TECHNISCHE UNIVERSITÄT
CHEMNITZ

Department of Computer Science

Distributed and Self-organizing Systems Group

Master Thesis

Semantic Analysis and Classification of Source Code

Frank Siegel

Chemnitz, 27 September 2017

Examiner: Prof. Dr.-Ing. Martin Gaedke
Supervisor: M. Sc. Sebastian Heil

Frank Siegel
Semantic Analysis and Classification of Source Code
Master Thesis, Department of Computer Science
Chemnitz University of Technology, 27 September 2017

Abstract

Changing user expectations impose new requirements on existing software systems. The constant evolution of technologies and the discontinuation of support for obsolete technologies increase the pressure to renew existing software systems. However, legacy systems, that have been developed over a long period of time contain a huge amount of valuable knowledge about models, rules and solutions of the application domain in their source code. When renewing legacy systems, retaining existing knowledge is crucial. Reverse Engineering tries to extract the knowledge that is implicitly contained in the legacy code base.

While manual reverse engineering is slow, automated reverse engineering seeks to extract knowledge using various methods for code analysis. Knowledge extraction can be considered a three-step process: identification of areas in the code that contain valuable knowledge, classification of the type of this knowledge and generation of a representation of the knowledge. This thesis focuses on automating the first two steps. To achieve this, analysis of source code and comments using suitable artificial intelligence (AI) and natural language processing (NLP) methods is required. The thesis answers questions like how to automatically identify business rules, application domain concepts or area of responsibility of legacy source code.

The objective of this master thesis is to find an approach or a combination of approaches to solve the previously mentioned problem in the context of reverse engineering for software migration based on code analysis. This particularly includes the state of the art regarding automated reverse engineering, code analysis, AI and NLP methods. The demonstration of feasibility with an implementation prototype of the concept is part of this thesis as well as a suitable evaluation on a legacy code base including performance and quality measurements.

Table of Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	Application Programming Interface
ART	Adaptive Resonance Theory
CART	Classification and Regression Trees
CHI	Chi-square Statistic
CNN	Convolutional Neural Network
CSR	Compressed Sparse Row
DF	Document Frequency Thresholding
EL	Expected Entropy Loss
FNN	Feedforward Neural Network
idf	Inverse Document Frequency
itf	Inverse Term Frequency
IG	Information Gain
IP	Internet Protocol
kNN	k-nearest Neighbor
LLSF	Linear Least-squares Fit
LSI	Latent Semantic Indexing
MI	Mutual Information
MIT	Massachusetts Institute of Technology
NB	Naive Bayes

NTC	Neural Text Categorizer
Q&A	Questions and Answers
RBF	Radial Basis Function
RNN	Recurrent Neural Network
rf	Relevance Frequency
SOM	Self-organizing Map
SQL	Structured Query Language
SVD	Singular Value Decomposition
SVM	Support Vector Machine
tf	Term Frequency
TS	Term Strength
UI	User Interface
VC	Vapnik-Chervonenkis

1 Introduction

As Heraclitus already stated in 535 BC „There is nothing more constant than change.“ This also holds true for user expectations on a software system. As the requirements for the system change or new requirements emerge, old technologies are often no longer able to fulfil them. Also, the support for obsolete technologies might be discontinued leading to necessity of change. The software system consequently has to be adapted to a new platform and/or programming language.

Most recently the emergence of cloud computing was an example for such a new technology. Many software companies have migrated or are still migrating their systems to this platform [1]. Another example is the ever-increasing importance of mobile devices. Most companies already acknowledge the importance of mobile applications [2]. Still, a survey from 2016 discovered that 43% of them do not have any mobile application [3]. In order to migrate a system to one of these platforms, the knowledge contained in the existing application needs to be retrieved from its documentation, domain models and source code. The field of reverse engineering intensively focuses on this task.

1.1 Reverse Engineering

A legacy application is a software system, that is critical to daily operations of an organization, but based on obsolete technologies [4]. Over many years of development legacy applications aggregate large code bases [5]. This code implicitly contains big amounts of valuable knowledge about models, rules and solutions of the application domain. To avoid a full rewriting of the legacy application, a way to retrieve and transfer this knowledge needs to be found. However, this has proven to be a very difficult and time-consuming task. Over the last decades many reverse engineering approaches and tools have been developed to support this process.

Many reverse engineering approaches are built upon the documentation of legacy applications and existing domain models [6]. However, in many cases the documentation for legacy applications is either poorly written, incomplete or even not existing at all [5,6]. Thus, a redocumentation needs to be created. Even though there are many tools

for redocumentation, they rely on only structural code analysis [7], examples include Rigi [8], Columbus [9] and COBOL FGM [10]. However, besides the structural information, source code also contains a large amount of semantic information from the application domain. As [11] and [12] have shown, comments and identifier names contain crucial information for the recovery of concepts from source code. This information has been gathered through long requirements engineering and domain analysis processes and acquiring it again would be a very cost-intensive task. Source code knowledge extraction is the process of retrieving structural and semantic information from the source code of a legacy application.

1.2 Source Code Knowledge Extraction Process

Existing redocumentation approaches are based on structural code analysis. Structural code analysis identifies components of the software, such as modules, procedures or subsystems [13]. Current approaches require a human expert to either provide his domain knowledge and technical knowledge to heuristically locate concepts in the source code [14,15] or extract all semantic information manually, only supported by visualizations of the code structure [8,9]. This thesis employs a code analysis process, which extracts both structural and semantic information from the source code. The process of source code knowledge extraction can be divided into three phases: identification, classification and specification.

First of all, similar to structural code analysis, the components of the software system need to be identified. Here components are contiguous code pieces, which contain valuable information about a concept from the application domain. Many programming languages already inherently define a concept of code pieces, which contain strongly related information [6]. In object-oriented programming classes and methods represent such confined code pieces.

Afterwards, in the classification phase, a classifier assigns each code piece to one or more categories, according to the information it contains. These categories can be on different levels of granularity, from specific requirements specifications, such as user stories, to more general areas of responsibility, such as UI or Persistence.

Finally, having the categorized code pieces, the specification phase determines the exact concept expressed in each code piece and generates a representation of this knowledge. This task is similar to, but even harder than the halting problem. Since it is impossible to

automatically determine if a program terminates, it is also impossible to automatically determine what it does. Thus, a human expert is needed to perform the specification given the already classified code pieces.

As manual extraction of the semantic information from source code is a very expensive task, methods to automate the identification phase and classification phase are needed. Rule-based systems, crowdsourcing and machine learning are methods that can be applied for automation of this task. However, machine learning approaches yield several benefits, when compared to the other two methods.

While rule-based systems work well with structured and predictable data, they suffer when encountering unexpected conditions, that their rules do not incorporate. Machine learning methods, however, are able to learn very complex models and generalize from the training data to previously unknown situations [16]. As source code is based on natural human language, it inherits some of its characteristics, like ambiguity and synonymy [12]. Thus, the extraction of knowledge from source code constitutes a very complex task and manually constructed rules are not able to cover all its special cases and exceptions [16]. Additionally, machine learning systems can quickly be adapted to new code bases or programming languages, whereas rule-based systems would require the construction of a new set of rules for each new problem domain.

Crowdsourcing as a standalone approach does not scale well to large code bases, since it does not build any model for identification and classification of code pieces. The larger the legacy code base becomes, the more crowdworker effort is required and thus, the more costs incur. Furthermore, if crowdsourcing is used to identify and classify code pieces, the results might be very noisy and strong quality control is required [17]. Both of these problems can be alleviated using machine learning. Crowdsourcing, however, is often applied to facilitate supervised learning methods by constructing a set of training examples.

1.3 Problem Description

The current approaches to reverse engineering suffer from one of three problems: (a) they require an existing domain model and complete documentation, which often does not apply for legacy systems, (b) they only consider structural information from the source code and consequently are not able to extract the semantic knowledge at all or (c) they require a human expert to extract all the semantic information contained in the

source code, only being guided by visualizations of the code structure or search mechanisms on the code base. This still requires a large amount of manual effort in order to retrieve the semantic concepts and does not scale well to large legacy code bases.

1.4 Use Case

Information systems are software applications, that collect, store and process large amounts of data [18]. These systems are a critical part to the success of a business. Their application areas are manifold, examples are patient management in hospitals, transport fleet management for logistics companies or banking systems. For several decades information systems have been developed as monolithic and isolated applications, which withstood many technology changes. However, recently legacy information systems are faced with several challenges, that they are not able to solve. [19]

More and more use cases require the ability to provide answers to unplanned queries in real-time. However, many legacy information systems still use pre-relational databases like IMS, ISAM or COBOL file system. Even an expert in those data management technologies would need several hours or days to implement a complex query. [5,19]

Another highly topical challenge for information systems is the implementation of cloud-based solutions. Especially the adoption of a distributed database design for legacy information systems is very problematic. Even though in an ideal scenario a change of the database would only affect the database component of the application, such a scenario is shown to be unrealistic. Database logic is usually interwoven with many other application components. [19]

1.5 Scope and Objectives

As stated in section 1.1. this thesis considers the automatic extraction of semantic knowledge from source code in the course of reverse engineering. Therefore, in section 1.2. the 3-step process for source code knowledge extraction has been defined. This process shall be automated using approaches from the field of machine learning. However, as already described, only the identification phase and the classification phase can be

automated. Thus, this thesis focuses on those two phases to support human experts in carrying out the third phase.

Existing machine learning approaches for code categorization and the related field of text categorization will be evaluated. Based on this evaluation, one or more approaches for the automatic identification and classification of code pieces shall be developed and implemented.

The field of program and concept slicing, which tries to extract executable subprograms, is out of the scope of this thesis. Also processes of reengineering and software migration will not be an object of consideration.

1.6 Structure of This Thesis

Section 2 covers the state of the art in the field of automatic source code knowledge extraction. Several requirements for feature selection and classification methods will be defined and based on them existing approaches for code and text categorization will be evaluated. Section 3 outlines the approaches that were developed in the course of this thesis. Here a separation between the identification phase and the classification phase is made. In section 4, details about the implementation of those approaches are described, including an analysis of the used technologies. Finally, section 5 evaluates the performance of the developed approaches on the exemplary code base of SharpDevelop and section 6 comprises the conclusions and potential future work on this topic.

2 State of the Art

This section covers the state of the art in automatic source code knowledge extraction. Here, the research on source code knowledge extraction and its relation to text categorization is described first. Then a set of requirements for machine learning approaches in the field of semantic code analysis are defined, upon which existing approaches from the field of text categorization will be outlined and evaluated based on these requirements. Further, different representation methods for text and various preprocessing methods, such as term weighting, stemming and feature selection, are described.

2.1 Source Code Knowledge Extraction

The knowledge contained in source code consists of structural and semantic information. As stated in section 1.2., structural information is reflected by constructs, that are specific to a programming language, such as classes, modules or functions. This information can easily be extracted by parsers or heuristic methods. For the automatic extraction of semantic information from source code, an appropriate representation of this information needs to be constructed first. Therefore, two main sources of semantic information in code need to be considered: comments and identifier names. As [11] has shown, it is not possible to assess the computational intent of a program without this semantic information. While the name of an identifier contains knowledge about roles and properties of the objects they describe, comments are often used to convey information about a statement, function or module [20]. In [12] Caprile and Tonella investigated the language used in comments and identifier names. Through their research they found many similarities between source code and natural language. Comments and identifier names use a subset of natural language and often depict elliptic sentences [20]. Identifier names are usually composed of multiple words, combined either by underscores (e.g. some_variable) or capitalization (e.g. someVariable) [6].

The academic literature on source code knowledge extraction is sparse. Most of the existing approaches focus either on the detection of malware or security issues in the

source code [21-23] or the classification of whole programs, often in the context of organizing software repositories [24-26]. These approaches usually use machine learning approaches from the field of text categorization.

Text categorization comprises the automatic assignment of natural language documents to predefined categories [27,28]. For doing so, a document needs to be divided into a set of terms, which can include single words, word phrases or other word groupings based on thesauruses or statistical measures [29]. Based on the term representations of a set of training examples, machine learning classifiers can build a model of the training data. This model is then used to predict the category of an input document or to build clusters of similar documents. Several decades of extensive research in the field of text categorization have applied and compared many different machine learning methods.

Machine learning approaches for text categorization can generally be divided into supervised and unsupervised methods. Unsupervised methods try to find patterns in a set of unlabeled training examples and detect the presence those patterns in a new input document. On the other hand, supervised methods learn a classification or regression model based on a set of labeled training examples, which are already assigned to one or more categories [30]. Training examples, that belong to a category are also termed "positive examples", in contrast to the "negative examples" outside of this category.

2.2 Requirements for Machine Learning Approaches

In this section, a set of requirements is defined, that will be used to evaluate the suitability of different machine learning approaches in the context of source code knowledge extraction. Here, the MoSCoW method is applied to prioritize the different requirements.

R1: Model complexity

In section 1.1., we outlined that understanding source code and extraction knowledge from it is a very difficult task. To accomplish this task with a machine learning method, the underlying model needs to be sufficiently complex. Generally, machine learning methods can be divided in linear and nonlinear models. Research in text categorization has shown, that the problem of understanding text is often not linearly separable [24]. Thus, linear models are not able to fully solve this problem.

While nonlinear models might not be able to fully learn the characteristics of code pieces and their categories, they can still achieve good results. Consequently, this requirement is put in the Should category of the MoSCoW method.

The model used for source code knowledge extraction should be nonlinear.

R2: Robustness against overfitting

Overfitting is a very common problem in the field of machine learning. If an overly complex model is learned on a small set of training examples, it tends to specialize on this particular training set and disregards the underlying distribution of input data. Consequently, while this model performs very well on the training examples and has a minimal training error, it is not able to correctly classify new input documents from the same distribution. This leads to a large generalization error and a practically unusable classification model.

As overfitting largely depends on the training data, even a non-robust machine learning method does not always produce an unusable classifier. Thus, this requirement is assigned to the Should category of the MoSCoW method.

A machine learning method should prevent overfitting or allow measures to minimize its impact.

Note that the required amount of labeled training data for a supervised learning algorithm depends on both the complexity of the classification model and the inherent robustness of the learning algorithm. Classification methods, that are more complex and less robust against overfitting, require a large amount of training examples to achieve a good generalization. As collections of already labeled examples are hard to find and manual labeling is a very expensive task [31], a classifier that can learn effectively with small amount of training examples is preferable.

R3: Context sensitivity

Natural language text contains large amounts of contextual information. For instance, the word “set” can have many different meanings, depending on the presence or absence of other words in the text. If it is used along with the words “element” or “contain”, it is probably meant as a mathematical set. On the other hand, if words like “tennis” or “points” occur in the same text, the set is probably part of a tennis game. Furthermore,

the order of words in a natural language text has significant importance [28]. For example, consider the sentences “That is what he cooks for his parents” and “That is what he cooks his parents for”. However, many machine learning models either completely disregard word cooccurrences or only consider word cooccurrences while ignoring word order. Consequently, they lose information that is based on the context a word occurs in.

Context sensitivity is a very desirable property for machine learning methods in source code knowledge extraction. However, even a machine learning model, that is not context sensitive by design, can learn some contextual information when trained with n-grams. This requirement is hence assigned to the Should category of the MoSCoW method.

The machine learning model should be able to incorporate word cooccurrences and word order from its training examples.

R4: Phase-specific applicability

In the identification phase, the components of a legacy application need to be determined. Components can be syntactic units of the programming language, like functions or classes, but also other contiguous code pieces, like a set of statements describing a specific mechanism. An appropriate machine learning method needs to be able to construct a classifier, which can separate code pieces, that represent a component, from other, irrelevant code pieces. However, while it is easy for a human expert to find examples of relevant code pieces, finding a set of examples that describe all kinds of irrelevant code pieces is very hard. Thus, it is necessary to find machine learning methods that can construct classifiers only based on positive training examples. This is the problem of one-class classification.

Since the ability to perform one-class classification is absolutely mandatory for the identification phase, this requirement is put in the Must category of the MoSCoW method.

A machine learning approach must allow one-class classification to be applicable to the identification phase.

As described in section 1.2. there is a large variety of categories that can be used in the classification phase. There can also be code pieces that belong to multiple categories sim-

ultaneously. The function in Listing 2.1 can belong to either “Formatting”, “UI” or “Utilities”. This requirement assesses the ability of a machine learning method to distinguish between many categories and to assign a code piece to multiple categories at once.

Due to the large number of categories, that need to be distinguished in the classification phase, this requirement is also assigned to the Must category of the MoSCoW method.

To be applicable for the classification phase, a machine learning approach must be able to distinguish between many classes and assign a code piece to multiple categories simultaneously.

```
public string getRelativeDateTime(DateTime date)
{
    TimeSpan ts = DateTime.Now - date;
    if (ts.TotalMinutes < 1)
        return "just now";
    if (ts.TotalHours < 1)
        return (int) ts.TotalMinutes == 1
            ? "1 Minute ago"
            : (int) ts.TotalMinutes + " Minutes ago";
    if (ts.TotalDays < 1)
        return (int) ts.TotalHours == 1
            ? "1 Hour ago"
            : (int) ts.TotalHours + " Hours ago";
    if (ts.TotalDays < 7)
        return (int) ts.TotalDays == 1
            ? "1 Day ago"
            : (int) ts.TotalDays + " Days ago";
    if (ts.TotalDays < 30.4368)
        return (int) (ts.TotalDays/7) == 1
            ? "1 Week ago"
            : (int) (ts.TotalDays/7) + " Weeks ago";
    if (ts.TotalDays < 365.242)
        return (int) (ts.TotalDays/30.4368) == 1
            ? "1 Month ago"
            : (int) (ts.TotalDays/30.4368) + " Months ago";
    return (int) (ts.TotalDays/365.242) == 1
        ? "1 Year ago"
        : (int) (ts.TotalDays/365.242) + " Years ago";
}
```

Listing 2.1: Code piece that can belong to multiple categories

R5: Training time

The understanding of source code is a very difficult task and consequently requires a complex model. However, the training time of a machine learning model usually also scales up with its complexity. This is further influenced by the amount of training examples and the dimensionality of the input representation, both of which are large for the classification of source code. With all those factors included, the training time of a machine learning classifier can quickly become unreasonable.

This requirement measures the time needed for training a model through the big-O notation of the learning algorithm. Here, the polynomial classes constant, linear, quadratic, cubic and higher than cubic are distinguished. Since the training set size N and the input dimensionality D usually dominate other parameters, like the number of categories m, only those two parameters are taken into consideration. Since long training times are one of the most limiting factors in machine learning, this requirement is assigned to the Must category of the MoSCoW method.

To be suitable for source code knowledge extraction, a machine learning method must have a training time of at most cubic complexity.

R6: Prediction time

Prediction is the assignment of categories to an input document using an already trained machine learning classifier. As this process happens very frequently, a classifier needs to be able to conduct it very quickly. Often the results are even required in real-time as a human expert is using the classifier in an interactive task.

A trained machine learning model should be able to predict the category of an input document in a short amount of time. This is measured by means of the big-O notation of a prediction algorithm, similar to the requirement R5. However, while a training time of quadratic or cubic complexity might still be acceptable, the real-time requirements for prediction render every algorithm above linear complexity very troublesome. Thus, this requirement only distinguishes between the polynomial classes linear and higher than linear and is put in the Must category of the MoSCoW method.

An appropriate machine learning method must be able to predict new input documents in linear time.

R7: Learning with high-dimensional input spaces

In the classification of natural language texts, the input space tends to be very high-dimensional, up to hundreds of thousands or even millions of terms [27,32]. This also holds true for the classification of source code, as it is based on natural language. Even if feature selection is applied to the input space, a useful feature set still contains hundreds to thousands of dimensions. Some machine learning approaches are not able to incorporate this many input dimensions.

As the impact of high-dimensional input spaces can be limited by the application of feature selection, this requirement is put in the Should category of the MoSCoW method.

An appropriate machine learning method should be able to effectively learn on a high-dimensional input space.

R8: Interpretability by humans

In some situations, a human expert needs to understand the learned classification model. For example, some input documents might be classified wrongly and thus the machine learning model needs to be adapted. This requires the human expert to find out which underlying rules the classifier has learned. Therefore, he needs to ascertain the features that were most important for the wrong classification.

The understandability and interpretability of a classifier by human experts does not impact the prediction quality and is hence assigned to the Could category of the MoSCoW method.

Thus, the machine learning method could produce a model, that is interpretable by humans.

R9: Robustness against noisy data

The quality of a classification model depends strongly on the quality of the training examples. However, a training set usually contains a proportion of wrongly classified or uncharacteristic examples, which is referred to as “noise”.

Both the usage of high-quality training sets and the application of feature selection can strongly reduce the amount of noisy data. Thus, this requirement is only assigned to the Should category of the MoSCoW method.

Machine learning methods should automatically filter out these noisy examples and minimize their impact on the classifier.

R10: Ability to work with representation vectors

The most common representation of text in machine learning is a numerical vector. Here, each term, that occurs in the set of training examples, is represented by one dimension of the input vector [33]. The numerical values for a single training example are based on the number and distribution of occurrences for each term. For large training sets, this leads to sparse input vectors, as a training example is likely to contain only a fraction of the training set terms [34]. However, some machine learning methods have shown to be susceptible to sparse input vectors [33,35].

The application of feature selection methods can also alleviate the sparsity of the input vectors, as rare terms are commonly removed. Hence, this requirement is put in the Should category of the MoSCoW method.

A machine learning method should be able to work with sparse numerical input vectors or utilize other text representation methods, that do not produce sparse vectors.

R11: Incremental learning capability

In the process of source code knowledge extraction, the training set might increase over time as new source files or code bases are added. However, provided an already trained model, not all machine learning methods allow to easily extend the training set. The only other option would be to discard the existing model and relearn it from scratch. This would again consume a lot of time for large training sets and complex classification models.

Since this requirement does not impact the prediction quality of a machine learning model, it is assigned to the Could category of the MoSCoW method.

Thus, an appropriate machine learning method could allow to include new training examples into an already trained model.

2.3 Representation Methods in Text Categorization

To process text with machine learning methods, it needs to be transformed into an appropriate representation first. The most common representation form for text is a numerical vector, where each term is represented by one dimension [33]. This is also called bag-of-words, indicating that the order of terms is not maintained in this representation form. However, since the meaning of a term is strongly dependent on its context, much semantic information from the text is lost with this representation method

One approach that tries to solve this problem comprises the selection of n-grams as additional dimensions in the numerical vector. Depending on the choice of n, this includes some of the context a term occurs in. A downside of this method lies in the strong increase of the input dimensionality, which in turn increases training time. Further, most of the added n-grams contain very little information and introduce unnecessary redundancy.

For the use of a convolutional neural network, the authors of [28] have introduced another representation form, that includes the context of a term. Here, each term is represented by a one-hot vector and the input vector comprises sequences of those one-hot vectors (see Figure 2.1). As this representation form also leads to a very high dimensionality, a variant of this method merges the one-hot vectors inside each sequence. Even though contextual information can be preserved with this representation of text, it is only designed for the use of a convolutional network and cannot be applied to other machine learning approaches.

		Some	Example	Input
	Some	1	0	0
	All	0	0	0
Features	Example	0	1	0
	Input	0	0	1
	Have	0	0	0

Figure 2.1: Sequence of one-hot-vectors for an example input

Another approach in [33] solves the problem of sparse input vectors, which are problematic for several machine learning methods, such as SVMs and neural networks, and additionally cope with the problem of high dimensionality. Therefore, a text is represented as a vector of strings instead of a numerical vector. The dimensions of the input vector in this method do not correspond to single terms, but to predefined features of the text. Features can be either linguistic, such as the first noun or verb in the text, statistical, such as the highest frequent words, or positional, such as the last word in a sentence or a random word in the first paragraph. This representation method is also strongly associated with a special neural network, the neural text categorizer (NTC), and cannot be easily used with other machine learning approaches.

2.4 Preprocessing in Text Categorization

The most common representation form for text is a numerical vector. To increase prediction quality and training time of a machine learning algorithm, several preprocessing methods are commonly applied to numerical vectors. This includes stemming, feature selection and term weighting.

2.4.1 Stemming

Many derivations from a term, such as plurals or different verb tenses, contain almost equal information to their base term. However, they are usually treated as different dimensions in a numerical vector. Thus, stemming can be used to reduce all terms to their base form (e.g. “stood” to “stand” or “industries” to “industry”). Still, the usefulness of stemming is very controversial. While it has improved classification quality on some data sets, other experiments have shown no difference with or without stemming or even a decrease in classification quality [36]. Thus, this thesis will not further consider stemming.

2.4.2 Feature Selection

As described in section 2.2, numerical term vectors in text categorization are very high-dimensional, up to hundreds of thousands or even millions of terms [27,32]. An input space with this many dimensions is infeasible for many classification methods. Feature

selection methods try to find a smaller set of terms or combinations of terms that achieves a good classification quality. Often the use of feature selection has even shown to increase classification quality when compared to the original term vector, as irrelevant and potentially noisy terms are removed. The lower dimensionality of input vectors allows faster learning of a classifier and reduces the required amounts of memory. Thereby feature selection also benefits those classifiers, that would be able to work with the original high-dimensional input vectors. [37]

Feature selection can be conducted either by removing non-informative terms based on statistical measures on the training set or by constructing new features as a combinations of lower level features, such as single terms, and thereby constructing orthogonal dimensions on a higher level [27].

Term removal methods

Feature selection methods that perform term removal can further be divided into wrappers, embedded methods and filters [37].

Wrappers search on the space of all possible subsets of terms trying to find a suitable subset. Therefore, a classification method, that guides and halts the search, needs to be defined. This should be the same method that is used later on to train a classifier. For large input vectors, wrappers require an immense amount of processing power. Part of this problem can be overcome by using an appropriate search strategy, such as coarse or greedy strategies. [37]

Embedded methods are dependent on specific classification methods, that are able to extract features during their learning process. Examples for such methods are decision trees, the SVM method by Weston et. al and LASSO regression. Again, the resulting feature set is specific to the used classification method and does not work well with other methods. [37]

Filters are the most common form of term removal methods [27,37]. They calculate the value of a scoring function for each term and select the terms with the highest scores as features. Filter methods incorporate several advantages when compared to wrappers and embedded methods. A filter method is independent from the classification method used later on. Once a set of features is constructed, it can be used on different classifiers and allows the easy application of cross-validation. Also, filters are much faster than wrappers and embedded methods. [37]

In the following, several scoring functions will be outlined. Probabilities are generally estimated based on the training set. For instance, $P(t)$ is calculated as the number of training examples, that contain term t, divided by the total number of training examples. All such estimations can be done in linear time.

Document frequency thresholding (DF) is the simples of those functions. Similar to the inverse document frequency, it measures the number of training examples in which a given term occurs. All terms that are above a defined threshold are removed from the input vectors.

$$DF(t) = \sum_{i=1}^{N} \begin{cases} 1, & t \in d_i \\ 0, & t \notin d_i \end{cases}$$

where d_i is the i-th document and N is the number of training examples.

Since the document frequency can be computed in $O(N)$, document frequency thresholding scales well to large training sets. [27]

Information gain (IG) determines the number of bits of information that is obtained by knowing if a term is absent or present in a given document. Terms with an information gain, that is lower than a defined threshold are removed from the input vectors.

$$IG(t) = -\sum_{i=1}^{m} P(c_i) \log P(c_i) + P(t) \sum_{i=1}^{m} P(c_i|t) \log P(c_i|t) + P(\bar{t}) \sum_{i=1}^{m} P(c_i|\bar{t}) \log P(c_i|\bar{t})$$

where c_i is the i-th category and m is the number of categories.

The calculation of the information gain for a term can be done in $O(D * m + N)$. [27]

Mutual information (MI) assesses how strongly a term t is associated with category c_i [27]. This requires the estimation of the mutual information criterion for each category as follows:

$$MI(t, c_i) = \log \frac{P(c_i|t) * N}{P(t) * P(c_i)}$$

where c_i is the i-th category and N is the number of training examples.

The score of a term can then be computed by building either the average or maximum of all its mutual information criterions:

$$MI_{avg}(t) = \sum_{i=1}^{m} P(c_i)\, MI(t, c_i)$$

$$MI_{max}(t) = \max_{i=1..m} MI(t, c_i)$$

where c_i is the i-th category and m is the number of categories

This leads to a complexity of $O(D * m + N)$. A disadvantage of this scoring function is the lack of intrinsic normalization, as rare terms tend to have higher scores than common terms, even when distributed equally. [27,37]

In contrast to the mutual information function, the **chi-square statistic** (CHI) measures the independence of a term t and category c_i. Here too, a term-goodness criterion needs to be estimated for each category as follows:

$$\chi^2(t, c_i) = \frac{N * (P(c_i|t) * P(\overline{c_i}|\bar{t}) - P(c_i|\bar{t}) * P(\overline{c_i}|t))^2}{P(c_i) * P(\overline{c_i}) * P(t) * P(\bar{t})}$$

where c_i is the i-th category and N is the number of training examples.

The chi-square score is then computed as either the average or maximum over all term-goodness criterions for term t:

$$\chi^2{}_{avg}(t) = \sum_{i=1}^{m} P(c_i)\chi^2(t, c_i)$$

$$\chi^2{}_{max}(t) = \max_{i=1..m} \chi^2(t, c_i)$$

Here, the computation complexity is again $O(D * m + N)$. Intrinsic normalization is applied between the scores inside each category and thus makes them comparable. However, the chi-square function has shown to be inconsistent for terms with a low frequency. [27]

The **expected entropy loss** (EL) determines the discrimination power of a term for a given category by scoring terms higher, if they predominantly occur either inside of the category or outside of it. Therefore, three entropy values need to be computed for the given category: the prior entropy $e(c_i)$, the posterior entropy for documents that contain a given term $e_t(c_i)$ and the posterior entropy for documents that do not contain this term $e_{\bar{t}}(c_i)$.

$$e(c_i) = -P(c_i) * \log P(c_i) - P(\bar{c}_i) * \log P(\bar{c}_i)$$

$$e_t(c_i) = -P(c_i \mid t) * \log P(c_i \mid t) - P(\bar{c}_i \mid t) * \log P(\bar{c}_i \mid t)$$

$$e_{\bar{t}}(c_i) = -P(c_i \mid \bar{t}) * \log P(c_i \mid \bar{t}) - P(\bar{c}_i \mid \bar{t}) * \log P(\bar{c}_i \mid \bar{t})$$

where c_i is the i-th category.

These three entropy values are used in the computation of the expected entropy loss as follows:

$$E(t, c_i) = e(c_i) - e_t(c_i) * P(t) + e_{\bar{t}}(c_i) * P(\bar{t})$$

Computing the expected entropy loss of a term can be done in $O(N)$, since it only uses estimated probabilities. [24]

Term strength (TS) measures the likeliness that a term appears in "closely related" documents. Usually two documents are considered "closely related", if the cosine value of their numerical vectors is above a defined threshold. So, unlike DF, IG, MI, CHI and EL, term strength is based on document clustering.

$$s(t) = P(t \in y \mid t \in x)$$

where x and y are two distinct closely related documents.

The calculation of the term strength can be done in $O(N^2)$. [27]

A variety of other functions, such as the quadratic Pearson correlation coefficient, Fisher's criterion and the T-test criterion, have been used for feature selection [37]. However, this thesis will not cover them in detail, as they are not used as commonly as DF, IG, MI, TS and EL.

Feature construction methods

In natural language, different terms can be related through semantic characteristics, such as synonymy and polysemy. However, term removal methods treat each term as a separate dimension and thus, cannot model these characteristics. Feature construction methods address this problem by building new features as a combination of multiple terms. Thereby, semantically related terms can be grouped in a newly constructed feature. [38]

The most commonly used approach for feature construction is **Latent Semantic Indexing** (LSI). This method analyses the correlations between different terms in the training set and creates a set of orthogonal feature dimensions based on these correlations. To

perform this transformation, singular value decomposition (SVD) is applied to the term-document matrix. However, as the feature space created by SVD is a rotation of the original term space, the constructed features cannot be interpreted easily by humans. [38]

In the constructed feature space, the lowest dimensions also contain the most information, while higher dimensions become less and less relevant [38]. In practice, using the first 100 to 300 dimensions has shown to provide the best results [39,40]. As SVD is a method from linear algebra, LSI works best for linearly separable classification problems. Thus, when using it with nonlinear classifiers, some information from the original data is lost [38]. Another drawback of LSI is its high computational complexity in $O(min(N^2D, ND^2))$ [41].

2.4.3 Term Weighting

Many different methods have been proposed for the weighting of terms in numerical vectors. They are usually composed of three factors: the term frequency factor, the collection frequency factor and the normalization factor.

Term frequency factor

The term frequency factor involves the number of occurrences of a term for the given document. As the simplest method, a binary threshold function can be applied, mapping each term that occurs in the document to the value 1 and all other terms to 0. However, it has been shown that the number of occurrences of a term in a document is a significant indicator for the importance of this term [29]. Thus, multiple methods that include the term frequency have been proposed. Some of the most common approaches for this are a raw term frequency (tf), the logarithm of the term frequency (log tf) or the inverse term frequency (itf). For their formulas, see Table 2.1. The experiments conducted in [42] have shown that the term weights assigned by tf, log tf and itf are not significantly different from each other.

Approach	Calculation Formula
Binary	$\begin{cases} 1, & tf > 0 \\ 0, & tf = 0 \end{cases}$
tf	tf
log tf	$\log(tf + 1)$
itf	$1 - \frac{r}{r + tf}$ $(usually\ r = 1)$

Table 2.1: Calculation formulas for term frequency factor approaches

Collection frequency factor

The relevancy of a term does not only depend on its number of occurrences in the given document, but also on its commonness in the whole training set. Terms, that are used very frequently over all training examples, such as "and", "some" or "have", are usually not very useful. Thus, the collection frequency factor considers the frequency of a term in the whole training set. [42]

The simplest approach for this is the inverse document frequency (idf). This approach weights terms higher, if they occur in a lower percentage of the training examples. Other approaches use information about already assigned categories in a supervised training set to calculate the collection frequency factor. In [42] relevance frequency (rf) has been proposed, using the assumption that occurrences of a term in documents that belong to a given category have a higher impact than occurrences in documents outside of the category. Also, many methods from feature selection, such as chi-square, mutual information and information gain, have been applied as the collection frequency factor. For more details on these methods, see section 2.4.2. In Table 2.2 the formulas for all mentioned collection frequency factor approaches are shown. [42]

Approach	Calculation Formula
Constant	1
Inverse document frequency	$\log\frac{N}{N_t}$
Relevance frequency	$\log(2+\frac{A}{\max(1,C)})$
Chi-square	see section 2.4.2
Mutual information	see section 2.4.2
Information gain	see section 2.4.2

Table 2.2: Calculation formulas for collection frequency factor approaches

where N is the total number of training examples, N_t is the number of training examples that contain term t, A is the number of positive training examples that contain term t and C is the number of negative training examples that contain term t.

Normalization factor

With the term frequency factor and the collection frequency factor, long documents still tend to have higher weights than shorter documents. Thus, cosine normalization is often applied to limit all weights between 0 and 1. [42]

$$w(t,d_i)=\frac{w(t,d_i)}{\sqrt{\sum_{j=1}^{N} w(t_j,d_i)^2}}$$

where d_i is the i-th document, $w(t,d_i)$ is the weight of term t in document d_i and N is the number of training examples.

2.5 Machine Learning Approaches in Text Categorization

A large variety of machine learning methods have been applied to text categorization. This section describes those different approaches and evaluates their suitability for source code knowledge extraction.

2.5.1 Regression Methods

Regression is the process of approximating a mapping between a set of predictor variables and an output variable. The predictor variables are therefore combined by a predefined parametrized regression function. Based on a set of training examples, the parameters in the regression function are estimated to achieve a good fit to the output variable [43]. The choice of a regression function is crucial for the quality of the resulting model. Even though the regression function is usually different from the original distribution, a good approximation can be achieved with the right choice.

A regression method can also be applied to supervised classification problems. For this purpose, the information about assignment of a document to a category is included as the output variable and the input dimensions are used as predictor variables. By setting a threshold on the output value, the regression function can be used to predict new input documents. [44]

In practice, regression functions other than linear and logistic are rarely applied, since more complex functions also introduce considerably more parameters, which are hard to estimate.

Linear Regression

A linear regression model defines the regression function as a linear combination of all input dimensions, so that a parameter is added for each input dimension. This leads to the following equation:

$$output = p_0 + p_1 x_1 + p_2 x_2 + \ \dots + p_n x_n$$

where p_i is the i-th parameter and x_i is the value of the input vector in the i-th dimension.

To estimate the parameter values, linear least-squares fit (LLSF) is often applied. This method constructs a term-category matrix M based on the input vectors $I(d_i)$ and the category vectors $O(d_i)$, so that the error in the following equation is minimized for the training set:

$$O(d_i) = M * I(d_i)$$

When applied to classification, linear regression methods do not work well with a fixed threshold. To optimally separate positive and negative examples, each dimension would need a different threshold value. Furthermore, when adding a training example, that is

very different from the current threshold value, the optimal threshold shifts significantly. To solve this problem and allow fixed thresholds, a logistic regression function is often used instead. [45]

Logistic Regression

The logistic function can be interpreted as a probability for an input to belong to a given class. A fixed threshold on this function can effectively determine all documents, that belong to the given category with a probability greater or equal to the threshold value [46]. The following equation shows a multivariate logistic regression function:

$$output = \frac{e^{p_0 + p_1 x_1 + p_2 x_2 + \ldots + p_n x_n}}{1 + e^{p_0 + p_1 x_1 + p_2 x_2 + \ldots + p_n x_n}}$$

where p_i is the i-th parameter and x_i is the value of the input vector in the i-th dimension.

Estimation of the parameters is usually conducted with maximum likelihood estimation. This method maximizes the probability, that the training examples are drawn from a logistic distribution with the given parameters. [47]

Both linear and logistic regression build a model with linear complexity, which is very prone to overfitting. It is commonly agreed, that these regression methods should be trained with at least 10 training examples per input dimension in order to achieve good generalization [48]. Since the input dimensions are treated independently and only summed up as a linear combination, neither word cooccurrences nor word order is incorporated into the classifier and all contextual information is disregarded [36]. Regression methods can include multiple categories into their output vector, which are also treated independently [36]. This multi-class model is basically equivalent to a set of binary classifiers, with one classifier for each category. However, regression methods are not applicable to one-class classification problems. As they learn a functional mapping of input dimensions to output categories, the training set needs to contain both positive and negative training examples. A classifier, which is learned on only positive examples, would predict every new input document to be inside the category. Both LLSF and maximum likelihood estimation can train a regression model in $O(ND^2)$ [49,50] and thus, have a cubic training time. The prediction of a new document is done by simply computing the value of the regression function with the given document weights. This leads to a linear prediction time in $O(D)$. Since regression methods treat all dimensions independently, they are able to learn input spaces of any dimensionality [36]. The learned parameter values allow humans to easily interpret a classifier and determine the impact

of each dimension on the prediction of a given input document. Further, the independence of dimensions reduces the impact of noisy values in a training example. As no cooccurrences are learned, noisy values only effect one dimension and are usually heavily outweighed by all correct training examples [51]. Also, since every dimension is considered separately, the problem of sparse input vectors boils down to the problem of unbalanced data for a dimension, as most inputs are zero. In [52] it has been shown, that regression methods work well with unbalanced data. Incremental learning can easily be conducted with regression methods. The already learned parameters can be used as the starting point of a new optimization process including the recently added training examples.

Priority	Requirement	Result
Must	R4: Phase-specific applicability	Only classification phase
	R5: Training time	Cubic
	R6: Prediction Time	Linear
Should	R1: Model complexity	Linear
	R2: Robustness against overfitting	No
	R3: Context sensitivity	No
	R7: Learning with high-dimensional input spaces	Yes
	R9: Robustness against noisy data	Yes
	R10: Ability to work with representation vectors	Yes
Could	R8: Interpretability by humans	Yes
	R11: Incremental learning capability	Yes

Table 2.3: Evaluation of regression methods

2.5.2 Rocchio Method

The Rocchio method for classification is based on the assumption that every category corresponds to one cluster of documents in the input space. Each of these clusters is represented by its centroid based on the Rocchio formula for relevance feedback from the field of information retrieval [36]. This formula is given as follows:

$$c_i(t) = \beta * \sum_{d_j \in A} \frac{w_j(t)}{|A|} - \gamma * \sum_{d_j \in C} \frac{w_j(t)}{|C|}$$

where A is the number of positive training examples that contain term t, B is the number of negative training examples that contain term t, $w_j(t)$ is the weight of term t in document d_j, and $c_i(t)$ is the value of the i-th category centroid for term t.

Basically, the Rocchio formula balances between closeness to the positive training examples and distance from the negative training examples. The parameters β and γ determine the impact of the positive and negative examples on the classifier. With all centroids calculated, a new input document is simply assigned to the category of the nearest centroid [36].

A Rocchio classifier is based on a linear model, with each centroid basically being a linear combination of the training examples. The representation of a category by its centroid prevents overfitting of the model, as the Rocchio method has shown to work well with considerably small amounts of training data [53]. However, similar to regression methods, a Rocchio classifier considers each dimension independently and is not able to incorporate word cooccurrences and word order. Thus, it loses all contextual information when applied to source code knowledge extraction. The Rocchio method inherently supports multi-class classification and can distinguish an arbitrary number of categories, as long as each category corresponds to one cluster of documents in the input space [54]. Also, it can be applied to one-class classification by setting the parameters $\beta = 1$ and $\gamma = 0$. For this, a threshold on the distance between an input document and the single centroid needs to be defined, so that all documents below this threshold are assigned to the category. The training of a Rocchio classifier comprises the calculation of a centroid for each category with the Rocchio formula. This requires a quadratic training time in the order of $O(ND + Dm)$ [54]. When predicting the category of a new input document, the distance of its input vector to each centroid needs to be computed. With the Euclidian distance metric, this requires $O(D)$ operations for each centroid, leading to an overall complexity of $O(Dm)$ [54]. As only N and D are considered, the prediction time has a linear complexity. Since the classifier is based only on a distance metric between two vectors, it is also applicable to high-dimensional input spaces. The centroid of a category is often also interpreted as its profile, representing the most typical input vector for this category [36]. With the help of such a profile, a human expert can easily identify the most important features for the corresponding category and measure their impact on the prediction of a specific input document. Further, the construction of centroids with independent dimensions makes the Rocchio method significantly robust to noisy training examples, as they are outweighed by the correct training examples. As every dimension is considered separately, sparse input vectors essentially only lead to unbalanced input

data in each of those dimensions. The Rocchio method has shown to deal with unbalanced data well [55]. However, incremental learning is not possible with Rocchio classifiers. When a training example is added, either the positive or negative term in the Rocchio formula need to be completely recomputed for each centroid.

Priority	Requirement	Result
Must	R4: Phase-specific applicability	Both phases
	R5: Training time	Quadratic
	R6: Prediction Time	Linear
Should	R1: Model complexity	Linear
	R2: Robustness against overfitting	Yes
	R3: Context sensitivity	No
	R7: Learning with high-dimensional input spaces	Yes
	R9: Robustness against noisy data	Yes
	R10: Ability to work with representation vectors	Yes
Could	R8: Interpretability by humans	Yes
	R11: Incremental learning capability	No

Table 2.4: Evaluation of the Rocchio method

2.5.3 Support Vector Machine

A linear classification model can be seen as a hyperplane that separates the input space into a positive and negative subspace. Such a hyperplane is defined by its weight vector and its bias, as shown in Figure 2.2.

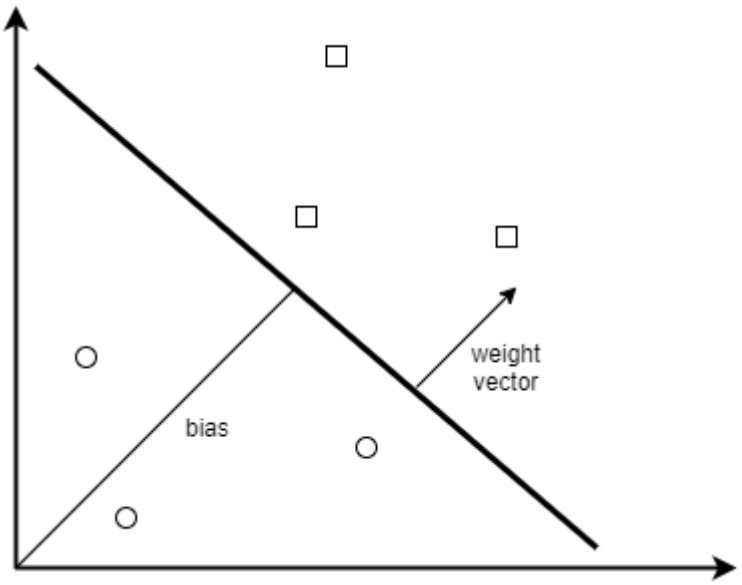

Figure 2.2: Hyperplane defined by its weight and bias

For a binary linearly separable classification problem, there are infinitely many separating hyperplanes, that fit the training set. However, these hyperplanes can differ significantly with regard to their generalization ability. In Figure 2.3, a binary linearly separable classification problem with two possible linear classifiers is shown. Even though both classifiers fit the training set, the classifier C1 generalizes much better to the overall distribution.

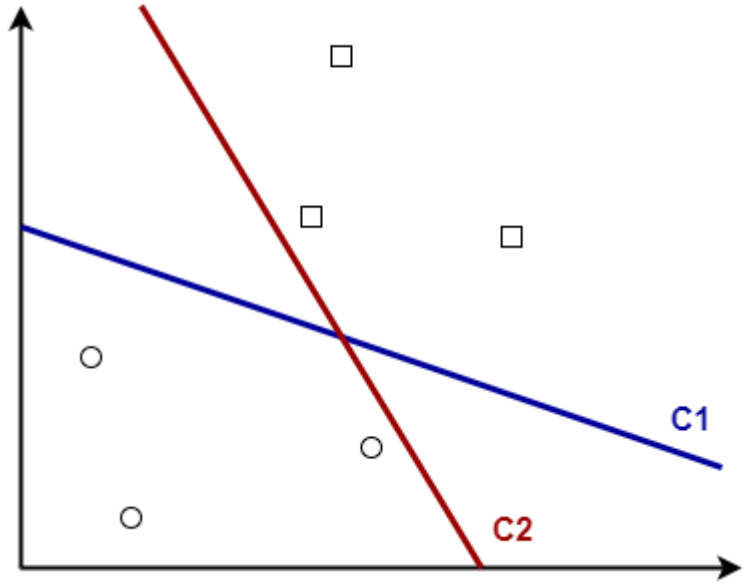

Figure 2.3: Two different separating hyperplanes for a linearly separable problem

A support vector machine (SVM) addresses this problem by choosing the hyperplane with the highest margin. The margin of a separating hyperplane is defined as its distance to the input vector of the nearest training example of either the positive or negative category [56]. The theoretical basis for this approach is the principle of structural risk minimization. This principle states, that the generalization error of a machine learning classifier depends on a) the error on the training set and b) the Vapnik-Chervonenkis (VC) dimension of the learning model [57]. For a linearly separable classification problem, the training error of a separating hyperplane is always zero. Thus, the generalization error of such a separating hyperplane only depends on the VC dimension of its training model. As Vapnik has shown in [58], choosing a separating hyperplane with a higher margin results in a lower VC dimension. Consequently, an SVM classifier achieves the best generalization amongst all separating hyperplanes.

The training process of an SVM consists of maximizing the margin of the classifier while correctly separating the positive and negative training examples. This is equivalent to minimizing the absolute value of the hyperplane weight vector as follows [56]:

$$min_{w,b} \frac{1}{2}|w|^2$$

$$s.t.\ y_i(w^T x_i + b) \geq 1 \quad \forall i \in [1, N]$$

where w is the weight vector of the hyperplane, b is the bias of the hyperplane, y_i is the category assignment of the i-th training example and x_i is the input vector of the i-th training example.

This is called the primal form of the SVM optimization problem. As the primal form is a convex quadratic optimization problem, its optimal solution (w^*, b^*) can be found [56]. When the maximum margin hyperplane is determined, the category of a new input document can be predicted by the following decision rule:

$$w^{*T}x + b$$

Here, a positive value indicates that the document should be assigned to the category.

However, in this basic form SVMs are only applicable to separable problems. When given a not separable training set, the learning process fails. To overcome this problem, a soft-margin SVM also allows training examples inside the margin. A slack variable ξ is added to the optimization formula to penalize those examples. This leads to the following optimization problem [56]:

$$min_{w,b}\frac{1}{2}|w|^2 + C\sum_{i=1}^{N}\xi_i$$

$$s.t.\ y_i(w^T x_i + b) + \xi_i \geq 1 \quad \forall i \in [1, N]$$

$$s.t.\ \xi_i \geq 0 \quad \forall i \in [1, N]$$

The parameter C is used to balance between the maximization of the margin and the minimization of the slack penalty.

Still, such a soft-margin SVM classifier is limited to linear problems and cannot fully solve most text categorization problems. To overcome this constraint, the SVM optimization problem can be reformulated to its dual form by applying a Lagrangian. This form is defined as follows [56]:

$$max_{a_i}\sum_{i=1}^{N}a_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}a_i a_j y_i y_j x_i x_j$$

$$s.t.\ \sum_{i=1}^{N}a_i y_i = 0 \quad \forall i \in [1, N]$$

$$s.t.\ a_i \geq 0 \quad \forall i \in [1, N]$$

In this dual form, an SVM allows the application of the "kernel trick". By the use of a nonlinear kernel function, the input vectors of an SVM can be mapped to a higher-dimensional feature space [59]. In order to do so, this kernel function is applied to the dot product of two input vectors as follows [56]:

$$K(x_i, x_j) = \varphi(x_i)\varphi(x_j)$$

where φ is the kernel function.

This changes the dual SVM formulation to [56]:

$$max_{a_i} \sum_{i=1}^{N} a_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j y_i y_j K(x_i, x_j)$$

$$s.t. \sum_{i=1}^{N} a_i y_i = 0 \quad \forall i \in [1, N]$$

$$s.t.\ a_i \geq 0 \quad \forall i \in [1, N]$$

Some of the most common kernel functions are the polynomial function, the Gaussian radial basis function and the sigmoid function [60].

To allow the application of SVMs to multi-class problems, several methods have been proposed. These methods either combine multiple binary SVMs to a multi-class classifier or reformulate the SVM optimization problem to be applicable to multiple categories. A combination of multiple binary SVM classifiers can be implemented by either the one-against-one or the one-against-all scheme.

Here, the one-against-one method creates one binary SVM for each pair of categories, which yields $\frac{m(m-1)}{2}$ binary classifiers, where m is the number of categories. Each of those is learned with the training examples of its two categories as the positive and negative training set respectively. When predicting the category of a document, a voting strategy between the binary SVMs is applied. One possible voting strategy is the "Max Wins" approach, where each binary SVM votes for its winning category and the category with the most votes is selected in the end. [61]

In contrast to this, the one-against-all method creates only one binary SVM for each class, so that m classifiers are built. Each binary SVM is learned with the training examples of its category as the positive training set and all other training examples as the negative

training set. The prediction of an input document is conducted by computing a confidence score for this document with each SVM and selecting the category with the highest confidence score. [61]

For the reformulation of the SVM optimization formula to a multi-class problem, several different methods have been proposed. The multi-class SVM from Weston and Watkins [62] combines the m binary optimization formulas from the one-against-all approach into one multi-class SVM formulation. Crammer and Singer proposed another multi-class SVM formulation based on the one-against-all approach in their paper [63].

The authors of [61] provide a detailed comparison of these different multi-class SVM approaches and evaluate their performance on several text categorization problems.

A major problem of nonlinear SVMs in text categorization is their memory requirement. As the size of the kernel matrix scales quadratically with the number of training documents, its computation gets infeasible for large training sets [56,59]. In this case, the SVM training algorithm needs to be adjusted, so that it can learn incrementally on subsets of the training data. Two possible approaches to deal with this problem are decomposition methods and iterative chunking.

Decomposition methods involve the separation of the training set into many subsets, which can be learned independently. To solve these sub-problems, the dual SVM formulation is used. Here, only the a_i from the currently active subset are changeable, while the other a_i are fixed. While this method solves the problem of memory requirement, it only converges slowly and requires more time in the training process. Additionally, the selection of the subsets is critical to the performance of a decomposition method. An inappropriate subset selection leads to even longer training times and potentially to non-convergence of the training algorithm. [59]

In iterative chunking, each iteration in the training process consists of selecting a subset and training an SVM only with the selected training examples. Once this training process is finished, the subset for the next iteration is initialized with the current support vectors. For all other training examples, their error on the Karush-Kuhn-Tucker conditions is calculated and the k training examples with the highest error are added to the subset. This is repeated until a stopping criterion, such as a threshold on the total error of the classifier, is reached. [56]

The basic form of an SVM can only build linear models. However, through the application of kernel function to the input documents, also nonlinear models of different complexity can be learned. Due to the optimality of a maximum-margin classifier, an SVM inherently prevent overfitting and has shown to work well with small training sets [64]. As input documents need to be represented as numerical vectors, word order cannot be incorporated into an SVM classifier. However, since the input dimensions are not considered independently, but each input document represents one data point, word cooccurrences can be learned. SVMs are applicable to both one-class classification [65] and multi-class classification [62,63] and thus they are also applicable to both identification and classification phase. The training time of a basic kernel SVM ranges between $O(N^2D)$ and $O(N^3D)$ [66]. Fast implementations, such as [67] and [68], have at most a cubic training time. The prediction time of a trained SVM classifier is in $O(Ds)$, where s is the number of support vectors [69]. Since the number of support vectors in text categorization is usually much smaller than the number of training examples or the number of dimensions, it can be disregarded, leading to a linear prediction time. As has been shown in the research on text categorization, SVMs are able to learn high-dimensional input spaces very well [32-34]. However, the importance of a single feature cannot be easily determined from the support vectors and thus, SVM classifiers are very hard to understand and interpret for a human [70]. Noisy training examples can heavily influence the performance of an SVM, even if soft-margin SVMs are used to tolerate some error in the training set. Especially if noise occurs as extreme values, it has a high impact on the slack variables. Thus, noisy data needs to be filtered out before applying an SVM algorithm. Sparse input vectors also pose a problem to SVMs, as they result in many zero values when building a dot product between two vectors [33]. Generally, a trained SVM classifier cannot easily incorporate new training examples, as it only stores the support vectors and discards all other input documents. Even though there are some implementations of incremental SVM learning algorithms, such as SVMHeavy and LaSVM, those algorithms suffer in other areas. This often includes a lower quality of the resulting classifier or larger memory requirements. However, there are cases in which new training examples can be applied to an already trained classifier. If the training examples are correctly classified by the existing classifier and are outside of the margin, no retraining is needed.

Priority	Requirement	Result
Must	R4: Phase-specific applicability	Both phases
	R5: Training time	Cubic
	R6: Prediction Time	Linear
Should	R1: Model complexity	Nonlinear
	R2: Robustness against overfitting	Yes
	R3: Context sensitivity	Only cooccurrences
	R7: Learning with high-dimensional input spaces	Yes
	R9: Robustness against noisy data	No
	R10: Ability to work with representation vectors	No
Could	R8: Interpretability by humans	No
	R11: Incremental learning capability	No

Table 2.5: Evaluation of support vector machines

2.5.4 Example-based Classifiers

In contrast to most classification methods, example-based classifiers do not build an explicit classification model from the training examples. Instead, they predict the categories of new documents based on the most similar examples in the training set. Thus, a different generalization from the training data is generated in each prediction process [36]. In the following, the most commonly used example-based classifier, the k-nearest-neighbor method, will be described and evaluated.

K-Nearest-Neighbors

When predicting the category of an input document, the k-nearest-neighbor (kNN) method searches for the k most similar training examples. Similarity is here measured as the cosine between the vectors of two documents [27]. Based on the determined training examples, a confidence score is calculated for each category. This is done by summing up the weighted similarities of the training examples, that are in a given category. Finally, the category with the highest confidence score is assigned to the input document. [71]

The choice of k is essential for the quality of a kNN classifier. A higher k leads to a better generalization ability and minimizes the impact of noisy training examples [72]. However, the prediction time also increases along with the value of k. Several experiments have shown that values between 20 and 45 achieve the best prediction quality [36].

Since kNN as an example-based classification method does not build an explicit model, prediction times can get very long and impractical. Possible solutions to this problem include dimensionality reduction through feature selection and the reduction of the number of training examples with clustering [71]. However, both of these methods are not able to fully solve this problem.

Example-based classifiers, such as kNN, apply a nonlinear classification model, as the prediction of an input document is not based on any linear combinations of training examples, but on a similarity measure. To prevent overfitting, the number of training examples used in the prediction of a new input document needs to be sufficiently high [72]. As kNN works with simple numerical vectors, it is not able to consider word order in its input documents. However, since each training example is treated as a data point and dimensions are not independent from each other, word cooccurrences are incorporated into the classifier. kNN supports multiple output categories and can easily be applied to multi-class problems. Further, it can also be used for one-class classification by computing the confidence score for an input document and assigning it to the category if this score is above a defined threshold. As a lazy learning method, kNN builds no explicit model and thus, has a constant training time. In turn, this leads to high prediction times, as the prediction of a new input document requires a quadratic time in $O(NDk)$ [73]. As shown in [72], kNN does not work well with high-dimensional input spaces and sparse input representations. Also, it cannot be easily interpreted by humans, since it does not build an explicit classification model. Only the most similar training examples are known, while no information about the importance of different features is available. A noisy training example can impact the classifier significantly, as it might be selected as one of the training examples used for prediction [71]. Even though a high k can alleviate this problem, misclassifications are still likely to happen in a noisy training set. A new training examples can easily be added to an existing kNN classifier, as it builds no explicit model and the new training example is simply appended to its current list of training examples.

Priority	Requirement	Result
Must	R4: Phase-specific applicability	Both phases
	R5: Training time	Constant
	R6: Prediction Time	Higher than linear
Should	R1: Model complexity	Nonlinear
	R2: Robustness against overfitting	Yes
	R3: Context sensitivity	Only cooccurrences
	R7: Learning with high-dimensional input spaces	No
	R9: Robustness against noisy data	No
	R10: Ability to work with representation vectors	No
Could	R8: Interpretability by humans	No
	R11: Incremental learning capability	Yes

Table 2.6: Evaluation of example-based classifiers

2.5.5 Decision Tree Classifiers

A decision tree classifier is a rooted tree consisting of multiple decision rules. Each internal node of this tree is labeled with a term and formulates several distinct cases through its departing branches. A leaf node in the decision tree is labeled with a category. To predict the category of a new document, the tree is traversed beginning from the root node. Here, the decision of which branch should be taken from an internal node is determined by the weight of its labeled term. Upon reaching a leaf node, the category of this node is assigned to the document. [36]

In the training phase, the decision tree is initialized as a simple root node that refers to all training examples. The training set is then iteratively split into multiple subsets according to a splitting condition. Each split creates several child nodes under the current node, so that each child node matching a case in the condition. The selection of the splitting condition is conducted by the use of a loss function. This iterative process stops once a stopping criterion is reached. Afterwards, pruning methods can be applied to improve the generalization ability of the decision tree. [74]

CART

The Classification and Regression Trees (CART) algorithm allows only two cases for each condition, which leads to the creation of a binary decision tree. As the loss function,

it uses the Gini index function, that determines how well a split separates the positive and negative training examples of each category as follows:

$$G = \sum_{k=1}^{m} (p_k * (1 - p_k))$$

where p_k is the percentage of training examples from the k-th category, which match the splitting condition, and m is the number of categories.

C4.5

C4.5 is a decision tree learning algorithm using the information gain of the training subsets based on their entropy as its loss function. The entropy function is given as follows:

$$E = \sum_{i=1}^{m} -P(c_i) * log_2 P(c_i)$$

where m is the number of categories and c_i is the i-th category.

Based on the entropy of the current training set $e(s)$ and the entropy of each subset according to the splitting condition $e(s_i)$, the information gain is defined as

$$G = e(s) - \sum_{i=1}^{|c|} P(c_i) * e(s_i)$$

where $|c|$ is the number of cases in the splitting condition and c_i is the i-th case.

Random Forest

Even when applying pruning to a decision tree classifier, it is still prone to overfitting on the training set. To alleviate this problem, the idea of combining multiple different decision trees to one classifier emerged. For this to be practical, all these decision trees need to be highly uncorrelated. Such a classifier consisting of multiple uncorrelated decision trees is called random forest. The prediction of an input document in a random forest classifier is done by a majority vote of all contained decision trees. [75]

One possible method of constructing a set of uncorrelated decision trees is to train them with randomly selected subsets of features. To still achieve the same dimensionality as the original input, this selection is conducted with replacement, meaning that features can appear multiple times. [75]

Decision trees build a nonlinear model through the application of nested decision rules [76]. As a decision tree becomes larger and deeper, it strongly tends to overfit on the training set. Even with the application of pruning and the construction of random forests, overfitting remains a significant problem of decision trees. Since decision rules only consider one term without regard for its position in the input document, a decision tree cannot consider the word order. However, it is able to learn word cooccurrences expressed by nested rules. The leaf nodes of a decision tree can be labeled by an arbitrary amount of categories. Thus, this machine learning method can easily be applied to multi-class classification. However, it cannot perform one-class classification, as the learning algorithms are based on separating the training set into multiple subsets, depending on the category assignments of the training examples. The training complexity of a decision tree depends on its learning algorithm. With the CART algorithm, the construction of a decision tree requires a training time in $O(N * \log(N) * D)$ [77]. When applying the C4.5 algorithm, the training time is in the order of $O(ND^2)$ [78]. Thus, both algorithms require a cubic training time. Predicting the category of a new input document is done by evaluating the nested decision rules until a leaf node is found. Since each decision rule considers one term, the overall prediction time is in $O(D)$. Decision trees have shown to suffer from high-dimension input spaces, as they need to include much more decision rules and grow very deep [79]. The impact of different terms on the overall classifier is displayed by the decision rules and can be easily understood and interpreted by a human expert [35]. However, noisy training sets pose a big problem for decision tree classifiers, as the learning algorithms cannot filter this noise out [35]. Further, decision trees are not well suited for sparse input vectors, since more decision rules have to be learned to correctly fit a sparse training set and the trees have to grow much deeper [80]. Also, once a decision tree is constructed, it cannot be easily modified to include new training examples. Thus, the training process needs to be started from scratch, if the training set is extended.

Priority	Requirement	Result
Must	R4: Phase-specific applicability	Only classification phase
	R5: Training time	Cubic
	R6: Prediction Time	Linear
Should	R1: Model complexity	Nonlinear
	R2: Robustness against overfitting	No
	R3: Context sensitivity	Only cooccurrences
	R7: Learning with high-dimensional input spaces	No
	R9: Robustness against noisy data	No
	R10: Ability to work with representation vectors	No
Could	R8: Interpretability by humans	Yes
	R11: Incremental learning capability	No

Table 2.7: Evaluation of decision tree classifiers

2.5.6 Probabilistic Classifiers

Probabilistic classifiers involve the estimation of distributional probabilities based on the training examples. Here, the Bayes' theorem is used for the computation of the conditional probability, that a given input document belongs to the category. This theorem is given as follows [36]:

$$P(c_i|d_j) = \frac{P(c_i) * P(d_j|c_i)}{P(d_j)}$$

where c_i is the i-th category and d_j is the j-th training example.

However, the estimation of all the probabilities is very hard and scales badly to high-dimensional input spaces. Assuming that the Bernoulli model is used for the probability distribution, the number of free parameters amounts to $2^D * c$, where D is the number of input dimensions and c is the number of categories [54]. In order to reduce the number of free parameters, probabilistic classifiers apply an independence assumption on their input data.

Naive Bayes

The naive Bayes (NB) classifier is one of the most popular probabilistic classification methods. It applies the Bayes' theorem with strong conditional independence assumptions and thus, treats the input dimensions independently from each other. However, this assumption is not consistent with the properties of natural language, as words in a text can be highly correlated to each other. Thus, the performance of NB classifiers drops in the field of text categorization. Still, it is very widely used and can often keep up with other text categorization methods. [35]

In the training phase, a conditional probability is computed for each pair of term and category as follows:

$$P(c_i|t_k) = \frac{P(c_i) * P(t_k|c_i)}{P(t_k)}$$

where c_i is the i-th category and t_k is the k-th term in the input space.

The prediction of a new input document with an NB classifier is done by simply calculating the probability that the document belongs to a category and assigning it to the most probable class [35]. This calculation comprises the product of all term probabilities as follows:

$$P(c_i|d_j) = \prod P(c_i|t_k)$$

Probabilistic classifiers predict the category of an input document through the product of different probabilities. Thus, they apply a nonlinear model [81]. NB classifiers are robust to overfitting and have shown to work well with small training sets [82]. However, through the application of conditional independence assumptions, all contextual information from word cooccurrences and word order is disregarded. As the Bayes' theorem inherently supports multiple categories, probabilistic classifiers are easily applicable to multi-class classification. Also, the construction of one-class NB classifiers is possible, as described in [83]. The training of an NB model consists of the estimation of probabilities based on the term occurrence count in the training set. This requires a linear training time in $O(N)$ [84]. When predicting a new input document, the probability of each class needs to be calculated, based on the learned term probabilities. Therefore, the prediction time of an NB classifier is in $O(Dm)$. The dimensionality of the input space does not significantly impact a probabilistic classifier, as each dimension is just represented by a probability. Thus, also high-dimensional input spaces can be learned without problems

[85]. A NB model is not directly interpretable by humans, as only the probabilities based on the term occurrences in the training set are known [36]. Since dimensions are treated independently with a conditional independence assumption, noisy values are heavily outweighed by all correct values in the training set. Thus, a probabilistic classifier is able to perform well even with a considerable amount of noise [86]. Sparse numerical vector representations do not affect the prediction quality of a probabilistic model, as only the term occurrence counts over the whole training set are considered [87]. However, to incorporate new training examples into the classification model, all probabilities need to be recalculated and consequently, incremental learning is not possible [32].

Priority	Requirement	Result
Must	R4: Phase-specific applicability	Both phases
	R5: Training time	Linear
	R6: Prediction Time	Linear
Should	R1: Model complexity	Nonlinear
	R2: Robustness against overfitting	Yes
	R3: Context sensitivity	No
	R7: Learning with high-dimensional input spaces	Yes
	R9: Robustness against noisy data	Yes
	R10: Ability to work with representation vectors	Yes
Could	R8: Interpretability by humans	No
	R11: Incremental learning capability	No

Table 2.8: Evaluation of probabilistic classifiers

2.5.7 Artificial Neural Networks

Artificial neural networks are a machine learning method inspired by the biological nervous system. They consist of many units, that are similar to neurons, and weighted connections between them [20]. Each unit takes multiple input values and calculates an output value based on a nonlinear activation function. Some commonly used activation functions are the logistic, Gaussian radial basis, sigmoid and rectified linear unit function [20,38]. The connections between the units of a neural network determine its architecture. Usually, the units are pooled in layers, so that the output of one layer serves as the input of one or multiple other layers. However, there is a large variety between different architectures regarding the number of layers, the number of units per layer and

the connections between layers. The training of a neural network is often conducted by the backpropagation algorithm. In doing so, the input vector of a training example is first propagated through the neural network using the current weights to compute a prediction. Afterwards the error between this prediction and the expected output value of the training example is determined and propagated back through the neural network. While doing so, gradient descent is applied to each weight to reduce the error for this training example. This is done iteratively over all training examples until a given error threshold is reached [60].

Due to their high complexity, neural networks tend to overfit on the training set. Experiments in [38] have shown, that the generalization error of a neural network usually reaches its optimum after a few hundred iterations on the training set and then starts increasing again. To overcome this problem, several methods have been proposed. Regularization adds an extra term to the error function to minimize both the error on the training set and the norm of the weight vector [88]. Early stopping monitors the generalization error of the neural network and stops the training when the generalization error starts increasing again [89]. Dropout refers to the deactivation of some units during a learning iteration, so that these units are neither considered during forward propagation nor backpropagation [90].

Feedforward Neural Network

Feedforward neural networks (FNN) are the most basic form of a neural network classifier. They consist of an input layer, which receives the input vectors and passes them through one or multiple hidden layers, that extract useful features from the input data. The last hidden layer propagates its results to the output layer, which then computes an output value for each category. In this architecture, only forward connections between layers exist. [91]

Neural Text Categorizer

The representation of text as a numerical vector entails a very high dimensionality, often hundreds of thousands or even millions of input dimensions, and a sparse distribution inside these vectors, as documents usually only contain a fraction of the total number of terms. Both of these characteristics weaken the prediction performance and training time of a neural network [35]. In order to solve the problems of high dimensionality and sparse input vectors in neural networks, the authors of [33] proposed the Neural Text

Categorizer (NTC). This special neural network uses string vectors as its input, in contrast to the conventional numerical vector representation. Here, a fixed number of predefined features are applied as the input dimensions. These features are based on properties of the input document and can be divided into three categories: a) linguistic features, such as the first noun in the text or the last verb in a paragraph, b) statistical features, such as the highest frequent words or the highest weighted word according to a given weighting function, and c) positional features, such as the first word of a sentence or a random word in the first paragraph. The connections in an NTC are not associated with a single weight, but maintain a table that maps input strings to a weight associated with this string.

For the learning process, all tables are initialized with the corresponding term frequencies. Afterwards, the NTC is learned iteratively on the set of training examples until a stopping criterion is met. In each iteration, a training example is propagated into the network and the output node with the highest activation is selected as the winning category. If this winning category matches the category of the training example, the algorithm proceeds with the next example without changing any weights. However, if the training example was misclassified, the weights for the predicted category are adjusted, so that its activation level for this training example shrinks, and the weights for the correct category are adjusted to achieve a better match to the training example.

Recurrent Neural Network

In contrast to an FNN, the architecture of a recurrent neural network (RNN) includes not only forward connections, but also backward connections. This way, the output value of a layer for the last training example is used by another layer as an additional input. Thus, an RNN maintains part of its current state for future iterations and is able to learn the context of a given input term [20].

The training of an RNN is usually conducted by Backpropagation Through Time (BPTT). This learning algorithm is similar to the conventional backpropagation algorithm, but propagates the output error not only to the current activation, but also to previous activations, that impacted the current state of the RNN [92].

The Long Short-Term Memory (LSTM) network is a very popular variant of RNNs, which is able to learn long-term dependencies of its inputs. This architecture mainly consists of three different gates: the input gate, the forget gate and the output gate. The input gate computes an activation level from the current training example and populates this

activation level into the LSTM network. To include the context from the last training example, the forget gate determines which values from the last activation of the network should be selected and how these values should be weighted. Finally, the output gate computes a new output value from the current input and the context, that was maintained from the last activation of the network. Figure 2.4 shows these three gates in the architecture of an LSTM network. [92]

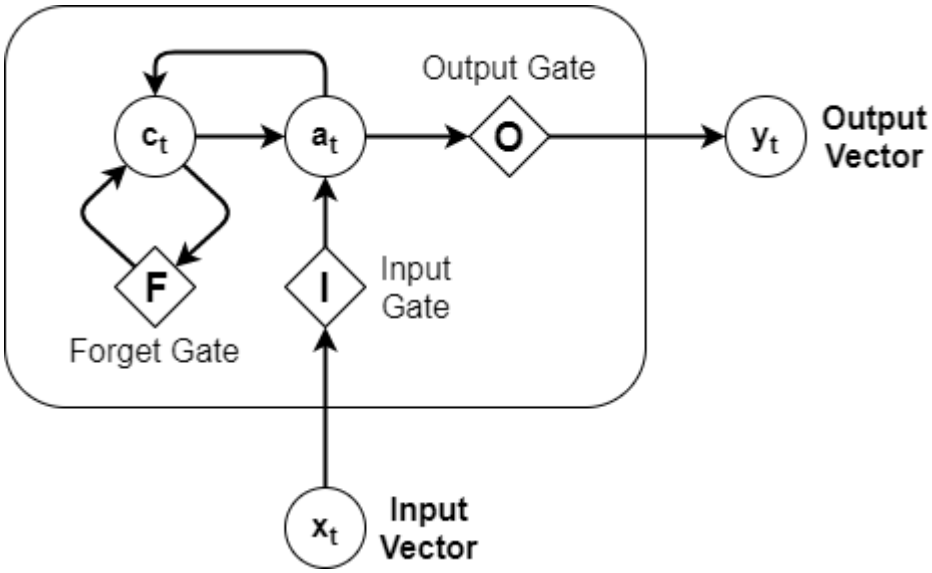

Figure 2.4: LSTM network architecture

Convolutional Neural Network

Convolutional neural networks (CNN) were originally used in image classification. They consist of three different kinds of layers: convolution layers, pooling layers and classification layers. A convolution layer separates the image into regions, which contain a square of pixels, and applies a nonlinear function to each of them. The different regions can overlap, often using a small offset value, such as an offset of one pixel. The pooling layer then reduces the output of a convolution layer to abstract higher level features. For this, the output of the convolution layer is separated into non-overlapping regions and the pixels in each region are merged together. Common methods for this are taking the average value or maximum of the region. Classification layers are located after all convolution and pooling layers and work like the layers in a feedforward neural network, mapping the outputs of convolution and pooling to different categories. [28]

The authors of [28] adopted this approach for the field of text categorization. Here, a main advantage of CNNs is their ability to take word order into consideration. In order to do so, text can be represented by a sequence of one-hot vectors, so that each word in the text is mapped to a one-hot vector. A region in the convolution layer is constructed by concatenating several consecutive word vectors. These word vectors are then merged by computing their average or maximum in the pooling layer. However, in contrast to images, text documents do not have a fixed size. Thus, the pooling layer needs to reduce the variable sized output of the convolution layer to a fixed size input for the classification layers. This can be done by using a fixed number of pooling units and selecting the region size for pooling accordingly.

Further, the authors of [28] proposed several parallel CNNs, which significantly increased the prediction performance. For this purpose, they combined multiple convolution layers with different region sizes and merged their outputs in the classification layers.

Adaptive Resonance Theory Network

The authors of [93] proposed the Adaptive Resonance Theory (ART) network as a self-organizing neural network for unsupervised clustering. An ART network consists of an input layer, where each unit corresponds to a dimension of the input vector space, and an output layer, which represents the categories. Initially the output layer is empty and over the course of the training process, new output units are added. Training an ART network is done as an iterative process for each training example. In doing so, a training example is first populated to all input units and then an activation level is computed for each output unit. The output unit with the highest activation is selected as the winning unit and propagates its activation back to the input layer. This backpropagated vector is considered the prototype of the category corresponding to this output unit. If the prototype is sufficiently similar to the input vector according to a defined threshold, the category is enlarged by the current training example and the weights of the winning unit are adjusted. On the other hand, if the similarity of the prototype is below the defined threshold, the input is again propagated to all output units except the previously winning one to determine a new winning unit. This is repeated until either an output unit is found that is sufficiently similar to the input vector or all output units failed to pass the similarity test. In the latter case, a new output unit is created with weights according to the current training example. The architecture of an ART network is shown in Figure 2.5.

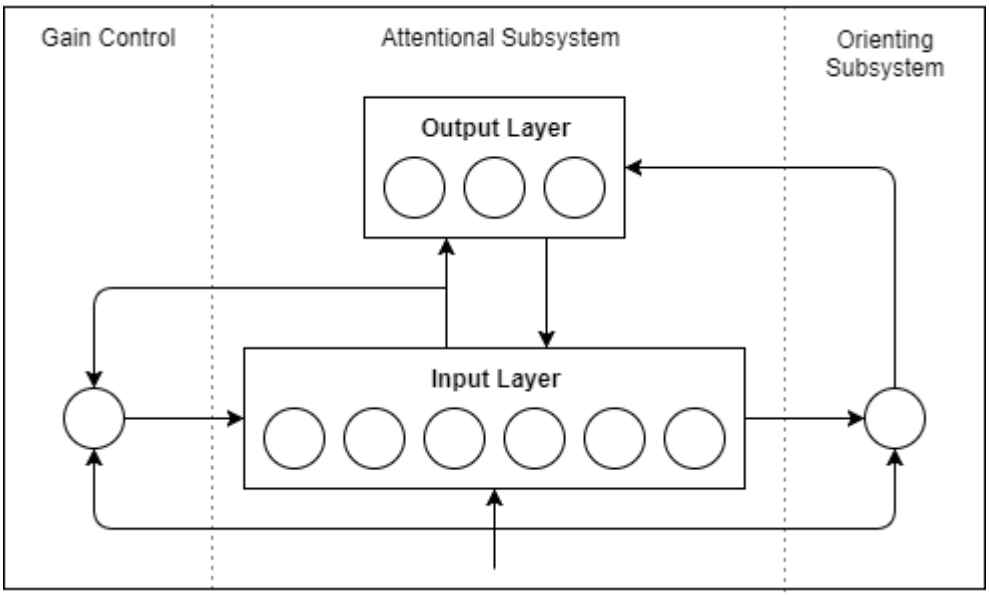

Figure 2.5: ART network architecture

Self-organizing Map

A self-organizing map is a structure comprising an input layer and a grid of output units, that can be used to cluster similar documents. Each input unit corresponds to a dimension of the input vector space, while each output unit corresponds to one cluster. To establish a mapping from the input to the output values, weighted connections are established between each input and output unit. [93]

In order to train a self-organizing map, an iterative process is applied. In each iteration, one training example is selected randomly and then mapped onto the grid of output units, according to the current weights. The output unit with the highest activation level is selected as the winner and subsequently its weights are adjusted to further increase its activation level for the selected training example. Further, the weights of topological neighbors of the winning unit in the output grid are also adjusted according to the selected training example. Changes on a weight are usually done by simply reducing its difference to its corresponding input value. Over the course of many iterations, the learning rate of the winning unit, the learning rate of neighboring units and the range for neighbor unit adaption need to be reduced to guarantee convergence. [93]

Through the application of its nonlinear activation function, a neural network builds a nonlinear model. Even though this model tends to overfit on the training set, this problem can be prevented by methods like regularization, early stopping and dropout [38]. As neural networks do not consider their input dimensions independently, all of them are able to learn word cooccurrences. However, only the NTC, RNNs and CNNs can include the word order from their training examples, while FNNs, ART networks and SOMs disregard this contextual information. The training time of a neural network depends on both its architecture and training algorithm. The backpropagation algorithm has a computational complexity of $O(w^3)$, where w is the number of weights in the neural network [94]. As the number of weights is much larger than D, in an optimistic case this leads to a cubic training time for just one iteration. The number of iterations until convergence is unbounded, but must be greater than the number of training examples [94]. Thus, the training time of the backpropagation algorithm is of higher than cubic. However, this algorithm is only used in FNNs, RNNs and CNNs. The described training process of an NTC only requires a quadratic to cubic training time, as the number of weights in this neural network depends on the number of selected string features instead of the size of the total training set vocabulary. An ART network also has a learning algorithm, that is significantly faster than the conventional backpropagation algorithm. Additionally, this algorithm only needs to iterate over the training set once and thus, has a quadratic complexity in $O(ND)$. The described training algorithm for an SOM requires a quadratic training time in $O(N^2)$ [95]. In the prediction phase, all neural networks require a prediction time in $O(Dh)$, where h is the number of hidden nodes in the network [96]. Also, all neural network approaches are able to learn a high-dimensional input space without further problems. However, the classification model of a neural network consists of the learned weights and is not interpretable by a human. The only exception of this is the ART network, where the weights of each output node can be interpreted as the profile of the matching category. Neural networks have shown to be very robust against noisy data and even benefit from the presence of some noise in the training set, as this noisy data prevents the neural network from overfitting and thus, increases prediction quality [87]. The FNN and RNN approaches suffer from sparse input vectors, as this adds much unnecessary data, that their training algorithms need to learn [97]. The other described approaches are either robust to sparse inputs or use other representation forms than numerical vectors, which are string vectors for the NTC and one-hot vectors for the CNN approach. However, with the exception of the ART network, none of the neural network approaches can include new training examples into their already learned weights [32].

Priority	Requirement	Result
Must	R4: Phase-specific applicability	Only classification phase
	R5: Training time	FNN: Higher than cubic NTC: Cubic RNN: Higher than cubic CNN: Higher than cubic ART: Quadratic SOM: Quadratic
	R6: Prediction Time	Linear
Should	R1: Model complexity	Nonlinear
	R2: Robustness against overfitting	Yes
	R3: Context sensitivity	FNN: Partially NTC: Yes RNN: Yes CNN: Yes ART: Partially SOM: Partially
	R7: Learning with high-dimensional input spaces	Yes
	R9: Robustness against noisy data	Yes
	R10: Ability to work with representation vectors	FNN: No NTC: Yes RNN: No CNN: Yes ART: Yes SOM: Yes
Could	R8: Interpretability by humans	FNN: No NTC: No RNN: No CNN: No ART: Yes SOM: Yes
	R11: Incremental learning capability	FNN: No NTC: No RNN: No CNN: No ART: Yes SOM: No

Table 2.9: Evaluation of neural networks

2.5.8 Comparison of the Machine Learning Approaches

The machine learning approaches from the field of text categorization, which have been described in the previous sections, shall now be compared with regard to the requirements from section 2.2. This serves as the basis for the selection of the machine learning approach, that will be used in the proposed concept. Table 2.10 shows a comparison of the different machine learning methods. Due to space limitations, the following abbreviations will be used:

Reg = Regression Methods

Roc = Rocchio Method

Ex = Example-based Classifiers

DT = Decision Tree Classifiers

Prob = Probabilistic Classifiers

ANN = Artificial Neural Networks

	Requirements														
	Must						**Should**						**Could**		
		R4-IP	**R4-CP**	**R5**	**R6**	**R1**	**R2**	**R3**	**R7**	**R9**	**R10**	**R8**	**R11**
Reg		No	Yes	Cubic	Linear	Linear	No	No	Yes	Yes	Yes	Yes	Yes
Roc		Yes	Yes	Quadratic	Linear	Linear	Yes	No	Yes	Yes	Yes	Yes	No
SVM		Yes	Yes	Cubic	Linear	Nonlinear	Yes	Part.	Yes	No	No	No	No
Ex		Yes	Yes	Constant	> Linear	Nonlinear	Yes	Part.	No	No	No	No	Yes
DT		No	Yes	Cubic	Linear	Nonlinear	No	Part.	No	No	No	Yes	No
Prob		Yes	Yes	Linear	Linear	Nonlinear	Yes	No	Yes	Yes	Yes	Yes	No
ANN	**FNN**	No	Yes	> Cubic	Linear	Nonlinear	Yes	Part.	Yes	Yes	No	No	No
	NTC			Cubic				Yes			Yes	No	No
	RNN			> Cubic				Yes			No	No	No
	CNN			> Cubic				Yes			Yes	No	No
	ART			Quadratic				Part.			Yes	Yes	Yes
	SOM			Quadratic				Part.			Yes	Yes	No

Table 2.10: Comparison of the machine learning approaches

While all of the described machine learning approaches can be applied to the classification phase, only the Rocchio method, SVM, kNN and NB are viable for the identification phase. Especially none of the considered neural network approaches can be applied to one-class classification. As the field of text categorization only considers one-class classification as an edge case, most of the approaches in this field are tailored to multi-class classification problems. Most of the approaches satisfy the requirements that were formulated for training and prediction time. However, the neural network approaches FNN, RNN and CNN are infeasible due to a training time, that is significantly above cubic complexity. The kNN approach is the only classification method, that requires a prediction time higher than linear. The ART network and the SOM generally achieve the best results, only having a slight disadvantage regarding their inability to consider word order. From the three methods, that are able to incorporate word order in their classification model, the NTC is the only one with a sufficient training time.

3 Concept

This section covers the different approaches for source code knowledge extraction, that were developed in the course of this thesis. Here, the identification phase and the classification phase are considered separately. For both of these phases, this thesis proposes two different approaches. Figure 3.1 shows the two phases as well as their input and output artifacts.

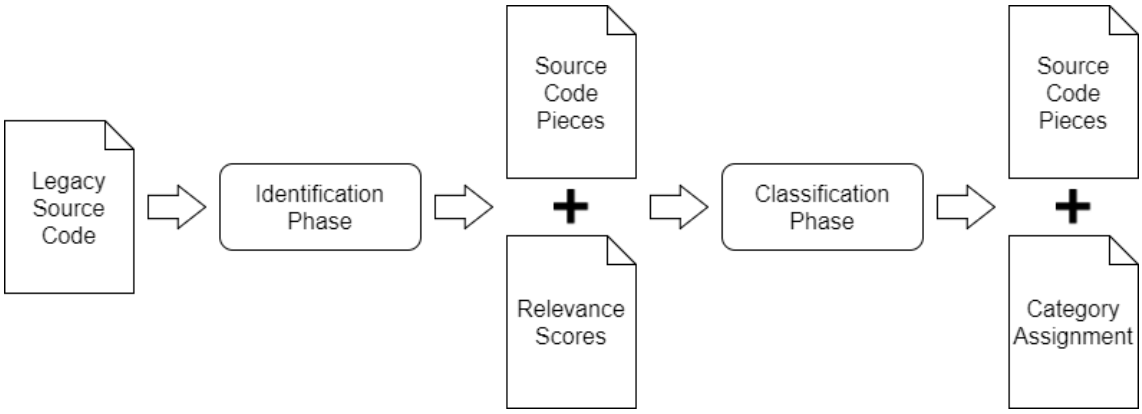

Figure 3.1: Input and output artifacts of the identification and classification phase

3.1 Identification Phase

In the identification phase, contiguous code pieces that describe a concept from the application domain, which can be understood by experts from this domain, need to be determined from a legacy code base. As stated in section 1.2, many programming languages already define syntactic constructs, which contain strongly related information. In object-oriented programming languages, this includes functions, classes, interfaces or enumerated types. However, also a sequence of statements, that describes a specific mechanism, can be a relevant code piece.

Thus, the proposed approaches will be evaluated based on their ability to a) extract code pieces that represent syntactic constructs of the programming language, such as functions, classes or interfaces, and b) determine other contiguous code pieces, that describe a concept from the application domain. In the following, these two types of code pieces will be termed syntactic and semantic code pieces, respectively.

Both proposed approaches assume, that code pieces are sequences of lines in a source code file. Therefore, a code piece cannot contain only part of a line. While this method may not be able to consider all kinds of relevant code pieces, it is likely to be sufficient for source code files, that are formatted appropriately. Further, this thesis only considers code pieces with a minimum length of three lines. This is based on the assumption, that very short code pieces, like one or two statements, do not contain enough information to describe a concept from the application domain. Popular code formatting guides, like PEP 8 [98] or C Elements of Style [99], strongly recommend to put each statement on a new line. In a properly formatted code file, two lines cannot contain more than two statements and thus, are not sufficient to describe a concept from the application domain.

3.1.1 Approach 1 – Parsing

A naive approach for the identification phase only considers the syntactic code pieces. Here, the legacy source code is parsed to extract all code pieces, that represent a given syntactic construct. While this approach is able to identify the syntactic code pieces perfectly, it cannot determine any relevant semantic code pieces. However, since the results of this naive approach are predictable, it is well suited as a baseline for the evaluation of classification phase approaches. In the application to the code base of SharpDevelop, which is written in C#, classes, methods, interfaces, structs and enumerated types are considered. Figure 3.2 shows this naive approach for the identification phase.

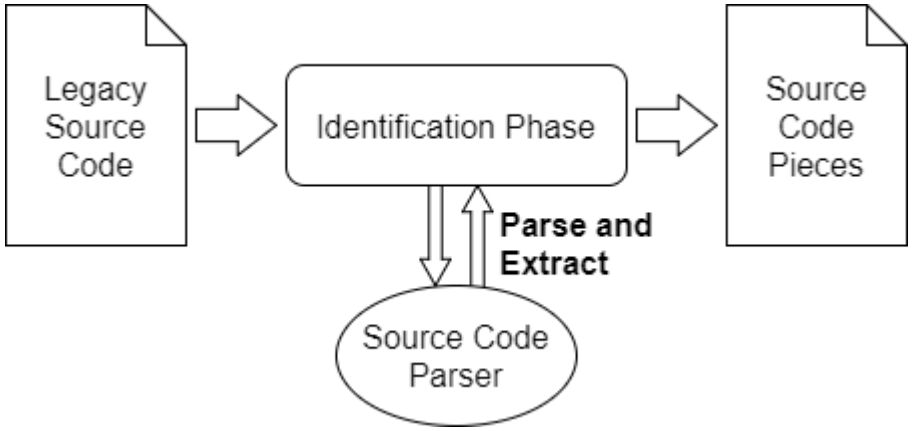

Figure 3.2: Parsing approach for the identification phase

3.1.2 Approach 2 – One-class Classification with StackOverflow

The second proposed approach for the identification phase utilizes machine learning classifiers to predict the relevant code pieces. This introduces two main challenges: a) the acquisition of training data in the form of examples for relevant code pieces and b) learning a classifier with only positive training examples, which is known as the problem of one-class classification.

As stated in section 2.2, the manual construction of a set of training examples is a very time-consuming and expensive task. Thus, a collection of already labeled training examples needs to be found. In the field of text categorization, many training and evaluation collections are freely available. This includes, for example, the Reuters and Ohsumed corpora. However, there are no such standardized collections for the classification of source code.

To solve this problem, the authors of [21] utilized user posts from the popular Q&A site StackOverflow for the task of malware capability inference. StackOverflow contains technical discussions on programming, APIs and operating systems [100]. Often, the user posts contain source code pieces, that can be used as training examples for a classification algorithm. With 440 000 users, around one million questions and 2.8 million answers, a large amount of training data is available. Further, each question is associated with tags, that can be used to limit the training set to specific domains or to label the code pieces with regard to a category. This category can also be defined as a combination of tags or other attributes, like the title of a question or the body of a post. However, as

all StackOverflow questions and answers are generated by its users, the quality of these posts cannot be assured and noise should be expected. To measure the quality of each post and filter out the noisy ones, two different ratings can be utilized. Each post has a score, which is based on upvotes and downvotes by other users. Posts that contain noisy data are likely to have a low or even negative score. Further, each user has a reputation score, which can be used to estimate his overall proficiency. [101]

Second, there is the problem of finding an appropriate classifier for one-class classification. As shown in section 2.5, four different classification methods are applicable to this problem. These are the Rocchio method with the parameter γ set to 0, one-class SVMs, one-class kNNs and one-class NB classifiers. However, none of these classification methods is able to consider word order in their input documents. Arguably, word order is of significant importance for the identification of relevant code pieces. Thus, n-grams will be constructed from the code pieces before building the numerical vector representations. This way, the word order can be partially incorporated into the classifier.

This thesis considers two different characteristics of code pieces pivotal for the identification phase. First, all terms occurring in the code piece contain information about the concepts behind this code. This information is especially important to determine relevant semantic code pieces. Second, the boundaries of a code piece are important for identifying both syntactic and semantic code pieces, as any contiguous code piece is basically defined by its boundaries. To encode information about the boundaries into a numerical vector representation, a start and end token will be added to each code piece. When combined with the use of n-grams, a classification algorithm can learn to identify relevant code pieces based on common start and end.

However, the construction of n-grams worsens the problem of high dimensionality and sparsity for numerical vector representations. This makes the application of feature selection indispensable. In section 2.4.2, several feature selection methods were outlined. When used for one-class classification, many of these methods are impractical for various reasons. The DF scoring function removes the features, that appear in many training examples. In one-class classification problems, this approach is counterproductive, as features are generally more impactful when appearing very often. Other feature selection methods, that use information about category assignment of the training examples, are also rendered useless in one-class classification problems, as they would rank each feature equal. Thus, the scoring functions IG, MI, CHI and EL are not applicable to the identification phase. This leaves the scoring function TS and the feature construction

method LSI to be applied in this phase. Additionally, an inverted DF scoring function, that ranks common features in the training set higher, will be implemented.

Another variant of this approach uses some of the syntactic information, which is provided by a source code parser. Here, several syntactic tags are added to the code pieces before they are used in training or prediction. This includes tags for the start and end of syntactic units, like functions, classes or structs. However, this preprocessing step also requires a large amount of time in the learning phase, as all code pieces, that are used for training, need to be parsed.

In Figure 3.3, the one-class classification approach for the identification phase is shown.

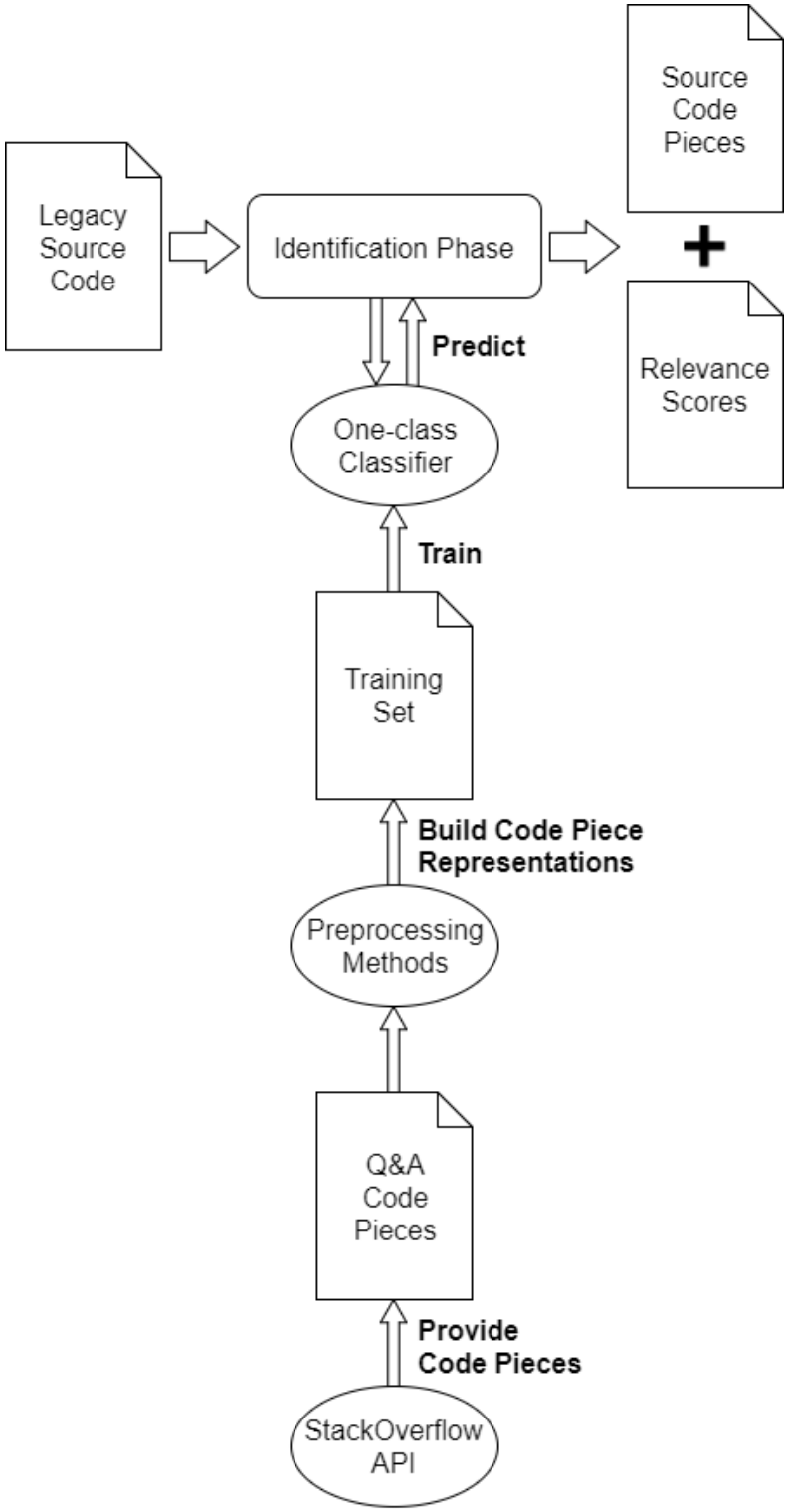

Figure 3.3: One-class classification approach using StackOverflow

3.2 Classification Phase

After a set of relevant code pieces was built in the identification phase, these code pieces now need to be assigned to different categories. The extent of a category can range from specific requirements specifications, such as user stories, to more general areas of responsibility, such as UI or persistence logic. The evaluation of existing machine learning approaches in section 2.5. has shown, that all considered approaches are applicable to the classification phase. In the following, a rationale for the choice of the classification methods will be given.

While FNN, RNN and CNN allow the construction of a very complex model, their training algorithms do not scale well to high-dimensional input spaces and large training sets. Thus, these three approaches will not be considered further. Similarly, the prediction time of a kNN suffers from high-dimensional inputs and large training sets. Further, decision trees will be disregarded due to their inability to learn high-dimensional input spaces with sparse data and their proneness to overfitting. The NTC has been discarded, as currently no implementation of this special neural network is available. From the two linear classification methods, the Rocchio method has shown to be generally superior to linear or logistic regression, with the only exception being its lack of incremental learning capability. This disadvantage is in turn compensated by the short training time of a Rocchio classifier. Consequently, the Rocchio method is the linear classification method, that will be implemented in this concept. As for nonlinear classifiers, SVM and NB will be applied to supervised classification and the ART network and SOM will be used for the unsupervised approach.

Both proposed approaches for the classification phase include the construction of n-grams, as none of the selected classification methods is able to consider the word order in its input documents. To reduce the high-dimensional input space caused by the application of n-grams, various feature selection methods will be implemented. This includes DF, IG, MI, CHI and TS as term removal methods and LSI as a feature construction method.

3.2.1 Approach 1 – Supervised Classification with StackOverflow

The identification phase approach in section 3.1.2. uses code pieces from posts on the Q&A site StackOverflow as its training data. Each question on StackOverflow is associated with several tags, which can be used to label the code pieces. In the classification phase, categories can be built based on the tags and other attributes of a post, such as the title of a question or the body of a post. With this labeled training set, a multi-class machine learning classifier can be trained. For this approach, a multi-class SVM implementation, a Rocchio classifier and a NB classifier will be implemented. Figure 3.4 shows the supervised classification approach using StackOverflow posts and their associated tags as the training data.

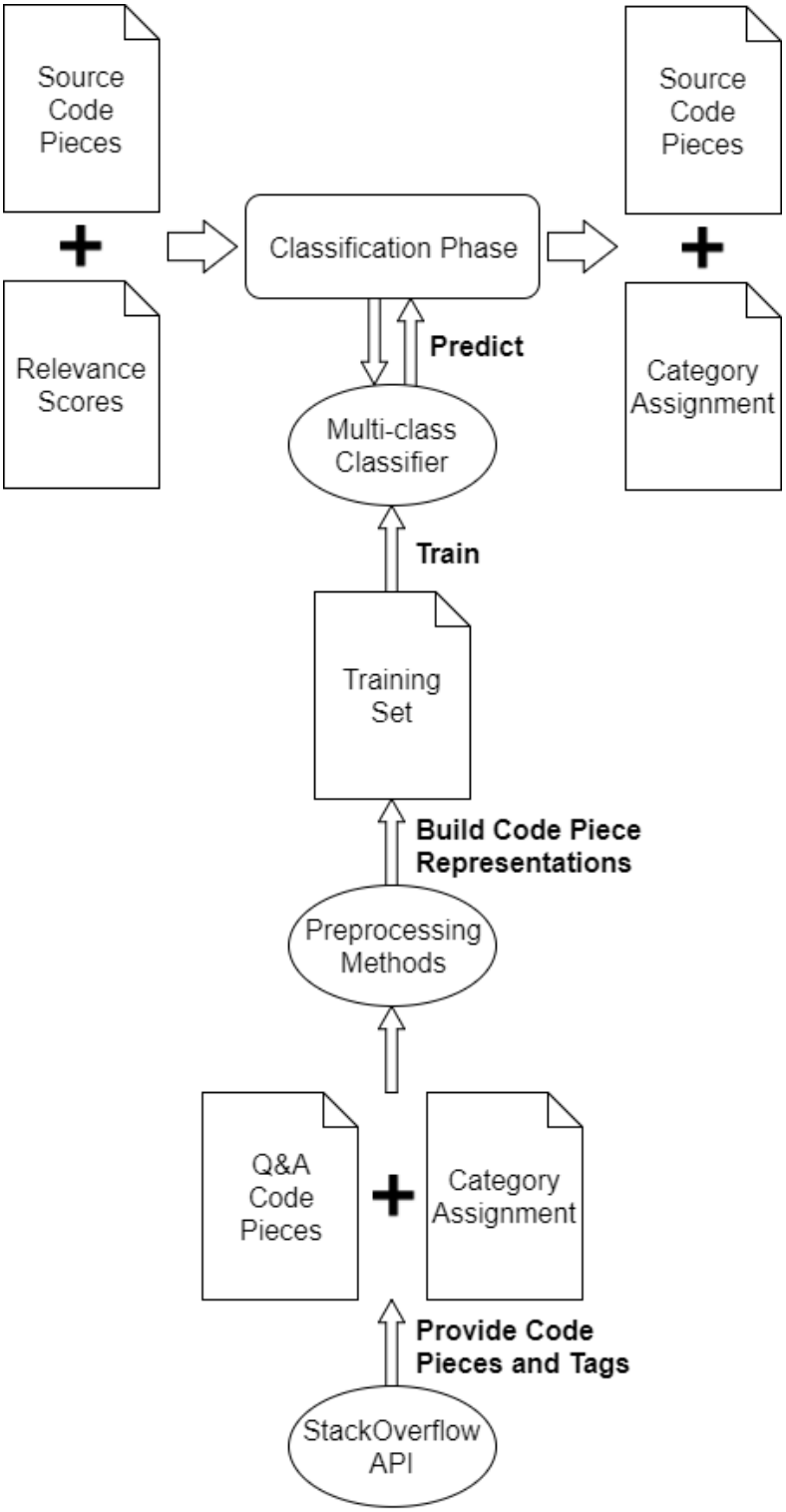

Figure 3.4: Supervised approach to the classification phase

3.2.2 Approach 2 – Self-organizing Learning

The ART network and the SOM are unsupervised learning methods and thus, can be applied without defining the categories beforehand. In their learning process, they build clusters of semantically related documents. These clusters can be used by a human expert to quickly assign categories to the code pieces. Both the ART network and the SOM allow adjustments to the extent of the clusters through the similarity threshold and the grid size, respectively.

This approach uses the code pieces from StackOverflow to build clusters, that are later used for prediction. After the learning phase, categories are constructed from the clusters based on the tags of their corresponding training examples. This unsupervised approach to the classification phase is depicted in Figure 3.5.

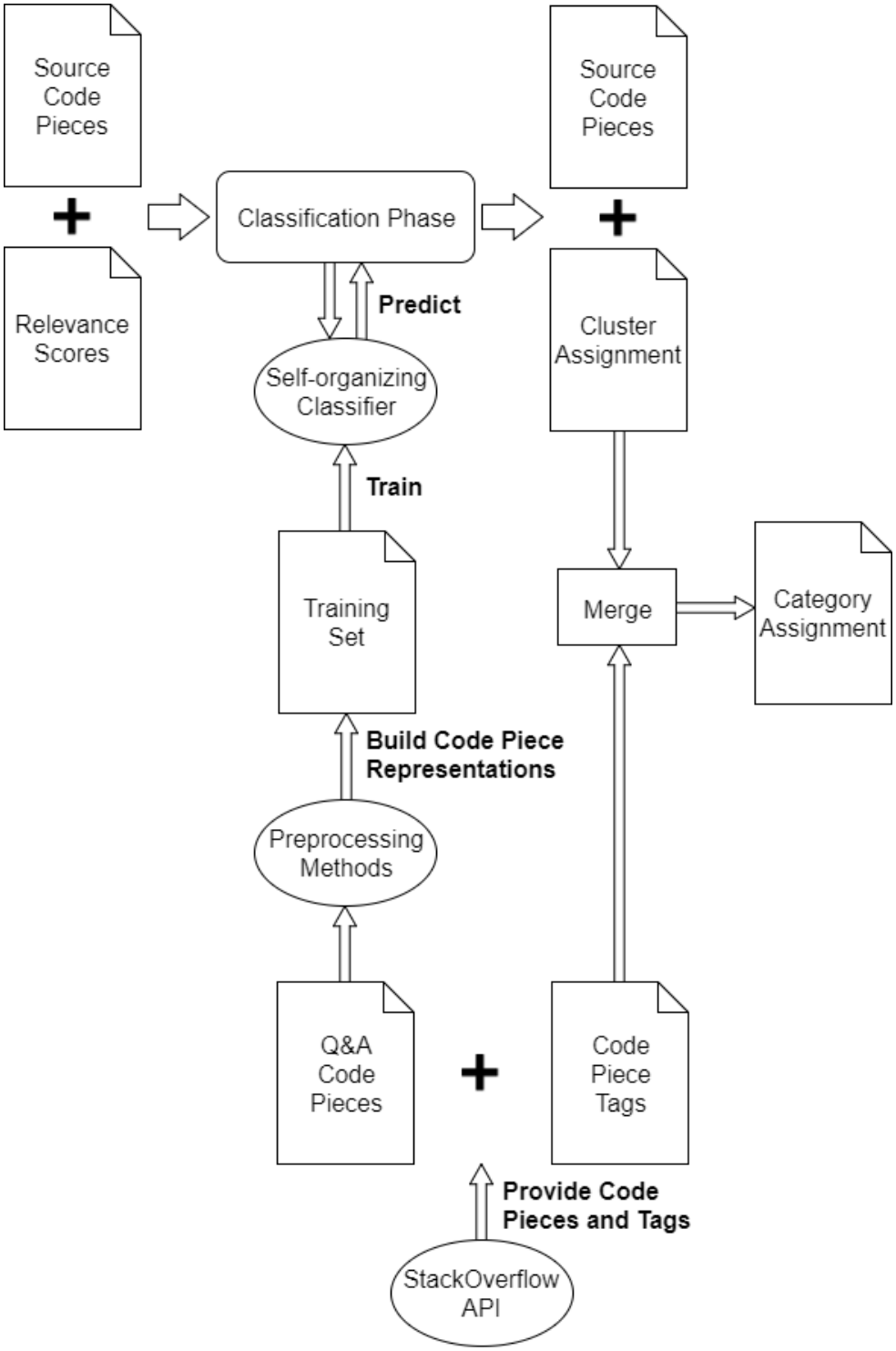

Figure 3.5: Unsupervised approach to the classification phase using self-organizing learning

4 Implementation

The different approaches, that were proposed in the last section have been implemented in the course of this thesis. This section comprises a short reasoning about the used technologies, details about the implemented prototype as well as a deployment guide.

4.1 Used Technologies

Python is one of the most popular programming languages in the area of machine learning [102] with a manifold of libraries being available. Consequently, it was chosen as the programming language for the implementation of the proposed concept. As the basis for the architecture of the prototype, the Django library was used. This framework allows fast development of a web application from a high-level perspective based on the model-view-controller design pattern. On the database side, an instance of Microsoft SQL Server was used due to personal preference and experience.

Two of the proposed approaches require code pieces from posts on StackOverflow as their training examples. Thus, the questions and answers need to be retrieved from StackOverflow and stored in the database. The py-stackexchange library is used as a Python binding to the StackOverflow API.

For the application of different feature selection and classification methods, the scipy and scikit-learn libraries were used. These two libraries support most of the required methods, including implementations of multi-class and one-class SVMs, a multi-class NB classifier and a multi-class Rocchio classifier as well as the feature selection methods MI, CHI and LSI. Implementations of a one-class Rocchio and a one-class NB classifier and the feature selection methods DF, IG and TS have been developed in the prototype.

The proposed approaches for the identification phase require a source code parser. As the evaluation in section 5. uses the code base of SharpDevelop, which is written in C#, Microsoft's compiler platform Roslyn was chosen for this task. Since no Python bindings for Roslyn exist, a small standalone C# console application has been developed for the

identification of syntactic code pieces and the optional tagging of syntactic units in the one-class classification approach.

4.2 Implementation Details

Here, details concerning the implementation of the prototype will be described, including three different components: the data model, the StackOverflow component, the classification component and the parsing component. Figure 4.1 shows an overview over the architecture of the implemented prototype and the interaction between the different components.

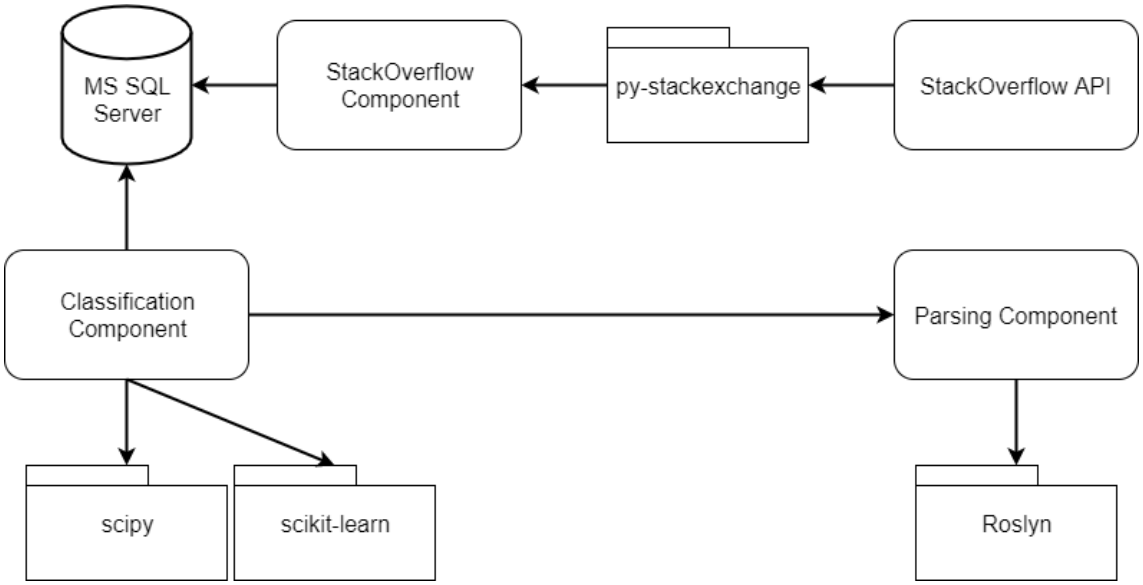

Figure 4.1: Architecture of the prototype

4.2.1 Data Model

The following class diagram shows the data model of the prototype, which is also the basis for the database scheme. This model is mostly based on the data provided by the StackOverflow API, with the exception of the CodePiece class, that is used to store the extracted code pieces.

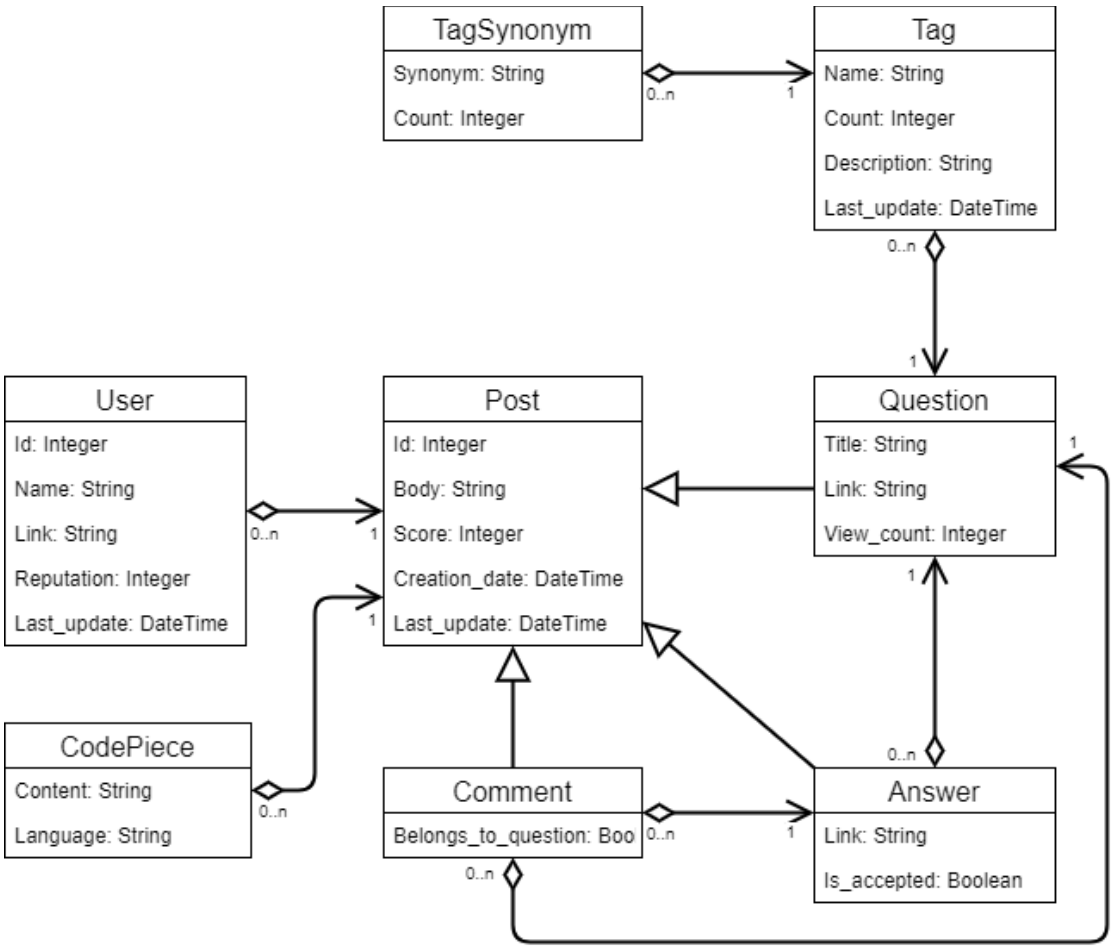

Figure 4.2: Data model of the prototype

4.2.2 StackOverflow Component

The StackOverflow component is concerned with the retrieval of the required data from the StackOverflow API. For this task, the py-stackexchange library is used. The prototype contains two different retrieval functions, that should be used consecutively. First, all tags and synonyms need to be fetched from the StackOverflow API and stored in the database. After this, the retrieval of questions as well as their answers and comments can be started. This is based on the creation date of the questions, so that the oldest questions are fetched first. A new retrieval process on an already populated database can

then avoid requesting the same questions again by starting with the date of the newest question in the database.

Several constraints are imposed on the usage of the StackOverflow API. The number of requests, that can be put from a single IP, is limited to 30 per second. If this condition is violated, then a temporary ban is imposed on this IP, ranging from 30 seconds to a few minutes. Another constraint limits the number of total requests an application can send per day. Without any application key, only 300 requests are permitted. This limit increases to 10000 requests per day, when the application key is provided [103]. Application keys can be registered by users of the StackExchange network at https://stackapps.com/apps/oauth/register. The prototype currently contains two different application keys, so that 20000 requests per day can be executed. Through the use of bulk requests, up to 100 objects can be fetched per request. This way, approximately 33000 questions as well as their answers, comments and authors can be retrieved every day.

4.2.3 Classification Component

In the classification component, the various preprocessing methods as well as the classification methods have been implemented. It is designed to allow an easy combination of different methods. The general flow for preprocessing an input document and training of a classifier is shown in Figure 4.3.

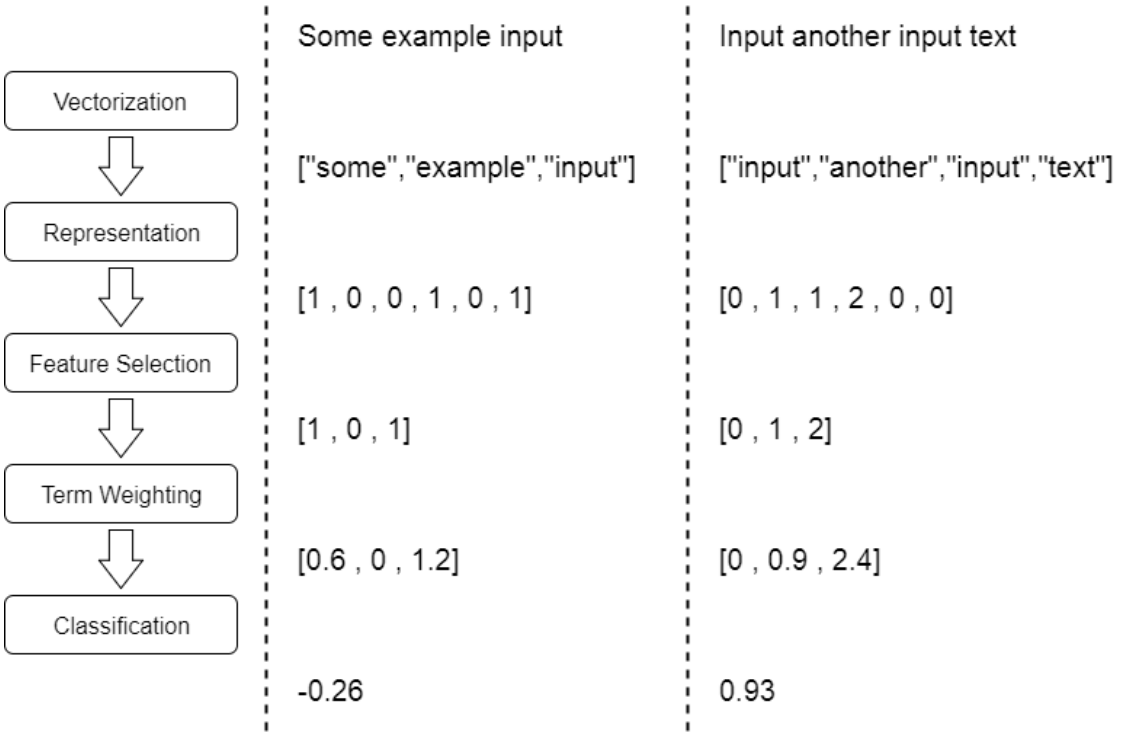

Figure 4.3: Preprocessing and classification steps

The content of a code piece is first split into terms to obtain a first vector representation. In this step, also the n-grams are added, depending on the selected n-gram size. Currently, every word is considered as a single term and the split is executed on non-alphabetic characters. Once the term string vectors for all training examples are gathered, they are represented as numerical vectors, with each dimension corresponding to one term. As this leads to very large and sparse vectors, the memory requirements to store these representations become infeasibly high once more than a few thousand training examples and n-gram sizes above three are used. Thus, the numerical vectors are stored as a compressed sparse row (CSR) matrix, that only manages the nonzero values and their indices [104]. Figure 4.4 shows an example for this representation form. This matrix is then optionally processed further with the chosen feature selection and term weighting methods. Finally, a classifier is trained on the representation matrix and the corresponding category assignments.

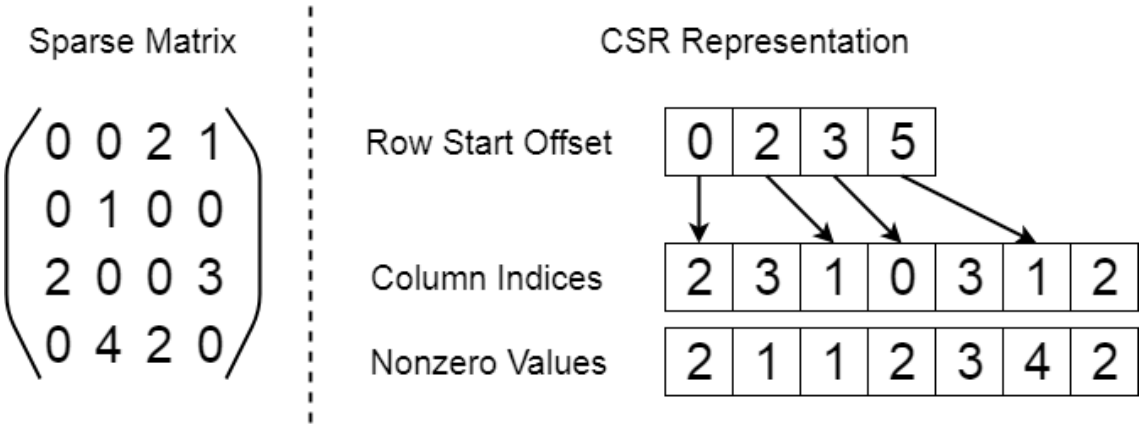

Figure 4.4: CSR matrix

For the prediction phase, a legacy source code file needs to be split into code pieces. As stated in section 3.1., this thesis only considers sequences of at least three lines as code pieces. Thus, all possible code pieces matching this criterion, namely all line sequences with a length of at least three lines, are extracted from the code file and used as an input for the prediction. The number of extracted code pieces equals $\frac{(L-1)*(L-2)}{2}$, where L is the number of lines in the file. Therefore, a large number of prediction operations needs to be performed. Depending on the chosen classification method, outputs can either be continuous, ranging between -1 and 1, or binary.

The implemented feature selection methods include DF, CHI, MI, IG, TS, LSI and an inverted DF. As term weighting methods, binary weighting, tf*idf, CHI, MI and IG have been implemented. Available classification methods in the prototype include one-class SVM (RBF kernel, nu=0.8, gamma=0.1), one-class Rocchio, one-class NB, multi-class SVM(RBF kernel, c=1), multi-class Rocchio, multi-class NB (multivariate Bernoulli model) and an ART network.

A wrapper class for the classification process allows the specification of the feature selection, term weighting and classification method as well as the desired n-gram size, the number of features to select, the normalization that should be applied to the numerical vectors and a flag for the addition of start and end tags to each code piece.

The retrieval of code pieces from the database has shown to be a bottleneck. Thus, a cache file is generated at the start of a supervised identification or classification process, containing the code pieces that were retrieved from the database. If a new identification

or classification process is started with the same tags filter, the cache file is used instead of the database.

4.2.4 Parsing Component

As the proposed approaches for the identification phase require a source code parser, a parsing component has been implemented in the prototype. Since the code base of SharpDevelop, that is used in the evaluation, is written in C#, Roslyn has been used as the parsing library. The parsing component has been developed as a C# console application, as Roslyn does not support python bindings. This component includes two main functionalities: the extraction of syntactic code pieces for the first identification phase approach and the tagging of syntactical units as a preprocessing step in the second identification phase approach. In both cases, the code, that should be processed, is passed to the application as a command line argument, along with the action that should be performed. Also, both functionalities consider classes, methods, constructors, structs, interfaces, enums and events as the syntactic units, that should be extracted or tagged. The extraction process returns the start and end positions of all found syntactic units, which are then evaluated in the classification component of the Python application. When tagging is selected as the desired action, alphabetic tokens are added at the start and end of each class, method, constructor, struct, interface, enum or event declaration as well as at the beginning of each method, constructor or event body. These tokens are specific to the syntactic units, so that, for example, classes are tagged differently than methods. After the syntactic tags are added, the resulting preprocessed code is returned to the Python application.

5 Evaluation

The approaches, that were developed and implemented in the course of this thesis, have been evaluated on the codebase of SharpDevelop. Here, the identification phase and classification phase are treated separately. The naive parsing approach to the identification phase is used as a baseline for the evaluation of the classification phase approaches. In this section, the results of this evaluation are described.

SharpDevelop is an open-source IDE, which is written in C#. It contains a large variety of different functionalities, such as UI components, search algorithms or parsers. The code base of SharpDevelop is available under the MIT license. All C# source code files in this code base amount to a total of 5066 files.

5.1 Evaluation Metrics in Text Categorization

In the field of text categorization, evaluation of different approaches is usually conducted with collections of already labeled texts, so-called corpora. Many of these corpora, such as the Reuters and Ohsumed corpora, are freely available on the Internet. These labeled text collections are then split into a training set and a test set. The test set is used to calculate various quality metrics of the different approaches. [105]

Two of the most important quality metrics for machine learning classifiers are precision and recall. Precision measures the percentage of correctly classified input documents based on the number of positively predicted documents. On the other hand, recall determines the percentage of all positively labeled documents from the test set, that were also predicted to the positive category. The following two formulas show the calculation of these two metrics [6,27,38]:

$$recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

$$precision = \frac{true\ positives}{true\ positives + false\ negatives}$$

Usually, there is a tradeoff between precision and recall, so that maximizing precision decreases the recall and the other way around. However, a classifier is only useful, if it achieves both high precision and high recall. Thus, these two metrics are commonly combined in the evaluation of machine learning approaches. One possibility for this is the calculation of the precision at different recall thresholds, such as 0%, 10%, 20% etc. up to 100% [38]. These precision values are then averaged for each test document to obtain an overall precision score [27]. Another widely used combination of precision and recall is the F1 measure. This metric is defined as follows [71,106]:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

A large variety of other metrics for machine learning approach, such as accuracy or error, exist [36]. However, these metrics are not as commonly used as precision, recall and their combinations.

5.2 Evaluation in Source Code Knowledge Extraction

Unlike the field of text categorization, source code knowledge extraction does not have existing corpora for the evaluation of the machine learning approaches. Thus, the following sections will define their specific quality measures, specific to their approaches.

In addition to the quality measurements, the amount of time required by the different approaches will be determined. This is measured on a computer running Windows 8.1 with 16 GB RAM and an Intel Core i7-4510U 2GHz dual core processor.

For the two approaches using posts from StackOverflow, 88028 questions and 435070 answers were retrieved from the StackOverflow API and stored in the database. From these posts, 188761 code pieces were extracted, of which 32733 are written in C#. These code pieces constitute the training set for the two StackOverflow approaches.

5.3 Identification Phase

The approaches for the identification phase have been evaluated on the code base of SharpDevelop. In the following, the results of this evaluation are shown.

5.3.1 Approach 1 – Parsing

The naive approach to the identification phase facilitates a source code parser to identify code pieces. While this approach is guaranteed to return only correct code pieces, it can only consider syntactic code pieces and disregards all semantic code pieces. Currently, classes, methods, constructors, structs, interfaces, enums and events are extracted by the parsing component.

In the prototype, the content of each source file is passed to the parsing component as a command line argument. This only works up to a certain source file size, as length of a command line argument in Windows is limited to 8191 characters [107]. From the 5066 C# source code files in the code base of SharpDevelop, 4450 files were successfully processed by the parsing component. In this process, 24998 syntactic code pieces were extracted within 28 minutes.

5.3.2 Approach 2 – One-class Classification with StackOverflow

In the second approach for the identification phase, different one-class classifiers have been trained on code pieces from posts on StackOverflow. The prototype supports a one-class SVM, a one-class Rocchio classifier and a one-class NB classifier. As these classification methods do not incorporate the word order from their training examples, ngrams with a length of 2 and 3 terms have been added to the term string vectors during the vectorization. Feature selection was conducted through the inverse DF filter method or LSI. Here, 200 or 1000 features were extracted in different iterations. All iterations include the addition of start and end tags to the code pieces used in training or prediction phase. Further, the addition of syntactic tags has been tested with each of the three approaches.

The quality metrics, that were described in section 5.1, cannot be directly applied to this approach, as the correct code pieces are not known and thus, no test set is available. Instead, the identified code pieces were investigated manually to assess their usefulness and to detect the tendencies of the used classifiers. Further, as a quality measure for the one-class approaches, the precision of the 50 highest ranked code pieces is calculated. If the prediction algorithm supports only binary outputs, 50 random code pieces from the positive prediction set are selected to calculate this measure.

Regardless of the used classification method, the time needed to identify the code pieces from one source code file averages 4 min. The majority of both training and prediction time is used in the construction of the numerical input vectors and in the feature selection.

One-class SVM

The identified code pieces from the one-class SVM were very similar regardless of the used feature selection method, the amount of features or the presence of syntactic tags. Many of the code pieces were not useful, as they mixed parts of comments and source code together or only contained arbitrary parts of syntactical units. Generally, the identified code pieces were very short, ranging from three to ten lines of code. While the prediction contained almost all code pieces with three to five lines, longer code pieces were more seldom. Also, these longer code pieces tended to be more relevant and often comprised the start of a function declaration. This tendency shows, that the one-class SVM was able to learn the start of syntactic code pieces to a certain extent. However, overall the set of predicted code pieces was still very noisy and impractical. Neither syntactic nor semantic code pieces could be reliably extracted from the source files. As the predictions of a one-class SVM are binary, 50 random identified code pieces were examined. Of these 50 code pieces, only 5 described a useful semantic or syntactic code piece, which equals a precision of 10%. Listing 5.1 show examples for identified code pieces with the one-class SVM.

```
/// </summary>
[TypeConverter(typeof(DirectoryNameConverter))]
public sealed class DirectoryName : PathName
{
```

```
public static DirectoryName Create(string DirectoryName)
{
    if (string.IsNullOrEmpty(DirectoryName))
        return null;
    else
```

```
    object value, Type destinationType)
{
       if (destinationType == typeof(string)) {
         return value.ToString();
```

```
public static Font LoadFont(string fontName, int size, FontStyle style,
GraphicsUnit unit)
{
    try
    {
        return new Font(fontName, size, style, unit);
    }
    catch (Exception ex)
    {
        LoggingService.Warn(ex);
        return SystemInformation.MenuFont;
    }
}
```

Listing 5.1: Examples for identified code pieces from the one-class SVM

One-class Rocchio Method

The one-class Rocchio classifier showed a better prediction quality than the one-class SVM. Here, different results were obtained between the two feature selection methods LSI and inverse DF. Rocchio classifiers trained with LSI features were able to identify some of the methods from the source code. When inverse DF was used for feature selection, the results contained more semantic code pieces. The set of identified code pieces is still very noisy, as regardless of the parameters only 12 to 14 of the 50 highest ranked predictions depicted relevant code pieces. This results in a precision of approximately

26%. The use of syntactic tags had no significant impact on the prediction results. Listings 5.2 and 5.3 show several identified code pieces from a Rocchio classifier using LSI or inverse DF, respectively.

```
/// <summary>
/// Creates a DirectoryName instance from the string.
/// It is valid to pass null or an empty string to this method (in that case,
a null reference will be returned).
/// </summary>
public static DirectoryName Create(string DirectoryName)
{
    if (string.IsNullOrEmpty(DirectoryName))
        return null;
    else
        return new DirectoryName(DirectoryName);
}
[Obsolete("The input already is a DirectoryName")]
public static DirectoryName Create(DirectoryName directoryName)
{
    return directoryName;
}
```

```
public static Font DefaultMonospacedFont
{
    get
    {
        if (defaultMonospacedFont == null)
        {
            defaultMonospacedFont =
                LoadDefaultMonospacedFont(FontStyle.Regular);
        }
        return defaultMonospacedFont;
    }
}
```

```
public static bool operator !=(DirectoryName left, DirectoryName right)
{
    return !(left == right);
}
```

Listing 5.2: Examples for identified code pieces from the one-class Rocchio classifier with feature selection through LSI

```
if (iconobj is Icon) {
    ico = (Icon)iconobj;
} else {
    ico = BitmapToIcon((Bitmap)iconobj);
}
iconCache[name] = ico;
return ico;
```

```
get {
    if (defaultMonospacedFont == null) {
        defaultMonospacedFont =
            LoadDefaultMonospacedFont(FontStyle.Regular);
       }
    return defaultMonospacedFont;
}
```

```
public override bool CanConvertFrom(ITypeDescriptorContext context, Type
sourceType)
{
    return sourceType == typeof(string) ||
           base.CanConvertFrom(context, sourceType);
}
```

Listing 5.3: Examples for identified code pieces from the one-class Rocchio classifier with feature selection through inverse DF

One-class NB Classifier

The implemented one-class NB classifier only uses binary feature values. Thus, LSI is not considered for feature selection with this classification method. However, with all used parameter combinations, the one-class NB approach did not yield useful code pieces. Highly ranked code pieces are usually very short, with three to five lines of code, and often only contain long comments. Of the 50 predictions with the highest ranking, none represented a useful syntactic or semantic code piece, leading to a precision of 0%. Listing 5.4 shows examples for code pieces, that were identified by the one-class NB classifier.

```
// PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE
// FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CON-
TRACT, TORT OR
// OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
USE OR OTHER
// DEALINGS IN THE SOFTWARE.

---

/// <param name="name">
/// The name of the requested bitmap.
/// </param>
/// <exception cref="ResourceNotFoundException">

---

    } finally {
        NativeMethods.DestroyIcon(hIcon);
    }
}
```

Listing 5.4: Examples for identified code pieces from the one-class NB classifier

5.4 Classification Phase

In this section, the two proposed approaches for the classification phase are evaluated on the code base of SharpDevelop. The naive parsing approach for the identification phase is used to provide the code pieces, that are classified in the following approaches.

5.4.1 Approach 1 – Supervised Classification with StackOverflow

In the first approach for the classification phase, code pieces from posts on StackOverflow have been used to train multi-class classifiers. Here, a multi-class SVM, a multi-class Rocchio classifier and a multi-class NB were implemented. Categories are built based on the tags of StackOverflow posts. In this evaluation, the tag “winforms” has been used as a category and the classifiers have been learned on a subset of the SharpDevelop code base, including the projects “Core”, “ICSharpCode.Core.WinForms“ and “ICSharpCode.SharpDevelop.Widgets”. The 714 code pieces, that were identified by the parsing component, have been manually analyzed and 198 of them have been recognized as members of the category “winforms”. Based on this analysis,

prediction and recall as well as the F1 measure are computed for each classification method.

As a feature selection method, LSI has been used for the multi-class SVM and the Rocchio method and IG has been used for the multi-class NB classifier. Both methods were set to select 200 features.

For the prediction of the categories, all approaches require a time of approximately five minutes.

Multi-class SVM

The multi-class SVM approach assigned 151 code pieces to the category "winforms", of which 112 were true positives and 39 were false positives. 86 of the 198 code pieces, that belong to the category "winforms", were not found by the SVM. Thus, this method has a recall of 57% and a precision of 74%. The F1 measure equals to 0.64.

Multi-class Rocchio Method

Of the 210 code pieces, that were predicted to the positive category by the multi-class Rocchio classifier, 109 were true positives and 101 were false positives. From the 198 code pieces in the category "winforms", 89 were not found by this classification method. This results in a recall of 55% and a precision of 52%, with an F1 measure of 0.53.

Multi-class NB

The NB classifier assigned 296 code pieces to the positive category, with 148 being true positives and 148 being false positives. 50 of the 198 code pieces in the category "winforms" were not found by this classifier. The resulting recall equals 75% and the precision amounts to 50%, which leads to a F1 measure of 0.60.

The diagram in Figure 5.1 shows the precision and recall of these three methods on the test set.

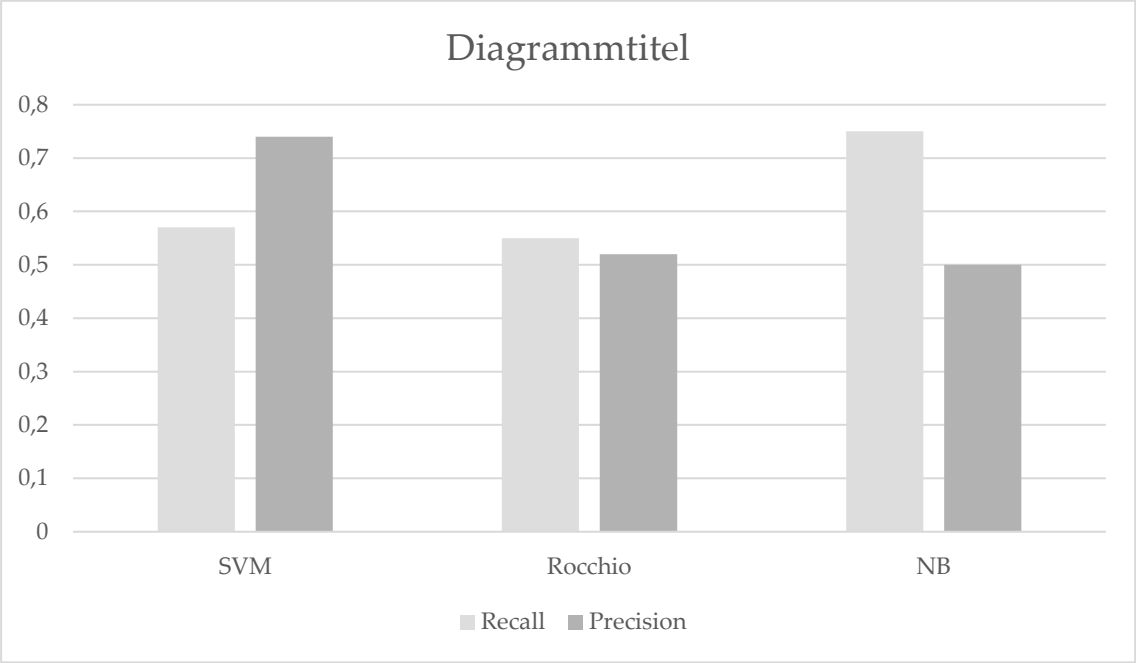

Figure 5.1: Recall and precision of the three multi-class approaches

The SVM and NB approach were clearly superior to the Rocchio method. With these two approaches, the tradeoff between precision and recall can be seen, as the SVM has a higher precision than the NB classifier and is able to predict more accurately, while the NB approach is able to recognize a higher percentage of the total amount of code pieces in the category "winforms".

5.4.2 Approach 2 – Self-organized Learning with StackOverflow

The second proposed approach for the classification phase trains an ART network on the code pieces, that were retrieved from StackOverflow. The ART network builds clusters based on these code pieces and then assigns new code pieces in the prediction phase to the most similar cluster. The different clusters then need to be associated with a category. Therefore, the associated tags of the training code pieces can be used to determine, if a cluster correlates with one or multiple tags. For the evaluation, the ART network was trained with eight output clusters and a similarity threshold of 0.002.

From the eight provided clusters, only three were strongly populated, while the other clusters only contained a few code pieces or were entirely empty. Each of the three populated clusters tended towards code pieces from a specific subdomain of the application. Cluster 1 comprised many code pieces, that described core functionalities of SharpDevelop. Cluster 2 tended to include utility and helper functions or classes. And finally, cluster 3 contained a large amount of UI functionality, which also includes many of the code pieces, that were assigned to the category "winforms" in the supervised classification approach in section 5.4.1. However, these three clusters failed to clearly separate the different subdomains. Listings 5.5, 5.6 and 5.7 show exemplary code pieces from the three strongly populated clusters.

```
public void ConfigureUserAddIns(string addInInstallTemp, string userAddInPath)
{
    if (!externalAddInsConfigured)
    {
        throw new InvalidOperationException("ConfigureExternalAddIns must be
called before ConfigureUserAddIns");
    }
    AddInManager.AddInInstallTemp = addInInstallTemp;
    AddInManager.UserAddInPath = userAddInPath;
    if (Directory.Exists(addInInstallTemp))
    {
        AddInManager.InstallAddIns(disabledAddIns);
    }
    if (Directory.Exists(userAddInPath))
    {
        AddAddInsFromDirectory(userAddInPath);
    }
}
```

```
public CoreStartup(string applicationName)
{
    if (applicationName == null)
        throw new ArgumentNullException("applicationName");
    this.applicationName = applicationName;
}
```

Listing 5.5: Examples for code pieces in cluster 1

```
void RemoveAt(int i)
{
    int lastIndex = innerList.Count - 1;
    innerList[i] = innerList[lastIndex];
    innerList.RemoveAt(lastIndex);
}
```

```
public bool Equals(int i, ISequence other, int j)
{
    StringSequence seq = other as StringSequence;

    if (seq == null)
        return false;

    return content[i] == seq.content[j];
}
```

Listing 5.6: Examples for code pieces in cluster 2

```
public MenuCheckBox(Codon codon, object caller, IEnumerable<ICondition> condi-
tions)
{
    this.RightToLeft = RightToLeft.Inherit;
    this.caller = caller;
    this.codon = codon;
    this.conditions = conditions;
    UpdateText();
}
```

```
protected override void OnClick(System.EventArgs e)
{
    base.OnClick(e);
    command = CommandWrapper.Unwrap(command);
    if (command != null)
    {
        MenuService.ExecuteCommand(command, caller);
    }
}
```

Listing 5.7: Examples for code pieces in cluster 3

6 Conclusions and Future Work

Legacy systems contain a large amount of valuable knowledge from many years of requirements engineering and development. However, the demand to modernize these systems rises steadily as new technologies emerge and old technologies become discontinued and deprecated. Section 1.1 has shown, that in order to modernize a legacy system and keep the valuable knowledge it contains, methods of reverse engineering and more specifically redocumentation need to be applied. Therefore, the three-phase process of source code knowledge extraction was described in section 1.2 and the benefits of applying machine learning methods to this process were outlined. However, there is still little research on the use of machine learning methods for the identification and classification of useful code pieces. Due to many similarities between source code and natural language texts, different approaches from this field of research were evaluated in the state of the art. Based on these considerations, four different approaches for source code knowledge extraction were proposed in section 3, with two approaches addressing the identification phase and the other two managing the classification phase. Here, the training sets for the different approaches are constructed based on code pieces in posts from the popular Q&A site StackOverflow. Section 4 covers the used technologies and the architecture and implementation details of the prototype, that was developed in the course of this thesis. Finally, section 5 evaluates the performance of the proposed approaches with regard to both quality and speed of the predictions.

From the three implemented one-class approaches, the Rocchio classifier has shown the best potential to extract useful code pieces. However, even using this method, the resulting set of code pieces is still very noisy. This indicates, that either more complex models are needed or that the source code needs to be further preprocessed to be learnable by the implemented classification methods. FNNs, RNNs and CNNs were not further considered in the concept, as they require a very long training time. Yet, their models are potentially better suited to learn the complex characteristics of source code. Especially RNNs and CNNs are promising methods for the identification phase, as they can incorporate the word order of the training examples into their model. Possibilities for further preprocessing methods include splitting of identifiers in the source code and the extension of the syntactical tagging. Also, instead of using all possible line sequences with at

least three lines as inputs to the prediction, these code pieces could be filtered previously to the prediction. Some of the line sequences, such as code pieces, that contain only part of a statement, can be disregarded as irrelevant.

In the classification phase, the supervised classification approach has shown a reasonable performance on the used test set. This can be further improved by the extension of the training set, as there are many more available code pieces on StackOverflow. Additionally, other characteristics of the posts, like the title and body, can be included to construct new categories, as the provided tags alone have shown to be too shallow. This construction of categories can be accomplished by the formulation of queries, which combine multiple characteristic of a post.

The ART network could not achieve a very useful result, even if the clusters showed some tendencies of grouping semantically similar code pieces. Here, the used parameters, like the number of clusters and the similarity threshold, are subject to further experimentation.

Overall the usage of code pieces from posts on StackOverflow has shown a great potential for source code knowledge extraction. The proposed approaches reveal many tendencies and indicate opportunities for the improvement of prediction quality.

Bibliography

[1] L. Feinberg, "Cloud Migration Survey 2016," CloudEndure, 2016. [Online]. Available: https://www.cloudendure.com/blog/2016-cloud-migration-survey/. [Accessed: 25-Sep-2017].

[2] M. Haselmayr, "Here's Why Your Business Needs Its Own Mobile App," Forbes.com, 2014. [Online]. Available: https://www.forbes.com/sites/allbusiness/2014/11/17/heres-why-your-business-needs-its-own-mobile-app. [Accessed: 25-Sep-2017].

[3] "Mobile Apps – Increasingly Important Part of IT Portfolio," Computereconomics.com, 2016. [Online]. Available: http://www.computereconomics.com/article.cfm?id=2229. [Accessed: 25-Sep-2017].

[4] "legacy application or system - Gartner IT Glossary," Gartner IT Glossary. [Online]. Available: http://www.gartner.com/it-glossary/legacy-application-or-system. [Accessed: 25-Sep-2017].

[5] "Definition and Characteristics of Legacy Systems," Atlantistechgroup.com. [Online]. Available: http://www.atlantistechgroup.com/Resources/Legacy/Definition.htm. [Accessed: 25-Sep-2017].

[6] A. Marcus et al., "An information retrieval approach to concept location in source code," 11th Working Conference on Reverse Engineering, 2004.

[7] M. Torchiano et al., "Empirical comparison of graphical and annotation-based re-documentation approaches," IET Software, vol. 4, no. 1, 2010.

[8] H. M. Kienle and H. A. Müller, "Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation," Science of Computer Programming, vol. 75, no. 4, 2010.

[9] R. Ferenc et al., "Columbus - reverse engineering tool and schema for C," International Conference on Software Maintenance, 2002. Proceedings., 2002.

[10] "pro et con: COBOL FGM," Proetcon.de. [Online]. Available: http://www.proetcon.de/en/migration/fgm/fgm.html. [Accessed: 25-Sep-2017].

[11] T. J. Biggerstaff, "Design recovery for maintenance and reuse," Computer, vol. 22, no. 7, 1989.

[12] C. Caprile and P. Tonella, "Nomen est omen: analyzing the language of function identifiers," Sixth Working Conference on Reverse Engineering (Cat. No.PR00303), 1999.

[13] K. Wong et al., "Structural redocumentation: a case study," IEEE Software, vol. 12, no. 1, 1995.

[14] N. E. Gold et al., "Unifying program slicing and concept assignment for higher-level executable source code extraction," Software: Practice and Experience, vol. 35, no. 10, 2005.

[15] T. J. Biggerstaff et al., "The concept assignment problem in program understanding," Proceedings Working Conference on Reverse Engineering, 1993.

[16] K. Guruswamy, "TeradataVoice: Data Science: Machine Learning Vs. Rules Based Systems," Forbes.com, 2015. [Online]. Available: https://www.forbes.com/sites/teradata/2015/12/15/data-science-machine-learning-vs-rules-based-systems/. [Accessed: 25-Sep-2017].

[17] T. Fung, "The pros and cons of crowdsourcing," Theaustralian.com.au, 2013. [Online]. Available: http://www.theaustralian.com.au/business/business-spectator/the-pros-and-cons-of-crowdsourcing/news-story/9ec9c88a62137d0e425bdfecb2c623ff. [Accessed: 25-Sep-2017].

[18] V. Zwass, "information system," Encyclopedia Britannica, 2016. [Online]. Available: https://www.britannica.com/topic/information-system. [Accessed: 25-Sep-2017].

[19] J.-L. Hainaut et al., "Migration of Legacy Information Systems," Software Evolution, 2008.

[20] E. Merlo et al., "Feed-forward and recurrent neural networks for source code informal information analysis," Journal of Software Maintenance and Evolution: Research and Practice, vol. 15, no. 4, 2003.

[21] J. Saxe et al., "CrowdSource: Automated inference of high level malware functionality from low-level symbols using a crowd trained machine learning model," 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE), 2014.

[22] K. Rieck et al., "Automatic Analysis of Malware Behavior using Machine Learning," Journal of Computer Security, vol. 19, no. 4, 2011

[23] S. Rasthofer et al., "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks," Proceedings 2014 Network and Distributed System Security Symposium, 2014.

[24] S. Ugurel et al., "Whats the code?," Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD 02, 2002.

[25] R. Krovetz et al., "Classification of Source Code Archives," Proceedings of the 26th annual international ACM SIGIR conference on Research and development in information retrieval, 2003.

[26] M. Linares-Vásquez et al., "On using machine learning to automatically classify software applications into domain categories," Empirical Software Engineering, vol. 19, no. 3, 2012.

[27] Y. Yang and J. O. Pedersen, "A Comparative Study on Feature Selection in Text Categorization," ICML '97 Proceedings of the Fourteenth International Conference on Machine Learning, 1997

[28] R. Johnson and T. Zhang, "Effective Use of Word Order for Text Categorization with Convolutional Neural Networks," Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2015.

[29] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," Information Processing & Management, vol. 24, no. 5, 1988.

[30] N. Castle, "Supervised vs. Unsupervised Machine Learning," Datascience.com, 2017. [Online]. Available: https://www.datascience.com/blog/supervised-and-unsupervised-machine-learning-algorithms. [Accessed: 25-Sep-2017].

[31] I. Montani and M. Honnibal, "Supervised learning is great — it's data collection that's broken," Explosion AI, 2017. [Online]. Available: https://explosion.ai/blog/supervised-learning-data-collection. [Accessed: 25-Sep-2017].

[32] J. Tin-yau Kwok, "Automated Text Categorization Using Support Vector Machine," Proceedings of the International Conference on Neural Information Processing, 1998

[33] T. Jo, "NTC (Neural Text Categorizer): Neural Network for Text Categorization," International Journal of Information Studies, vol. 2, no.2, 2010.

[34] T. Joachims, "Text categorization with Support Vector Machines: Learning with many relevant features," Machine Learning: ECML-98 Lecture Notes in Computer Science, 1998.

[35] P. Y. Pawar and S. H. Gawande, "A Comparative Study on Different Types of Approaches to Text Categorization," International Journal of Machine Learning and Computing, 2012.

[36] F. Sebastiani, "Machine learning in automated text categorization," ACM Computing Surveys, vol. 34, no. 1, 2002.

[37] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," The Journal of Machine Learning Research, vol. 3, 2003

[38] E. Wiener et al., "A Neural Network Approach to Topic Spotting," Proceedings of SDAIR-95, 4th Annual Symposium on Document Analysis and Information Retrieval, 1995

[39] S. Deerwester et al., "Indexing by Latent Semantic Analysis," Journal of the American Society for Information Science, 1990

[40] T. K. Landauer et al., "An introduction to latent semantic analysis," Discourse Processes, vol. 25, 1998

[41] "What is the time complexity of truncated SVD?," Mathoverflow.net, 2015. [Online]. Available: https://mathoverflow.net/questions/161252/what-is-the-time-complexity-of-truncated-svd. [Accessed: 26-Sep-2017].

[42] M. Lan et al., "Supervised and Traditional Term Weighting Methods for Automatic Text Categorization," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 31, no. 4, 2009.

[43] "What is Linear Regression? - Statistics Solutions," Statistics Solutions, 2013. [Online]. Available: http://www.statisticssolutions.com/what-is-linear-regression/. [Accessed: 26-Sep-2017].

[44] J. Brownlee, "Logistic Regression for Machine Learning - Machine Learning Mastery," Machine Learning Mastery, 2016. [Online]. Available: https://machinelearningmastery.com/logistic-regression-for-machine-learning/. [Accessed: 26-Sep-2017].

[45] "Why not approach classification through regression?," Stats.stackexchange.com, 2012. [Online]. Available: https://stats.stackexchange.com/questions/22381/why-not-approach-classification-through-regression. [Accessed: 26-Sep-2017].

[46] J. Brownlee, "Logistic Regression for Machine Learning - Machine Learning Mastery," Machine Learning Mastery, 2016. [Online]. Available: http://machinelearningmastery.com/logistic-regression-for-machine-learning/. [Accessed: 26-Sep-2017].

[47] S. A. Czepiel, "Maximum Likelihood Estimation of Logistic Regression Models: Theory and Implementation," 2002. [Online]. Available: https://czep.net/stat/mlelr.pdf. [Accessed: 26-Sep-2017].

[48] "Sample size for logistic regression?," Stats.stackexchange.com, 2012. [Online]. Available: https://stats.stackexchange.com/questions/26016/sample-size-for-logistic-regression. [Accessed: 26-Sep-2017].

[49] "Computational complexity of least square regression operation," Math.stackexchange.com, 2011. [Online]. Available: https://math.stackexchange.com/questions/84495/computational-complexity-of-least-square-regression-operation. [Accessed: 26-Sep-2017].

[50] T. Minka, "A Comparison of Numerical Optimizers for Logistic Regression," Unpublished draft, 2003.

[51] J. Johnson, “Logistic regression,” The Shape of Data, 2013. [Online]. Available: https://shapeofdata.wordpress.com/2013/05/17/logistic-regression/. [Accessed: 26-Sep-2017].

[52] S. F. Crone and S. Finlay, “Instance sampling in credit scoring: An empirical study of sample size and balancing,” International Journal of Forecasting, vol. 28, no. 1, 2012.

[53] R. Gatta et al., “The impact of different training sets on medical documents classification,” AIAM'14 Proceedings of the 3rd International Conference on Artificial Intelligence and Assistive Medicine, vol. 1213, 2014.

[54] C. D. Manning et al., “Introduction to Information Retrieval,” Cambridge University Press, 2008.

[55] B. Liu et al., “Building text classifiers using positive and unlabeled examples,” Data Mining, 2003.

[56] Y.-J. Lee et al., “Introduction to Support Vector Machines and Their Applications in Bankruptcy Prognosis,” Handbook of Computational Finance, 2011.

[57] M. Sewell, “Structural Risk Minimization,” Svms.org. [Online]. Available: http://www.svms.org/srm/. [Accessed: 26-Sep-2017].

[58] V. N. Vapnik, The nature of statistical learning theory. New York: Springer, 2010.

[59] C.-W. Hsu and C.-J. Lin, “A Simple Decomposition Method for Support Vector Machines,” Machine Learning, vol. 46, 2002

[60] R. Rojas, Neural Networks. Berlin: Springer, 1996.

[61] C.-W. Hsu and C.-J. Lin, “A Comparison of Methods for Multiclass Support Vector Machines,” IEEE Transactions on Neural Networks, vol. 13, no. 2, 2002.

[62] J. Weston and C. Watkins, “Multi-class support vector machines,” Technical Report CSD-TR-98-04, Department of Computer Science, Royal Holloway, University of London, 1998.

[63] K. Crammer and Y. Singer, "On The Learnability and Design of Output Codes for Multiclass Problems," Machine Learning, vol. 47, 2002

[64] P. Matykiewicz and J. Pestian, "Effect of small sample size on text categorization with support vector machines," BioNLP '12 Proceedings of the 2012 Workshop on Biomedical Natural Language Processing, 2012.

[65] N. Japkowicz, "Concept-learning in the absence of counter-examples: an autoassociation-based approach to classification," Rutgers University, 1999.

[66] L. Bottou and C.-J. Lin, "Support vector machine solvers," Large scale kernel machines 3.1, 2007.

[67] I. W. Tsang et al., "Core vector machines: Fast SVM training on very large data sets," Journal of Machine Learning Research, 2005.

[68] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," 1998.

[69] J. Wu, "Support Vector Machines," Nanjing University, China, 2017.

[70] "How does one interpret SVM feature weights?," Stats.stackexchange.com, 2012. [Online]. Available: https://stats.stackexchange.com/questions/39243/how-does-one-interpret-svm-feature-weights. [Accessed: 26-Sep-2017].

[71] S. Jiang et al, "An improved K-nearest-neighbor algorithm for text categorization," Expert Systems with Applications 39.1, 2012.

[72] K. Collins-Thompson, "K-Nearest Neighbors: Classification and Regression - University of Michigan | Coursera," Coursera. [Online]. Available: https://www.coursera.org/learn/python-machine-learning/lecture/I1cfu/k-nearest-neighbors-classification-and-regression. [Accessed: 26-Sep-2017].

[73] "k-NN computational complexity," Stats.stackexchange.com, 2016. [Online]. Available: https://stats.stackexchange.com/questions/219655/k-nn-computational-complexity. [Accessed: 26-Sep-2017].

[74] W.-Y. Loh, "Classification and regression trees," Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 1.1, 2011.

[75] "Random forests - classification description," Stat.berkeley.edu. [Online]. Available: https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm. [Accessed: 26-Sep-2017].

[76] "Is decision tree algorithm a linear or nonlinear algorithm?," Datascience.stackexchange.com, 2015. [Online]. Available: https://datascience.stackexchange.com/questions/6787/is-decision-tree-algorithm-a-linear-or-nonlinear-algorithm. [Accessed: 26-Sep-2017].

[77] "What is in general time complexity of random...(2015) - Quora," Quora.com, 2015. [Online]. Available: https://www.quora.com/What-is-in-general-time-complexity-of-random-forest-What-are-the-important-parameters-that-affect-this-complexity. [Accessed: 26-Sep-2017].

[78] J. Su and H. Zhang, "A fast decision tree learning algorithm," AAAI, vol. 6, 2006.

[79] "1.10. Decision Trees — scikit-learn 0.19.0 documentation," Scikit-learn.org. [Online]. Available: http://scikit-learn.org/stable/modules/tree.html. [Accessed: 26-Sep-2017].

[80] "Is there a Random Forest implementation that works well with very sparse data?," Stats.stackexchange.com, 2012. [Online]. Available: https://stats.stackexchange.com/questions/28828/is-there-a-random-forest-implementation-that-works-well-with-very-sparse-data. [Accessed: 26-Sep-2017].

[81] "How is Naive Bayes a Linear Classifier?," Stats.stackexchange.com, 2015. [Online]. Available: https://stats.stackexchange.com/questions/142215/how-is-naive-bayes-a-linear-classifier. [Accessed: 26-Sep-2017].

[82] "Why are Naive Bayes classifiers considered relatively immune to overfitting? - Quora," Quora.com, 2015. [Online]. Available: https://www.quora.com/Why-are-Naive-Bayes-classifiers-considered-relatively-immune-to-overfitting. [Accessed: 26-Sep-2017].

[83] K. Wang and S. J. Stolfo, "One-class training for masquerade detection," Workshop on Data Mining for Computer Security, Melbourne, Florida. 2003.

[84] "Computational complexity of learning (classification) algorithms - fitting the parameters," Cstheory.stackexchange.com, 2011. [Online]. Available:

https://cstheory.stackexchange.com/questions/4278/computational-complexity-of-learning-classification-algorithms-fitting-the-p. [Accessed: 26-Sep-2017].

[85] J. Fan et al., "High-dimensional classification," The Annals of Statistics, vol. 36, no. 6, 2008.

[86] "Bayesian Classifiers," Mines.humanoriented.com. [Online]. Available: http://mines.humanoriented.com/classes/2010/fall/csci568/portfolio_exports/lguo/bayesian.html. [Accessed: 26-Sep-2017].

[87] Y. Luo and F. Yang, "Deep Learning With Noise," [Online]. Available: http://www.cs.cmu.edu/~fanyang1/dl-noise.pdf. [Accessed: 26-Sep-2017].

[88] "Machine Learning Explained: Regularization," R-bloggers, 2017. [Online]. Available: https://www.r-bloggers.com/machine-learning-explained-regularization/. [Accessed: 26-Sep-2017].

[89] "Early Stopping - Deeplearning4j: Open-source, Distributed Deep Learning for the JVM," Deeplearning4j.org. [Online]. Available: https://deeplearning4j.org/earlystopping. [Accessed: 26-Sep-2017].

[90] A. Budhiraja, "Learning Less to Learn Better — Dropout in (Deep) Machine learning," Medium, 2016. [Online]. Available: https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5. [Accessed: 26-Sep-2017].

[91] T. Gupta, "Deep Learning: Feedforward Neural Network – Towards Data Science – Medium," Medium, 2017. [Online]. Available: https://medium.com/towards-data-science/deep-learning-feedforward-neural-network-26a6705dbdc7. [Accessed: 26-Sep-2017].

[92] "A Beginner's Guide to Recurrent Networks and LSTMs - Deeplearning4j: Open-source, Distributed Deep Learning for the JVM," Deeplearning4j.org. [Online]. Available: https://deeplearning4j.org/lstm.html. [Accessed: 26-Sep-2017].

[93] D. Merkl, "Content-based software classification by self-organization," Proceedings of ICNN95 - International Conference on Neural Networks, 1995.

[94] "What is the time complexity of backpropagtion algorithm in neural networks?," Researchgate.net, 2015. [Online]. Available: https://www.researchgate.net/post/What_is_the_time_complexity_of_backpropagtion_algorithm_in_neural_networks. [Accessed: 26-Sep-2017].

[95] D. G. Roussinov and H. Chen, "A scalable self-organizing map algorithm for textual classification: A neural network approach to thesaurus generation," Communication and Cognition in Artificial Intelligence Journal, 1998.

[96] "Computational Complexity of Prediction using SVM and NN?," Stats.stackexchange.com, 2014. [Online]. Available: https://stats.stackexchange.com/questions/94596/computational-complexity-of-prediction-using-svm-and-nn. [Accessed: 26-Sep-2017].

[97] "Why are deep neural networks so bad with sparse data? - Quora," Quora.com, 2017. [Online]. Available: https://www.quora.com/Why-are-deep-neural-networks-so-bad-with-sparse-data. [Accessed: 26-Sep-2017].

[98] "PEP 8 -- Style Guide for Python Code," Python.org, 2001. [Online]. Available: https://www.python.org/dev/peps/pep-0008/. [Accessed: 26-Sep-2017].

[99] S. Oualline, C Elements of Style, M&T Books, 1992

[100] J. Saxe et al., "Mining Web Technical Discussions to Identify Malware Capabilities," 2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops, 2013.

[101] A. Anderson et al., "Discovering value from community activity on focused question answering sites," Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD 12, 2012.

[102] E. Bouzioukou, "What is the Best Programming Language for Machine Learning?," Datafloq.com, 2017. [Online]. Available: https://datafloq.com/read/what-is-the-best-programming-language-for-machine-/3053. [Accessed: 26-Sep-2017].

[103] "Throttles - Stack Exchange API," Api.stackexchange.com. [Online]. Available: http://api.stackexchange.com/docs/throttle. [Accessed: 26-Sep-2017].

[104] "2.5.2.2.5. Compressed Sparse Row Format (CSR) — Scipy lecture notes," Scipy-lectures.org. [Online]. Available: http://www.scipy-lectures.org/advanced/scipy_sparse/csr_matrix.html. [Accessed: 26-Sep-2017].

[105] S. Deo, "Machine Learning : Training Set – Validation Set – Test Set," Tutorials for Data Science , Machine Learning, AI & Big Data, 2015. [Online]. Available: http://analyticspro.org/2015/09/21/machine-learning-training-set-validation-set-test-set/. [Accessed: 26-Sep-2017].

[106] S. Alsaleem, "Automated Arabic Text Categorization Using SVM and NB," Int. Arab J. e-Technol. 2.2, 2011.

[107] "Command prompt (Cmd. exe) command-line string limitation," Support.microsoft.com. [Online]. Available: https://support.microsoft.com/en-us/help/830473/command-prompt-cmd--exe-command-line-string-limitation. [Accessed: 26-Sep-2017].

","

Appendix A Deployment Guide

This thesis provides the implementation of the proposed prototype and a database file, that contains already retrieved StackOverflow posts, tags and extracted code pieces. The following deployment guide describes a step-by-step manual to run the prototype:

1. Install Python 3.6.1 with pip
2. Add the Python 3.6.1 installation folder and the Scripts sub-folder to the PATH environment variable (e.g. C:\Program Files\Python36 and C:\Program Files\Python36\Scripts)
3. Install tensorflow via pip:

   pip install --upgrade https://storage.googleapis.com/tensorflow/windows/cpu/tensorflow-1.2.0rc2-cp36-cp36m-win_amd64.whl

   (for 64-bit Windows, other versions available at https://storage.googleapis.com/tensorflow)
4. Download the whl-files for numpy+mkl, scipy and scikit-learn from http://www.lfd.uci.edu/~gohlke/pythonlibs
5. Install numpy+mkl and scipy via pip from the downloaded whl-files

   pip install --upgrade [path to numpy+mkl.whl] [path to scipy.whl]
6. Install scikit-learn via pip from the downloaded whl-file

   pip install --upgrade [path to scikit-learn.whl]
7. Install neupy via pip

   pip install --upgrade neupy
8. Install py-stackexchange via pip

   pip install --upgrade py-stackexchange

9. Install django via pip

   pip install --upgrade django

10. Install django-pyodbc-azure via pip

    pip install --upgrade django-pyodbc-azure

11. Configure the SQL Server connection in settings.py

    For a local SQL Server with Windows Authentication:

```
DATABASES = {
    'default': {
        'ENGINE': 'sql_server.pyodbc',
        'NAME': 'nitrogen',
        'USER': '',
        'PASSWORD': '',
        'HOST': '127.0.0.1',
    },
}
```

12. Copy the provided SQL database file nitrogen.mdf into the DATA folder of your SQL Server installation. For a local SQL Server 2014 installation this folder is per default located in *C:\Program Files\Microsoft SQL Server\MSSQL12.MSSQLSERVER\MSSQL*. Afterwards restart the SQL Server service.
    If you want to start the prototype on an empty database, create a new database with the name “nitrogen” in your SQL Server instance and run the console command python manage.py makemigrations from the project folder.
13. Start the application by running the console command *python manage.py runserver 8000* from the project folder.

TECHNISCHE UNIVERSITÄT CHEMNITZ

Studentenservice – Zentrales Prüfungsamt

Selbstständigkeitserklärung

Name: Siegel Vorname: Frank geb. am: 08.06.1993 Matr.-Nr.: 279172	Bitte beachten: 1. Bitte binden Sie dieses Blatt am Ende Ihrer Arbeit ein.

Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: 27.09.2017 Unterschrift: .....

---

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.