Namespace List

Here is a list of all documented namespaces with brief descriptions:

[detail level 1 2 3 4]

▼ N **meta**	The ModErn Text Analysis toolkit is a suite of natural language processing, classification, information retreival, data mining, and other applications of text processing
▼ N **analyzers**	Contains various ways to segment text and deal with preprocessed files (POS tags, parse trees, etc)
N **filters**	Contains filters that mutate existing token streams in a filter chain
N **tokenizers**	Contains tokenizers that start off a filter chain
N **caching**	Containers to be used for caching purposes
▼ N **classify**	Algorithms for feature selection, KNN search, and confusion matrices
N **kernel**	Kernel functions for linear classifiers
N **loss**	Loss functions for sgd
N **corpus**	Various ways to convert corpus formats into META-readable documents
▼ N **graph**	Contains implementations of the graph data structure and algorithms that operate over them
▼ N **algorithms**	
N **internal**	Contains helper functions used by the centrality measures
N **index**	Indexes to create efficient representations of data
▼ N **io**	Compressed file readers and writers, configuration file readers, a simple parser, and memory-mapped file support
N **libsvm_parser**	Parser specifically for libsvm-formatted files
N **lm**	Contains implementations of statistical language models
N **logging**	Namespace which contains all of the logging interface classes
N **parallel**	Implementation of a thread pool and a parallel for loop
N **parser**	Contains functions that relate to phrase structure trees and parsing of natural language
N **printing**	Contains functions that print to the terminal and provide progress bars
N **sequence**	Sequence representations and labeling models/algorithms
N **stats**	Probability distributions and other statistics functions
N **testing**	Contains unit testing functions for the META toolkit
N **topics**	Topic modeling functionality
N **utf**	Functions for converting to and from various character sets
N **util**	Shared resources and utilities

Generated on Tue Mar 3 2015 23:20:41 for ModErn Text Analysis by doxygen 1.8.9.1

Analyzers and Filters

Analyzers, Tokenizers, and Filters

The first step in creating an index over any sort of text data is the “tokenization” process. At a high level, this simply means converting your individual text documents into sparse vectors of counts of terms—these sparse vectors are then typically consumed by an indexer to output an `inverted_index` over your corpus.

MeTA structures this text analysis process into several layers in order to give you as much power and control over the way your text is analyzed as possible. The important components are

- `analyzer` s, which take `document` s from the `corpus` and convert their content into sparse vectors of counts, storing that data back in the `document` ,
- `tokenizer` s, which take a `document` ’s content and split it into a stream of tokens, and
- `filter` s, which take a stream of tokens, perform operations on them (like stemming, filtering,

and other mutations), and also produce a stream of tokens

An `analyzer`, in most cases, will take a "filter chain" that is used to generate the final tokens for its tokenization process: the filter chains are always defined as a specific `tokenizer` class followed by a sequence of 0 or more `filter` classes, each of which reads from the previous class's output. For example, here is a simple filter chain that lowercases all tokens and only keeps tokens with a certain length:

```
icu_tokenizer -> lowercase_filter -> length_filter
```

Using the Default Filter Chain

MeTA defines a "sane default" filter chain that you are encouraged to use for general text analysis in the absence of any specific requirements. To use it, you should specify the following in your configuration file:

```
[[analyzers]]
method = "ngram-word"
ngram = 1
filter = "default-chain"
```

This configures your text analysis process to consider unigrams of words generated by running each of your documents through the default filter chain. This filter chain should work well for most languages, as all of its operations (including but not limited to tokenization and sentence boundary detection) are defined in terms of the Unicode standard wherever possible.

To consider both unigrams and bigrams, your configuration file should look like the following:

```
[[analyzers]]
method = "ngram-word"
ngram = 1
filter = "default-chain"

[[analyzers]]
method = "ngram-word"
ngram = 2
filter = "default-chain"
```

Each `[[analyzers]]` block defines a single analyzer and its corresponding filter chain: you can use as many as you would like—the tokens generated by each analyzer you specified will be counted and placed in a single sparse vector of counts. This is useful for combining multiple different kinds of features together into your document representation. For example, the following configuration would combine unigram words, bigram part-of-speech tags, tree skeleton features, and subtree features.

```
[[analyzers]]
method = "ngram-word"
ngram = 1
filter = "default-chain"

[[analyzers]]
method = "ngram-pos"
ngram = 2
filter = [{type = "icu-tokenizer"}, {type = "ptb-normalizer"}]
crf-prefix = "path/to/crf/model"

[[analyzers]]
method = "tree"
filter = [{type = "icu-tokenizer"}, {type = "ptb-normalizer"}]
features = ["skel", "subtree"]
tagger = "path/to/greedy-tagger/model"
parser = "path/to/sr-parser/model"
```

Getting Creative: Specifying Your Own Filter Chain

If your application requires specific text analysis operations, you can specify directly what your filter chain should look like by modifying your configuration file. Instead of `filter` being a string parameter as above, we will change `filter` to look very much like the `[[analyzers]]` blocks: each `analyzer` will have a series of `[[analyzers.filter]]` blocks, each of which defines a step in the filter chain. **All filter chains must start with a tokenizer.** Here is an example filter chain for unigram words like the one at the beginning of this tutorial:

```
[[analyzers]]
method = "ngram-word"
ngram = 1
    [[analyzers.filter]]
    type = "icu-tokenizer"

    [[analyzers.filter]]
    type = "lowercase"

    [[analyzers.filter]]
    type = "length"
    min = 2
    max = 35
```

MeTA provides many different classes to support building filter chains. Please look at the API documentation for more information. In particular, the `analyzers::tokenizers` namespace (doxygen/namespacemeta_1_1analyzers_1_1tokenizers.html) and the `analyzers::filters`

namespace (doxygen/namespacemeta_1_1analyzers_1_1filters.html) should give you a good idea of the capabilities—the static public attribute `id` for a given class is the string you need to use for the "type" in the configuration file.

Extending MeTA With Your Own Filters

In certain situations, you may want to do more complex text analysis by defining your own components to plug into the filter chain. To do this, you should first determine what kind of component you want to add.

- Add an `analyzer` if you want to define an entirely new kind of token (e.g., tree features).
- Add a `tokenizer` if you want to change the way that tokens are generated directly using the document's plain-text content.
- Add a `filter` if you want to mutate, remove, or inject tokens into an existing stream of tokens.

Adding an Analyzer

To define your own analyzer is to specify your own mechanism for document tokenization entirely. This is typically done in cases where analyzing the text of a document directly is not sufficient (or not meaningful). A good example of the need for a new analyzer is the existing `libsvm_analyzer`, which tokenizes documents whose content is actually already pre-processed to be in the standard libsvm format. Other examples include the subclasses of `tree_analyzer` which operate on pre-processed document trees.

Adding your own analyzer is relatively straightforward: you should subclass from `analyzer` and implement the `tokenize(corpus::document)` method. One slight caveat to be aware of is that `analyzer`s are required to be clonable by the internal implementation, but this is easily solved by adapting your subclassing specification from

```
class my_analyzer : public meta::analyzers::analyzer
{
    /* things */
};
```

to

```
class my_analyzer : public meta::util::clonable<analyzer, my_analyzer>
{
    /* things */
};
```

and providing a valid copy constructor. (The polymorphic cloning facility is taken care of by the base `analyzer` combined with the `util::clonable` mixin.)

Your `tokenize` method is responsible for incrementing the counts of your features in the given `corpus::document` object given to the method. Features are identified by unique strings.

Your `analyzer` object will be a **thread-local instance** during indexing, so be aware that member variables are not shared across threads, and that access to any static member variables should be properly synchronized. We *strongly encourage* state-less analyzers (that is, analyzers that are capable of operating on a single document at a time without keeping context information).

To be able to use your analyzer by specifying it in a configuration file, it must be registered with the factory. You can do this by calling the following function in `main()` somewhere before you create your index:

```
meta::analyzers::register_analyzer<my_analyzer>();
```

The class `my_analyzer` should also have a static member `id` that specifies the string that should be used to identify that analyzer to the factory—this id must be unique.

If you require special construction behavior (beyond default construction), you may specialize the `make_analyzer()` function for your specific analyzer class to extract additional information from the configuration file: that specialization would look something like this:

```
namespace meta
{
namespace analyzers
{
template <>
std::unique_ptr<analyzer>
    make_analyzer<my_analyzer>(const cpptoml::table& global,
                               const cpptoml::table& local);
}
}
```

The first parameter is the configuration group for the *entire* configuration file, and the second parameter is the local configuration group for your analyzer block. Generally, you will only use the local configuration group unless you need to read some global paths from the main configuration file.

Adding a Tokenizer

To define your own tokenizer is to specify a new mechanism for initially separating the textual content of a document into a series of discrete “tokens”. These tokens may be modified later via filters (they may be split, removed, or otherwise modified), but a tokenizer’s job is to do this initial separation work. Creating a new tokenizer should be a relatively rare occurrence, as the existing `icu_tokenizer` should perform well for most languages due to its adherence to the Unicode standard (and its related annexes).

Adding your own tokenizer is very similar to adding an analyzer: you need to subclass `token_stream` now, and the same clonable caveat remains, so your declaration should look something like this:

```
class my_tokenizer : public meta::util::clonable<token_stream,
                                                 my_tokenizer>
{
    /* things */
};
```

Remember to provide a valid copy constructor!

Your tokenizer class should implement the virtual methods of the `token_stream` class (doxygen/classmeta_1_1analyzers_1_1token__stream.html).

- `next()` obtains the next token in the sequence
- `set_content()` changes the underlying content being tokenized
- `operator bool()` determines if there are more tokens left in your token stream

To be able to use your tokenizer by specifying it in a configuration file, it must be registered with the factory. You can do this by calling the following function in `main()` somewhere before you create your index:

```
meta::analyzers::register_tokenizer<my_tokenizer>();
```

The class `my_tokenizer` should also have a static member `id` that specifies the string to be used to identify that tokenizer to the factory—this id must be unique.

If you require special construction behavior (beyond default construction), you may specialize the `make_tokenizer()` function for your specific tokenizer class to extract additional information from the configuration file: that specialization would look something like this:

```
namespace meta
{
namespace analyzers
{
namespace tokenizers
{
template <>
std::unique_ptr<token_stream>
    make_tokenizer<my_tokenizer>(const cpptoml::table& config);
}
}
}
```

The configuration group passed to this function is the configuration block for your tokenizer.

Adding a Filter

To add a filter is to specify a new mechanism for transforming existing token streams after they have been created from a document. This should be the most common occurrence, as it's also the most general and encompasses things like lexical analysis, filtering, stop word removal, stemming, and so on.

Creating a new filter is nearly identical to creating a new tokenizer class: you will subclass `token_stream` (using the `util::clonable` mixin) and implement the virtual functions of `token_stream`. The major difference is that a filter class's constructor takes as its first parameter the `token_stream` to read from (this is passed as a `std::unique_ptr<token_stream>` to signify that your filter class should take ownership of that source).

Registration of a new filter class is done as follows:

```
meta::analyzers::filters::register_filter<my_filter>();
```

And the following is the specialization of the `make_tokenizer()` function that would be required if you need special construction behavior:

```
namespace meta
{
namespace analyzers
{
namespace filters
{
template <>
std::unique_ptr<token_stream>
    make_fitler<my_filter>(std::unique_ptr<token_stream> source,
                           const cpptoml::table& config);
}
}
}
```

Documentation for MeTA: ModErn Text Analysis (https://github.com/meta-toolkit/meta)

MeTA: ModErn Text Analysis (/meta/) A Modern C++ Data Sciences Toolkit

Basic Text Analysis A whirlwind tour of some features of MeTA

MeTA focuses on providing rich functionality for text mining applications. This tutorial will use the provided `profile` application to demonstrate some of the text analysis features that are built in to MeTA. It is not completely comprehensive, but it should give you an idea of some of the analysis you can do using MeTA as a framework.

We **strongly** encourage you to read the source for this demo application, which is located in `src/tools/profile.cpp` and its corresponding `src/tools/CMakeLists.txt`. This file is heavily commented and should serve as an example starting point for using MeTA programmatically.

Setup

Obtain a text file for analysis. You can either create this manually, copy some text from the Internet, or even use this very tutorial as your text source. Save it in your `build` directory as `doc.txt`. We will use this file as raw data, processing it through several different tools provided by MeTA.

Shallow text analysis

To start, we will analyze `doc.txt` from a "shallow" point of view. We will look at our document as just a set of sentences composed of individual tokens.

Stop word removal

When viewing text in this mode, it's often beneficial to reduce the number of unique tokens we are considering, especially if they do not contribute to the meaning of the text. Words like *the*, *of*, and *to* do very little in telling us about the content of `doc.txt`, so it's often safe to remove them and still keep the gist of the document intact. Removing these uninformative, frequently occurring words can dramatically reduce the amount of text we need to process.

MeTA provides a default list of these "stop words" in `data/lemur-stopwords.txt`, which have been taken from the Lemur project (http://www.lemurproject.org/).

Remove the stop words from your document by running the following command:

```
./profile config.toml doc.txt --stop
```

The output should be saved to the file `doc.stops.txt`. To see how this was done, refer to the `stop()` function in `src/tools/profile.cpp`. This is utilizing MeTA's tokenizers and filters (/meta/analyzers-filters-tutorial.html) to create a pipeline that outputs processed tokens.

Stemming

In many applications (and particularly for retrieval) it is beneficial to reduce words down to their base forms. For example, if someone searches for *running*, it is reasonable to return results that match *run* or *runs* in addition to documents that match *running* directly. MeTA ships with an implementation of the Porter2 Stemmer for English (http://snowball.tartarus.org/algorithms/english/stemmer.html), which is used in the default analysis pipeline. Some example stems are listed below.

```
{run, runs, running} -> run
{argue, argued, argues, arguing} -> argu
{lies, lying, lie} -> lie
```

Reduce all words down to their stems in your document by running the following command:

```
./profile config.toml doc.txt --stem
```

The output should be saved to the file `doc.stems.txt`. To see how this was done, refer to the `stem()` function in `src/tools/profile.cpp`. This is another example of utilizing MeTA's tokenizers and filters (/meta/analyzers-filters-tutorial.html).

Frequency analysis

One convenient way of representing documents is the so called “bag of words” representation. In this view, documents are represented simply as vectors of word counts. All position information is thrown away. While this seems naive at first, it is actually a fairly effective representation for a number of applications, including document retrieval, topic modeling, and document classification.

You can calculate the bag of words representation for your document by running the following commands:

```
# a bag of individual words
./profile config.toml doc.txt --freq-unigram

# a bag of two consecutive word sequences
./profile config.toml doc.txt --freq-bigram

# a bag of three consecutive word sequences
./profile config.toml doc.txt --freq-trigram
```

The output should be saved to the files `doc.freq.n.txt` where “n” is 1, 2, or 3 depending on the n-gram choice (unigram, bigram, or trigram) specified in the flags above. To see how this was done, refer to the `freq()` method in `src/tools/profile.cpp`. This is an example of using an analyzer (/meta/analyzers-filters-tutorial.html) to reduce a document into a sparse feature vector (here, each feature is the name of an n-gram).

Natural language processing (NLP)

Depending on the application, it may be useful to extract features that are more linguistically motivated than just the tokens themselves. MeTA supports a few common tasks used when trying to get a more *deep* understanding of a document: part-of-speech tagging and phrase structure (or constituency) parsing.

Part-of-speech tagging

Every word in the English language can be assigned at least one class (or tag) that indicates its “part(s)-of-speech”. For example, the word *the* is a determiner (DT), and the word *token* is a singular noun (NN). Here is a list of commonly used part-of-speech tags (https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html) for English as defined by the Penn Treebank project.

The task of taking a sequence of words and assigning each word a part-of-speech (POS) tag is referred to as “part-of-speech tagging”. MeTA supports part of speech tagging with a few different models. Let’s use one of them to POS tag our `doc.txt`.

First, visit the releases page (https://github.com/meta-toolkit/meta/releases), click the latest version, and download the model files for the “greedy part-of-speech tagger”. Extract these into a folder and take note of the path to that folder.

Ensure that you have a section in `config.toml` that looks like the following:

```
[sequence]
prefix = "path/to/your/tagger/folder/"
```

Now, you should be able to POS-tag your `doc.txt` by running the following command:

```
./profile config.toml doc.txt --pos
```

The output should be written to `doc.pos-tagged.txt`. To see how this was done, refer to the `pos()` method in `src/tools/profile.cpp`. This is an example of using a greedy Perceptron-based tagger (/meta/pos-tagging-tutorial.html) for POS tagging.

Parsing

Sometimes it is beneficial to get a deeper understanding of the structure of a sentence beyond just assigning each word a part-of-speech. Language is recursive and hierarchical—one task in NLP is to determine the phrase structure tree that describes how the parts of a sentence connect with one another. Wikipedia has a good example. (http://en.wikipedia.org/wiki/Parse_tree#Constituency-based_parse_trees)

MeTA has support for inferring the phrase structure tree that was used to generate a sentence. Let's use this capability to transform each sentence in our `doc.txt` into a separate phrase structure tree.

First, visit the releases page (https://github.com/meta-toolkit/meta/releases), click on the latest version, and download the model files for the "greedy shift-reduce constituency parser". Extract these into a folder and take note of the path to that folder.

Ensure that you have a section in `config.toml` that looks like the following:

```
[parser]
prefix = "path/to/your/parser/folder/"
```

Since the parser relies on having POS-tagged sentences, ensure that you've done the part-of-speech tagging section of this tutorial as well.

Parse the sentences in `doc.txt` by running the following command:

```
./profile config.toml doc.txt --parse
```

The output should be written to `doc.parsed.txt`. To see how this was done, refer to the `parse()` method in `src/tools/profile.cpp`. This is an example of using an efficient shift-reduce constituency parser (/meta/parsing-tutorial.html) for deriving phrase structure trees.

Classification

Classification in MeTA

Beyond its search and information retrieval capabilities, MeTA also provides functionality for performing document classification on your corpora. We will start this tutorial by discussing how you can perform document classification experiments on your data **without writing a single line of code**, and then discuss more complicated examples later on.

The first step in setting up your classification task is, of course, selecting your corpus. MeTA's built-in `classify` program supports the following corpus inputs:

- anything from which an `inverted_index` can be generated
- pre-processed, libsvm-formatted corpora

MeTA's internal representation for its `forward_index` uses libsvm-formatted data, so if you already have your data in this format you are encouraged to use it. Otherwise, you can generate libsvm-formatted data automatically from any existing `inverted_index`.

Creating a Forward Index From Scratch

To create a `forward_index` directly from your corpus input, your configuration file would look something like this:

```
corpus-type = "line-corpus"
dataset = "20newsgroups"
forward-index = "20news-fwd"
inverted-index = "20news-inv"

[[analyzers]]
method = "ngram-word"
ngram = 1
filter = "default-chain"
```

The process looks something like this: first, an `inverted_index` will be created (or loaded if it already exists) over your corpus' data, and then it will be un-inverted to create the `forward_index`. The process of un-inverting is a one-time cost, and is necessary to provide efficient access to the document vectors for the classifiers. Once you have generated your `forward_index`, you *never need to generate it again* unless you want to change your document representation. Once you have a `forward_index`, you can use it with *any* of the classifiers provided by MeTA.

Creating a Forward Index from LIBSVM Data

In many cases, you may already have pre-processed corpora to perform classification tasks on, and MeTA gracefully handles this. The most common input for these classification tasks is the LIBSVM file format, and MeTA supports this format directly as an input corpus.

To create a `forward_index` from data that is already in LIBSVM format, your configuration file would look something like this:

```
corpus-type = "line-corpus"
dataset = "rcv1"
forward-index = "rcv1-fwd"
inverted-index = "rcv1-inv"

[[analyzers]]
method = "libsvm"
```

The `forward_index` will recognize that this is a LIBSVM formatted corpus and will simply generate a few metadata structures to ensure efficient random access to document vectors. An `inverted_index` is not created through this method, and so you will not be able to use classifiers such as `knn` that require an `inverted_index`.

Selecting a Classifier

To actually run the `classify` executable, you will need to decide on a classifier to use. You may see a list of these in the API documentation for the `classifier` class (doxygen/classmeta_1_1classify_1_1classifier.html) (they are listed as subclasses). The public static id member of each class is the identifier you would use in the configuration file.

A recommended default configuration is given below, which learns an SVM via stochastic gradient descent:

```
[classifier]
method = "one-vs-all"
    [classifier.base]
    method = "sgd"
    loss = "hinge"
    prefix = "sgd-model"
```

Here is an example configuration that uses Naive Bayes:

```
[classifier]
method = "naive-bayes"
```

Here is an example that uses *k*-nearest neighbor with *k* = 10 and Okapi BM25 as the ranking function:

```
[classifier]
method = "knn"
k = 10
    [classifier.ranker]
    method = "bm25"
```

Running `./classify config.toml` from your build directory will now create a `forward_index` (if necessary) and run 5-fold cross validation on your data using the prescribed classifier. Here is some sample output:

```
            chinese   english   japanese
          ------------------------------
  chinese | 0.802     0.011     0.187
  english | 0.0069    0.807     0.186
 japanese | 0.0052    0.0039    0.991

-----------------------------------------------
Class       F1 Score    Precision   Recall
-----------------------------------------------
chinese     0.864       0.802       0.936
english     0.88        0.807       0.967
japanese    0.968       0.991       0.945
-----------------------------------------------
Total       0.904       0.867       0.949
-----------------------------------------------
1005 predictions attempted, overall accuracy: 0.947
```

Online Learning

If your dataset cannot be loaded into memory, you should look at the online learning tutorial (online-learning.html) for more information about how to handle this case.

Manual Classification

If you want to customize the classification process (such as providing your own test/training split, or changing the number of cross-validation folds), you should interact with the classifiers directly by writing some code. Refer to `classify.cpp` and the API documentation for `classifier` (doxygen/classmeta_1_1classify_1_1classifier.html).

Here's a simple example that changes the number of folds to 10:

```
auto f_idx = meta::index::make_index<index::memory_forward_index>(argv[1]);
auto config = cpptoml::parse_file(argv[1]);

auto class_config = config.get_table("classifier");
auto classifier = meta::classify::make_classifier(*class_config, f_idx);

auto confusion_mtrx = classifier->cross_validate(f_idx->docs(), 10);
confusion_mtrx.print();
confusion_mtrx.print_stats();
```

And here's a simple example that uses your own training/test split:

```
auto f_idx = meta::index::make_index<index::memory_forward_index>(argv[1]);
auto config = cpptoml::parse_file(argv[1]);

auto class_config = config.get_table("classifier");
auto classifier = meta::classify::make_classifier(*class_config, f_idx);

auto train = /* filter f_idx->docs() for your training set */;
auto test = /* filter f_idx->docs() for your test set */;

classifier->train(train);
auto confusion_mtrx = classifier->test(test);
confusion_mtrx.print();
confusion_mtrx.print_stats();
```

Writing Your Own Classifiers

Any classifiers you write should subclass `classify::classifier` and implement its virtual methods. You should be clear as to whether your classifier directly supports multi-class classification (subclass from `classify::classifier` directly) or only binary classification (sublcass from `classify::binary_classifier`).

If you would like to be able to create your classifier by specifying it in a configuration file, you will need to provide a public static id member that specifies the text that identifies your classifier class, and register it with the toolkit somewhere in `main()` like this:

```
// if you have a multi-class classifier
meta::classify::register_classifier<my_classifier>();

// if you have a multi-class classifier that requires an
// inverted_index
meta::classify::register_multi_index_classifier<my_classifier>();

// if you have a binary classifier
meta::classify::register_binary_classifier<my_binary_classifier>();
```

If you need to read parameters from the configuration group given for your classifier, you should specialize the `make_classifier()` function like so:

```
// if you have a multi-class classifier
namespace meta
{
namespace classify
{
template <>
std::unique_ptr<classifier>
    make_classifier<my_classifier>(
        const cpptoml::table& config,
        std::shared_ptr<index::forward_index> idx);
}
}

// if you have a multi-class classifier that requires an
// inverted_index
namespace meta
{
namespace classify
{
template <>
std::unique_ptr<classifier>
    make_multi_indexclassifier<my_classifier>(
        const cpptoml::table& config,
        std::shared_ptr<index::forward_index> idx,
        std::shared_ptr<index::inverted_index> inv_idx);
}
}

// if you have a binary classifier
namespace meta
{
namespace classify
{
template <>
std::unique_ptr<classifier>
    make_binary_classifier<my_binary_classifier>(
        const cpptoml::table& config,
        std::shared_ptr<index::forward_index> idx,
        class_label positive_label,
        class_label negative_label);
}
}
```

MeTA: ModErn Text Analysis **A Modern C++ Data Sciences Toolkit**

MeTA System Overview

MeTA's goal

Our task is to improve upon and complement the current body of open source machine learning and information retrieval software. The existing environment of this open source software tends to be quite fragmented.

There is rarely a single location for a wide variety of algorithms; a good example of this is the LIBLINEAR software package for SVMs. While this is the most cited of the open source implementations of linear SVMs, it focuses solely on kernel-less methods. If presented with a nonlinear classification problem, one would be forced to find a different software package that supports kernels (such as the same authors' LIBSVM package). This places an undue burden on the researchers—not only are they required to have a detailed understanding of the research problem at hand, but they are now forced to understand this fragmented nature of the open source software community, find the appropriate tools in this mishmash of implementations, and compile and configure the appropriate tool.

Even when this is all done, there is the problem of data formatting—it is unlikely that the tools have standardized upon a single input format, so a certain amount of "data munging" is now required. This all detracts from the actual research task at hand, which has a marked impact on the speed of discovery.

We want to address these issues. In particular, we want to provide a unifying framework for text indexing and analysis methods, allowing researchers to quickly run controlled experiments. We want to modularize the feature generation, instance representation, data storage formats, and algorithm implementations; this will allow for researchers to make seamless transitions along any of these dimensions with minimal effort.

How does MeTA store data?

All processed data is stored in an `index`. Currently, we have two indexes: `forward_index` and

`inverted_index`. The former is keyed by document IDs, and the latter is keyed by term IDs.

- `forward_index` is used for applications such as topic modeling and most classification tasks
- `inverted_index` is used to create search engines, or do classification with *k*-nearest-neighbor

Since each MeTA application takes an index as input, all processed data is interchangeable between all the components. This also gives a great advantage to classification: **MeTA supports out-of-core classification by default!** If your dataset is small enough (like most other toolkits assume), you can use a cache such as `no_evict_cache` to keep it all in memory without sacrificing any speed.

Index usage is explained in much more detail in the Search Tutorial, though it might make more sense to read the about Filters and Analyzers first.

Corpus input formats

There are currently three corpus input formats:

- `line_corpus`: each dataset consists of one to three files:
  - `corpusname.dat`: each document appears on one line.
  - `corpusname.dat.names`: optional file that includes the name or path of the document on each line, corresponding to the order in `corpusname.dat`. These are the names that (*e.g.*) are returned when running the search engine.
  - `corpusname.dat.labels`: optional file that includes the class or label of the document on each line, again corresponding to the order in `corpusname.dat`. These are the labels that are used for the classification tasks.
- `gz_corpus`: similar to `line_corpus`, but each of its files and metadata files are compressed using gzip compression:
  - `corpusname.dat.gz`: compressed version of `corpusname.dat`
  - `corpusname.dat.names.gz`: compressed version of `corpusname.dat.names`
  - `corpusname.dat.labels.gz`: compressed version of `corpusname.dat.labels`
  - `corpusname.dat.numdocs`: an uncompressed file containing only the count of the number of documents in the corpus (this is used to be able to provide progress reporting when tokenizing documents)
- `file_corpus`: each document is its own file, and the name of the file becomes the name of the document. There is also a `corpusname-full-corpus.txt` which contains (on each line) a required label for each document followed by the path to the file on disk. If the documents don't have specified class labels, just precede the line with [none] or similar.

If only being used for classification, MeTA can also take libsvm-formatted input to create a `forward_index`.

Datasets

There are several public datasets that we've converted to the `line_corpus` format:

- 20newsgroups, originally from here.
- IMDB Large Movie Review Dataset, originally from here.
- Any libsvm-formatted dataset can be used to create a `forward_index`.

Tokenizers, Filters, and Analyzers

MeTA processes data with a system of tokenizers, filters and analyzers before it is indexed.

Tokenizers come first. They define how to split a document's content into tokens. Some examples are:

- `icu_tokenizer`: converts documents into streams of tokens by following the unicode standards for sentence and word segmentation.
- `character_tokenizer`: converts documents into streams of single characters.

Filters come next, and can be chained together. They define ways that text can be modified or transformed. Here are some examples of filters:

- `length_filter`: this filter accepts tokens that are within a certain length and rejects those that are not.
- `icu_filter`: applies an ICU transliteration to each token in the sequence.
- `list_filter`: this filter either accepts or rejects tokens based on a list. For example, you could use a stopword list and reject stopwords.
- `porter2_stemmer`: this filter transforms each token according to the Porter2 English Stemmer rules.

Analyzers operate on the output from the filter chain and produce token counts from documents. Here are some examples of analyzers:

- `ngram_word_analyzer`: collects and counts sequences of *n* words (tokens) that have been filtered by the filter chain
- `ngram_pos_analyzer`: same as `ngram_word_analyzer`, but operates on part-of-speech tags from MeTA's CRF
- `tree_analyzer`: collects and counts occurrences of parse tree structural features, currently the only known implementation of work described in this paper.
- `libsvm_analyzer`: converts a libsvm `line_corpus` into MeTA format.

For a more detailed intro, see Filters and Analyzers.

Unit tests

We're using ctest, which is configured to run all the tests available in the `unit-test.cpp` file. You may run

```
ctest --output-on-failure
```

to execute the unit tests while displaying output from failed tests.

The file `unit-test.cpp`, takes various tests as parameters. For example,

```
./unit-test inverted-index
```

or

```
./unit-test all
```

Please note that you must have a valid configuration file (config.toml) in the project root, since many unit tests create input based on paths stored there.

You won't normally have to worry about running `./unit-test` yourself since we have ctest set up. However, since ctest creates a new process for each test, it is hard to debug. If you'd like to debug with tools such as valgrind or gdb, running `./unit-test` manually is a better option.

Bugs

On GitHub, create an issue and make sure to label it as a bug. Please be as detailed as possible. Including a code snippet would be very helpful if applicable.

You many also add feature requests the same way, though of course the developers will have their own priority for which features are addressed first.

Contribute

Forks are certainly welcome. When you have something you'd like to add, submit a pull request and a developer will approve it after review.

We're also looking for more core developers. If you're interested, don't hesitate to contact us!

Citing MeTA

For now, please cite the homepage (http://meta-toolkit.github.io/meta/) and note which revision you used.

Once (hopefully) we have a paper, you would be able to cite that.

Developers

Sean Massung started writing MeTA in May 2012 as a tool to help with his senior thesis at UIUC. He is currently a graduate student there, working on information retrieval and natural language processing with Professor ChengXiang Zhai.

Chase Geigle needs to write his bio.

License

Online Learning

Online Learning in MeTA

In many cases, the datasets we deal with in large machine learning problems are too large to fit into the working-set memory of a modest computer. Fortunately, MeTA's use of indexes for storing its data make it very capable of handling these cases when coupled with a classifier that supports online learning (such as `sgd` coupled with any of the appropriate loss functions, or a `one_vs_all` or `one_vs_one` ensemble of these classifiers). In this tutorial, we'll explore performing online learning of an `sgd`-trained support vector machine (SVM) on a dataset from the LIBSVM dataset website, rcv1.binary (http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#rcv1.binary). (This dataset is not actually large enough to require an online learning algorithm, but it is large enough to demonstrate the value in this approach, and one can easily see how the method can be extended to datasets that truly do require an online-learning approach.)

Download the training and testing sets and place them in your data directory under a folder called `rcv1`. Then, construct an `rcv1.dat` from the following command (or equivalent):

```
bzcat rcv1_test.binary.bz2 rcv1_train.binary.bz2 > rcv1.dat
```

(Note: we will be using the given test set as the training set and the given training set as the test set: this is mostly just so that the training set is sizeable enough to appreciate the memory usage of the toolkit during training: reversing the splits gives us approximately 1.2GB of training data to process).

The approach we will take here is to run the `sgd` training algorithm several times on "mini-batches" of the data. The dataset has a total of 677399 training examples, and if we loaded these all into memory it would take nearly a gigabyte of working memory. Instead, we will load the data into memory in chunks of <= 50000 documents and train on each chunk individually.

Below is the relevant portion of our `config.toml` for this example:

```
corpus-type = "line-corpus"
dataset = "rcv1"
forward-index = "rcv1-fwd"
inverted-index = "rcv1-inv"

# the size of the mini-batches: this may need to be set empirically based
# on the amount of memory available on the target system
batch-size = 50000
# the document-id where the test set begins
test-start = 677399

[[analyzers]]
method = "libsvm"

[classifier]
method = "one-vs-all"
    [classifier.base]
    method = "sgd"
    loss = "hinge"
    prefix = "sgd-model"
```

Now, we can run the provided application with `./online-classify config.toml`, see results that look something like the following:

```
$ ./online-classify config.toml
1395363665: [info]     Loading index from disk: rcv1-fwd
(/home/chase/projects/meta/src/index/forward_index.cpp:137)
 > Counting lines in file: [================================] 100% ETA 00:00:00
Training batch 14/14

           -1       1
         ------------------
      -1 | 0.966    0.0344
       1 | 0.0274   0.973

-------------------------------------------------
Class       F1 Score    Precision   Recall
-------------------------------------------------
-1          0.968       0.966       0.97
1           0.97        0.973       0.968
-------------------------------------------------
Total       0.969       0.969       0.969
-------------------------------------------------
20242 predictions attempted, overall accuracy: 0.969
Took 119s
```

At the time of this tutorial's writing, this classification process peaks at about 95MB—a far cry away from the 1.2GB training set! This general process should be able to be extended to work with any dataset that cannot be fit into memory, provided an appropriate `batch-size` is set in the configuration file.

Parsing

Many applications that deal with natural language desire constituency (or syntax) trees that describe the individual phrases in sentences and how they are hierarchically combined. Wikipedia has a nice example (http://en.wikipedia.org/wiki/Parse_tree#Constituency-based_parse_trees).

MeTA provides a constituency parser in the `meta::parser` namespace. This is a very efficient parser that is based on the shift-reduce parsing algorithm. For more information on the algorithm itself, refer to the following papers:

- Fast and Accurate Shift-Reduce Constituency Parsing (http://people.sutd.edu.sg/~yue_zhang/pub/acl13.muhua.pdf)
- Transition-Based Parsing of the Chinese Treebank using a Global Discriminative Model (http://www.aclweb.org/anthology/W09-3825)

Using the parser

First, you will need to download a parser model file from the releases page (https://github.com/meta-toolkit/meta/releases) on the Github repository. MeTA currently supplies two trained shift-reduce parser models:

- A greedy, best-first parser (i.e. a beam size of 1)
- A beam-search parser with a maximum beam size of 4

Choosing between the two models is a time/performance tradeoff. The greedy parser is signifigantly faster than the beam-search parser, but achieves slightly lower accuracy than the beam-search parser. It should be noted, however, that either model is likely much faster than traditional PCFG-based parsers due to their lower algorithmic complexity during parsing.

In order to use the parser, you will need to have a `[parser]` configuration group in `config.toml` that looks like the following:

```
[parser]
prefix = "path/to/parser/model"
```

You will also likely want a part-of-speech tagging model, as the parser operates over pre-tagged sequences (it does not assign POS tags on its own). You should refer to the part-of-speech tagging tutorial (/meta/pos-tagging-tutorial.html) for more information on the part-of-speech tagging models that MeTA provides.

Interactive parsing

You can interactively parse sentences using the `profile` tool. See the `profile` tutorial (/meta/profile-tutorial.html) for a walkthrough of that demo application.

As an analyzer

The shift-reduce parser is now what powers MeTA's structural tree feature (http://web.engr.illinois.edu/~massung1/files/icsc-2013.pdf) extraction. In order to use the parser's supplied `tree_analyzer` in your analyzer pipeline, you will need to add an analyzer group to `config.toml` that looks like the following:

```
[[analyzers]]
method = "tree"
filter = [{type = "icu-tokenizer"}, {type = "ptb-normalizer"}]
features = ["skel", "subtree"]
tagger = "path/to/greedy-tagger/model"
parser = "path/to/sr-parser/model"
```

There are a few things to note here. First, you **must** provide a path to a trained parser **as well as** a trained **greedy tagger**. The analyzer currently is not configured to use the CRF tagger (though this may be added in the future: patches welcome!).

You may modify the filter chain if you would like, but we strongly recommend sticking with he above setup as it is designed to match the original Penn Treebank tokenization format that the supplied models were trained on.

The supported features that can be specified in the `features` array are “branch”, “depth”, “semi-skel”, “skel”, “subtree”, and “tag”. These correspond to the feature types given in the structural tree feature paper linked above.

While you could use multiple `[[analyzers]]` groups for each individual tree feature you would like to compute, it is much preferable to list all of the features you want in the `features` key to avoid re-parsing sentences for each feature.

Programmatically

To use the shift-reduce parser inside your own program, your code might look like this:

```
// load the models
meta::sequence::perceptron tagger{"path/to/perceptron/model/folder"};
meta::parser::sr_parser parser{"path/to/sr-parser/model/folder"};

meta::sequence::sequence seq;
// - code for loading/creating the sequence here -

// tag the sequence
tagger.tag(seq);

// parse the tagged sequence
auto tree = parser.parse(seq);

// print the parse tree in an indented, human-readable format
tree.pretty_print(std::cout);

// print the parse tree in a collapsed, machine-readable format
std::ofstream outfile{"my_output_file.trees"};
outfile << tree;
```

Have a look at the API documentation for the `meta::parser::sr_parser` class (/meta/doxygen/classmeta_1_1parser_1_1sr__parser.html) for more information.

Training the parser

In order to train your own models using our provided programs, you will need to have a copy of the Penn Treebank (v3) extracted into your data prefix. (see the overview tutorial (/meta/overview-tutorial.html)). Your folder structure should look like the following:

```
prefix
|---- penn-treebank
      |---- treebank-3
            |---- parsed
                  |---- mrg
                        |---- wsj
                              |---- 00
                              |---- 01
                              ...
                              |---- 24
```

You will need to expand your `[parser]` configuration group in `config.toml` to look like the following:

```
[parser]
prefix = "parser"
treebank = "penn-treebank" # relative to data prefix
corpus = "wsj"
section-size = 99
# these are the standard training/development/testing splits for parsing
train-sections = [2, 21]
dev-sections = [22, 22]
test-sections = [23, 23]
```

You can also add the following additional keys to configure training behavior:

- `train-threads` : controls the number of training threads used (defaults to the number of processors on the machine)
- `train-algorithm` : controls the algorithm used for training. This can be one of the following:
  - "early-termination" (default): trains a greedy parser (beam size = 1)
  - "beam-search": trains a parser using beam search
- `beam-size` : controls the size of the beam used during training. This option is ignored if the algorithm is "early-termination"

There are more options that can be tweaked, but the remaining options must be done programmatically. See the API documentation for the `meta::parser::sr_parser` class (/meta/doxygen/classmeta_1_1parser_1_1sr__parser.html) for more information (in particular, the `training_options` struct). If you want to try different options, you can use the code provided in `src/parser/tools/parser_train.cpp` as a starting point.

The parser may take several hours to train, depending on your training parameters. The greedy parser ("early-termination") trains much faster than the beam search parser.

Testing the parser

If you follow the instructions above, you should be able to test a parser model by running the following:

```
./parser-test config.toml tree-output
```

The application will run over the test set configured in `config.toml` and reports many of the evalb metrics (/doxygen/classmeta_1_1parser_1_1evalb.html). However, it is **always** best practice to run the output trees against the standard evalb program itself (http://nlp.cs.nyu.edu/evalb/), which is why the trees are also output to a file.

The greedy parser obtains the following results:

```
Labeled Recall:    86.9455
Labeled Precision: 86.6949
Labeled F1:        86.82
```

The beam search parser (with a beam size of 4) obtains the following results:

```
Labeled Recall:    88.2171
Labeled Precision: 88.0778
Labeled F1:        88.1474
```

Part of Speech Tagging

MeTA also provides models that can be used for part-of-speech tagging (http://en.wikipedia.org/wiki/Part-of-speech_tagging). These models, at the moment, are designed for tagging English text, but they should be able to be trained for any language desired once appropriate feature extractors are defined.

To **use** these models, you should download a tagger model file from the releases page (https://github.com/meta-toolkit/meta/releases) on the Github repository. MeTA currently has two different POS-tagger models available:

- A linear-chain conditional random field (`meta::sequence::crf` (/meta/doxygen/classmeta_1_1sequence_1_1crf.html))
- An averaged Perceptron, greedy tagger (`meta::sequence::perceptron` (/meta/doxygen/classmeta_1_1sequence_1_1perceptron.html))

Using Taggers

Using the CRF

First, extract your model files into a directory. You should modify your `config.toml` to contain a `[crf]` group like so:

```
[crf]
prefix = "path/to/crf/model/folder"
```

where `prefix` has been set to the folder that contains the model files you extracted.

Interactive tagging

You can interactively tag sentences using the provided `pos-tag` tool.

```
./pos-tag config.toml
```

This application will load the CRF model, and then proceed to tag sentences typed in at the prompt. You can stop tagging by simply inputting a blank line.

As an analyzer

The CRF model can also be used as an analyzer during index creation to create features based on n-grams of part-of-speech tags. To do so, you would need to add an analyzer group to your configuration that looks like the following:

```
[[analyzers]]
type = "ngram-pos"
ngram = 2 # the order n-gram you want to generate
filter = [{type = "icu-tokenizer"}, {type = "ptb-normalizer"}]
crf-prefix = "path/to/crf/model/folder"
```

You can alter the filter chain if you would like, but we strongly recommend sticking with the above setup as it is designed to match the original Penn Treebank tokenization format that the supplied model is trained on.

Programmatically

To use the CRF inside your own program, your code might look like this:

```
// load the model
meta::sequence::crf model{"path/to/crf/model/folder"};

// create a tagger
auto tagger = crf.make_tagger();

// load the sequence analyzer (for feature generation)
auto analyzer = meta::sequence::default_pos_analyzer();
analyzer.load(*crf_prefix);

meta::sequence::sequence seq;
// - code for loading/creating the sequence here -

// tag a sequence
const auto& ana = analyzer; // access the analyzer via const ref
                            // so that no new feature ids are generated
ana.analyze(seq);
tagger.tag(seq);

// print the tagged sequence
for (const auto& obs : seq)
    std::cout << obs.symbol() << "_" << analyzer.tag(obs.label()) << " ";
std::cout << "\n";
```

Have a look at the API documentation for the `meta::sequence::crf` class (/meta/doxygen/classmeta_1_1sequence_1_1crf.html) for more information.

Using the greedy tagger (Perceptron)

First, extract your model files into a directory. You should modify your `config.toml` to contain a `[sequence]` group like so:

```
[sequence]
prefix = "path/to/perceptron/model/folder/"
```

where `prefix` has been set to the folder that contains the model files you extracted.

Interactive tagging

The `pos-tag` tool doesn't currently use this tagger (patches welcome!), but you can still interactively tag sentences using the `profile` tool. See the `profile` tutorial (/meta/profile-tutorial.html) for a walkthrough of that demo application.

Programmatically

To use the greedy Perceptron-based tagger inside your own program, your code might look like this:

```
// load the model
meta::sequence::perceptron tagger{"path/to/perceptron/model/folder"};

meta::sequence::sequence seq;
// - code for loading/creating the sequence here -

// tag a sequence
tagger.tag(seq);

// print the tagged sequence
for (const auto& obs : seq)
    std::cout << obs.symbol() << "_" << obs.tag() << " ";
std::cout << "\n";
```

This API is a bit simpler than that of the CRF. For more information, you can check the API documentation for the `meta::sequence::perceptron` class (/meta/doxygen/classmeta_1_1sequence_1_1perceptron.html).

Training Taggers

In order to train your own models using our provided training programs, you will need to have a copy of the Penn Treebank (v2) extracted into your data prefix (see the overview tutorial (/meta/overview-tutorial.html)). Your folder structure should look like the following:

```
prefix
|---- penn-treebank
      |---- treebank-2
            |---- tagged
                  |---- wsj
                        |---- 00
                        |---- 01
                        ...
                        |---- 24
```

Training a CRF

To train your own CRF model from the Penn Treebank data, you should be able to use the provided `crf-train` executable. You will first need to adjust your `[crf]` group in your `config.toml` to look something like this:

```
[crf]
prefix = "desired/crf/model/location"
treebank = "penn-treebank" # relative to data prefix
corpus = "wsj"
section-size = 99
# these are the standard training/development/testing splits for POS tagging
train-sections = [0, 18]
dev-sections = [19, 21]
test-sections = [22, 24]
```

You should now be able to run the training procedure:

```
./crf-train config.toml
```

This will train a CRF model using the default training options. For more information on the options available, please see the API documentation for the `meta::sequence::crf` class (/meta/doxygen/classmeta_1_1sequence_1_1crf.html) (in particular, the parameters struct). If you would like to try different options, you can use the code provided in `src/sequence/crf/tools/crf_train.cpp` as a starting point. You will need to change the call to `crf.train()` to use a non-default `parameters` struct.

The model will take several hours to train. Its termination is based on convergence of the loss function.

Training a greedy Perceptron-based tagger

To train your own greedy tagger model from the Penn Treebank data, you should be able to use the provided `greedy-tagger-train` executable. You will need to first adjust your `[sequence]` group in your `config.toml` to look something like this (very similar to the above):

```
[sequence]
prefix = "desired/perceptron/model/location"
treebank = "penn-treebank" # relative to data prefix
corpus = "wsj"
section-size = 99
# these are the standard training/development/testing splits for POS tagging
train-sections = [0, 18]
dev-sections = [19, 21]
test-sections = [22, 24]
```

You should now be able to run the training procedure:

```
./greedy-tagger-train config.toml
```

This will train the averaged Perceptron model using the default training options. The termination criteria is simply a maximum iteration count, which defaults to 5 as of the time of writing. This means that the greedy tagger is **signifigantly** faster to train than its corresponding CRF model. In practice, the two achieve nearly the same accuracy with our default settings (the CRF being just slightly better).

If you want to adjust the number of training iterations, you can use the code provided in `src/sequence/tools/greedy_tagger_train.cpp` as a starting point. You will need to change the call to `tagger.train()` to use a non-default `training_options` struct.

Testing Taggers

If you follow the instructions above for the tagger type you wish to test, you should be able to test them with their corresponding testing executables. For the CRF, you would use

```
./crf-test config.toml
```

and for the greedy Perceptron-based tagger, you would use

```
./greedy-tagger-test config.toml
```

Both will run over the testing section defined in `config.toml` and report precision, recall, and F1 score for each class, as well as the overall token-level accuracy. The current CRF model achieves 97% accuracy, and the greedy Perceptron model achieves 96.9% accuracy.

Search

Search in MeTA

At the heart of MeTA lies its indexes. Every piece of data that MeTA's algorithms operate on must first reside in an index. In this tutorial, we'll walk you through how to create your own `inverted_index` and how to use your `inverted_index` to create a search engine.

Initially setting up the config file

By default, there are several fields at the top of your config file. These are general settings that deal with paths:

```
prefix = "/home/sean/projects/meta-data/"
libsvm-modules = "../deps/libsvm-modules/"
punctuation = "../data/sentence-boundaries/sentence-punctuation.txt"
stop-words = "../data/lemur-stopwords.txt"
start-exceptions = "../data/sentence-boundaries/sentence-start-exceptions.txt"
end-exceptions = "../data/sentence-boundaries/sentence-end-exceptions.txt"
function-words = "../data/function-words.txt"
```

I have set `prefix` to be where I store my datasets. You can leave the other paths as they are; they'll work by default.

Next, we have a section for the corpus (20newsgroups) and its analyzer.

```
corpus-type = "line-corpus"
dataset = "20newsgroups"
#list = "20news"
forward-index = "20newsgroups-fwd"
inverted-index = "20newsgroups-inv"
encoding = "latin-1"

[[analyzers]]
method = "ngram-word"
ngram = 1
filter = "default-chain"
```

`corpus-type` tells MeTA whether our dataset is a `line-corpus` (one doc per line in a large file) or a `file-corpus` (each doc is an individual file). If you're using a `file-corpus`, you need to specify the `list` parameter (commented out above) so MeTA can find the file `20news-full-corpus.txt` in the dataset directory.

If the executable needs to search the index, we can add the following to say which ranker to use:

```
[ranker]
method = "bm25"
```

For all the premade ranking functions, you can additionally specify any of the ranking function parameters:

```
[ranker]
method = "bm25"
k1 = 1.2
b = 0.75
k3 = 500
```

For a list of MeTA's ranking functions, see the index::ranker (http://meta-toolkit.github.io/meta/doxygen/classmeta_1_1index_1_1ranker.html) documentation.

Using the example apps

Now that we know how to set up our config file, let's experiment with some indexes. **It's possible to index and search a corpus without writing any code!** In the root directory of MeTA, there are several example apps that deal with creating indexes for search engines or information retrieval tasks:

- `index.cpp` : indexes a corpus
- `interactive-search.cpp` : searches for documents based on user input
- `query-runner.cpp` : runs a list of queries specified by a file

Let's go through these one by one:

Index

```
./index config.toml
```

You can then see it being indexed. After it's done, some stats are printed out, such as number of documents, average document length, and number of unique terms. If you run it again, it will just print out the stats since it detects that the index has already been created.

Interactive Search

```
./interactive-search config.toml
```

Running this will first create an index if one does not exist. Then, it will prompt the user to type a query. The current analyzer is run on the text query to turn it into a document, and then that document is used to search. The top results are returned.

Query Runner

For this app, we need to add a path to the query file.

```
querypath = "./"
```

The query file contains one query per line, and they are run on the current index. You quickly make your own query file in the current directory if you don't have one. It must be named `dataset-queries.txt` , where in our example `dataset` is `20newsgroups` .

```
./query-runner config.toml
```

The top 10 documents for each query are then displayed along with the time taken to run all the queries.

Writing code

Now we'll look at some examples of how to create and search an index. All this code can be found in the demo apps as well.

Creating an index

This is the simplest way to make an inverted index:

```
auto idx = index::make_index<index::inverted_index>(argv[1]);
```

This assumes that `argv[1]` is the path to the TOML config file, which is standard MeTA practice.

Warning: this index does not have a cache! A cache is used when running queries on the index. The cache saves frequently used `postings_data` objects so we don't need to do a disk seek for every term while scoring.

Here are two different ways to specify a cache:

```
// Create an inverted index using a DBLRU cache. The arguments forwarded to
//  make_index are the config file for the index and any parameters for the
//  cache. In this case, we set the maximum hash table size for the
//  dblru_cache to be 10000.
auto idx = index::make_index<index::dblru_inverted_index>(argv[1], 10000);

// Create an inverted index using a splay cache. The arguments forwarded
//  to make_index are the config file for the index and any parameters
//  for the cache. In this case, we set the maximum number of nodes in
//  the splay_cache to be 10000.
auto idx = index::make_index<index::splay_inverted_index>(argv[1], 10000);
```

Searching an index

Now that we have an index, let's search it! First, we need to create a `ranker`:

```
auto config = cpptoml::parse_file(argv[1]);
auto group = config.get_table("ranker");
if (!group)
    throw std::runtime_error{"\"ranker\" group needed in config file!"};
auto ranker = index::make_ranker(*group);
```

We can also manually specify our `ranker` instead of reading from the config file:

```
index::okapi_bm25 ranker{1.2, 0.75, 500.0};
```

We don't have to specify the arguments for the ranking parameters, but we did anyway.

Finally, we are able to search our index with our ranker. Let's assume we used `make_ranker` to create the ranker, so the `ranker` object is a `std::unique_ptr` to the actual object.

```
corpus::document query{"[doc path]", doc_id{0}};
query.content("my query text");

auto ranking = ranker->score(*idx, query);
```

`ranking` is a sorted vector of `pair`s of (`doc_id`, `double`). By default, the top 10 documents are returned. To return a different number, just add a third parameter:

```
auto ranking = ranker->score(*idx, query, 25);
```

Iterate through `ranking` and display your results! Check out the index::ir_eval (http://meta-toolkit.github.io/meta/doxygen/classmeta_1_1index_1_1ir__eval.html) class if you have relevance judgements for your queries.

Writing your own ranker

Perhaps you'd like to experiment and write your own ranking function. Any rankers you write should subclass `index::ranker` and implement the pure virtual function `score_one(const score_data& sd)`. `score_data` is a struct that contains useful information to ranking functions. Read up on it here (http://meta-toolkit.github.io/meta/doxygen/structmeta_1_1index_1_1score__data.html).

If you would like to be able to create your ranker by specifying it in a configuration file, you will need to provide a public static string id member that specifies the text that identifies your ranker class, and register it with the toolkit somewhere in main() like this:

```
meta::index::register_ranker<my_ranker>();
```

If you need to read parameters from the config group, you should specialize the `make_ranker()` function as follows:

```
namespace meta
{
namespace index
{
template <>
std::unique_ptr<ranker> make_ranker<my_ranker>(const cpptoml::table& config)
{
    // read config file
    // set default params if config options not found (for example)
}
}
}
```

Documentation for MeTA: ModErn Text Analysis (https://github.com/meta-toolkit/meta)

Topic Models

Topic Modeling in MeTA

Topic modeling has become an integral part of any text-analyst's toolbox, and MeTA will allow you to explore generating topics for any of your corpora in a general way. It supports several different flavors of inference for the LDA topic model: see the API documentation for `lda_model` for a list (doxygen/classmeta_1_1topics_1_1lda__model.html). At the time of writing, the following inference methods are present:

- `lda_cvb` , which uses collapsed variational bayes,
- `lda_gibbs` , which uses collapsed Gibbs sampling, and
- `parallel_lda_gibbs` , which uses an approximation of collapsed Gibbs sampling, but does so exploiting multi-processor systems

To run topic the topic modeling application bundled with MeTA, you need to configure the `lda_model` in your configuration file. Here is an example configuration:

```
[lda]
inference = "cvb" # or gibbs, or pargibbs
max-iters = 1000
alpha = 0.1
beta = 0.1
topics = 10
model-prefix = "lda-model"
```

(For more information on the parameters, please see the documentation for the corresponding inference method you've chosen; for instance, collapsed variational bayes (doxygen/classmeta_1_1topics_1_1lda__cvb.html), Gibbs sampling (doxygen/classmeta_1_1topics_1_1lda__gibbs.html), or approximate parallel Gibbs sampling (doxygen/classmeta_1_1topics_1_1parallel__lda__gibbs.html).)

Running `./lda config.toml` will now run your chosen inference method for either the maximum number of specified iterations, or until the model determines it has converged (whichever one is sooner). Because the number of iterations can't be known ahead of time, progress output is given on a per-iteration basis only, and each different inference model may print out inference-specific summary information after each iteration (for example, for the Gibbs samplers or the maximum for collapsed variational bayes).

Viewing the Topics Found

MeTA bundles an executable `./lda-topics` that can report the top words in each topic found during inference. To use it, you need to specify the configuration file used to do inference, the path to the model's `.phi` file (which stores for each), and the number of words per topic to output. Below is some sample output for a simple dataset where we found two topics and used the default filter chain for text analysis (which includes `porter2_stemmer`).

```
$ ./lda-topics config.toml lda-model.phi 15
Topic 0:
-----------------------
smoke (3274): 0.391293
smoker (3276): 0.0849028
restaur (3006): 0.0825975
ban (259): 0.0507666
cigarett (584): 0.0261952
non (2445): 0.0232204
health (1726): 0.0188438
seat (3134): 0.017816
smell (3267): 0.0168126
harm (1709): 0.0162674
peopl (2649): 0.0154918
complet (690): 0.0151501
japan (2010): 0.0137623
nonsmok (2448): 0.0128775
tobacco (3638): 0.0120414

Topic 1:
-----------------------
job (2019): 0.156795
part (2607): 0.127387
student (3443): 0.12414
colleg (652): 0.0905726
time (3629): 0.0708495
money (2341): 0.064298
work (4013): 0.0372687
studi (3444): 0.0274538
learn (2110): 0.0253973
import (1857): 0.0167224
earn (1076): 0.015446
experi (1257): 0.0141118
school (3119): 0.00852317
parent (2602): 0.00835476
societi (3296): 0.00793623
```

Documentation for MeTA: ModErn Text Analysis (https://github.com/meta-toolkit/meta)

meta::index Namespace Reference

Indexes to create efficient representations of data. More...

Classes

class **absolute_discount**
Implements the absolute discounting smoothing method. More...

class **cached_index**
Decorator class for wrapping indexes with a cache. More...

class **chunk**
Represents a portion of a **disk_index**'s postings file. More...

class **chunk_handler**
An interface for writing and merging inverted chunks of **postings_data** for a **disk_index**. More...

class **dirichlet_prior**
Implements Bayesian smoothing with a Dirichlet prior. More...

class **disk_index**
Holds generic data structures and functions that **inverted_index** and **forward_index** both use. More...

class **forward_index**
The **forward_index** stores information on a corpus by doc_ids. More...

class **inverted_index**
The **inverted_index** class stores information on a corpus indexed by term_ids. More...

class **ir_eval**
Evaluates lists of ranked documents returned from a search engine; can give stats per-query (e.g. More...

class **jelinek_mercer**
Implements the Jelinek-Mercer smoothed ranking model. More...

class **language_model_ranker**
Scores documents according to one of three different smoothed language model scoring methods described in "A Study of Smoothing Methods for Language Models Applied to Ad Hoc Information Retrieval" by Zhai and Lafferty, 2001. More...

class **okapi_bm25**
The Okapi BM25 scoring function. More...

class **pivoted_length**
The pivoted document length normalization ranking function. More...

class **postings_data**
A class to represent the per-PrimaryKey data in an index's postings file. More...

class **ranker**
A ranker scores a query against all the documents in an inverted index, returning a list of documents sorted by relevance. More...

class **ranker_factory**
Factory that is responsible for creating rankers from configuration files. More...

struct **score_data**
A **score_data** object contains information needed to evaluate a ranking function. More...

class **string_list**
A class designed for reading large lists of strings that have been persisted to disk. More...

class **string_list_writer**
A class for writing large lists of strings to disk with an associated index file for fast random access. More...

class **vocabulary_map**
A read-only view of a B+-tree-like structure that stores the vocabulary for an index. More...

class **vocabulary_map_writer**
A class that writes the B+-tree-like data structure used for storing the term id mapping in an index. More...

Typedefs

using **dblru_inverted_index** = **cached_index**< **inverted_index**, **caching::default_dblru_cache** >
Inverted index using default DBLRU cache.

using **splay_inverted_index** = **cached_index**< **inverted_index**, **caching::splay_cache** >
Inverted index using splay cache.

using **memory_forward_index** = **cached_index**< **forward_index**, **caching::no_evict_cache** >
In-memory forward index.

using **dblru_forward_index** = **cached_index**< **forward_index**, **caching::default_dblru_cache** >
Forward index using default DBLRU cache.

using **splay_forward_index** = **cached_index**< **forward_index**, **caching::splay_cache** >
Forward index using splay cache.

Enumerations

enum **index_file** {
DOC_IDS_MAPPING = 0, **DOC_IDS_MAPPING_INDEX**, **DOC_SIZES**, **DOC_LABELS**, **DOC_UNIQUETERMS**, **LABEL_IDS_MAPPING**, **POSTINGS**, **TERM_IDS_MAPPING**, **TERM_IDS_MAPPING_INVERSE**
}
Collection of all the files that comprise a disk_index.

Functions

template<class Index , class... Args>

std::shared_ptr< Index > **make_index** (const std::string &config_file, Args &&...args)
Factory method for creating indexes. More...

template<class Index , template< class, class > class Cache, class... Args>

std::shared_ptr< **cached_index**< Index, Cache > > **make_index** (const std::string &config_file, Args &&...args)
Factory method for creating indexes that are cached. More...

template<class PrimaryKey , class SecondaryKey >

io::compressed_file_reader & **operator>>** (**io::compressed_file_reader** &in, **postings_data**< PrimaryKey, SecondaryKey > &pd)
Reads semi-compressed postings data from a compressed file. More...

template<>

io::compressed_file_reader & **operator>>** (**io::compressed_file_reader** &in, **postings_data**< std::string, doc_id > &pd)
Reads semi-compressed postings data from a compressed file. More...

template<class PrimaryKey , class SecondaryKey >

bool **operator==** (const **postings_data**< PrimaryKey, SecondaryKey > &lhs, const **postings_data**< PrimaryKey, SecondaryKey > &rhs)

template<>

std::unique_ptr< **ranker** > **make_ranker< absolute_discount >** (const cpptoml::table &)
Specialization of the factory method used to create **absolute_discount** rankers.

template<>

std::unique_ptr< **ranker** > **make_ranker< dirichlet_prior >** (const cpptoml::table &)
Specialization of the factory method used to create **dirichlet_prior** rankers.

template<>

std::unique_ptr< **ranker** > **make_ranker< jelinek_mercer >** (const cpptoml::table &)
Specialization of the factory method used to create **jelinek_mercer** rankers.

template<>	
std::unique_ptr< **ranker** >	**make_ranker< okapi_bm25 >** (const cpptoml::table &) Specialization of the factory method used to create **okapi_bm25** rankers.
template<>	
std::unique_ptr< **ranker** >	**make_ranker< pivoted_length >** (const cpptoml::table &) Specialization of the factory method used to create **pivoted_length** rankers.
std::unique_ptr< **ranker** >	**make_ranker** (const cpptoml::table &) Convenience method for creating a ranker using the factory. More...
template<class Ranker >	
void	**register_ranker** () Registration method for rankers. More...

Detailed Description

Indexes to create efficient representations of data.

Function Documentation

template<class Index , class... Args>

std::shared_ptr<Index> meta::index::make_index (const std::string & config_file,
Args &&... args
)

Factory method for creating indexes.

inverted_index is a friend of the factory method used to create cached versions of it.

forward_index is a friend of the factory method used to create cached versions of it.

forward_index is a friend of the factory method used to create it.

Usage:

```
auto idx = index::make_index<derived_index_type>(config_path);
```

Parameters

config_file The path to the configuration file to be used to build the index

args any additional arguments to forward to the constructor for the chosen index type (usually none)

Returns

A properly initialized index

template<class Index , template< class, class > class Cache, class... Args>

std::shared_ptr<cached_index<Index, Cache> >
meta::index::make_index **(const std::string & config_file,**
Args &&... args
)

Factory method for creating indexes that are cached.

inverted_index is a friend of the factory method used to create cached versions of it.

forward_index is a friend of the factory method used to create cached versions of it.

forward_index is a friend of the factory method used to create it.

Usage:

```
auto idx =
    index::make_index<dervied_index_type,
                      cache_type>(config_path, other, options);
```

Other options will be forwarded to the constructor for the chosen cache class.

Parameters

config_file the path to the configuration file to be used to build the index.

args any additional arguments to forward to the constructor for the cache class chosen

Returns

A properly initialized, and automatically cached, index.

template<class PrimaryKey , class SecondaryKey >

io::compressed_file_reader &
meta::index::operator>> **(io::compressed_file_reader & in,**
postings_data< PrimaryKey, SecondaryKey > & pd
)

Reads semi-compressed postings data from a compressed file.

Parameters

in The stream to read from

pd The postings data object to write the stream info to

Returns

the input stream

template<>

io::compressed_file_reader&
meta::index::operator>> **(io::compressed_file_reader & in,**
postings_data< std::string, doc_id > & pd
) inline

Reads semi-compressed postings data from a compressed file.

Parameters

in The stream to read from

pd The postings data object to write the stream info to

Returns

the input stream

template<class PrimaryKey , class SecondaryKey >

bool meta::index::operator== (const postings_data< PrimaryKey, SecondaryKey > & lhs,
const postings_data< PrimaryKey, SecondaryKey > & rhs
)

Parameters

lhs The first **postings_data**

rhs The **postings_data** to compare with

Returns

whether this **postings_data** has the same PrimaryKey as the paramter

std::unique_ptr< ranker > meta::index::make_ranker (const cpptoml::table & config)

Convenience method for creating a ranker using the factory.

Factory method for creating a ranker.

This should be specialized if your given ranker requires special construction behavior (e.g., reading parameters).

template<class Ranker >

void meta::index::register_ranker ( )

Registration method for rankers.

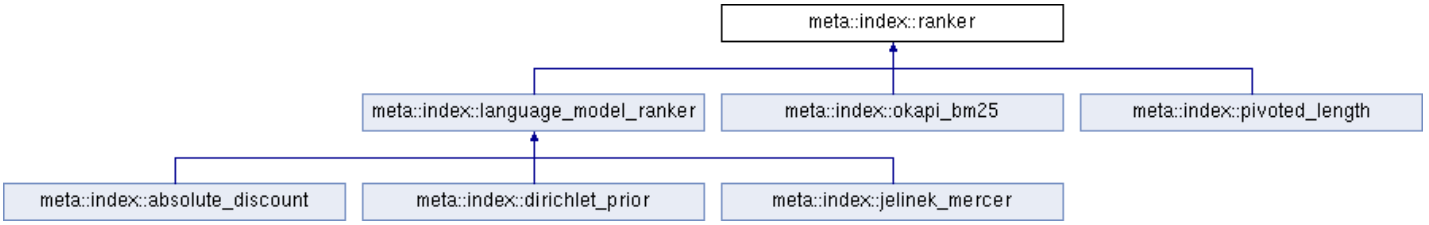
Clients should use this method to register any new rankers they write.

meta::index::ranker Class Reference abstract

A ranker scores a query against all the documents in an inverted index, returning a list of documents sorted by relevance. More...

`#include <ranker.h>`

Inheritance diagram for meta::index::ranker:



Public Member Functions

std::vector< std::pair< doc_id, double > >	**score** (**inverted_index** &idx, **corpus::document** &query, uint64_t num_results=10, const std::function< bool(doc_id d_id)> &filter=[](doc_id){return true;})
virtual double	**score_one** (const **score_data** &sd)=0 Computes the contribution to the score of a document for a matched query term. More...
virtual double	**initial_score** (const **score_data** &sd) const Computes the constant contribution to the score of a particular document. More...
virtual	**~ranker** ()=default Default destructor.

Private Attributes

std::vector< double >	**results_** results per doc_id

Detailed Description

A ranker scores a query against all the documents in an inverted index, returning a list of documents sorted by relevance.

Member Function Documentation

std::vector< std::pair< doc_id, double > >
meta::index::ranker::score (**inverted_index &** **idx,**
corpus::document & **query,**
uint64_t **num_results = 10,**
const std::function< bool(doc_id d_id)> & **filter = [](doc_id) { return true; }**
)

Parameters

idx	The index this ranker is operating on
query	The current query
num_results	The number of results to return in the vector
filter	A filtering function to apply to each doc_id; returns true if the document should be included in results

virtual double meta::index::ranker::score_one (const score_data & sd) pure virtual

Computes the contribution to the score of a document for a matched query term.

Parameters

sd The score_data for this query

Implemented in **meta::index::okapi_bm25**, **meta::index::pivoted_length**, and **meta::index::language_model_ranker**.

double meta::index::ranker::initial_score (const score_data & sd) const virtual

Computes the constant contribution to the score of a particular document.

Parameters

sd The score_data for the query

Reimplemented in **meta::index::language_model_ranker**.

The documentation for this class was generated from the following files:

- /home/chase/projects/meta/include/index/ranker/**ranker.h**
- /home/chase/projects/meta/src/index/ranker/**ranker.cpp**

meta::index::score_data Struct Reference

A **score_data** object contains information needed to evaluate a ranking function. More...

`#include <score_data.h>`

Public Member Functions

score_data (**inverted_index** &p_idx, double p_avg_dl, uint64_t p_num_docs, uint64_t p_total_terms, const **corpus::document** &p_query)
Constructor to initialize most elements. More...

Public Attributes

inverted_index &	**idx** index queries are running on
double	**avg_dl** average document length
uint64_t	**num_docs** total number of documents
uint64_t	**total_terms** total number of terms in the index
const **corpus::document** &	**query** the current query
term_id	**t_id** doc term id
uint64_t	**query_term_count** query term count
uint64_t	**doc_count** number of docs that t_id appears in
uint64_t	**corpus_term_count** number of times t_id appears in corpus
doc_id	**d_id** document id
uint64_t	**doc_term_count** number of times the term appears in the current doc
uint64_t	**doc_size** total number of terms in the doc
uint64_t	**doc_unique_terms**

number of unique terms in the doc

Detailed Description

A **score_data** object contains information needed to evaluate a ranking function.

Data is set by the base ranker class as needed, so the derived ranking classes don't make many unncessary calls to the inverted index.

Constructor & Destructor Documentation

```
meta::index::score_data::score_data ( inverted_index &          p_idx,
                                      double                    p_avg_dl,
                                      uint64_t                  p_num_docs,
                                      uint64_t                  p_total_terms,
                                      const corpus::document &  p_query
                                    )                                          inline
```

Constructor to initialize most elements.

Parameters

p_idx	The index that is being used
p_avg_dl	The average doc length in the index
p_num_docs	The number of docs in the index
p_total_terms	The total number of terms in the index
p_query	The current query

The documentation for this struct was generated from the following file:

- /home/chase/projects/meta/include/index/**score_data.h**

meta::topics::lda_cvb Class Reference

lda_cvb: An implementation of LDA that uses collapsed variational bayes for inference. More...

`#include <lda_cvb.h>`

Inheritance diagram for meta::topics::lda_cvb:

Public Member Functions

lda_cvb (std::shared_ptr< **index::forward_index** > idx, uint64_t num_topics, double alpha, double beta)
Constructs the lda model over the given documents, with the given number of topics, and hyperparameters \(\alpha\) and \(\beta\) for the priors on \(\phi\) (topic distributions) and \(\theta\) (topic proportions), respectively. More...

virtual **~lda_cvb** ()=default
Destructor: virtual for potential subclassing.

void **run** (uint64_t num_iters, double convergence=1e-3)
Runs the variational inference algorithm for a maximum number of iterations, or until the given convergence criterion is met. More...

Public Member Functions inherited from meta::topics::lda_model

Protected Member Functions

void **initialize** ()
Initializes the parameters randomly.

double **perform_iteration** (uint64_t iter)
Performs one iteration of the inference algorithm. More...

virtual double **compute_term_topic_probability** (term_id term, topic_id topic) const override

virtual double **compute_doc_topic_probability** (doc_id doc, topic_id topic) const override

Protected Member Functions inherited from meta::topics::lda_model

Protected Attributes

std::vector< std::vector< **stats::multinomial**< topic_id > > > **gamma_**
Variational distributions \(\gamma_{ij}\), which represent the soft topic assignments for each word occurrence \(i\) in document \(j\).

	More...
std::vector< **stats::multinomial**< term_id > >	**phi_** The word distributions for each topic, \(\phi_t\).
std::vector< **stats::multinomial**< topic_id > >	**theta_** The topic distributions for each document, \(\theta_d\).

▸ **Protected Attributes inherited from meta::topics::lda_model**

Detailed Description

lda_cvb: An implementation of LDA that uses collapsed variational bayes for inference.

Specifically, it uses the CVB0 algorithm detailed in Asuncion et. al.

See also

http://www.ics.uci.edu/~asuncion/pubs/UAI_09.pdf

Constructor & Destructor Documentation

meta::topics::lda_cvb::lda_cvb (**std::shared_ptr< index::forward_index > idx,**
uint64_t num_topics,
double alpha,
double beta
)

Constructs the lda model over the given documents, with the given number of topics, and hyperparameters \(\alpha\) and \(\beta\) for the priors on \(\phi\) (topic distributions) and \(\theta\) (topic proportions), respectively.

Parameters

idx	The index containing the documents to model
num_topics	The number of topics to infer
alpha	The hyperparameter for the Dirichlet prior over \(\phi\)
beta	The hyperparameter for the Dirichlet prior over \(\theta\)

Member Function Documentation

void meta::topics::lda_cvb::run (uint64_t num_iters,
double convergence = 1e-3
) virtual

Runs the variational inference algorithm for a maximum number of iterations, or until the given convergence criterion is met.

The convergence criterion is determined as the maximum difference in any of the variational parameters \(\gamma_{dij}\) in a given iteration.

Parameters

num_iters The maximum number of iterations to run the sampler for

convergence The lowest maximum difference in any \(\gamma_{dij}\) to be allowed before considering the inference to have converged

Implements **meta::topics::lda_model**.

double meta::topics::lda_cvb::perform_iteration (uint64_t iter) protected

Performs one iteration of the inference algorithm.

Parameters

iter The current iteration number

Returns

the maximum change in any of the \(\gamma_{dij}\)s

double
meta::topics::lda_cvb::compute_term_topic_probability (term_id term,
topic_id topic
) const override protected virtual

Returns

the probability that the given term appears in the given topic

Parameters

term The term we are concerned with

topic The topic we are concerned with

Implements **meta::topics::lda_model**.

double
meta::topics::lda_cvb::compute_doc_topic_probability (doc_id doc,
topic_id topic
) const override protected virtual

Returns

the probability that the given topic is picked for the given document

Parameters

doc The document we are concerned with

topic The topic we are concerned with

Implements **meta::topics::lda_model**.

Member Data Documentation

std::vector<std::vector<stats::multinomial<topic_id> > >
meta::topics::lda_cvb::gamma_ protected

Variational distributions \(\gamma_{ij}\), which represent the soft topic assignments for each word occurrence \(i\) in document \(j\).

Indexed as gamma_[d][i]

The documentation for this class was generated from the following files:

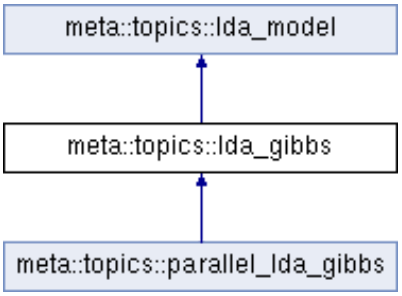
- /home/chase/projects/meta/include/topics/**lda_cvb.h**
- /home/chase/projects/meta/src/topics/**lda_cvb.cpp**

meta::topics::lda_gibbs Class Reference

A LDA topic model implemented using a collapsed gibbs sampler. More...

`#include <lda_gibbs.h>`

Inheritance diagram for meta::topics::lda_gibbs:



Public Member Functions

lda_gibbs (std::shared_ptr< **index::forward_index** > idx, uint64_t num_topics, double alpha, double beta)
Constructs the lda model over the given documents, with the given number of topics, and hyperparameters \(\alpha\) and \(\beta\) for the priors on \(\phi\) (topic distributions) and \(\theta\) (topic proportions), respectively. More...

virtual **~lda_gibbs** ()=default
Destructor: virtual for potential subclassing.

virtual void **run** (uint64_t num_iters, double convergence=1e-6)
Runs the sampler for a maximum number of iterations, or until the given convergence criterion is met. More...

Public Member Functions inherited from meta::topics::lda_model

Protected Member Functions

topic_id **sample_topic** (term_id term, doc_id doc)
Samples a topic from the full conditional distribution \(P(z_i = j | w, \boldsymbol{z})\). More...

virtual double **compute_sampling_weight** (term_id term, doc_id doc, topic_id topic) const
Computes a weight proportional to \(P(z_i = j | w, \boldsymbol{z})\). More...

virtual double **compute_term_topic_probability** (term_id term, topic_id topic) const override

virtual double **compute_doc_topic_probability** (doc_id doc, topic_id topic) const override

virtual void **initialize** ()
Initializes the first set of topic assignments for inference. More...

virtual void **perform_iteration** (uint64_t iter, bool init=false)
Performs a sampling iteration. More...

virtual void **decrease_counts** (topic_id topic, term_id term, doc_id doc)

	Decreases all counts associated with the given topic, term, and document by one. More...
virtual void	**increase_counts** (topic_id topic, term_id term, doc_id doc) Increases all counts associated with the given topic, term, and document by one. More...
double	**corpus_log_likelihood** () const
lda_gibbs &	**operator=** (const **lda_gibbs** &)=delete **lda_gibbs** cannot be copy assigned.
	lda_gibbs (const **lda_gibbs** &other)=delete **lda_gibbs** cannot be copy constructed.

▸ **Protected Member Functions inherited from meta::topics::lda_model**

Protected Attributes

std::vector< std::vector< topic_id > >	**doc_word_topic_** The topic assignment for every word in every document. More...
std::vector< **stats::multinomial**< term_id > >	**phi_** The word distributions for each topic, \(\phi_t\).
std::vector< **stats::multinomial**< topic_id > >	**theta_** The topic distributions for each document, \(\theta_d\).
std::mt19937_64	**rng_** The random number generator for the sampler.

▸ **Protected Attributes inherited from meta::topics::lda_model**

Detailed Description

A LDA topic model implemented using a collapsed gibbs sampler.

See also

http://www.pnas.org/content/101/suppl.1/5228.full.pdf

Constructor & Destructor Documentation

```
meta::topics::lda_gibbs::lda_gibbs ( std::shared_ptr< index::forward_index > idx,
                                     uint64_t                                num_topics,
                                     double                                  alpha,
                                     double                                  beta
                                   )
```

Constructs the lda model over the given documents, with the given number of topics, and hyperparameters \(\alpha\) and \(\beta\) for the priors on \(\phi\) (topic distributions) and \(\theta\) (topic proportions), respectively.

Parameters

idx	The index that contains the documents to model
num_topics	The number of topics to infer
alpha	The hyperparameter for the Dirichlet prior over \(\phi\)
beta	The hyperparameter for the Dirichlet prior over \(\theta\)

Member Function Documentation

```
void meta::topics::lda_gibbs::run ( uint64_t num_iters,
                                    double   convergence = 1e-6
                                  )                                    virtual
```

Runs the sampler for a maximum number of iterations, or until the given convergence criterion is met.

The convergence criterion is determined as the relative difference in log corpus likelihood between two iterations.

Parameters

num_iters	The maximum number of iterations to run the sampler for
convergence	The lowest relative difference in \(\log P(\mathbf{w} \mid \mathbf{z})\) to be allowed before considering the sampler to have converged

Implements **meta::topics::lda_model**.

topic_id meta::topics::lda_gibbs::sample_topic (term_id term,
doc_id doc
) protected

Samples a topic from the full conditional distribution \(P(z_i = j | w, \boldsymbol{z})\).

Used in both initialization and each normal iteration of the sampler, after removing the current value of \(z_i\) from the vector of assignments \(\boldsymbol{z}\).

Parameters

term The term we are sampling a topic assignment for

doc The document the term resides in

Returns

the topic sampled the given (term, doc) pair

double meta::topics::lda_gibbs::compute_sampling_weight (term_id term,
doc_id doc,
topic_id topic
) **const** protected virtual

Computes a weight proportional to \(P(z_i = j | w, \boldsymbol{z})\).

Parameters

term The current word we are sampling for

doc The document in which the term resides

topic The topic \(j\) we want to compute the probability for

Returns

a weight proportional to the probability that the given term in the given document belongs to the given topic

Reimplemented in **meta::topics::parallel_lda_gibbs**.

double
meta::topics::lda_gibbs::compute_term_topic_probability (term_id term,
topic_id topic
) const override protected virtual

Returns

the probability that the given term appears in the given topic

Parameters

term The term we are concerned with.

topic The topic we are concerned with.

Implements **meta::topics::lda_model**.

double
meta::topics::lda_gibbs::compute_doc_topic_probability (doc_id doc,
topic_id topic
) const override protected virtual

Returns

the probability that the given topic is picked for the given document

Parameters

doc The document we are concerned with.

topic The topic we are concerned with.

Implements **meta::topics::lda_model**.

void meta::topics::lda_gibbs::initialize ( ) protected virtual

Initializes the first set of topic assignments for inference.

Employs an online application of the sampler where counts are only considered for the words observed so far through the loop.

Reimplemented in **meta::topics::parallel_lda_gibbs**.

void meta::topics::lda_gibbs::perform_iteration (uint64_t iter,
bool init = false
) protected virtual

Performs a sampling iteration.

Parameters

iter The iteration number

init Whether or not to employ the online method (defaults to `false`)

Reimplemented in **meta::topics::parallel_lda_gibbs**.

void meta::topics::lda_gibbs::decrease_counts (topic_id topic,
term_id term,
doc_id doc
) protected virtual

Decreases all counts associated with the given topic, term, and document by one.

Parameters

topic The topic in question

term The term in question

doc The document in question

Reimplemented in **meta::topics::parallel_lda_gibbs**.

void meta::topics::lda_gibbs::increase_counts (topic_id topic,
term_id term,
doc_id doc
) protected virtual

Increases all counts associated with the given topic, term, and document by one.

Parameters

topic The topic in question

term The term in question

doc The document in question

Reimplemented in **meta::topics::parallel_lda_gibbs**.

double meta::topics::lda_gibbs::corpus_log_likelihood ( ) const protected

Returns

\(\log P(\mathbf{w} \mid \mathbf{z})\)

Member Data Documentation

std::vector<std::vector<topic_id> > meta::topics::lda_gibbs::doc_word_topic_ protected

The topic assignment for every word in every document.

Note that the same word occurring multiple times in one document could potentially have many different topics assigned to it, so we are not using term_ids here, but our own contrived intra document term id.

Indexed as [doc_id][position].

The documentation for this class was generated from the following files:

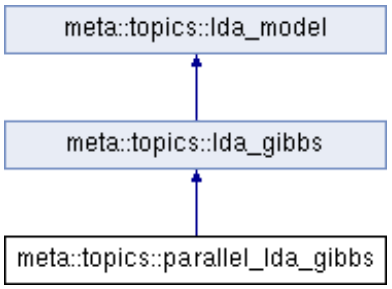
- /home/chase/projects/meta/include/topics/**lda_gibbs.h**
- /home/chase/projects/meta/src/topics/**lda_gibbs.cpp**

meta::topics::parallel_lda_gibbs Class Reference

An LDA topic model implemented using the Approximate Distributed LDA algorithm. More...

`#include <parallel_lda_gibbs.h>`

Inheritance diagram for meta::topics::parallel_lda_gibbs:



Public Member Functions

virtual **~parallel_lda_gibbs** ()=default
Destructor: virtual for potential subclassing.

▸ **Public Member Functions inherited from meta::topics::lda_gibbs**

▸ **Public Member Functions inherited from meta::topics::lda_model**

Protected Member Functions

virtual void **initialize** () override
Initializes the first set of topic assignments for inference. More...

virtual void **perform_iteration** (uint64_t iter, bool init=false) override
Performs a sampling iteration of the AD-LDA algorithm. More...

virtual void **decrease_counts** (topic_id topic, term_id term, doc_id doc) override
Decreases all counts associated with the given topic, term, and document by one. More...

virtual void **increase_counts** (topic_id topic, term_id term, doc_id doc) override
Increases all counts associated with the given topic, term, and document by one. More...

virtual double **compute_sampling_weight** (term_id term, doc_id doc, topic_id topic) const override
Computes a weight proportional to \(P(z_i = j | w, \boldsymbol{z})\). More...

▸ **Protected Member Functions inherited from meta::topics::lda_gibbs**

▸ **Protected Member Functions inherited from meta::topics::lda_model**

Protected Attributes

parallel::thread_pool **pool_**
The thread pool used for parallelization.

std::unordered_map< std::thread::id, std::vector< **stats::multinomial**< term_id > > > **phi_diffs_**

Stores the difference in topic_term counts on a per-thread basis for use in the reduction step. More...

- **Protected Attributes inherited from meta::topics::lda_gibbs**
- **Protected Attributes inherited from meta::topics::lda_model**

Detailed Description

An LDA topic model implemented using the Approximate Distributed LDA algorithm.

Based on the algorithm detailed by David Newman et. al.

See also

http://www.jmlr.org/papers/volume10/newman09a/newman09a.pdf

Member Function Documentation

void meta::topics::parallel_lda_gibbs::initialize ( ) override protected virtual

Initializes the first set of topic assignments for inference.

Employs an online application of the sampler where counts are only considered for the words observed so far through the loop.

Reimplemented from **meta::topics::lda_gibbs**.

```
void
meta::topics::parallel_lda_gibbs::perform_iteration ( uint64_t iter,
                                                      bool     init = false
                                                    )
```
override protected virtual

Performs a sampling iteration of the AD-LDA algorithm.

This consists of splitting up the sampling of (document, word) topic assignments across threads, keeping for each thread a difference in counts for the potentially shared topic counts. Once the sampling has finished, the counts are reduced down (serially) before the iteration is completed.

Parameters

iter The current iteration number

init Whether or not this iteration should use the online method for initializing the sampler

Reimplemented from **meta::topics::lda_gibbs**.

```
void meta::topics::parallel_lda_gibbs::decrease_counts ( topic_id topic,
                                                         term_id  term,
                                                         doc_id   doc
                                                       )
```
override protected virtual

Decreases all counts associated with the given topic, term, and document by one.

Parameters

topic The topic in question

term The term in question

doc The document in question

Reimplemented from **meta::topics::lda_gibbs**.

void meta::topics::parallel_lda_gibbs::increase_counts (topic_id topic,
term_id term,
doc_id doc
) override protected virtual

Increases all counts associated with the given topic, term, and document by one.

Parameters

topic The topic in question

term The term in question

doc The document in question

Reimplemented from **meta::topics::lda_gibbs**.

double
meta::topics::parallel_lda_gibbs::compute_sampling_weight (term_id term,
doc_id doc,
topic_id topic
) const override protected virtual

Computes a weight proportional to \(P(z_i = j | w, \boldsymbol{z})\).

Parameters

term The current word we are sampling for

doc The document in which the term resides

topic The topic \(j\) we want to compute the probability for

Returns

a weight proportional to the probability that the given term in the given document belongs to the given topic

Reimplemented from **meta::topics::lda_gibbs**.

Member Data Documentation

std::unordered_map<std::thread::id, std::vector<stats::multinomial<term_id> > >
meta::topics::parallel_lda_gibbs::phi_diffs_ protected

Stores the difference in topic_term counts on a per-thread basis for use in the reduction step.

Indexed as [thread_id][topic]

The documentation for this class was generated from the following files:

- /home/chase/projects/meta/include/topics/**parallel_lda_gibbs.h**
- /home/chase/projects/meta/src/topics/**parallel_lda_gibbs.cpp**

LIBLINEAR -- A Library for Large Linear Classification

Machine Learning Group at National Taiwan University
Contributors

NEW We recently released LibShortText, a library for short-text classification and analysis. It's built upon LIBLINEAR.

NEW Version 1.96 released on November 15, 2014. It conducts some minor fixes. Also note that we now provide 64-bit windows binary exe files.

An experimental version using 64-bit int is in LIBSVM tools. It in theory can handle up to 2^64 instances/features if memory is enough.

We are interested in **large sparse** regression data. Please let use know if you have some. Thank you.

NEW **A practical guide to LIBLINEAR** is now available in the end of LIBLINEAR paper.

Some extensions of LIBLINEAR are at LIBSVM Tools.

LIBLINEAR is the winner of ICML 2008 large-scale learning challenge (linear SVM track). It is also used for winning KDD Cup 2010.

Introduction

LIBLINEAR is a **linear** classifier for data with **millions** of instances and features. It supports

- L2-regularized classifiers
  L2-loss linear SVM, L1-loss linear SVM, and logistic regression (LR)
- L1-regularized classifiers (after version 1.4)
  L2-loss linear SVM and logistic regression (LR)
- L2-regularized support vector regression (after version 1.9)
  L2-loss linear SVR and L1-loss linear SVR.

Main features of **LIBLINEAR** include

- Same data format as **LIBSVM**, our general-purpose SVM solver, and also similar usage
- Multi-class classification: 1) one-vs-the rest, 2) Crammer & Singer
- Cross validation for model selection
- Probability estimates (logistic regression only)
- Weights for unbalanced data
- MATLAB/Octave, Java, Python, Ruby interfaces

Documentation

FAQ is here

When to use LIBLINEAR but not LIBSVM

There are some large data for which with/without nonlinear mappings gives similar performances.

Without using kernels, one can quickly train a much larger set via a linear classifier. **Document classification** is one such application. In the following example (20,242 instances and 47,236 features; available on LIBSVM data sets), the cross-validation time is significantly reduced by using LIBLINEAR:

```
% time libsvm-2.85/svm-train -c 4 -t 0 -e 0.1 -m 800 -v 5 rcv1_train.binary
Cross Validation Accuracy = 96.8136%
345.569s
% time liblinear-1.21/train -c 4 -e 0.1 -v 5 rcv1_train.binary
Cross Validation Accuracy = 97.0161%
2.944s
```

Warning: While LIBLINEAR's default solver is very fast for document classification, it may be **slow** in other situations. See Appendix C of our SVM guide about using other solvers in LIBLINEAR.

Warning: If you are a beginner and your data sets are not large, you should consider LIBSVM first.

Download LIBLINEAR

The current release (Version 1.96, November 2014) of **LIBLINEAR** can be obtained by downloading the zip file or tar.gz file.

The package includes the source code in C/C++. A README file with detailed explanation is provided. For **MS Windows** users, there is a subdirectory in the zip file containing binary executable files.

Please read the COPYRIGHT notice before using **LIBLINEAR**.

Documentation and Codes used for experiments in our papers

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and **C.-J. Lin**. LIBLINEAR: A library for large linear classification *Journal of Machine Learning Research* 9(2008), 1871-1874.

The appendices of this paper give all **implementation details** of LIBLINEAR.

In the end of this paper there is a **practical guide** to LIBLINEAR

See also some examples in Appendix C of the SVM guide.

Code used for experiments in our LIBLINEAR papers can be found here.

Interfaces to LIBLINEAR

Language	Description	Maintainers and Their Affiliation	Supported LIBLINEAR version	Link
MATLAB	A simple MATLAB interface	LIBLINEAR authors at National Taiwan University.	The latest	Included in LIBLINEAR package
Octave	A simple Octave interface	LIBLINEAR authors at National Taiwan	The latest	Included in LIBLINEAR

		University.		package
Java	Java version of LIBLINEAR	Benedikt Waldvogel	1.92	Java LIBLINEAR
Python	A python interface has been included in LIBLINEAR since version 1.6.	LIBLINEAR authors at National Taiwan University.	The latest	Included in LIBLINEAR package
Python	Python wrapper of LIBLINEAR	Uwe Schmitt	1.32	Zip/tar.gz file
Ruby	A Ruby interface via SWIG	Kei Tsuchiya (extended from the work of Tom Zeng)	1.93	liblinear-ruby-swig
Perl	A Perl interface	Koichi Satoh	1.93	perl module
Weka	Weka wrapper	Benedikt Waldvogel	1.5	Weka LIBLINEAR
R	R interface to LIBLINEAR	Thibault Helleputte	1.94	R LIBLINEAR
Common LISP	Common Lisp wrapper of LIBLINEAR	Gábor Melis	1.92	Common LISP wrapper
Scilab		Holger Nahrstaedt from the Technical University of Berlin	1.8	Scilab interface

---

Please send comments and suggestions to Chih-Jen Lin.