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"
```

The path to the models in the `tree` and `ngram-pos` analyzers is wherever you put the files downloaded from the current release (https://github.com/meta-toolkit/meta/releases).

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)