Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

InIT Institut für angewandte Informationstechnologie

Bachelorarbeit (Informatik)

Developing PlebML: A modular machine learning framework

Autoren	Marek Arnold Dominic Egger
Hauptbetreuung	Mark Cieliebak
Nebenbetreuung	Fatih Uzdilli

Datum	06.05.2015

Zusammenfassung

In der Software Industrie werden Anwendungen des maschinellen Lernens seit längerer Zeit eingesetzt, eines der Probleme dabei ist, dass dafür oft Spezialisten benötigt werden. Eine einfach zu verwendende und erweiterbare Bibliothek für maschinelles Lernen würde vielen Software Entwicklern den Einstieg in diese Thematik erleichtern. In dieser Arbeit wurde eine Software Bibliothek für maschinelles Lernen entwickelt, aufgeteilt in eine saubere Software Architektur nach den Bedürfnissen des maschinellen Lernens. Zusätzlich wurden im Rahmen dieser Arbeit Interfaces mit Standard Funktionen für einfache Textklassifikations Aufgaben entwickelt und an mehreren Datensets getestet, weiter wurden Anleitungen zur Erweiterung und Nutzung des Frameworks erarbeitet. Das Resultat ist ein robuster und erweiterbarer Kern, sowie ein Domänen spezifisches Modul zur Textklassifikation.

Abstract

For quite some time now machine learning has been of great importance in the software development industry, however it’s also a very demanding and complicated subject. Providing an easy to use software library with strong extensibility would make machine learning much more accessible to software developers new to the topic. To do this, this project offers a software library that is divided not only by the principle of clean software architecture but also by knowledge requirements in terms of machine learning. Similarly it offers a guided approach to solving machine learning tasks and extending the framework that is reflected in both its software architecture and the best practice guidelines proposed in this document. The results show a framework with a strong extensible core and problem domain specific modules which allow to tackle different machine learning tasks with the structurally same approach.

Contents

1 Introduction 6
1.1 Goals 6
1.2 User Stories and Purpose of PlebML 7
1.3 Differentiation from Existing Solutions 8
2 Software Architecture 10
2.1 Logical Separation of PlebML 10
2.1.1 Feature Vector Generation 11
2.1.2 Machine Learning Component 12
2.1.3 Pipelines as Workflow Collections 12
2.2 Source Code Modularization 13
2.2.1 Core Library 13
2.2.2 Core Modules 25
2.2.3 Tasks 32
2.2.4 Design Concerns 32
3 Text classification with PlebML 33
3.1 Model Object Class 33
3.2 Preprocessor 34
3.2.1 GeneralTokenizer 34
3.2.2 SimplestTokenizer 34
3.2.3 SimpleEMailNormalizer 34
3.3 Included Features 35
3.3.1 N-Gram Utility Class 35
3.3.2 Dictionary System 36
3.4 Pipelines and Builder 37
3.4.1 TextClassificationBuilder 37
3.4.2 AdvancedTextClassificationPipeline 38
3.4.3 BasicTextClassificationPipeline 38
4 Best Practices for Core Module Design 39
5 Measurements and Results 40
5.1 Machines 40
5.2 Performance Enhancement Evaluation 41
5.2.1 Feature Extraction 41
5.2.2 Machine Learning 41
5.2.3 Conclusions 41
5.3 Tested Datasets 42
5.3.1 Semeval 42
5.3.2 Spam 43
5.3.3 Movie Review Sentiment 44
5.3.4 Black box features 45
5.4 Feature Extraction Performance Measurements 46
5.4.1 Spam 46
5.4.2 Semeval 47

6 Review of Results and Implications for Further Work 49
6.1 Goal Review 49
6.2 Open Points 49
7 Bibliography 50
8 Image index 51
9 Table index 52
10 Appendix 53
10.1 Entry 1 – Used Features [1] 53
10.2 Entry 2 - Getting Started With PlebML 55
10.3 What Is PlebML 55
10.3.1 [BASIC] The Structure of PlebML 55
10.3.2 [BASIC] Using the Basic Pipeline 56
10.3.3 [ADVANCED] Using the Advanced Pipeline and the Abstract Pipeline 58
10.3.4 [ADVANCED] Configuring the Builder 59
10.3.5 [ADVANCED] Adding a Feature 60
10.3.6 From Here on Out 60
10.4 Entry 3 – Feature Extraction Performance Measurements 61
10.4.1 Spam 61
10.4.2 Semeval 64

1 Introduction

Machine learning has become more and more important along with the advance of Big Data. Many applications like search and recommendation engines, NLP and speech recognition have shown its practical usability. Google uses machine learning algorithms in their search engine. YouTube, Amazon, Netflix and many more make use of such techniques in their recommendation engines. With the growing amount of data available, manual programming is often just not possible anymore.
However many existing solutions are either highly specific in the machine learning tasks they can solve, only offer a collection of learning algorithms or have a system integration intended for people with a great deal of knowledge of machine learning.
With rising demand for machine learning solutions not every software developer can be an expert in machine learning and the time of experts should be used as efficiently as possible. In this project we wanted to design a machine learning library that can be integrated by a layman in the field and fine-tuned by an expert later. To achieve this we imposed a very strict pattern on how machine learning tasks should be approached to allow extension and customization by someone with very little expertise in machine learning with the appropriate specification.

This project deals with the design and implementation of a machine learning software library and requires knowledge in the Java programming language, software architecture and machine learning.

This work is based on a codebase taken from “Development of a framework for text classification and participation at SemEval.” [1] It was an entry to the Semeval 2015 contest. This code however had some flaws and as such our goals are twofold. For one redesign the framework with the above problem in mind and two, optimize the code for improved runtime performance.

1.1 Goals

In this section we describe the goals we set for PlebML at the start of the project. Some are technical, some relate to usability.

1. There should be default configurations and default workflows such that the framework can be used by someone with very little or no expertise in machine learning. The usage should consist of only very few lines of code.

2. In comparison to our codebase, performance should be improved, both in terms of RAM usage and elapsed runtime.

3. The software architecture needs to be overhauled as the separation of concern is weak, the interfacing has issues, extending the framework is very cumbersome and implementing a wide variety of tasks is very hard.

4. Designing best practices for various workflows concerning the extension and customization of PlebML itself. Part of designing certain extensions is providing the needed code to satisfy goal 1.

1.2 User Stories and Purpose of PlebML

PlebML is intended to give software developers with very limited knowledge of machine learning access to a full-fledged machine learning tool. To keep the requirements of machine learning low we impose a structured approach to solve a machine learning task and divide the extensibility of PlebML along certain knowledge levels. This structure will be topic of discussion in the “Software Architecture” chapter. To cite the “Getting Started with PlebML” [2] guide of PlebML:

“*What is PlebML?*

PlebML is an easily extensible machine learning framework intended to give various levels of interaction needing different levels of machine learning expertise ranging from novice (should know what features are and what training and prediction is) to expert. PlebML is made up of three parts. The Core Library which contains the actual machine learning algorithms and wrapper to machine learning libraries, as well as various utilities, interfaces and abstract classes needed by the second part.
Core Modules provide modular functionality for a specific kind of machine learning tasks such as our stock module for text classification. A Core Module contains the class responsible for a sensible representation of the raw data within the system as well as the actual features and some other things which will be discussed below. Extending and creating these Core Modules is one of the main topics of this guide.
The third and easiest part of PlebML are Tasks. Tasks consist of two parts. One is the handling and importing of the raw data until it can be handed to a Core Module. The second part is interaction with a Core Module’s Pipeline and Builders. It is also the topic of the BASIC level chapters in this guide.” [2]

To further specify our goals we defined a set of user stories. They also show how typical use cases for PlebML.

- As a software developer tasked with implementing machine learning in an existing software project, I want to be able to seamlessly integrate an existing configuration of the library without having to configure the machine learning part of the library itself.
  - Using the stock text classification configuration of PlebML to compete in a Kaggle or Semeval competition.
  - Or to implement a spam filter for a mail server.
  - Or sort news article by topic.

- As a machine learning expert I want to be able to extend this library to fulfil a wide variety of different machine learning tasks.
  - Extending the existing text classification module with new features to handle non-English languages or non-Latin alphabets better.
  - Adding a new module to handle picture recognition or any other non-text based machine learning task.

- As a software developer when integrating the library I don’t want to be forced to implement all the existing code. I want to be able to only import what I need. The library should be modular.
  - When doing a text classification task, I should not have to have superfluous code dealing with picture recognition or vice versa.

- As a software developer I want to be able to integrate the library into an existing machine learning environment to either take over feature vector generation or machine learning.
  - When using Spark MLib or the Google Prediction API or something similar I can use the feature vector generation mechanism of PlebML to generate the vectors needed by those library.
  - Or the other way around, if I have feature vectors already I should be able to use the machine learning component of PlebML without having to use the feature vector generation previously.

- As a software developer I want to be given a set of guideline on how to extend PlebML as well as best practices to conform to.
  - When designing a new module or additions I should have a reference stating clearly what those additions should fulfil.

1.3 Differentiation from Existing Solutions

Initially when looking for other existing solutions we only looked for text classification tools. However we quickly decided to generalize PlebML to accomplish other tasks as well. As such the comparison between those tools and PlebML is in regards to the stock configuration of PlebML which includes text classification capability. To get a more general comparison we also picked several state-of-the-art machine learning tools.

- *MALLET* [3]
  MALLET is a software library for a wide variety of text analysis.

  "MALLET is a Java-based package for statistical natural language processing, document classification, clustering, topic modelling, information extraction, and other machine learning applications to text." [3]

  It offers both an API and a command line tool. In some points we have similar architecture. MALLET generates feature vectors by passing the initial raw data through a pipeline made up of serially called components. PlebML offers a similar approach but is strongly structured and made up of configurable stages in the pipeline instead of single components. MALLET represents the read raw data within a predefined class. This is done because MALLET only deals with Text. PlebML can be extended to deal with a wide variety of raw data and as such can use any class to represent imported raw data.

- *DKPro* [4]
  Again like MALLET, DKPro deals with text analysis only.

  "We present DKPro TC, a framework for supervised learning experiments on textual data. The main goal of DKPro TC is to enable researchers to focus on the actual research task behind the learning problem and let the framework handle the rest." [5]

  DKPro can build on top of several other frameworks including MALLET, Liblinear and Weka. One major point of DKPro is a workflow which ensures that all experiments are fully replicable. This is a feature not given in PlebML. With this it differs in the primary use case for PlebML being integration into a surrounding project and for DKPro being experimental work.

- *RTextTools* [6]
  RTextTools is written in the programming language R. This language is mostly used for statistics. This already changes certain approaches to data handling as PlebML is written in Java. It focuses on text classification with nine different algorithms and offers a nine step workflow on how to get to a trained model and use that. This makes it very different from PlebML which builds on the idea of task specific modules with an overarching static workflow on how problems should be approached. Much like DKPro it is meant to be used as standalone software not to be integrated into existing projects.

- *Google Prediction API* [7]
  The Google Prediction API is a cloud based RESTful API for a variety of machine learning tasks. Training data can be uploaded and then be trained on the Google cloud, after that the resulting model can be queried for prediction results. The uploaded training data consists of an expected prediction result and a feature vector made up of numeric or text features. If only given the raw data the user has to make his own feature extraction. As such this API could be plugged into PlebML as a machine learning component.

- *Amazon ML Service [8]*
  Amazon ML Service is a cloud based API for various machine learning tasks. Users have to upload training data in a CSV file afterwards the web interface transformations, like splitting up address strings in more meaningful features (ZIP, state etc.), to the data can easily be applied and finally there is a support in the feature selection based on the training data provided by the user, but only basic features like n grams are implemented, complex features like dictionaries have to be applied externally. The Amazon ML Service could be used in PlebML as a machine learning component, in this way the easy extraction of advanced features in PlebML would be combined with the cloud based machine learning API of Amazon.

- *Knime [9]*
  Knime is an open source analytics platform written in Java. It is heavily modularized and extensible to suit a huge variety of tasks. Knime offers a graphical interface to setup workflows easily and through plugins for specific task categories like text classification specialized features are available. Furthermore the modular design allows the use of a variety of open source libraries within Knime and to extend it as needed. Its main disadvantage in contrast to PlebML is the lack of default workflows for given sets of problems, and therefore some knowledge about machine learning is required to get any useful results.

- *Scikit-learn [10]*
  Scikit-learn is an open source machine learning project written in Python that is built on NumPy, SciPy, and matplotlib. Its API offers a variety of methods for supervised and unsupervised training and further it offers pipelines to simplify workflows. Some basic mechanics for the feature extraction and the transformation of the resulting feature vectors are implemented, but advanced features like dictionaries are not included. The main advantage of PlebML compared to scikit-learn is the variety of implemented features for text classification.

- *Spark MLib* [11]
  Spark MLib by Apache is part of the spark cluster computing system. It does not contain a full suite of functionalities for feature vector generation. Instead it focuses on the learning algorithms and utility side of things. As such it could be plugged into PlebML as a machine learning component. While it does have some feature extractors for text (such as TF-IDF and Word2Vec) it does not have any means for text segmentation. Instead they refer to Stanford's NLP Work. Comparing the existing machine learning part of PlebML (A modified version of Liblinear [12]) Spark MLib offers functionality that Liblinear does not, such as dimensionality reduction. PlebML offers those functionalities either by itself or does not need them as of now.

- *Apache Mahout [13]*
  Apache Mahout is a suit of machine learning libraries. It offers a huge variety of machine learning functions and provides high scalability for some tasks with Hadoop. The focus of the suit is to provide highly scalable and robust machine learning algorithms but it does not provide direct support for the feature vector generation. The main disadvantage of Mahout is its complexity, which leads to a huge initial effort when using Mahout. Additionally to generate feature vectors external tools have to be used. Apache Mahout could be integrated in PlebML as a highly scalable machine learning component to provide the variety of Mahouts algorithms with the simplicity of PlebML.

- *Weka 3 [14]*
  Weka is a collection of machine learning algorithms for data mining tasks written in Java. The main goal of the Weka project is to make machine learning algorithms publicly available. As Apache Mahout, Weka 3 does not offer direct support for the feature extraction, but it could be integrated in PlebML as a machine learning component.

- *Gate [15]*
  The Gate product family is a set of frameworks, "*It is specifically targeted at NLP tasks including text classification, chunk learning (e.g. for named entity recognition) and relation learning.*" *[16]* The implemented feature extraction mechanisms and the machine learning algorithms are configurable via a configuration file, further many functionalities are available via plugins. For the machine learning Gate integrates LibSVM and it offers and interface to Weka. The disadvantage of Gate is the huge configuration file needed for complex tasks which makes it difficult for inexperienced users.

2 Software Architecture

In this chapter the software architecture and modularization concept of both, the logical and the source code level are explained. It also shows how various core concepts of PlebML are separated and interact with each other.

PlebML has two different levels of modularization. The first that will be laid out in this section deals with the logical separation of parts, this separation is intended to give a structured way to solve a machine learning task. The second level of separation deals with how the source code is distributed in packages and deals with more technical questions such as reusability and separation of concern.

2.1 Logical Separation of PlebML

There are three distinct parts in the logical separation that are intended to give a structured approach to solve a machine learning task with PlebML as shown in *Figure 1*.

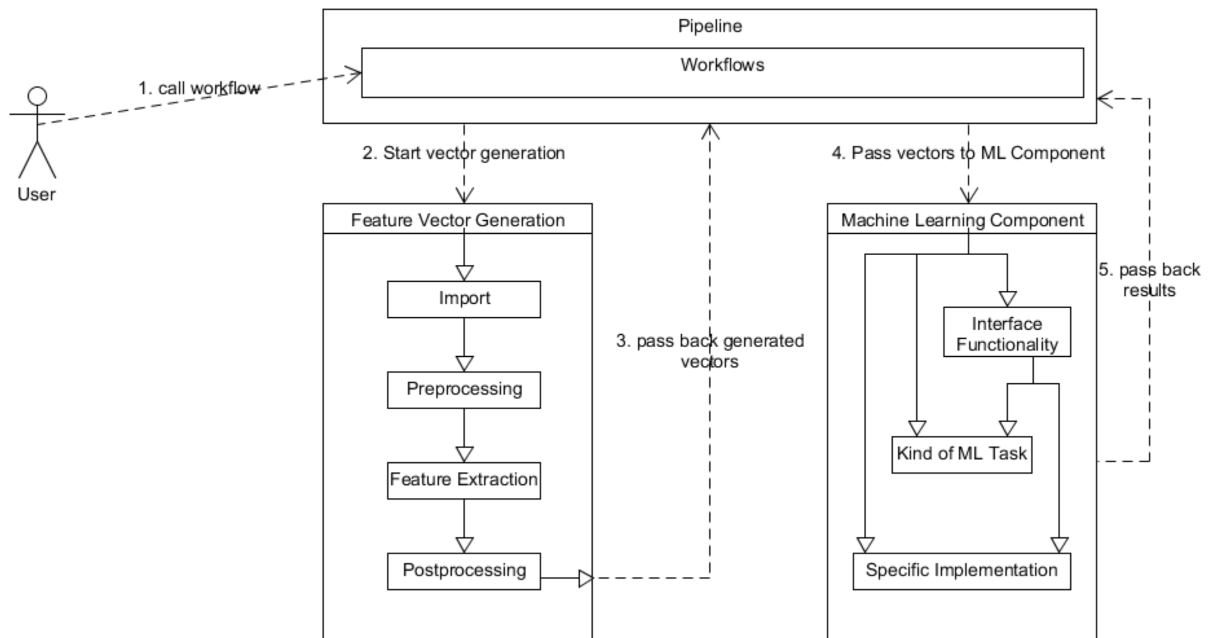

Figure 1 - Schematic of an abstract workflow

2.1.1 Feature Vector Generation

One of the primary challenges in machine learning is the generation of feature vectors. We decided to implement an extensible mechanic of successive stages to model the process of feature vector generation.
The depiction above shows data and call flow in an abstract workflow over the whole of PlebML's logical separation.
Feature vector generators are instantiated by a factories that extend an abstract class. These builders allow extensive configuration for each stage of the vector generation and system options such as parallelization.

- *Import*
  The Import stage retrieves raw data or raw training data from a specified location. This raw data can be anything but has to be consistent with what the preprocessor and the features in the later stages expect. The training data that an Importer can provide always consists of a raw data entry and an expected prediction result.

- *Preprocessing*
  The preprocessor expects raw data of the same type as the Importer imported. Its primary responsibility is to transform the raw data into a more meaningful format, clean it up and enrich it with additional data. For example in our text classification module we split the raw String into paragraphs, sentences and tokens, then reduce unneeded data (such as URLs or email addresses) and finally add data needed for feature extraction such as negation or lemmatization. The result of the preprocessor is a specific model object. In our text classification module this would be an object of the Document class.

- *Feature Extraction*
  The feature extraction receives model objects generated by the preprocessor and extracts features from it. It is configured by passing it a number of instances of classes that extend the IFeature interface and their respective configuration objects. How to exactly set this up and extend it will be covered in a later chapter. This stage will result in either a list of vectors or a list of pairs of vectors and expected prediction results, depending on whether training data was imported or regular raw data.

- *Postprocessing*
  The postprocessing stage serves to model processes like scaling or extracting statistical data from the generated vectors. Depending on what is done with the resulting vectors afterwards this stage is entirely optional and can be left out.

2.1.2 Machine Learning Component

The machine learning component is where the actual training and prediction happens. It can be anything from a wrapper for a library (like our implementation of Liblinear) to an entirely new self-written solver implementation or a new wrapper for another machine learning solution like Google Prediction API. It consists of three abstraction levels. This is meant to be able to construct different levels of generalizations of Pipelines.

- *1. Interface Functionality*
  This is the most abstract level of requirements a machine learning component has to satisfy. It consists of an interface with the method signatures of training, prediction, loading an already trained component and saving a trained component. It does not contain any default implementations.

- *2. Kind of machine learning Task*
  On this abstraction level different kinds of machine learning tasks can be differentiated. For instance classification-tasks or regression-tasks. This layer serves to make generalized implementations of certain algorithms (i.e. scoring for this sort of task) available that still can be overwritten in the actual specific implementation. All Classes on this layer should be abstract.

- *3. Specific Implementation*
  In this layer are the actual implementations of the interface methods and possibly overwritten methods of the abstract superclass.

In most cases an abstract class (for instance text classifier) implements the machine learning component interface and an actual implementation (in our case the wrapper for the Liblinear library) extends this abstract class. This would allow to write algorithms specific to classification or text classification on the abstract class and swap the actual machine learning component beneath while keeping the abstract class constant. One example for such algorithms could be scoring. This can be forgone in case of a highly specific machine learning component by only implementing the interface.

2.1.3 Pipelines as Workflow Collections

Pipelines are the primary mean to interact with the mechanics of PlebML. They are meant to offer a variety of workflows that span over the other two components of the framework. They can range from a highly abstract pipeline that requires a complete configuration to very specific pipeline that already contains preconfigured feature vector generation and a specific implementation of a machine learning component. Below are two examples of an abstract and a very specific pipeline.

- *Abstract Pipeline*
  An abstract pipeline offers only rudimentary functionality as it is only allowed to operate on the abstract layers of PlebML. Within the Machine learning component it only operates on the interface functionality, at most on the kind of machine learning task layer (it's possible to write a generic classification pipeline like that). As for the feature vector generation it should accept a configured generator in its constructor. As for a concrete example refer to the class ch.zhaw.init.plebml.coreLibrary.Pipeline.

- *Specific Pipeline*
  A specific Pipeline operates on a specific implementation of a machine learning component and on a specific builder. This allows for far more concrete workflows to be built on the pipeline but severely restricts its general applicability. For an example of a concrete pipeline refer to the class ch.zhaw.init.plebml.coreModules.textclassification.BasicTextclassificationpipeline.

Pipelines can be circumvented by the user. For instance if the feature vectors are already generated by another system they can still be used in PlebML however then only the machine learning components are useful. Or the other way around if the user wishes to use the feature vectors in another machine learning environment they can be exported. This means the machine learning components don't need to be instantiated. In such cases circumventing the pipelines makes sense as it would generate overhead with no gain as the workflows provided by it would include not needed components of PlebML.

2.2 Source Code Modularization

In this section we describe the different classes and interfaces and how they are separated from each other. This section is made up of three sub-sections discussing the three distinct parts of PlebML. Those three parts are the Core Library, Core Modules and Tasks. The Core Library is responsible for the machine learning algorithms, feature indexing, interfaces and abstract classes dictating the structure of the next part. The Core Modules are problem domain specific modules which contain things as the model object class representing the data within PlebML, the features which are extracted from that data as well as other components further discussed in the Core Module chapter. Tasks are responsible for System integration and interacting with the Core Modules.

2.2.1 Core Library

The core library constitutes functionality that is constant across any use-case for PlebML. It also structures any built core module (core modules will be shortly explained in a paragraph below) by imposing a set of restrictions with interfaces and abstract classes. Some best practices when building a core module cannot be enforced within the code itself. Those guidelines are described in the next chapter. The relations between the Core Modules and the Core Library are shown schematically in *Figure 2*. The relations between the Core Modules themselves will be explained in further detail in the Core Modules.

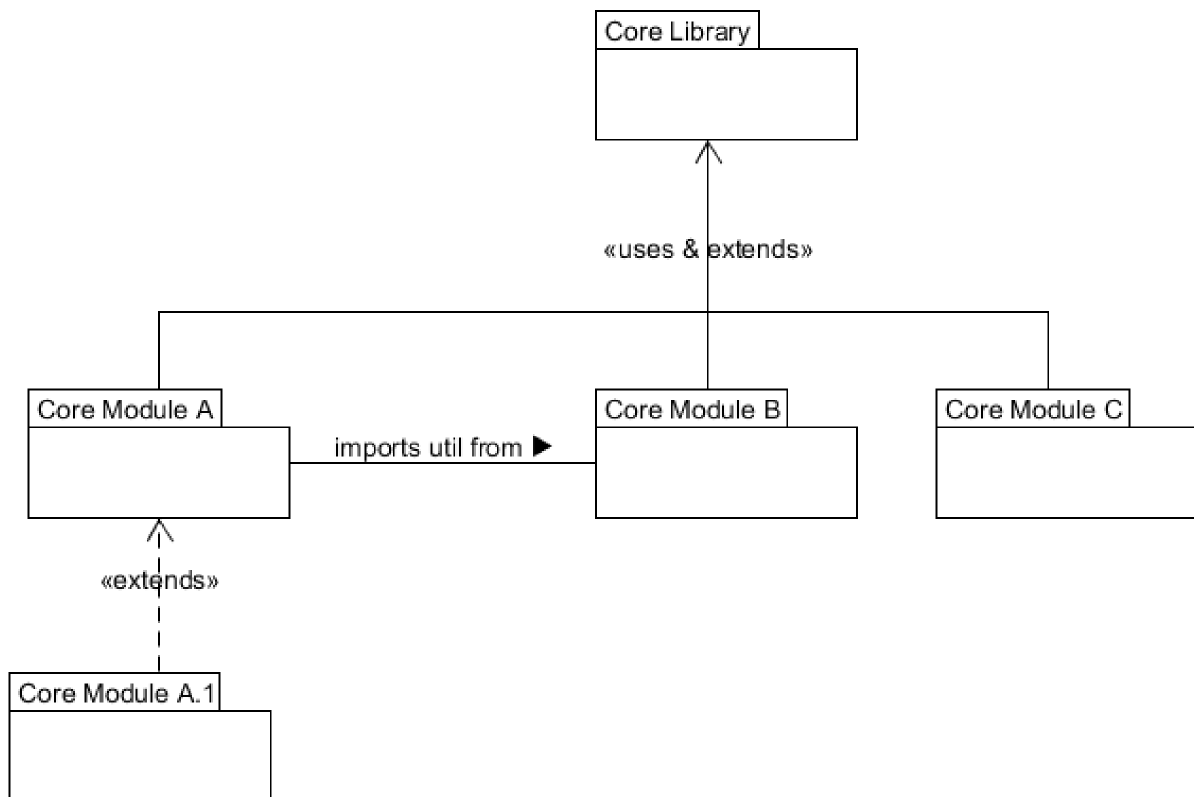

Figure 2 - Core Module Dependencies

2.2.1.1 A Short Note on Core Modules

As described the Core Library consist of abstract functionality and many things are only defined in from of interfaces, generics and abstract classes. For instance the data types of the raw input provided by the importer or the model object class are such things. Also the core library does not include any implementation of features. Core Modules are packages designed to handle a specific sort of machine learning task such as text classification and they contain all the actual implementations required by the core library, handle all the generics in the vector generation chain and offer sensible pipelines.

2.2.1.2 Feature Vector Generation

Abstract Builders

A builder is responsible for constructing a feature vector generator according to its settings. In the core library it only exists as an abstract class and it needs to be specified in each core module. This is because the builder also entails settings for the preprocessor which are specific to each core module. However having this abstract class ensures a consistent structure for builders in every core module.

As seen in Figure 3 on the next page the structure of the builder is similar to a folder structure. Each separate stage in the feature vector extraction process has its own object holding its settings. Each Class that extends the AdvancedSubOption Class has a make method which generates a part needed to build the feature vector generator. Also using a method on these Option Classes will return the same object again, allowing to chain the method calls. This results in very compact and readable code configuring the builder.

Also this graphic shows what is implemented on the abstract layer in the core library and what has to be extended in each core module.

Please note that *Figure 3* is not an exhaustive class diagram but should only show the structure imposed for the builders by the core library.

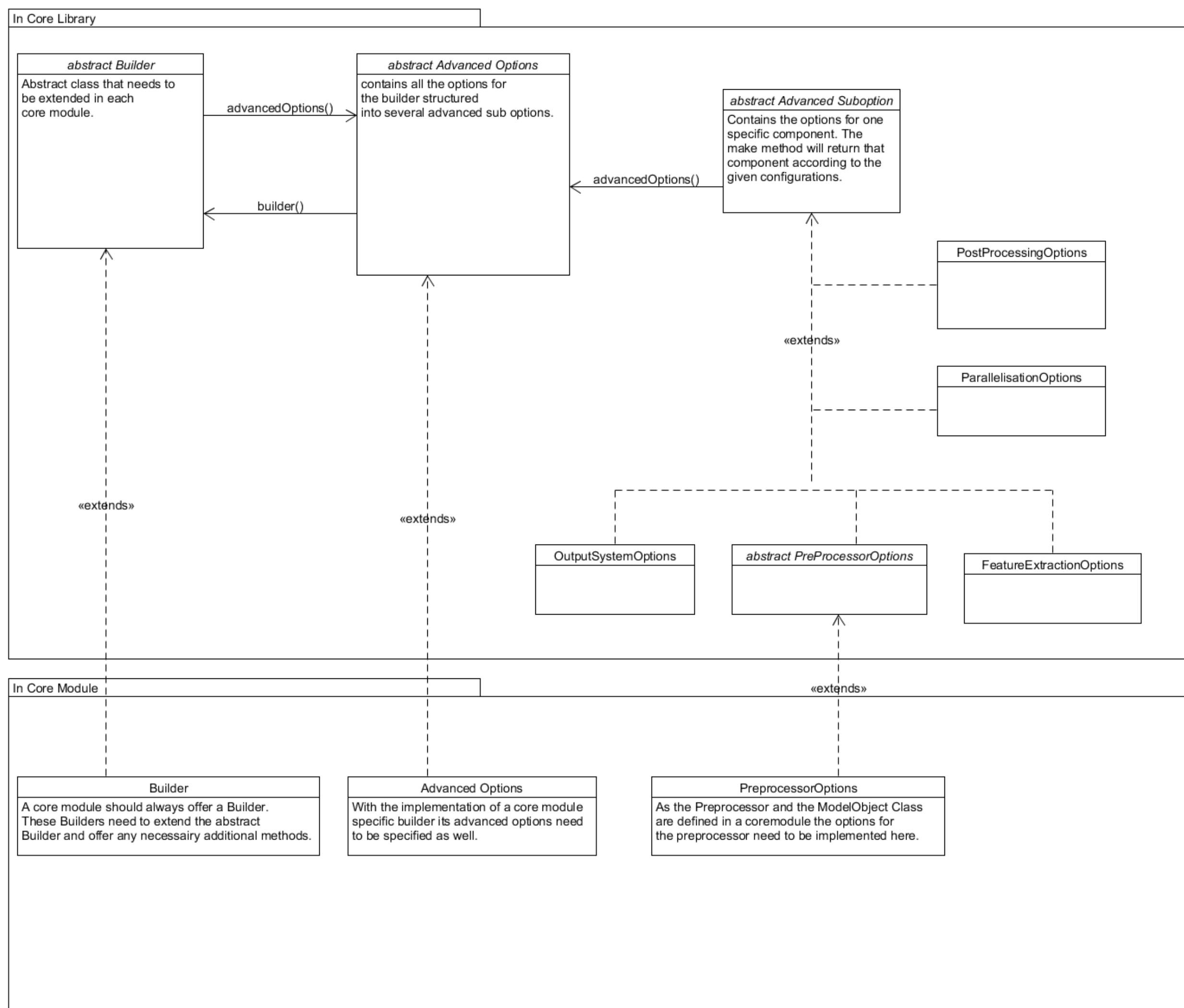

Figure 3 - Builder interaction

Importer Interfaces and Data Structures

Importers serve as interface between PlebML and physical storage, providing data needed for training and prediction alike.
The import package consists of the Importer interface and the TrainingData data structure. The Importer interface consists of two methods that have to be implemented:

```
Iterator<TrainingData<RawDataType, PredictionType>> importTrainingSetFromFile(String path);
Iterator<RawDataType> importDataToClassifyFromFile(String path);
```

There are also methods that support (String… paths) signatures. They are implemented per default using iterator concatenation. It is advised to overwrite them if there is a more performant way to handle these calls.

This Interface requires the first two generics of PlebML specifying the raw data type being imported and the intended result type of prediction.

The TrainingData Class is very simple and consists of an object of the raw data type and its expected prediction result.

Preprocessing Interfaces and Utilities

The preprocessor is responsible to convert the raw data provided by an importer to a meaningful object type on which the features can operate efficiently at a later stage. On an abstract level the preprocessor consist of the IPreprocessor interface and the IPreprocesorComponent interface. Furthermore the model object specified in the Core module should implement the IPreprocessorDataReciever interface but more to that in a later chapter.
While the IPreprocessor is responsible for the complete construction of a model object it is made up of separate steps called IPreprocesorComponent. These components should structure the raw data in a meaningful way and write pertinent metadata on the resulting object. The preprocessing step finally results in a new object of the ModelObjectType which is a generic, guaranteeing type consistency between preprocessor and used features.

The IPreprocessor interface consists of the following methods:

```
public ModelObjectType preprocess(RawDataType data);
public void loadComponents(Class<? extends IPreprocessorComponent>... components);
public void loadComponent(IPreprocessorComponent component, String identifier);
default public void unInitialize(){}
```

The preprocess method is the main part of this Interface. It should invoke all components that got loaded by dependency crawling done by the preprocessor options on the builder or that were added by the user.
As the preprocessor is implemented in the core module the next chapter will deal with best practice and design recommendations for preprocessor and model object classes.

The loadComponent methods are needed by the dependency crawling and should always be supported in any preprocessor. If that is not possible a runtime exception is to be thrown, informing the user to disable the dependency crawling in the builder settings.

The uninitialized method is implemented per default to do nothing. It is called once the preprocessing stage is complete and should clean up any physical resources required by the preprocessor.

The IPreprocessorComponent interface consists of these methods:
```
public void setIdentifier(String identifier);
public String getIdentifier();

default boolean isGlobal()
{
    return this.getIdentifier()==null;
}
```

There are two different kind of preprocessor components. There are global components, of there only one per class exists and their results are shared between all feature implementations. The data provided by these components can be retrieved by class. The other kind of component is the named one, the data of these components can be retrieved with their identifier. Most components should be global but there are cases where different feature implementation require the same class of preprocessor component to have run but with different settings. This mechanism allows to account for that use case.

The IPreprocessorDataReciever is the interface intended for the model object class. It allows for abstract interaction between the preprocessor and model object. It consists of these methods:
```
public void acceptDataFromGlobalComponent
        (Class<? extends IPreprocessorComponent> component, Object data);

public void acceptDataFromNamedComponent
        (IPreprocessorComponent component, String identifier, Object data);

public <V> V getDataFromGlobalComponent
        (Class<? extends IPreprocessorComponent<V>> component);

public <V> V getDataFromNamedComponent(String identifier);
```

These methods allow preprocessor components to write data on these objects. Features can then later retrieve the necessary data from the object.

The preprocessor core package also contains the class PreprocessorComponentSuperclass which offers some method implementations that are helpful. However using this superclass means that class inheritance is used instead of interface implementation.

The most important feature on the PreprocessorComponentSuperclass is the following method:
```
protected void setDataOnReceiver(IPreprocessorDataReceiver receiver, Object data)
```

This method automatically handles the distinction of global and named components when setting data on an IPreprocessorDataReceiver.

FeatureVectorGenerator

The FeatureVectorGenerator handles the dataflow during feature extraction, the imported data is lazily transformed by the preprocessor. Subsequently all features are applied on the preprocessed data. The extraction is done by the FeatureExtractor and afterwards the generated feature vectors are postprocessed. The parallelism during the extraction is configurable via the ParallelisationStrategy and the memory management through the MemoryStrategy. The different stages are described in the following chapters.

Public Methods

The FeatureVectorGenerator offers several public methods that allow to extract features for different purposes as shown in *Table 1*.

	Store new indices	*Discard new indices*
Labelled data	Supervised Training generateLabelledFeatureVectors()	Evaluation generateLabelledFeatureVectorsDiscardingNewIndices()
Unlabelled data	Unsupervised Training Not implemented	Prediction generateFeatureVectorsDiscardingNewIndices()

Table 1 - FeatureVectorGenerator method overview

When generating labelled feature vectors the FeatureVectorGenerator automatically removes duplicated vectors with the same or different labels.

Parallelisation Strategies

There are three strategies for parallelisation, so the resource consumption during the feature extraction can be adapted to the available hardware and other resource consuming tasks within the same JVM. The FeatureVectorGenerator is responsible to use the FeatureExtractor accordingly. The different strategies are described below, for the readability the names are shortened, SER stands for SERIAL, PAR for PARALLEL and FET for FEATURE.

SER_DATA_SER_FET: Data models are processed serially and the features are applied serially as well.

SER_DATA_PAR_FET: Data models are processed serially and the features are applied in a parallel manner.

PAR_DATA_PAR_FET: Data models are processed in parallel, features implementing the interface ConcurrentHeavyLoadFeature or ConcurrentMidLoadFeature are applied on several documents at once, and all other features are still applied serially.

FeatureExtractor

The FeatureExtractor extracts features for preprocessed documents, it offers several public methods for different levels of parallelisation as shown in *Table 2* and the functionality of the methods is described in the following sub chapters.

	Parallel Data	*Serial Data*
Parallel Features	extractFeatureVectorConcurrent()	extractFeatureVectorParallel()
Serial Features		extractFeatureVectorSerial()

Table 2 - FeatureExtractor method overview

extractFeatureVectorSerial

All features are applied serially.

extractFeatureVectorParallel

All features are applied in parallel using the parallel stream of Java 8.

extractFeatureVectorConcurrent

Before the first use of this function initializeConcurrentThreadPool() has to be called to initialize the thread pool that will extract the features concurrently and after the last call to extractFeatureVectorConcurrent() done() has to be called to shut down the thread pool.

For the concurrent extraction of features a queue of data queues for each feature is maintained as illustrated in *Figure 4*. Simultaneously a map from data models to countdown latches is maintained.
The worker threads in the thread pool take one data queue from the queue and apply the according feature to all data models in the queue serially, counting down the countdown latches of the according data models. Once the queue is empty it is put back on the queue of queues and the next one is polled.

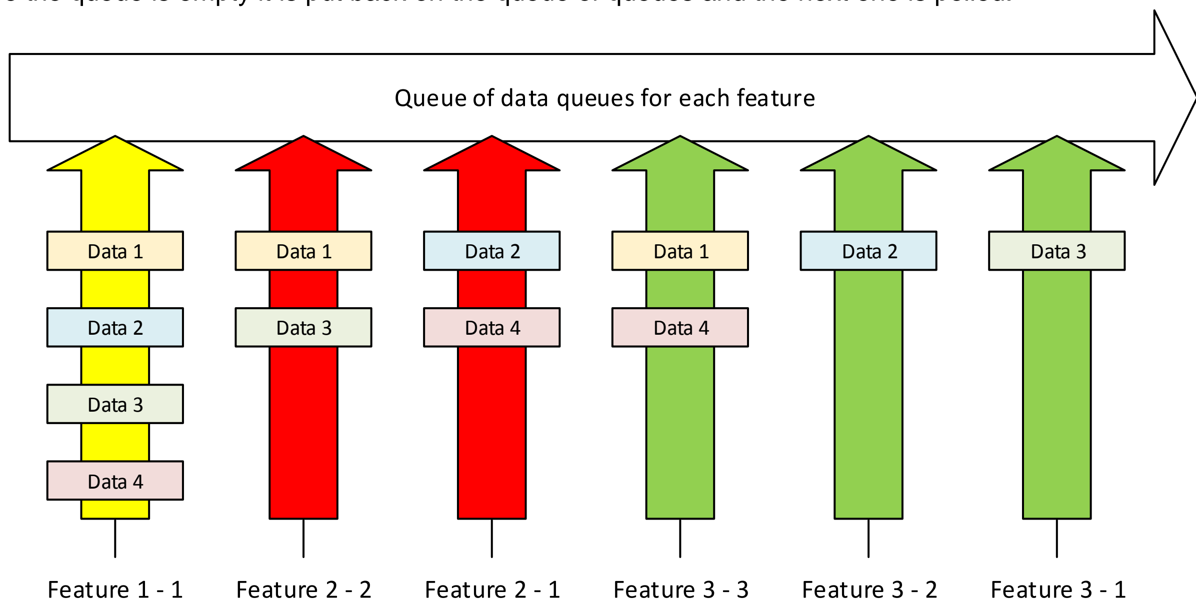

Figure 4 - Visualization of queues for concurrent feature extraction

In *Figure 4* an example for a queue of data queues is shown, where a total of three features and four data models are ready to be processed, Feature 2 implements the interface ConcurrentMidLoadFeature and Feature 3 implements the interface ConcurrentHeavyLoadFeature.

A call to extractFeatureVectorConcurrent() first adds a mapping from the data model to be processed to a new countdown latch with the amount of features, in the example above this would be three, to the map of countdown latches. Next the data model is added to a queue of each feature, for features which have multiple queues the data models are distributed equally between all queues. Finally the caller waits for the countdown latch to reach zero.

FeatureExtractorDiscardingNewIndices

The FeatureExtractorDiscardingNewIndices extends. The FeatureExtractor the difference in their behaviour is that the FeatureExtractorDiscardingNewIndices uses a ConcurrentForgettingVectorSpace that discards any unknown indices.

VectorSpace

The VectorSpace is responsible for the distribution of indices for the features and has to be capable of persisting the mappings of features to indices. Each mapping consists of two Strings for the key and an Integer for the value. The first part of the key (key1) is the full class name of the feature, the second one (key2) is an identifier provided by the feature. A part of the classes in the package vectorspace is shown in *Figure 5*, please note that this figure is for simplicity not complete. The two implementations of ConcurrentVectorSpace are ConcurrentPersistentVectorSpace which generates new indices and persists them and ConcurrentForgettingVectorspace which discards all unknown features.

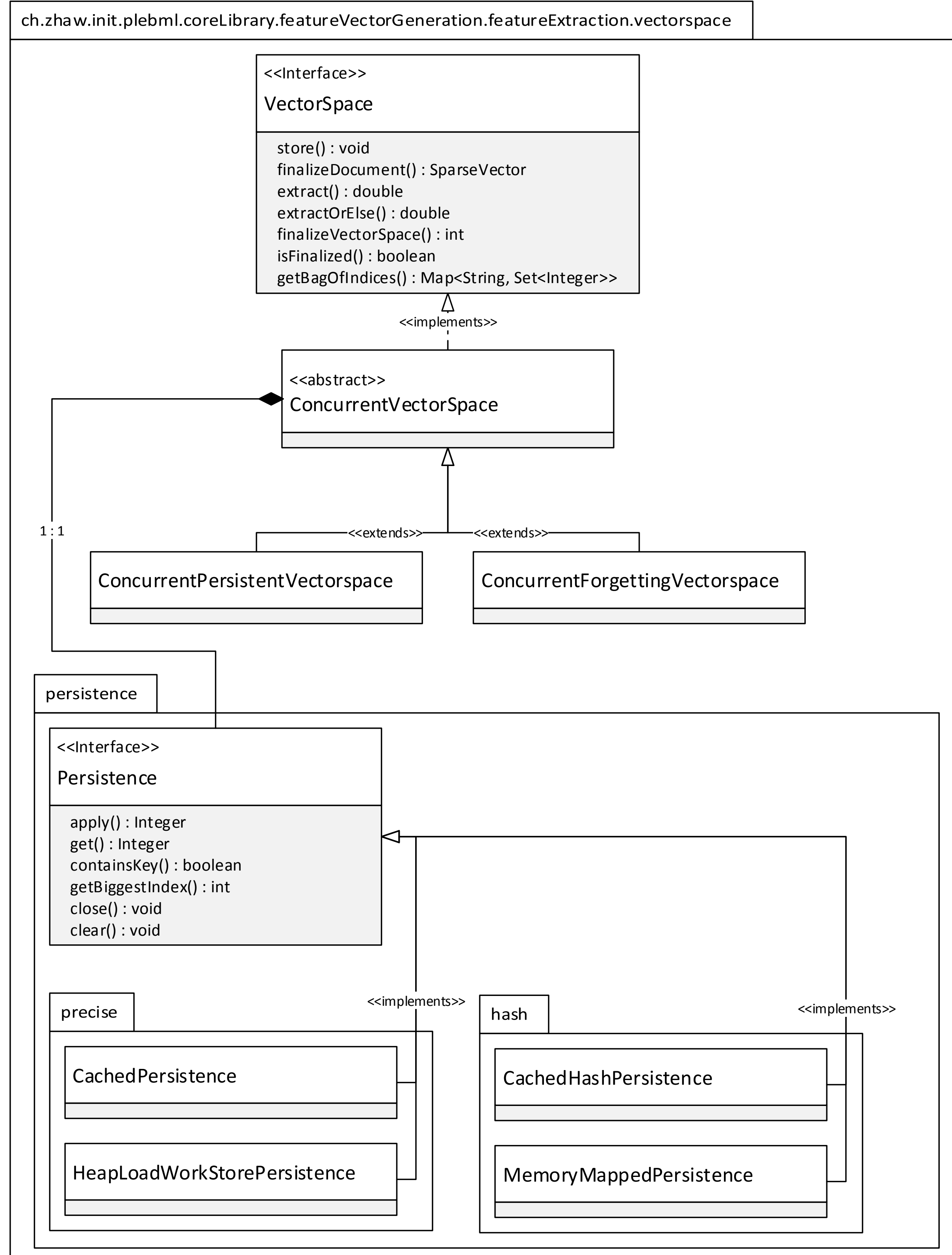

Figure 5 - Incomplete UML of ch.zhaw.init.plebml.vectorspace

Memory Strategies

The memory strategy determines the persistence used by the VectorSpace. A persistence is responsible of persisting and generating key – index bindings for given keys, all strategies use the documented algorithm for String.hashCode() in Java 8 and a hardcoded algorithm for the hash code of a pair of Strings, which guarantees to get the same hash for the same string and string pair on all JVMs. The available strategies are described below. The two cached variants are based on the H2 database engines, accessed by hikari connection pools and wrapped by a cache implementation from the guava library.

Normal Indexing

This implementations store the full key pair along with the according index.

HEAP: The VectorSpace is fully kept on heap as a two level map, to persist the VectorSpace a persistent map implementation of MapDB is used. This is the fastest strategy, but if not enough ram is available to keep the full map on the heap, this is not feasible. This behaviour is implemented by HeapLoadWorkStorePersistence.

CACHED: To eliminate redundancy in the persisted data, two tables are used, table1 for key1 along with a unique id for this part of the key and table2 containing the unique id for key1, key2 and the mapped index. For index only lookups an index over all values of table2 is maintained (key1_id, key2, index).
To improve the concurrent performance and to reduce the used disk space several additional actions haven been performed:

- The data is distributed over 64 databases with 1024 tables each, this greatly reduces the size of the table indices and improves concurrency as multiple threads can work on different tables. To distribute the entries among the different databases and tables, the hash of both key parts is used.
- Locks on database level are disabled and replaced by in memory locks per table.

Hashing Trick

The “hashing trick” is described in many papers for example in a work by Joshua Attenberg [17], in PlebML the hashing trick is used in two implementations. These only store the hash of the key pair along with the according index. In the current implementations the amount of possible hashes is reduced by a modulo operation to 2^{28}. This greatly reduces the used disk space, but may result in the same index for different features when their hashes collide.

CACHED_HASH: The data is stored in a table containing the hashes and the according indices. The table has an index over both columns. As in the CACHED implementation the data is distributed over several databases and tables to reduce the used disk space and to improve concurrent access, but as the mappings of hashes to indices need far less space, the amount of databases and tables is reduced to 32 databases and 128 tables per database.

MEMORY_MAPPED_HASH: The VectorSpace consist solely of a flat dense array of indices saved in a file and mapped to memory, the size of the array is given by the number of possible hashes. The mapping to memory is done with a combination of a RandomAccessFile and several MappedByteBuffers. The size of a MappedByteBuffer is limited to Integer.MAX_VALUE bytes, to circumvent this limitation and to improve concurrent access 64 buffers are used.
This strategy has a constant access time for all keys, for small problems the drawback is the huge dense array resulting in increased disk consumption.

Postprocessing

Postprocessing is applied on all feature vectors once all features are extracted in the training stage, afterwards it can be applied on single vectors independently. The PostProcessor itself is a wrapper for a bunch of PostProcessingFunctions. As of now only scaling functions are implemented as shown in *Figure 6*.

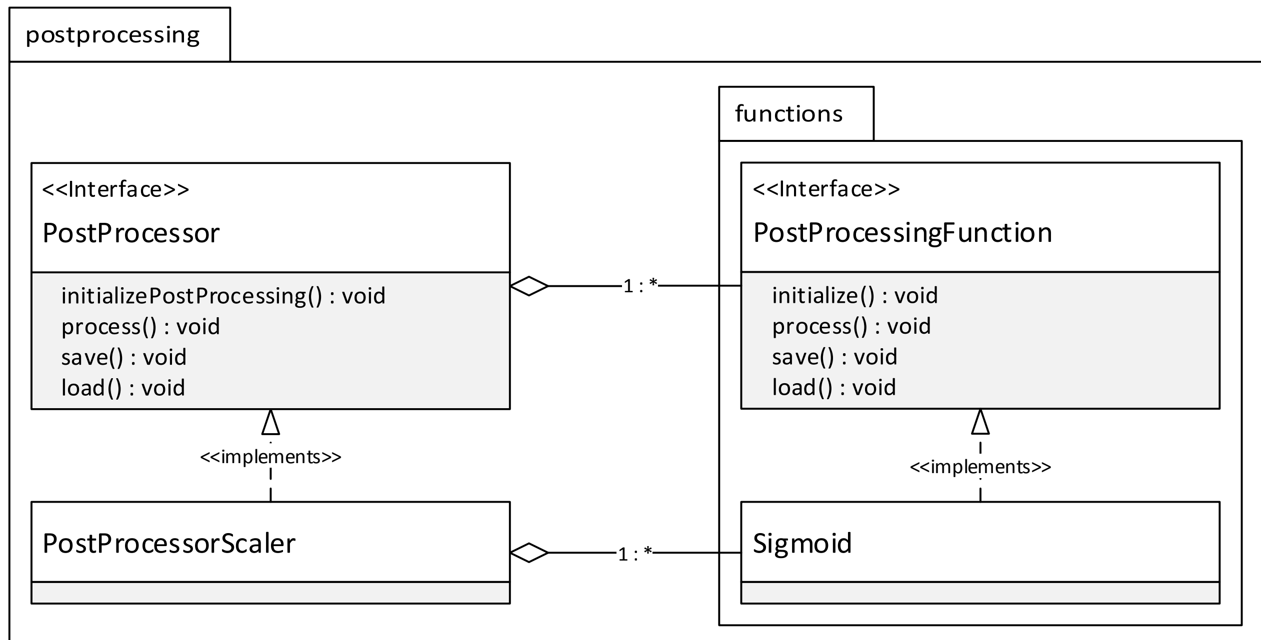

Figure 6 - UML of ch.zhaw.init.plebml.coreLibrary.postprocessing

PostProcessorScaler

The responsibility of the PostProcessorScaler is to handle a bunch of PostProcessingFunctions and to delegate the method calls accordingly. The PostProcessorScaler takes a map from feature classes to PostProcessingFunctions as an argument in the constructor, this mapping means that for a specific feature a specific function has to be applied. Each feature class can therefore have at most one function within a PostProcessorScaler. When the postprocessing gets initialized with initializePostProcessing() a map from feature classes to a set with the indices of that feature and a collection of feature vectors, the PostProcessorScaler simply calls initialize on all PostProcessingFunctions with the according set of indices and the feature vectors.

PostProcessingFunction

A PostProcessingFunction is responsible for a specific postprocessing task for example a sigmoid scaling. Before the first call to postprocess, the PostProcessor has to be initialized with a set of indices and a collection of feature vectors. PostProcessingFunction is responsible to save all data needed to process further feature vectors with a call to save.

Sigmoid

Sigmoid is a simple implementation of a PostProcessingFunction that applies a sigmoid scaling. As sigmoid scaling can be done on all values independently the sigmoid function only needs to store the set of indices it has to scale.

2.2.1.3 Machine Learning

For the machine learning part there is a modified and wrapped version of the Liblinear port of Benedikt Waldvogel [18]. Several adaptions and optimizations were performed on the original library to greatly reduce memory consumption and to improve its architecture. Details about the modifications and wrappers follow next.

Problem Representation

Several flaws concerning the representation of a problem have been spotted and fixed. The changes made to the original library are described in the following chapters.

Problem Interface

The original implementation of the problem representation violated the clean code principles by granting public access to all its members. This made it impossible to easily use different implementations of problem representations. To solve this issue an interface for the representation has been introduced, which now offers public getter methods to access the data and all methods of Liblinear were adapted to the new interface, this allows for a far more flexible design of the problem implementation.

Problem Memory Consumption

The original problem representation used a two dimensional array of feature objects holding the index and the according value of this feature. For large problems this leads to a significant memory overhead due to the additional twelve bytes for each feature object.
To eliminate this overhead a new implementation of the problem interface has been introduced. The original representation was modified to hold two two-dimensional arrays of primitives for the indices and values of the features. The link from index to value which was provided by the feature object is now given by the position in the arrays, so that the index[i][j] corresponds to the value[i][j].

To illustrate the reduction in memory consumption we calculate the memory consumption of an example problem with 20'000 observation points, 5000 features per observation. Please note that only parts that were changed are calculated. The calculations are shown in *Table 3* and *Table 4*.

The previous representation needed roughly:

Description	*Size computation*	*Result*
Array of observation point arrays	$1 * 12B + 4B = 16B$	16B
Array per observation point	$20'000 * 12B \cong 234KB$	234KB
Feature objects	$5'000 * 20'000 * (12 + 8 + 4)B \cong 2.2GB$	2.2GB
Total		2.2GB

Table 3 - Memory calculation for the original problem

The representation of the same problem with the new implementation needs roughly:

Description	*Size computation*	*Result*
Array of index arrays	$1 * 12B + 4B = 16B$	16B
Array of value arrays	$1 * 12B + 4B = 16B$	16B
Index array per observation point	$20'000 * 12B \cong 234KB$	234KB
Value array per observation point	$20'000 * 12B \cong 234KB$	234KB
indices	$5'000 * 20'000 * 4B \cong 381MB$	318MB
values	$5'000 * 20'000 * 8B \cong 763MB$	763MB
Total		1.1GB

Table 4 - Memory calculation for the adapted problem

So the effective memory consumption for the problem representation is roughly halved. This greatly improves the computation performance as well, as far less data has to be transferred between main memory and the processor.

SubProblem

To handle permutations to the original problem during the solve method, the SubProblem was introduced. It holds a reference to the original problem and applies the permutations lazily.

Multithreading

The original implementation offered no help concerning out of memory errors while solving problems. This is especially important when multiple threads solve problems at the same time. For this purpose the HeapGuard has been implemented. Its responsibility is to stall threads that would require too much heap space to solve their problem and might result in an OutOfMemoryError (OOME). Further OOMEs during training, which may still appear due to heap fragmentation, are handled and the thread will automatically try to solve the problem again, once more heap is available. With the HeapGuard it is possible to work with as many threads as desired without the risk of an unhandled OOME that might harm the correctness of the execution.

Listeners

Two listeners have been introduced to grant the user some information about the progress during cross validation and about the actions of the HeapGuard.

Liblinear Wrapper

To conform to the MachineLearningComponent interface that is used within PlebML and to handle changes to the library interface in a single place the adapted Liblinear library is wrapped. In addition of handling the passing of method calls, the wrapper handles the listeners of Liblinear and converts their events to events on the event bus.

Liblinear Factory

The LiblinearMLComponentFactory offers several methods to easily find the best classifier for a given problem. This includes methods to find the best c for a given solver type and methods to find the best solver type and c.

OptimizeOverC

To find the best c for a given solver type within the provided c range the problem is solved and evaluated for c values in a logarithmic scale with the provided granularity as suggested in [19]. The method is overloaded to use cross validation or to evaluate against a static set and to use defaults for several parameters.

OptimizeOverSolvers

For this purpose a set of recommended c range and step size for each solver type were found by experiments on various datasets. To find the best solver over all solvers, every solver is tested with the method optimizeOverC and the recommended parameters. This allows users with no knowledge about the different solver types to find the best solver for their problem.

2.2.1.4 Event Bus System

In PlebML all events are distributed over an event bus with a publish-subscribe pattern. Additionally to the EventBus there is an EventBusFactory to maintain a singleton per identifier. An incomplete UML figure of the event bus system is shown in *Figure 7*.

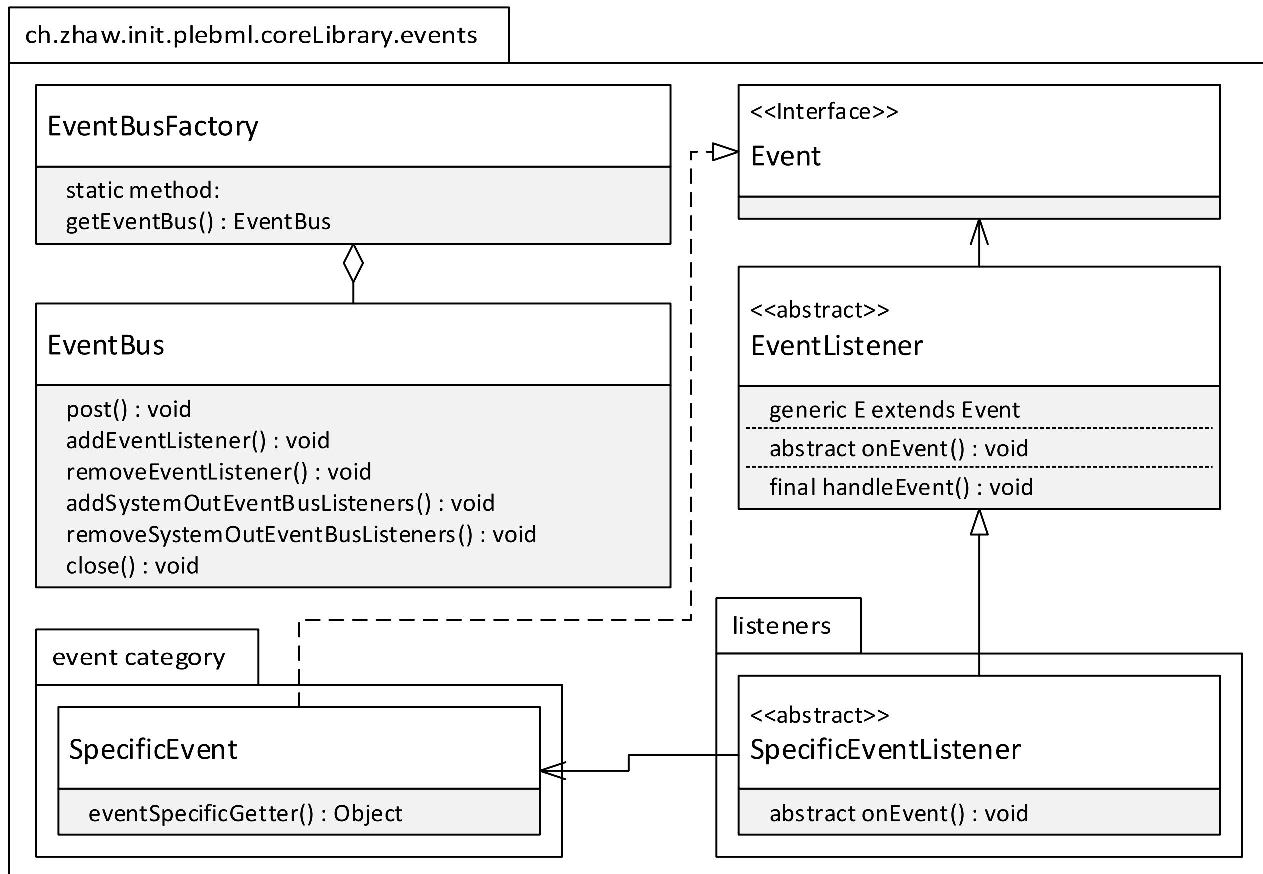

Figure 7 - Incomplete UML figure of the event bus system

Pros

There are several benefits of using an event bus instead of the traditional listener pattern.
First of all there are no listeners registered and wrapped along all levels of abstractions, which reduces the amount of code to write and maintain tremendously. For example to raise an event one can simply call post on the event bus with the according event as parameter, instead of writing fire methods on all levels of abstraction.
Further there exists no hard wiring between the consumers and the producers of events, this allows a far more flexible software design and easier refactoring as far less dependencies have to be maintained.

Con

Many Java developers might be unfamiliar with the usage of a publish-subscribe pattern.

Solutions

To hide the publish-subscribe pattern from the developers, for each event an abstract EventListener is implemented so the developers only need to implement an onEvent() method of the according EventListener.

2.2.2 Core Modules

Core Modules are packages designed to solve a particular kind of machine learning problem. As an example we implemented a text classification core module. Core modules deal with the feature vector extraction side of PlebML described above.
Each module has to implement a series of components to be functional and should satisfy a set of best practice guidelines not enforced within the source code.
In the next sections we describe what each core module has to contain. *Figure 8* shows what sort of dependencies between core modules are intended within PlebML's architecture.

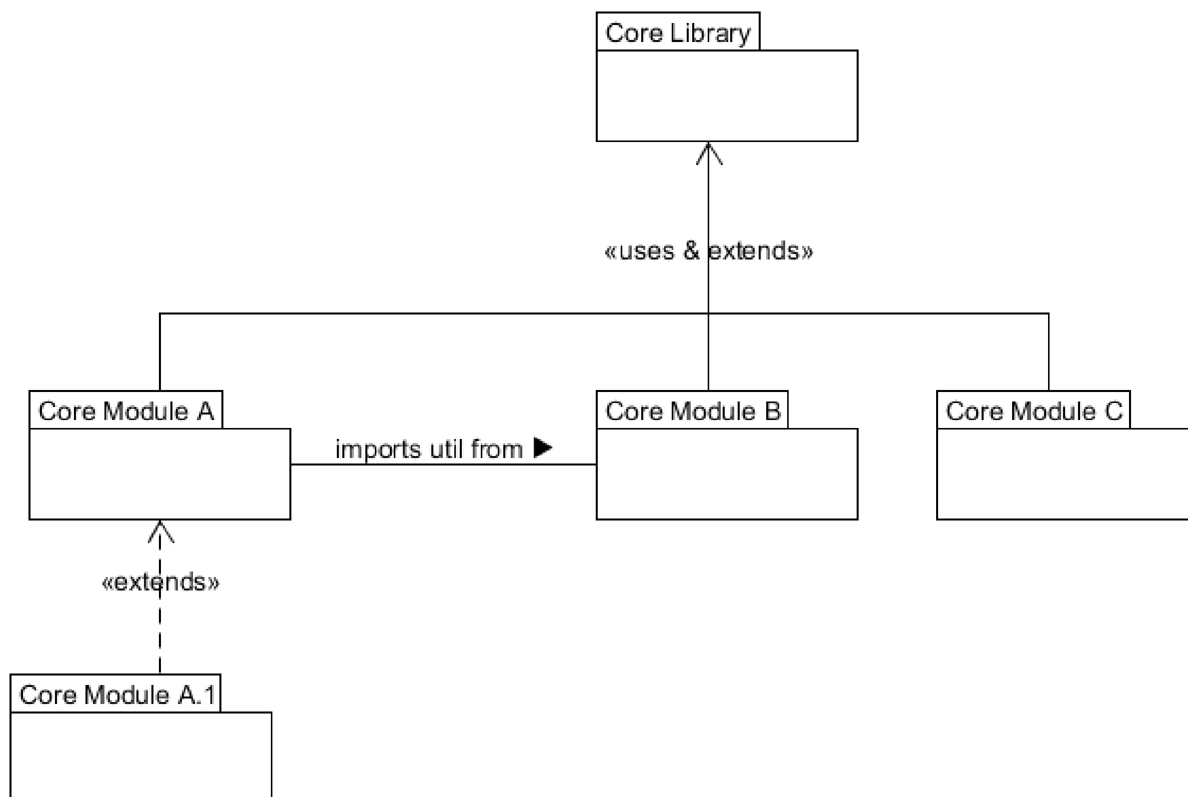

Figure 8 - Core Module Dependencies

As shown in *Figure 8*, each and every core module makes extensive use of the core library as it has to implement many interfaces and abstract classes.

Core modules might deal with similar problems but with very different data structures, in such cases it might make sense to share utility functions. This however should not be done without deliberation as it strongly links those two core modules together. Copying or re implementing the utility functions might sometimes be a safer way, even though it would result in a DRY violation.

Another feasible case is that a core module is implemented on a very general level, such as our text classification module. If the problem that is to be solved is a text classification but a very specific kind, parts of the core module can be extended. This would particularly concern the model object class and the preprocessor components.

2.2.2.1 Model

The model represents the output of the preprocessor as well as what the feature implementations get passed. As features might be written by many different people, clean structure, simplicity and efficiency should be emphasized when designing a model. If for instance the raw data type would be an HTML-String one possibility would be to construct a DOM-Tree from it.
Also classes that contain metadata written by preprocessor components should implement the IPreprocessorDataReciever interface to allow interaction on an abstract level.
Designing the model and deciding on how to represent the data to the features is the first step of designing a core module.

2.2.2.2 Preprocessor Implementation

The layer of interfaces and abstract classes of the preprocessor already implicitly proposes a structure like shown in *Figure 9*:

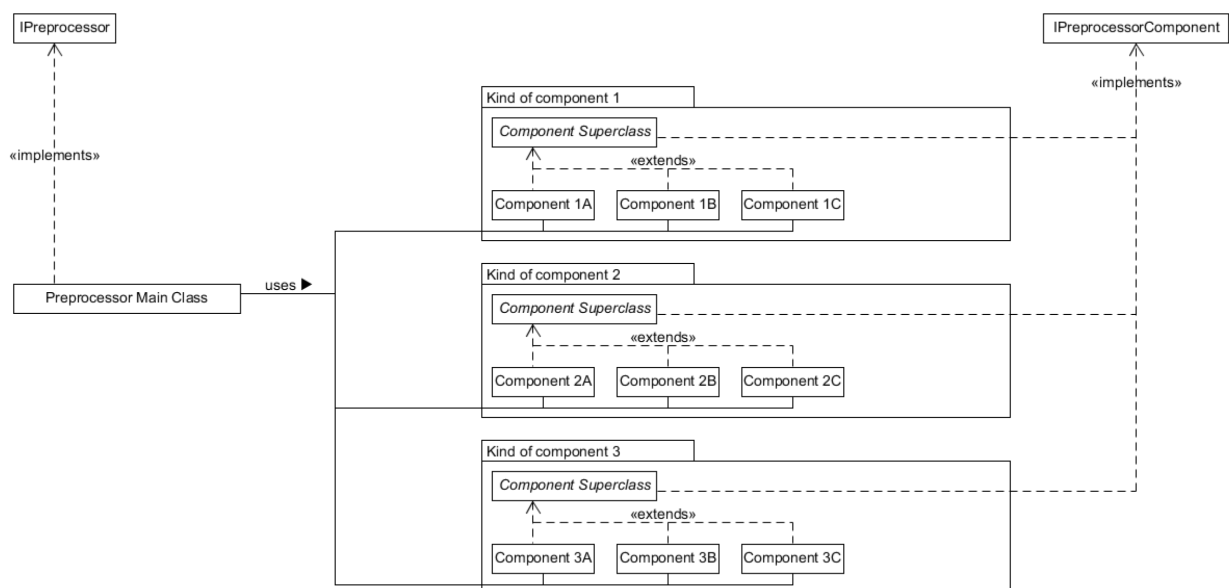

Figure 9 - Preprocessor structuring

Structuring the preprocessor into several stages or groups of components often makes sense. In many cases the raw input passed by an importer needs to be structured, sanitized and enriched with additional meta-data. Of course depending on the input these categories are not applicable, or many more might be of importance. However the preprocessor should specify what kind of components there are and in which chronological order they are executed.

Since one category can have many components, it is important to think about how they influence each other and whether the order in which they are added to the preprocessor makes any difference.

The preprocess method on the preprocessor main class is responsible for invoking all components in the correct order and returning a preprocessed model object at the end.

When designing the preprocessor in conjunction with the model object there are some important considerations:

- What additional data will the feature extraction need?
- How are multiple components within the same group handled?
- Do I retain the original raw input data on the model object?
- How is the preprocessor configured?

2.2.2.3 Feature Implementations

The architecture of the Feature itself is trivial as it is only a simple Interface. To understand its interaction with the rest of PlebML refer to the chapter “Feature extractor”. In this section we describe the structure of a feature implementation and a few pointers on how to implement it.
The first thing that should be done when starting the feature implementation in a core module is to extend the IFeature interface to put the required model object generic in place. All feature implementations in the core module must then implement this new interface.

```
public void extract(
        ModelObjectType modelObj,
        FeatureVector<ModelObjectType> vectorToWriteTo,
        List<ConfigType> configs
        );
```

This is the main method of concern within each feature implementation class. The passed ModelObject is generated by the preprocessor of the same core module and should contain all pertinent information for this feature.
The best practice to ensure all related preprocessor components were invoked is to use the @RequiresComponentClass annotation. This annotation allows to model dependencies between features and the preprocessing stage.

```
@RequiresComponentClass(componentClass = TweetNLPPOSTagger.class)
public class NumberOfPOSTags implements ISentimentAnalysisDefaultVectorFormat<Void> {}
```

This is an example from our text classification Core module showing the class header of a feature implementation. As mentioned before the IFeature interface was extended, now only requiring the class generic for the passed configuration objects. Also the usage of the @RequiresComponentClass annotation is shown. The componentClass parameter in this annotation has to be a class that implements the IPreprocessorComponent interface.

In the extraction method, the second parameter is where all the feature vector entries this class calculates are to be put. The FeatureVector class behaves very much like any map in Java. This is why collisions between keys should be avoided within one feature class.

The configuration list is initially given to the builder when adding this feature. It is then received by the feature when its extract method is executed. Best practice is to create a separate method that extracts the feature values based on one configuration instance and then loop through the whole list and call this method for each configuration instance.

In general developers are encourage to write thread save extract methods when they implement new features to allow concurrent execution. Features whose extract method is thread safe should implement one of the concurrent marker interfaces ConcurrentMidLoadFeature and ConcurrentHeavyLoadFeature. Best practice is to implement ConcurrentMidLoadFeature only for features who need far less processing time than ConcurrentHeavyLoadFeature as a wrong placed ConcurrentMidLoadFeature can lead to performance impacts.

Features should be as independent as possible allowing many software engineers to contribute their own features to the core module. This is also why the interface is rather lightweight. To find out which constellations of features proof especially effective for which sort of tasks, extensive experimentation has to be done. At the moment PlebML does not offer automatic mechanisms for such experiments, however they can be implemented on a Pipeline by the core module developers themselves.

2.2.2.4 PostProcessor Implementation

The implementation of the postprocessor is responsible to delegate method calls to all its functions. Therefore its structure is quite simple.

2.2.2.5 PostProcessorFunction Implementation

A postprocessing function implements the PostProcessingFunction which has the following methods:

```
void initialize(Iterable<SparseVector> vectors, Set<Integer> bagOfIndices)
```

This method gets called once for each postprocessing function when all feature vectors for training are extracted. Its purpose is to allow functions to gather information about the data, for example the minimum and maximum value of a feature. The first parameter contains an Iterable over all feature vectors. A function is responsible for all features with an index contained in the second parameter. The bag of indices has to be persisted by the function. After a call to this method the function has to be capable of processing feature vectors independently.

```
void process(SparseVector vector)
```

After the function has been initialized with a call to initialize(...) or load(...) it has to be capable of handling calls to process. Within this method it has to apply the transformations on the vector in place. It is important that the method only reads and writes values with the indices it is responsible for.

```
void save(String workingDir)
```

With a call to this method all values needed to process feature vectors have to be persisted. The function is responsible to create a directory named with the full class name of the function within the directory referenced by workingDir and to persist all data needed within this directory.

```
void load(String workingDir)
```

With a call to this method all values needed to process feature vectors have to be loaded from the directory referenced by workingDir.

2.2.2.6 Builder Implementation

This is what the usage of a fully implemented Builder looks like. Note the usage of option chaining mentioned earlier in this work. It resembles a folder structure the indentation indicating the navigation.

```
TextClassificationBuilder builder = new TextClassificationBuilder();
builder
    .advancedOptions()
        .preprocessor()
            .addTokenizer(new SuperSimpleTokenizer())
            .addMutation(new SimpleUserNameNormalizer())
            .addMutation(new SimpleURLNormalizer())
            .constructPreProcessorFromFeatureRequirements()
    .advancedOptions()
        .featureExtraction()
            .addFeature(
                        new NGramFeature(),
                        new NGramConfigWrapper(1, Sets.newHashSet()),
                        new NGramConfigWrapper(2, Sets.newHashSet()),
                        new NGramConfigWrapper(3, Sets.newHashSet()),
                        new NGramConfigWrapper(4, Sets.newHashSet()),
                        new NGramConfigWrapper(1, Sets.newHashSet(), ExtractorModes.LEMMA),
                        new NGramConfigWrapper(2, Sets.newHashSet(), ExtractorModes.LEMMA),
                        new NGramConfigWrapper(3, Sets.newHashSet(), ExtractorModes.LEMMA),
                        new NGramConfigWrapper(4, Sets.newHashSet(), ExtractorModes.LEMMA)
                    )
            .addFeatureWithRecommendedConfigs(new NonContiguousNGram())
            .addFeatureWithRecommendedConfigs(new NonContiguousNGramWithPOSTagAsWildcard())
            .addFeature(new NrOfHashtags())
            .addFeature(new NrOfAllCapsToken())
            .addFeature(new NumberOfPOSTags())
            .addFeature(new GloveFeatures())
            .addFeature(new NrOfNegatedContexts())
            .addFeature(new NrOfElongatedWords())
            .addFeatureWithRecommendedConfigs(new LastTokenContainsPunctuation())
            .addFeatureWithRecommendedConfigs(new ContinuousPunctuation())
            .addFeature(new CMUTweetClusterFeature())
            .addFeature(new ScoreTotal())
            .addFeature(new ScorePos())
            .addFeature(new ScoreNeg())
            .addFeature(new LastTokenScore())
            .addFeature(new LastTokenNegScore())
            .addFeature(new LastTokenPosScore())
            .useMemoryStrategy(MemoryStrategy.HEAP)
    .advancedOptions()
        .parallelisation()
            .useParallelisationStrategy(ParallelisationStrategy.SERIAL_DATA_SERIAL_FEATURE)
    .advancedOptions()
        .postProcessor()
            .putPostProcessingStepForClass(ScoreTotal.class, new Sigmoid())
            .putPostProcessingStepForClass(ScorePos.class, new Sigmoid())
            .putPostProcessingStepForClass(ScoreNeg.class, new Sigmoid())
            .putPostProcessingStepForClass(LastTokenScore.class, new Sigmoid())
            .putPostProcessingStepForClass(LastTokenNegScore.class, new Sigmoid())
            .putPostProcessingStepForClass(LastTokenPosScore.class, new Sigmoid())
    .advancedOptions()
        .outputSystemOptions()
            .addSysOutEventListeners()
```

When implementing a new builder for a core module the AbstractBuilder class has to be extended. This will require various generics to be specified and the explanation will be with our text classification example.

Besides extending the AbstractBuilder there have to be two other classes. One extending the AdvancedOptions class and one extending the PreProcessorAdvancedSubOptions class. Often the class extending the AdvancedOptions will not contain much actual implementation but only serves to specify the needed generics. The preprocessor options however have to be completely implemented according to the specifications of the preprocessor in this core module.

These are the three class headers of the classes that need to be created:

```
public class TextClassificationBuilder extends AbstractBuilder <TextWrapper, Document,
Integer, TextClassificationBuilder, TextClassificationAdvancedOptions>

public class TextClassificationAdvancedOptions extends AdvancedOptions <Document,
TextClassificationBuilder, TextClassificationAdvancedOptions, PreprocessorOptions>

public class PreprocessorOptions extends PreProcessorAdvancedSubOptions
<TextClassificationPreProcessor, TextClassificationAdvancedOptions,
PreprocessorOptions>
```

The generics have to be specified as follows:

For the Builder Class:

- TextWrapper is in this case the RawData type that is provided by the Importers.
- Document is the model object class generated by the preprocessor.
- Integer is the result provided by the machine learning prediction.
- TextClassificationBuilder must be the class of the builder itself so that the return values of chaining functions do not have to be casted.
- TextClassificationAdvancedOptions is the class that extends AdvancedOptions.

For the AdvancedOptions Class:

- Document is again the model object class generated by the preprocessor.
- TextClassificationBuilder is the type of Builder that this AdvancedOptions implementation belongs to. It is returned by the builder() method for chaining purposes.
- TextClassificationAdvancedOptions is the class of the AdvancedOptions itself. Again this is needed for chaining.
- PreprocessorOptions is the class specifying the options for the Preprocessor of this Core module.

For the PreProcessorAdvancedSubOptions Class:

- TextClassificationPreProcessor is the class that implements the IPreprocessor interface. This will be returned by this class' make method.
- TextClassificationAdvancedOptions this is the class of AdvancedOption this belongs to. It well be returned by the advancedOptions method of this class for chaining purposes.
- PreprocessorOptions is again the class itself needed for the option chaining.

There are some patterns that need to be followed when implementing a builder. One is the constructor of the advanced options. It should look like this:

```
private TextClassificationAdvancedOptions(TextClassificationBuilder _self) {
    super(_self);
}
```

When the advanced options are instantiated they will be passed the instance of the builder. This again needs to be passed along to the super constructor, so that the navigation with the builder, respectively advancedOptions methods works.

The next pattern is when overwriting the PreProcessorAdvancedSubOptions or adding/overwriting any other AdvancedSubOption Class.

```
private PreprocessorOptions(TextClassificationAdvancedOptions _selfP) {
    super(_selfP);
    super.instance = new TextClassificationPreProcessor();
}
```

Each AdvancedSubOption class contains an attribute called instance. This is what is returned by the make method unless that method is specifically overwritten. This instance should be made available to be configured in the constructor of the class. The reason for this will be explained when the next and final pattern is shown.

This pattern shows how to implement a new options method to an AdvancedSubOption class.

```
public PreprocessorOptions addTokenizer(IDocumentTokenizer tokenizer) {
    super.instance.getConfiguration().addTokenizer(tokenizer);
    return this;
}
```

As shown option methods should mostly be delegates to the instance attribute of this AdvancedSubOption class. This is why it is important to have instantiated the instance object in the constructor.
There are other ways to handle this, such as the usage of a properties or configuration object and then constructing the instance object directly in an overwritten make method. However this is discouraged as the builder should have as little factory logic as possible to ensure possible usage of the built instances outside of the builder environment.

The other important point here is that every options method has to return the this variable. This is absolutely needed to make chaining possible. Also note the return type of the method being the AdvancedSubOption class in question itself.

Besides these patterns there is another important point to consider when implementing builders. A builder should offer various static factory methods returning a preconfigured builder. These default configuration can then be used by a less versed user or in a pipeline.

2.2.2.7 Pipelines

Pipelines on a core module can be very specific, making use of a particular builder and machine learning component. They can be hierarchically orders so that first an advanced pipeline is made with very intricate and highly detailed workflows and then a basic pipeline that extends the advanced one and offers much more easy to use, already parameterized calls to the advanced pipeline.

When designing a Pipeline particular attention should be paid to the Javadoc. Since the methods on a pipeline class build consecutive steps of a workflow it should be documented which method build one workflow, which method can be substituted by another and which calls are optional.

2.2.3 Tasks

Tasks are packages that use a core module in its as-is state with a particular dataset and use case in mind. To this end the following has to be done:

2.2.3.1 Importer

PlebML's interface to physical storage is done by importers. These need to handle the file reading to supply the preprocessor with the needed raw data. Since the format of a file or storage location can vary strongly with each usage of a core module these importers are provided by the task package.

2.2.3.2 System Integration

The Task level implementation is also responsible to interact with the surroundings of PlebML. Whether this is a simple main class or an integration into an existing software project is decided here. Interaction with the core module should if possible be via pipeline-classes.

However depending on the circumstances the system integration can also use other layers of PlebML directly. There are three usage concepts for PlebML according to its software structure.

Usage Concept of PlebML

PlebML offers conceptually three layers for system integration depending on which part of PlebML are to be used.

Task-Level Integration

As described above, task-level integration is the most desirable as the only thing to be done is providing the necessary Importers and interaction with the provided pipelines. At most an own pipeline has to be written if those in the core module do not offer the desired functionality. This level of integration uses PlebML as a whole.

Module-Level Integration

Module-level integration is most likely used when only the feature vector generation part is of interest. This could be when the system around PlebML already has its own machine learning engine. In this case interaction will mainly be with the builder and the resulting feature vector generators. This is because the feature implementations are actually part of a Core Module, not the Core Library. It is possible that some conversion has to be done so that the results of the feature vector conform to the needed format.

Core-Level Integration

The primary use-case of pure core-level integration is usage of PlebML's machine learning components with already generated feature vectors. For this to happen the given vectors have to be converted into the PlebML format and integration of the provided machine learning component has to be done manually. As for the abstraction layer for the machine learning part see the chapters “Machine Learning Component” and “Machine Learning”

2.2.4 Design Concerns

Due to the chosen architecture it is hard to share utility functionality (such as mathematical formulas, string utils and such) between core modules. Either the core modules are made dependent from one another, which can lead to problems when distributing PlebML with a selection of Core modules, or the necessary code is in each core module packaged redundantly which constitutes a DRY violation.

3 Text classification with PlebML

In this chapter we describe a specific implementation of a core module that is part of our current PlebML code. It will mainly focus on changes in comparison to our codebase.

3.1 Model Object Class

The model object class of the text classification core module is split into a hierarchical structure of Documents, Paragraphs, Sentences and Tokens. The basic structure is still the same as it was in our code base. However it has been refactored to conform to our new structural requirements and to be easier to use. Figure 10 is not an exhaustive class diagram but shows the most important methods and attributes. As mentioned before the Token class now implements the IPreprocessorDataReciever interface allowing abstract storage of metadata on the model. With this refactoring, usage of metadata on Feature implementations is very streamlined and intuitive.

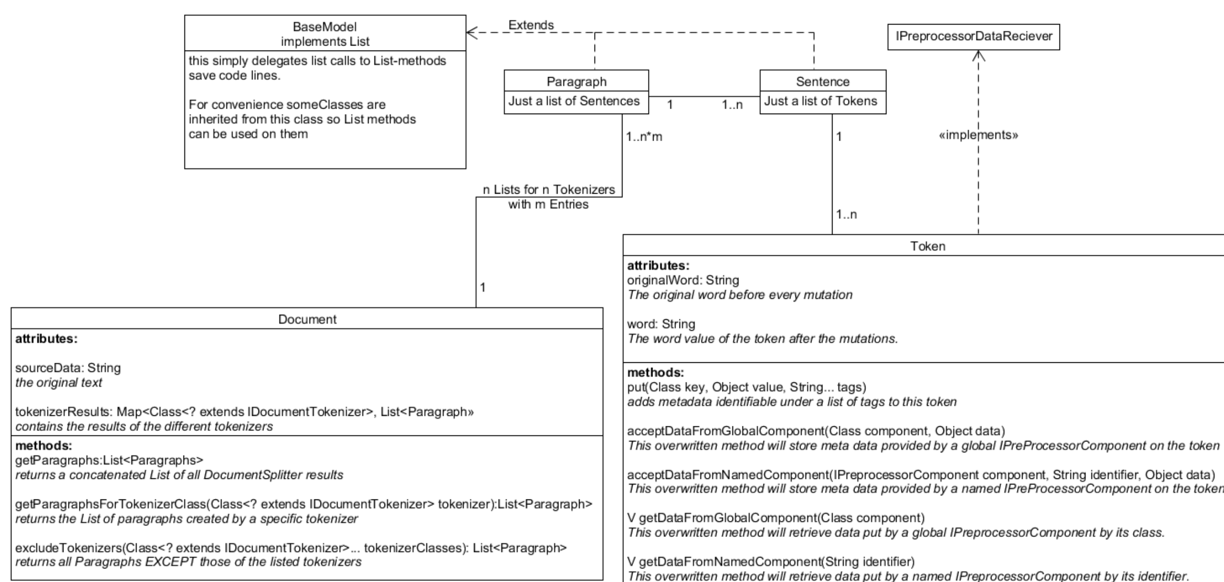

Figure 10 - Changes to Model Class

```
@RequiresComponentClass(componentClass = TweetNLPPOSTagger.class)
public class NumberOfPOSTags implements ISentimentAnalysisDefaultVectorFormat<Void>
```

This class has the RequiresComponentClass annotation prompting the builder to load the TweetNLPOSTagger preprocessor component when constructing the preprocessor. When this features extract method is called the meta data can easily be retrieved with this code:

```
t.getDataFromGlobalComponent(TweetNLPPOSTagger.class) //t is a token
```

A feature should always assume that the correct preprocessor settings were run. It could happen that the dependency crawling for the preprocessor was not executed and the user did not add the TweetNLPPOSTagger component. In this case the method would return null. It is the features job to handles such a case as it is impossible for the Token (which holds the relevant data) to know sensible default values for each component. If this is a problem the Token offers orDefault() variations of the data retrieval methods, very similar to those of the java maps. They accept a second parameter which represents a default value to be returned if the original result would have been null.

3.2 Preprocessor

The preprocessor has not been structurally changed but some components have been added.

3.2.1 GeneralTokenizer

The GeneralTokenizer is a tokenizer designed for text written in the Roman alphabet. It splits the text in paragraphs sentences and words and additionally analyses the document for an xml tree like structure.

When tokenizing the GeneralTokenizer first sanitizes the document to remove line separators and other attributes that would obfuscate xml like tokens and then splits it in tokens line by line. There are two kinds of tokens, normal tokens are considered to be words and are therefore simply delimited by spaces, tokens that start with a ‘<’ and for which a corresponding closing ‘>’ character is found on the same line are considered to be xml tokens and everything between the opening and closing character is treated as a single token, in the further processing all attributes of these tokens are removed from the text of the token and instead kept as metadata on the token.

Within a xml tree like structure all tokens are kept in a single paragraph, sentences are almost formed the same way as for normal text, with the only addition that each xml token ends the previous sentence and is placed as single token in a new sentence. An xml tree like structure begins with an opening xml token that has the form “<...>” and ends either in a correct way where all opening tokens are closed by a closing token “</…>” or in an erroneous way, where a closing token appears for which no opening token exists.

Outside of xml tree like structures paragraphs are separated by two or more consecutive new lines and sentences are ended by either punctuation or new lines.

3.2.2 SimplestTokenizer

This tokenizer is, as its name implies, the most basic implementation of a tokenizer. It splits up the whole document by spaces. The generated words are organized in sentences by simple punctuation and all sentences are kept in a single paragraph.

3.2.3 SimpleEMailNormalizer

To normalize email addresses within documents this mutation was implemented. It replaces all email addresses with a regex expression by a default term. This is important to avoid fitting for example a spam filter to certain addresses instead of the general structure of spam.

3.3 Included Features

The included features have stayed largely the same and can be found in the appendix. There are a few utilities that are noteworthy as they might be used very often when adding new features for text classification.

3.3.1 N-Gram Utility Class

The NGram class is instantiated with a size and a delimiter parameter as shown here:

```
public NGram(int size, String delimiter)
```

The size is the length of the n-gram measured in number of tokens. The next important method is:

```
public void addNext(String s, Consumer<String> callOnceFull)
```

This method accepts a String and adds it to the current n-gam. Once the n-gram reached a number of elements equal to the size parameter it will join the elements together delimited by the constructor's delimiter parameter and pass it to the callOnceFull consumer. The passed string is meant to be used as local key in any n-gram feature.

There is also an option to create n-grams with wildcards meaning that between index 0 and index n all elements and element combination will be substituted for every possible wildcard permutation. Consider the n-gram It_is_a_nice_day. It would be passed to the callOnceFull consumer in the following permutations:

It_*_a_nice_day.
It_is_*_nice_day.
It_is_a_*_day.
It_*_*_nice_day.
It_is_*_*_day.
It_*_*_*_day.

The * can be any wildcard passed be that a static string or dynamic data like a POS tag. The method to work with wildcards only differs slightly.

```
public void addNextWildCarding(String s, String possibleWildCard, Consumer<String>
callOnceFull)
```

It is strongly recommended to only work with one of the two methods (adding with wildcards or without) per n-gram instance.

3.3.2 Dictionary System

The system for dictionaries has been completely reworked to be more generalized and powerful. Dictionaries can now be transient, meaning that it is composed of several other dictionaries and lookups are distributed to each component dictionary and reduced by a given function to a single result. A transient dictionary can also be re-constructed so that it is not transient anymore but instead holds all reduced entries of the component dictionaries. Then it can be stored to a physical location. This is often recommended for speedups if the required dictionary is not conditionally constructed.
This example code snippet shows how a Dictionary is constructed:

```
Dictionary partialNeg = DictionaryRepo.loadFromRawDictionary(
        NEG_DICT_NAME,
        "bingLiuPartialNeg",
        lineReaderFunction
);

Dictionary partialPos = DictionaryRepo.loadFromRawDictionary(
        POS_DICT_NAME,
        "bingLiuPartialPos",
        lineReaderFunction
);

Dictionary partialCombined = DictionaryRepo.loadFromRawDictionary(
        COMBINED_DICT_NAME,
        "bingLiuPartialCombined",
        lineReaderFunctionPartial
);

Function<List<Map.Entry<Dictionary, Integer>>, Integer> reducer =  (hits) -> {
    return hits.stream().mapToInt((entry)->{return entry.getValue();}).sum();
};


Dictionary<Integer> dict = new Dictionary<Integer>(DICT_IDENTIFIER)
        .addFromDictionary(partialNeg)
        .addFromDictionary(partialPos)
        .addFromDictionary(partialCombined)
        .reduceMultipleHitsWith(reducer)
        .constructFromTransientData()
        .store(DICT_IDENTIFIER);

return dict;
```

The DictionaryRepo is a class with some static loader functions parameterized by the physical location of the dictionary file, the dictionary identifier and a line reader function. This example also shows how to construct a transient dictionary, then make it non-transient and finally store it so that on the next usage of this dictionary it can be directly loaded from physical storage.

3.4 Pipelines and Builder

This chapter gives an overview over the existing pipelines and builders in the text classification core module.

3.4.1 TextClassificationBuilder

The structure and usage of the builder has already been discussed as part of the previous chapter “Builder Implementation”. This section will shortly summarize the existing default configurations.

```
public static TextClassificationBuilder getDefaultEnglishLanguageBuilder()
```

This default configuration will return a suitable feature vector generator for English language text classification tasks. The configuration is:

- **Preprocessing**
    - Simple Tokenizer
      Delimits by whitespaces in text.
    - Username Normalizer
      Substitutes all string starting with @ with a generic token.
    - URL Normalizer
      Substitutes all URLs with a generic token.
    - Dependency crawling is enabled
- **Feature Extraction**
    - N-Grams
    - Lemma N-Gram
    - Wild card N-Grams
    - Number of all caps tokens
    - Number of POS Tags
    - Number of negated contexts
    - Number of elongated words
    - Whether the last token contains punctuation
    - Occurrence of continuous punctuation

Memory usage strategy and parallelization strategy can be changed. They are on their respective default values of heap-use and serial.

The other default configuration is

```
public static TextClassificationBuilder getDefaultRomanAlphabetLanguageBuilder()
```

In comparison to the configuration above it removes all POS and negation features as they are dependent on the fact that the texts language is English.

In both cases it is recommended to add and/or replace the simple tokenizer with more specific ones.

3.4.2 AdvancedTextClassificationPipeline

This pipeline offers granular parameterization over the most common optimization workflows. Its important methods are:

```
public AdvancedTextClassificationPipeline
             (FeatureVectorGenerator featureVectorGenerator)
```

Use this constructor if you plan to use one of the c-parameter optimization methods. This constructor will instantiate a dummy placeholder for the ML-component, intended to be later replaced by an optimizer call.

```
public AdvancedTextClassificationPipeline
             (FeatureVectorGenerator featureVectorGenerator, String ident)
```

This constructor loads a saved ML-component with the ident parameter. This is used if the component in question is already trained and the pipeline is to be used for evaluation and prediction.

```
public void optimizeMLComponentCParameter
        (OptimizationGoal goal,
         LiblinearClassifierSolverType solver,
         double eps,
         double minLogC,
         double maxLogC,
         double cStepSize,
         int nrOfFolds,
         Importer importer ,
         String... files)
```

This is the c-parameter optimization method. It will search for the c-parameter resulting in the best score according to its optimization goal. It will replace the existing ML-component for this pipeline.

```
public EvaluationResult evaluate(Importer importer, String... files)
```

This Method will run a text classification specific kind of evaluation on the given data. The data has to be training data and the ML-component has to be trained already. The read data will then be predicted and the result will be compared to the entries in the training file. The evaluation result offers various statistical measures for the performance of the Pipeline such as precision, recall and F1-score.

3.4.3 BasicTextClassificationPipeline

The basic pipeline extends the advanced one and offers already parameterized calls for the optimization methods. The parameters are empirically chosen to result in decent scores while retaining an acceptable runtime.

Example usage of the basic pipeline:

1) Instantiation, training and storing.

```
BasicTextClassificationPipeline pipeline =
BasicTextClassificationPipeline.defaultPipeline("./" + ident);
pipeline.findGoodClassifier(
        new TweetImporter(),
        "/Datasets/semevalsentiment/fromnrc/task-B-train.tsv",
        "/Datasets/semevalsentiment/fromnrc/task-B-dev.tsv");
pipeline.storeMachineLearningComponent("sentiment15RC");
```

2) Instantiating, loading and evaluating.

```
BasicTextClassificationPipeline pipeline =
BasicTextClassificationPipeline.loadTrainedDefaultPipeline("./" + ident, ident);
EvaluationResult result = pipeline.evaluate
       (new TweetImporter(),
        "/Datasets/semevalsentiment/task-B-test2013-twitter.tsv");
```

It is important to note that the settings for the feature vector generation have to be constant between training and evaluation/prediction. Otherwise there will be non-existent indices and a resulting score loss.

4 Best Practices for Core Module Design

When designing a core module, several things should be done that are considered best practice for PlebML.

1) The model class should implement the IPreProcessorDataReciever interface.

2) The preprocessor should consists of a main class and components implementing the IPreprocessorComponent interface. Also it should support the methods necessary for dependency crawling.

3) The feature package should contain an interface extending the IFeature interface and it should define the model object class generic. All features should then implement this new interface.

4) The feature implementations should make use of the dependency crawling annotations.

5) The core module should have a builder extending the abstract builder according to specifications in the "Builder Implementation" chapter.

6) This builder should offer static factory methods for itself with sensible default configurations.

7) The core module should offer one or more pipelines. Often it will make sense to start with a highly granular and advanced pipeline and then generalize it by extension as done in the text classification module. Keep in mind that having basic pipelines makes the usage of the core module much easier to people not versed in machine learning.

8) Include one or more sample tasks showing the usage and specialties of the core module.

5 Measurements and Results

5.1 Machines

For all measurements the machines described in *Table 5* and *Table 6* were used:

Machine A	
Processor	Intel Xeon 8x 2.53GHz
Main memory	16GB
Disk	Virtual disk 40GB
Java version	Java(TM) SE Runtime Environment (build 1.8.0_25-b18)
JVM version	Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
OS	Windows 7 Enterprise sp1

Table 5 - Machine A description

Machine B	
Processor	Intel Core i7-3520M 4x ~3.39GHz
Main memory	8GB
Disk	Samsung SSD 840 PRO Series ~100kIOPS 256GB
Java version	Java(TM) SE Runtime Environment (build 1.8.0_20-b26)
JVM version	Java HotSpot(TM) 64-Bit Server VM (build 25.20-b23, mixed mode)
OS	Windows 7 Professional sp1

Table 6 - Machine B description

5.2 Performance Enhancement Evaluation

In this chapter the performance enhancements are evaluated based on the performance of the original framework [1].

To measure the performance enhancements of the new framework, its performance is compared to the original framework based on the tweet sentiment task. The direct comparison could only be performed on Machine A due to the massive memory consumption of the original framework that leads to OutOfMemoryErrors (OOME) on Machine B.

5.2.1 Feature Extraction

To compare the performance in the feature extraction stage, the set of features used in the original framework was applied on 9912 tweets out of the tweet sentiment dataset. In *Table 7* the times used to extract the denoted set of features are shown, in the column Original the time of the original framework on Machine A is noted and in the other two columns the times of PlebML on Machine A and B.

Test	*Original on Machine A*	*Original on Machine B*	*PlebML on Machine A*	*PlebML on Machine B*
Time	5:42	OOME	2:17	2:14

Table 7 - Performance comparison of the feature extraction

As shown in *Table 7* the time needed to extract the features with PlebML is less than half the time needed with the original framework and additionally it’s possible to extract the features with far less memory and therefore smaller machines can be used.

5.2.2 Machine Learning

In this section the performance of the original frameworks wrapper for Liblinear is compared to the improved version of Liblinear in PlebML concerning memory consumption and elapsed time. For this purpose the best c parameter is searched within $[2^{-8}, 2^{-7}, \dots, 2^{-1}]$ using the solver L1R_L2LOSS_SVC with 10 fold cross validation for the tweet sentiment problem. The results are shown in *Table 8*.

Framework *Machine*	*Original* *A*	*Original* *B*	*PlebML* *A*	*PlebML* *B*
Used time	88:34	OOME	24:31	29:42
Used heap	8.5GB	OOME	6GB	3.2GB

Table 8 - Performance comparison of the machine learning

The improved version of Liblinear is more than three times as fast as the original library and additionally solves the same problem with less than half the memory.

5.2.3 Conclusions

With a total time to extract features and to find a classifier of approximately 27 minutes PlebML is considerably faster than the original framework which needs more than 94 minutes for the same task. Additionally with PlebML it is possible to solve this task with less than half the memory and therefor smaller machines like Machine B can be used to solve the same problem.

5.3 Tested Datasets

In this chapter the various datasets used to test the framework and the results achieved with these datasets are presented.

5.3.1 Semeval

This dataset originates from the Semeval competition 2014, the original framework was written for this dataset. It consists of more than 11'000 tweets labelled with their sentiment, from which 1853 are reserved for the test set.

5.3.1.1 Results

For this dataset the same selection of features was used as by the original framework. To find the best classifier for this data set, the optimizeMLComponent method of the advanced pipeline was used. The detailed results of the cross validation done for this search are presented in *Figure 11*. In this figure the y-axis measures the average F1 score achieved with the solver type and log c value that are stated below the x-axis.

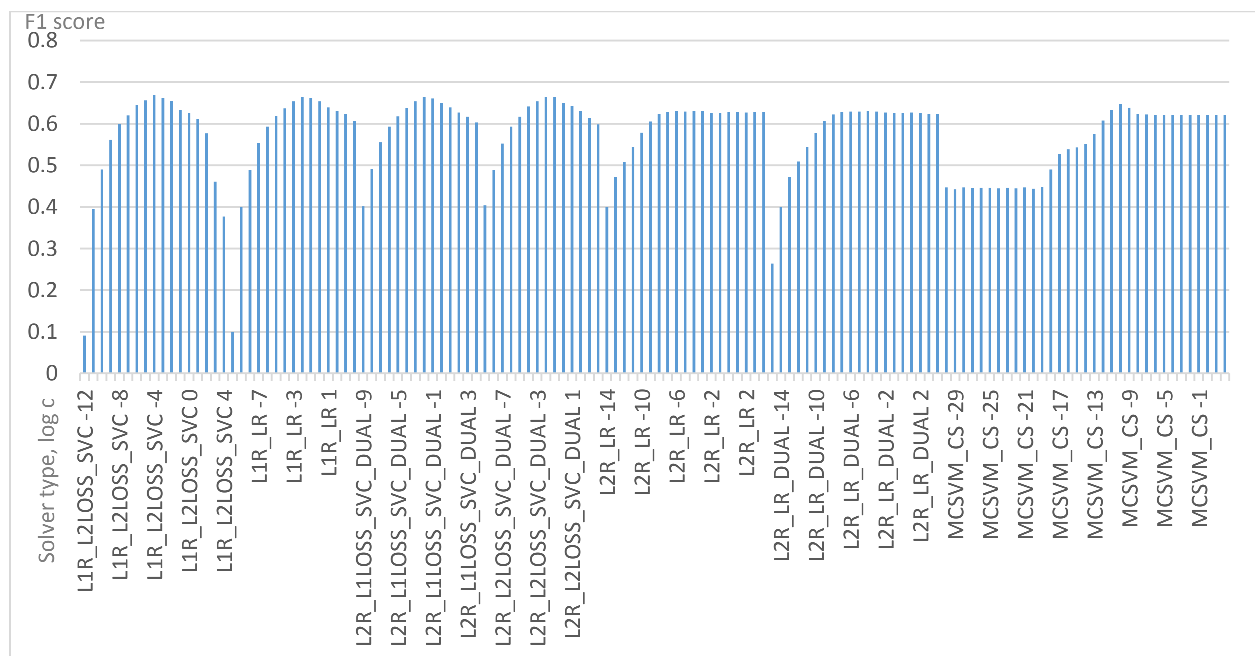

Figure 11 - Semeval scores of optimizeMLComponent for different solvers and log c.

The best value (0.669) was achieved with the solver type L1R_L2LOSS_SVC and a log c parameter of -4, the evaluation with the test set yields an F1 score of 0.682.

5.3.2 Spam

This dataset is composed of several publicly available labelled spam and non-spam email datasets, which were cleaned from viruses. The datasets used are: spamassasin, lingspam and enronspam as found on csmining [20].
The dataset consisting of about 62’000 labelled emails was split in half by random choice to get a train and a test set.

5.3.2.1 Results

For this dataset in addition to the default features for English language of PlebML the GeneralTokenizer and the corresponding features were used to benefit from the XML structure in some emails. Further all email addresses and URLs have been normalized to avoid over fitting. Due to the size of the problem it was not feasible to simply optimize over all solver types. Therefore only for one solver type the optimal classifier has been searched, the results of the cross validation are shown in *Figure 12*. In this figure the values are the averaged F1-score and on the x-axis the solver type and the corresponding log c value are noted.

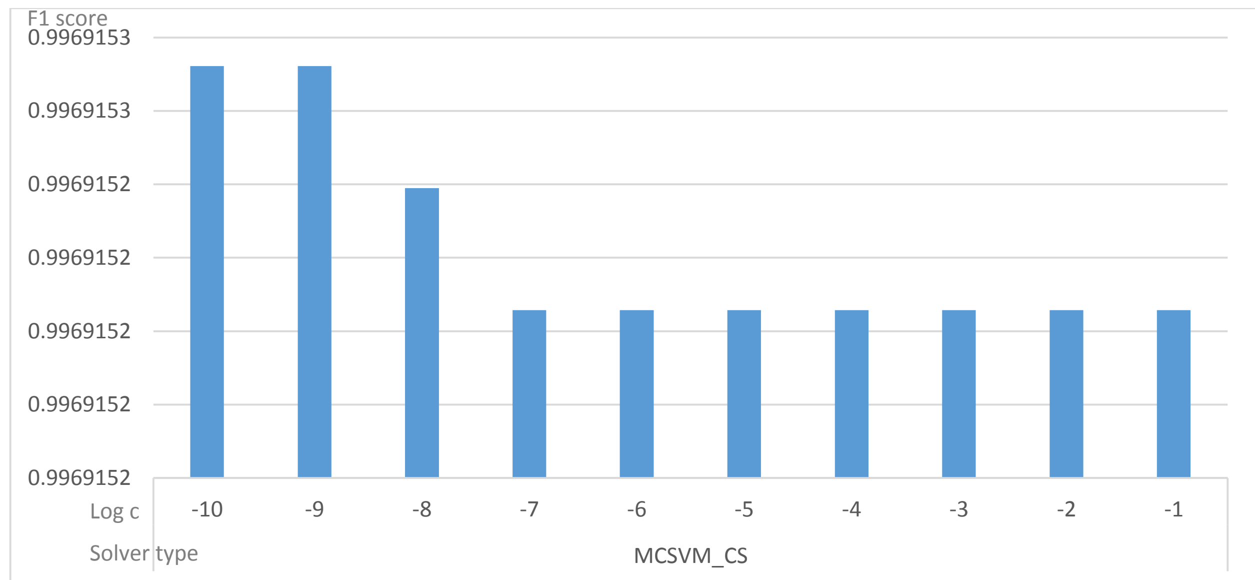

Figure 12 - Spam optimizeOverC MCSVM

The evaluation of the best candidate found with cross validation resulted in a final score of:
Ham precision: 0.996 and ham recall: 0.999 where ham stands for non-spam emails.

5.3.3 Movie Review Sentiment

This data set is part of a Kaggle competition, they describe it as follows:
"The Rotten Tomatoes movie review dataset is a corpus of movie reviews used for sentiment analysis, originally collected by Pang and Lee [21]*. In their work on sentiment treebanks, Socher et al.* [22] *used Amazon's Mechanical Turk to create fine-grained labels for all parsed phrases in the corpus. This competition presents a chance to benchmark your sentiment-analysis ideas on the Rotten Tomatoes dataset. You are asked to label phrases on a scale of five values: negative, somewhat negative, neutral, somewhat positive, positive. Obstacles like sentence negation, sarcasm, terseness, language ambiguity, and many others make this task very challenging."* [23]

The movie review dataset is used to test the PlebML with a huge load of test instances, as the dataset consists of more than 150'000 labelled phrases, and the feature extraction results in features in more than 5'700'000 dimensions.

5.3.3.1 Results

For this dataset the same features as for the Semeval dataset were used. Due to the size of the problem it was not feasible to simply optimize over all solver types. Therefore only for one solver type the optimal classifier has been searched, the results of the cross validation are shown in *Figure 13*. In this figure the values are the averaged F1-score and on the x-axis the solver type and the corresponding log c value are noted.

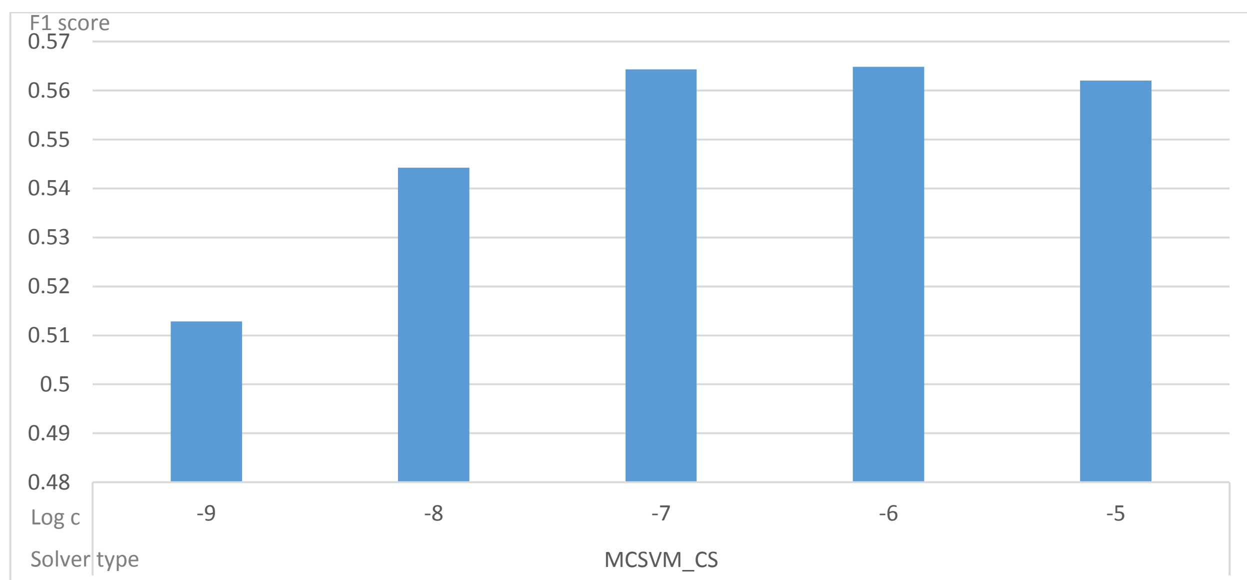

Figure 13 - Movie review sentiment optimizeOverC

The best score reached with cross validation is 0.565 with a log c value of -6. A classifier trained with these settings was used for a submission on Kaggle and reached a final score of 0.633 and a theoretical position in the competition of 216 out of 862 participants. The winner of the competition achieved a score of 0.765.

5.3.4 Black Box Features

This dataset originated in a Kaggle [24] competition where the participants had to classify a set of features without knowing where this features originated from, additionally to a small set of labelled feature vectors a huge set of unlabelled vectors was provided. This dataset was used to test how easy the framework can be adapted to different problems.

5.3.4.1 Adaptions

For this problem a new core module had to be implemented consisting of a new document model and a feature to handle the new model. As the data to classify already only consists of features, the feature extraction is trivial. Additionally to the new core module an importer had to be written for this dataset. Further adaptions like features that would use the unlabelled vectors for clustering are not implemented yet.

5.3.4.2 Results

For this dataset all solver types that support multiclass problems provided by PlebML were tested with the recommended settings and for each solver type the best classifier was used for a submission on Kaggle, the results are shown in *Figure 14* and *Figure 15*, in the first figure the cross validation results are shown and in the second the scores reached in the Kaggle competition are presented.

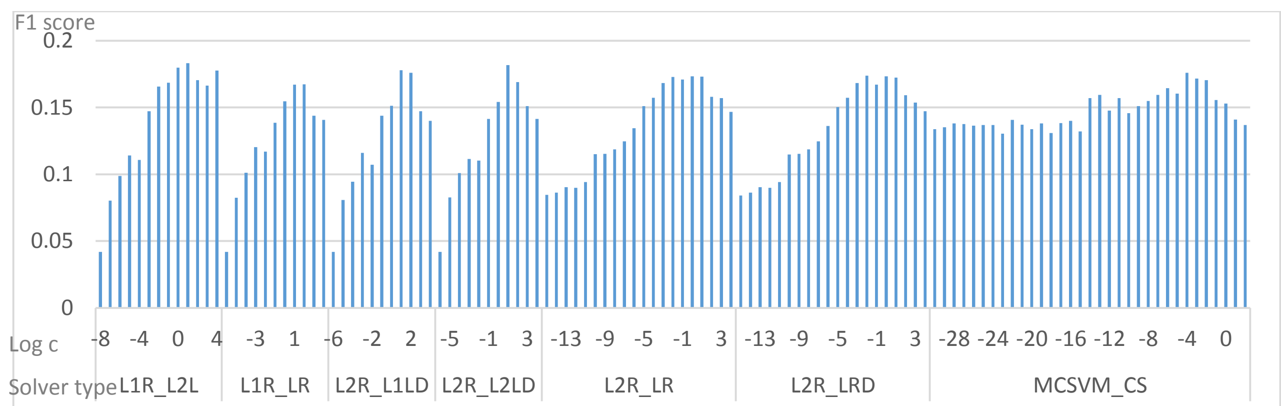

Figure 14 - Black box cross validation results

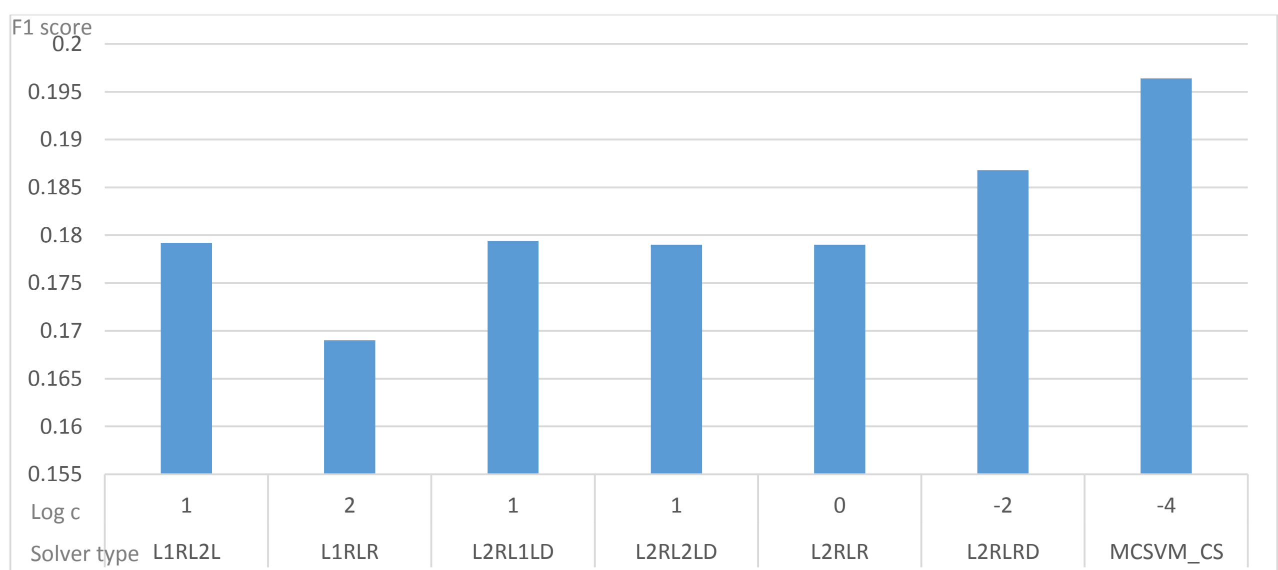

Figure 15 - Black box Kaggle submission results

The best submission on Kaggle with a score of 0.196 was achieved with a classifier trained with the solver type MCSVM_CS and a log c parameter of -4, this score is above the average of all participants of the competition who did not use the unlabelled feature vectors.

5.4 Feature Extraction Performance Measurements

In this section a selection of performance measurements during the feature extraction is presented for different setups and datasets, the complete collection of all measurements made is in the Appendix. In all figures in this chapter the left y-axis means features per second, the right y-axis means feature vectors per second and in the x-axis is the number of processed documents.

5.4.1 Spam

In this setup features for our spam data set are extracted. This dataset is well suited for a high load test as each email may consist of hundreds of lines. On Machine A only the heap memory strategy is tested, as it has only a small network disk with very high access times, on Machine B the MEMORY_MAPPED_HASH and the HEAP strategy are tested. The CACHED and CACHED_HASH strategy were not tested for this dataset, as they are designed for smaller loads.

5.4.1.1 Fastest Feature Extraction

The fastest feature extraction was achieved on both machines using the parallelisation strategy SER_DATA_PAR_FET, but due to the memory channel as main bottleneck, the speedup compared to SER_DATA_SER_FET is minor. On Machine B in combination with the MEMORY_MAPPED_HASH strategy more than 250'000 features were generated and about 115 email were processed per second as shown in *Figure 16*.

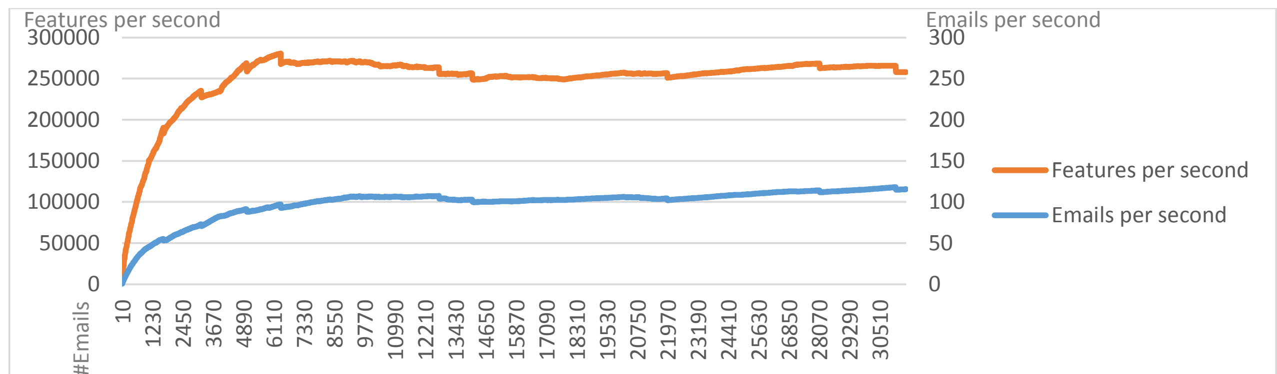

Figure 16 - Spam feature extraction on Machine B, SER_DATA_PAR_FET and MEMORY_MAPPED_HASH

5.4.1.2 PAR_DATA_PAR_FET Performance Issues

With the parallelisation strategy PAR_DATA_PAR_FET the system requires not only more memory, what lead to the OOME on Machine B, but also more memory bandwidth, as multiple emails are processed at once and therefore cache misses significantly increase. This leads to the poor performance of the strategy PAR_DATA_PAR_FET in this setup.

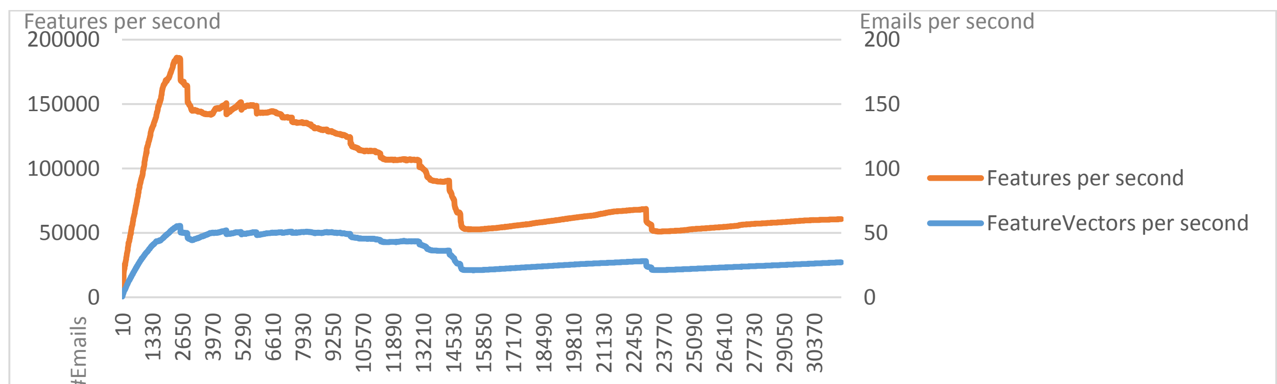

Figure 17 - Spam feature extraction on Machine B, PAR_DATA_PAR_FET and MEMORY_MAPPED_HASH

5.4.1.3 Performance Drop on Heap, Machine B

The performance drop in the two experiments with the memory strategy HEAP on Machine B which completed successfully is caused by massive paging as Machine B barely had enough memory to extract the features on heap. One can see the performance drop in *Figure 18* at about 27'000 processed emails.

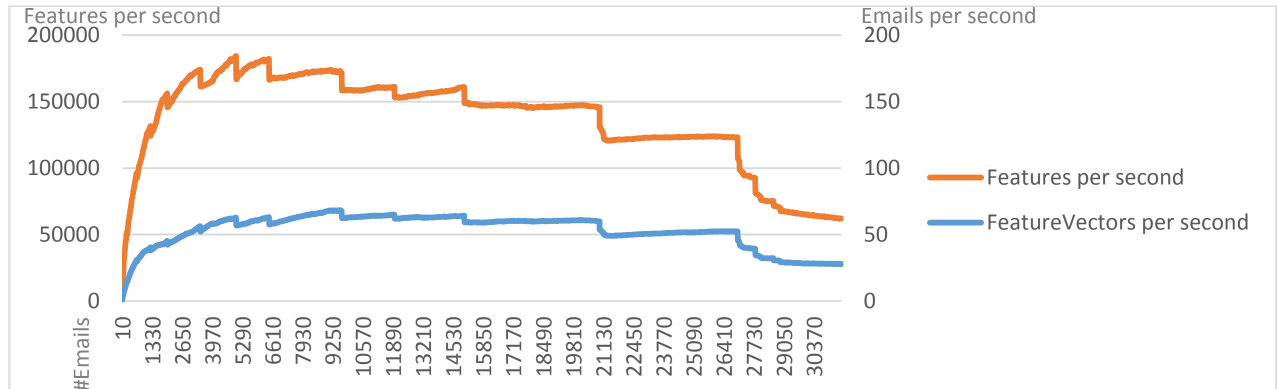

Figure 18 - Spam feature extraction on Machine B, SER_DATA_SER_FET and HEAP

5.4.2 Semeval

In this setup features for the tweet sentiment data set are extracted. This dataset is well suited to test a low load per document, as each tweet only consist of a single line of text. On Machine A only the heap memory strategy is tested, as it has only a small network disk with very high access times, on Machine B all strategies are tested.

5.4.2.1 Cached

The experiments with the cached memory strategies have shown that they are slower than the other strategies, but they provide the significant advantage of the smaller amount of required resources in terms of main memory and disk space. The results are shown in *Figure 19*.

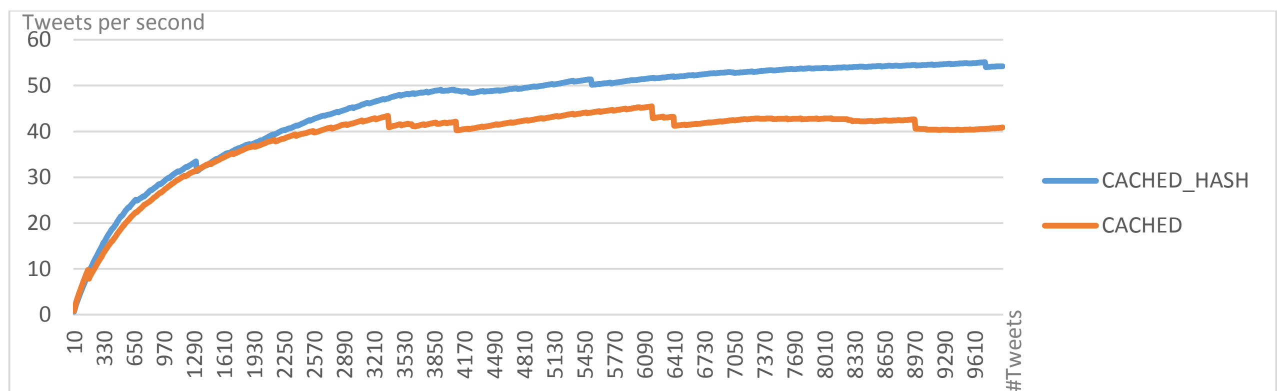

Figure 19 - Semeval feature extraction on Machine B CACHED, CACHED_HASH and SER_DATA_PAR_FET

5.4.2.2 PAR_DATA_PAR_FET on Machine B

On Machine B the increased memory usage of the strategy PAR_DATA_PAR_FET lead to a minor performance drop compared to the other strategies. One possible reason for the reduced performance of this strategy on Machine B is the hardware architecture, as its processor only has two physical cores and the main memory is shared with other hardware. Therefore only a small theoretical speedup could be gained compared to SER_DATA_PAR_FET, but due to the increased need of synchronization the performance was worse. In *Figure 20* the performance of the parallelisation strategies is presented.

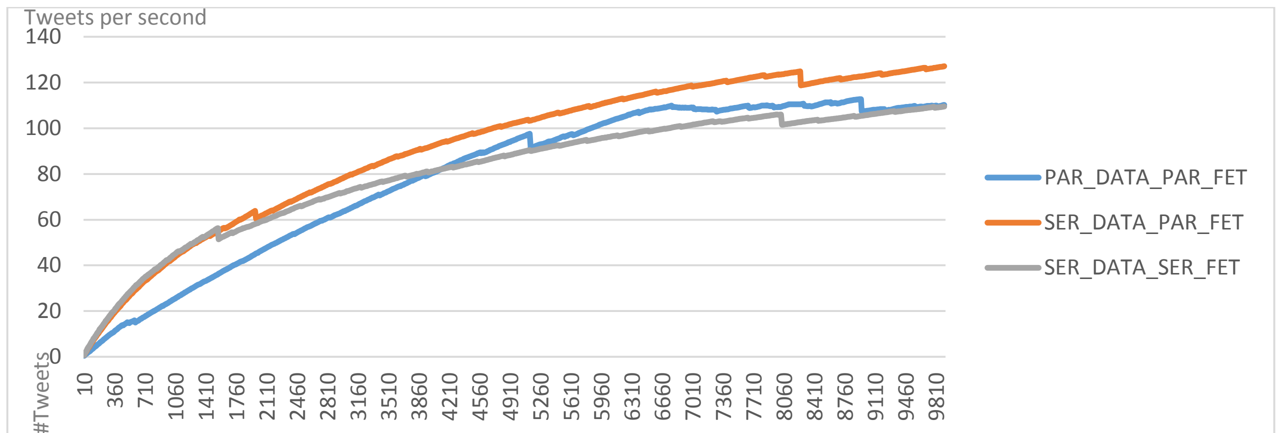

Figure 20 - Semeval feature extraction on Machine B, HEAP

5.4.2.3 Fastest Feature Extraction

The fastest feature extraction was achieved with the combination of the strategies PAR_DATA_PAR_FET and HEAP on Machine A as shown in *Figure 21*. In this setup more than 160'000 features and about 150 feature vectors were generated per second.

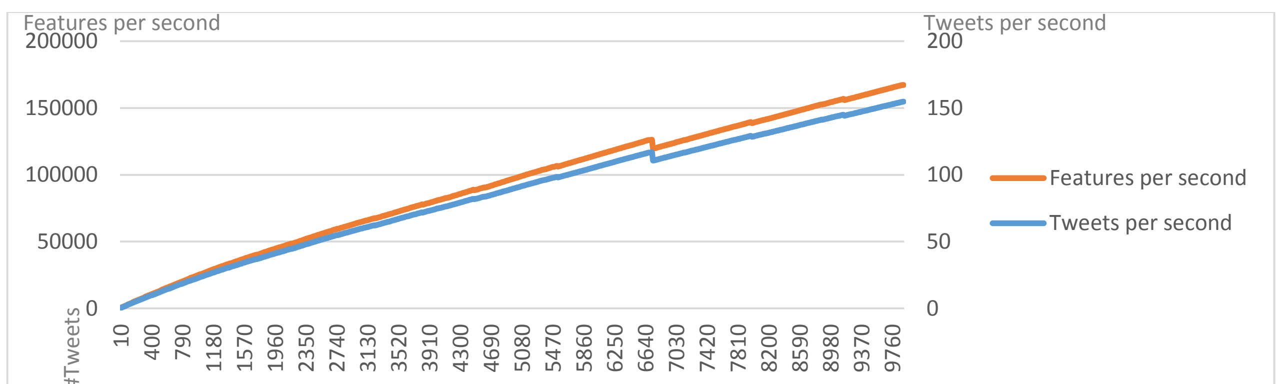

Figure 21 - Semeval feature extraction on Machine A, PAR_DATA_PAR_FET and HEAP

6 Review of Results and Implications for Further Work

In this chapter the goals initially stated in this work are revisited and we check whether we achieved them or not. Also we will discuss some open points that could be pertinent for further work.

6.1 Goal Review

1. There should be default configurations and default workflows such that the framework can be used by someone with very little or no expertise in machine learning. The usage should consist of only very few lines of code.

We have implemented default configurations on the builder in form of factory methods. These will provide preconfigured feature vector generators for a specific kind of machine learning task. We also proposed a best practice for developers that introduce new builders in core modules that encourages to provide such default configurations for each builder.
With them being directly on the builder class in a Core Module they are easily accessible to a software developer using PlebML and can be found at the same place for every Core Module.

2. In comparison to our codebase performance should be improved, both in term of RAM usage and elapsed runtime.

We dedicated the chapter “Performance” to this goal and cite our conclusion here again:

“With a total time to extract features and to find a classifier of approximately 27 minutes PlebML is considerably faster than the original framework which needs more than 94 minutes for the same task. Additionally with PlebML it is possible to solve this task with less than half the memory and therefor smaller machines like Machine B can be used to solve the same problem.” [25]

3. The software architecture needs to be overhauled as the separation of concern is weak, the interfacing has issues, extending the framework is very cumbersome and implementing a wide variety of tasks is very hard.

This was solved on one side by introducing the Core Library and Core Module architecture to ensure easy extensibility and separation into pipelines, feature vector generators and machine learning components on the other side for separation of concern. The new architecture allows partial integration of PlebML into other systems in case one of these 3 parts is already provided by the surrounding system.

6.2 Open Points

However there are some open points that would need to be addressed in further work:

- External resources are not handled very well as PlebML in its current state requires a rigid directory order and does not allow this to be changed. One possibility to solve this is to use Streams instead of filenames or location strings. This would also make the Importer interface obsolete, removing some implementation overhead.
- Currently PlebML has no unifying naming scheme. This should be refactored to conform to Java clean code.
- By now only sparse vectors are extracted during the feature extraction, to eliminate the overhead of the sparse representation for dense vectors, as found for example in image classification, a dense vector implementation should be introduced.
- The parallelisation during the feature extraction yields only a minor speedup, this has to be addressed in a later version of PlebML.
- PlebML offer only a module for text classification, more modules for a larger variety of problem domains should be implemented.

These are currently the five major open points in PlebML that should be addressed first.

7 Bibliography

[1] D. Egger and P. Julmy, Development of a framework for text classi-, Winterthur: ZHAW, 2014.

[2] M. Arnold and D. Egger, Getting Started with PlebML, Winterthur: ZHAW, 2015.

[3] A. McCallum, “MALLET,” University of Pennsylvania, [Online]. Available: http://mallet.cs.umass.edu/topics.php. [Accessed 14 5 2015].

[4] P. D. I. Gurevych, “DKPro Text Classification Framework,” Technische Universität Darmstadt, [Online]. Available: https://code.google.com/p/dkpro-tc/. [Accessed 14 05 2015].

[5] J. Daxenberger, O. Ferschke, I. Gurevych and T. Zesch, “DKPro TC: A Java-based Framework for Supervised Learning,” [Online]. Available: http://anthology.aclweb.org//P/P14/P14-5011.pdf. [Accessed 28 May 2015].

[6] T. P. Jurka, L. Collingwood, A. E. Boydstun, E. Grossman and W. v. Atteveldt, “RTextTools,” 2012. [Online]. Available: http://www.rtexttools.com/. [Accessed 14 05 2015].

[7] Google, “Prediction API,” Google, [Online]. Available: https://cloud.google.com/prediction/. [Accessed 28 May 2015].

[8] “Amazon ML Service,” [Online]. Available: http://aws.amazon.com/de/machine-learning/faqs/. [Accessed 23 05 2015].

[9] “Knime,” [Online]. Available: https://www.knime.org/. [Accessed 23 05 2015].

[10] “scikit-learn,” [Online]. Available: http://scikit-learn.org. [Accessed 23 05 2015].

[11] “Machine Learning Library (MLlib),” Apache, [Online]. Available: https://spark.apache.org/docs/1.2.0/mllib-guide.html. [Accessed 14 05 2015].

[12] F. Rong-En, C. Kai-Wei, H. Cho-Jui, W. Xiang-Rui and L. Chih-Jen, “LIBLINEAR: A Library for Large Linear Classification,” 23 Augsut 2014. [Online]. Available: http://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf. [Accessed 28 May 2015].

[13] “Apache Mahout,” [Online]. Available: http://mahout.apache.org/. [Accessed 23 05 2015].

[14] “Weka 3,” [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/. [Accessed 23 05 2015].

[15] “Gate,” [Online]. Available: https://gate.ac.uk/. [Accessed 23 05 2015].

[16] “Gate,” [Online]. Available: https://gate.ac.uk/sale/tao/splitch19.html#chap:ml. [Accessed 23 05 2015].

[17] J. Attenberg, “Collaborative email-spam filtering with the hashing-trick,” in *Sixth Conference on Email and Anti-Spam (CEAS}*, 2009.

[18] B. Waldvogel, “Liblinear-Java,” [Online]. Available: https://github.com/bwaldvogel/liblinear-java. [Accessed 03 05 2015].

[19] C.-W. Hsu, C.-C. Chang and C.-J. Lin, “A Practical Guide to Support Vector Classion,” Department of Computer Science, National Taiwan University, Taipei 106, Taiwan, Initial version: 2003 Last updated: April 15, 2010.

[20] “csmining,” [Online]. Available: www.csmining.org. [Accessed 10 03 2015].

[21] B. Pang and L. Lee, “Exploiting class relationships for sentiment categorization with respect to rating scales,” in *Seeing stars*, 2005, pp. 115-124.

[22] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng and C. Potts, “Conference on Empirical Methods in Natural Language Processing (EMNLP 2013),” in *Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank*.

[23] “Kaggle - Sentiment Analysis on Movie Reviews,” [Online]. Available: https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews. [Accessed 10 03 2015].

[24] “Kaggle - Challenges in Representation Learning: The Black Box Learning Challenge,” [Online]. Available: https://www.kaggle.com/c/challenges-in-representation-learning-the-black-box-learning-challenge. [Accessed 10 03 2015].

[25] M. Arnold and D. Egger, Developing PlebML: A modular machine, Winterthur: ZHAW, 2015.

[26] K. Lang, “NewsWeeder: Learning to Filter Netnews,” in *in Proceedings of the 12th International Machine Learning Conference (ML95)*, 1995.

[27] “LingPipe,” Alias-i, [Online]. Available: http://alias-i.com/lingpipe/. [Accessed 14 05 2015].

8 Image index

Figure 1 - Schematic of an abstract workflow 10
Figure 2 - Core Module Dependencies 13
Figure 3 - Builder interaction 14
Figure 4 - Visualization of queues for concurrent feature extraction 18
Figure 5 - Incomplete UML of ch.zhaw.init.plebml.vectorspace 19
Figure 6 - UML of ch.zhaw.init.plebml.coreLibrary.postprocessing 21
Figure 7 - Incomplete UML figure of the event bus system 24
Figure 8 - Core Module Dependencies 25
Figure 9 - Preprocessor structuring 26
Figure 10 - Changes to Model Class 33
Figure 11 - Semeval scores of optimizeMLComponent for different solvers and log c. 42
Figure 12 - Spam optimizeOverC MCSVM 43
Figure 13 - Movie review sentiment optimizeOverC 44
Figure 14 - Black box cross validation results 45
Figure 15 - Black box Kaggle submission results 45
Figure 16 - Spam feature extraction on Machine B, SER_DATA_PAR_FET and MEMORY_MAPPED_HASH 46
Figure 17 - Spam feature extraction on Machine B, PAR_DATA_PAR_FET and MEMORY_MAPPED_HASH 46
Figure 18 - Spam feature extraction on Machine B, SER_DATA_SER_FET and HEAP 47
Figure 19 - Semeval feature extraction on Machine B CACHED, CACHED_HASH and SER_DATA_PAR_FET 47
Figure 20 - Semeval feature extraction on Machine B, HEAP 48
Figure 21 - Semeval feature extraction on Machine A, PAR_DATA_PAR_FET and HEAP 48
Figure 22 Spam feature extraction on Machine A, SER_DATA_SER_FET and HEAP 61
Figure 23 Spam feature extraction on Machine B, SER_DATA_SER_FET and HEAP 61
Figure 24 Spam feature extraction on Machine B, SER_DATA_SER_FET and MEMORY_MAPPED_HASH 61
Figure 25 Spam feature extraction on Machine A, SER_DATA_PAR_FET and HEAP 62
Figure 26 Spam feature extraction on Machine B, SER_DATA_PAR_FET and HEAP 62
Figure 27 Spam feature extraction on Machine B, SER_DATA_PAR_FET and MEMORY_MAPPED_HASH 62
Figure 28 Spam feature extraction on Machine A, PAR_DATA_PAR_FET and HEAP 63
Figure 29 Spam feature extraction on Machine B, PAR_DATA_PAR_FET and MEMORY_MAPPED_HASH 63
Figure 30 Semeval feature extraction on Machine A, SER_DATA_SER_FET and HEAP 64
Figure 31 Semeval feature extraction on Machine B, SER_DATA_SER_FET and HEAP 64
Figure 32 Semeval feature extraction on Machine B, SER_DATA_SER_FET and CACHED 64
Figure 33 Semeval feature extraction on Machine B, SER_DATA_SER_FET and CACHED_HASH 65
Figure 34 Semeval feature extraction on Machine B, SER_DATA_SER_FET and MEMORY_MAPPED_HASH 65
Figure 35 Semeval feature extraction on Machine A, SER_DATA_PAR_FET and HEAP 65
Figure 36 Semeval feature extraction on Machine B, SER_DATA_PAR_FET and HEAP 66
Figure 37 Semeval feature extraction on Machine B, SER_DATA_PAR_FET and CACHED 66
Figure 38 Semeval feature extraction on Machine B, SER_DATA_PAR_FET and CACHED_HASH 66
Figure 39 Semeval feature extraction on Machine B, PARALLEL_MID and MEMORY_MAPPED_HASH 67
Figure 40 Semeval feature extraction on Machine A, PAR_DATA_PAR_FET and HEAP 67
Figure 41 Semeval feature extraction on Machine B, PAR_DATA_PAR_FET and HEAP 67
Figure 42 Semeval feature extraction on Machine B, PAR_DATA_PAR_FET and CACHED 68
Figure 43 Semeval feature extraction on Machine B, PAR_DATA_PAR_FET and CACHED_HASH 68
Figure 44 Semeval feature extraction on Machine B, PAR_DATA_PAR_FET and MEMORY_MAPPED_HASH 68

9 Table index

Table 1 - FeatureVectorGenerator method overview 17
Table 2 - FeatureExtractor method overview 18
Table 3 - Memory calculation for the original problem 22
Table 4 - Memory calculation for the adapted problem 22
Table 5 - Machine A description 40
Table 6 - Machine B description 40
Table 7 - Performance comparison of the feature extraction 41
Table 8 - Performance comparison of the machine learning 41

10 Appendix

10.1 Entry 1 – Used Features [1]

This is an Excerpt from the work “Development of a framework for text classification and participation at SemEval.” Additional information can be found in the chapter “Used features” in the referenced work.

Feature	Implemented	Reference
Avg./Min./Max. values over all pre-trained GloVe vectors for all occurring words in a tweet	Yes	
Contiguous and non-contiguous n-grams. POS tag substitution for non-contiguous n-grams instead of the generic wildcard.	Yes	
Contiguous character n-grams 3 to 5	Yes	
Number of words that are all capital letters.	Yes	
The number of occurrences of each part of speech tag	Yes	
The number of hashtags	Yes	
Lexicon feature: The total amount of tokens with an emotion greater than 0	Yes	
Lexicon feature: total score of all tokens in a tweet	Yes	
Lexicon feature: maximal score of a token in a tweet	Yes	
Lexicon feature: the score of the last token in a tweet	Yes	
The number of contiguous punctuation mark (exclamation or question marks)	Yes	
Whether the last token contains an exclamation or a question mark	Yes	
Presence or absence of positive/negative emoticons	No	
Whether the last token (or word) is a positive/negative emoticon	Yes	
The number of elongated words (a character repeated more than 2 times)	Yes	
Presence or absence of tokens from Brown-Clusters generated by the CMU POS-Tagging tool	Yes	
The number of negated contexts	Yes	
The presence of URL or hashtag, one feature each	No	
The presence of a question mark token in the tweet	No	
Feature weighing: If the original token is all upper case, increase the weight of the feature	No	
Feature weighing: If the original token has elongation, increase the weight of the feature	No	
Feature weighing: the token is adjacent to an emoticon. Increase/decrease depends on emoticon	No	
Feature weighing: the score of each token is divided in half if it is in question context	No	
Weighing of a term by ∆BM25 heuristic (Paltoglou and Thelwall, 2010)	No	
Concise Semantic Analysis (Li et al 2011) (Monroy et al 2013)	No	

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

10.2 Entry 2 - Getting Started With PlebML

This chapter is a standalone guide on how to get started with PlebML. It is intended for software devs without machine learning experience and machine learning experts alike. The sections are color-coded depending on the needed expertise.

Green sections deal with general usage of PlebML
Orange sections deal with usage and extension of PlebML requiring knowledge of machine learning.

10.3 What Is PlebML

PlebML is an easily extensible machine learning framework intended to give various levels of interaction needing different levels of machine learning expertise ranging from novice (should know what features are and what training and prediction is) to expert. PlebML is made up of three parts. The Core Library which contains the actual machine learning algorithms and wrapper to machine learning libraries, as well as various utilities, interfaces and abstract classes needed by the second part.
Core Modules provide modular functionality for a specific kind of machine learning tasks such as our stock module for text classification. A Core Module contains the class responsible for a sensible representation of the raw data within the system as well as the actual features and some other things which will be discussed below. Extending and creating these Core Modules is one of the main topics of this guide.
The third and easiest part of PlebML are Tasks. Tasks consist of two parts. One is the handling and importing of the raw data until it can be handed to a Core Module. The second part is interaction with a Core Module's Pipeline and Builders. It is also the topic of the BASIC level chapters in this guide

10.3.1 [BASIC] The Structure of PlebML

PlebML has three key parts:

- **Feature Vector Generator**
  Feature vectors are what PlebML builds from the raw data string you pass into the text classification module. They are a bunch of measurements done on each string and are used by the machine learning part of the framework to do its required calculations. Such measurements could be the length of the string or the occurrence of a particular word when dealing with texts.
  These Feature Vector Generators need to be configured and instantiated. This is done by factories called Builders. These Builders should be part of the core module you're using. In the case of the stock text classification module it's called TextClassificationBuilder.
  Even though you can set every little possible thing on these builders often you might want to try one of the default configurations. They are offered as static methods on the builder class returning an instantiated ready to use builder. You can still modify configurations on the builder after retrieving it with a default configuration method.

- **Machine Learning Components**
  These components are responsible for training, evaluating and prediction. Training needs to be done before the component can be used for prediction on an unknown data entry. Evaluation can be done to measure the performance of a machine learning component and must be done with unused training data for reliable results. For all these uses the machine learning component requires feature vectors. These can be either generated by the above described feature vector generator or can be loaded from an external source and then converted into the needed sparse vector format.

- **Pipelines**
  These objects are the primary and easiest way to interact with PlebML. Every core module should have a basic pipeline that offers easy to use workflows for the entire machine learning process.
  The workflows on these classes will bridge the gap between the other two separate parts of PlebML. Especially if you don't have much expertise in machine learning, using the basic pipelines should be the easiest way to work with PlebML.

10.3.2 [BASIC] Using the Basic Pipeline

If the developer of the core module followed the best practice notes of PlebML, there should be one or several basic pipelines in each module. In the stock text classification module this is called the BasicTextClassificationPipeline and the following examples are explained with this pipeline. To use it you have to do the following things:

1) **Write an Importer:**
Core modules mostly don't come with importers because they vary wildly between each application. Importers read the data that PlebML needs from physical storage and have to implement the Importer interface found in the ch.zhaw.init.plebml.coreLibrary.featureVectorGeneration.importer. There are two methods to be implemented, one to import TrainingData which consists of a raw data object (in the text classification case a String) and an expected prediction result (again, if you're working with our stock module that would be Integer). We recommend an Enum to Integer mapping within your application so each class of text has its own enum label. All the example tasks of PlebML contain an importer package which contain implementations of said interface.

2) **Instantiate the Pipeline:**
Basic pipelines should have two ways to be instantiated. Either with a constructor which requires a Feature Vector Generator or with static method utilizing the default configurations of the Builder. In the text classification module the BasicTextClassificationPipeline offers both of these possibilities.

Constructor:
```
BasicTextClassificationPipeline(FeatureVectorGenerator featureVectorGenerator)
```

Static factory Methods:
```
defaultRomanAlphabetPipeline(String identifier)
defaultEnglishLanguagePipeline(String identifier)
```

The identifiers are very important. You have to keep them constant between training your system and then using it for prediction. They are internally used for indexing purposes.

3) **Training Your Machine Learning Component**

A basic pipeline should offer you one or several methods to get a machine learning component with a decent score performance. Those methods can vary in parametrization and runtime. The BasicTextClassificationPipeline offers the following.

```
findGoodClassifier(OptimizationGoal goal, Importer importer , String... files)
findGoodClassifierBestGuess(OptimizationGoal goal, Importer importer , String... files)
```

The findGoodClassifier method will run a long time as it contains a very large set of search parameters. The findGoodClassifierBestGuess method will run during a lot shorter time but may have worse results it is basically the same than the other method but with a much smaller set of search parameters that commonly result in a good score. Just note that neither of those will result in the *best* classifier.

The OptimizationGoal is another interface which lets you decide how the score is calculated. There are overloads of this methods which use the standard OptimizationGoal of averaging F1-Score over all encountered text classes. In some cases you might want to define different weights for your different textclasses (such as in our spamfilter example) that can be done by specifying an OptimizationGoal.

The Importer is just an instance of the Importer you wrote in step one and the files strings are the physical locations of the training data.

After training is done you probably want to store the ML-component so you don't have to retrain it after each program restart. This method lets you do that. It is based on a working directory within the PlebML folder. So make sure the model name is unique.

```
storeMachineLearningComponent(String modelName)
```

4) **Loading and Using a Trained Machine Learning Component**
After having trained the component it is ready to be used. The two major methods you want to look at are the predict and evaluate method. The predict method is located in the abstract pipeline and works the same for every existing pipeline. The evaluation method however should be implemented in the core module.

```
EvaluationResult evaluate(Importer importer, String... files)
```

The evaluate method works on training data. You have to use training data that you did *not* use in the training process. It will predict each entry in the training data and then compare it to the expected prediction result within the training data files. That lets you measure how accurate your ML-component is.

```
PredictionResult<RawObjectType, PredictionType> predict(RawObjectType data)
```

The predict method is used for data that has no known prediction result. So that is what is used once the system runs and does its intended job. In the case of our text classification result the RawObjectType is String and the PredictionType Integer. With the enum in your Importer you can then map the Integer back to an enum entry.

In most cases training and using the ML-component does not happen during the same runtime or even in the same program. That's why it's possible to store and load ML-components. In the last section we already explained how to store it and here's how to load it again:

Methods:
```
loadMachineLearningComponent(String modelName)
```

Use this if you instantiated the BasicPipeline with a custom builder. Make sure you have the same configuration of builder as when you trained the component and you used the correct identifier when calling the builders make method.

Static factory methods:
```
BasicTextClassificationPipeline loadTrainedEnglishLanguageDefaultPipeline
       (String identifier,
        String modelName)

BasicTextClassificationPipeline loadTrainedRomanAlphabetDefaultPipeline
       (String identifier,
        String modelName)
```

If you used one of the default factory methods on the pipeline you can use the respective static load methods to instantiate the whole pipeline for it again.

10.3.3 [ADVANCED] Using the Advanced Pipeline and the Abstract Pipeline

The methods on advanced pipelines are meant for users with some expertise in machine learning and offer more control over the used parameters in comparison to those on the basic pipelines.

For instance the AdvancedTextClassificationPipeline offers the following method:

```
public void optimizeMLComponentCParameter
        (OptimizationGoal goal,
         LiblinearClassifierSolverType solver,
         double eps,
         double minLogC,
         double maxLogC,
         double cStepSize,
         int nrOfFolds,
         Importer importer ,
         String... files)
```

The class is set to use our implementation of Liblinear as its machine learning component and has some simple algorithms to optimize the C-hyperparameter. Instantiation works similar to the basic pipelines, however there are no default factory methods.

An own pipeline can be easily implemented by extending the core library's AbstractPipeline. This can be useful if you want to use another implementation of ML-component for instance. If you only want to add different workflows extend the AdvancedPipeline in your task package.

10.3.4 [ADVANCED] Configuring the Builder

The Builder represents a configurable factory for the feature vector generator. The builder offers a make-method on an abstract level which will then create the feature vector generator according to its configuration. Usage of the TextClassificationBuilder looks like this:

```
TextClassificationBuilder builder = new TextClassificationBuilder();
builder
    .advancedOptions()
        .preprocessor()
            .addTokenizer(new SuperSimpleTokenizer())
            .addMutation(new SimpleUserNameNormalizer())
            .addMutation(new SimpleURLNormalizer())
            .constructPreProcessorFromFeatureRequirements()
    .advancedOptions()
        .featureExtraction()
            .addFeature(
                        new NGramFeature(),
                        new NGramConfigWrapper(1, Sets.newHashSet()),
                        new NGramConfigWrapper(2, Sets.newHashSet()),
                        new NGramConfigWrapper(3, Sets.newHashSet()),
                        new NGramConfigWrapper(4, Sets.newHashSet()),
                        new NGramConfigWrapper(1, Sets.newHashSet(), ExtractorModes.LEMMA),
                        new NGramConfigWrapper(2, Sets.newHashSet(), ExtractorModes.LEMMA),
                        new NGramConfigWrapper(3, Sets.newHashSet(), ExtractorModes.LEMMA),
                        new NGramConfigWrapper(4, Sets.newHashSet(), ExtractorModes.LEMMA)
                    )
            .addFeatureWithRecommendedConfigs(new NonContiguousNGram())
            .addFeatureWithRecommendedConfigs(new NonContiguousNGramWithPOSTagAsWildcard())
            .addFeature(new NrOfHashtags())
            .addFeature(new NrOfAllCapsToken())
            .addFeature(new NumberOfPOSTags())
            .addFeature(new GloveFeatures())
            .addFeature(new NrOfNegatedContexts())
            .addFeature(new NrOfElongatedWords())
            .addFeatureWithRecommendedConfigs(new LastTokenContainsPunctuation())
            .addFeatureWithRecommendedConfigs(new ContinuousPunctuation())
            .addFeature(new CMUTweetClusterFeature())
            .addFeature(new ScoreTotal())
            .addFeature(new ScorePos())
            .addFeature(new ScoreNeg())
            .addFeature(new LastTokenScore())
            .addFeature(new LastTokenNegScore())
            .addFeature(new LastTokenPosScore())
            .useMemoryStrategy(MemoryStrategy.HEAP)
    .advancedOptions()
        .parallelisation()
            .useParallelisationStrategy(ParallelisationStrategy.SERIAL_DATA_SERIAL_FEATURE)
    .advancedOptions()
        .postProcessor()
            .putPostProcessingStepForClass(ScoreTotal.class, new Sigmoid())
            .putPostProcessingStepForClass(ScorePos.class, new Sigmoid())
            .putPostProcessingStepForClass(ScoreNeg.class, new Sigmoid())
            .putPostProcessingStepForClass(LastTokenScore.class, new Sigmoid())
            .putPostProcessingStepForClass(LastTokenNegScore.class, new Sigmoid())
            .putPostProcessingStepForClass(LastTokenPosScore.class, new Sigmoid())
    .advancedOptions()
        .outputSystemOptions()
            .addSysOutEventListeners()
```

Note the usage of chaining. Each option methods within a section in the advanced options returns the option object again. While every other option is defined within the core library, the preprocessor options vary from core module to core module. As for the stock text classification module the preprocessor consist of three stages.

- Tokenization
  Split the string into one document consisting of paragraphs, sentences and tokens.

- Mutation
  Sanitize the document to reduce sparsity with different normalizers.

- Metadata extraction
  Add Metadata such as POS-Tags or negation scope to tokens.

One exception to the preprocessor settings being implemented on the coremodule is the constructPreprocessorFromFeatureRequirements method. This method will set a flag on the builder to try an abstract dependency crawling on the added feature implementation instances and add the components itself. The used core module has to support this or it will fail.

10.3.5 [ADVANCED] Adding a Feature

Each core module should have a features package with an interface extending the IFeature interface. When adding a new feature, create a new class extending said interface and implement the extract method. The interface in the core module feature package should already put the model object generic in place the second generic determines the class of config object you expect passed to your extract method. In case there are no config objects, use Void.

There are a few notable things when adding a new Feature:

1) **Model Requirements:**

```
@RequiresComponentClass(componentClass = TweetNLPPOSTagger.class)
public class NumberOfPOSTags implements ISentimentAnalysisDefaultVectorFormat<Void>
```

With the @RequiresComponentClass you can instruct the dependency crawling to add a preprocessor component of the given class to the preprocessor.

2) **Accessing the Required Data:**
At least one part of the model object class representing the data after preprocessing should implement the IPreProcessorDataReciever interface, that's were all the metadata is written to.

```
t.getDataFromGlobalComponent(TweetNLPPOSTagger.class)
//t is the IPreprocessorDataReciever
```

This is how you can retrieve the data again. When writing the according preprocessor component you have to put the data in place. For more documentation see the paper references in "From here out"

3) **Writing Feature Vector Entries:**

```
extract(Document modelObj, FeatureVector<Document> vectorToWriteTo, List<Void> configs)
```

This is the signature of the extract method of an IFeature<Document, Void> Implementation. The first parameter is the preprocessed data, the second is the feature vector where you have to put your entries. The vectorToWriteTo very much behaves like a java map, make sure your keys are unique within the feature class or the entry will be overwritten. The last parameter is a list of configurations that was passed in the builder addFeature call.

Adding a Feature can be a very involved task and this is only the simplest case. For a full description look up the paper referenced below.

10.3.6 From Here on Out

Much more is possible with PlebML beyond pure usage of finished core modules. Entire modules can be developed by third party software engineers and whole machine learning engines can be linked into the core library. However this is only a getting started guide that should get you settled with PlebML. For a more in-depth look at the framework, its architecture and how to extend it beyond the scope of this guide, refer to the paper "Developing PlebML: A modular machine learning framework"

10.4 Entry 3 – Feature Extraction Performance Measurements

In this section the performance during the feature extraction is measured for different setups and datasets. In all figures in this chapter the left y-axis means features per second, the right y-axis means feature vectors per second and in the x-axis is the number of processed documents. The feature extraction starts slow, as many features are initialized lazily when they are first used.

10.4.1 Spam

In this setup features for our spam data set are extracted, this dataset is well suited for a high load test as each email may consist of hundreds of lines. On Machine A only the heap memory strategy is tested, as it has only a small network disk with very high access times, on Machine B the MEMORY_MAPPED_HASH and the HEAP strategy are tested. The CACHED and CACHED_HASH strategy were not tested for this dataset, as they are designed for smaller loads

10.4.1.1 SER_DATA_SER_FET

Heap, Machine A

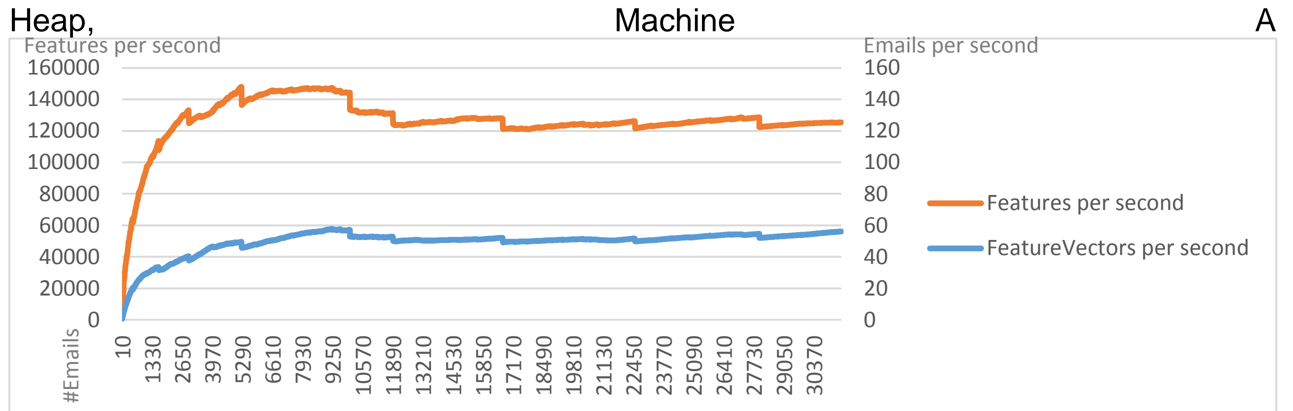

Figure 22 Spam feature extraction on Machine A, SER_DATA_SER_FET and HEAP

Heap, Machine B

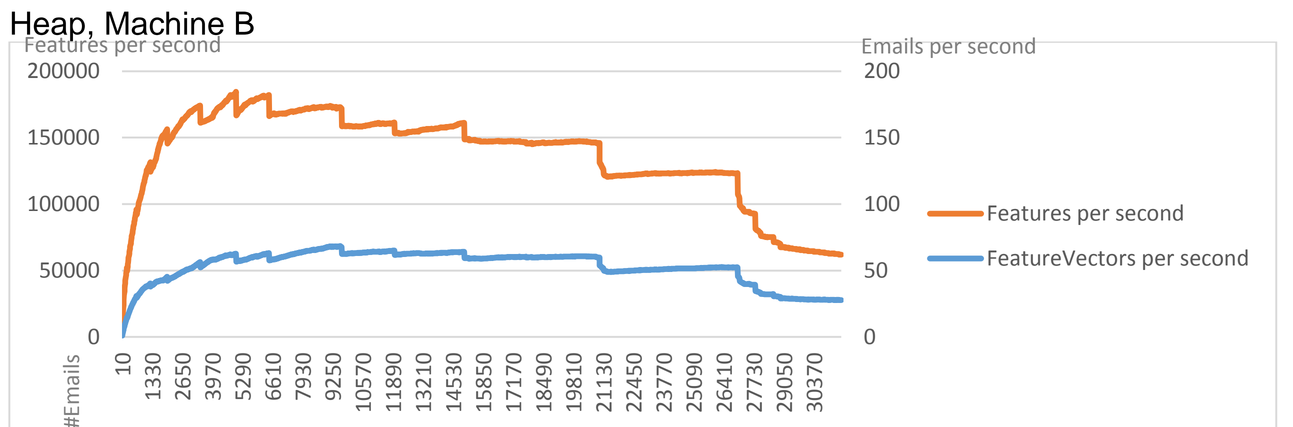

Figure 23 Spam feature extraction on Machine B, SER_DATA_SER_FET and HEAP

Memorymapped, Machine B

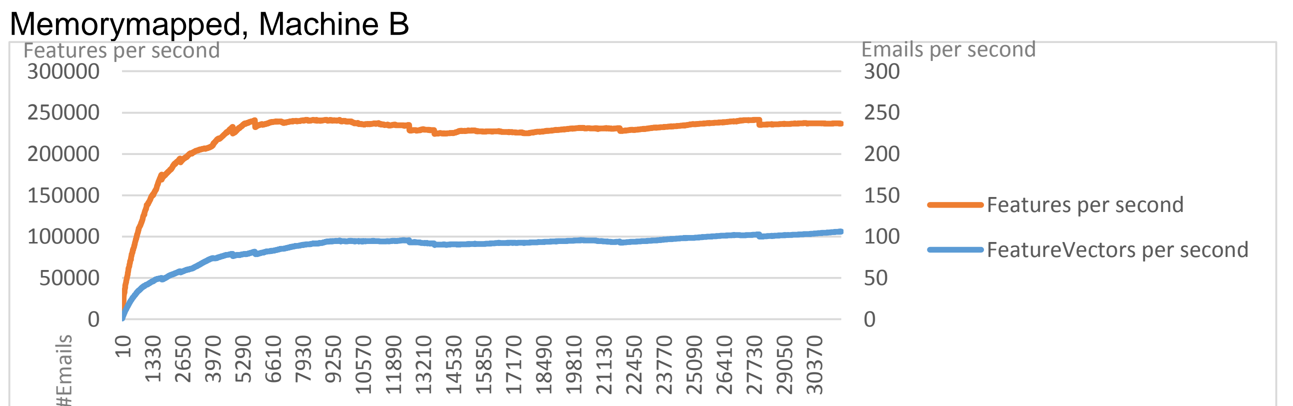

Figure 24 Spam feature extraction on Machine B, SER_DATA_SER_FET and MEMORY_MAPPED_HASH

10.4.1.2 SER_DATA_PAR_FET

Heap, Machine A

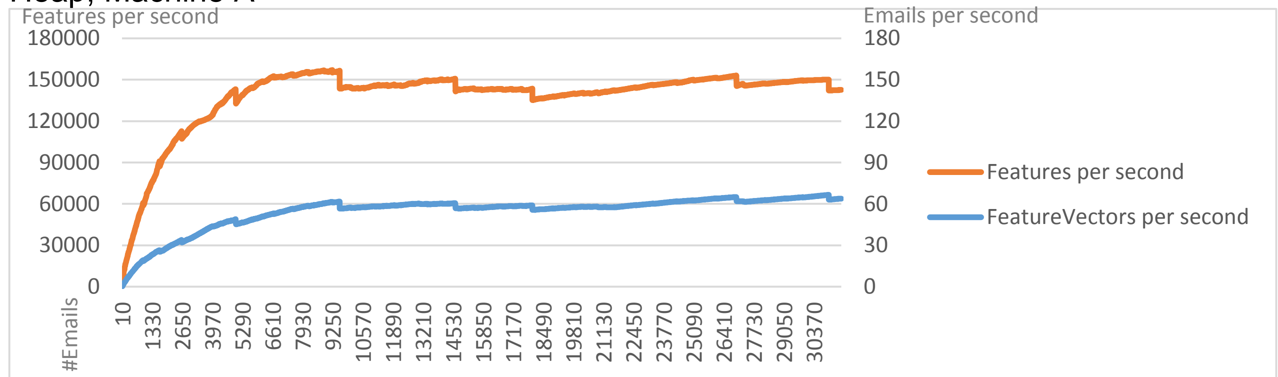

Figure 25 Spam feature extraction on Machine A, SER_DATA_PAR_FET and HEAP

Heap, Machine B

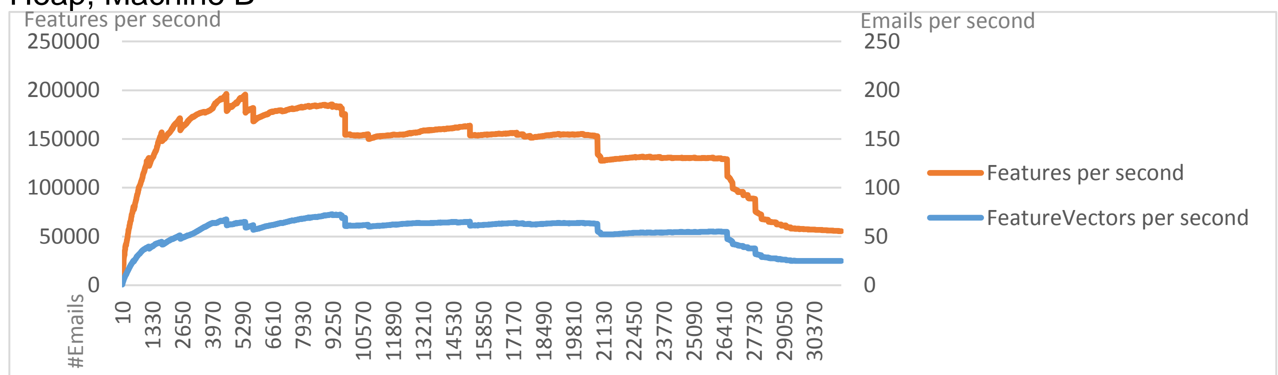

Figure 26 Spam feature extraction on Machine B, SER_DATA_PAR_FET and HEAP

Memorymapped, Machine B

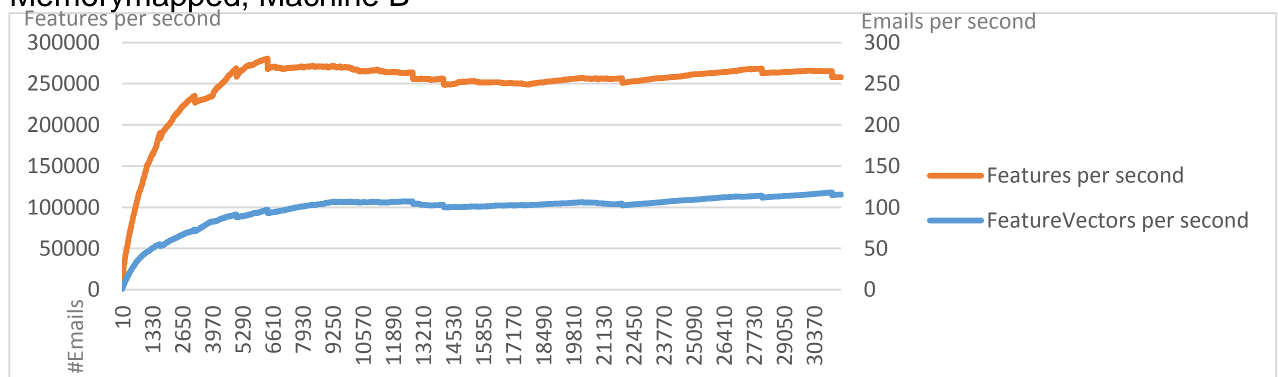

Figure 27 Spam feature extraction on Machine B, SER_DATA_PAR_FET and MEMORY_MAPPED_HASH

10.4.1.3 PAR_DATA_PAR_FET

Heap, Machine A

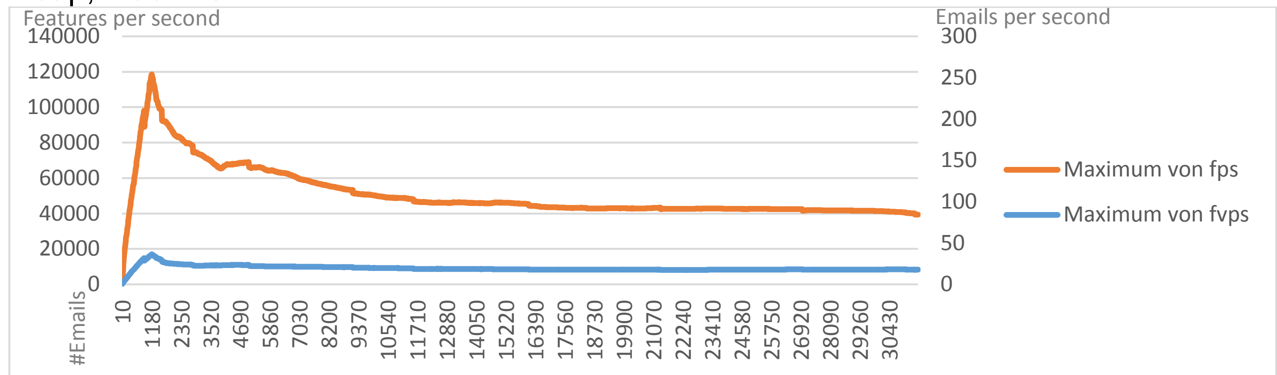

Figure 28 Spam feature extraction on Machine A, PAR_DATA_PAR_FET and HEAP

Heap, Machine B

This experiment did not complete successfully due to an OOME towards the end of the execution.

Memory mapped, Machine B

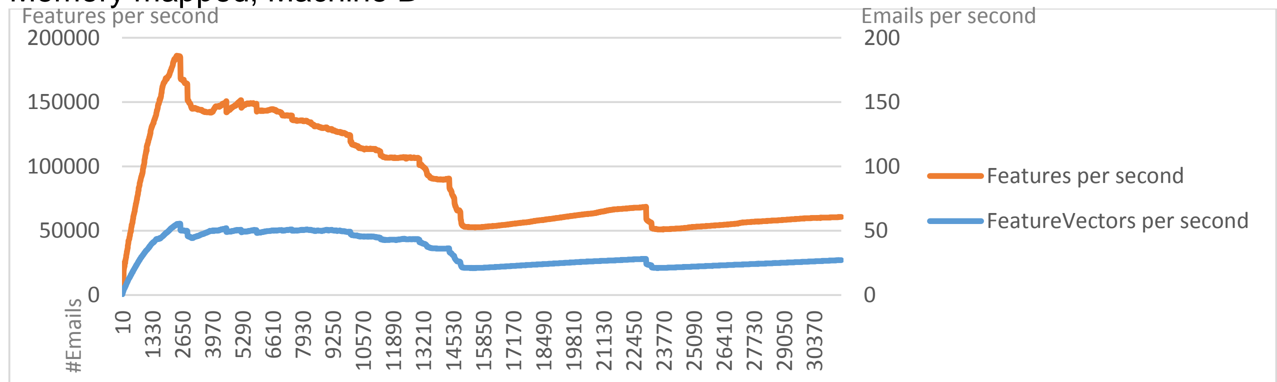

Figure 29 Spam feature extraction on Machine B, PAR_DATA_PAR_FET and MEMORY_MAPPED_HASH

10.4.2 Semeval

In this setup features for the tweet sentiment data set are extracted, this dataset is well suited to test a low load per document, as each tweet only consist of a single line of text. On Machine A only the heap memory strategy is tested, as it has only a small network disk with very high access times, on Machine B all strategies are tested.

10.4.2.1 SER_DATA_SER_FET

In this section the performance of the serial feature extraction, where documents and features are processed serially, is tested with different memory strategies.

Heap, Machine A

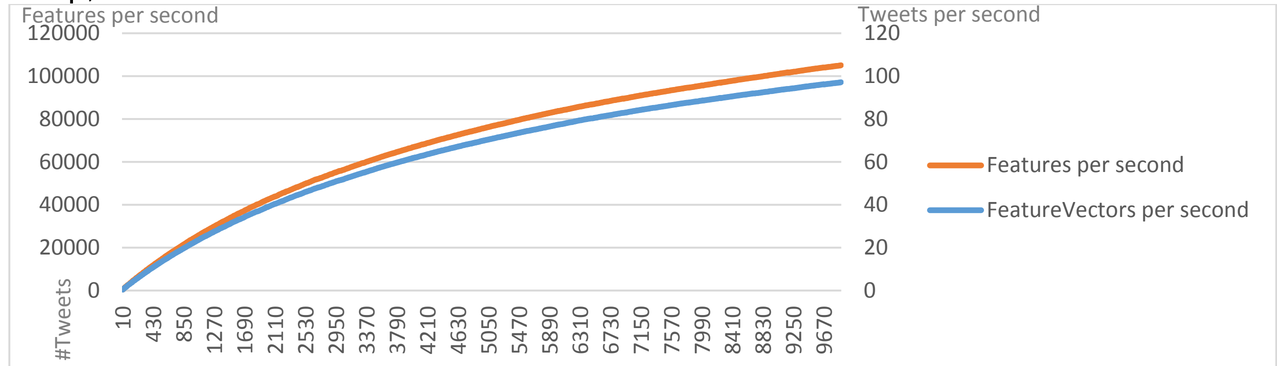

Figure 30 Semeval feature extraction on Machine A, SER_DATA_SER_FET and HEAP

Heap, Machine B

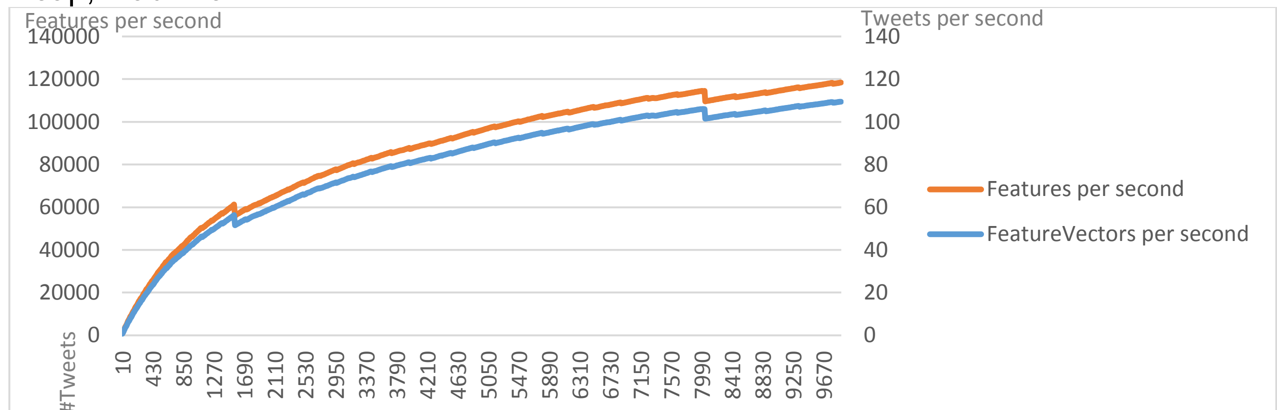

Figure 31 Semeval feature extraction on Machine B, SER_DATA_SER_FET and HEAP

Cached, Machine B

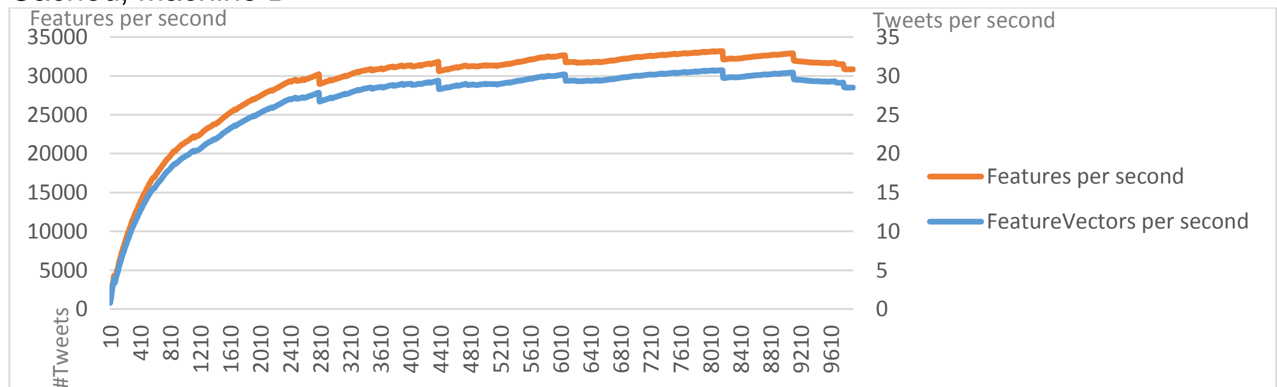

Figure 32 Semeval feature extraction on Machine B, SER_DATA_SER_FET and CACHED

CachedHash, Machine B

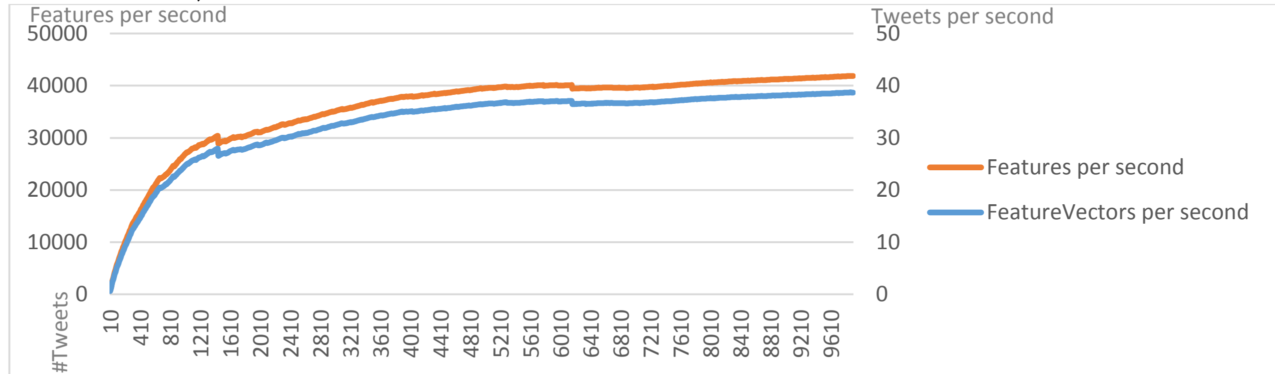

Figure 33 Semeval feature extraction on Machine B, SER_DATA_SER_FET and CACHED_HASH

Memorymapped, Machine B

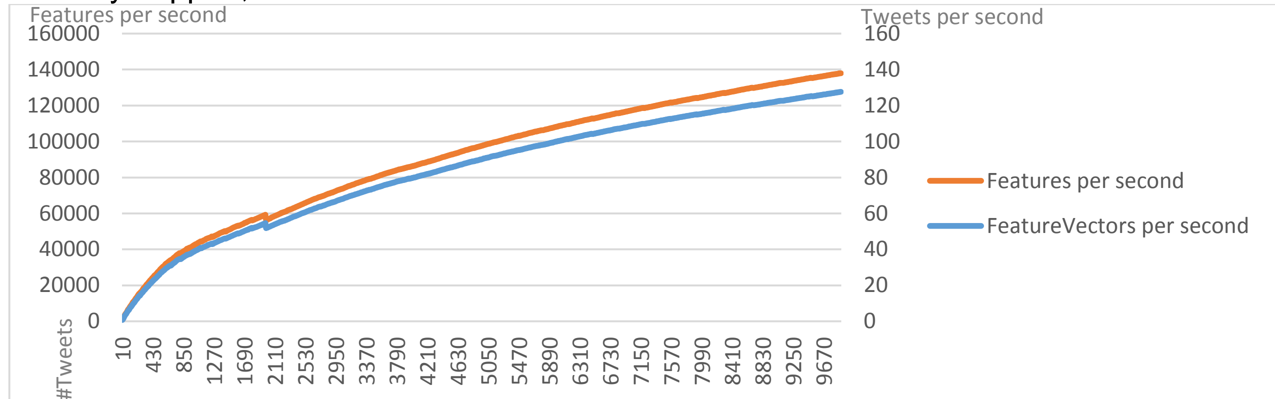

Figure 34 Semeval feature extraction on Machine B, SER_DATA_SER_FET and MEMORY_MAPPED_HASH

10.4.2.2 SER_DATA_PAR_FET

In this section the parallel mid feature extraction, where the documents are processed serially, but all features are applied in parallel, is tested with different memory strategies.

Heap, Machine A

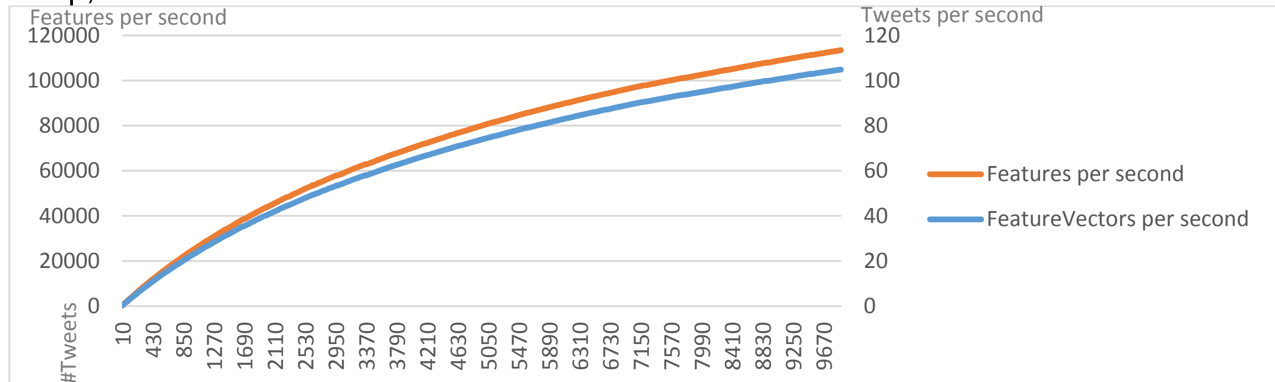

Figure 35 Semeval feature extraction on Machine A, SER_DATA_PAR_FET and HEAP

Heap, Machine B

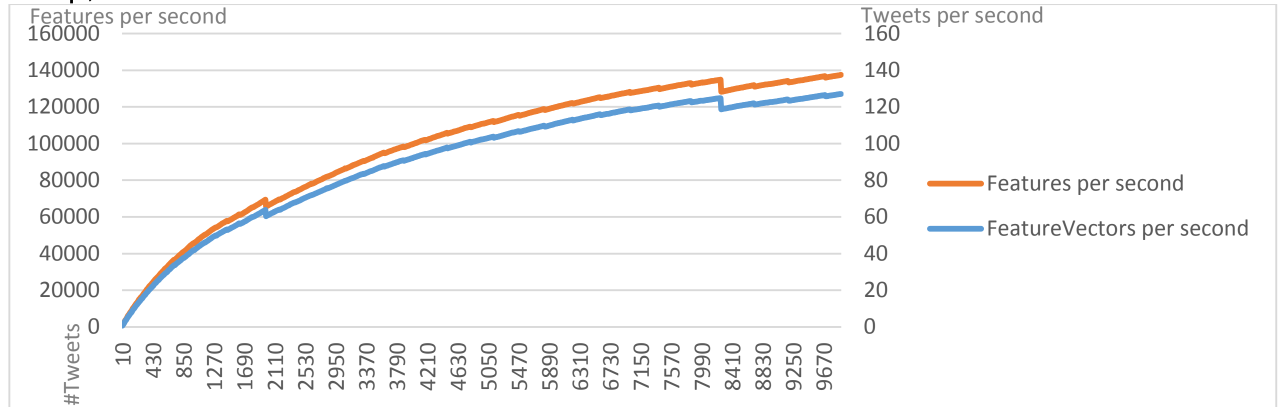

Figure 36 Semeval feature extraction on Machine B, SER_DATA_PAR_FET and HEAP

Cached, Machine B

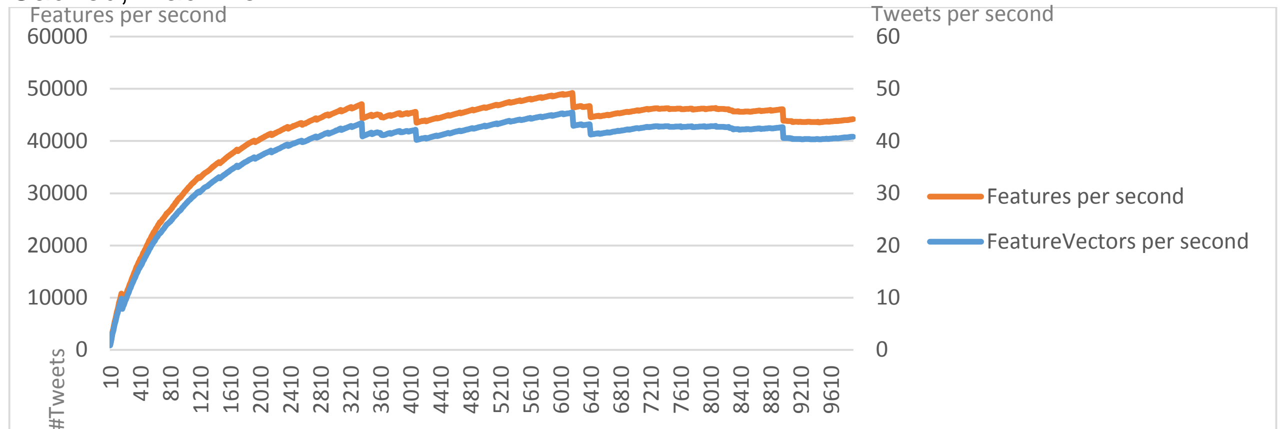

Figure 37 Semeval feature extraction on Machine B, SER_DATA_PAR_FET and CACHED

CachedHash, Machine B

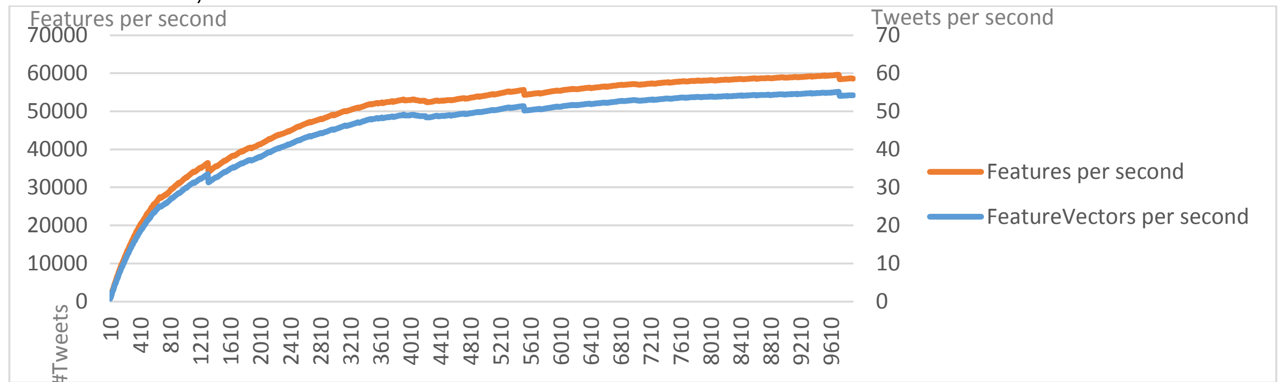

Figure 38 Semeval feature extraction on Machine B, SER_DATA_PAR_FET and CACHED_HASH

Memorymapped, Machine B

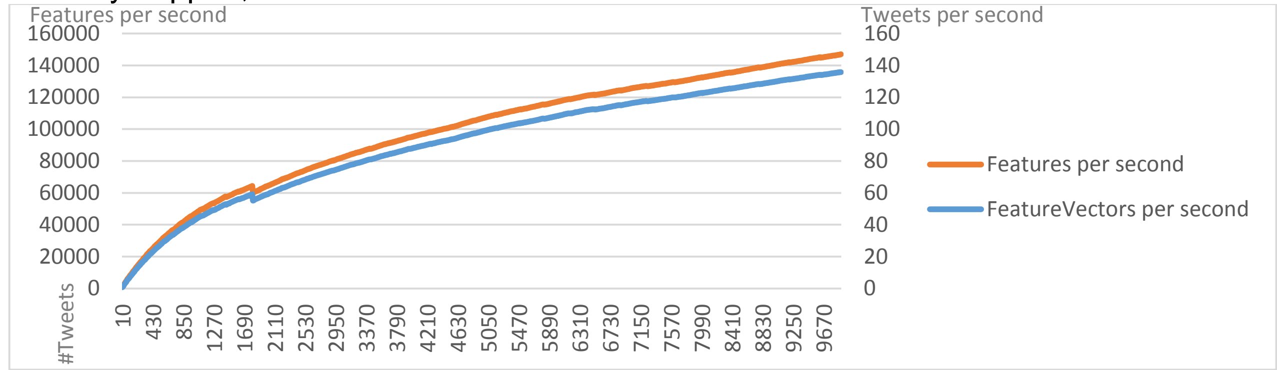

Figure 39 Semeval feature extraction on Machine B, PARALLEL_MID and MEMORY_MAPPED_HASH

10.4.2.3 PAR_DATA_PAR_FET

In this section the parallel max feature extraction, where the documents and features are processed in parallel, is tested with different memory strategies.

Heap, Machine A

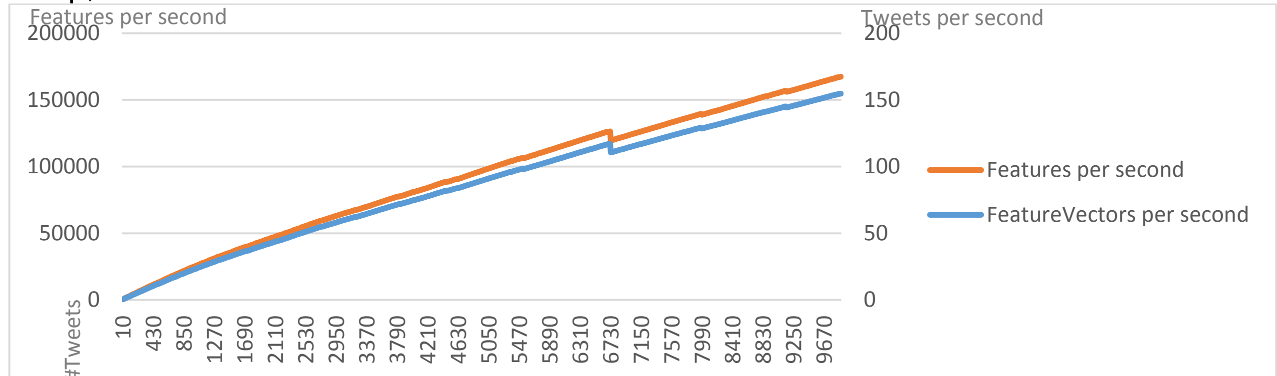

Figure 40 Semeval feature extraction on Machine A, PAR_DATA_PAR_FET and HEAP

Heap, Machine B

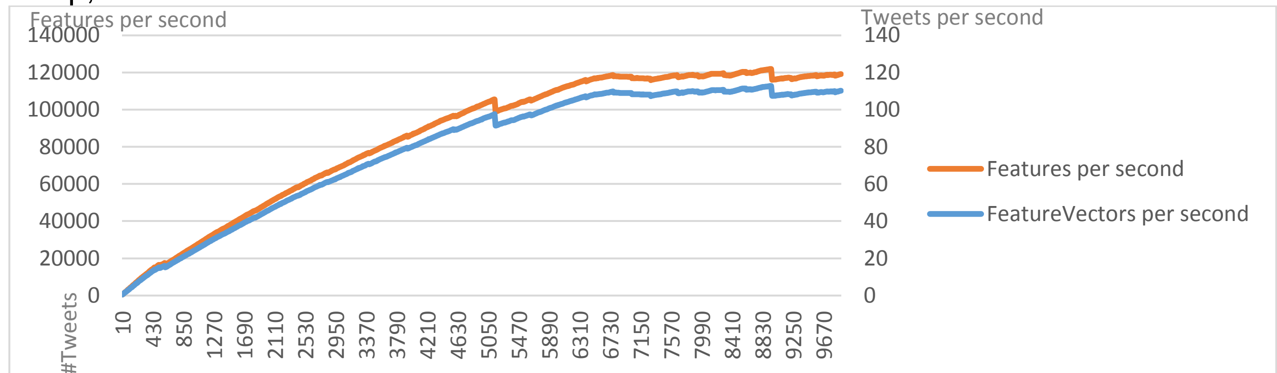

Figure 41 Semeval feature extraction on Machine B, PAR_DATA_PAR_FET and HEAP

Cached, Machine B

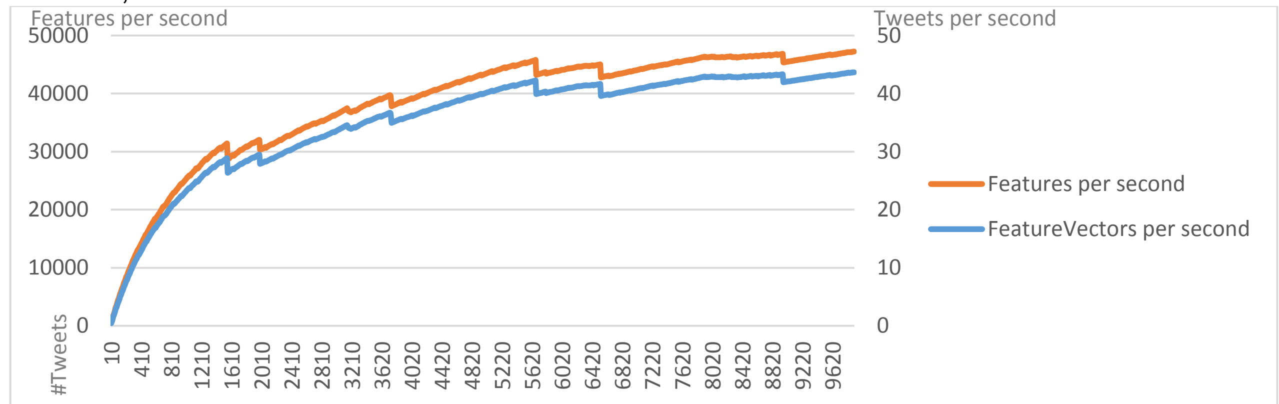

Figure 42 Semeval feature extraction on Machine B, PAR_DATA_PAR_FET and CACHED

CachedHash, Machine B

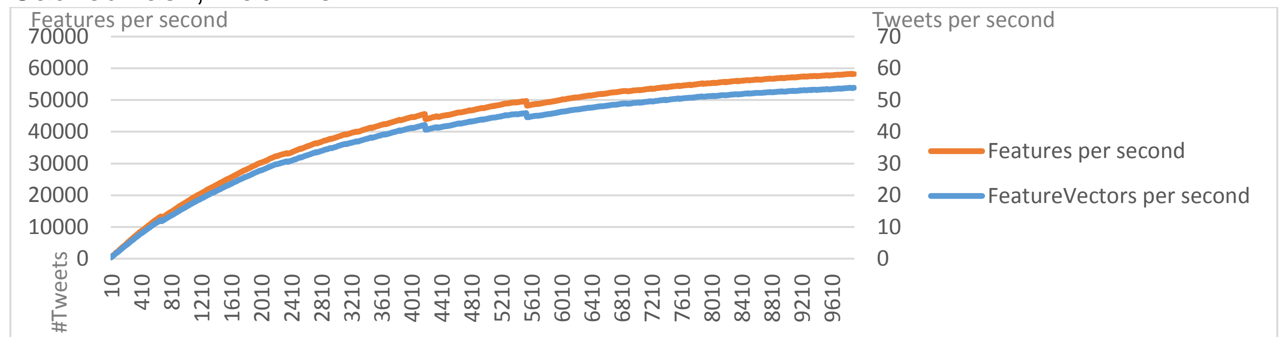

Figure 43 Semeval feature extraction on Machine B, PAR_DATA_PAR_FET and CACHED_HASH

Memorymapped, Machine B

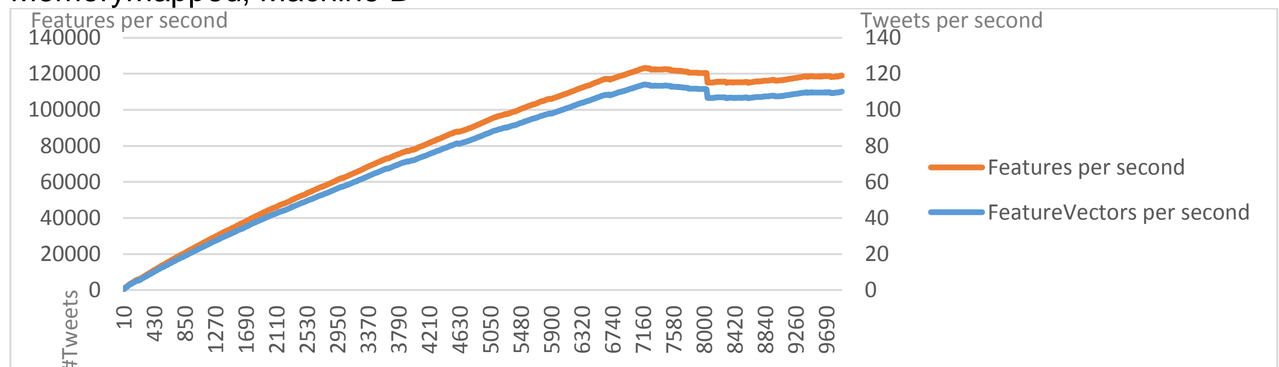

Figure 44 Semeval feature extraction on Machine B, PAR_DATA_PAR_FET and MEMORY_MAPPED_HASH