Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Projektarbeit Informatik

Development of a framework for text classification and participation at SemEval.

Autoren	Dominic Egger Pascal Julmy
Hauptbetreuung	Mark Cieliebak
Nebenbetreuung	Fatih Uzdilli
Industriepartner	
Externe Betreuung	
Datum	09.01.2015

Summary

In jüngster Zeit wurde - dank zunehmender Rechenleistung - Sentimentanalyse besser anwendbar für Alltagsprobleme und somit auch interessanter für Geschäftsanwendungen.
Diese Arbeit beschäftigt sich mit der Entwicklung und Evaluierung eines Frameworks für Sentimentanalyse mit einem Schwerpunkt auf der Analyse von Tweets und der Teilnahme am Wettbewerb *SemEval 2015*.
Es werden die Gründe für Designentscheide und Probleme im Entwicklungsprozess durchleuchtet, und ausserdem Beispiele, Ausblicke und Anregungen für weitere Entwicklung des Frameworks gegeben.
Zur Implementierung und zum Testen jeden Teiles des Frameworks wurden Arbeiten, Daten und Ergebnisse von früheren *SemEval*-Teilnehmern verwendet. Die Ergebnisse der Gewinner von *SemEval 2014* (NRC Canada) stellten ein Zwischenziel dar und wurden Mitte November erreicht. Anschliessend wurde auf denselben Daten eine grosse Zahl Experimente mit wechselnden Konfigurationen ausgeführt, um die Resultate weiter zu verbessern. Die Konfiguration mit dem besten Ergebnis wurde einen Monat später für die Teilnahme an *SemEval 2015* ausgewählt.
Der erreichte achte Rang aus 40 Teilnehmern (mit einer Präzision von 62,61%) deutet auf Potenzial des gewählten Ansatzes hin und sollte weiterverfolgt werden.

Abstract

With increase in computing power sentiment analysis has become more and more applicable to solve concrete problems in everyday life in recent years – thus getting more interesting for business application.
The aim of this thesis is the development and evaluation of a framework for sentiment analysis with a focus on analyzing tweets, and its participation at the *SemEval 2015* challenge.
The thesis expounds the reasons for design decisions and troubles during development and provides usage examples, outlooks and useful information for further development.
Papers, data and scores from previous *SemEval* participants were used to implement and test each part of the framework during development. The scores from the winning team of *SemEval 2014* (NRC Canada) were taken as the intermediate goal and have been reached in mid-November. After that a multitude of experiments were conducted with varying framework configurations on the same data in order to further increase the score. The best scoring configuration was then used to participate in *SemEval 2015* one month later.

The 8th rank was reached (out of 40 participating teams) with a precision of 62.61%. On the basis of these final results, it can be concluded that the chosen design bears significant potential and should be further developed.

Index

Introduction

Tasks and expectations
By definition the assigned task consisted of a framework development for sentiment analysis of text. Additionally the framework was to be used in the participation at the *SemEval 2015* competition, specifically in the task surrounding sentiment analysis of tweets. The original document concerning the assignment[1] as well as a shortened task description[2] describing the *SemEval 2015* task can be found in the addendum.

The data used to train the machine learning engine and to test its capabilities were provided by *SemEval* from previous years.

What is sentiment analysis?
Sentiment analysis (also known as sentiment detection or opinion mining) tries to get a read on the attitude a given text expresses towards a topic. This field of research proves vital especially for marketing and other topics interested in emotional response towards a certain topic.

What is the SemEval challenge?
The SemEval committee describes it as follows on Wikipedia:

> "*SemEval* (*Sem*antic *Eval*uation) is an ongoing series of evaluations of computational semantic analysis systems […]. The evaluations are intended to explore the nature of meaning in language. While meaning is intuitive to humans, transferring those intuitions to computational analysis has proved elusive."[3]

Designed as a competition between teams, the *SemEval* challenge boosts creativity and allows for new concepts to be tried out. After finishing a challenge the results are published and the winning teams are obliged to hand in a paper explaining their design, in order for teams participating in feature challenges to be able to profit from previous findings.

Intended audience
This paper targets software engineers with an interest in machine learning and sentiment analysis. In the beginning any knowledge needed to understand machine learning and sentiment analysis discussed in this paper will be covered. The paper also addresses people generally interested in machine learning, however the software engineering part (section "Framework description") is not intended for this audience.

Literature research
Most of the research focused on papers of other contestants of *SemEval* from previous years. They are listed in the literature index.

Important note
In the following document there is an important difference between "framework feature" and "Learning feature" (or simply referred to as "feature"). The latter refers to an aspect of information extracted from a text whereas the first one is a functionality the framework provides.

Further information to each cursively written expression can be found in the glossary.

Theoretical basics

This chapter covers any required additional knowledge to understand this paper's content. It is necessary in order to understand decisions made in further chapters. Without a target audience in mind only machine learning specific topics will be addressed in this chapter.

Machine learning

In this part a very simplified view on machine learning will be provided. This is not to be considered an in-depth explanation but merely a helpful overview.

The data used for machine learning consists of two parts: firstly the measurements, called features, and secondly a value representing a prediction based on its features. Original price, size, type and age can be taken as examples for features of a car when trying to predict its sales value. These data entries are called training data and in their entirety form a training set. To predict the missing value of a new entry, two things are of importance: for one, a predictive function which yields a value for a given set of measurements, and a cost function which describes how much difference this predictive function accumulates on every entry of the training set.
By minimizing the difference provided by the cost function, the predictive function can be optimized; Therefore making predictions more accurate.

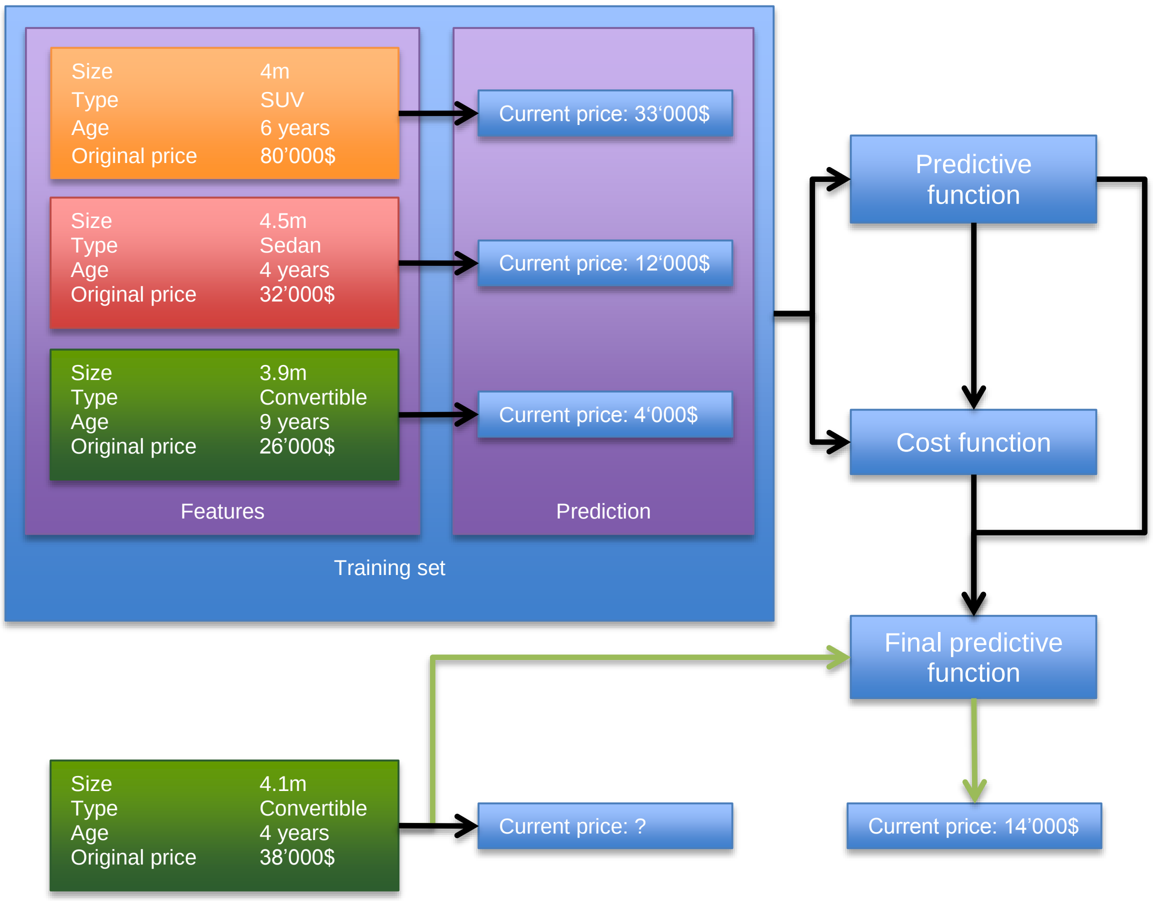


Figure 1 Basic machine learning process

In the case of *SemEval* there are a few differences.

- Features are measurements of text, specifically on tweets.
- The missing values are discrete rather than real since there are only 3 possible values (neutral, negative or positive). This means that the problem at hand is in fact a classification problem.

F1-score

The F1-score is a statistical measure for the fitness of a specific machine learning configuration against a known dataset. It is calculated for each class of tweets (positive, negative and neutral) independently and the mean value of positive and negative scores is used for the final score.

It is calculated as follows:

$$F_\beta(\omega) = (1+\beta^2) * \frac{precision_\omega * recall_\omega}{(\beta^2 * precision_\omega) + recall_\omega}$$

$$\omega \in \Omega \mid \Omega := \{positive, negative, neutral\}$$

The “1” in F1 refers to the parameter Beta which is responsible for the weighing between recall and precision. For any Beta greater than 1 the score becomes recall oriented and for any beta smaller than 1 it becomes precision oriented. For example $F_0(\omega) = \frac{precision_\omega * recall_\omega}{recall_\omega} = precision_\omega$

As mentioned before there are three possible outcomes (neutral, negative and positive) for the *SemEval* task. For each of these classes a table as shown below can be constructed. It is essential to point out that the positive/negative in this table does not relate to the classification of a tweet but to the fact of whether a given tweet was positively or negatively classified in relation to a specific class.

In this example a tweet is analyzed in respect to the positive text class.

Table 1 F1-Score explanation

	Tweet is positive	**Tweet is not positive**
Classifier predicts positive	True Positive (tp) This is a success	False Positive (fp) This is an error
Classifier predicts not positive	False Negative (fn) This is an error	True Negative (tn) This is a success

Precision and recall are defined as follows:

$$precision_\omega = \frac{tp_\omega}{tp_\omega + fp_\omega}$$

The precision describes the ratio of actually positive tweets in all tweets that were classified as positive.

$$recall_\omega = \frac{tp_\omega}{tp_\omega + fn_\omega}$$

The recall describes the ratio of tweets which have been classified as positive to all positive tweets.

Framework description

In this chapter the architecture and software engineering of the developed framework prototype are discussed. It does not deal with text classification directly and is aimed at potential users of the framework and those interested in software engineering.

*More information on our sentiment analysis part and our results in SemEval can be found in the chapters "Implementing SemEval in the framework" and "*Measurements and Results*".*

Why a framework?

The decision to develop a framework instead of a specific solution for the *SemEval* task was made because of the possibility of reusing it entirely for other *SemEval* tasks and other text classification tasks. One of the core ideas was to ensure smooth concurrent integration between multiple developers especially in respect to feature development.

Framework design decisions

The authors decided upon splitting the framework into several parts loosely interconnected by a pipeline for ease of usage. The work was conducted under severe design constraints to ensure both the framework's performance and extensibility in terms of modularity.

In areas where future developers might want to intercept the framework's default behavior only interfaces were used instead of classes, abstract or otherwise. This ensures that a programmer has full inheritance potential at his or her disposal.

The features are independent from any typing so that another machine learning engine could be fitted in with minimal effort. This is achieved through the use of generics and their attentive integration throughout the whole framework.

To make sure that a large number of programmers can work concurrently on a task the feature interface is very rudimentary. This allows programmers without any deep knowledge of the framework to contribute feature implementations to a project.

Planned software features

These are the feature highlights of our framework. Some of them have not been implemented due to time and/or resource constraints and are marked accordingly. For the features not implemented there will be descriptions in the relevant sections, detailing the implementation approach.
Some of these features interact with parts of the framework not yet explained, the sections providing this information can be found on the following pages.

Processing of large String quantities to extracted feature vectors

The Framework is designed to use a series of iterator transformations, starting with accepting an iterator of strings into the preprocessor. This yields an iterator of preprocessed *Document*-objects which in turn can be fed into the feature extraction. The memory usage is thereby kept at a reasonably low level and it allows for large test and training sets to be used.

Feature extraction from an abstract enriched document model

Instead of having the features extract their values directly from a string, they get a preprocessed document model that is enriched with all the required metadata. This keeps the implementation of features leaner. Considering that the feature implementation is the point where the most developers will be working concurrently, this will raise the overall robustness of the system. The document model itself provides direct access to some of the most frequently used queries on the underlying texts.

Type independent feature implementation [partially implemented]
Different machine learning engines require different formats of vectors. Because of this the authors strived to keep the framework - especially the features - type-independent of their actual implementation. And even existing features could be easily wrapped to result in other vector types.

Easy adding of new features
To keep the usage of the framework efficient we aimed to keep the work required to implement a new feature to a minimum. This is done by providing an interface with only two methods, one of which has a default implementation. When adding a new feature there should only be two concerns: How to extract the feature values from the document model and what configurations this feature provides.

Tracking of values over all handled feature vectors [partially implemented]
Especially for scaling, values like mean, standard deviation, min and max of a specific feature are important and the framework provides a way to track these values. Multiple functions can be provided to be executed on a per-feature-, per-vector- or a per-vector-collection-basis.

Strong use of closures for fast prototyping
Features can be implemented directly as closures without having to create new classes. However this is only recommended for prototyping as the code gets bloated quickly this way. Many other features such as scaling and key globalization use closures as well.

Dependency specification [not yet implemented]
Often a certain feature will rely on certain metadata extractors or tokenizers being used. The Framework should be able to detect whether a configuration satisfies all constraints given by the features. Currently this is not the case and a feature falls back on a default configuration in case of error.

Configuration files & External tooling [not implemented]
Testing for the best possible configuration for a given problem and testing new features in most cases requires a large number of runs. Configuring the exact settings of all modules with external files and tools would make this more easily reproducible and may help to reduce code clutter in the project. At the current stage of the project individual main classes are required to run a set of configurations.

Serialization and reuse of data [not implemented]
The document model should be serializable and storable. Often only the feature configuration changes while the preprocessor settings stay the same. Being able to reuse already computed data would cut down on runtime. With certain metadata in place (e.g. spelling correction), computing the document model can take a great deal of time.

Statistics & metrics [not implemented]
For coherent testing the precise configuration of the framework, the resulting performance measure and various environmental data like run time, CPU load and Memory usage have to be recorded. This not only makes tweaking the configuration easier but also ensures reproducibility of experimental results. In this paper said results are retrieved manually, with this software feature the process would be automated.

Intelligent feature selection [not implemented]
The basic idea behind this is that the Framework is able to change its own configuration to attain better results. However during framework development it was quickly determined that such a feature was beyond the scope of this paper. Instead plans were formulated to specify interfaces enabling the framework to do multiple runs, changing its own configuration according to a user specified algorithm.

In itself the whole Framework is split into several overarching modules:

Document model

As depicted the document model consists of a document, which represents an instance of text that has to be classified. Each document consists of multiple paragraphs which in turn can contain any number of sentences which themselves consist of tokens.
The document model was designed in such a way as for features to be able to respect sentence and/or paragraph boundaries if necessary.

A token can be tagged with a multitude of metadata information that later can be retrieved to form entries on a feature vector. Examples of metadata information can be found under “Used Metadata”.

Several convenience methods have been added that allow easy, simple querying of tokens, sentences, paragraphs, and their respective metadata. Most of the time a feature requires all tokens, but in certain cases only the tokens generated by a specific tokenizer are of interest. Because of this the document can provide both the results of a specific tokenizer and all other results except a certain set of tokenizers.

On serialization and reuse of data

One software feature not yet implemented would provide the possibility to persist the document model for a given set of document strings and a specified configuration. This would save time when repeatedly testing different feature selections on the same set of documents that were preprocessed in the same way.
Due to time constraints the implementation is missing, but the necessary architecture has already been put into place.

When implementing this feature it is important to keep track of possible changes in all members of the *PreProcessorConfig*, meaning checking the timestamp of the class-file against the timestamp attribute on the document object. If there are any timestamps newer than the recorded time it would indicate a change in the code of the used mutations, metadata extractors or the document splitters and would need to be computed again.

On type independence

This framework is intended for text classification, however in its current state a discrete classification of texts is enforced. With further generalization employing generics, it would be possible to remove that restriction and handle arbitrary prediction values. Other parts of the framework are already implemented in such a way. For instance the feature vectors are fully type-independent.

On advanced querying

Especially when handling larger documents, the ability to employ advanced queries for paragraph, sentence or token selection would make the framework easier to use and the feature implementations leaner. This could be done by employing the filter technique provided by Java 8 Streams, meaning that both~~,~~ the *BaseModel*-class and the *Document*-class would have to implement the *Stream<T>*-interface.

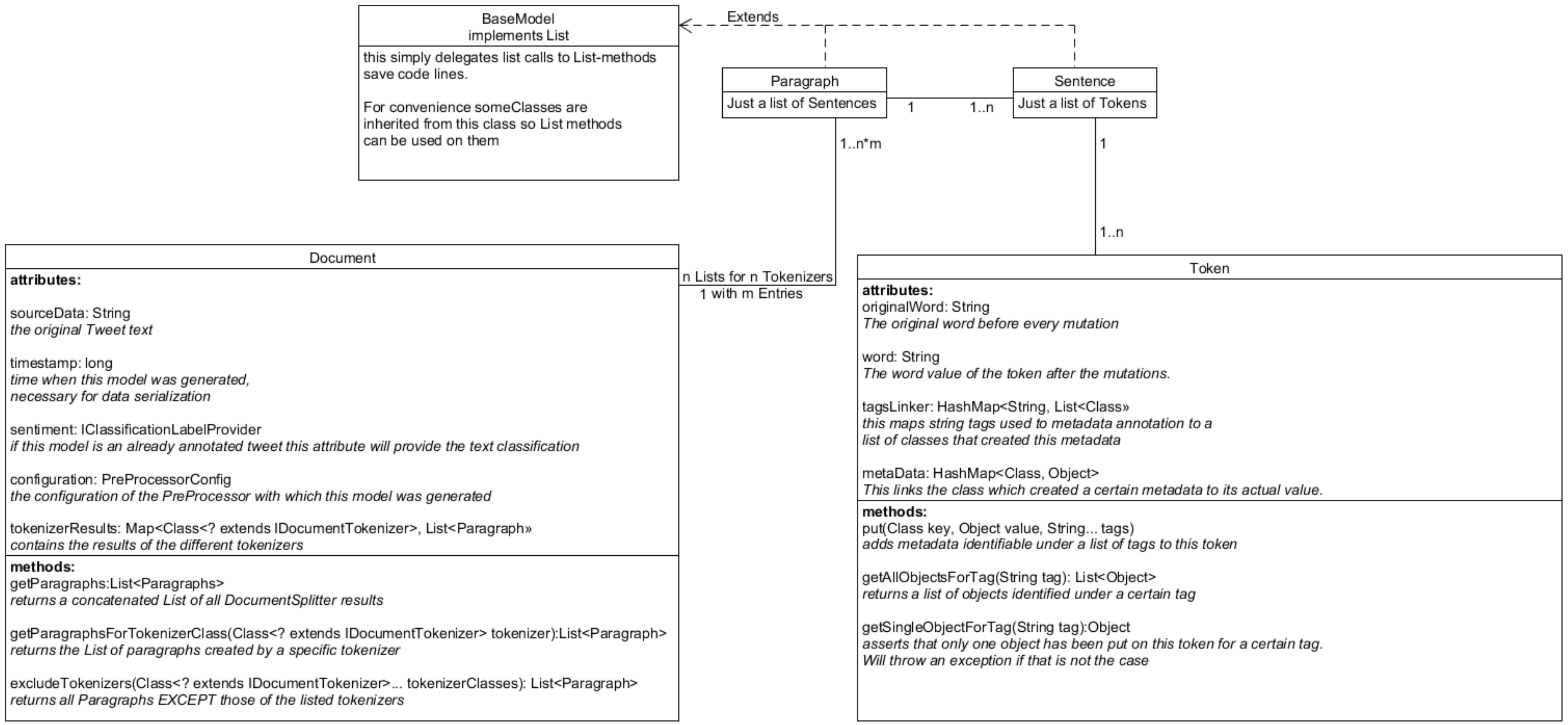

BaseModel
implements List
this simply delegates list calls to List-methods
save code lines.
For convenience someClasses are
inherited from this class so List methods
can be used on them
Extends
Paragraph
Just a list of Sentences
Sentence
Just a list of Tokens
1
1..n
1..n*m
1
1..n
n Lists for n Tokenizers
1 with m Entries
Document
attributes:
sourceData: String
the original Tweet text
timestamp: long
time when this model was generated,
necessary for data serialization
sentiment: IClassificationLabelProvider
if this model is an already annotated tweet this attribute will provide the text classification
configuration: PreProcessorConfig
the configuration of the PreProcessor with which this model was generated
tokenizerResults: Map<Class<? extends IDocumentTokenizer>, List<Paragraph»
contains the results of the different tokenizers
methods:
getParagraphs:List<Paragraphs>
returns a concatenated List of all DocumentSplitter results
getParagraphsForTokenizerClass(Class<? extends IDocumentTokenizer> tokenizer):List<Paragraph>
returns the List of paragraphs created by a specific tokenizer
excludeTokenizers(Class<? extends IDocumentTokenizer>... tokenizerClasses): List<Paragraph>
returns all Paragraphs EXCEPT those of the listed tokenizers
Token
attributes:
originalWord: String
The original word before every mutation
word: String
The word value of the token after the mutations.
tagsLinker: HashMap<String, List<Class»
this maps string tags used to metadata annotation to a
list of classes that created this metadata
metaData: HashMap<Class, Object>
This links the class which created a certain metadata to its actual value.
methods:
put(Class key, Object value, String... tags)
adds metadata identifiable under a list of tags to this token
getAllObjectsForTag(String tag): List<Object>
returns a list of objects identified under a certain tag
getSingleObjectForTag(String tag):Object
asserts that only one object has been put on this token for a certain tag.
Will throw an exception if that is not the case

Figure 2 Document Model

Preprocessor

The Preprocessor consists of three separate stages: Splitting a string, mutating it and extracting metadata from it. These three steps are described in detail in the following sections. In simplified terms this means that the Preprocessor converts a raw string into the required document model. It is important to notice that while there can be any number of mutations and metadata extractors the use of only one document splitter is possible.

Document Splitter

The responsibility of the document splitter is to accept a raw spring and convert it into the internal model of a text: the document model. All features will later use this model to construct the vector supplied to the machine learning engine. On a primitive level a document splitter might split the String by punctuation and then each of the resulting substrings by whitespace to fill the model with the required data of tokens (words), sentences and paragraphs. Several different tokenizers can be used at the same time. That is due to the fact that certain metadata extractors and other components need the text tokenized in a very specific manner to be able to work properly.
The framework currently uses the TweetNLP[4] framework, as well as a light-weight tokenizer written by ourselves to process the document strings.

Mutations

Mutations are simple changes are the first step taken after the document has been split into its parts. They are not meant to represent any significant data in terms of machine learning but merely to reduce the scarcity of the resulting feature vectors. An example in the framework would be a mutation which replaces all URLs in a text with a generic wildcard as the URLs destinations were not of interest.

Metadata Extractors

These components are meant to enrich the document with metadata. Very common examples are the tagging of tokens with POS-Tags or their negation-status. There is an implicit dependency between features and extractors, as features have to be aware of the format a metadata extractor uses to put its data on the token. Metadata the framework currently extracts are POS-Tags, lemmas and negation scope. A more concise description of each kind of metadata can be found in the section “Used Metadata”.

Interaction with the Pipeline

The Pipeline contains the Preprocessor and provides several delegates for the preprocessor and its configuration (*PreProcessorConfig*). However nothing the pipeline does has any functional impact at the preprocessing stage.

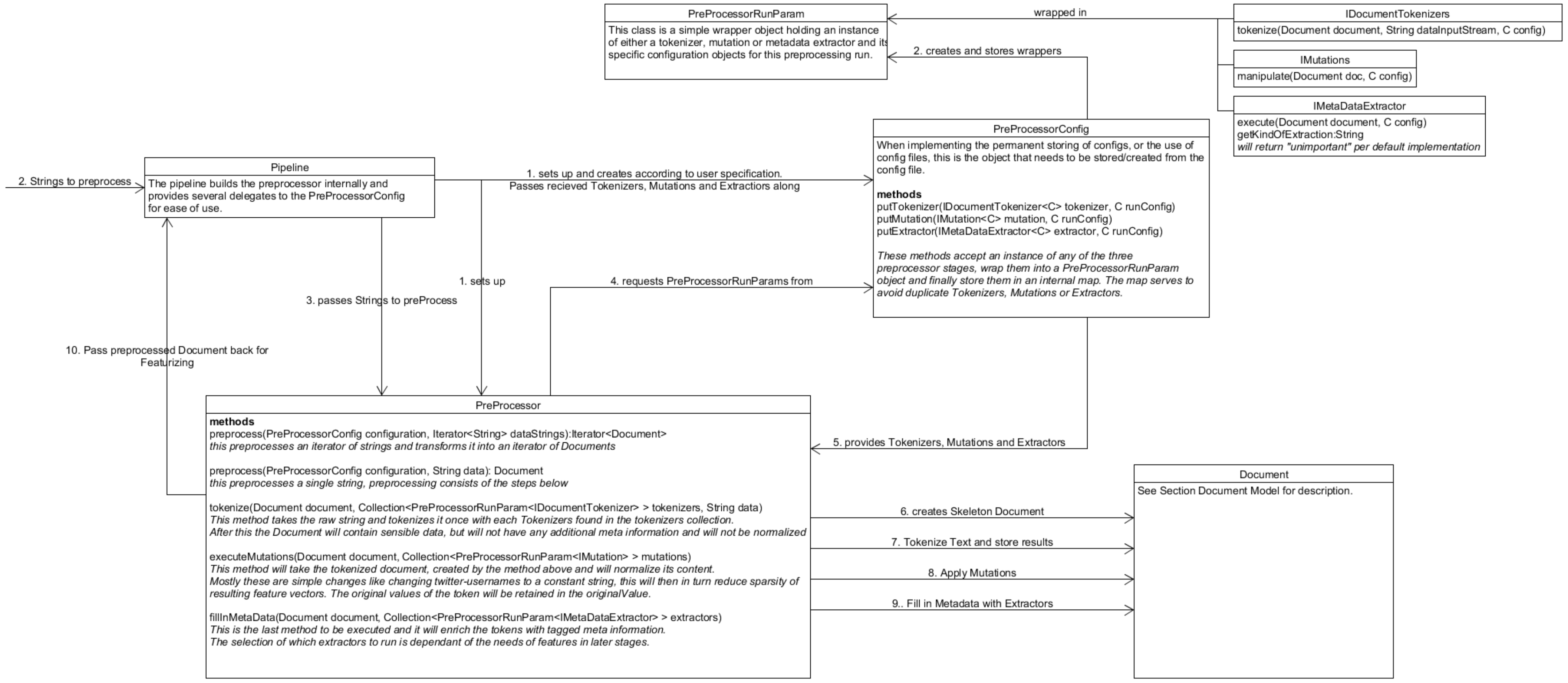

PreProcessorRunParam
This class is a simple wrapper object holding an instance of either a tokenizer, mutation or metadata extractor and its specific configuration objects for this preprocessing run.
wrapped in
IDocumentTokenizers
tokenize(Document document, String dataInputStream, C config)
2. creates and stores wrappers
IMutations
manipulate(Document doc, C config)
IMetaDataExtractor
execute(Document document, C config)
getKindOfExtraction:String
will return "unimportant" per default implementation
PreProcessorConfig
When implementing the permanent storing of configs, or the use of config files, this is the object that needs to be stored/created from the config file.
methods
putTokenizer(IDocumentTokenizer<C> tokenizer, C runConfig)
putMutation(IMutation<C> mutation, C runConfig)
putExtractor(IMetaDataExtractor<C> extractor, C runConfig)
These methods accept an instance of any of the three preprocessor stages, wrap them into a PreProcessorRunParam object and finally store them in an internal map. The map serves to avoid duplicate Tokenizers, Mutations or Extractors.
Pipeline
2. Strings to preprocess
The pipeline builds the preprocessor internally and provides several delegates to the PreProcessorConfig for ease of use.
1. sets up and creates according to user specification.
Passes recieved Tokenizers, Mutations and Extractions along
1. sets up
3. passes Strings to preProcess
4. requests PreProcessorRunParams from
10. Pass preprocessed Document back for Featurizing
5. provides Tokenizers, Mutations and Extractors
PreProcessor
methods
preprocess(PreProcessorConfig configuration, Iterator<String> dataStrings):Iterator<Document>
this preprocesses an iterator of strings and transforms it into an iterator of Documents
preprocess(PreProcessorConfig configuration, String data): Document
this preprocesses a single string, preprocessing consists of the steps below
tokenize(Document document, Collection<PreProcessorRunParam<IDocumentTokenizer> > tokenizers, String data)
This method takes the raw string and tokenizes it once with each Tokenizers found in the tokenizers collection.
After this the Document will contain sensible data, but will not have any additional meta information and will not be normalized
executeMutations(Document document, Collection<PreProcessorRunParam<IMutation> > mutations)
This method will take the tokenized document, created by the method above and will normalize its content.
Mostly these are simple changes like changing twitter-usernames to a constant string, this will then in turn reduce sparsity of resulting feature vectors. The original values of the token will be retained in the originalValue.
fillInMetaData(Document document, Collection<PreProcessorRunParam<IMetaDataExtractor> > extractors)
This is the last method to be executed and it will enrich the tokens with tagged meta information.
The selection of which extractors to run is dependant of the needs of features in later stages.
Document
See Section Document Model for description.
6. creates Skeleton Document
7. Tokenize Text and store results
8. Apply Mutations
9.. Fill in Metadata with Extractors

Figure 3 PreProcessor

Feature extraction

The second part of the framework deals with extracting feature vectors out of the preprocessed document model. During the creation of the pipeline the programmer has to define what format the feature-vectors (Key/Value) will have. In our case this was <*String, Double*>*.* The core function of the feature extraction is to evaluate a set of preprocessed documents and extract a feature vector with the specified format for each document. Later these feature-vectors will be passed to a machine learning engine.

For this purpose an interface was defined for the feature itself. The following method signatures are required for a feature:

```
default public List<C> getAllowedConfigObjects(){
            return Arrays.asList();
        }

public Map<K, V> extract(Document doc, List<C> configs)
```

The return value of the extract method represents the partial vector generated by each feature. The keys in these vectors are unique for each feature-implementation-class. They are made globally unique by the feature extractor by means of a passed function.

The *configs*-list represents a set of different modes the feature can run with.
A feature will run independently once for every mode passed in this list. For instance the n-gram feature can create n-grams of varying length so the configurations might be a list of integers telling the feature the length of the n-grams it has to generate.

The resulting vectors are *sparse*. In case the utilized machine learning engine cannot handle sparse vectors a converter has to be written. This will often be the case as most of these engines bring their own format for vectors that might not use the *Map*-interface or even Java at all.

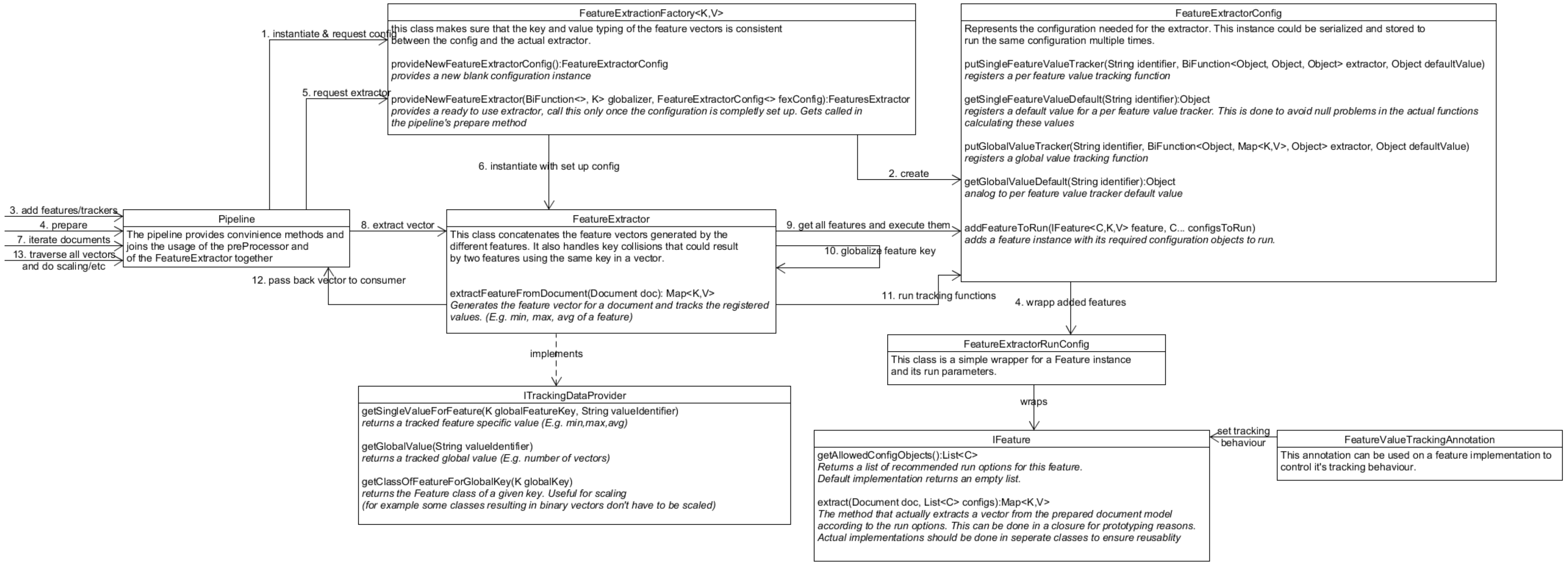
On Value tracking

Besides the pure vector generation, the feature extractor has several other responsibilities. When setting up the pipeline the developer can specify numerous closures that will be evaluated on either each vector or on each feature. By using these closures it is, for instance, possible to track the maximum value of a feature or the number of generated vectors. It is often used to calculate values required for scaling later in the process.

This particular framework feature is also the reason that the pipeline requires a prepare-method. The feature extractor is instantiated with several internally generated closures upon invocation of the prepare-method. Due to this any changes to the configuration of the pipeline after calling *prepare()* could break its function.

Even though this makes the use of the feature extractor and the pipeline more rigid, it also provides a measurable boost in runtime thanks to the reduction in *if*-clauses.

Features can also be *annotated* to omit certain tracking calculations or even disable them altogether to further increase the runtime. Disabling the scaling could be useful on boolean yes/no features for instance.


1. instantiate & request config
5. request extractor
6. instantiate with set up config
2. create
3. add features/trackers
4. prepare
7. iterate documents
13. traverse all vectors and do scaling/etc
8. extract vector
9. get all features and execute them
10. globalize feature key
12. pass back vector to consumer
11. run tracking functions
4. wrapp added features
implements
wraps
set tracking behaviour

FeatureExtractionFactory<K,V>
this class makes sure that the key and value typing of the feature vectors is consistent between the config and the actual extractor.

provideNewFeatureExtractorConfig():FeatureExtractorConfig
provides a new blank configuration instance

provideNewFeatureExtractor(BiFunction<>, K> globalizer, FeatureExtractorConfig<> fexConfig):FeaturesExtractor
provides a ready to use extractor, call this only once the configuration is completly set up. Gets called in the pipeline's prepare method

FeatureExtractorConfig
Represents the configuration needed for the extractor. This instance could be serialized and stored to run the same configuration multiple times.

putSingleFeatureValueTracker(String identifier, BiFunction<Object, Object, Object> extractor, Object defaultValue)
registers a per feature value tracking function

getSingleFeatureValueDefault(String identifier):Object
registers a default value for a per feature value tracker. This is done to avoid null problems in the actual functions calculating these values

putGlobalValueTracker(String identifier, BiFunction<Object, Map<K,V>, Object> extractor, Object defaultValue)
registers a global value tracking function

getGlobalValueDefault(String identifier):Object
analog to per feature value tracker default value

addFeatureToRun(IFeature<C,K,V> feature, C... configsToRun)
adds a feature instance with its required configuration objects to run.

Pipeline
The pipeline provides convinience methods and joins the usage of the preProcessor and of the FeatureExtractor together

FeatureExtractor
This class concatenates the feature vectors generated by the different features. It also handles key collisions that could result by two features using the same key in a vector.

extractFeatureFromDocument(Document doc): Map<K,V>
Generates the feature vector for a document and tracks the registered values. (E.g. min, max, avg of a feature)

FeatureExtractorRunConfig
This class is a simple wrapper for a Feature instance and its run parameters.

ITrackingDataProvider
getSingleValueForFeature(K globalFeatureKey, String valueIdentifier)
returns a tracked feature specific value (E.g. min,max,avg)

getGlobalValue(String valueIdentifier)
returns a tracked global value (E.g. number of vectors)

getClassOfFeatureForGlobalKey(K globalKey)
returns the Feature class of a given key. Useful for scaling (for example some classes resulting in binary vectors don't have to be scaled)

IFeature
getAllowedConfigObjects():List<C>
Returns a list of recommended run options for this feature. Default implementation returns an empty list.

extract(Document doc, List<C> configs):Map<K,V>
The method that actually extracts a vector from the prepared document model according to the run options. This can be done in a closure for prototyping reasons. Actual implementations should be done in seperate classes to ensure reusablity

FeatureValueTrackingAnnotation
This annotation can be used on a feature implementation to control it's tracking behaviour.

Figure 4 Feature extraction

The pipeline

The pipeline is a loose interconnection meant to reduce the amount of code that has to be written to get from a file containing the raw data to the finished feature vectors.

Several parameters have to be set upon creation of the pipeline:

- The class of keys used in the feature-vectors.
- The class of values in the feature-vectors.
- An entity that handles the mapping of integer text-classification labels to usable text-classes.
- A closure that maps the local keys generated by a single feature to a global key within the vector to avoid collisions resulting from vectors using the same local keys.
- An object responsible for feature-scaling (this is optional)

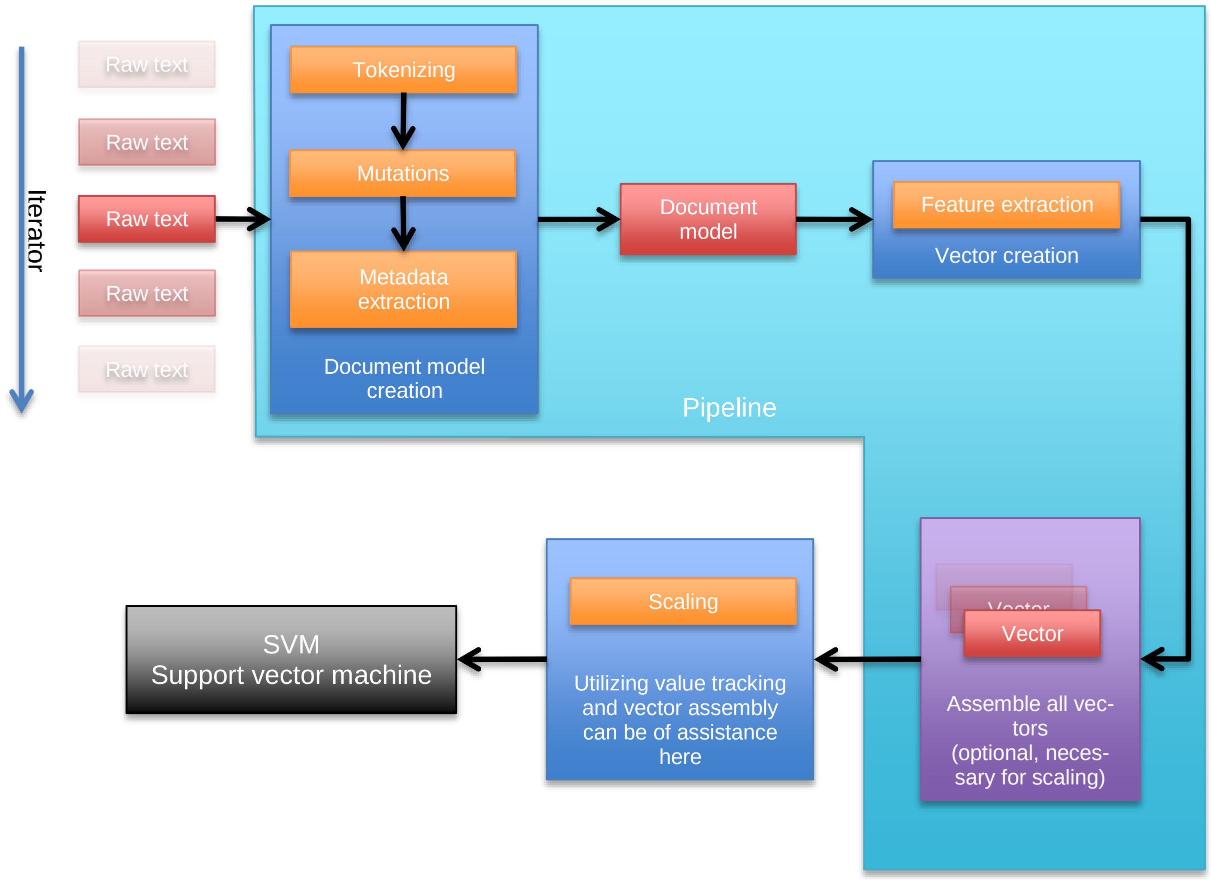


Figure 5 Graphical representation of pipeline processing data

After that, to use the pipeline all the tokenizers, metadata-extractors, mutations, and features have to be specified. The actual feature vectors can then be extracted and passed to a consumer which in turn prepares them to be passed to the actual machine learning engine. Before the classifier is started the pipeline can be used to scale vectors with the value tracking feature as well as the traverse method on the pipeline itself.

Here is some example code explaining the usage of the pipeline.

1. Instantiating the pipeline

```
BiFunction<String, IFeature<String, Double, ?>, String> keyGlobalizer =
       (key, feature) -> {
            return feature.getClass().getSimpleName() + "_" + key;
        };

pipeline = new Pipeline<>(keyGlobalizer, true);
```

The *BiFunction* describes how a locally generated key within a feature has to be globalized in order to avoid naming collisions from occurring between features.

The *boolean*-flag in the pipeline constructor tells the pipeline whether to keep track of vector references or not. Setting this to *true* will lead to much larger memory consumption but will allow iteration over all generated vectors in the end, allowing for instance scaling.

2. Adding a Tokenizer, Mutation & Extractor

```
pipeline.addTokenizer(new TweetNLPTokenizer());
pipeline.addMutation(new SimpleURLNormalizer());
pipeline.addExtractor(new TweetNLPPOSTagger(),TweetNLPPOSTagger.DEFAULT_MODEL);
```

These methods look the same as far as their signature is concerned; they need an instance of an *IMutation*, *IDocumentTokenizer* or an *IMetaDataExtractor* and their respective set of configurations.

3. Adding a feature to the pipeline

```
pipeline.addFeatureToRun(
       new NGramFeature(),
       new NGramConfigWrapper(1, Arrays.asList()),
       new NGramConfigWrapper(2, Arrays.asList()),
       new NGramConfigWrapper(3, Arrays.asList()),
       new NGramConfigWrapper(4, Arrays.asList()),
       new NGramConfigWrapper(1, Arrays.asList(),
                              NGramFeature.LEMMA_EXTRACTION_CLOSURE),
       new NGramConfigWrapper(2, Arrays.asList(),
                              NGramFeature.LEMMA_EXTRACTION_CLOSURE),
       new NGramConfigWrapper(3, Arrays.asList(),
                              NGramFeature.LEMMA_EXTRACTION_CLOSURE),
       new NGramConfigWrapper(4, Arrays.asList(),
                              NGramFeature.LEMMA_EXTRACTION_CLOSURE)
);

pipeline.addFullFeatureToRun(new NonContiguousNGram());
pipeline.addFeatureToRun(new NrOfAllCapsToken());
```

The first example shows how a feature can be added with a custom list of run settings. The type of the configuration object is defined by the feature itself. This particular setup will lead to this feature being run eight times: once for every configuration.

The second example adds a feature that will retrieve the configurations automatically by calling the feature's *getAllowedConfigObjects()*-method.

In the third example a feature is added without any configuration object. Some features do not require one as there is nothing to be configured.

4. Adding value trackers

```
pipeline.addSingleFeatureValueTracker("min",
       (oldVal, newVal) -> {
              return Double.min((Double)oldVal, (Double)newVal);
       }, 0d
);

 pipeline.addGlobalValueTracker("nrOfVectors",
       (oldVal, vector)->{
              return ((Integer)oldVal)+1;
       }, 0
);
```

The top example will keep track of the minimum value of each feature. This value can later be used for scaling or statistics purposes.

The second block in the example simply counts the number of vectors. It is however a good example of a global value tracker. As we do not generate lists of vectors, this information cannot be accessed just by calling *size()* on it.

In these closures all the passed values are of type *Object* and the developer must be aware or even keep track of the used types.

5. Extracting the feature vectors

```
pipeline.prepare();
```

This is the first step that has to be done before the actual extraction can start. This simple statement will instantiate the extractor within the pipeline and finalize its settings.

```
fillDataSet(DataSet<String> trainSet, Importer importer, String resourceName)
{

pipeline.extractAnnotatedFeatures(importer, resourceName, (features, label) ->
{
    trainSet.addEntry(new TrainingInstance<>(features, label.getLabel()));
});

}
```

After preparing the pipeline an importer processes a file with training data. The training set is a data collection outside the responsibility of the framework and will later be passed to *Liblinear*.

Internally the pipeline provides a string from the file provided by the importer to the preprocessor which will then pass a document model to the feature extractor. This yields a finished feature vector and a sentiment label extracted from the training document.

In the case it was data that needed to be classified after training rather than already labeled training data, the only difference would be that there was no label to be passed.

6. Using tracked values and traversing vectors

```
pipeline.traverse((tracker, vector)-> {
        vector.forEach((key, value)-> {

Integer nrOfUses = (Integer)tracker.getSingleValueForFeature(key, "nrOfFeatureUses");
Integer nrOfVectors = (Integer)tracker.getGlobalValue("nrOfVectors");

if( nrOfUses != null && nrOfVectors != null &&
    tracker.getClassOfFeatureForGlobalKey(key).equals(NGramFeature.class))
{
        Double frequency = nrOfUses.doubleValue()/nrOfVectors.doubleValue();
        Double scalingFactor = -Math.log(frequency)+1;
        maxNGramScalingFactor = Double.max(maxNGramScalingFactor, scalingFactor);
        vector.put(key, scalingFactor*value);
}

        });
});
```

In step 1 the pipeline was instantiated with a *true*-flag. This allows for traversing the vectors without having to pass them explicitly.

The code above shows an attempt at tf-idf (see chapter “TF-IDF Scaling”) scaling we tried. When traversing the vectors, the closure receives a vector as well as a tracker containing all the tracked values set up in step 4.

This method has its limitations as it needs several traversals to calculate certain values, such as variance.

Machine learning engine

In theory any library that implements machine learning could be used. In most cases however some wrapping is required. This holds true especially for transforming the framework's feature vector representation (which uses Java maps) to whatever representation of vectors the library requires.

The framework contains a wrapper for the Liblinear[5] machine learning library. The wrapper has several vital tasks:

- It provides different parallelized, self-optimizing classifiers intended to reach a higher F1-score. This can be achieved by providing another dataset between the training- and the test-dataset to cross reference against, or by n-folds.
- It transforms the resulting sparse vectors to an array representation which Liblinear can work with. This includes building the vector space.
- Training and evaluating datasets.

When starting a task using the framework it is recommended to choose the typing of the feature vectors according to the requirements of the utilized machine learning library.

Liblinear[5]

The source code of Liblinear is included in the project. Due to restrictions in the design of this library some of the source code had to be changed to return deterministic results when multiple instances are run in parallel.
Several of the methods in Liblinear's source code make use of a random instance which leads to non-deterministic results if several classifiers are active at once. We lifted that random instance into the classifier so instead of one shared random object each classifier has its own.

The C-parameter

The C-parameter can be seen as regularization parameter for a SVM. To search the best C for a given configuration a minimum, a maximum and a C-step size are defined. Each possible C results in a classifier which then is evaluated and the setting with the best result is chosen.

Optimizing

There are two ways Liblinear can be optimized:

- *With N-Folds*
  With the N-Fold method, the training data is separated into N equally sized partitions. One of these partitions is then set aside as validation data and a classifier is trained on each of the remaining N-1 partitions. The resulting classifiers then classify the data within the validation set and the best achieved F1-score is recorded. This score can then be compared to the F1-Score achieved on the test data.

- *With a static dataset*
  This works very similarly to the N-Folds but instead of sub partitioning the training data, a third dataset (besides the training and the test data) is provided to validate the classifier against.

Approaches for not yet implemented software features

As noted in the prior section “Planned software features” not all features have been implemented yet. This part of the paper will cover the implementation approaches of all software features left unexplained until now.

Implementation of dependency specification

This could be done with annotations on the feature implementations. Those annotations would need runtime retention and could be checked in the *prepare()*-method of the pipeline. If those features happen to require a component that has not been added to the pipeline it could try to load it on the fly or throw an exception.

These annotations would also allow external tooling to display component dependency of features.

Configuration files & external tooling

To make this easier both the preprocessor and the feature extractor have their own configuration objects. To implement configuration files an importer has to be written which reads a file with a given structure and returns the underlying configuration objects.

As for the external tooling, several options present themselves. Here is a list of ideas that came up during development:

- Configuration manager
  A configuration manager would make it easier to set up configurations for the framework, save them and rerun them.
- Score optimization tool
  This tool would provide the possibility to compare several different configurations that ran on the same dataset in order to compare the results of experiments.
- Feature repository
  A growing number of features may become impractical to handle as they have all to be imported into the project. To solve this problem a repository could be build. This could be done with existing solutions like maven or by class-loading by name.

Serialization and reuse of data

Especially when running experiments repeatedly a lot of calculation time is currently lost with the evaluation of redundant data. Often in experiments only the selection of features is changed while all preprocessor settings remain the same. Several things would have to be implemented to mitigate this problem. For one the document model would have to be serializable and therefore a loader would need to be written.

The next step would be to serialize the feature vector in a way that a developer can select subsets of features of each vector. This would make repeat feature extraction obsolete. However this has to be done carefully, for if the vectors would be stored after scaling and only subsets of features would be used, problems could ensue due to the previous scaling now being incorrect.

Statistics & metrics

Currently all the statistics are done by hand and hence there are not many included in this paper. Memory usage, runtime, used settings and achieved results should be stored in a clear and concise history. This would make any step taken in solving a machine learning problem reproducible.

To implement this, the wrappers of the machine learning engine must be subject to an interface, and a specific observer object has to be developed that keeps track of all the metrics. Preferably the framework would be aware of the major version control systems, in order for it to keep track of the code revision a specific run has been carried out in.

Intelligent feature selection

The problem of which features to select for an optimal run is not yet solved in the framework. Although we are aware of the need to calculate the individual impact of a given feature on the final score, we do not have any concrete approach for this problem yet.

Current framework prototype status

As previously mentioned this framework is a prototype and the participation in *SemEval* also serves as a test to evaluate its viability. This chapter details the current status of the prototype and its possible future development.

Open known issues

Impact of the order of tokenizers, extractors and features on the F1-Score

Recent test runs have shown that the order in which components are added to the pipeline have an impact on the achieved F1-Score. Current hypothesis suggest that this is an issue of Liblinear's reliance on random instances and a strict order of the features. However this has not yet been confirmed and needs further investigation.

Limitations of Java 8 functional programming

For the first time Java 8 supports the use of lambda expressions and closures. However they are merely syntactic sugar that resolves in the usage of anonymous inner classes. They do allow the usage of variables in the surrounding context of the closure, but they have to be final. This has led to certain problems and some inelegant code within the project. Though it has no influence on the functionality of the framework further development could be hindered by workarounds in the code that make the whole framework more rigid. A possible solution to this problem would be the use of another language. Due to the reliance on several Java libraries, *Scala* would be a premier candidate.

Possible future framework development

Rigorous testing

Before publishing this framework's 1.0 version, strict unit testing should be implemented to assure correct functionality to its users. In the current phase some unit tests have been implemented, especially for the used features, but they are not part of the framework. Due to the framework's reliance on third party libraries, testing will need some conceptual work to rule out bugs caused by these inclusions.

Technical documentation

As with the testing, a technical documentation for the users is required before publishing this framework. Additionally several code examples could help further the understanding of the framework for users.

Implementing SemEval in the framework

In this chapter the usage of the framework to participate in the *SemEval* contest is discussed. Focus lies on the selection of suitable tokenizers, mutations, extractors and features as well as descriptions of all the used components. In the next chapter “Measurements and Results” the scores we achieved with the below described components can be found.

The NRC-Canada baseline

NRC Canada has participated in several *SemEval* tasks with great success. It is self-evident that we heavily relied on their feature selection and set their score on the 2013 data set as our primary goal to achieve.

Used Mutations

We normalized all URLs to “XsomeURL” and all usernames to “XsomeUsername”.

Used Metadata

Metadata is any extracted information that serves to enrich the information of the document. These are then used to compute various features described below. The metadata we utilize are the following:

POS-Tags

Part of Speech tags denote the kind of a specific word given its surrounding context. This abstraction can be very powerful in situations where the semantic meaning of a word is not of paramount concern. In our case we utilized the TweetNLP[4] framework to compute the POS-Tags of a document.

Lemmas

Lemmas are generated by mapping the inflected forms of a word to its dictionary form. The StanfordNLP framework has an implementation to compute these values.

Spellchecking

Although spellchecking has turned out to yield results with an improvement of up to ~0.4% in certain tests, it is only implemented very rudimentarily yet. Each token is compared with a wordlist consisting of around 30‘000 English words and the *Levenshtein distance* is calculated. The word with the shortest distance is then chosen (or no word if the distance was 0) and annotated in the token. The relatively large increase in the F1-Score is mainly due to a reduced amount of n-grams (given that there are significantly more overlaps). As of now the spellchecking comes at a very high price in terms of performance. The unnecessarily high amount of distance calculations could be reduced with manageable effort.

Negation scope

Negation plays a role in the field of semantic analysis and we used Christopher Potts’ online tutorial [6] to model negation. It uses a regular expression to detect a list of predefined negation keywords and extends the scope of the negation to the end of a sentence.

Planned features

We created a table of features we planned to implement to achieve a better F1-score. At the time of the writing of this paper not all features have been implemented. The ones that are implemented are documented in "Used features"

Table 2: planend features

Feature	Imple-mented	Refer-ence
Avg./Min./Max. values over all pre-trained GloVe vectors for all occurring words in a tweet	Yes	NEW
Contiguous and non-contiguous n-grams. POS tag substitution for non-contiguous n-grams instead of the generic wildcard.	Yes	[7]
Contiguous character n-grams 3 to 5	Yes	[7]
Number of words that are all capital letters.	Yes	[7]
The number of occurrences of each part of speech tag	Yes	[7]
The number of hashtags	Yes	[7]
Lexicon feature: The total amount of tokens with an emotion greater than 0	Yes	[7]
Lexicon feature: total score of all tokens in a tweet	Yes	[7]
Lexicon feature: maximal score of a token in a tweet	Yes	[7]
Lexicon feature: the score of the last token in a tweet	Yes	[7]
The number of contiguous punctuation mark (exclamation or question marks)	Yes	[7]
Whether the last token contains an exclamation or a question mark	Yes	[7]
Presence or absence of positive/negative emoticons	No	[7]
Whether the last token (or word) is a positive/negative emoticon	Yes	[7]
The number of elongated words (a character repeated more than 2 times)	Yes	[7]
Presence or absence of tokens from Brown-Clusters generated by the CMU POS-Tagging tool	Yes	[7]
The number of negated contexts	Yes	[7]
The presence of URL or hashtag, one feature each	No	[8]
The presence of a question mark token in the tweet	No	[8]
Feature weighing: If the original token is all upper case, increase the weight of the feature	No	[8]
Feature weighing: If the original token has elongation, increase the weight of the feature	No	[8]
Feature weighing: the token is adjacent to an emoticon. Increase/decrease depends on emoticon	No	[8]
Feature weighing: the score of each token is divided in half if it is in question context	No	[8]
Weighing of a term by ΔBM25 heuristic (Paltoglou and Thelwall, 2010)	No	[9]
Concise Semantic Analysis (Li et al 2011) (Monroy et al 2013)	No	[9]
Emoticons: Sum of all scores	No	[10]

total length of the tweet	No	[10]
average length per word	No	[10]
number of words	No	[10]
topic modelling (id of the corresponding topic, semantic similarity)	No	[10]
Most common punctuation	No	[10]
Last punctuation in tweet	No	[10]
number of words surrounded by dashes or asterisks	No	[10]
POS n-grams	Yes	[11]
Dependency parsing using StanfordNLP[12] Toolkit	No	[13]
Punctuation of the last token (whether the last token contains punctuation or not)	No	[14]
Ratio of tokens that were able to be matched to a Brown cluster.	No	NEW
Lemma n-gram / Lemma bag of words	No	NEW

Used features

It has to be noted that the features itself are not part of the developed framework and vary from task to task. The features listed below are those we implemented for the SemEval 15 task. As a basis for the feature selection we decided to use the paper “NRC-Canada: Building the State-of-the-Art in Sentiment Analysis of Tweets” by S. M. Mohammad, S. Kiritchenko, and X. Zhu [7].

The team of NRC-Canada proved with the F1-Score they reached that their approach was suitable basis. Apart from this paper we also consulted their follow-up paper from 2014 titled “NRC-Canada-2014: Recent Improvements in the Sentiment Analysis of Tweets” by the same authors [15].

As part of our research we also read many of the other contestants’ papers on the most commonly used features, so we would be able to prioritize our work. We implemented the following features:

n-grams

N-grams are an ordered list of n contiguous tokens in a sentence. We modeled several n-gram features which all evaluate the presence or absence of n-grams in a document. As for an example with the sentence “The weather is nice today.” the following 3-Grams would occur: “The weather is”, “weather is nice” and “is nice today”.

Besides the normal n-grams we also modeled non-contiguous n-grams where one or more tokens in the n-gram are replaced with wildcards or the respective POS-tags. Additionally, character n-grams have been implemented which represent the presence or absence of substrings of words.

Using this sort of non-contiguous wildcards seems to have a positive impact on the overall F1-score against our test data as can be seen in “Measurements and Results”.

Dictionary features

Several dictionaries, both manually created and computer generated, have been used to determine the sentiment behind a given word. Besides our own implementation we used Team Swiss-Chocolate’s[11] to complement our own since their combined use yielded the best results. The dictionary files used by the framework are the same that have been used by NRC Canada in 2014.

In order to provide programmer-friendly use of dictionary functions within the framework it was necessary to identify similar attributes and behaviours of all dictionaries with the hope of finding a scheme which then could be added to the framework.
Certain distinctive features emerged and led to a division into two groups:

> *Group 1*: Dictionaries consisting solely of a list of words with similar meaning (e.g. a list of negative words). We call these simply “Dictionaries”. Dictionaries implement the *IDictionary* interface which provides very limited functionality. They only return a *true/false* value indicating if a dictionary contains a word or not.
>
> *Group 2*: Dictionaries containing words with weighted attributes (e.g. good = +1, bad = -1). We call these “*Tone Dictionaries*”. They can be seen as extensions to Group 1 as they provide the same functionality as well as some more: Tone dictionaries implement the *IToneDictionary* interface and therefore provide access to additional methods such as emotion counts on tokens, sentences, paragraphs and entire documents (this sums up all the values in a given scope).

Number of POS-tags

The number of occurrences of each encountered POS-tag. As described in the section “Used Metadata” tokens are annotated with Part-of-Speech-tags computed by the TweetNLP[4] framework. For each type of POS-tag one entry in the feature vector indicates the number of tokens associated with that tag.

Brown cluster feature

The TweetNLP[4] provides two files containing word clusters of the English language, computed with the Brown clustering algorithm. This means that many different words will be mapped to a single semantically close word; this reduces the overall amount of different words within the tweets allowing for a less sparse representation of the content.
In our feature we check whether a token can be matched to a cluster and, if so, to which one. This results in a vector entry similar to a uni-gram (n-gram of length 1).

GloVe feature

GloVe is being developed by the Stanford University and is a way to get a vector representation of words. We then calculate the maximum, minimum and average values of these representations. More information about this feature can be found on the Stanford NLP website[16].

Continuous punctuation

The number of substrings consisting of two or more continuous punctuation marks of the same kind.

Number of all capital tokens

The number of tokens which are written entirely in capital letters. This is often considered to represent a raised voice by the online community.

Number of elongated words

The number of elongated words is counted by checking each token for substrings of length three or more, solely consisting of the same character.

Number of hash tags

This feature counts the number of hash tags used by a given tweet. It considers each token starting with a "#".

Number of negated contexts

The number of sentence fragments in a tweet, which are bracketed within a negation word and a punctuation mark. For a more detailed description of negation please refer to the section "Used Metadata".

Team Swiss-Chocolate's String featurizer[11]

Team Swiss-Chocolate implemented an application to participate in *SemEval* 2014. We used it as a reference for a few features and wrote a wrapper to use their code directly in our project. The features we use are character n-grams as well as certain dictionary features.

Scaling

Through scaling values of features are transformed into values between a certain minimum and maximum. This makes sure that for example statistical outliers are not weighed too much and features that count occurrences do not outweigh Boolean-decision features.

We tried different mechanisms to scale features, depending on various attributes of these, such as whether they are *sparse* or not, their value range and so on. For our n-gram features we implemented a logarithmic TF-IDF scaling, for our dictionary features we implemented a sigmoid function to scale the feature values.

TF-IDF Scaling

TF-IDF stands for term frequency – inverse document frequency. With our value tracking mechanisms we track the number of generated feature vectors (1 per document) and the number of uses of each n-gram. The basic idea is that the lower the ratio between uses of a specific n-gram and the number of vectors, the more information that n-gram holds and the higher its weight should be. Our calculations are as follows:

$$\mathrm{N} := \{the\ set\ of\ all\ occuring\ N - Grams\}$$
$$v \in \mathrm{N}$$

$$frequency_v := \frac{NrOfUses_v}{NrOfVectors}$$

This will have a maximum value of 1 because the number of uses of an n-gram cannot be larger than the number of vectors. This means that

$$0 < frequency_v \ \leq 1$$

holds true. To get the actual scaling factor needed to calculate the new vector values a logarithmic function is used.

$$scalingFactor_v := -\log(\mathrm{frequency}_v) + 1$$

This leads to the following conclusion:

$$0 < frequency_v \ \leq \ 1 \rightarrow \lim_{frequency_v \rightarrow 1} scalingFactor_v = 1$$

This means that the higher the frequency of a given n-gram is, the lower is the scaling factor.

In a given vector the n-gram v is then multiplied by its respective scaling factor $scalingFactor_v$. Because this can lead to very large values we need to scale this back to a value between 0 and 1. This is achieved by getting a max value

$$maxScalingFactor := \max(scalingFactor_v)\ for\ v \in \mathrm{N}$$

and then dividing every previously scaled value by this new factor.

It has to be noted that this way of scaling has proven detrimental to our overall F1 Score. These tests were executed with the baseline configuration of the ablation tests described in “Measurements and Results” on the test sets of 2013 and 2014.

Table 3 TF-IDF Scaling Results

Test set	Using TF-IDF scaling?	Result
2013	No	69.4602
2013	Yes	67.2556
2014	No	68.1685
2014	Yes	69.4909

Sigmoid Function on dictionary features

For our dictionary features we implemented a sigmoid function to keep the values between 0 and 1. The sigmoid function is defined as follows.

$$Sigmoid(x) = \frac{1}{1 + e^{x}}$$

All dictionary features as well as the dictionary features within Swiss-Chocolate's[11] code utilize the sigmoid function. In the below table the increase in F1-Score is shown.

The following measurements were executed with the baseline configuration of the ablation tests on the test sets 2013 and 2014. Refer to "Measurements and Results" for further information on the testing configuration.

Table 4 Sigmoid Function Results

Test set	Using Sigmoid function?	Result
2013	No	33.6697
2013	Yes	69.4602
2014	No	42.3896
2014	Yes	68.1685

The current hypothesis for this significant score drop is that without the sigmoid function certain features can have very large values while all other features are close to between -1 and 1.

Measurements and Results

In this chapter the achieved scores on the various test data are shown. Each measurement documents the configuration of the framework, used test data and parameters for the Liblinear machine learning library.

The final F1-Score is calculated as the mean of the positive and negative F1-Scores. Results are rounded to 4 digits after the decimal point.

Due to the way the data is constructed by *SemEval* the upper bound of achievable score is 75%.

Because of the usage of randomized values in the Liblinear-library the results have a certain degree of uncertainty. The extent of this uncertainty is currently unknown by the authors.

Test Data

The Test Data consist of two bundles provided by SemEval. The first is the training and evaluation data of 2013. We use this data to compare our framework's performance to the 2013 results of NRC Canada. The other bundle consists of the current 2014 data used in last year's competition.

Table 5: Test Data overview

Test Data Name	**Year**
2013/task-B-train.tsv	2013
2013/task-B-dev.tsv	2013
task-B-test2013-twitter.tsv	2013
2014/task-B-train.tsv	2014
2014/task-B-dev.tsv	2014
task-B-test2014-twitter.tsv	2014

Ablation testing

Description

Once we established a set of features that worked reasonably well, we would start removing single features and in thematic groups to find out which have the most impact on our score.

Initial feature set and configuration

For more in-depth descriptions of the features, refer to the chapter “Used features”.

Feature name	Configuration information
n-gram	1 to 4 grams with the word values and 1 to 4 grams with the lemma values
Non-contiguous n-grams	3 to 4 grams with the the middle part replaced by the * wildcard and 3 to 4 grams with the middle part replaced by the POS-tags where available
Number of all capital tokens	Non variable configuration
Number of hashtags	Non variable configuration
Number of POS-tags	Non variable configuration
GloVe feature vectors	Non variable configuration
Number of negated contexts	Non variable configuration
Number of elongated words	Non variable configuration
Last token contains punctuation	Checks for full stops, exclamation marks and question marks
Continuous punctuation	Checks for exclamation marks and question marks
Brown cluster feature	Non variable configuration
Whether the last token is negative	On all available lexica
Whether the last token is positive	On all available lexica
The score of the last token	On all available lexica
The maximum score found	On all available lexica
The total score	On all available lexica
The number of tokens with positive emotion	On all available lexica
The emotion of the last token	On all available lexica
Team Swiss-chocolates string featurizer	All lexica features with sigmoid function, as well as the character n-grams with 3 to 5 in length.

Used Testing Data

Trained on:
2013/task-B-train.tsv
2013/task-B-dev.tsv

Tested on:
task-B-test2013-twitter.tsv

Results

For these tests we used an N-Fold type optimization with a C step size of 0.1 and an N-Fold size of 10. The best result is marked by bold writing.

Table 6 Ablation test results on test set 2013

Subtracted feature(s)	**Pos F1 Score**	**Neu F1Score**	**Neg F1 Score**	**Fin F1Score**
None	**72.2539**	**75.3752**	**66.6666**	**69.4603**
N-gram	71.5308	74.5819	64.7343	68.1325
Noncontiguous n-gram	72.3891	75.2151	65.8517	69.1204
Number of all capital tokens	71.7508	75.0417	66.0729	68.9118
Number of hashtags	72.1627	75.0694	65.7938	68.9783
Number of POS-tags	72.0372	75.3124	65.4221	68.7297
GloVe Vectors	72.3735	74.5359	65.0420	68.7078
Number of negated contexts	71.8022	75.1041	65.9091	68.8557
Number of elongated tokens	72.4027	75.2772	66.0641	69.2334
Last token contains punctuation	71.8627	74.8130	65.2459	68.5543
Continuous punctuation	72.1805	75.3671	65.8497	69.0151
Brown clustering	72.0372	75.0070	65.8576	68.9474
Team Swiss-Chocolate’s featurizer	71.5953	74.0166	62.2372	66.9163
Last token is negative	72.2202	75.2355	65.9557	69.0880
Last token is positive	71.9971	75.2980	65.8537	68.9254
Score of the last token	71.9543	75.1663	65.4635	68.7089
Maximum Score of any token	72.3373	75.1867	65.1278	68.7326
Total score	71.8022	75.1596	65.9091	68.8557
Tokens with positive emotion	71.8481	74.9931	65.4129	68.6305
Emotion of the last token	71.9543	75.1663	65.4635	68.7089
All n-gram features	71.8335	75.0069	65.4781	68.6558
All dictionary features	70.4554	74.1487	61.9835	66.2194

Used Testing Data

Trained on:
2014/task-B-train.tsv
2014/task-B-dev.tsv

Tested on:
task-B-test2014-twitter.tsv

Results

For these tests we used an N-Fold type optimization with a C step size of 0.1 and an N-Fold size of 10.

Table 7 Ablation test results on test set 2014

Subtracted feature	Pos F1 Score	Neu F1Score	Neg F1 Score	Fin F1Score
None	74.4321	69.4710	61.9047	68.1684
N-gram	74.9854	68.8420	61.9048	68.4451
Noncontiguous n-gram	74.6806	69.4140	64.0371	69.3589
Number of all capital tokens	74.4619	69.3196	62.9370	68.6995
Number of hashtags	74.5349	69.1026	63.3803	68.9578
Number of POS-tags	74.6078	68.5861	61.6114	68.1096
GloVe Vectors	74.6251	68.0493	61.5385	68.0818
Number of negated contexts	74.4619	69.2161	61.7225	68.0922
Number of elongated tokens	75.1600	69.4980	62.8175	68.9888
Last token contains punctuation	74.4916	69.9552	64.1509	69.3213
Continuous punctuation	74.2857	69.2258	63.0841	68.6849
Brown clustering	74.4321	69.3538	63.3803	68.9062
Team Swiss-Chocolate's featurizer	72.6310	68.7970	57.9075	65.2693
Last token is negative	74.3590	69.2652	63.9999…	69.1795
Last token is positive	74.2256	69.4323	63.5514	68.8885
Score of the last token	74.8109	69.5597	62.8571	68.8340
Maximum Score of any token	74.6512	69.3982	64.1509	69.4011
Total score	74.4916	69.5597	63.1579	68.8247
Tokens with positive emotion	74.2857	69.2602	62.4113	68.3485
Emotion of the last token	74.8109	69.5597	62.8571	68.8340
All n-gram features	**74.4916**	**69.7106**	**65.1163**	**69.8039**
All dictionary features	72.5431	69.1739	57.7777…	65.1605

Conclusions drawn from ablation testings

The ablation testing shows that removing even a single feature has an unpredictable impact. The simple removal of feature groups, especially dictionary features, entails significant consequences. The unexpected performance of the ablation test (on test set 2014) without any n-gram features may be attributed to the random seed that the used classifier utilizes. In order to precisely identify the inherent randomness further analysis of the Liblinear classifier would be required.

The most important conclusion drawn from this testing is, that high quality dictionaries and the use of a wide variety of dictionary features brings a significant boost in F1-score.

Results of the participation in SemEval

The name of the team the authors participated with is SwissChocolate. The results are drawn from http://alt.qcri.org/semeval2015/task10/index.php?id=results as off the 04.01.2014. The document that is linked there will serve as a source and can be found in the literature index [17]. The relevant parts can be found here.

“The submissions marked in yellow were submitted after the deadline.”[17]

Table 8 SemEval Task 10 Subtask B Results

	Team	Twitter 2015
1	Webis	64.84
2	unitn	64.59
3	lsislif	64.27
4	INESC	64.17
5	Splusplus	63.73
6	wxiaoac	63.00
7	IOA	62.62
8	Swiss-Chocolate	62.61
9	CLaC-SentiPipe	62.00
10	TwitterHawk	61.99
11	SWATCS65	61.89
12	UNIBA	61.55
13	KLUEless	61.20
14	NLSLB2015	60.93
15	ZWJYYC	60.77
16	Gradiant-Analytics	60.62
17	UIR-PKU	60.03
18	mockingjay	59.83
19	ECNU	59.72
20	CIS-positiv	59.57
21	SWASH	59.26
22	GTI	58.95
23	iitpsemeval	58.80
24	elirf	58.58
25	SWATAC	58.43
26	SWATCMW	57.60
27	WarwickDCS	57.32
28	SenticNTU	57.06
29	DIEGOLab	56.72
30	Sentibase	56.67
31	Whu_Nlp	56.39
32	UPF-taln	55.59
33	RGUSentimentMiners123	53.73
34	IHS-RD	52.65
35	RoseMerry	51.18
36	Wizdee	49.19
37	UMDuluth-CS8761	47.77
38	UDLAP2014	42.10
39	SHELLFBK	32.45
40	whu-iss	24.80

Table 9 SemEval Task 10 Subtask B Progress Test

	Team	LiveJournal 2014	SMS 2013	Twitter 2013	Twitter 2014	Twitter 2014 Sarcasm
1	Splusplus	75.34	67.16	72.80	74.42	42.86
2	unitn	72.48	68.37	72.79	73.60	55.44
3	INESC	69.78	63.78	71.97	72.52	56.23
4	IOA	74.52	68.14	71.32	71.86	51.48
5	lsislif	73.01	63.42	71.34	71.54	46.57
6	KLUEless	73.50	67.66	70.64	70.89	45.36
7	Webis	71.64	63.92	68.49	70.86	49.33
8	ZWJYYC	71.60	64.72	69.56	70.77	46.34
9	TwitterHawk	70.17	62.12	68.44	70.64	56.02
10	CLaC-SentiPipe	73.59	63.05	70.42	70.16	51.43
11	wxiaoac	73.36	64.04	66.43	68.96	54.38
12	Swiss-Chocolate	73.95	65.56	68.80	68.74	48.22
13	UIR-PKU	70.65	65.32	93.62	68.12	54.22
14	NLSLB2015	66.12	61.05	66.96	67.45	39.87
15	SWATCS65	73.37	65.49	68.21	67.23	37.23
16	mockingjay	69.91	62.25	65.68	67.04	57.50
17	Gradiant-Analytics	72.63	61.97	65.29	66.87	59.11
18	SenticNTU	68.70	60.53	63.50	66.85	45.18
19	SWATAC	68.67	61.30	65.86	66.64	39.45
20	ECNU	74.40	68.49	65.25	66.37	45.87
21	CIS-positiv	71.47	65.14	64.82	66.05	49.23
22	GTI	70.50	63.50	64.03	65.65	55.38
23	SWATCMW	69.52	65.43	65.67	65.62	37.48
24	WarwickDCS	68.98	61.92	66.57	65.47	45.03
25	UNIBA	70.05	65.50	61.66	65.11	37.30
26	iitpsemeval	73.70	60.56	60.78	65.09	47.32
27	UPF-taln	64.50	57.84	66.15	65.05	50.93

28	DIEGOLab	63.74	58.60	62.49	63.99	47.62
29	Whu_Nlp	71.83	61.31	65.97	63.93	46.93
30	Sentibase	67.55	59.26	61.56	63.29	47.07
31	SWASH	69.43	56.49	63.07	62.93	48.42
32	whu-iss	61.98	54.28	56.51	61.31	47.78
33	RoseMerry	62.54	53.00	52.33	61.27	49.25
34	elirf	68.33	60.20	57.05	61.17	45.98
35	RGUSentimentMiners123	64.39	57.14	56.41	59.44	44.72
36	UMDuluth-CS8761	60.23	50.64	54.17	55.82	43.74
37	Wizdee	57.94	46.59	49.37	53.92	42.07
38	UDLAP2014	50.11	39.35	41.93	45.93	41.04
39	SHELLFBK	34.06	26.14	32.14	32.20	35.58
40	IHS-RD	???	???	???	???	???

Conclusion of this paper

In this paper we showed how we developed an abstract and extensible framework for text classification with machine learning. The participation in SemEval was a goal as well as a proof of concept for the design we implemented.
We have shown that implementing abstraction layers and a module driven design allows for rapid development of features and flexible ways to integrate the framework with existing software.

The achievement of rank 8 on the new 2015 data test and rank 12 on the progress test has shown that a flexible framework serves as a capable foundation for text classification. Considering the timeframe in which this paper had to be written, and the fact of having had to develop both the framework and the specific implementation for the participation in SemEval, the results are satisfactory.

Glossary

A	
B	
C	
D	
E	
F	
Feature	Framework component which analyses a document and returns a vector.
G	
H	
I	
J	
K	
L	
Levenshtein distance	The Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other.
M	
Mutation	Manipulation on input text (tweet) before tokenizing (e.g. removing double whitespaces).
N	
O	
P	
Q	
R	
S	
SVM	See *Support Vector Machine*
Support Vector Machine	Algorithm used by the framework to classify inputs. Details can be found under “Machine learning”.
Sparsity	(from sparse) Thinly scattered or distributed
T	

U
V
W
X
Y
Z

Indices

Literature index

[1] M. Cieliebak, “PADA - Archiv.” [Online]. Available: https://tat.zhaw.ch/pada_archiv/arbeit_archiv.jsp?command=show&vers=115&lang=de&arbeitID=13276. [Accessed: 02-Dec-2014].

[2] SemEval2015, “SemEval-2015 Task 10: Sentiment Analysis in Twitter < SemEval-2015 Task 10.” [Online]. Available: http://alt.qcri.org/semeval2015/task10/. [Accessed: 02-Dec-2014].

[3] “SemEval - Wikipedia, the free encyclopedia.” [Online]. Available: http://en.wikipedia.org/wiki/SemEval. [Accessed: 04-Jan-2015].

[4] “Twitter Natural Language Processing -- Noah’s ARK.” [Online]. Available: http://www.ark.cs.cmu.edu/TweetNLP/. [Accessed: 18-Dec-2014].

[5] C.-J. Lin, X.-R. Wang, C.-J. Hsieh, K.-W. Chang, and R.-E. Fan, “LIBLINEAR -- A Library for Large Linear Classification.” .

[6] C. Potts, “Sentiment Symposium Tutorial: Linguistic structure.” [Online]. Available: http://sentiment.christopherpotts.net/lingstruc.html. [Accessed: 08-Nov-2014].

[7] S. M. Mohammad, S. Kiritchenko, and X. Zhu, “NRC-Canada: Building the State-of-the-Art in Sentiment Analysis of Tweets,” *Proc. seventh Int. Work. Semant. Eval. Exerc.*, vol. 2, no. SemEval, pp. 321–327, Aug. 2013.

[8] G. Tobias, J. Vancoppenolle, and R. Johansson, “RTRGO : Enhancing the GU-MLT-LT System for Sentiment Analysis of Short Messages,” no. SemEval, pp. 497–502, 2014.

[9] S. Amir, M. Almeida, B. Martins, and J. Silva, “TUGAS : Exploiting Unlabelled Data for Twitter Sentiment Analysis,” no. SemEval, pp. 673–677, 2014.

[10] S. Pinto, A. Bento, H. Gonc, and P. Gomes, “CISUC-KIS : Tackling Message Polarity Classification with a Large and Diverse set of Features,” no. SemEval, pp. 166–170, 2014.

[11] M. Jaggi, E. T. H. Zurich, and M. Cieliebak, “Swiss-Chocolate : Sentiment Detection using Sparse SVMs and Part-Of-Speech n -Grams,” no. SemEval, pp. 601–604, 2014.

[12] C. D. Manning, J. Bauer, J. Finkel, and S. J. Bethard, “The Stanford CoreNLP Natural Language Processing Toolkit.”

[13] A. Denis, S. Cruz-lara, N. Bellalem, and L. Bellalem, “Synalp-Empathic : A Valence Shifting Hybrid System for Sentiment Analysis,” no. SemEval, pp. 605–609, 2014.

[14] C. Van Hee, M. Van De Kauter, D. Clercq, and E. Lefever, “LT3 : Sentiment Classification in User-Generated Content Using a Rich Feature Set,” no. SemEval, pp. 406–410, 2014.

[15] X. Zhu, S. Kiritchenko, and S. M. Mohammad, “NRC-Canada-2014 : Recent Improvements in the Sentiment Analysis of Tweets,” no. SemEval, pp. 443–447, 2014.

[16] C. D. M. Jeffrey Pennington, Richard Socher, “GloVe: Global Vectors for Word Representation.” [Online]. Available: http://nlp.stanford.edu/projects/glove/. [Accessed: 02-Dec-2014].

[17] “SemEval-2015 Task 10 Results (Official) - Google Docs.” [Online]. Available: https://docs.google.com/document/d/1WV-

XTvQDpuH_IfKrjzeZ361s1ykcskDNNuOV3oI39_c/edit?usp=sharing&pli=1. [Accessed: 04-Jan-2015].

Table index

Image index

Addendum

SemEval2015 Task 10 Subtask B

“Subtask B: Message Polarity Classification: Given a message, classify whether the message is of positive, negative, or neutral sentiment. For messages conveying both a positive and negative sentiment, whichever is the stronger sentiment should be chosen.”[2]

Original task description (German)

„In der Sentiment-Analyse soll für einen Text entschieden werden, ob dieser positiv, negativ oder neutral ist. Ein weitverbreiteter Ansatz hierfür ist die Verwendung eines Classifiers, der mit einer annotierten Dokumentmenge trainiert wird. Der Classifier verwendet dazu verschiedene Features wie n-grams, Satzzeichen oder Wort-Cluster.

Es gibt eine sehr grosse Menge von potentiellen Features, und viele wurden bereits in verschiedenen Systemen implementiert. Dabei hat sich gezeigt, dass verschiedene Kombinationen von Features unterschiedlich gute Performance erzeugen.

In dieser Arbeit soll ein System entwickelt werden, das auf Basis einer grossen Menge von Feature-Implementierungen eine Kombination auswählt, die eine möglichst gute Performance liefert.
Bei Interesse kann das Thema Sentiment-Analyse in einer Bachelor-Arbeit im nächsten Semester weiterbearbeitet werden.“ [1]