathbb{N} \cup \{0\}$ with probability $\mathcal{P}[N] = \frac{1}{p^{N+1}}$ becomes obvious in the following simple proof. It establishes that the random feature product $z_i(x)z_i(y)$ approximates the kernel product $k(x, y)$ with high probability for any $x, y \in \mathbb{R}^d$, i.e., $E[z_i(x)z_i(y)] = k(x, y)$:

$$\begin{aligned} E[z_i(x)z_i(y)] &= E_N[E_{w_1,\ldots,w_N}[z_i(x)z_i(y)]|N] \\ &= E_N[a_N p^{N+1} E_{w_1,\ldots,w_N}[\prod_{j=1}^{N} w_j^T x \prod_{j=1}^{N} w_j^T y]] \\ &= E_N[a_N p^{N+1}(E_w[w^T x \cdot w^T y])^N] \\ &= E_N[a_N p^{N+1}\langle x, y\rangle^N] \\ &= \sum_{n=0}^{\infty} \frac{1}{p^{n+1}} \cdot a_n p^{n+1}\langle x, y\rangle^n \\ &= f(\langle x, y\rangle) = k(x, y) \end{aligned}$$

See the paper of Kar and Karnick [44] for more details.

18.2.2.1 Formal Description

Algorithm `Random Features for Polynomial Kernels`

Input: Training data X, an $n \times d$ matrix representing n data points in dimension d, Parameters of a polynomial kernel, i.e. degree r, dimension of linear feature space D

Output: X', an $n \times D$ dimension matrix of data in feature space to be sent to linear solver $\Omega, \mathcal{N}$ generated by the algorithm, to be used by the predictor.

1: $\mathcal{N} \leftarrow$ array of D entries, each generated from the exponential probability distribution $\frac{1}{2^{N+1}}$
2: $\Omega \leftarrow$ matrix of dimensions $d \times sum(\mathcal{N}) \cdot D$ with $\{-1, 1\}$ entries, chosen with a fair coin flip
3: $X' \leftarrow X \cdot \Omega$, now a matrix of dimensions $n \times sum(\mathcal{N})$
4: **for** each row in X' **do**
5: $\quad row' \leftarrow$ dividing row into D segments, with ith segment consisting of $\mathcal{N}_i$ number of entries.
6: $\quad row'' \leftarrow$ array of D entries aggregating from each segment of row' according to (18.2.1)
7: $X'' \leftarrow$ the transformed X' divided by $\frac{1}{\sqrt{D}}$
8: return $X'', \Omega, \mathcal{N}$

18.2.2.2 Parallelization

In the above algorithm, Step 3 is done by broadcasting the matrix Ω to each row of X, and computing the matrix-vector product locally in parallel for each row; Similarly, Step 4 can also be distributed since the computations for each row are independent of the others.

18.3 Novelty Detection

Suppose we have training data $x_1, x_2, \ldots x_n \in \mathbb{R}^d$, the goal of novelty detection is to learn a hyperplane in $\mathbb{R}^d$ that separates the training data from the origin with maximum margin. We model this as a one-class classification problem by transforming the training data to $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R}^d \times \{1\}$, indicating that the dependent variable of each training instance is assumed to be the same. The origin is treated as a negative class data point and all input data points are treated as part of the positive class. Given such a mapping, we use the SVM classification mechanisms detailed in Sections 18.1 and 18.2 to learn a one-class classification model.

19 Graph

Authors Orhan Kislal, Nandish Jayaram, Rashmi Raghu, Jingyi Mei, Nikhil Kak

History
- **v0.1** Initial version, SSSP only.
- **v0.2** Graph Framework, SSSP implementation details.
- **v0.3** PageRank
- **v0.4** APSP
- **v0.5** Weakly Connected Components
- **v0.6** Breadth First Search (BFS)
- **v0.7** Hyperlink-Induced Topic Search (HITS)

This module implements various graph algorithms that are used in a number of applications such as social networks, telecommunications and road networks.

19.1 Graph Framework

MADlib graph representation depends on two structures, a *vertex* table and an *edge* table. The vertex table has to have a column of vertex ids. The edge table has to have 2 columns: source vertex id, destination vertex id. For most algorithms an edge weight column is required as well. The representation assumes a directed graph, an edge from x to y does *not* guarantee the existence of an edge from y to x. Both of the tables may have additional columns as required. Multi-edges (multiple edges from a vertex to the same destination) and loops (edge from a vertex to itself) are allowed. This representation does not impose any ordering of vertices or edges. An example graph is given in Figure 19.1 and its representative tables are given in Table 19.1.

19.2 Single Source Shortest Path

Given a graph and a source vertex, single source shortest path (SSSP) algorithm finds a path for every vertex such that the sum of the weights of its constituent edges is minimized.

Shortest path is defined as follows. Let $e_{i,j}$ be the edge from vertex i to vertex j and $w_{i,j}$ be its weight. Given a graph G, the shortest path from s to d is $P = (v_1, v_2 \ldots, v_n)$ (where $v_1 = s$ and $v_n = d$) that over all possible n minimizes the sum $\sum_{i=1}^{n-1} f(e_{i,i+1})$.

Bellman-Ford Algorithm [5, 30] is based on the following idea: We start with a naive approximation for the cost of reaching every vertex. At each iteration, these values are refined based on the edge list and the existing approximations. If there are no refinements at any given step, the algorithm returns the calculated results. If the algorithm does not converge in $|V| - 1$ iterations, this indicates the existence of a negative cycle in the graph.

Algorithm `SSSP`$(V, E, start)$
Input: Vertex set V, edge set E, starting vertex *start*
Output: Distance and parent set for every vertex *cur*

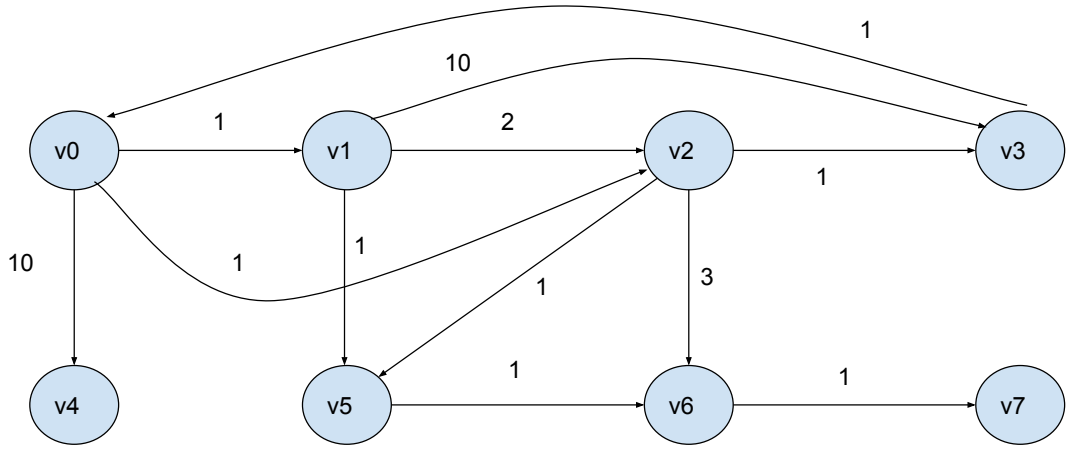

Figure 19.1: A sample graph

vid
0
1
2
3
4
5
6
7

src	dest	weight
0	1	1
0	2	1
0	4	10
1	2	2
1	3	10
1	5	1
2	3	1
2	5	1
2	6	3
3	0	1
5	6	1
6	7	1

Table 19.1: Graph representation of vertices (left) and edges(right) in the database

1: $toupdate(0) \leftarrow (start, 0, start)$
2: **for** every $i \in 0 \dots |V| - 1$ **do**
3: **for** every tuple $t \in toupdate(i)$ **do**
4: **for** every edge $e \mid e.src = t.id$ **do**
5: $local \leftarrow e.val + t.val$
6: **if** $local < toupdate(i + 1, e.dest).val$ **then**
7: $toupdate(i + 1, dest) \leftarrow (local, e.src)$
8: **for** every tuple $t \in toupdate(i + 1)$ **do**
9: **if** $t.val < cur(t.id).val$ **then**
10: $cur(t.id) \leftarrow (t.val, t.parent)$

edge: $(src, dest, val)$. The edges of the graph.

cur: $id \rightarrow (val, parent)$. The intermediate SSSP results.

toupdate: $iter \rightarrow (id \rightarrow (val, parent))$. The set of updates.

Changes from the standard Bellman-Ford algorithm:

Line 3: We only check the vertices that have been updated in the last iteration.

Line 6: At each iteration, we update a given vertex only one time. This means the toupdate set cannot contain multiple records for the same vertex which requires the comparison with the existing value.

This is not a 1-to-1 pseudocode for the implementation since we don't compare the 'toupdate' table records one by one but calculate the overall minimum. In addition, the comparison with 'cur' values take place earlier to reduce the number of tuples in the 'toupdate' table.

19.2.1 Implementation Details

In this section, we discuss the MADlib implementation of the SSSP algorithm in depth.

Algorithm `SSSP`$(V, E, start)$

1: **repeat**
2: Find Updates
3: Apply updates to the output table
4: **until** There are no updates

The implementation consists of two SQL blocks that are called sequentially inside a loop. We will follow the example graph at Figure 19.1 with the starting point as v_0. The very first update on the output table is the source vertex. Its weight is 0 and its parent is itself (v_0). After this initialization step, the loop starts with Find Updates (the individual updates will be represented with <dest,value,parent> format). Looking at the example, it is clear that the updates should be <1,1,0>, <2,1,0> and <4,10,0>. We will assume this iteration is already completed and look how the next iteration of the algorithm works to explain the implementation details.

Algorithm `Find Updates`$(E, old\_update, out\_table)$

```
1:  INSERT INTO new_update
2:      SELECT DISTINCT ON (y.id) y.id AS id,
3:          y.val AS val,
4:          y.parent AS parent
5:      FROM out_table INNER JOIN (
6:              SELECT edge_table.dest AS id, x.val AS val, old_update.id AS parent
7:              FROM old_update
8:                  INNER JOIN edge_table
9:                  ON (edge_table.src = old_update.id)
10:                 INNER JOIN (
11:                     SELECT edge_table.dest AS id,
12:                         min(old_update.val + edge_table.weight) AS val
13:                     FROM old_update INNER JOIN
14:                         edge_table AS edge_table ON
15:                         (edge_table.src=old_update.id)
16:                     GROUP BY edge_table.dest
17:                 ) x
18:                 ON (edge_table.dest = x.id)
19:             WHERE ABS(old_update.val + edge_table.weight - x.val) < EPSILON
20:         ) AS y ON (y.id = out_table.vertex_id)
21:     WHERE y.val<out_table.weight
```

The Find Updates query is constructed in 4 levels of subqueries: *find values, find parents, eliminate duplicates and ensure improvement.*

- We begin our analysis at the innermost subquery, emphfind values (lines 11-16). This subquery takes a set of vertices (in the table *old_update*) and finds the reachable vertices. In case a vertex is reachable by multiple vertices, only the path that has the minimum cost is considered (hence the name find values). There are two important points to note:
  - The input vertices need the value of their path as well.
    - In our example, both v_1 and v_2 can reach v_3. We would have to use $v_2 \rightarrow v_3$ edge since that gives the lowest possible path value.
  - The subquery is aggregating the rows using the *min* operator for each destination vertex and unable to return the source vertex at the same time to use as the parent value.
    - We know the value of v_3 should be 2 but we cannot know its parent (v_2) at the same time.
- The *find parents* subquery is designed to solve the aforementioned limitation. We combine the result of *find values* with *edge* and *old_update* tables (lines 7-10) and get the rows that has the same minimum value.
  - Note that, we would have to tackle the problem of tie-breaking.
    - Vertex v_5 has two paths leading into: <5,2,1> and <5,2,2>. The inner subquery will return <5,2> and it will match both of these edges.
  - It is redundant to keep both of them in the update list as that would require updating the same vertex multiple times in a given iteration.
- At this level, we employ the *eliminate duplicates* subquery. By using the $DISTINCT$ clause at line 2, we allow the underlying system to accept only a single one of them.
- Finally, we introduce the *ensure improvement* subquery to make sure these updates are actually leading us to shortest paths. Line 21 ensures that the values stored in the *out_table* does not increase and the solution does not regress throughout the iterations.

Applying updates is straightforward as the values and the associated parent values are replaced using the *new_update* table. After this operation is completed the *new_update* table becomes *old_update* for the next iteration of the algorithm.

Please note that, for ideal performance, *vertex* and *edge* tables should be distributed on *vertex id* and *source id* respectively.

19.3 All Pairs Shortest Paths

Given a graph and a source vertex, all pairs shortest paths (APSP) algorithm finds a path for every vertex pair such that the sum of the weights of its constituent edges is minimized. Please refer to the Section 19.2 on single source shortest path for the mathematical definition of shortest path.

Our implementation has a dynamic programming approach, based on the matrix multiplication inspired APSP algorithm [62]. The idea is similar to the one from SSSP implementation. We start with a naive approximation for the cost of every vertex pair (infinite). At each iteration, these values are refined based on the edge list and the existing approximations. This refinement step is similar to a matrix multiplication. For every vertex pair i, j, we check every edge $e : j \rightarrow k$ to see

if it is possible to use e to reduce the cost of path $i \to k$. If there are no refinements at any given step, the algorithm returns the calculated results. If the algorithm does not converge in $|V| - 1$ iterations, this indicates the existence of a negative cycle in the graph.

Algorithm APSP(V, E)
Input: Vertex set v, edge set E
Output: Distance and parent set for every vertex pair

```
1: while update is True do
2:     update ← False
3:     for every vertex pair i, j do
4:         for every edge j → k do
5:             if val(i → j) + val(j → k) < val(i → k) then
6:                 val(i → k) ← val(i → j) + val(j → k)
7:                 parent(i → k) ← j
8:                 update ← True
```

19.3.1 Implementation Details

The implementation details are similar to the SSSP as the requirements and restrictions such as finding the parent, distinct updates, etc. are common in both cases. This section will mostly focus on the differences in the APSP implementation.

Algorithm Find Updates(E, out)

```
1:  INSERT INTO update
2:      SELECT DISTINCT ON (y.src, y.dest) y.src AS src, y.dest AS dest
3:          y.val AS val,
4:          y.parent AS parent
5:      FROM out INNER JOIN (
6:              SELECT
7:                  x.src AS src, x.dest AS dest,
8:                  x.val AS val, out.dest AS parent
9:              FROM out
10:                 INNER JOIN edge_table
11:                 ON (edge_table.src = out.dest)
12:                 INNER JOIN (
13:                     SELECT out.src AS src, edge_table.dest AS dest,
14:                         min(out.val + edge_table.weight) AS val
15:                     FROM out INNER JOIN
16:                         edge_table ON
17:                         (edge_table.src=out.dest)
18:                     GROUP BY out.src, edge_table.dest
19:                 ) x
20:                 ON (edge_table.src = x.src AND edge_table.dest = x.dest)
21:             WHERE ABS(out.val + edge_table.weight - x.val) < EPSILON
22:         ) AS y ON (y.src = out.src AND y.dest = out.dest)
23:     WHERE y.val < out.val
```

The only major difference comes in the innermost subquery (lines 13-18). The *group by* clause ensures that we try to reduce the weight for every *out.src* (i) and *edge_table.dest* (k) pair. The *inner join on* clause ensures that there is a connecting edge ($j \to k$) that can be used for the i, j pair. The rest of the changes are mostly trivial as the algorithm needs to check for both source and destination during joins (instead of just the destination).

19.4 PageRank

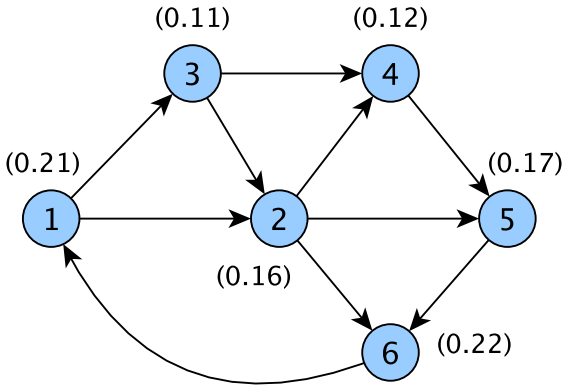

Figure 19.2: An example graph for PageRank

PageRank is a link analysis algorithm that assigns a score to every vertex measuring the relative importance of vertices within the set of all vertices. PageRank [16] was first used by Google to measure the importance of website pages where the World Wide Web was modeled as a directed graph. Figure 19.2 shows an example graph with the PageRank value of each vertex. The intuition behind the algorithm is that the number and quality of links to a vertex determine the authoritativeness of the vertex, which is reflected in the PageRank scores as shown in the figure.

The pagerank module in MADlib implements the model of a random surfer who follows the edges of a graph to traverse it, and jumps to a random vertex after several clicks. The random surfer is modeled using a damping factor that represents the probability with which the surfer will continue to follow links in the graph rather than jumping to a random vertex. MADlib's pagerank module outputs a probability distribution that represents the likelihood that the random surfer arrives at a particular vertex in the graph.

PageRank is an iterative algorithm where the PageRank scores of vertices from the previous iteration are used to compute the new PageRank scores. The PageRank score of a vertex v, at the i^{th} iteration, denoted by $PR(v_i)$ is computed as:

$$PR(v_i) = \frac{1-d}{N} + d \sum_{u \in M(v)} \left(\frac{PR(u_{i-1})}{L(u)}\right) \tag{19.4.1}$$

where N is the number of vertices in the graph, d is the damping factor, $M(v)$ represents the set of vertices that have an edge to vertex v, $L(u)$ represents the out-degree of vertex u, i.e., the number of out-going edges from vertex u, and $PR(u_{i-1})$ represents the PageRank score of vertex u in the $(i-1)^{st}$ iteration.

$\frac{1-d}{N}$ represents the tiny probability with which the surfer would randomly jump to vertex v, rather than arriving at v following links in the graph. This ensures that there is some probability of visiting every vertex in the graph even if they do not have any incoming edges. Note that the PageRank score computed for a vertex v using 19.4.1 in the i^{th} iteration is not updated until the new score is computed for all the vertices in the graph. The computation terminates either when the PageRank score of no vertex changes beyond a threshold across two consecutive iterations, or when a pre-set number of iterations are completed.

Personalized Pagerank: The Personalized Pagerank variant of Pagerank module in MADlib takes an extra argument as a set of user provided vertices. These personalization vertices will have a higher jump probability as compared to other vertices and random surfer is more likely to jump on these personalization vertices. These personalization vertices are initialized with an initial probabilty of $\frac{1}{N}$ where N is the total number of personlaized vertices in the graph and rest of the vertices in the graph are assigned an initial probability of 0. Pagerank calculated for these vertices is biased as a random jump probability is assigned to only these vertices during the pagerank calculation,which is equal to (1 - damping factor).

19.4.1 Implementation Details

In this section, we discuss the MADlib implementation of PageRank in depth. We maintain two tables at every iteration: *previous* and *cur*. The *previous* table maintains the PageRank scores of all vertices computed in the previous iteration, while *cur* maintains the updated scores of all vertices in the current iteration.

Algorithm `PageRank`(V, E)

1: Create *previous* table with a default PageRank score of $\frac{1}{N}$ for every vertex
2: **repeat**
3: Create empty table *cur*.
4: Update *cur* using PageRank scores of vertices in *previous*
5: Update PageRank scores of vertices without incoming edges
6: Drop *previous* and rename *cur* to *previous*
7: **until** PageRank scores have converged or *max* iterations have elapsed

The implementation consists of updating the PageRank scores of all vertices at every iteration, using the PageRank scores of vertices from the previous iteration. The PageRank score of every vertex is initialized to $\frac{1}{N}$ where N is the total number of vertices in the graph. The out-degree of every vertex in the graph (represented by $L(u)$ in eq. 19.4.1), is captured in table *out_cnts*. The following query is used to create and update the PageRank scores in *cur* table using the PageRank scores in *previous* table.

Algorithm `Update PageRank scores`$(previous, out\_cnts, d, N)$

```
1:  CREATE TABLE cur AS
2:      SELECT edge_table.dest AS id,
3:          SUM(previous1.pagerank/out_cnts.cnt)*d + (1-d)/N AS pagerank
4:      FROM edge_table
5:          INNER JOIN previous ON edge_table.dest = previous.id
6:          INNER JOIN out_cnts ON edge_table.src = out_cnts.id
7:          INNER JOIN previous AS previous1 ON edge_table.src = previous1.id
8:      GROUP BY edge_table.dest

10: -- Update PageRank scores of vertices without any incoming edges:
11: INSERT INTO cur
12:     SELECT id, (1-d)/N AS pagerank
13:     FROM previous
14:     WHERE id NOT IN (
15:         SELECT id
16:         FROM cur
17:     )
```

The PageRank computation is terminated either when a fixed number of iterations are completed, or when the PageRank scores of all vertices have converged. The PageRank score of a vertex is deemed converged if the absolute difference in its PageRank scores from *previous* and *cur* is less than a specified threshold. The following query is used to find all the vertices whose PageRank scores have not converged yet.

Algorithm `Update PageRank scores`($previous, cur, threshold$)

```
1: SELECT id
2: FROM cur
3: INNER JOIN previous ON cur.id = previous.id
4: WHERE ABS(previous.pagerank - cur.pagerank) > threshold
```

19.4.2 Best Practices

The pagerank module in MADlib has a few optional parameters: damping factor d, number of iterations max, and the threshold for convergence *threshold*. The default values for these parameters when not specified by the user are 0.85, 100 and $\frac{1}{N*1000}$ respectively.

The damping factor denotes the probability with which the surfer uses the edges to traverse the graph. If set to 0, it implies that the only way a surfer would visit a vertex in the graph is by randomly jumping to it. If set to 1, it implies that the only way the surfer can reach a vertex is by following the edges in the graph, thus precluding the surfer from reaching a vertex that has no incoming edges. It is common practice to set damping factor to 0.85 [16], and the maximum number of iterations to 100. The convergence test for PageRank in MADlib checks for the delta between the PageRank scores of a vertex across two consecutive iterations. Since the initial value of the PageRank score is set to $\frac{1}{N}$, the delta will be small in the initial iterations when N is large (say over 100 million). We thus set the threshold to $\frac{1}{N*1000}$, and it is to be noted that this is not based on any experimental study. Users of MADlib are encouraged to consider this factor when setting a value for threshold, since a high *threshold* value would lead to early termination of PageRank computation, thus resulting in incorrect PageRank values.

19.5 Weakly Connected Components

Given a directed graph G, a weakly connected component is a subgraph G_{sub} of G, such that there exists a path from every vertex in G_{sub} to every other vertex in G_{sub}, ignoring the direction of the edges.

The weakly connected component module implemented in MADlib is based on GRAIL [27]. All vertices are initialized with their own vertex ID as the component ID, and are considered to be active. In every iteration, each active vertex's component ID is updated with the smallest component ID value of all its neighbors. Any vertex whose component ID is not updated in the current iteration is deemed as an inactive vertex for the next iteration. Execution continues until there are no active vertices left. Since each vertex is initialized with its own ID as the component ID, and updated based on neighboring nodes' component IDs, the final component ID of a component will be equal to the smallest vertex ID in the corresponding subgraph. Figure 19.3 shows an example directed graph with two disconnected subgraphs. The subgraph containing vertices 1, 2, 3, 4, 5 and 6 forms a weakly connected component, and is assigned component ID 1, while the subgraph containing vertices 12, 14, 21 and 23 forms the second component and is assigned component ID 12.

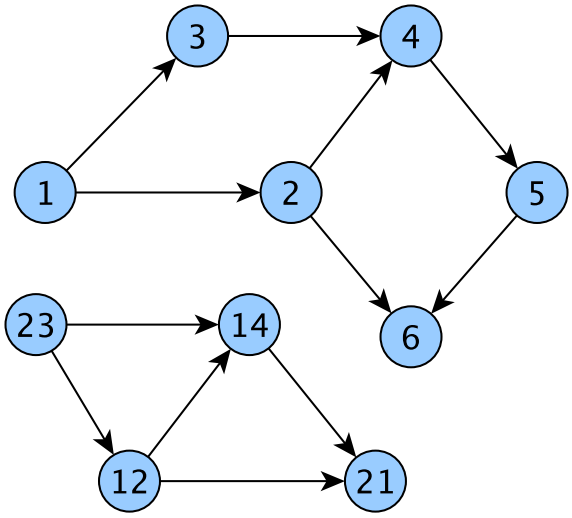

Figure 19.3: An example disconnected directed graph

19.5.1 Implementation Details

In this section, we discuss the MADlib implementation of weakly connected components in depth. We maintain the following tables at every iteration: *oldupdate*, *message* and *newupdate*. In *newupdate*, the component ID of each vertex is initialized to infinity, while the component ID of vertices in the *message* table is initialized to their corresponding vertex ID.

Algorithm `Weakly Connected Components`(V, E)

1: Create *newupdate* table with a default component ID of $infinity$ for every vertex
2: Create *message* table with a default component ID of the corresponding *id* (vertex ID) for every vertex
3: **repeat**
4: Update the *oldupdate* table
5: Update *toupdate* table with active vertices
6: Update the *newupdate* table
7: Update *message* table with potential new component IDs for each vertex
8: **until** There are no active vertices in *toupdate* table

The *message* table contains the component IDs associated with all its immediate neighbors. At each iteration, *oldupdate* is updated with the minimum of all the associated component IDs found for a vertex in *message*.

Algorithm `Update oldupdate table`

```
1: SELECT id, MIN(message.component_id) as component_id
2: FROM message
3: GROUP BY id
```

Table *toupdate* records all vertices whose component IDs must be updated, and are thus marked active.

Algorithm `Update toupdate table with active vertices`

```
1:  -- Find vertices whose component ID must be updated
2:  CREATE TABLE toupdate AS
3:  SELECT id, component_id
4:  FROM oldupdate, newupdate
5:  WHERE oldupdate.id = newupdate.id AND
6:          oldupdate.component_id < newupdate.component_id

8:  -- Update the component IDs
9:  UPDATE newupdate SET
10: component_id = toupdate.component_id
11: FROM toupdate
12: WHERE newupdate.id = toupdate.id
```

Finally, the *message* table is updated with potential new component IDs for active vertices using the following query:

Algorithm `Update message table`$(toupdate, edge)$

```
1:  CREATE TEMP TABLE message AS
2:  SELECT id, MIN(component_id) AS component_id
3:  FROM (
4:      SELECT edge.src AS id,
5:          toupdate.component_id
6:      FROM toupdate, edge
7:      WHERE edge.dest = toupdate.id
8:      UNION ALL
9:      SELECT edge.dest AS id,
10:         toupdate.component_id
11:     FROM toupdate, edge
12:     WHERE edge.src = toupdate.id
13: ) AS t
14: GROUP BY id
```

At the end of the computation, *newupdate* will have the component ID associated with each vertex in G. The component ID of all the vertices in a component is equal to the smallest vertex ID in the corresponding subgraph.

19.5.2 Edge Table Duplication

The queries explained in the Section 19.5.1 expose a potential performance drawback in Greenplum systems. In general, we advise that the edge tables should be distributed by their source columns. However, in WCC, we use both source and destination columns of the edge table in JOIN clauses. In addition, we employ a GROUP BY clause using the column that did not serve as the join key. Algorithm `Update message table` shows that when *dest* is used for the JOIN clause, *src* is renamed to *id* to be used for GROUP BY and vice versa. This query forces multiple redistribute motions in the database which might cause performance degradation. To address this issue, we create a duplicate of the edge table and distribute on the destination column (only for Greenplum systems).

19.6 Breadth-first Search

Given a graph G and a user-specified origin vertex *src*, this algorithm searches and discovers connected nodes in a graph in breadth-first order [12]. The graph can be treated as either directed

or undirected based on a parameter specified when calling the function. There is also a parameter to control the number of hops (edges) to traverse from the source vertex. If not specified, all nodes accessible from the source node will be discovered.

19.6.1 Implementation Details

Algorithm `Breadth First Search`(V, E, src)

1: Set $dist \leftarrow 0$
2: Create *message* table with *src* vertex, $NULL$ parent, and *dist*
3: Copy *message* table to output table *out*
4: **repeat**
5: Create *toupdate* table using *out* and *message* tables
6: $dist \leftarrow dist + 1$
7: Update *message* table with newly found candidate vertices, parent and *dist*
8: Copy *message* table to *out*
9: **until** There are no candidate vertices remaining in *message* table

The implementation details are similar to SSSP, albeit simpler. We only have to track the number of hops and not the sum of weights, but other requirements and restrictions such as finding the parent, distinct updates, etc. are common in both cases. The output table is initialized only to the *src* vertex to begin with. A *message* table is also maintained that contains the list of vertices to traverse and explore in the next iteration, which is also initialized with the *src* vertex. BFS then runs iteratively until no more candidate vertices remain in the *message* table, as outlined in `Breadth First Search`.

At every iteration, *toupdate* table is updated with vertices that are neighbors of vertices in the *message* table, that are not already visited in the past (one scan of the *out* table reveals all the vertices that have already been visited in the past). All such newly found neighboring vertices in the current iteration will have one or more parents, based on how many vertices in the *message* table have a direct edge to them. Each such vertex in the *message* table is marked as the parent of such newly found neighboring vertices in the *toupdate* table.

The *message* table is then cleared and updated with the contents of *toupdate* table, except that for each new neighboring vertex considered, only one of the parents is recorded as its parent (the node with the smallest id among all parent nodes). The content of this updated *message* is then copied to the *out* table, and this process continues until *message* table is empty.

19.7 HITS

Hyperlink-Induced Topic Search (HITS) [45] developed by Jon Kleinberg is a link analysis algorithm that rates Web pages. The idea behind the algorithm is to assign Hub and Authority scores to all vertices. Authorities are analagous to web pages that have good authoritative content and get linked by other web pages. Hubs can be thought of as large directories that themselves do not hold any authoritative content but provide direct links to other authoritative pages.

HITS is an iterative algorithm where the Hub and Authority scores of vertices from the previous iteration are used to compute the new scores. The Hub and Authority scores of a vertex v, at the

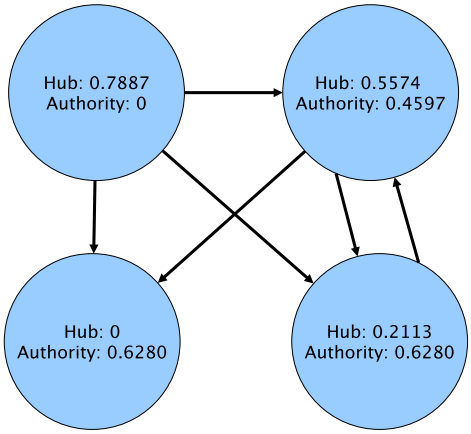

Figure 19.4: An example graph for HITS showing normalized hub and authority scores

i^{th} iteration, denoted by $HUB(v_i)$ and $AUTHORITY(v_i)$ is computed as:

$$\begin{aligned} AUTHORITY(v_i) &= \sum_{u \in M(v)} (HUB(u_{i-1})) \\ HUB(v_i) &= \sum_{u \in M(v)} (AUTHORITY(v_i)) \end{aligned} \tag{19.7.1}$$

where N is the number of vertices in the graph, $M(v)$ represents the set of vertices that have an edge to vertex v, and $HUB(u_{i-1})$ and $AUTHORITY(v_i)$ represent the Hub score of vertex u in the $(i-1)^{th}$ iteration and Authority score of vertex v in the $(i)^{th}$ iteration.

19.7.1 Implementation Details

In this section, we discuss the MADlib implementation of HITS in depth. We maintain two tables at every iteration: *message* and *cur*. The *cur* table maintains the Hub and Authority scores of all vertices computed in the previous iteration, while *message* maintains the updated scores of all vertices in the current iteration.

Algorithm `HITS`(V, E)

1: Create *cur* table with a default Hub and Authority score of 1 for every vertex
2: **repeat**
3: Create empty table *message*.
4: Update Authority score in *message* using Hub scores of vertices in *cur*
5: Update Hub score in *message* using Authority scores of vertices in *message*
6: Normalize Hub and Authority scores in *message* using L2 normalization
7: Rename *message* to *cur*
8: **until** both Authority and Hub scores have converged or *max* iterations have elapsed

The following query is used to create and update the Hub and Authority scores in *message* table using the Hub scores in *cur* table.

Algorithm `Update Hub and Authority scores`$(cur, edge\_table)$

```
1: -- Create message table and update authority scores
2: CREATE TABLE message AS
```

```
    SELECT cur.id AS id,
        COALESCE(SUM(curalias.hub), 0.0) AS authority,
    cur.hub AS hub
    FROM cur
        LEFT JOIN edge_table ON cur.id = edge_table.dest
        LEFT JOIN cur AS curalias ON curalias.id = edge_table.dest
    GROUP BY cur.id, cur.hub
    ORDER BY cur.id

-- Update hub scores in message table:
UPDATE message
    SET hub = subquery.hub FROM
        (
        SELECT  message.id AS id, COALESCE(SUM(msgalias.authority), 0) AS hub
        FROM message
        LEFT JOIN edge_table ON message.id = edge_table.src
        LEFT JOIN message AS msgalias ON message.id = edge_table.dest
        GROUP BY  message.id
        ) AS subquery
    WHERE subquery.id = message.id
```

The Hub and Authority computations are terminated either when a fixed number of iterations are completed, or when both the Hub and Authority scores of all vertices have converged. The Hub/Authority score of a vertex is deemed converged if the absolute difference in its Hub/Authority scores from *cur* and *message* are less than a specified threshold. The following query is used to find all the vertices whose Hub/Authority scores have not converged yet.

Algorithm `Check for Hub and Authority convergence`$(cur, message, threshold)$

```
        SELECT DISTINCT cur.id FROM message
        INNER JOIN cur ON cur.id=message.id
        WHERE ABS(cur.authority-message.authority) > threshold
        OR ABS(cur.hub-message.hub) > threshold
```

19.7.2 Best Practices

The HITS module in MADlib has a few optional parameters: number of iterations max, and the threshold for convergence *threshold*. The default values for these parameters when not specified by the user are 100 and $\frac{1}{N*1000}$ respectively. It is to be noted that this is not based on any experimental study. Users of MADlib are encouraged to consider this factor when setting a value for threshold, since a high *threshold* value would lead to early termination of computation, thus resulting in incorrect Hub and Authority scores.

20 Neural Network

Authors Xixuan Feng, Cooper Sloan, Rahul Iyer, Nikhil Kak

History
- **v0.1** Initial version
- **v0.2** Added a section for momentum updates. Also updated the mlp-train-iteration algorithm to include momentum calculations.

This module implements artificial neural network [75].

20.1 Multilayer Perceptron

Multilayer perceptron is arguably the most popular model among many neural network models [76]. Here, we learn the coefficients by minimizing a least square objective function, or cross entropy ([8], example 1.5.3). The parallel architecture is based on the paper by Zhiheng Huang [78].

20.1.1 Solving as a Convex Program

Although the objective function is not necessarily convex, gradient descent or incremental gradient descent are still commonly-used algorithms to learn the coefficients. To clarify, gradient-based methods are not different from the famous backpropagation, which is essentially a way to compute the gradient value.

20.1.2 Formal Description

Having the convex programming framework, the main tasks of implementing a learner include: (a) choose a subset of algorithms; (b) implement the related computation logic for the objective function, e.g., gradient.

For multilayer perceptron, we choose incremental gradient descent (IGD). In the remaining part of this section, we will give a formal description of the derivation of objective function and its gradient.

Objective function. We mostly follow the notations in example 1.5.3 from Bertsekas [8], for a multilayer perceptron that has N layers (stages), and the k^{th} stage has n_k activation units ($\phi : \mathbb{R} \to \mathbb{R}$), the objective function for regression is given as

$$f_{(x,y)}(u) = \frac{1}{2}\|h(u,x) - y\|_2^2,$$

and for classification the objective function is given as

$$f_{(x,y)}(u) = \sum_i (\log(h_i(u,x)) * z_i + (1 - \log(h_i(u,x))) * (1 - z_i),$$

where $x \in \mathbb{R}^{n_0}$ is the input vector, $y \in \mathbb{R}^{n_N}$ is the output vector (one hot encoded for classification), [1] and the coefficients are given as

$$u = \{u_{k-1}^{sj} \mid k = 1, ..., N,\ s = 0, ..., n_{k-1},\ j = 1, ..., n_k\},$$

And are initialized from a uniform distribution as follows:

$$u_k^{sj} = uniform(-r, r),$$

where r is defined as follows:

$$r = \sqrt{\frac{6}{n_k + n_{k+1}}}$$

With regularization, an additional term enters the objective function, given as

$$\sum_{u_k^{sj}} \frac{1}{2} \lambda u_k^{sj2}$$

This still leaves $h : \mathbb{R}^{n_0} \to \mathbb{R}^{n_N}$ as an open item. Let $o_k \in \mathbb{R}^{n_k}, k = 1, ..., N$ be the output vector of the k^{th} layer. Then we define $h(u, x) = o_N$, based on setting $o_0 = x$ and the j^{th} component of o_k is given in an iterative fashion as [2]

$$o_k^j = \phi\left(\sum_{s=0}^{n_{k-1}} o_{k-1}^s u_{k-1}^{sj}\right), \quad k = 1, ..., N,\ j = 1, ..., n_k$$

Gradient of the End Layer. Let's first handle $u_{N-1}^{st}, s = 0, ..., n_{N-1}, t = 1, ..., n_N$. Let y^t denote the t^{th} component of $y \in \mathbb{R}^{n_N}$, and h^t the t^{th} component of output of h.

$$\begin{aligned}
\frac{\partial f}{\partial u_{N-1}^{st}} &= \left(h^t(u, x) - y^t\right) \cdot \frac{\partial h^t(u, x)}{\partial u_{N-1}^{st}} \\
&= \left(o_N^t - y^t\right) \cdot \frac{\partial o_N^t}{\partial u_{N-1}^{st}} \\
&= \left(o_N^t - y^t\right) \cdot \frac{\partial \phi\left(\sum_{s=0}^{n_{N-1}} o_{N-1}^s u_{N-1}^{st}\right)}{\partial u_{N-1}^{st}} \\
&= \left(o_N^t - y^t\right) \cdot \phi'\left(\sum_{s=0}^{n_{N-1}} o_{N-1}^s u_{N-1}^{st}\right) \cdot o_{N-1}^s
\end{aligned}$$

To ease the notation, let the input vector of the j^{th} activation unit of the $(k+1)^{th}$ layer be

$$net_k^j = \sum_{s=0}^{n_{k-1}} o_{k-1}^s u_{k-1}^{sj},$$

where $k = 1, ..., N,\ j = 1, ..., n_k$, and note that $o_k^j = \phi(net_k^j)$. Finally, the gradient

$$\frac{\partial f}{\partial u_{N-1}^{st}} = \left(o_N^t - y^t\right) \cdot \phi'(net_N^t) \cdot o_{N-1}^s$$

For any $s = 0, ..., n_{N-1}, t = 1, ..., n_N$, we are given y^t, and $o_N^t, net_N^t, o_{N-1}^s$ can be computed by forward iterating the network layer by layer (also called the feed-forward pass). Therefore, we now know how to compute the coefficients for the end layer $u_{N-1}^{st}, s = 0, ..., n_{N-1}, t = 1, ..., n_N$.

[1] Of course, the objective function can be defined over a set of input-output vector pairs, which is simply given as the addition of the above f.

[2] $o_k^0 \equiv 1$ is used to simplified the notations, and o_k^0 is not a component of o_k, for any $k = 0, ..., N$.

20.1.2.1 Backpropagation

For inner (hidden) layers, it is more difficult to compute the partial derivative over the input of activation units (i.e., $net_k, k = 1, ..., N-1$). That said, $\frac{\partial f}{\partial net_N^t} = (o_N^t - y^t)\phi'(net_N^t)$ is easy, where $t = 1, ..., n_N$, but $\frac{\partial f}{\partial net_k^j}$ is hard, where $k = 1, ..., N-1, j = 1, .., n_k$. This hard-to-compute statistic is referred to as *delta error*, and let $\delta_k^j = \frac{\partial f}{\partial net_k^j}$, where $k = 1, ..., N-1, j = 1, .., n_k$. If this is solved, the gradient can be easily computed as follow

$$\frac{\partial f}{\partial u_{k-1}^{sj}} = \boxed{\frac{\partial f}{\partial net_k^j}} \cdot \frac{\partial net_k^j}{\partial u_{k-1}^{sj}} = \boxed{\delta_k^j} o_{k-1}^s,$$

where $k = 1, ..., N-1, s = 0, ..., n_{k-1}, j = 1, .., n_k$. To solve this, we introduce the popular backpropagation below.

Error Back Propagation. Since we know how to compute $\delta_N^t, t = 1, ..., n_N$, we try to compute $\delta_k^j, j = 1, ..., n_k$, given $\delta_{k+1}^t, t = 1, ..., n_{k+1}$, for any $k = 1, ..., N-1$. First,

$$\delta_k^j = \frac{\partial f}{\partial net_k^j} = \frac{\partial f}{\partial o_k^j} \cdot \frac{\partial o_k^j}{\partial net_k^j} = \frac{\partial f}{\partial o_k^j} \cdot \phi'(net_k^j)$$

$$\frac{\partial f}{\partial o_k^j} = \sum_{t=1}^{n_{k+1}} \left(\frac{\partial f}{\partial net_{k+1}^t} \cdot \frac{\partial net_{k+1}^t}{\partial o_k^j} \right), \quad k = 1, ..., N-1, \ j = 1, ..., n_k$$

Using the above equation, we can solve delta error backward iteratively

$$\begin{aligned}
\delta_k^j &= \frac{\partial f}{\partial o_k^j} \cdot \phi'(net_k^j) \\
&= \sum_{t=1}^{n_{k+1}} \left(\frac{\partial f}{\partial net_{k+1}^t} \cdot \frac{\partial net_{k+1}^t}{\partial o_k^j} \right) \cdot \phi'(net_k^j) \\
&= \sum_{t=1}^{n_{k+1}} \left(\delta_{k+1}^t \cdot \frac{\partial \left(\sum_{s=0}^{n_k} o_k^s u_k^{st} \right)}{\partial o_k^j} \right) \cdot \phi'(net_k^j) \\
&= \sum_{t=1}^{n_{k+1}} \left(\delta_{k+1}^t \cdot u_k^{jt} \right) \cdot \phi'(net_k^j)
\end{aligned}$$

To sum up, we need the following equation for error back propagation

$$\boxed{\delta_k^j = \sum_{t=1}^{n_{k+1}} \left(\delta_{k+1}^t \cdot u_k^{jt} \right) \cdot \phi'(net_k^j)}$$

where $k = 1, ..., N-1$, and $j = 1, ..., n_k$.

Momentum updates. Momentum[42][23] can help accelerate learning and avoid local minima when using gradient descent. We also support nesterov's accelarated gradient due to its look ahead characteristics.
Here we need to introduce two new variables namely velocity and momentum. momentum must be in the range 0 to 1, where 0 means no momentum. The velocity is the same size as the coefficient

and is accumulated in the direction of persistent reduction, which speeds up the optimization. The momentum value is responsible for damping the velocity and is analogous to the coefficient of friction.
In classical momentum we first correct the velocity, and then update the model with that velocity, whereas in Nesterov momentum, we first move the model in the direction of momentum*velocity, then correct the velocity and finally use the updated model to calculate the gradient. The main difference being that in classical momentum, we compute the gradient before updating the model whereas in nesterov we first update the model and then compute the gradient from the updated position.

Classic momentum update

$$v = \mu * v - \eta * \frac{\partial f}{\partial u_{k-1}^{sj}} \quad \text{(velocity update)},$$
$$u = u + v$$

Nesterov momentum update

$$ua = u + \mu * v \quad \text{(nesterov's initial coefficient update to the model)},$$
$$v = \mu * v - \eta * \frac{\partial f}{\partial ua_{k-1}^{sj}} \quad \text{(velocity update, use the lookahead model } ua \text{ for gradient calculations)},$$
$$u = u - \eta * \frac{\partial f}{\partial ua_{k-1}^{sj}}$$

where u is the coefficient vector, v is the velocity vector, μ is the momentum value, η is the learning rate and $\frac{\partial f}{\partial ua_{k-1}^{sj}}$ is the gradient calculated at the updated position ua

20.1.2.2 The *Gradient* Function

Algorithm `mlp-gradient`(u, x, y)
Input: Coefficients $u = \{u_{k-1}^{sj} \mid k = 1, ..., N,\ s = 0, ..., n_{k-1},\ j = 1, ..., n_k\}$,
start vector $x \in \mathbb{R}^{n_0}$,
end vector $y \in \mathbb{R}^{n_N}$,
activation unit $\phi : \mathbb{R} \to \mathbb{R}$
Output: Gradient value $\nabla f(u)$ that consists of components $\nabla f(u)_{k-1}^{sj} = \frac{\partial f}{\partial u_{k-1}^{sj}}$

1: $(net, o) \leftarrow$ `feed-forward`(u, x, ϕ)
2: $\delta_N \leftarrow$ `end-layer-delta-error`(net, o, y, ϕ')
3: $\delta \leftarrow$ `back-propogate`(net, o, y, u, ϕ')
4: **for** $k = 1, ..., N$ **do**
5: **for** $s = 0, ..., n_{k-1}$ **do**
6: **for** $j = 1, ..., n_k$ **do**
7: $\nabla f(u)_{k-1}^{sj} \leftarrow \delta_k^j o_{k-1}^s$ ▷ Can be put together with the computation of delta δ
8: **return** $\nabla f(u)$

Activation Units ϕ. Common examples of activation units are

$$\phi(\xi) = \frac{1}{1+e^{-\xi}}, \qquad \text{(logistic function)},$$
$$\phi(\xi) = \frac{e^{\xi}-e^{-\xi}}{e^{\xi}+e^{-\xi}}, \qquad \text{(hyperbolic tangent function)}$$
$$\phi(\xi) = max(x,0), \qquad \text{(rectified linear function)}$$

Algorithm `feed-forward`(u, x, ϕ)

Input: Coefficients $u = \{u_{k-1}^{sj} \mid k = 1, ..., N,\ s = 0, ..., n_{k-1},\ j = 1, ..., n_k\}$,
input vector $x \in \mathbb{R}^{n_0}$,
activation unit $\phi : \mathbb{R} \to \mathbb{R}$

Output: Input vectors $net = \{net_k^j \mid k = 1, ..., N,\ j = 1, ..., n_k\}$,
output vectors $o = \{o_k^j \mid k = 0, ..., N,\ j = 0, ..., n_k\}$

for $k = 0, ..., N$ **do**
  $o_k^0 \leftarrow 1$
$o_0 \leftarrow x$ ▷ For all components $o_0^j, x^j,\ j = 1, ..., n_0$
for $k = 1, ..., N$ **do**
  for $j = 1, ..., n_k$ **do**
    $net_k^j \leftarrow 0$
    for $s = 0, ..., n_{k-1}$ **do**
      $net_k^j \leftarrow net_k^j + o_{k-1}^s u_{k-1}^{sj}$
    $o_k^j = \phi(net_k^j)$ ▷ Where the activation function for the final layer is identity for regression and softmax for classification.
return (net, o)

Algorithm `back-propogate`$(\delta_N, net, u, \phi')$

Input: input vectors $net = \{net_k^j \mid k = 1, ..., N,\ j = 1, ..., n_k\}$,
output vectors $o = \{o_k^j \mid k = 0, ..., N,\ j = 0, ..., n_k\}$,
end vector $y \in \mathbb{R}^{n_N}$,
coefficients $u = \{u_{k-1}^{sj} \mid k = 1, ..., N,\ s = 0, ..., n_{k-1},\ j = 1, ..., n_k\}$,
derivative of activation unit $\phi' : \mathbb{R} \to \mathbb{R}$

Output: Delta $\delta = \{\delta_k^j \mid k = 1, ..., N,\ j = 1, ..., n_k\}$

for $t = 1, ..., n_N$ **do**
  $\delta_N^t \leftarrow (o_N^t - y^t)$ ▷ This applies for identity activation and mean square error loss and softmax activation with cross entropy loss
for $k = N-1, ..., 1$ **do**
  for $j = 0, ..., n_k$ **do**
    $\delta_k^j \leftarrow 0$
    for $t = 1, ..., n_{k+1}$ **do**
      $\delta_k^j \leftarrow \delta_k^j + \delta_{k+1}^t u_k^{jt}$
    $\delta_k^j \leftarrow \delta_k^j \phi'(net_k^j)$
return δ

Algorithm `mlp-train-iteration`(X, Y, η, μ, n)

Input: start vectors $X_{i...m} \in \mathbb{R}^{n_0}$,
end vectors $Y_{i...m} \in \mathbb{R}^{n_N}$,
learning rate η,
momentum μ,
nesterov flag n

Output: Coefficients $u = \{u_{k-1}^{sj} \mid k = 1, ..., N,\ s = 0, ..., n_{k-1},\ j = 1, ..., n_k\}$

`Randomnly initialize u`
`Initialize velocity v to 0`
for $i = 1, ..., m$ **do**
    if n == True **then**
        $u \leftarrow u + \mu * v$ (nesterov's initial coefficient update)
    $\nabla f(u) \leftarrow$ `mlp-gradient`(u, X_i, Y_i)
    $v \leftarrow \mu * v - (\eta \nabla f(u) u + \lambda u)$ ▷ (nesterov's initial coefficient update)
    if $\mu == 0$ and n == False **then**
        $u \leftarrow u + v$ ▷ (classic(non-nesterov) momentum update)
    else
        $u \leftarrow u - (\eta \nabla f(u) u + \lambda u)$
return u

Algorithm `mlp-train-parallel`(X, Y, η, s, t)

Input: start vectors $X_{i...m} \in \mathbb{R}^{n_0}$,
end vectors $Y_{i...m} \in \mathbb{R}^{n_N}$,
learning rate η,
segments s,
iterations t,

Output: Coefficients $u = \{u_{k-1}^{sj} \mid k = 1, ..., N,\ s = 0, ..., n_{k-1},\ j = 1, ..., n_k\}$

`Randomnly initialize u`
for $j = 1, ..., s$ **do**
    $X_j \leftarrow$ `subset-of-X`
    $Y_j \leftarrow$ `subset-of-Y`
for $i = 1, ..., t$ **do**
    for $j = 1, ..., s$ **do**
        $u_j \leftarrow copy(u)$
        $u_j \leftarrow$ `mlp-train-iteration`(X_j, Y_j, η)
    $u \leftarrow$ `weighted-avg`$(u_{1...s})$
return u

!TEX root = ../design.tex

21 k Nearest Neighbors

Authors Orhan Kislal

History **v0.1** Initial version: knn and kd-tree.

21.1 Introduction

Some notes and figures in this section are borrowed from [1] and [41].

K-nearest neighbors (KNN) is one of the most commonly used learning algorithms. The goal of knn is to find a number (k) of training data points closest to the test point. These neighbors can be used to predict labels via classification or regression.

KNN does not have a training phase like the most of learning techniques. It does not create a model to generalize the data, instead the algorithm uses the whole training dataset (or a specific subset of it).

KNN can be used for classification, the output is a class membership (a discrete value). An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors. It can also be used for regression, output is the value for the object (predicts continuous values). This value is the average (or median) of the values of its k nearest neighbors.

21.2 Implementation Details

The basic KNN implementation depends on the table join between the training dataset and the test dataset.

```
1: (SELECT test_id,
2:         train_id,
3:         fn_dist(train_col_name, test_col_name) AS dist,
4:         label
5: FROM train_table, test_table) AS knn_sub
```

Once we have the distance between every train - test pair, the algorithm picks the k smallest values.

```
1: SELECT row_number() OVER
2:     (PARTITION BY test_id ORDER BY dist) AS r,
3:     test_id,
4:     train_id,
5:     label
6: FROM knn_sub
7: WHERE r <= k
```

Finally, the prediction is completed based on the labels of the selected training points for each test point.

21.3 Enabling KD-tree

One of the major shortcomings of KNN is the fact that it is computationally expensive. In addition, there is no training phase; which means every single prediction will have to compute the full table join. One of the ways to improve the performance is to reduce the search space for test points. Kd-tree option is developed to enable trading the accuracy of the output with higher performance by reducing the neighbor search space.

Kd-trees are used for organizing data in k dimensions. It is constructed like a binary search tree where each level of the tree is using a specific dimension for finding splits.

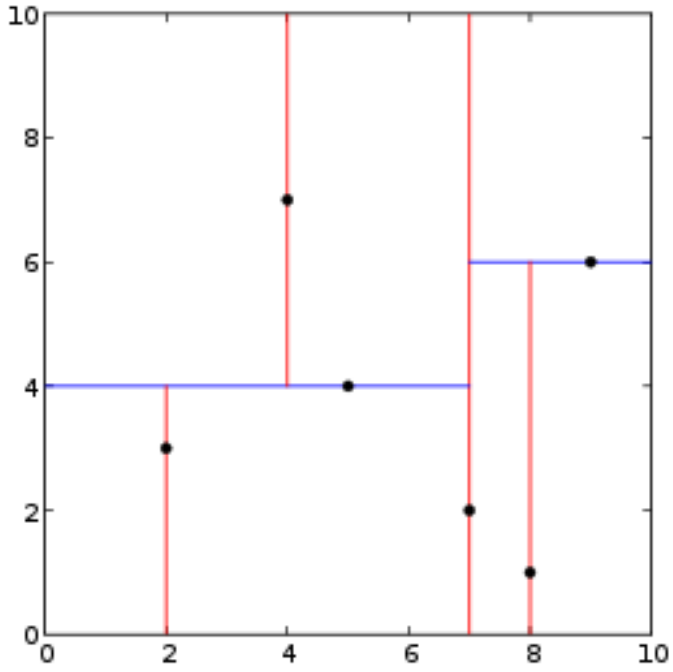

Figure 21.1: A 2D kd-tree of depth 3

A kd-tree is constructed by finding the median value of the data in a particular dimension and separating the data into two sections based on this value. This process is repeated a number of times to construct smaller regions. Once the kd-tree is prepared, it can be used by any test point to find its assigned region and this fragmentation can be used for limiting the search space for nearest neighbors.

Once we have the kd-tree regions and their borders, we find the associated regions for the test points. This gives us the first region to search for nearest neighbors. In addition, we allow the user to request for multiple regions to search. This means we have to decide which additional regions to include in our search. We implemented a backtracking algorithm to find these regions. The core idea is to find the closest border for each test point and select the region on the other side of the border. Note that points that reside in the same region might have different secondary (or tertiary, etc.) regions. Consider the tree at Figure 21.1. A test point at $< 5, 2 >$ is in the same region as $< 3, 3.9 >$. However, their closest borders and the associated secondary regions are wildly different. In addition, consider $< 3, 3.9 >$ and $< 6, 3.9 >$. They both have the same border as their closest

one ($y = 4$). However, their closest regions do differ. To make sure that we get the correct region, the following scheme is implemented. For a given point P, we find the closest border, $dim[i] = x$ and P's relative position to it ($pos = -1$ for lower and $+1$ for higher). We conjure a new point that consists of the same values as the test point in every dimension except i. For $dim[i]$, we set the value to $x - pos * \epsilon$. Finally, we use the existing kd-tree to find this new point's assigned region. This region is our expansion target for the point P. We repeat this process with the next closest border as requested by the user.

The knn algorithm does not change significantly with the addition of regions. Assuming that the training and test datasets have their region information stored in the tables, the only necessary change is ensuring that the table join uses these region ids to limit the search space.

```
1: (SELECT test_id,
2:         train_id,
3:         fn_dist(train_col_name, test_col_name) AS dist,
4:         label
5: FROM train_table, test_table
6: WHERE train_table.region_id = test_table.region_id
7: ) AS knn_sub
```

Bibliography

[1] *A quick introduction to k nearest neighbors algorithm.* URL: https://medium.com/@adi.bronshtein/a-quick-introduction-to-k-nearest-neighbors-algorithm-62214cea29c7.

[2] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. "NP-hardness of Euclidean sum-of-squares clustering." In: *Machine Learning* 75 (2009), pp. 245–248. DOI: 10.1007/s10994-009-5103-0.

[3] D. Arthur and S. Vassilvitskii. "k-means++: the advantages of careful seeding." In: *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'07)*. 2007, pp. 1027–1035. ACM: 1283494.

[4] D. Arthur, B. Manthey, and H. Röglin. *k-Means has Polynomial Smoothed Complexity*. 2009. arXiv:0904.1113 [cs.DS].

[5] R. Bellman. "On a routing problem." In: *Quarterly of applied mathematics* (1958), pp. 87–90.

[6] J. Bennett and S. Lanning. "The Netflix Prize." In: *KDD Cup and Workshop*. 2007.

[7] D. Bertsekas. "Incremental proximal methods for large scale convex optimization." In: *Mathematical Programming* 129 (2 2011), pp. 163–195. ISSN: 0025-5610. DOI: 10.1007/s10107-011-0472-0.

[8] D. Bertsekas. *Nonlinear programming*. Athena scientific optimization and computation series. Athena Scientific, 1999. ISBN: 9781886529007. URL: http://books.google.com/books?id=TgMpAQAAMAAJ.

[9] D. M. Blei, A. Y. Ng, and M. I. Jordan. "Latent dirichlet allocation." In: *J. Mach. Learn. Res.* 3 (Mar. 2003), pp. 993–1022. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=944919.944937.

[10] L. Borsi, M. Lickes, and L. Soldo. "The stratified Cox Procedure." In: 27 (2011).

[11] L. Bottou and O. Bousquet. "The Tradeoffs of Large Scale Learning." In: *NIPS*. Ed. by J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis. Curran Associates, Inc., 2007.

[12] *Breadth-first search*. URL: https://en.wikipedia.org/wiki/Breadth-first_search.

[13] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984.

[14] L. Breiman and A. Cutler. *Random Forests*. URL: http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm.

[15] L. Breiman, A. Cutler, A. Liaw, and M. Wiener. *randomForest: Breiman and Cutler's random forests for classification and regression*. URL: http://cran.r-project.org/web/packages/randomForest/index.html.

[16] S. Brin and L. Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine." In: *Seventh International World-Wide Web Conference (WWW)*. 1998.

[17] J. Bruin. *Supplemental notes to Applied Survival Analysis Applied Survival Analysis*. 2011. URL: http://www.ats.ucla.edu/stat/examples/asa/test_proportionality.htm.

[18] A. Cameron and P. Trivedi. *Microeconometrics using Stata*. Stata Press, 2009. ISBN: 9781597180481.

[19] E. J. Candès and B. Recht. "Exact matrix completion via convex optimization." In: *Commun. ACM* 55.6 (2012), pp. 111–119.

[20] M.-T. Chao. "A general purpose unequal probability sampling plan." In: *Biometrika* 69.3 (1982), pp. 653–656. DOI: 10.1093/biomet/69.3.653.

[21] F. Chen, X. Feng, C. Re, and M. Wang. "Optimizing statistical information extraction programs over evolving text." In: *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE. 2012, pp. 870–881.

[22] D. Cox. "Regression models and life-tables." In: *Journal of the Royal Statistical Society. Series B (Methodological)* 34.2 (1972), pp. 187–220.

[23] *CS231n Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.github.io/neural-networks-3/#sgd.

[24] S. J. DeRose. "Grammatical Category Disambiguation by Statistical Optimization." In: *Computational Linguistics* 14.1 (1988), pp. 31–39.

[25] A. Diekmann and B. Jann. "Regression Models for Categorical Dependent Variables." In: (2008).

[26] J. C. Duchi, A. Agarwal, and M. J. Wainwright. "Distributed Dual Averaging In Networks." In: *NIPS*. Ed. by J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta. Curran Associates, Inc., 2010, pp. 550–558.

[27] J. Fan, A. G. S. Raj, and J. M. Patel. "The Case Against Specialized Graph Analytics Engines." In: *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. 2015.

[28] W. Feller. *An Introduction to Probability Theory and Its Applications*. 3rd. Wiley, 1968.

[29] X. Feng, A. Kumar, B. Recht, and C. Ré. "Towards a unified architecture for in-RDBMS analytics." In: *SIGMOD Conference*. Ed. by K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman. ACM, 2012, pp. 325–336. ISBN: 978-1-4503-1247-9.

[30] L. R. Ford Jr. *Network flow theory*. Tech. rep. DTIC Document, 1956.

[31] J. Fox. *Applied regression analysis and generalized linear models*. Sage Publications, 2008.

[32] J. Fox. *Cox Proportional Hazards Regression for Survival Data: Appendix to An R and S-PLUS Companion to Applied Regression*. 2002.

[33] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. "Large-scale matrix factorization with distributed stochastic gradient descent." In: *KDD*. Ed. by C. Apté, J. Ghosh, and P. Smyth. ACM, 2011, pp. 69–77. ISBN: 978-1-4503-0813-7.

[34] P. M. Grambsch and T. M. Therneau. "Proportional hazards tests and diagnostics based on weighted residuals." In: *Biometrika* 81.3 (1994), pp. 515–526.

[35] T. L. Griffiths and M. Steyvers. "Finding Scientific Topics." In: *PNAS* 101.suppl. 1 (2004), pp. 5228–5235.

[36] G. Guennebaud, B. Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org.

[37] W. W. Hager and H. Zhang. "A survey of nonlinear conjugate gradient methods." In: *Pacific journal of Optimization* 2.1 (2006), pp. 35–58.

[38] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2008.

[39] M. R. Hestenes and E. Stiefel. *Methods of conjugate gradients for solving linear systems*. 1952.

[40] D. W. Hosmer Jr, S. Lemeshow, and S. May. *Applied survival analysis: regression modeling of time to event data*. Vol. 618. Wiley. com, 2011.

[41] *How to use a KdTree to search*. URL: http://pointclouds.org/documentation/tutorials/kdtree_search.php.

[42] Ilya Sutskever. *TRAINING RECURRENT NEURAL NETWORKS*. URL: http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf.

[43] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning*. Springer, 2013.

[44] P. Kar and H. Karnick. "Random Feature Maps for Dot Product Kernels." In: 2012.

[45] J. M. Kleinberg. "Authoritative Sources in a Hyperlinked Environment." In: *J. ACM* 46.5 (Sept. 1999), pp. 604–632. DOI: 10.1145/324133.324140.

[46] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data." In: *ICML*. 2001, pp. 282–289.

[47] D. Lin and L.-J. Wei. "The robust inference for the Cox proportional hazards model." In: *Journal of the American Statistical Association* 84.408 (1989), pp. 1074–1078.

[48] S. Lloyd. "Least squares quantization in PCM." In: *IEEE Transactions on Information Theory* 28.2 (Mar. 1982). Technical Report appeared much earlier in: *Bell Telephone Laboratories Paper* (1957), pp. 129–137. DOI: 10.1109/TIT.1982.1056489.

[49] M. Mahajan, P. Nimbhorkar, and K. Varadarajan. "The planar k-means problem is NP-hard." In: *Theoretical Computer Science* (June 2010). In Press. DOI: 10.1016/j.tcs.2010.05.034.

[50] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. URL: http://nlp.stanford.edu/IR-book/.

[51] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz. "Building a Large Annotated Corpus of English: The Penn Treebank." In: *Computational Linguistics* 19.2 (1993), pp. 313–330.

[52] A. McCallum, K. Nigam, and L. H. Ungar. "Efficient clustering of high-dimensional data sets with application to reference matching." In: *Proceedings of the 6th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'00)*. 2000, pp. 169–178. DOI: 10.1145/347090.347123.

[53] A. I. McLeod and D. R. Bellhouse. "A Convenient Algorithm for Drawing a Simple Random Sample." In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 32.2 (1983), pp. 182–184. JSTOR: 2347297.

[54] J. L. Morales and J. Nocedal. "Automatic Preconditioning by Limited Memory Quasi-Newton Updating." In: *SIAM Journal on Optimization* 10.4 (2000), pp. 1079–1096.

[55] J. Nocedal and S. Wright. *Numerical optimization*. Springer series in operations research. Springer, 2006. ISBN: 9780387303031. URL: http://books.google.com/books?id=eNlPAAAAMAAJ.

[56] T. Ogita, S. M. Rump, and S. Oishi. "Accurate Sum and Dot Product." In: *SIAM Journal on Scientific Computing* 26.6 (June 2005), pp. 1955–1988. DOI: 10.1137/030601818.

[57] C. C. Paige and M. A. Saunders. "LSQR: An algorithm for sparse linear equations and sparse least squares." In: *ACM Transactions on Mathematical Software (TOMS)* 8.1 (1982), pp. 43–71.

[58] B. Panda, J. Herbach, S. Basu, and R. Bayardo. "PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce." In: *Proceedings of the VLDB Endowment* (2009).

[59] S. Press. *stcox - Cox proportional hazards model*.

[60] A. Rahmini and B. Recht. "Random Features for Large-Scale Kernel Machines." In: 2007.

[61] B. Recht, M. Fazel, and P. A. Parrilo. "Guaranteed Minimum-Rank Solutions of Linear Matrix Equations via Nuclear Norm Minimization." In: *SIAM Review* 52.3 (2010), pp. 471–501.

[62] A. Rendell. *All Pairs Shortest Paths*. `http://users.cecs.anu.edu.au/~Alistair.Rendell/Teaching/apac_comp3600/module4/all_pairs_shortest_paths.xhtml`. Accessed: 2017-06-07.

[63] C. Sanderson. *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*. Tech. rep. NICTA, 2010.

[64] F. Sha and F. C. N. Pereira. "Shallow Parsing with Conditional Random Fields." In: *HLT-NAACL*. 2003.

[65] S. Shalev-Shwartz, Y. Singer, and N. Srebro. "Pegasos: Primal Estimated Sub-Gradient Solver for SVM." In: 2007.

[66] H. Simon and H. Zha. "Low Rank Matrix Approximation Using The Lanczos Bidiagonalization Process With Applications." In: *SIAM J. Sci. Comput* 21 (2000), pp. 2257–2274.

[67] N. Srebro and T. Jaakkola. "Weighted Low-Rank Approximations." In: *ICML*. Ed. by T. Fawcett and N. Mishra. AAAI Press, 2003, pp. 720–727. ISBN: 1-57735-189-4.

[68] *Test the Proportional Hazards Assumption of a Cox Regression (R manual for Cox Zph)*.

[69] The PostgreSQL Global Development Group. *PostgreSQL 9.1.3 Documentation*. 2011. URL: `http://www.postgresql.org/docs/9.1`.

[70] A. Vattani. "k-means requires exponentially many iterations even in the plane." In: *Proceedings of the 25th Annual ACM Symposium on Computational Geometry (SCG'09)*. 2009, pp. 324–332. DOI: `10.1145/1542362.1542419`.

[71] A. J. Viterbi. "Viterbi algorithm." In: *Scholarpedia* 4.1 (2009), p. 6246.

[72] J. S. Vitter. "Random sampling with a reservoir." In: *ACM Transactions on Mathematical Software* 11.1 (1985), pp. 37–57. DOI: `10.1145/3147.3165`.

[73] J. S. Vitter. "Faster methods for random sampling." In: *Communications of the ACM* 27.7 (1984), pp. 703–718. DOI: `10.1145/358105.893`.

[74] Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang. "PLDA: Parallel Latent Dirichlet Allocation for Large-Scale Applications." In: *Proceedings of the 5th International Conference on Algorithmic Aspects in Information and Management*. AAIM '09. San Francisco, CA, USA: Springer-Verlag, 2009, pp. 301–314. ISBN: 978-3-642-02157-2. DOI: `10.1007/978-3-642-02158-9_26`. URL: `http://dx.doi.org/10.1007/978-3-642-02158-9_26`.

[75] Wikipedia. *Artificial neural network*. URL: `http://en.wikipedia.org/wiki/Artificial_neural_network`.

[76] Wikipedia. *Multilayer perceptron*. URL: `http://en.wikipedia.org/wiki/Multilayer_perceptron`.

[77] J. Wright, A. Ganesh, S. Rao, Y. Peng, and Y. Ma. "Robust Principal Component Analysis: Exact Recovery of Corrupted Low-Rank Matrices via Convex Optimization." In: *NIPS*. Ed. by Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta. Curran Associates, Inc., 2009, pp. 2080–2088. ISBN: 9781615679119.

[78] Zhiheng Huang. *Accelerating Recurrent Neural Network Training via Two Stage Classes and Parallelization*. URL: https://www.microsoft.com/en-us/research/publication/accelerating-recurrent-neural-network-training-via-two-stage-classes-and-parallelization/.

[79] H. Zou and T. Hastie. "Regularization and variable selection via th elastic net." In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67 (2 2005), pp. 301–320.