Sentiment Classification of Chess Annotations

Master-Thesis von Florian Beck
1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Markus Zopf

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Sentiment Classification of Chess Annotations

Vorgelegte Master-Thesis von Florian Beck

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Markus Zopf

Tag der Einreichung:

Erklärung zur Master-Thesis
gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Florian Beck, die vorliegende Master-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Darmstadt, den 7. Mai 2019

_____

(Florian Beck)

Abstract

This paper examines how the corresponding annotation symbol can be found for comments in chess games. With the help of such an assignment, additional information could be generated for chess games that have so far only been provided with comments. As a result, these games can be better evaluated and used for learning procedures based on them.

For analysis, supervised classification problems are set up for both comments on moves and positions. Appropriate comment data are extracted from chess databases, preprocessed by natural language processing and finally transformed into a suitable model for classification. For this purpose, four different models are examined: one based on the absolute frequencies of the words (count), one based on the relative frequencies of the words (TF-IDF), a word embedding trained on the chess comments and a word embedding pre-trained on Google News. For three different data sets with mixed-length, short and long comments, several multiclass classifiers are tested.

While the classifiers did not show a clear ranking, the count-based model achieved the best accuracy values, followed by the TF-IDF-based model. Of the two word embeddings, the one pre-trained on Google News performed better than the one trained on the chess commentaries. In addition, for the shorter comments better accuracies could be achieved than for mixed-long or long comments.

However, all the achieved accuracies are too low to use the models for transforming comments into annotation symbols. An improvement could be achieved by removing certain stopwords, which are likely to have a negative effect especially on long comments. Likewise, a further development of the self-trained word embedding could lead to an improvement of the accuracies, since despite the lowest accuracy values it calculates a promising model with meaningful semantic similarities in the chess vocabulary.

Zusammenfassung

In dieser Arbeit wird untersucht, inwiefern für Kommentare aus Schachpartien das zugehörige Annotationssymbol gefunden werden kann. Mithilfe einer solchen Zuweisung könnten zu bisher nur mit Kommentaren versehenen Schachspielen zusätzliche Informationen generiert werden, sodass die Spiele besser auswertbar sind und beispielsweise für darauf aufbauende Lernverfahren genutzt werden können.

Zur Analyse werden überwachte Klassifikationsprobleme sowohl für Kommentare zu Zügen als auch zu Stellungen formuliert. Es werden geeignete Kommentardaten aus Schachdatenbanken extrahiert, mittels Natural Language Processing vorverarbeitet und schließlich in einem für die Klassifikation geeigneten Modell aufbereitet. Hierfür werden vier unterschiedliche Modelle untersucht: eines basierend auf den absoluten Häufigkeiten der Wörter (Count), eines auf den relativen Häufigkeiten der Wörter (TF-IDF), ein auf den Schachkommentaren trainiertes Word Embedding und ein auf Google News vortrainiertes Word Embedding. Für drei verschiedene Teile des Datensatzes mit gemischt-langen, kurzen und langen Kommentaren werden mehrere Multiklassen-Klassifizierer getestet.

Während bei den Klassifizierern keine eindeutige Rangfolge erkennbar war, erzielte das Count-basierte Modell die besten Genauigkeitswerte, gefolgt vom TF-IDF-basierten Modell. Bei den beiden Word Embeddings schnitt das auf Google News vortrainierte besser ab als das auf den Schachkommentaren trainierte. Außerdem konnten für die kürzeren Kommentare bessere Genauigkeiten erreicht werden als für gemischt-lange oder lange Kommentare.

Die erzielten Genauigkeiten sind jedoch insgesamt zu niedrig, um die Modelle für eine Umwandlung von Kommentaren zu Annotationssymbolen nutzen zu können. Eine Verbesserung könnte durch eine Entfernung bestimmter Stoppwörter erreicht werden, die vermutlich besonders bei langen Kommentaren einen negativen Einfluss haben. Ebenso könnte eine Weiterentwicklung des selbst trainierten Word Embeddings zu einer Verbesserung der Genauigkeiten führen, da dieses trotz der niedrigsten Genauigkeitswerte ein vielversprechendes Modell mit sinnvollen semantischen Ähnlichkeiten im Schachvokabular berechnet.

Contents

List of Figures

List of Tables

1 Introduction

Chess games are noted in a uniform form like the PGN format in order to be able to perform analyses and evaluations at a later date. These data usually contain information about the players and the event as well as the exact move sequence of the chess game. In addition, especially in large chess databases, decisive moves and positions are provided with comments after the game has been analyzed by grandmasters. For a better arrangement of these comments there are standardized symbols and NAGs, which for example directly indicate whether a move was good or bad. These have the advantage that they are easier to evaluate due to their clear categorization and are also generally understandable, which is not the case with comments in natural language.

In many cases the comments were therefore replaced by the annotation symbols if a comment did not seem necessary. For more detailed explanations, which can not only be expressed with an annotation symbol, both annotation symbols and comments are used. Finally, there are comments without a corresponding annotation symbol. This may be intentional if the commentary only names the commentator, summarizes a game or refers to similar chess games. However, some comments refer specifically to moves and positions and could therefore be provided with an annotation symbol.

For the evaluation of annotation symbols in chess there are already approaches as described in [WF14] and [WF15], where the annotations are used to learn preferences. In order to be able to use pure comment data for such evaluations, annotation symbols must have been assigned to the comments beforehand.

1.1 Problem Definition

In this thesis, a suitable approach for the correct assignment of annotation symbols to given chess annotations is to be found. It is a specialization of a supervised sentiment classification problem since the used data for the learning process contains only comments that are already labeled with the correct annotation symbol. This problem is split up into five separate questions:

- the correct categorization into move- and position-annotating comments for given chess comments

- the correct assignment of two annotation symbols ($!, ?$) to given chess move comments

- the correct assignment of six annotation symbols ($!!, !, !?, ?!, ?, ??$) to given chess move comments

- the correct assignment of three annotation symbols ($+-, =, -+$) to given chess position comments

- the correct assignment of seven symbols ($+-, \pm, +=, =, =+, \mp, -+$) to given chess position comments

1.2 Goal of the Thesis

The main question of this work is whether and how well the appropriate annotation symbol for a given chess comment can be determined. Thus, the goal to find the configuration with the best accuracy and other evaluation metrics with regard to the classification problems just presented. For this purpose, four data models will be developed, ten classifiers selected and the configurations tested on three different data sets. In addition, further suitable processing methods are to be investigated and, if appropriate, added in order to optimize the results. The final analysis is also intended to provide insights into the remaining possibilities for improvement.

1.3 Structure of the Thesis

The thesis is split into five more chapters:

The second chapter initially provides background knowledge in the topics dealt with in this thesis. It starts with the presentation of sentiment analysis as part of text mining. Afterwards the concept of word embeddings is discussed, which can serve as a data model in such text mining problems. This is followed by a description of multiclass classification problems and several suitable approaches to solve such problems. A separate look is taken at the subgroup of ordinal classification problems. Finally, cost-sensitive methods are presented that take into account different weightings of misclassifications.

The third chapter describes the general concept with which a text mining process can be created. It first deals with the requirements of problem and goal definition and the criteria for a suitable data selection. For the preparation of the data and their transformation into a model suitable for analysis, methods of natural language processing are presented.

Finally, possible analysis methods and evaluation methods are presented from which suitable techniques for the problem can be selected.

In the fourth chapter, the procedure presented in the third chapter is applied to the text mining process in chess annotations. First the format PGN and the annotation symbols NAG are explained and the five problems are specified. In the following the used tools like NLTK, Weka and further libraries are listed. Finally, the classifiers and evaluation methods used are mentioned.

The fifth chapter contains all results and their evaluations. After the analysis of different tokenizer configurations, statistics about the comments are generated, which can be used to gain basic knowledge about the analyzed data set. In addition, the attributes and models used for the data set are evaluated. The majority of the results are finally taken up by the comparisons of the achieved accuracies for all configurations and problems. In the end, the best results achieved are checked for optimality in a cost-sensitive evaluation.

The last chapter summarizes the most important conclusions from the results and gives an outlook which further approaches could be used to improve the results.

2 Fundamentals

This chapter deals with some basics of the topics into which the problem discussed in the paper can be categorized. The first two sections Sentiment Analysis and Word Embeddings focus on data mining approaches for data in natural language. The remaining three sections Multiclass Classification, Ordinal Classification and Cost-sensitive Learning address the analysis and evaluation of data in (non-binary) classification problems.

2.1 Sentiment Analysis

Sentiment analysis, also known as sentiment classification or opinion mining, is the field of study that analyzes people's opinions, sentiments, evaluations, attitudes, and emotions from written language. It is one of the most active research areas in natural language processing and is also widely studied in data mining, web mining, and text mining [Liu12]. Due to the rapidly increasing amount of data on the Internet and the Web, the data sources for sentiment analysis have also grown considerably. In particular, since the spread of Web 2.0, more and more people have actively participated in the exchange of knowledge or opinions via the Internet. This results in large amounts of data of natural language, which are, however, often characterized by subjectivity.

Nowadays, in addition to the large area of data in social media, evaluations of products and services play an important role in sentiment analysis. Between 73% and 87% of the consumer report that online product evaluations influence their purchase decision and with very good evaluations even a 20% - 99% higher price is accepted, whereby the exact added value depends on the product [PL08, section 1.1]. Since these data are available in very large quantities, an automatic evaluation is desirable.

Sentiment classification problems can occur at different levels of difficulty. Plutchik distinguishes in his model between the eight emotions joy, trust, fear, surprise, sadness, disgust, anger and anticipation. For each of these emotions there is a mild and an intense form, e.g. annoyance is the mild and rage the intense form of anger. The arrangement of similar emotions next to each other and opposing emotions opposite to each other results in Plutchik's wheel of emotions, which can be seen in figure 1 [Plu80].
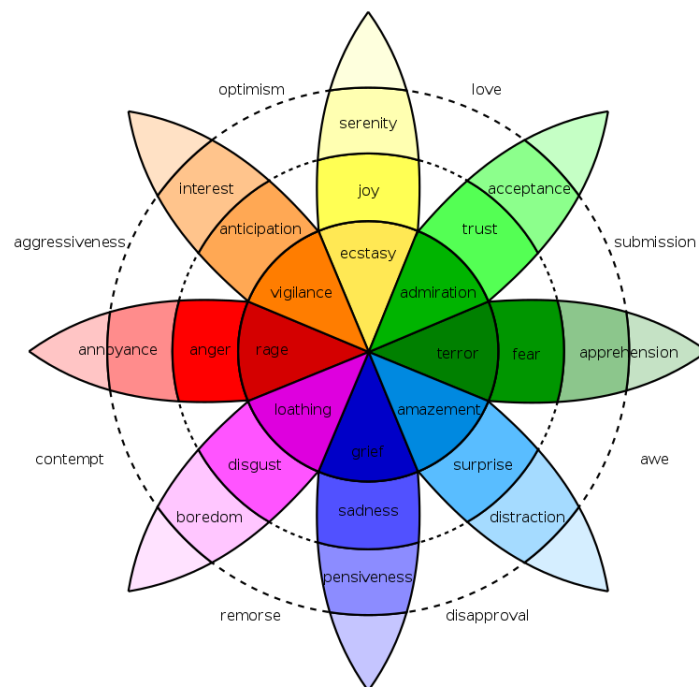

Figure 1: Plutchik's wheel of emotions[1]

In most cases, however, a single scale is sufficient to classify the author's opinion. The only distinction is made between a positive and a negative attitude. With the help of sentiment words such as "good", "great" or "terrible" a

---

[1] `https://en.wikipedia.org/wiki/Robert_Plutchik#/media/File:Plutchik-wheel.svg`, (07.05.2019, 06:24)

reliable statement about the expressed opinion can be made quickly. These sentiment words are collected in various encyclopedias and are constantly being further developed. Nevertheless, some complications can occur here as well. Negated sentences can contain sentiment words, but their meaning and thus also their sentiment is reversed by the negation. This is even more complicated with sarcastic statements, in which the sentiment is reversed, although there is no negation in the text. Finally, there are also sentences that do not contain a single sentiment word and nevertheless express a clear opinion. An example of this is the sentence "This washer uses a lot of water", which implicitly expresses a negative attitude [Liu12, section 1.2.2].

2.2 Word Embeddings

The application of text mining or sentiment analysis strategies requires a suitable application of the documents to be analyzed, usually by converting them into machine-understandable numeric values. A common representation form of document in natural language processing is the vector space model. Each word which appears in at least one document, i.e. which is part of the vocabulary, forms a dimension in the vector space model. Finally, each document is represented as a vector in this vector space by assigning a number to each word in the vocabulary, e.g. its absolute or relative term frequency. This allows finding documents that contain certain words and comparing documents on the basis of their contained words using a distance function in the vector space.

However, the vector space model has some limitations and disadvantages. These include the usually high number of dimensions caused by a big vocabulary, which results in high memory and computing costs. Moreover, it leads to sparse data, because most words of the vocabulary do not occur in a document and its value is therefore 0. Besides, synonyms or similar words are treated in different dimensions and thus documents using those different words cannot be detected as similar.

Word embeddings are a collection of techniques to map documents or single words to a vector space whereby the above-mentioned problems are solved. A predefined vector space of a fixed size N is used and the approach of a distributed representation provides better and multiple degrees of similarities between two words. Mikolov et al. developed a new word embedding model that not only reduces the count of dimensions heavily, but also offers large improvements in accuracy of semantic and syntactic questions [Mik+13]. Hence algebraic functions can be used to answer questions like the currency of a country: vector("Euro") − vector("Germany") + vector("South Africa") calculates a vector close to the vector("Rand").

The underlying concept is based on a neural network with an input layer, an output layer and a hidden projection layer in between. Mikolov et al. uses two different models shown in figure 2 that are both based on the context of the word. The continuous bag-of-words (CBOW) model gets the context of a word as the input, i.e. the k preceding and k succeeding words, in the figure with the so-called window size $k = 2$. With the skip-gram model the input and output are reversed; from a given word, the context is inferred. In both cases, the output is a learned vocabulary, with a vector of the previously defined length N assigned to each word [Mik+13]. Based on this vector, now further learning methods can be applied.
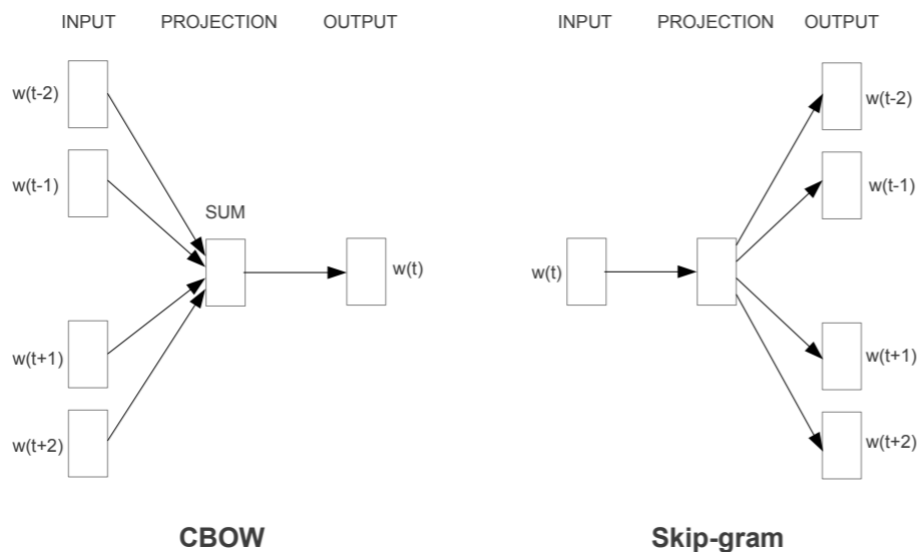
Figure 2: CBOW and Skip-gram model [Mik+13]

Even if the output itself is compared to the traditional vector space model meaningless to a human, the combination of multiple vectors offers considerable information. In addition to the already mentioned use of algebraic functions, the vocabulary can be searched for synonyms or other related words by looking for a similar vector like the given word has. To calculate the similarity of two vectors, the cosine similarity can be used. Its value is between -1 indicating completely different words and 1 indicating identical words, where 0 stands for no correlation between the words. The cosine similarity of two vectors **a** and **b** is defined as follows:

$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|_2 \|\mathbf{b}\|_2} = \frac{\sum_{i=1}^{n} a_i \cdot b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2} \cdot \sqrt{\sum_{i=1}^{n} (b_i)^2}}$$

Kusner et al. compare several further measurements of similarity of words or documents and their performance in different corpora. They also present the Word Mover's Distance (WMD), a novel distance function between text documents leading to unprecedented low k-nearest neighbor document classification error rates [Kus+15].

The models presented by Mikolov et al. as well as the cosine similarity measurements are implemented and collected in the framework word2vec and can be used for training and evaluation in the most common programming languages. Another common model is GloVe[2], coined from Global Vectors, which in addition to the local context-based learning used in Word2Vec integrates global statistics of matrix factorization. The GloVe model produces a vector space with meaningful substructure, as evidenced by its performance of 75% on a recent word analogy task [PSM14]. Both frameworks offer pre-learned word embeddings on big corpora, among them also topic-specific data sets. However, it is also possible to learn a new embedding on an own data set.

2.3 Multiclass Classification

Multiclass classification is a problem in the field of supervised learning. The task is the prediction of one class out of a finite set of three or more possible classes for each instance. As can be seen in figure 3, it can be distinguished from other problems in supervised learning. First, as the name implies, it is a classification problem because the range of values is finite rather than continuous, as is the case with a regression problem. Furthermore, each instance should be associated with exactly one class, which distinguishes it from the similar sounding multi-label classification, since each instance can be associated with any number of classes, also known as labels. Finally, the model to be learned has to choose from a set of k classes with $k > 2$, i.e. from a set $\{1, 2, \ldots, k\}$. This makes the problem slightly more difficult than binary classification with $k = 2$. Ordinal classification is a specialization of multiclass classification and will be discussed in detail in section 2.4.
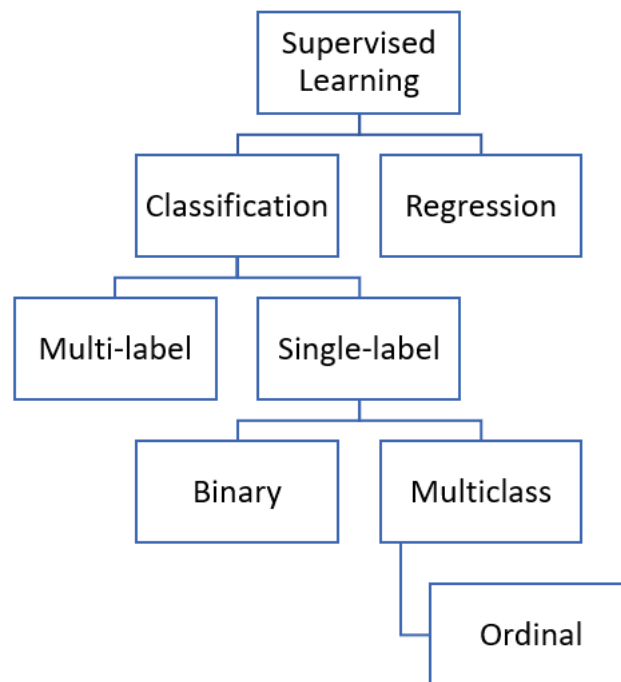
Figure 3: Problem hierarchy of supervised learning

Most algorithms applied to classification problems are explained by their application to binary classification problems, which cover a large part of the real-world use cases. However, these approaches cannot be adopted to multiclass problems without further ado. Aly describes with the decomposition into binary classification, hierarchical classification and the extension of algorithms the three groups of approaches to cope with the difficulties of multiclass classification [Aly05]:

- Decomposition into Binary Classification

  A frequently chosen approach is the transformation of the multiclass classification problem into several binary classification subproblems, for which there are three different possibilities. In any case the results of subproblems have to be joined to be able to make a final prediction which is similar to ensemble learning.

  The first option called one-against-all splits the multiclass classification problem into k binary classification subproblems. The classifier f_i treats the instances of class i as positive and instances of the $k-1$ other classes as negative. An example of one-against-all using an SVM-classifier is shown in figure 4a. For an unseen instance x the class k with the highest confidence score will be predicted:

$$y(x) = \underset{i \in \{1 \dots k\}}{\operatorname{argmax}} f_i(x)$$

  However, there is no guarantee that the real-valued quantities $f_i(x)$ for different classifiers will have appropriate scales. Furthermore, especially for a high k the ratio between positive and negative instances is low, which complicates the creation of a model. This problem can be addressed by giving greater weight to the positive instances [Bis06, section 7.1.3].

(a) One-against-all   **(b)** One-against-one
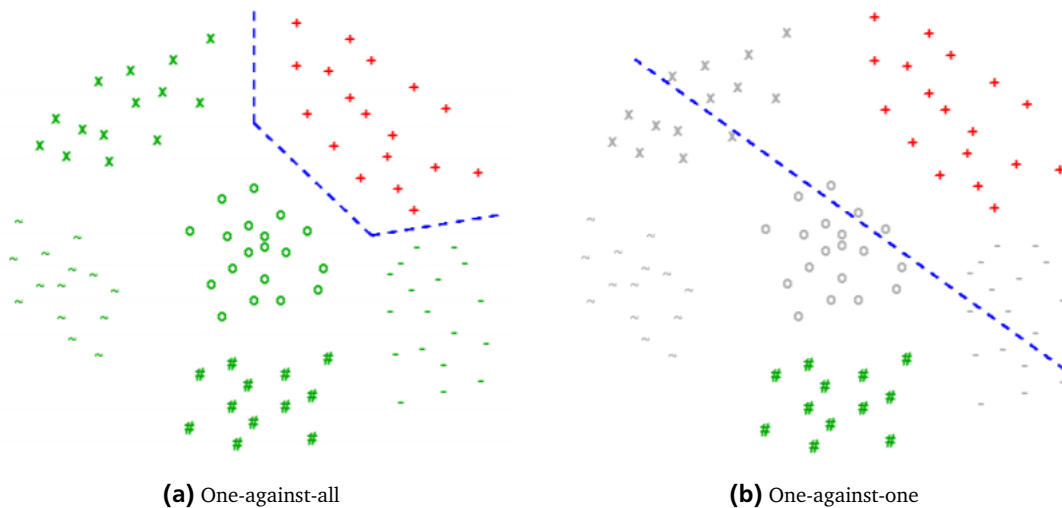
Figure 4: One-against-all vs. one-against-one approach[3]

  The second option called one-against-one splits the multiclass classification problem into even more binary classification subproblems, in fact $k(k-1)/2$. This number results from the fact that now in each subproblem two classes are pairwise distinguished from each other. Since in the binary classifier between the classes i and j all instances of the classes except i and j can be discarded (see figure 4b), there are smaller subproblems than in one-against-all, which usually also create simpler models. However, the number of subproblems is $\frac{k}{2}$-times as high, which usually results in significantly more training time [Bis06, section 7.1.3].

  The final decision is made by voting. All pairwise comparisons are summed up and the winner with the most partial victories is the output class. However, using the binary outputs of the subproblems, i.e. only the preferred class, often leads to several winners. There are different ways to handle these ambiguities. The first idea is weighting the subproblems differently, for example by modifying the output from 1 to the accuracy and from 0 to 1−accuracy. Another idea called pairwise coupling, which includes probabilities for every subproblem in the calculation, can be used to extend the one-against-one approach. Wu et al. present a few pairwise coupling techniques that are more stable than using the simple voting approach [WLW04].

---

3   J. Fürnkranz, Machine Learning and Data Mining | Learning Rule Sets, V3.0, p.45&47

The third option is the usage of error-correcting output codes. Each class is assigned to a unique binary string of length n; we will refer to these strings as "codewords". During training for an example from class i, the desired outputs of these n binary functions are specified by the codeword for class i. With artificial neural networks, these n functions can be implemented by the n output units of a single network [DB94, chapter 1]. Though, it is also possible to train all n binary classifiers individually with an arbitrary algorithm. A test instance is assigned to the class that is closest to the code determined by the n functions.

Now the question arises, how the codewords respectively the binary functions are chosen most effectively. Out of all 2^k binary combinations for a k-class problem, after removing 2^{k-1} complements (e.g. all starting with 1) and the uniform classifier there are $2^{k-1} - 1$ possible classifiers, from which to be selected. A good error-correcting output code for a k-class problem should satisfy two properties [DB94, section 2.3]:

- **Row Separation**

  If in a code converting an arbitrary codeword into another arbitrary codeword needs d changes of bits, the code has a so-called Hamming distance of d. The higher the Hamming distance, the better the row separation. A code with Hamming distance d can detect up to $d-1$ and correct up to $\left\lfloor \frac{d-1}{2} \right\rfloor$ bits.

- **Column Separation**

  Each bit-position function f_i should be uncorrelated with the functions to be learned for the other bit positions f_j; $j \neq i$. If two columns i and j (or the complement of j) are similar or identical, then when a deterministic learning algorithm such as the decision tree classifier C4.5 is applied to learn f_i and f_j, it will make similar (correlated) mistakes.

The classifier sets in the one-against-all and one-against-one approaches are feasible solutions. The one-against-all approach guarantees Hamming distances of 2 for both row and column separation, the one-against-one approach even $2 \times (k-2)$ for the row and 4 (if $k > 7$) for the column separation. The optimal row distance is achieved by the union of all $2^{k-1} - 1$ binary combinations and called exhaustive code. Until $k = 5$ the exhaustive code is just the union of all classifiers of the one-against-all and one-against-one approaches (or their inversions, see table 1), but for $k > 6$ the number of additional classifiers increases quickly, wherefore instead of an exhaustive code also a suitable subset of those classifiers can be sufficient and improve the column separation.

class	f_1	f_2	f_3	f_4	f_5
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0
5	0	0	0	0	1

(a) One-against-all

class	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
1	1	1	1	1	0	0	0	0	0	0
2	1	0	0	0	1	1	1	0	0	0
3	0	1	0	0	1	0	0	1	1	0
4	0	0	1	0	0	1	0	1	0	1
5	0	0	0	1	0	0	1	0	1	1

(b) One-against-one

class	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
3	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
4	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
5	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

(c) Exhaustive error-correcting output code

Table 1: Different output codes for a multiclass classification problem with five classes

The exhaustive error-correcting code for $k = 5$ is shown in table 1. The hamming-distance of the code is $d = 8$. Therefore, the code can detect up to $d - 1 = 7$ and correct up to $\left\lfloor \frac{d-1}{2} \right\rfloor = 3$ bits. Error-correcting output codes are a robust alternative to one-against-all and one-against-one approaches which outperforms them in most cases but also entail the disadvantages of a more complex training and model [DB94, chapter 4].

- Hierarchical Classification

  Hierarchical classification, also known as nested dichotomies, is strictly speaking also a type of decomposition of a multiclass classification problem into binary classification problems. However, in this approach the output of the classifiers is not always used for the prediction but as the input for the next classifier. In this way, a hierarchy of

classifiers and their subproblems is created. The first classifier is trained on the whole input space; the total set of all classes is suitably divided into two subsets between which the classifier distinguishes. This procedure is then continued recursively with the two subsets until all subsets only contain instances of a single class. This procedure creates a tree structure whose leaves are the set of instances of a class. As a result, this tree contains k leaves and $k-1$ inner nodes representing the binary classifiers [DFK05]. Two possibly outcomes of a multiclass classification problem with 5 classes are shown in figure 5.
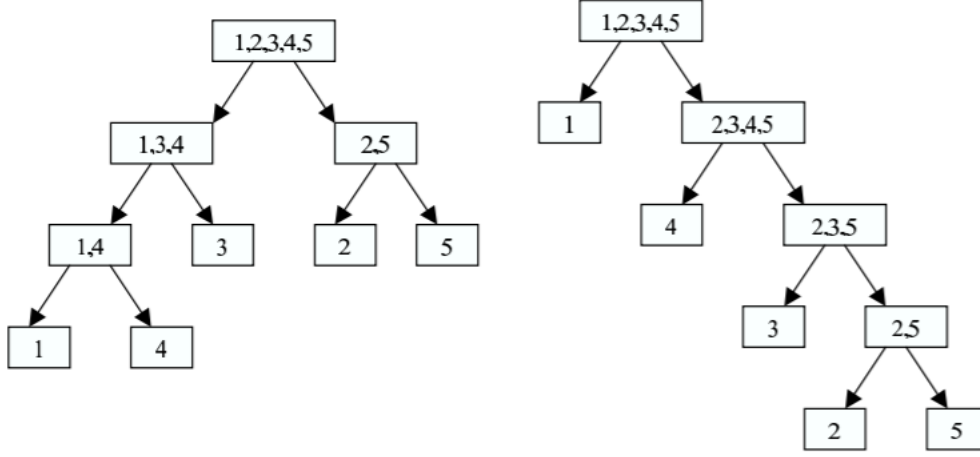
Figure 5: Two nested dichotomies for a multiclass problem with five classes [DFK05][4]

The probability of a class can be calculated by multiplying all probability estimates of the classifier along the path from the root to the class. This value differs for disparate nested dichotomies, wherefore usually various trees are computed and their probabilities are averaged. Though, for high k there are too many different possibilities to build a nested dichotomy. Balanced nested dichotomies prove to be particularly suitable, since they require the shortest running time without affecting the accuracy negatively [DFK05]. Dong et al. developed methods to balance the nested dichotomies either by class or by data amount, i.e. the subsets are created in a way that either their difference of classes or their difference of data is minimal. The left dichotomy shown in figure 5 is class-balanced, because the difference of classes between all subsets is either zero or one. It is as well data-balanced, if all classes have nearly the same number of instances. The right dichotomy is not class-balanced, because the first two splits have class ratios of $4:1$ and $3:1$. However, this dichotomy is data-balanced if class 1 contains nearly as much instances as all other classes, class 2 nearly as much as 3, 4 and 5 together and class 3 as much as 4 and 5 together.

- Extension of Algorithms

The third idea is the inversion of the first two approaches. Instead of modifying the problem and using the same classifier as for the binary classification, we now modify the classifier such that we can leave the classifier unchanged. Typical algorithms in machine learning are decision trees, pattern- or rule-based classifiers, probabilistic classifiers, SVM classifiers, neural network classifiers and proximity-based classifiers. A short description for every algorithm can be found in section 3.5. Decision trees, pattern- or rule-based classifiers, probabilistic classifiers as Naïve Bayes and proximity-based classifiers as k-nearest-neighbor can be easily adapted to multiclass problems. For the other two algorithms some preliminary considerations are necessary.

SVM classifiers use hyperplanes of dimension $D-1$ described by linear functions to separate the whole input space of dimension D into two subspaces which ideally only contain a single class. Obviously, a single hyperplane is not sufficient for separating more than two classes. Approaches like one-against-all and one-against-one lead to ambiguous subspaces where either more than one class or no class is assigned to. We can avoid these difficulties by considering a single k-class discriminant comprising k linear functions (with parameters $\mathbf{w}_i$ and w_{i0}) of the form

$$y_i(\mathbf{x}) = \mathbf{w}_i^{\mathrm{T}}\mathbf{x} + w_{i0}$$

and then assigning a point $\mathbf{x}$ to class i if $y_i(\mathbf{x}) > y_j(\mathbf{x})$ for all $j \neq i$. The decision boundary between class i and class j is therefore given by $y_i(\mathbf{x}) = y_j(\mathbf{x})$ and hence corresponds to a $(D-1)$-dimensional hyperplane defined by

$$\left(\mathbf{w}_i - \mathbf{w}_j\right)^{\mathrm{T}}\mathbf{x} + \left(w_{i0} - w_{j0}\right) = 0.$$

The decision regions of such a discriminant are always singly connected and convex [Bis06, section 4.1.2].

A neural network classifier has in general n neurons in the output layer, each of them with a binary output. For binary classification problems, a single neuron in the output layer, i.e. $n = 1$, is sufficient since it only needs to distinguish between two classes, represented by the two outputs 0 and 1. For multiclass classification problems, $n = 1$ is not sufficient. The straightforward solution would be to define $n = k$ neurons in the output layer, one for each of the k classes. But already $\lfloor \log_2 k \rfloor$ output neurons are enough if the classes are encoded as the first k numbers in binary representation. By using error-correcting output codes containing more than $\lfloor \log_2 k \rfloor$ output neurons, the binary code can also be made more resistant to errors [Aly05, section 2.1].

As mentioned in the beginning of this section, ordinal classification problems are a subgroup of multiclass classification problems. The following section describes the special properties of this type of problems and other approaches that are even better adapted to these properties than those presented here.

2.4 Ordinal Classification

Ordinal classification problems are a specialization of multiclass classification problems. They refer to an important category of real-world problems, in which the classes exhibit a natural order. Thus, the class attribute is not nominal, but ordinal. However, the possibility of calculating the difference (essential for an interval class attribute) or even the quotient (essential for a ratio class attribute) is usually not given [FH01, chapter 1]. Ordinal classification may be viewed as a bridging problem between the two standard machine-learning tasks of classification and regression, because the set of classes has the property of a finite range given in classification problems as well as the property of a natural order given in regression problems [KP04, chapter 1].

As a specialization of a multiclass classification problem, ordinal classification problems can be approached with the same strategies as described above in section 2.3. However, the additional information of the linear order of the classes is ignored. Two strategies tailored to ordinal classification problems taking this information into account are ordinal regression and an adapted transformation to binary classification problems.

- Ordinal Regression

  In regression tasks, the output is a numeric value in a continuous range. The idea of ordinal regression is to treat the ordinal output as a numeric value. In this way the same approaches as for regression tasks can be used only with the extension that the predicted values still have to be mapped to the fixed classes. Nevertheless, most regression methods require some metric assumptions like building differences or ratios within the ordinal scale [RLT14, chapter 1]. Hence the result of the model depends on the choice of numeric values and their distances from each other. This leads to problems in many real-world applications, for example in the quantification of ordinal scales in opinion surveys.

  Ordinal logistic models avoid this issue by only considering the ranking order of the classes [Har15, chapter 13]. The most commonly used ordinal logistic model is the proportional odds model. It uses the logarithms of odds that the predicted class y is one of the first i classes together with a linear function, whereby the probability after forming is given by

  $$P(y \leq i | \mathbf{x}) = \frac{1}{1 + \exp[-(\mathbf{w}^\mathrm{T}\mathbf{x} + w_{i0})]}$$

  where $\mathbf{x}$ is the test instance and $\mathbf{w}$ and w_{i0} the parameters of the linear function. There are many possible extensions to this ordinal model, among others replacing the cumulative probabilities with conditional probabilities or using a hazards model [Har15, chapter 13].

- Transformation to Binary Classification Problems

  A simpler approach using a transformation to binary classification problems is presented by Frank and Hall. In contrast to the binary classification approaches in multiclass classification it takes advantage of the order of the classes. The ordinal class attribute with ordered values $v_1 < v_2 < \ldots < v_k$ is converted into $k-1$ binary attributes. The i-th binary attribute represents the test if the class attribute y exceeds the value of the i-th class: $y > v_i$. This construction of binary attributes requires exactly two classifiers and their probabilities to determine whether a test instance $\mathbf{x}$ belongs to a class i:

  $$P(y = v_i \,|\, \mathbf{x}) = P(y > v_{i-1} \,|\, \mathbf{x}) - P(y > v_i \,|\, \mathbf{x})$$

For the lowest class, the first probability is 1 and for the highest, the second is 0, which further simplifies the calculation. Finally, the class with the highest probability is predicted. Frank and Hall showed that for ordinal classification problems this approach outperforms the "naïve" one-against-all algorithm, which treats the class values as an unordered set [FH01].

2.5 Cost-sensitive Learning

In many classification problems the goal is the optimization of some performance measurement, most often the accuracy. However, the best solution in respect to the accuracy does not have to be the one with minimized costs, which is also often a goal to be achieved. Costs cannot only arise from misclassifications, but also from additional tests or time for the creation of a better model. Turney divides the reasons for costs that a machine learning problem can entail into nine groups [Tur02]:

- misclassification errors
- tests
- teacher
- intervention
- unwanted achievements
- computation
- cases
- human-computer interaction
- instability

In each of the nine groups, a distinction can be made between constant and conditional costs. In the case of test costs, we speak of constant costs if a certain test has a fixed cost value for its execution. Note that different test may have different costs if they still remain constant for all instances in all circumstances. On the contrary, conditional test costs depend on a criterion. For a medical test this criterion might be the test result, the age of a patient or the result of prior tests. In the remaining section, of all cost types only the misclassification costs are discussed and therefore abbreviated to costs.

The standard approach to evaluate the performance of a classifier is the calculation of the accuracy, i.e. the number of correct classified instances in relation to the total number of instances. The error costs result directly from the number of misclassified instances, which only has to be multiplied by a constant cost factor. Thus, Turney calls this value constant misclassification error cost. However, there are various problems in machine learning where such a simplification would lead to a miscalculation of the total costs because the conditional error costs are ignored.

In many cases, the equal treatment of all error types is the cause of the incorrect cost estimate. For this we first consider the binary classification with only two classes and therefore only two possible error types. A false positive (FP) occurs when the outcome is incorrectly predicted as yes or positive when it is actually no or negative. The reverse case is called false negative (FN). False positives and false negatives have rarely equal costs in real-world applications. In spam-classification, a no-spam e-mail classified as spam is worse than a spam e-mail which have not been detected correctly. An accepted customer which is not capable of paying back a loan has bigger costs than a customer wrongly classified as insolvent. A sick patient who is classified as healthy is more problematic than a healthy patient who is classified as sick. In order to map this, a cost value is determined for each of the two errors FP and FN or the cost ratio is determined.

If we transfer the cost calculation to the multiclass classification, there are more error types and therefore more cost factors to be defined: Each instance of class i, misclassified as class j will be multiplied with the correspondent cost factor c_{ij}. For a multiclass classification problem with n classes we get a $n \times n$ cost matrix similar to the confusion matrix, which has $n(n-1)$ possibly different cost factors and whose main diagonal contains zeros.

A special case arises for classification problems with ordinal classes. The value of c_{ij} should reflect the extent of the difference between i and j. Thus, it is common to assume that $c_{ij} = 0$ when $i = j$. In addition, the cost c_{ij} is assumed to be larger when i is further away from j. Two common functions satisfy the requirements and have been widely used in practice [RLT14, section 3.1]:

- Absolute cost vectors $\qquad c_{ij} = |i - j|$

- Squared cost vectors $\qquad c_{ij} = (i - j)^2$

Note that the squared cost charges more than the absolute cost when i is further away from j since $|i-j| \geq 1 \forall i, j \in \mathbb{Z}$. The cost matrix of a multiclass classification problem with n classes using absolute cost vectors looks like this [KP04, chapter 3]:

$$
\begin{bmatrix}
0 & 1 & 2 & \ldots & n-1 \\
1 & 0 & 1 & \ldots & n-2 \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
n-2 & \ldots & 1 & 0 & 1 \\
n-1 & \ldots & 2 & 1 & 0
\end{bmatrix}
$$

By squaring each element of the matrix, one obtains the matrix of squared cost vectors. In both cases it is a symmetrical matrix, i.e. $c_{ij} = c_{ji} \forall i, j \in \{1, \ldots, m\}$. In some cases, it may be useful to weight the over- and underestimation of classes differently. If the overestimation of a class is to be penalized more severely than the underestimation, all elements to the right of the main diagonal are multiplied by a constant value $\lambda > 1, \lambda \in \mathbb{R}$. In a comparable way all values of the matrix can be adjusted as required to the problem which has to be solved.

Given such a cost matrix, the cost of a particular learned model on a given test set can be calculated just by summing the relevant elements of the cost matrix for the model's prediction for each test instance. Here, the costs are ignored when making predictions, but taken into account when evaluating them [WF05, section 5.7]. In this way the classifier obviously does not have to deliver the cost-optimal result. However, there are different possibilities to include costs in advance [Qin+10, section 2.3]:

- Change of the class distribution of the training data

  This approach incorporates the misclassification cost into the data pre-processing step by re-sampling or re-weighting the training data in proportion of their misclassification cost.

- Modification of the learning algorithm

  This approach modifies the error-based classifiers directly to handle misclassification cost, but each classifier needs to be modified separately.

- Boosting approach

  This approach generates a set of different weak classifiers in sequential trial and then constructs a composite classifier by voting them. This approach is that it is applicable to any kind of error-based classifiers.

- Conditional probability estimates / Direct cost-sensitive learning

  This approach incorporates the misclassification cost into the data post-processing step by using the probability estimation generated by error-based classifiers and the cost function to directly compute the optimal class for each test example. This approach is easy to implement, but needs good calibration methods to generate accurate probability estimation.

Note that the use of the costs as in one of the approaches presented here does not preclude a subsequent evaluation of the costs. In this way it can be checked whether a classifier who incorporates the costs into the learning process is actually superior to a classifier without knowledge of the costs.

3 Concept

This chapter describes how to solve a general problem of extracting knowledge out of natural language sources and defines which steps have to be considered. Therefore, in the following the focus is set on universal approaches to accomplish such a task. Specific ideas and solutions adjusted to the sentiment classification task for chess annotations will be dealt with in chapter 4.

The solution of a text mining problem comprises several steps from definition to evaluation. Figure 6 shows a possible process model based on approaches of Schieber and Hilbert [SH14] and Kwartler [Kwa17]. The model consists of six steps, which are now presented one after the other.
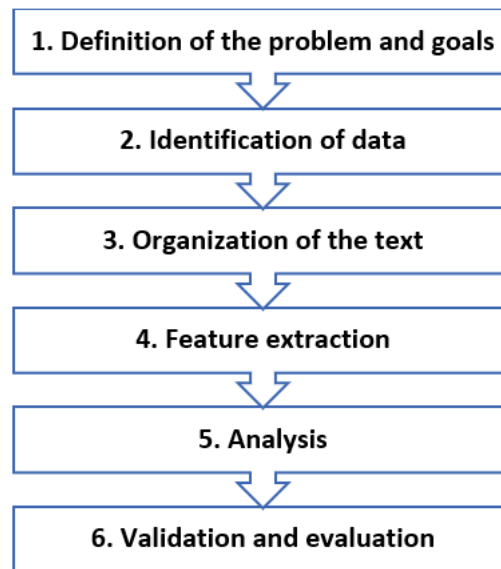
Figure 6: Processing pipeline of text mining

3.1 Definition of the Problem and Goals

In the first step we have to define the problem we want to solve. According to Mitchell a learning problem generally reads as follows: Improve over task T, with respect to performance measure P, based on experience E. The goal is therefore to generalize the experience in a way that allows to improve your performance on the task [Mit97, chapter 1]. The three parameters T, P and E are described in more detail in the following.

- Task T

  The task can be formulated by a simple verbal description, e.g. "play the game backgammon" or "sort the e-mail into the categories spam and no-spam". A more formally approach requires the exact definition of the input and the output of the task.
  The input could be a map or a dictionary, i.e. a set of attribute-value pairs. In relational data, each input would have the same set of attributes and therefore the same structure and length. But since there is no restriction to the input data to be homogeneous the input can be plain text as well. A possible input of natural language could be represented by a whole book, by a (web) page, by a paragraph or just by one sentence. In certain cases, even a single letter is an appropriate input, e.g. for the detection of handwritten letters.
  As well as for the input we need to determine the type of output we want to receive. But not only the type, also the precision in the range of values is important for the difficulty of the task. In the case of product reviews, the easiest output "good review vs. bad review" could be complicated by using the ten values of a five-star rating allowing half stars or by distinguishing between different ratings for the quality, the price-performance ratio, the delivery etc. The aforementioned examples have in common that the number of possible output values is fixed which means a classification problem is concerned, not a regression problem. In this work the focus will be set on classification tasks.

- Performance measure P

  The straightforward approach to create a performance measure for classification tasks would be to count the instances with the correct output and divide this number by the total number of instances. This value is called

accuracy. Just as well the complementary probability for misclassified instances, called error rate, can be observed. However, there are also possibilities to weight the classifications. If there are instances that are more important than others, these instances can be multiplied or associated with a weight greater than one. Furthermore, the misclassifications can be considered separately dependent on the correct and predicted class. In spam classification, it is usually significantly worse to classify a no-spam e-mail as spam than the other way around. To map this idea, we can assign to each pair of correct and predicted class a weight value.

This type of measurement is not appropriate for regression tasks. Dependent on the exactitude of the values the probability that the predicted value is exact the correct one is low. Instead of demanding an exact prediction, we can also use the difference between correct and predicted value as the performance (MAE) or the squared difference (MSE) [WF05, section 5.8].

In the case of an unsupervised problem only subjective estimates can be used. The learned model and its output are evaluated by an expert, which might entail a high expenditure of time and money.

The validation and evaluation techniques will be dealt with in detail in section 3.6.

- Experience E

  The experience of a learning problem is the knowledge the learner already possesses before solving the task. For example, a student solves exercises to gain experience before writing an exam. In machine learning, the experience is achieved through the acquisition of knowledge from databases. In both cases the learner tries to improve the performance in the task by the usage of the knowledge of similar data. If no similar data is already known, the learner can just guess the correct output. For this reason, a sufficiently large amount of data should be available to the learner in order to increase the experience and therefore also the chance to solve the task in the desired manner.

The following sections describe the procedure as generally as possible; we only assume that we face a classification problem in the field of text mining. Furthermore, the output value should be known for all instances, such that tasks of supervised learning can be applied automatically. As the input we use a document $d \in \mathbb{X}$, where $\mathbb{X}$ is the document space; and as the output a fixed set of classes $\mathbb{C} = \{c_1, c_2, \ldots, c_J\}$. Using a learning method or learning algorithm, we then wish to learn a classifier or classification function γ that maps documents to classes, i.e. $\gamma : \mathbb{X} \to \mathbb{C}$ [MRS08, section 13.1].

3.2 Identification of Data

In the second step we need to find one or several data sources that offer an adequate number of instances, i.e. data sets of the previously defined input and output type. There are four characteristics to be aware of:

- Completeness

  Each instance of the data should be complete, i.e. we know the input and the output of the instance. All attributes of the input should be filled with a value. If they are not filled, a default value can be used or another way how to manage missing values needs to be defined. Of course, there is also a need of the completeness of the output value. Otherwise, the instance cannot be used for training nor for the evaluation.

- Format

  In addition to the completeness, the data needs to be in the same format. Using data from data sources with different syntactical structure requires a pre-processing, such that the data can be compared and processed in a similar way in further steps. This involves the order and the atomicity or distribution of the data.

- Quantity

  To be able to discover meaningful knowledge, we need a minimal data volume, usually starting from several hundreds or thousands of instances. This concerns also the absolute and the relative amount for each class or even for the most important attribute values. An upper limit of instances does not exist. However, sufficient computing and storage capacities must be available.

- Quality

  The quality of the data goes along with the completeness, but it goes one step further. The desired values should not only be existent but also accomplish quality requirements, such as observing minimal and maximal values (numeric attributes) or lengths (linguistic attributes), providing a minimal precision or being available in a specific language. Analogous to checks and other constraints in databases, the data could be validated before using it in the mining process.

After completing the second step we end up with an appropriate data set for the task defined in the first step. The data is in a homogeneous representation, but not necessarily in a structured form.

A big part of all data in the internet exists in the form of natural language. Usually, it is hard to evaluate the information contained in this data, because it is not structured in the same way. For example, in product reviews every customer can write his comment in a different kind, so that there is neither a certain order of the information nor a specification, which information the comment should provide. However, on the basis of an additional star rating it is possible to get a fast assessment of the customer's attitude towards the product. So, if there is a need of further evaluation, it is helpful to have the data in a structured form instead of an unstructured form.

Often it is not desirable or even impossible to get the unstructured data directly in a structured form, so we have to do the transformation on our own. This process of converting the data from an unstructured into a structured form is called information extraction. It is assumed that the input data is a single string, i.e. a sequence of characters. The sequence can contain just a few characters (e.g. tweets, comments) or thousands of characters (e.g. book contents). Even if in the most common cases this string is natural language, the procedure is similar for other input strings as well. In order not to falsify the data, we have to take into account the character encoding. The goal of this step is to reorganize the input from a string to a collection of tokens, also known as lexing or tokenization. This process takes place in three sub-steps:

- Token Identification

  A lexical token, shortly token, is a string with an identified meaning. The input string is split into substrings where each substring without meaning is discarded and all others are stored as tokens. The definition of a token depends on the use case. In natural language processing, the standard approach is to use blanks or other whitespace characters and interpunctuation symbols as separators of tokens. As a result, there are words as tokens. This general solution can be customized by interpreting whitespace characters and interpunctuation symbols (or combinations of them) as tokens. Words can also be further split at a hyphen or split into characters. The other way around, two or more words can be combined to one token, e.g. for names of persons or places ("New York") or for usual phrases ("and so on"). All of the parts of the string that do not contain any information should be removed directly. Frequently used words as "and" or "that", also called stopwords, do not provide added value and can be removed from the token collection. There are lists of stopwords available for common textual resources, but they may be customized for specific data sets. Other challenges in natural language processing include handling spelling mistakes, acronyms and special characters such as smileys [AS16].

- Normalization

  In this step we want to find tokens that should be treated as identical, even if the strings of the token are different. In the simplest case we can make the interpretation of a token case-insensitive by converting all occurrences into lowercase. Before doing so it should be ensured that no information is lost as a result. In sentiment analysis, a large proportion of uppercase letters could indicate rage. Other types of normalization are lemmatization and stemming. Words in natural language can be modified by inflection, most of all by conjugation (modification of verbs caused by person, tense etc.) and declination (other part of speech caused by case, gender, number etc.). Lemmatization brings all parts of speech back into its basic form, e.g. the singular nominative case for nouns or the infinitive for verbs. Stemming reduces all words independent of its part of speech to the word stem, which need not be a proper word. Though, as a consequence information about the original token get lost which can lead to incompatibilities with the further analysis in the next steps. Obviously, the order of normalization and token merging or deletion from the first step may influence the result as well and should therefore be made consciously. Besides, not only the input, but also the output can be normalized. The merging of two or more output values to one joint class can also be considered as a normalization step.

- Categorization

  This step deals with the syntactic and the semantic analysis of the tokens. Syntactic analysis, also known as parsing, is the process of assigning each token a category describing its function in the context. In classical parsing of code files, categories can be identifiers, keywords, literals etc., in parsing of natural language resources, categories can be nouns, verbs, adjectives etc. Last-mentioned is known as part-of-speech-tagging (POS-tagging). Building on the syntactic analysis we can also assign a meaning to the token in addition to the category. This process is called semantic analysis and can cover difficulties of synonyms and polysemes (ambiguous words). These semantical relations and more for English vocabulary are provided by the lexical database WordNet[5].

After finishing the chosen approaches of those three steps, we obtain the unstructured text data in a structured collection of tokens, possibly extended or replaced by their category or meaning. By splitting the string into tokens in the beginning,

---

[5] https://wordnet.princeton.edu/

the tokens in the collection will be sorted in the same order as they occur in the input. The collection is in list form and the order is maintained. If the order and the count of tokens is irrelevant for further evaluation, a set can be used instead of a list. However, in most cases just the order is irrelevant but not the count of tokens, such that a multiset is the suitable form of collection. This multiset is known as "bag of words".

3.4 Feature Extraction

Based on the created output from the text organization step, now the characteristics of the input data are to be figured out by feature extraction. The idea is to calculate various scores, such that we can compare the input data instances with each other, especially concerning the sentiment and polarity. At the end of this step we want to obtain a representation that can be passed as a training set to learning algorithms.

The default feature extraction approach is typically used in web search engines during the indexing step in information retrieval. The input is a list of documents in bag of words-representation or something similar. This results in a document-term-matrix, where each row describes a document and each column the count of a specific term. This concept can be generalized and transferred to our question. The column remains the description of the document as a vector whereas each column is the value of a specific feature. The model is called Vector Space Model (VSM) as it represents each document as a vector of features which simplifies the comparison of two documents by cosine similarity. Possible features with direct relation to the token are:

- binary indication

  This feature holds the value 0, if the token does not occur, and the value 1 otherwise. This value follows immediately if the collection is represented as a simple set.

- term frequency (TF)

  This feature holds the count of the token as described in the information retrieval example. This value follows immediately if the collection is represented as a multiset. The value can be normalized by dividing it by the total number of tokens in the document. Another option is to relativize very high term frequency values by using the logarithm function.

- term frequency – inverse document frequency (TF-IDF)

  In contrast to the term frequency, TF-IDF decreases the weight by taking into account the number of documents in which a token appears. The idea is to reward rarely occurring tokens with a high feature value. If the token appears in all documents, the value will be 0. If the token appears in just one document, the value is maximal.

Features need not be dependent of only one token. There are more advanced features that use accumulation of several tokens like the average token length. The above-mentioned values can also all be calculated if the tokens are grouped by category [WZ10]. For problems concerning long input texts in natural languages like author detection another appropriate feature is the lexical diversity. It calculates the ratio between the count of distinct words (vocabulary) and the total word count.

If the collection is passed in list representation, we can additionally create features by using combinations of sequential tokens. These sequences are called n-grams in general; 2-grams (bigrams) and 3-grams (trigrams) are particularly frequently used. With n-grams, the context also flows into the analysis, which is an advantage, for example, when recognizing negations ("not", "good" vs. "not good").

It makes sense to find in the first step as much features as possible while the computing and storage capacities are not exceeded. Thus, we get a large amount of potentially informative features. However, before passing the feature data to the learning algorithm, the dimension should be reduced to avoid redundancy and to accelerate the learning process. This step called feature selection is of particular importance for classifiers that, unlike Naïve Bayes, are expensive to train. Second, feature selection often increases classification accuracy by eliminating noise features. A noise feature is one that, when added to the document representation, increases the classification error on new data. Such an incorrect generalization from an accidental property of the training set is called overfitting [MRS08, section 13.5].

Selecting a subset of features requires a measurement value to compare the utilities of the features. Typical methods are Gini index, information gain, mutual information, χ^2 or frequency-based feature selection [AZ12, section 2.1]. All of those methods are greedy, which can lead to redundant features. There are non-greedy methods that avoid redundancy, but they are rarely used in practice due to increased computing effort [MRS08, section 13.5]. Finally, regardless of the choice of measurement features the features can be selected if they exceed a threshold value or all features are ranked

and the best n features are selected.

A different approach for representing text data are word embeddings. The previous approaches are based on the one-hot representation: the feature vector has the same length as the size of the vocabulary. Besides the already mentioned problems due to less meaningful features and overfitting, in a one-hot representation the model cannot handle words that do not appear in the labeled training data [TRB10]. In contrast to this, word embeddings offer a distributed representation which is dense, lowdimensional, and real-valued. They are learned on a big general corpus and can therefore associate each word with a corresponding vector. The combination of those vectors yields the document vector. The features and advantages of word embeddings have already been discussed in section 2.2.

3.5 Analysis

In the previous chapter, the data was prepared so that we can now apply analysis procedures to the data in vector representation. The goal of this step is to find characteristics, dependencies and rules to be able to make predictions for new input data. In figure 7 there are some possible approaches listed that can be applied to sentiment analysis problems. They can be divided into the two techniques of machine learning approaches and lexicon-based approaches [AS16].
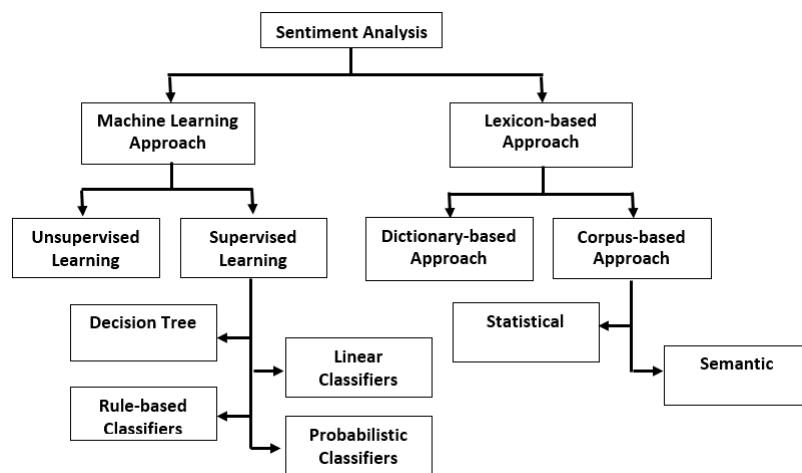
Figure 7: Learning approaches for sentiment analysis [HM18][6]

Machine learning approaches use mathematical models built by an artificial intelligence to solve the task given to them. They are split into unsupervised and supervised learning. In unsupervised learning no label of a class is provided, so there is no possibility to compare the calculated solution with the correct one. In this case a common learning approach is clustering. The input data is segmented into a (fixed) number of groups, where instances within a group should have similarities and thus form a category. In contrast, there are procedures that are to determine an output value for a given input. These are in the area of supervised learning. While regression methods can handle the assignment of continuous values, classification methods can only be applied if there is a limited and fixed number of possible output values, known as classes.

We already defined the domain and codomain of a classification function $\gamma : \mathbb{X} \rightarrow \mathbb{C}$ where $\mathbb{X}$ is the set of documents and $\mathbb{C}$ the set of classes. Irrespective of the choice of classification algorithm, it is recommended to split the evaluation of the procedure into a training set and a test set. Based on the training set, a model can be created that learns the characteristics of a class. Partly the training set is further subdivided, whereby a model is created on the first part of the training set and fine-tuning is performed by the second part of the set (validation set). During the evaluation the correct classes of the test instances are removed. Without prior viewing, now new classes are assigned to these instances by the learned model, which are then compared with the correct classes. If, as in the case described, only one class is assigned to each instance, it is the hard version of classifying. On the contrary, the soft version assigns to an instance a probabilistic value for each class [AZ12]. In the following some such classification learning approaches are introduced, which can be used in the area of text mining:

- Decision Trees

  Decision trees are designed with the use of a hierarchical division of the underlying data space with the use of different (text) features. From top to bottom, the partitions become more and more homogeneous by selection

---

of one or several appropriate features for the decision which can be determined by measurement like information gain or Gini coefficient. This procedure can be continued until all leaves are partitions containing only one class. To avoid overfitting and reduce complexity, scarcely informative sections of the tree can be cut by pruning. Finally, a test instance is associated to the class of the partition the decision path starting from the root leads to.

- Pattern- or Rule-based Classifiers

Rule-based classifiers are similar to decision trees; more precisely, decision trees can be represented as a set of rules. For each rule the left-hand side is a condition on the underlying feature set (usually expressed in Disjunctive Normal Form (DNF)), and the right-hand side is the class label. A rule should have a high support (absolute number of instances affected by the condition) as well as a high confidence (probability for class if condition is given). In text mining, the rule is typically expressed as a conjunction of terms that have to appear in the instance. The absence of terms is rarely used, because such rules are not likely to be very informative for sparse text data. For any test instance the class of the first rule where the condition is fulfilled will be assigned. That is why the last rule should cover all remaining instances and provide a default class.

- Probabilistic Classifiers

Probabilistic classifiers like Bayesian classifiers calculate for each class a probability, whereby the class with the highest value is chosen for the test instance. Naïve Bayes classifiers use therefore the product of all conditional probabilities, e.g. term occurrences in different classes of texts. In comparison to other classifiers, Naïve Bayes classifiers are highly scalable because of its linear complexity. Suitable probabilistic classifiers for text mining are the Bernoulli variate model and multinomial distributions.

- SVM Classifiers

Support-vector machines (SVM) are a subgroup of linear classifiers. A linear classifier calculates for a binary classification problem a linear predictor $p = \mathbf{a} \cdot \mathbf{x} + b$, where $\mathbf{x} = (x_1...x_n)$ is the feature vector of the input, $\mathbf{a} = (a_1...a_n)$ is a vector of linear coefficients with the same dimensionality as the feature space and b is a displacement constant. A natural interpretation of the linear predictor in the discrete scenario would be as a separating hyperplane between the different classes. The hyperplane with the maximum distance value to any instance, i.e. the one with the maximum margin of separation, is chosen. The SVM approach is quite robust to high dimensionality and ideally suited for text data because of the sparse high-dimensional nature of text [Joa97].

- Neural Network Classifiers

Simple neural networks are also a form of linear classifiers, since the function computed by a set of neurons is essentially linear. The simplest form of neural network, known as the perceptron (or single layer network) are essentially designed for linear separation, and work well for text. However, by using multiple layers of neurons, it is also possible to generalize the approach for non-linear separation. In such a network, the outputs of the neurons in the earlier layers feed into the neurons in the later layers. The training process of such networks is more complex, as the errors need to be back-propagated over different layers.

- Proximity-based Classifiers

Those classifiers use proximity measures for classification. Given n input attributes, all training instances are placed in a n-dimensional space. The proximity of two documents can be calculated by distance measurements like Euclidian or Manhattan distances. Then, the most common k-nearest-neighbor classifier identifies for a given test instance the k training instances with the smallest distance values. The most common or highest-weighted class among them becomes the class of the test instance.

In the simplest version all of the six classifier types are applied to binary classification problems. If there are three or more classes, the classifier possibly has to be extended to handle this multiclass problem, e.g. for SVM Classifiers. Another possibility is to split the multiclass problem into several binary problems as seen in section 2.3. The classification is made by several classifiers that are trained to differentiate either between one class and the rest (one-against-all) or pairwise between each two classes (one-against-one). Further concepts for the customization of classifiers are boosting and bagging, the formation of ensembles or the handling of ordinal classes [AZ12].

For sentiment analysis problems, apart from machine learning approaches also lexicon-based approaches are suitable. Lexicon-based approaches mainly rely on a sentiment lexicon, i.e., a collection of known and precompiled sentiment terms, phrases and even idioms, developed for traditional genres of communication. To determine the polarity score of a text, the polarity scores of the terms are combined in a certain way, e.g. by addition. There are two subtypes of lexicon-based approaches. The dictionary-based approach uses a list of terms called dictionary, where the collection and

scores are both created manually. Manual creation may be time-consuming, but it is usually possible to create optimally customized dictionaries with good results. The corpus-based approach uses already existent dictionaries of a specific domain, which have been constructed based on a big corpus. There are statistic techniques like latent semantic analysis (LSA) as well as semantic techniques using synonyms, antonyms or other relationships from thesaurus like WordNet [AS16].

3.6 Validation and Evaluation

The final step deals with the validation and evaluation of the results obtained from the previous analysis step. In section 3.1 some performance measurements already have been presented. In the following only measurements for classification problems are discussed. Before discussing about those measurements, first of all we will specify how to arrange the data meaningfully to be able to perform most of the measurements.

We already mentioned that for the evaluation the data should be split into a training and a test set. Formally speaking, a training set $\mathbb{D}$ of labeled documents $\langle d, c \rangle$ is given, where $\langle d, c \rangle \in \mathbb{X} \times \mathbb{C}$. For all unlabeled documents d in the test set, the chosen classification function(s) γ has to predict a class c [MRS08, section 13.1]. Generally, the larger the training set the better the classifier, although the returns begin to diminish once a certain volume of training data is exceeded. And the larger the test set, the more accurate the error estimate. There are different approaches to deal with this problem [WF05, chapter 5]:

- Holdout Method

  The simplest approach is to determine a specific percentage that should be used for training (usually two thirds). The rest of the data is cut and hold out for the final validation of unseen data. Note that here data can either be used for training or for testing.

- k-Fold-Cross-Validation

  The cross-validation tries to get away from wasting data by using all data both for training and for testing in different iterations. The data is split into k folds (usually 10); $k - 1$ of them are used for training and the remaining one for validation. The final measurement is the average of the measurement of all k iterations and should outbid the simple holdout method in the performance value as well as in its accuracy. A non-random decomposition into folds, e.g. by previous sorting of the data records, can falsify the results.

- Leave-One-Out-Validation

  For even smaller data sets the leave-one-out-validation is a suitable approach. Here only one test instance is validated while the rest is used for training. This results in n possible iterations where n is the size of the data set. Similar approaches for small data sets with even more possible iterations are the leave-k-out-validation ($\binom{n}{k}$ possiblities) and the bootstrap method [AC10, chapter 4].

Of course, it could be also possible not to distinguish between training and test data, i.e. the model is built and evaluated based on all data. However, this approach can lead to overfitted and therefore overestimated performance values that generally cannot be sustained in real application cases when predictions have to be made for unseen data.

The accuracy has already been presented as a simple yet meaningful measurement value. Before it is calculated, a confusion matrix M is usually created, where a value M_{ij} specifies how many instances of the class i were classified as class j. The accuracy is the sum of all values in the main diagonal of the confusion matrix (i.e. $i = j$) divided by the total number of instances.

Instead of using an absolute accuracy value it can make sense to use a relative one. In classification problems a baseline can be used as a comparison value for the accuracy. Witten and Frank use an expected value calculated by a default classifier as shown in table 2. They assume that the comparison classifier predicts each individual class with the same frequency as the original one. This classifier predicts for $60 + 18 + 4 = 82$ instances the correct class, while the original does for $88 + 40 + 12 = 140$. This results in $140 - 82 = 58$ additional correct classifications out of $200 - 82 = 118$ and a kappa statistic of $58 \div 118 = 49.2\%$. Another meaningful value to compare is the baseline accuracy, which is achieved by a classifier always selecting the most frequently occurring class. In the example above the baseline classifier achieves 100 correct classification which would decrease the kappa statistic to 40%. Obviously, a kappa statistic of 100% would indicate a perfect classifier, a value of 0% no improvement and a negative value even a deterioration.

Actual class		Predicted class			Total
		a	b	c	
	a	88	10	2	100
	b	14	40	6	60
	c	18	10	12	40
	Total	120	60	20	200

(a) actual

Actual class		Predicted class			Total
		a	b	c	
	a	60	30	10	100
	b	36	18	6	60
	c	24	12	4	40
	Total	120	60	20	200

(b) expected

Table 2: Outcomes of three-class-classification, cf.[WF05, section 5.7]

In the kappa statistics just presented it is assumed that all false classifications are equally weighted. However, there are also cases where a classifier with negative kappa statistics is preferable to a baseline classifier. Medical classifications of patients into the classes healthy and sick are a prime example of this. A classifier with 98% accuracy caused by 2% of false positives (healthy patients misclassified as sick) is usually preferable to the baseline classifier with 99% accuracy and the only output healthy and therefore 1% of false negatives (sick patients misclassified as healthy). This is because in this case false negatives weigh heavier than false positives. For this reason, in cost-sensitive classification problems the exact cost factors for each cell in the confusion matrix have to be defined before determining a baseline classifier and an appropriate kappa statistic. In ordinal classification problems the baseline classifier with minimal costs might have the average class as the only output instead of the most common class. The influence of costs in cost-sensitive learning have already been discussed in section 2.5.

Apart from the accuracy there are further measurement values like recall and precision. Given the confusion matrix M, precision and recall are defined as

$$precision_i = \frac{M_{ii}}{\sum_j M_{ji}} \quad \text{and} \quad recall_i = \frac{M_{ii}}{\sum_j M_{ij}}$$

The precision thus describes how high the quota of correctly classified instances is among all instances classified as class i, while the recall describes the quota of correctly classified instances among all instances that actually belong to class i. In binary classification problems, the formulas are simplified into

$$precision = \frac{tp}{tp+fp} \quad \text{and} \quad recall = \frac{tp}{tp+fn}$$

where tp is true positives, fp false positives and fn false negatives. Precision and recall generally have a negative correlation and cannot be optimized at the same time. As a compromise the F-measure can be used which forms the harmonic mean of precision and recall:

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

The Receiver Operating Characteristic (ROC) is an alternative to the precision and recall measurement which visualizes the ratio of true and false positives. The confusion matrix provides only one single point for a classifier, which is generally the better the closer it is to the optimal point $(0|1)$. This point indicates 0% false positives and therefore 100% true negatives and optimal precision as well as 100% true positives and therefore 0% false negatives and optimal recall. Because false positives and false negatives can be weighted differently, the Euclidian distance has to be adjusted by different weights for the x- and y-direction. If the output values of a classifier are ranked (e.g. by probability or relevance), a whole ROC curve can be created

- starting at the point $(0|0)$,
- moving $\frac{1}{P}$ unit to the top in case of a true positive (where P is the total number of positives),
- moving $\frac{1}{N}$ unit to the right in case of a false positive (where N is the total number of negatives)
- and ending at the point $(1|1)$.

In this case, the Area Under ROC Curve (AUC) is another possible indicator for a good classification. It also has a nice interpretation as the probability that the classifier ranks a randomly chosen positive instance above a randomly chosen negative one. Although those definitions and interpretations only work for binary classification problems, there are customized variants for multiclass classification, see [HT01].

4 Experimental Setup

In this chapter we consider the general problem described in the previous chapter this time in the field of chess annotations. Therefore, in the beginning of this chapter, the basic chess game format PGN and the corresponding annotations NAGs are introduced. Afterwards we will follow the six steps on which the process is based on. In the following all necessary definitions and tools are presented and some statistics of the data are given. The results of the analysis and their evaluation are discussed in chapter 5.

4.1 Problem Description

Chess games can be recorded in plain-text-files with the aim of reviewing and analyzing the game later on. Especially in professionally chess tournaments it is common to note not only the player data and the moves, but also some additional information about the game dynamics like an interim position or decisive good or bad moves. There are two ways to describe these game dynamics and annotate the chess game; either by comments in any natural language or by standardized codes and symbols like NAGs. Combinations of both variants are also common.

For the evaluation of chess games and their machine processing the unified and structured form given by the standardized codes and symbols is preferable to the unstructured form of the comments in natural language. For chess games which only contain annotations in commentary form, it would therefore be helpful to also provide them with standard codes. This results in the problem of converting the comment into an appropriate code. Using the scheme of Mitchell presented in section 3.1, we have:

- Task T : determine the correct standard code for a given chess annotation comment
- Performance measure P : percentage of correctly assigned codes in all code assignments (accuracy)
- Experience E : database with tuples of comments and correct codes

```
[Event "Deutschland "]
[Site "?"]
[Date "1995.??.??"]
[Round "?"]
[White "Lutz, Ch"]
[Black "Kramnik, V."]
[Result "0-1"]
[ECO "B33"]
[PlyCount "70"]
[EventDate "1995.??.??"]
```

1. e4 {B33: Sicilian: Pelikan and Sveshnikov Variations} **1... c5 2. Nf3 Nc6 3. d4 cxd4 4. Nxd4 Nf6 5. Nc3 e5 6. Ndb5 d6 7. Bg5 a6 8. Na3 b5 9. Nd5 Be7 10. Bxf6 Bxf6 11. c3 O–O 12. Nc2 Bg5 13. a4 bxa4 14. Rxa4 a5 15. Bc4 Rb8 16. b3 Kh8 17. O–O g6 18. Qe2 Bd7 19. Rfa1 19... Bh6** {last book move} **20. g3** { Consolidates f4} (20. Nde3 20... Be6 $14) **20... f5** $11 **21. exf5 gxf5 22. b4 22... e4** {Black wins space.} **23. bxa5 Ne5 24. Rb4 Rxb4 25. cxb4 f4 26. Nd4 e3 27. fxe3** (27. Nxf4 $2 {doesn't work because of} 27... exf2+ 28. Qxf2 28... Bxf4 $19) **27... f3** {He broke from his leash} (27... fxg3 28. hxg3 Qg5 29. Kh2 Nxc4 30. Nf4 $19) **28. Qa2 f2+ 29. Kg2 Qe8 30. Be2 30... Ng4** { The pressure on the isolated pawn grows} **31. Bf3** $4 (31. Qd2 Qh5 32. Bxg4 Qxg4 33. Nf4 Bxf4 34. exf4 Qh3+ 35. Kxf2 Qxh2+ 36. Ke1 Qxg3+ 37. Kd1 Qg1+ 38. Ke2 Bg4+ 39. Kd3 Qxa1 40. f5 $19) **31... Nxe3+** $19 **32. Nxe3 Qxe3 33. Qxf2** $4 { sad, but how else could White save the game?.} (33. Rd1 Bg7 34. Qb3 Bxd4 35. Qxe3 Bxe3 36. Be2 $19) **33... Bh3+** $1 {the final blow} **34. Kg1** { Black now must not overlook the idea Re1} (34. Kxh3 {A deflection} 34... Qxf2) **34... Qc3 35. Re1 Bd2** (35... Bd2 36. Ne2 36... Qxf3 $19 (36... Bxe1 $6 { is clearly weaker} 37. Nxc3 Bxf2+ 38. Kxf2 $19) (36... Rxf3 $2 37. Nxc3 Rxf2 38. Kxf2 Bxc3 39. Re7 $18)) **0–1**

<div align="center">blue: comments   red: NAGs   bold: game moves</div>

Figure 8: Sample PGN game

Before further specifying the input and output of the task T in section 4.1.3, we will first take a closer look at the structure of the chess data. The data sets used for the experience E will be presented in section 4.2 and the performance measure P will be adapted to the problem in section 4.6.

4.1.1 PGN Format

PGN is "Portable Game Notation", a standard designed for the representation of chess game data using ASCII text files. PGN is structured for easy reading and writing by human users and for easy parsing and generation by computer programs [Edw94, chapter 1]. A sample game in PGN notation is shown in figure 8.

A PGN game contains first a list of tuples with general information of the game ("tag pairs"). Seven of those tags are mandatory (Seven Tag Roster: Event, Site, Date, Round, White, Black, Result), the other tags are optional. Afterwards the "movetext" section starts. The chess moves themselves are represented using SAN (Standard Algebraic Notation). A move pair (one move of white and one of black) starts with the move pair number followed by a dot and a blank, then the move of white, another blank and the move of black, e.g.

> 7. Bg5 a6.

Each move contains the piece by a single upper-case letter except of the pawn (see table 3) followed by the square the piece is moved to (see figure 9). Hence, the example describes the seventh move of both players in the game; white moves his dark-squared bishop to the square g5 and black moves his a-file-pawn to a6. If a piece of the opponent is placed on the destination square, this piece is captured and in the move an "x" is inserted immediately before the destination square. In this case, if the capturing piece is a pawn, the lower-case letter of the previous file of the pawn is used at the beginning of the move, e.g. "exd5". Whenever a move pair is interrupted by a comment, the move of black is prefaced by the move pair number, an ellipsis and a blank:

> Nxf4 $2 {doesn't work because of} 27... exf2+

Additionally, there are some further moves with a special notation (see table 3). In cases of disambiguation of pieces, an additional letter for the file or a number for the rank is used. In summary, a move can contain between two and seven signs in SAN [Edw94, chapter 8].

Symbol	Meaning		Symbol	Meaning	Example
K	King		x	Capture	Rxa1
Q	Queen		+	Check	Nf6+
R	Rook		#	Checkmate	Bb7#
B	Bishop		0-0	Castling kingside	
N	Knight		0-0-0	Castling queenside	
blank	Pawn		=	Promotion	fxg1=Q+

Table 3: Basic chess notations
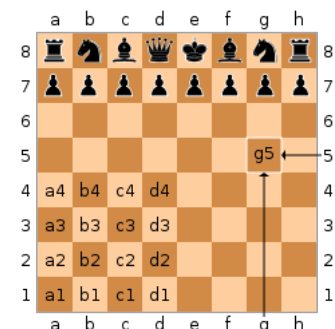
Figure 9: Square names in algebraic notation[7]

Parts of the moves are annotated using comments in braces. A comment can contain information about the opening of the game, about a single move or about the current position. In the last two cases the comment is often prefaced by one or several NAGs (see section 4.1.2) or the corresponding chess symbol. Since there is no restriction on the exact position of a comment, comments may refer to the move before or after itself. A comment can also connect two or more moves with each other. On the contrary, a comment can be interrupted by a move such that it is split into two parts, which may only make sense when seen together. All in all, there are four possibilities of comment-move combinations shown in the examples of table 4.

Combination	Example
Move, Comment	e4 {Black wins space.}
Comment, Move	{Weaker is} 39. Bxe6
Move, Comment, Move	Nxf4 $2 {doesn't work because of} 27... exf2+
Comment, Move, Comment	{Because of the blunder} 24. Txf8 {Black wins immediately}

Table 4: Comment-move combinations

Besides, by convention there should not be nested braces, however, sometimes nested braces are used to comment different move variants separately. Those variants need not be part of a comment and are written down in parenthesis. The enumeration of the moves proceeds within a variant and is set back before a new variant starts or the game itself continues.

4.1.2 NAGs

Numeric Annotation Glyphs (NAGs) are used to annotate chess games with assessments of moves or positions in a standard way. They are standard annotation symbols in PGN files, but can as well be used in other chess formats. A NAG is composed of a "$" followed by one or more digits. There are 140 standard NAGs in total:

- NAG zero is used as a placeholder
- NAGs with values from 1 to 9 annotate the move just played.
- NAGs with values from 10 to 135 annotate the current position.
- NAGs with values from 136 to 139 describe time pressure.

The NAGs with values from 140 to 255 are partially defined and used unofficially. The most common NAGs are listed in table 5 (see [Edw94, chapter 10]).

NAG	Symbol	Meaning
$1	!	good move
$2	?	poor move
$3	!!	very good move
$4	??	very poor move
$5	!?	speculative move
$6	?!	questionable move

(a) move-annotating NAGs

NAG	Symbol	Meaning
$10	=	drawish position
$11		equal chances, quiet position
$12		equal chances, active position
$13	∞	unclear position
$14	+ =	White has a slight advantage
$15	= +	Black has a slight advantage
$16	±	White has a moderate advantage
$17	∓	Black has a moderate advantage
$18	+−	White has a decisive advantage
$19	−+	Black has a decisive advantage

(b) position-annotating NAGs

Table 5: Meaning of NAGs

As shown in table 5, the most common NAGs have a corresponding symbol, which has been used traditionally. Those symbols are composed of the signs "!", "?", "+", "-", "=" and special signs. It should be emphasized that the subjective symbols do not mix up with the objective move symbols for check and promotion because they are used in different combinations.

4.1.3 Problem Specification

Now the structure of PGN chess files and the corresponding NAGs being clarified, we can identify use cases in which a sentiment analysis of chess annotations might be useful. In PGN files, comments are often assigned to the NAG that precedes this comment. If no NAG is given - which is the case for more than half of all comments (see table 7) - we could assign the correct NAG automatically if we would have a reliable learned model. Thus, we will collect the data of already correctly mapped comments and NAGs and recognize contained patterns therein. Concretely, the following problems will be discussed:

- Classification into move and position annotations

  As already seen in table 5, the NAGs are subdivided into NAGs annotating moves and NAGs annotating positions (those describing time pressure are rarely used and therefore negligible). Both annotation types are used at the same place in the PGN file, in particular directly after a move. Therefore, we need to recognize and learn other patterns in order to distinguish these two types of annotations. For this learning problem, the input space $\mathbb{X}$ and output space $\mathbb{C}$ are defined as follows:

  $$\mathbb{X} := \text{set of chess comments without annotation} \quad \mathbb{C} := \{1, 2\}$$

  The output class 1 is used for move annotations and the class 2 for position annotations.

- Classification of move annotations

  Among the move-annotating NAGs there are basically two groups of annotations; positive and negative ones. It should be noted that positivity and negativity does not refer generally to white or black, but from the viewpoint of the player with the move directly before the NAG. We can formulate the classification problem on the same input space in two degrees of difficulty:

  $$\mathbb{X} := \text{set of move comments without annotation} \quad \mathbb{C}_1 := \{1, 2\} \quad \mathbb{C}_2 := \{1, 2, 3, 4, 5, 6\}$$

  In the first output set, the class 1 is assigned to all positive move annotations (i.e. \$1, \$3, \$5) and the class 2 to the negative ones (i.e. \$2, \$4, \$6). In the second output set, each of the six NAGs gets an own class ranked by their "positiveness". This converts the binary classification problem to an ordinal classification problem with the following mapping of NAGs to classes (1 = most positive, 6 = most negative):

  $$1 : \$3 \quad 2 : \$1 \quad 3 : \$5 \quad 4 : \$6 \quad 5 : \$2 \quad 6 : \$4$$

- Classification of position annotations

  With the position-annotated NAGs we have a similar situation, but with the decisive difference that a neutral class also exists. So even the simpler classification problem already contains three classes:

  $$\mathbb{X} := \text{set of position comments without annotation} \quad \mathbb{C}_1 := \{1, 2, 3\} \quad \mathbb{C}_2 := \{1, 2, 3, 4, 5, 6, 7\}$$

  In the first output set, the class 1 is assigned to all position annotations with an advantage of white (i.e. \$14, \$16, \$18), the class 2 to the balanced position annotations (i.e. \$10, \$11, \$12, \$13) and the class 3 to the annotations with an advantage of black (i.e. \$15, \$17, \$19). In the second output set, classes 1 and 3 are each divided into three subclasses, which makes a total of seven ordered classes (1 = best for white, 7 = best for black):

  $$1 : \$18 \quad 2 : \$16 \quad 3 : \$14 \quad 4 : \$10, \$11, \$12, \$13 \quad 5 : \$15 \quad 6 : \$17 \quad 7 : \$19$$

Note that in all cases only one of the output values can be assigned, i.e. we only face single-label problems.

4.2 Data Set Extraction

As data sources a set of files[8] in standard PGN format is used as well as a bundle of commented games that have been extracted from Mega Database 2012[9] in ChessBase format. The related user interface ChessBase Reader offers the possibility to select the desired games and convert them into the standard PGN format. As a result, we obtain a set of files each containing several games in PGN format like seen in figure 8. In total, we analyze 39 files with 68,606 games. In the next step those files have to be read and converted into data sets with comments that can be used within the classification problem. For this purpose, the natural language toolkit NLTK is used.

NLTK[10] is a python library offering various technique for natural language processing (NLP). It can be used to extract information from web files in html or any other text file format. Besides, it offers access to big corpora and other lexical resources. The NLP process and its corresponding code in python using NLTK is shown in figure 10. Note that the pictured steps of tokenization and normalization are not considered in this section, but in section 4.3.
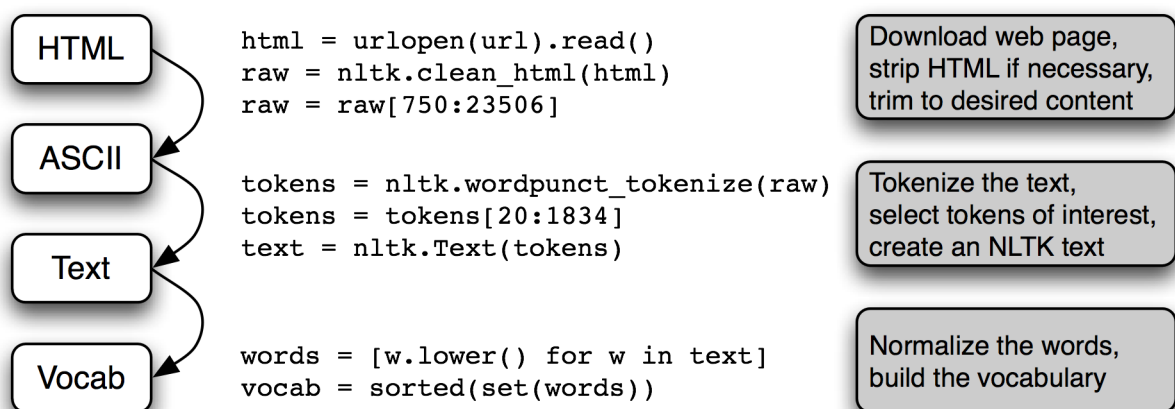

Figure 10: NLP pipeline cf.[BKL09, section 3.1]

Applied to the chess annotation problem, the raw text of the PGN files can be extracted and decoded by the following two commands

```
raw = open(file, 'rb').read()
string = raw.decode('iso-8859-1')
```

where the variable *file* contains both the relative path and the filename. The process is repeated for each filename saved in a list. An ISO 8859-1-decoding is used instead of an ASCII-decoding to detect the Western European letters used in comments and player names correctly.

However, we are not interested in the complete raw text of the PGN files, but only in the comments. As we have already seen in table 4, there are different comments in a PGN file. Since we want to be able to use supervised learning approaches, we need to know the correct class of a comment in the file. Therefore, the comments which are from importance are those connected to a traditional chess symbol or a NAG. To filter out such comments, we use regular expressions and distinguish between three cases:

- NAGs immediately followed by a comment:

  \\$(?P<class>[0-9]+)\s*\{(?P<comment>[^{}]*)\}

- NAGs followed by another NAG and thereafter a comment:

  \\$(?P<class>[0-9]+)\s*\\$[0-9]+\s*\{(?P<comment>[^{}]*)\}

- Standard symbols for move annotations (i.e. !, ?, !!, ??, !?, ?!) immediately followed by a comment:

  (?P<class>[!\?]{1,2})\s*\{(?P<comment>[^{}]*)\}

The match results are saved as tuples of the class (NAGs without dollar sign, symbols unchanged) and the comment. The final class is set depending on the rules of the considered classification problem by using a python dictionary, e.g. for the binary move annotation problem all the classes $1, 3, 5, !, !!, !?$ are mapped to the final class 1.

So far, we ensured the collected data to be complete (class is known), in a specific format (PGN, processed to tuple) and available in sufficient quantity. Before proceeding with the next step, we will perform some basic analysis on the extracted data to estimate the quantity of comments per class. This includes a comparison of the total count of all symbols and NAG types for every of the three tasks we specified in section 4.1.3. If the counts would differ a lot, different weights should be assigned to the instances to avoid imbalances and thus difficulties in classification. The data shown in table 6 has indeed some noticeable imbalances; instances with positive move annotations are more common than negative ones as well as surprisingly an advantage of white is more common than an advantage of black in the position annotations. However, these imbalances are in an uncritical range, which probably requires no weighting of instances.

Final class	Symbol & NAGs	Count	Percentage
1	all symbols, $1 − $9	224,619	44.04%
2	$10 − $135	268,060	52.57%
Total		492,679	96.61%
—	$136 − $255	17,313	3.39%

(a) all annotations

Final class	Symbol & NAGs	Count	Percentage
1	!!, $3	3,226	1.44%
2	!, $1	105,789	47.10%
3	!?, $5	50,636	22.54%
4	?!, $6	29,838	13.28%
5	?, $2	28,660	12.76%
6	??, $4	4,849	2.16%
Total		222,998	99.28%
—	$7 − $9	1,621	0.72%

(b) move annotations

Final class	NAGs	Count	Percentage
1	$18	23,500	8.77%
2	$16	42,975	16.03%
3	$14	53,599	20.00%
4	$10 − $13	58,774	21.93%
5	$15	17,841	6.66%
6	$17	16,092	6.00%
7	$19	11,559	4.31%
Total		224,340	83.69%
—	$20 − $135	43,720	16.31%

(c) position annotations

Table 6: Class distributions

The last criterion quality will be discussed in section 4.3. Possible problems are that the comment is in a different language than English or that the comment is too short to make an informed classification decision.

Apart from the statistics on the file data to be further processed, some information about the discarded data are relevant for the usage of the results we obtain. By collecting the number of all comments not annotated with a NAG or standard symbol yet we obtain the potential of improvement regarding the comments. As shown in table 7, only 45.39% of the comments are preceded by a NAG or symbol. For the remaining 54.61% of the comments, which are still more than half a million, the comment could be completed with an appropriate NAG or symbol. The other way around, this approach delivers a ratio of 25.06% NAGs and standard symbols that are followed by a comment. In contrast to the first case, adding a comment is not useful or necessary, while adding a NAG is usually possible for most comments except those describing general game information like opening variants or summaries.

	Count	Percentage of processed data
Classes & Comments	509, 992	100.00%
Comments	1, 123, 538	45.39%
Classes	2, 035, 121	25.06%

Table 7: PGN file statistics

To get the values of table 7, the number of comments is estimated by counting all occurrences of opening braces ({), the number of NAGs by all occurrences of the dollar sign ($) and the number of standard symbols by the number of matches of the regular expression \d\+*\s*[!\?]+ (an arbitrary combination of the symbols ! and ?, preceded by the number of the move field square and optional a check(mate) symbol and whitespace. Due to this rudimental approach of counting, the expressions could also match false positives, if some of the symbols are used in a different sense. However, this number of false positives is small and therefore negligible.

4.3 NLTK Preprocessing

The output of the previous step is a set of tuples containing the comment and the class. Since a direct evaluation of the comments is only limited possible, there is a need to split the comments into substrings with an identified meaning by tokenization. NLTK offers the method "word_tokenize" adjusted to natural language text data in addition to the straightforward method of splitting the comment by the whitespaces. However, the use of "word_tokenize" for punctuation symbols is not appropriate for all cases. If move variants are presented in a comment or additional NAGs are used, "word_tokenize" may not separate them as desired. A separate handling of such tokens is possible with a regular expression tokenizer.

The RegexpTokenizer of NLTK has a regular expression as its only parameter and splits the text into tokens using this expression. Several regular expressions can be combined by the conjunction symbol |. The tokenizer then takes the first of the expressions the part of the text matches, wherefore in the case of ambiguities the most specific or important expressions must be at the beginning. The parts of the text not matching any case of the regular expression are discarded.

For the creation of a suitable regular expression tokenizer for comments in PGN files, the token definition according to PGN file specification was taken into account [Edw94, chapter 7] and some comments were examined manually for their structure. This leads to a regular expression based on the following cases:

- PGN non-standard codes   #[\w\d]2

- NAG non-standard codes   \$\d+

- Move-annotating symbols   [!\?]+

- Position-annotating symbols   [\-\+/=]+

- Remis symbol   1/2

- Abbreviations   (?:\w\.)+

- Multiple dots   \.+

- Words (including hyphens or apostrophes)   [\w\d\-\']+

- Remaining non-whitespace characters   \S

The first two cases match non-standardized, but nevertheless frequently used abbreviations for common chess phrases. For example, the non-standard PGN code #C4 and the non-standard NAG $142 mean "Better is..." and could indicate a suboptimal move of class 4 or 5. The next three cases match symbols annotating moves and positions of follow-up variants. The regular expressions for abbreviations and multiple dots should reduce the number of tokens and avoid to mix this kind of expressions with other dots. Most tokens match the regular expression for words, that does not only include words of natural language, but also numbers, chess moves and the remaining position expressions $1-0$ and $0-1$. Finally, with the last expression the remaining non-whitespace characters like punctuation symbols are fetched. Different configurations of the tokenizer will be evaluated in section 5.1.

The normalization step is already involved in the tokenization process. Before splitting the comment into its tokens, it is already converted in lowercase, because this conversion does not influence the tokenization process. Immediately after the tokenization follows the lemmatization of the words. In this case lemmatization is preferred to stemming, because the language of lemmatized tokens can be better detected than for stemmed tokens. Besides, lemmatized tokens are easier to read than stemmed tokens. Categorization is waived, but it is implicitly used in the word embedding models, which receive the tokenized data as input later on. The normalization of the output, i.e. the conversion of the NAGs and symbols to the classes of the problem, is moved to after feature extraction and selection, because until then all comments can be processed uniformly for all problems.
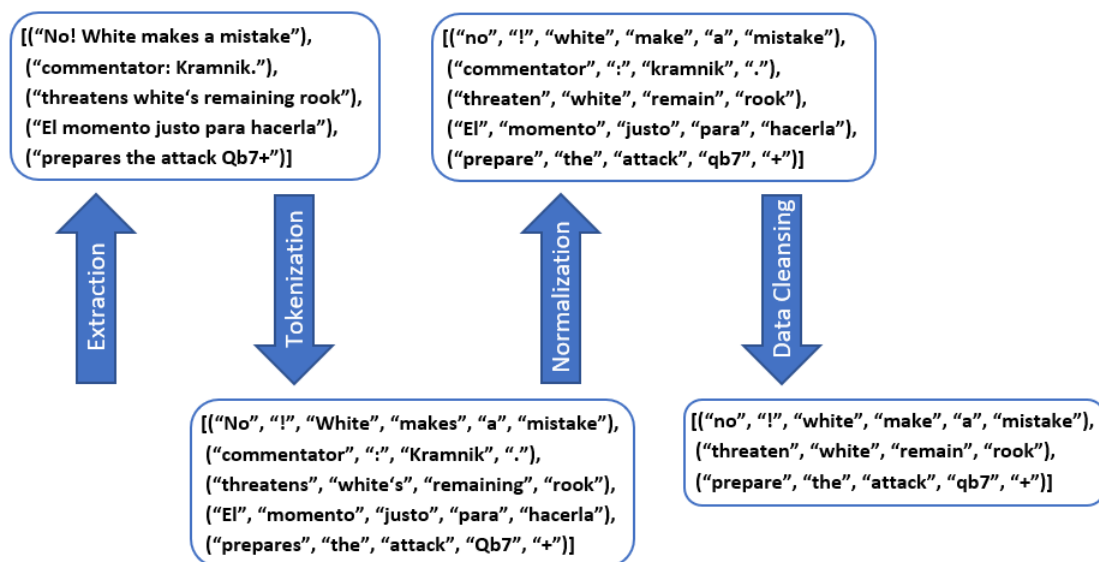

Figure 11: NLTK preprocessing

The tokenizer produces an ordered list of tokens as output. Before proceeding with the feature extraction and selection, we first perform the missing checks on the language and the length of the comment. Comments in different languages lead to different rules and are therefore difficult to process at the same time. Long comments often contain descriptions about openings, other variants or similar chess games and are therefore unsuitable. Comments that are too short, on the other hand, generally provide little information and are not very meaningful. Besides, comments with just one or two tokens often just contain the name of the commentator. In order to be able to make reliable classification decisions, we require at least three tokens that are part of the English vocabulary to be included in each further processed comment. Thus, a minimum length of three tokens is guaranteed as well. The maximum length should be set to 49 tokens.

Unfortunately, it is not possible to filter the games directly in the ChessBase Reader by the used comment language. For this reason, the additional language detection library langid is used to filter out comments already before the tokenization process if with 99% confidence the detected language is not English. Through this, over 70,000 comments will be discarded. The second language detection step starts after the tokenization and normalization. For each comment the tokens are iterated and checked if the lemmatized token is part of the English vocabulary provided by the NLTK corpus. If there are at least three such words, the comment will be forwarded to the feature extraction and selection process. Although false positives cannot be completely avoided with this approach, the proportion of long comments in non-English should be low due to the first filter. Not filterable are however some comments of the ChessBase database, which contain the same statements united in several different languages. For this reason, some tokens will be in a different language than English. The complete NLTK preprocess is shown in figure 11.

All in all, 70% of the comments are filtered due to the above-mentioned quality requirements. 156,970 out of 511,273 comments remain, which is still sufficient data to learn on. In order to keep the computing capacities and times within reasonable limits such that different configurations can also be tested, 4,000 instances are selected, including 2,000 move annotations and 2,000 position annotations. Two more datasets of the same size are created for short comments with a token length of 3 to 9 and for long comments with a token length of 10 to 49, without overlapping with the first dataset. These three datasets are passed as a list of tuples of tokenized comments and NAGs or symbols to the feature extraction and selection step. The order of the tokens remains unchanged by selecting a list as collection type.

4.4 Attribute Creation and Selection

In this step the text instances should be converted to a vector of numbers, whereby each number represents an attribute of the comments. All these vectors are strung together in a matrix, that is passed in an appropriate format to the analysis step. First different models for the generation of attributes are introduced, before the second step describes the transformation of the model data into an ARFF file.

4.4.1 Model Generation

In the following four different models are built in order to offer an appropriate attribute set for the analysis. The models have no attributes in common except of two; the number of tokens and the class attribute. For all models the attributes are all numeric except of the class attribute.

- Count-based Model

  The count-based model uses the CountVectorizer class of the machine learning library scikit-learn[11]. It converts the comments to a matrix of features, where each row represents a comment and each column a term of the vocabulary. The vocabulary is made up of all tokens, bigrams or trigrams that occur at least in five comments in the data set. The vectorizer counts the occurrences of each term in the vocabulary in each comment and stores the count in the corresponding cell of the matrix. As the comments are already lowercase and tokenized, these options do not have to be configured.

- TF-IDF-based Model

  The TF-IDF based model is similar to the count-based model, but uses the TfidfVectorizer class. The vectorizer is configured as in the first model, i.e. it builds a vocabulary of tokens, bigrams and trigrams with minimal one occurrence in five different comments. However, the matrix this time contains the relative values for the occurrences of the term. Two terms that have the same value in the count-based model now have different values if the number of documents in which the term appears varies. The more documents contain the term, the lower the value.

- Own Word2Vec-Model

  This model uses the word2vec module of the machine learning library gensim[12] to create a word embedding for chess annotations. A CBOW-word2vec-model is built and trained for all words in the chess comments. This time there is no need to include bigrams and trigrams in the model, because the context of a word is already taken into account by the window size of five. Similar to the first two models, all words with less than five occurrences are ignored and no part of the vocabulary. However, this time five is the limit for the total number of occurrences, i.e. a word only occurring in a single comment, but therein at least five times, will nevertheless be processed. As a result, the matrix contains a dense vector with 100 attributes for each word. To obtain the vector $\mathbf{v_c}$ for a comment c, the product of all vectors $\mathbf{v_w}$ and IDF-scores idf_w of words w in the comment that are also in the vocabulary are averaged:

  $$\mathbf{v_c} = \sum_{\substack{w \in c \\ w \in vocab}} \mathbf{v_w} \cdot idf_w$$

  Note that the weighted vectors have to be averaged and not summed up to handle different comment lengths. The term frequency is automatically included if the comment is treated as a list.

---

[11] `https://scikit-learn.org/stable/index.html`
[12] `https://radimrehurek.com/gensim/models/word2vec.html`

- Pretrained Word2Vec-Model

  This model is as well based on word embeddings, but instead of creating a new model, an existing model is imported. It learned word vectors for all terms appearing in a huge Google News data set. It offers 300 attributes for around 3,000,000 words. Terms that are not part of the vocabulary are usually nevertheless matched by a pattern and a corresponding vector. Like in the previous case, the output matrix contains the vector of the comments in the data set, where each vector is calculated as the average of the weighted word vectors.

The first two models offer a large number of attributes. Even if this number is already limited by the minimal number of five occurrences in the whole data set, an upper limit of 2,000 attributes is defined to limit the computing time of the algorithms. Both for the count-based and the TF-IDF-based model, the attributes are therefore ordered by their total term frequency and the top 2,000 remain in the model. For the short comment data set there is even no attribute selection required, because the vocabulary contains only 1,631 attributes. At this point it should be noted that usually it is not recommended to apply attribute selection to all data, but only to the training data, because otherwise the test data has been seen by the attribute selection process and the accuracies are too optimistic. Due to the fact, that for the attribute selection process the class attribute is not considered, this falsification should be minimal, if any.

4.4.2 Conversion to ARFF

Weka[13], short for Waikato Environment for Knowledge Analysis, is an open source software offering a collection of machine learning algorithms for data mining tasks that will be addressed to in section 4.5. The algorithms are applied to data represented in Attribute-Relation File Format (ARFF)[14]. Weka offers standard ARFF files to experiment with and to get to know the functionality of the machine learning methods. As well own ARFF files can be imported and used. For this purpose, an ASCII text file needs to be structured as shown in figure 12. It shows a possible output ARFF file for the two-class move annotations problem with two comment-value-pairs ("a brilliant counterattack of white", 1) and ("a big mistake of black", 2). The file consists of two blocks, the header information and the data information. Before the header information there might be comment lines with information about the author and version or further descriptions.

```
@RELATION  comment−move−1
@ATTRIBUTE count("brilliant") NUMERIC
@ATTRIBUTE tfidf("mistake") REAL
@ATTRIBUTE class {1,2}
@DATA
1, 0.0, 1
0, 0.06, 2
```
(a) complete

```
@RELATION  comment−move−1
@ATTRIBUTE count("brilliant") NUMERIC
@ATTRIBUTE tfidf("mistake") REAL
@ATTRIBUTE class {1,2}
@DATA
{0 1, 2 1}
{1 0.06, 2 2}
```
(b) sparse

Figure 12: Sample arff files

The first block of the header information contains the keyword "@RELATION" and an arbitrarily name for the relation in the first line. After that for each attribute the relation contains a triple of the keyword "@ATTRIBUTE", a unique name of the attribute and the data type of the attribute. The data type can be numeric (integer, real), string, date or nominal. For the first three data types, only the type needs to be indicated, whereas nominal attributes require a list of all possible values comma-separated in braces. The output value of an instance is an attribute as well and need to be specified, conventionally as the last attribute. In classification problems the class attribute has a fixed number of values and is represented as nominal in the form of a set, in regression problems it is numeric.

The second block begins with the keyword "@DATA" in the first line. After that for each instance the values of the attributes are listed comma-separated, in the same order as they were declared before, see figure 12a. Missing values are indicated by a "?". Data sets can consist for the most part of zero values, in particular those with attributes used in Information Retrieval like TF or TF-IDF. In order to reduce the creation time and the size of the file, in sparse ARFF files (see figure 12b) numeric values are zero by default and can be omitted. However, now the instances can consist of a different number of values. For this reason, each instance is represented as a comma-separated list of pairs, surrounded by braces. The first number of a pair is the attribute id (starting from zero), the second one the value. Missing values are not equal to zero and need to be indicated by a "?" further on.

---

[13] https://www.cs.waikato.ac.nz/ml/weka/
[14] https://www.cs.waikato.ac.nz/ml/weka/arff.html

For the matrix output of the count and TF-IDF based model, choosing the sparse representation form significantly reduces the size of the ARFF file. Since it only requires a small additional effort for the two Word2Vec models, a uniform transformation into sparse ARFF files is executed. Such a file is thus generated for each combination of the five problems, the four models and the three data sets, so that a total of 60 files are passed on for analysis. Note that now the NAGs and symbols are converted to the corresponding class depending on the problem.

4.5 Classification Algorithms

For the analysis of the data sets in the generated ARFF files the already mentioned software Weka will be used. It offers many tools to prepare, process and evaluate data by different machine learning techniques. For the purposes of this work mainly the configurations of the "Classify" tab in the Weka explorer are used as shown in figure 13.
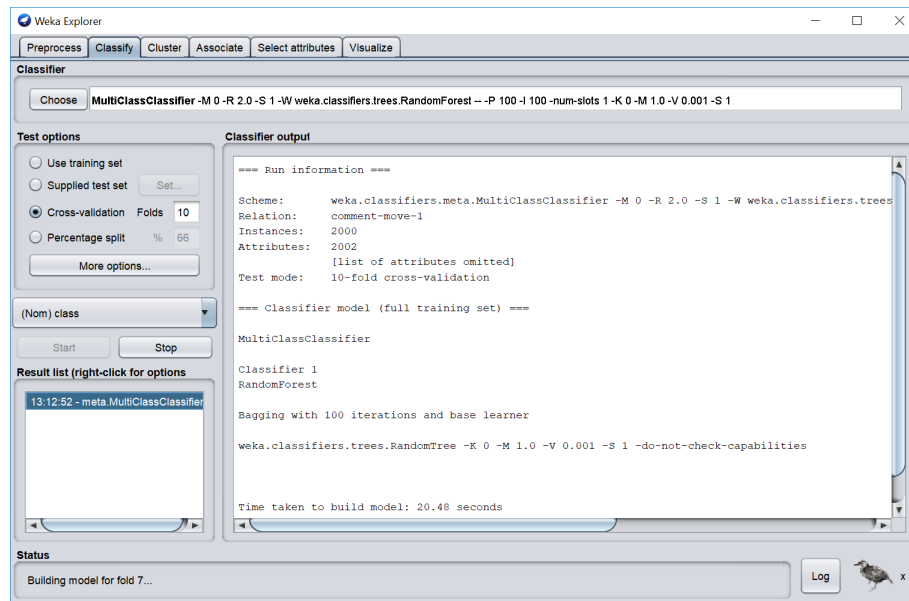
Figure 13: Classification in Weka

Before the classification can be started, the data record must first be loaded from the ARFF file in the "Preprocess" tab. Finally, a suitable classifier for the problem can be selected via the "Choose" button and applied via "Start". The following classifiers - among others based on the algorithms described in sections 2.3 and 2.3 - are used for the analysis:

- NaiveBayes (NB)

  The Naïve Bayes classifier uses a simple probabilistic model to predict the class of an instance. Since it calculates a probability value for each class, the classifier is also capable to handle multiclass problems. The classifier provides rather poor evaluation values, but is very fast and serves as a basic assessment of the problem and the models.

- RandomForest (RF)

  Random forests belong to the decision tree algorithms and use the ensemble learning method of bagging. The name random forest comes from the fact that a decision tree is created based on random subsets of the attributes. Thus, a forest of decision trees is formed, whereby the final prediction is made by equal weighting of all predictions of the trees. The training of the trees is fast and can be parallelized easily. Furthermore, the independence of the trained trees provides a robust model and good accuracy values. Random forests with 100 trees are used for testing.
  For the following algorithms of the multiclass, ordinal class and nested dichotomy classifier, the random forest classifier will be used to solve the binary classification problems into which the multiclass problem is broken down.

- MultiClassClassifier, method 0 (MCC0)

  The multiclass classifier of Weka offers four different methods to break the multiclass classification problem down to multiple binary classification problems. The standard method with the ID 0 is the one-against-all approach.

- MultiClassClassifier, method 1 (MCC1)

The algorithms using error-correcting output codes are as well included in the multiclass classifier. The method with ID 1 provides the ensemble learning with a random error-correcting output code.

- MultiClassClassifier, method 2 (MCC2)

  The method with ID 2 provides the ensemble learning with the exhaustive error-correcting output code. The code is therefore longer than the random code and needs more computing time.

- MultiClassClassifier, method 3 (MCC3)

  The last method with ID 3 is the one-against-one approach.

- MultiClassClassifier, method 3 with pairwise coupling (MCC3P)

  For the one-against-one approach, Weka additional offers the possibility to execute this algorithm with pairwise coupling.

- OrdinalClassClassifier (OCC)

  The ordinal class classifier can be installed separately and is based on the already introduced method of Frank and Hall, which divides a multiclass classification problem into binary classification problems considering the ordinal order of the class attribute. Of the classifiers tested, it is the only one to include this order in the classification process.

- ClassBalancedND (NDC)

  As for the ordinal class classifier, a package with nested dichotomy classifiers can be included as well. The class balanced nested dichotomy ensures a class equilibrium between the two subsets for each dichotomy.

- DataNearBalancedND (NDD)

  The other variant is the data near balanced nested dichotomy, where a data equilibrium between the two subsets for each dichotomy is ensured.

- ZeroR

  Finally, the zero-rule classifier is executed to provide a baseline that serves as comparison value for kappa statistics. The accuracy equals the percentage of the most frequent class and is therefore the same for all four models if the same data set is used.

Independent of the classifier, due to the higher number of attributes, the time required for the classification of the pre-trained word embedding model is about twice as long as for the own trained word embedding model and even five times as long for the count-based and TF-IDF-based models. Another factor for the computing time is the number of classes. However, additional classes also increase the required computing capacity.

For the three multiclass classification problems, i.e. the move annotation problem with six classes and both position annotation problems, all classifiers can be applied to and provide different classification models and results. The two binary classification problems do not require any additional decomposition of the problem, so all multiclass classifiers and the ordinal class classifier output the same result as the underlying binary classification algorithm, in this case random forest. Thus, only Naïve Bayes, random forest and nested dichotomy remain for the binary classification problems as methods to be compared.

The output of a classifier is the computed classification model, the correct and predicted classes for all instances if desired, and a set of evaluation metrics that will be discussed in the following chapter.

4.6 Evaluation Methods

The examined data sets have a size of 4,000 instances for the move vs. position annotation problem and a size of 2,000 instances for the other problems. The validation of the classification models is performed by 10-fold-cross-validation, i.e. 400 or 200 instances are hold out in each of the ten runs. All evaluation measurements will be averaged over the ten folds of the cross-validation.

The main evaluation metric will be the accuracy. The higher the accuracy of a classification model, the better the classifier and the underlying feature model are evaluated. In order to be able to compare the accuracy across the different problems, the kappa statistic is used. The baseline determined by the zero-rule classifier is used as reference value

for the calculation.

In the configurations with the best accuracy, the confusion matrix is also analyzed. For all defined problems, the general occurrence probability of a class can influence the frequency of different misclassification types, i.e. the values M_{ij} of the cells in the confusion matrix. However, this preference for frequent classes is not punished, since in all problems it is the same if n instances of class i are classified correctly or n instances of class j. Besides, it does not matter if a class i is misclassified as class j or the misclassification happens the other way around. Under these circumstances, evaluation methods such as precision, recall and ROC are unsuitable.

However, the two above-mentioned conditions can be represented by a cost-sensitive evaluation matrix in which the entries of the main diagonal have the same cost, namely zero, and which is symmetrical, i.e. $c_{ij} = c_{ji}$. By this definition, the additional analysis by cost matrices for the two binary classification problems move vs. position annotations and good vs. poor moves is superfluous, the costs result automatically from the product of the misclassified instances with the only cost value different from zero. For the remaining three problems, both the total costs based on absolute and on squared cost vectors as presented in section 2.5 are considered. An inclusion of these cost matrices in the classification process via cost-sensitive learning is not carried out.

5 Evaluation of Results

In this chapter the results of the whole text mining process are presented and evaluated. The chapter starts with the test results of different tokenizer configurations. Then, based on the best tokenizer configuration, evaluations of the tokens in the chess annotation data are presented. Before applying the classifiers on the data sets, the feature models are evaluated separately by checking the most informative attributes and by applying basic queries on the word embeddings. Afterwards, the achieved accuracies for all combinations of problem, data set, model and classifier are compared and analyzed. Finally, the misclassifications are viewed using examples, confusion matrices and different cost weights.

5.1 Tokenizer Evaluation

For the evaluation of the Tokenizer configurations, a small test set with a total of slightly more than 2,000 instances is used, which, however, includes all tokens, bigrams and trigrams that occur at least two times in the data set. Thus, the model contains nearly 13,000 attributes and is based on the counts of the terms. The Naïve Bayes classifier was used for the classification, which can consider all attributes without much additional calculation effort. As a further comparison value, the one-against-all classifier (MCC0) with random forest was used. In addition to the move annotation problems (too high computational effort), an attribute selection based on the gain ratio has been applied beforehand. In table 8 the different tokenizer configurations and the percentage variance of their achieved accuracy in comparison to the one of the standard configuration are listed. Hence, improvements compared to the standard configuration are indicated by a "+".

Tokenizer	Total	Move-1	Move-2	Position-1	Position-2
complete	86.0849%	76.6486%	54.3243%	43.4959%	21.9512%
	86.5566%	75.6757%	55.3514%	51.2195%	19.9187%
no PGN non-standard codes	±0%	−0.07%	±0%	±0%	±0%
	±0%	+0.50%	−0.68%	±0%	±0%
no NAG non-standard codes	±0%	±0%	±0%	±0%	±0%
	+0.11%	+0.29%	−0.68%	±0%	±0%
no move-annotating symbols	±0%	±0%	−0.10%	−0.93%	±0%
	±0%	+0.29%	−0.59%	±0%	±0%
no position-annotating symbols	±0%	−0.07%	−0.10%	±0%	±0%
	+0.05%	−0.57%	−0.20%	±0%	±0%
no remis symbol	±0%	±0%	±0%	±0%	±0%
	±0%	±0%	±0%	±0%	±0%
no abbreviations	+0.02%	−0.14%	−0.20%	+4.34%	−1.20%
	−0.09%	−0.57%	−0.10%	−0.42%	−17.33%
no multiple dots	−0.16%	+0.07%	+0.30%	±0%	±0%
	−1.42%	+0.29%	−0.59%	±0%	±0%
no merged words	−0.21%	−1.57%	+1.09%	+2.49%	−1.20%
	+0.39%	−0.66%	−0.30%	+1.15%	−5.24%
no words	−0.51%	−4.79%	−10.92%	+2.79%	+8.23%
	−1.01%	−2.99%	−1.56%	−12.71%	+23.61%
no remaining non-whitespace characters	−0.27%	−0.56%	−0.50%	−3.74%	−11.11%
	+0.33%	−0.21%	−1.07%	±0%	±0%

Table 8: Accuracies for different RegexTokenizer

The removal of the regular expressions for PGN or NAG non-standard codes as well as the removal of those for move- or position-annotating symbols had a slight deterioration of the accuracy as a consequence while the removal of the remis symbol as a token had no effect at all. The removal of the regular expression for abbreviations which also matches move counts, had a negative influence on the accuracy in eight of ten cases. For the regular expression for multiple dots, there were three improvements and three deteriorations, with the deteriorations weighing heavier. In the last three regular expressions for merged words, words and remaining non-whitespace characters, more than half of the accuracy values deteriorated. There are noticeable improvements for the position annotation problem with seven classes by the exclusion of words, i.e. the positions could be determined better by the counts of the letters than by the counts of the words. At the same time, however, this adaptation also led to significant deteriorations in the other problems. Overall, for all comparison configurations, the sum of the degradations is at least as large as the sum of the improvements, which is why the initial configuration of the tokenizer is retained.

The tokenizer tests were executed before the additional chess files of the Mega Database 2012 were available. For this reason, the ChessBase data sets were not included, which results in a different class distribution, especially in the move vs. position annotation problem. The high values of over 85% accuracy in the first column can be explained by the fact that in this first test data set about seven eighths of the comments were move annotations. Thus, the baseline is even above the best accuracy. In order to have absolute certainty that the best tokenizer configuration in the tests will also perform best for the new datasets, the datasets would have to be recreated and re-tested.

5.2 Data Set Evaluation

After the determination of the tokenizer, the used comment data can be processed uniformly into tokens. For a better understanding of the data, some statistics regarding those tokens are created. In the evaluations in this section, the tokens were converted to lowercase, but not lemmatized. Filtering by language or minimum comment length was also not performed.

In the first step, the number of comments is analyzed depending on their length in tokens. The statistics are shown in table 9 and figure 14. It can be seen that about half of the comments have a maximum token count of nine. 99 of these comments do not even contain a single token, since they consist only of whitespace characters. Of the remaining comments, only a small portion has a length of at least 50 tokens. Due to these circumstances, a length of three to nine tokens was defined for short comments and a length of ten to 49 tokens for long comments.

Length	Count
0-4	92,519
5-9	154,412
10-14	105,778
15-19	77,983
20-29	52,802
30-49	21,480
50-99	5,653
100-199	586
200+	60
all	511,273

Table 9: Comment length in tokens
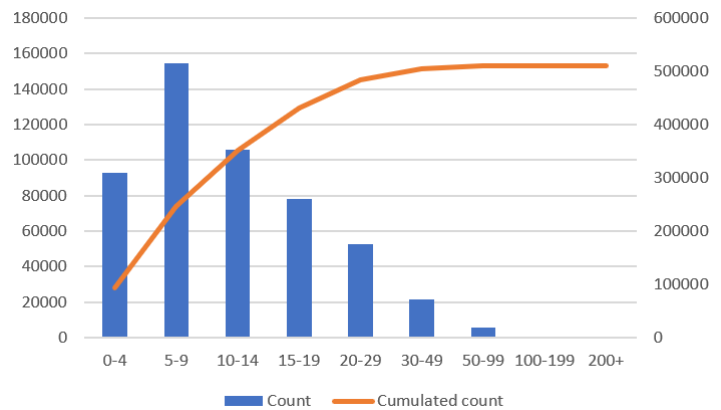
Figure 14: Count and cumulative count of comment length

Likewise, statistics about the tokens themselves can be created. An evaluation of how many tokens, bigrams and trigrams exist with a certain number of minimum occurrences is relevant because it determines the maximum number of attributes for the count- and TF-IDF-based model. This number is given for different numbers of at minimum occurrences in table 10. It should be noted that this number will be smaller if a lemmatization is applied beforehand.

Min. occurrences	Count tokens	Count bigrams	Count trigrams
1,000	654	458	130
300	1,719	1,948	706
100	3,913	6,382	3,013
30	9,280	21,730	15,314
10	20,881	63,250	60,923
3	51,693	204,975	286,032
1	158,652	798,608	1,869,193

Table 10: Frequency of tokens, bigrams and trigrams

Due to the many possible combinations, there is a very large total number of trigrams, which, however, decreases rapidly when the threshold is increased. In the case of bigrams, this tendency is not so pronounced. Even for a threshold value of 300 there are more bigrams than single tokens. Altogether it can be stated that a threshold of three already reduces the number of possible attributes to about 20% and a threshold of ten even to 5% of the original number. From the top row we can conclude that there are only slightly more than 1,000 terms which occur in at least 0.2% of the comments. This indicates a strong variance in the comments.

Besides, additional analyses were performed to determine the most common words, bigrams and trigrams. The values can be seen in the tables 11, 12 and 13. It is not sure whether such frequently occurring terms are meaningful at all, i.e. whether they offer an information gain. If this is the case, however, they can be used very well as learning and decision criteria, because they can be applied to a large number of comments.

Token	Count
,	459,014
/	310,510
.	241,009
the	138,730
[	123,708
]	123,670
)	116,210
(	115,868
%	111,512
a	70,800

(a) all tokens

Token	Count
the	138,730
a	70,800
to	64,120
cal	60,054
is	56,347
black	42,819
white	41,679
in	39,794
and	38,725
of	36,566

(b) all English words

Token	Count
cal	60,054
black	42,819
white	41,679
move	22,431
die	18,428
position	16,599
schwarz	14,037
v	13,798
ist	12,849
game	12,178

(c) all English non-stopwords

Table 11: Most frequent tokens in comments

Among the most common tokens, bigrams and trigrams are mainly punctuation marks and combinations of those that do not allow intuitive conclusions to be made about certain annotation types. The only exceptions are the bigrams "1/2 -" and "- 1/2" respectively the trigram "1/2 - 1/2", which could indicate a balanced position. If the list is limited to English words, there are numerous stopwords and combinations of these, but also the two tokens "white" and "black" and the phrases "this move", "the game" or "week in chess". If you consider only non-stopwords for the tokens, then with the words "move" and "position" two further interesting tokens are among the top ten, which should be relevant at least for the classification according to move and position annotations.

Bigram	Count
[%	111,424
% cal	60,054
% csl	51,190
/ \	26,923
1/2 -	19,115
, a	18,829
- 1/2	18,605
] [	18,169
/ cbm	15,877
) /	14,837

(a) all bigrams

Bigram	Count
of the	10,650
on the	10,381
in the	9,779
the game	7,464
this move	6,667
white has	5,829
black has	5,711
for the	5,650
this is	5,090
to the	4,613

(b) all English bigrams

Table 12: Most frequent bigrams in comments

Trigram	Count
[% cal	60,054
[% csl	51,190
1/2 - 1/2	18,603
] [%	17,863
, a /	13,696
, v /	9,252
, m /	8,728
, j /	8,717
, s /	6,229
, r /	5,096

(a) all trigrams

Trigram	Count
week in chess	2,322
the week in	2,319
this is the	1,651
in the game	1,566
this move is	1,498
in order to	1,217
white has a	1,203
this is a	1,117
of the game	1,018
in view of	968

(b) all English trigrams

Table 13: Most frequent trigrams in comments

5.3 Model Evaluation

Before the classification is performed, some analyses can already be done on the underlying models. In the two models, where each attribute correlates with a certain term, namely the count-based model and the TF-IDF-based model, meaningful attributes and thus terms that are relevant for classification decisions can be determined by calculating values such as the gain ratio. Table 14 shows the ten best attributes by gain ratio for the move vs. position annotation problem, the move annotation problem with two classes and the position annotation problem with three classes. In the following, no distinction is made between the count-based model and the TF-IDF-based model, since the values are very similar in both models.

The best attributes for the move annotation problem all indicate a poor move and are intuitive or at least comprehensible like count("be too"). The position annotation problem also includes three intuitively useful attributes: count(white

loose), count(balance) and count(equalize). For the rest, it is not obvious at first glance to what extent the classes are differentiated by the attribute. Even less obvious are the statements of the best rated attributes (except the last one) of the move vs. position annotation problem. Though, here the gain ratio values are generally lower.

Feature	Gain
count(the week in)	0.14348
count(the week)	0.14348
count(week)	0.14348
count(week in)	0.14348
count(week in chess)	0.14348
count(in chess)	0.14348
count(/ the week)	0.14348
count(/)	0.14107
count(ext)	0.13853
count(1/2)	0.13752

(a) Move vs. position

Feature	Gain
count(mistake)	0.21430
count(blunder)	0.20024
count(decisive mistake)	0.18957
count(decisive mistake .)	0.18122
count(be too)	0.17391
count(mistake .)	0.17252
count(mistake ,)	0.17119
count(the decisive mistake)	0.16829
count(blunder .)	0.16829
count(time trouble)	0.16177

(b) Move (2 classes)

Feature	Gain
count(white lose)	0.23008
count(the text)	0.21687
count(balance)	0.21345
count(equalize)	0.21345
count(repetition)	0.21345
count(win a pawn)	0.20842
count(play for a)	0.20842
count(queen .)	0.20842
count(ha no problem)	0.20776
count(no problem)	0.20776

(c) Position (3 classes)

Table 14: Most informative attributes in comments

The attributes shown in table 14, however, show partly strong dependencies to each other. If the attribute count("mistake") is already selected in the case of the move annotation problem, the attributes count("mistake .") and count("mistake ,") are redundant and no longer provide any additional information gain. For this reason, the rule-based classifier J48 was used to create a classification model with rules for the three problems mentioned in order to investigate suitable attribute combinations. The learned rules are structured according to the scheme "(condition A) and (condition B) => prediction (true positive count, false positive count) and can be seen in figure 15, 16 and 17. Note that in each case the last rule has no condition and predicts the most frequent class.

```
(count(mistake) >= 1) => class=2 (65.0/2.0)
(number_of_tokens <= 5) and (count(.) <= 0) and (count(of) >= 1) => class=2 (17.0/2.0)
(count(but) >= 1) and (count(this) >= 1) and (count(of) <= 0) => class=2 (49.0/16.0)
(count(lose) >= 1) and (number_of_tokens <= 13) => class=2 (37.0/7.0)
(number_of_tokens <= 5) and (count(by) >= 1) => class=2 (17.0/1.0)
(count(miss) >= 1) and (count(miss_this) <= 0) => class=2 (30.0/4.0)
(count(wa) >= 1) and (number_of_tokens >= 10) and (count(_comma_) <= 0)
        => class=2 (48.0/15.0)
(count(blunder) >= 1) => class=2 (21.0/0.0)
(number_of_tokens <= 6) and (count(too) >= 1) => class=2 (9.0/0.0)
(count(allow) >= 1) => class=2 (35.0/10.0)

 => class=1 (1672.0/413.0)
```

Figure 15: Rules for the 2-class move annotation problem

```
(count(0-1) >= 1) and (number_of_tokens >= 18) => class=3 (58.0/26.0)
(count(/) <= 0) and (count(win) >= 1) and (count(white) <= 0) and
        (number_of_tokens >= 9) => class=3 (34.0/11.0)

(count(1/2) >= 1) and (count(ext) <= 0) => class=2 (136.0/65.0)

 => class=1 (1772.0/733.0)
```

Figure 16: Rules for the 3-class position annotation problem

As in the tables, the rules for the move and position annotation problem are largely comprehensible, whether intuitive or not. In addition, the comment length seems to be relevant as it is used as part of rules in both models. The rules of the move vs. position annotation problem, on the other hand, are less comprehensible and it is questionable whether they could also be successfully applied in other data sets.

```
(count(/) >= 1) and (count(/_\) <= 0) => class=2 (699.0/49.0)
(count(and) >= 1) and (number_of_tokens <= 19) and (count(white) >= 1)
        => class=2 (153.0/35.0)
(count(to) <= 0) and (count(and) >= 1) and (count(ha) >= 1) => class=2 (71.0/12.0)
(count(this) <= 0) and (count(to) <= 0) and (count(.) >= 1) and (count(the) <= 0) and
        (count(with) >= 1) => class=2 (97.0/16.0)
(count(move) <= 0) and (count(be) >= 1) and (count(the) <= 1) and (count(.) >= 1) and
        (count(this) <= 0) => class=2 (504.0/194.0)
(count(move) <= 0) and (count(_comma_) >= 2) and (count(to) <= 0)
        => class=2 (117.0/45.0)
(count(move) <= 0) and (count(ha) >= 1) and (count(to) <= 0) => class=2 (85.0/32.0)

 => class=1 (2274.0/657.0)
```

Figure 17: Rules for the move vs. position annotation problem

For the word embeddings an evaluation like for the count- and TF-IDF-based models is not useful, because the concrete statements by the attributes of the word embeddings are unknown. Instead, the learned vectors of the vocabulary can be compared directly with each other to check whether the semantics of the comments have been captured meaningfully. For this purpose, the most similar words to the (chess) terms "check" and "blunder" with respect to the cosine similarity are compared for the word embedding learned on the chess comments (see table 15) and the word embedding learned on Google News (see table 16).

Word	Similarity
mate	0.5111
pin	0.5075
capture	0.4505
fork	0.4442
move	0.4145
perpetual	0.4108
guard	0.3812
penetration	0.3736
threat	0.3591
attack	0.3498

(a) Most similar to "check"

Word	Similarity
checking	0.7029
checks	0.6817
checked	0.6749
Checking	0.5686
recheck	0.5329
doublecheck	0.5239
Check	0.5161
VGGEN_often	0.5077
Charge_Issuing_worthless	0.4998
ABCActionNews.com	0.4760

(a) Most similar to "check"

Word	Similarity
mistake	0.8191
error	0.7349
oversight	0.7288
miscalculation	0.6497
inaccuracy	0.6169
touch	0.5317
time-trouble	0.5077
blow	0.4959
shoot	0.4909
concession	0.4849

(b) Most similar to "blunder"

Word	Similarity
gaffe	0.7873
blunders	0.7614
mistake	0.7505
faux_pas	0.6696
miscue	0.6602
misjudgment	0.6521
howler	0.6513
bungle	0.6400
tactical_blunder	0.6219
miscalculation	0.6167

(b) Most similar to "blunder"

Table 15: Cosine similarities in chess word2vec-model   **Table 16:** Cosine similarities in Google News word2vec-model

The self-trained word embedding gives as expected a list of related chess terms, while the pretrained word embedding calculates more general terms, which should be irrelevant for the chess comments. For the two words "mistake" and "miscalculation", which occur in both tables, the self-trained word embedding has higher similarity values to the given word "blunder" than the pretrained word embedding. Furthermore, the self-trained word embedding achieves a similarity of 97.78% for the two words "white" and "black", whereas in the pretrained word embedding it is only 80.92%. In principle, a similarity of these terms is desired, but a similarity that is too high could make it difficult to distinguish the two words from each other and thus complicate decisions whether a position is in favor of white or in favor of black.

5.4 Classification Results

In this thesis five different problems, four different models, ten different classifiers and three different data sets were presented. For all meaningful combinations of these variables, a classification was performed and the accuracy calculated. This section summarizes these results. All accuracies of the first data set are displayed and analyzed in tabular and graphical form for each problem. The best values are then compared with those of the two remaining data sets.

In all tables the classifiers have the abbreviations as described in section 4.5, furthermore the word embeddings are abbreviated with w2v-C for the model learned on the chess comments and w2v-GN for the model learned on Google News. The highest value of each table is marked in bold. In the case of the multiclass classification problems, the accuracies of the two error-correcting output code classifiers (MCC1 & MCC2) were not determined due to excessive computation time. Therefore, these two classifiers, as well as the Naïve Bayes classifier, whose values fell sharply compared to the rest, are not shown in the diagrams.

	count	TF-IDF	w2v-C	w2v-GN
ND	0.7785	0.7725	0.7345	0.7258
RF	**0.7825**	0.7728	0.7368	0.7180
NB	0.6963	0.7030	0.6630	0.6265
Baseline	0.5000			

Table 17: Accuracies for move vs. position annotation problem
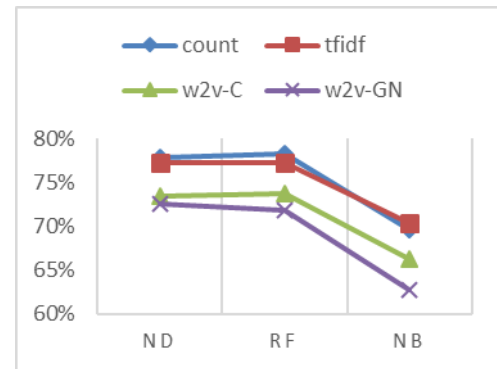
Figure 18: Comparison of accuracies for move vs. position annotation problem

On the move vs. position annotations only the classifiers random forest, nested dichotomy and Naïve Bayes are applied because it is a binary classification problem. As seen in table 17 and figure 18, a maximum accuracy of 78.25% is achieved with the count-based model and the random forest classifier. The same classifiers are also applied to the second binary classification problem of move annotations (see table 18 and figure 19). This time, the classifier nested dichotomy reaches the best accuracy of 76.90%, again using the count-based model.

	count	TF-IDF	w2v-C	w2v-GN
ND	**0.7690**	0.7575	0.6735	0.7305
RF	0.7590	0.7610	0.6670	0.7335
NB	0.6355	0.5990	0.4555	0.6895
Baseline	0.6580			

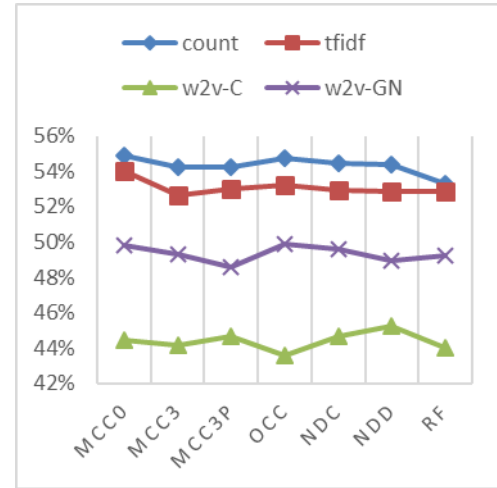**Table 18:** Accuracies for move annotation problem (2 classes)
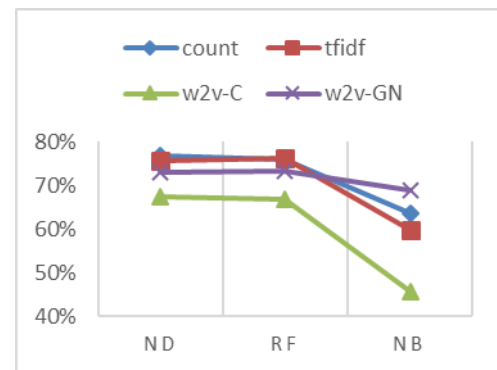
Figure 19: Comparison of accuracies for move annotation problem (2 classes)

All ten classifiers are applied to the three multiclass classification problems. Table 19 shows the results for the move annotation problem with six classes and a best value of 54.90% for the count-based model and the one-against-all classifier. It can be seen in figure 20 that for all classifiers the count-based model achieves the highest accuracies, closely followed by the TF-IDF model. The accuracies of the Google News word embeddings are about five percentage points below these models and the chess word embeddings even about ten percentage points below. This ranking of the models had already been indicated by the first two problems.

	count	TF-IDF	w2v-C	w2v-GN
MCC0	**0.5490**	0.5405	0.4445	0.4985
MCC3	0.5420	0.5265	0.4415	0.4930
MCC3P	0.5425	0.5300	0.4470	0.4855
OCC	0.5475	0.5320	0.4360	0.4990
NDC	0.5445	0.5295	0.4465	0.4960
NDD	0.5435	0.5285	0.4525	0.4895
RF	0.5330	0.5285	0.4400	0.4920
NB	0.4025	0.1750	0.1945	0.2435
MCC1			0.4085	0.4790
MCC2			0.4550	0.4975
Baseline		0.4430		

Table 19: Accuracies for move annotation problem (6 classes)

Figure 20: Comparison of accuracies for move annotation problem (6 classes)

As well for the position annotation problem with three classes the count-based model proves to be the best (see table 20 and figure 21). Again, the maximum value is achieved with another classifier, this time the one-against-one classifier with pairwise coupling. The accuracy value of 56.80% is only slightly above the baseline, which corresponds to a kappa statistic of less than three percent. Previously, maximum kappa values of 56.5% (move vs. position), 36.5% (move, 2 classes) and 19% (move, 6 classes) were reached. For the position annotation problem with seven classes the kappa value is again over 15%. These values are therefore very low overall, although this is partly due to class imbalance.

|	count	TF-IDF	w2v-C	w2v-GN
MCC0	0.5670	0.5570	0.5410	0.5645
MCC3	0.5620	0.5600	0.5410	0.5600
MCC3P	**0.5680**	0.5620	0.5455	0.5620
OCC	0.5570	0.5560	0.5395	0.5645
NDC	0.5595	0.5520	0.5385	0.5615
NDD	0.5595	0.5560	0.5500	0.5615
RF	0.5600	0.5645	0.5445	0.5640
NB	0.4100	0.3545	0.2755	0.3080
MCC1			0.5300	0.5460
MCC2			0.5435	0.5660
Baseline		0.5550		

Table 20: Accuracies for position annotation problem (3 classes)
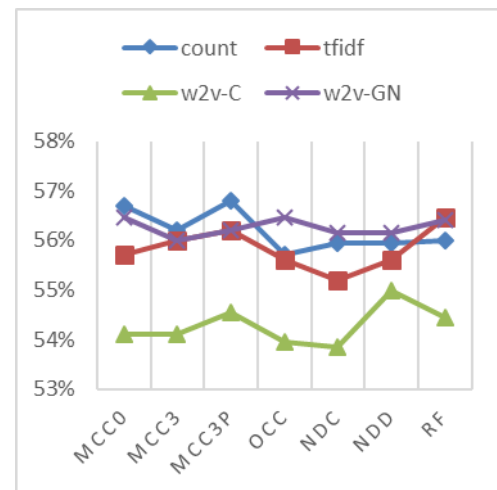
Figure 21: Comparison of accuracies for position annotation problem (3 classes)

The lowest accuracies by far are achieved in the position annotation problem with seven classes (see table 21 and figure 22). The maximum value of 36,20% is reached in the count-based model by the one-against-one classifier with pairwise coupling as well as by the class balanced nested dichotomy classifier. Again the diagram shows the clear ranking of the models: count-based-model followed by TF-IDF-based model, Google News word embedding and chess word embedding.

	count	TF-IDF	w2v-C	w2v-GN
MCC0	0.3520	0.3490	0.2940	0.3100
MCC3	0.3565	0.3480	0.2905	0.3250
MCC3P	**0.3620**	0.3525	0.2905	0.3360
OCC	0.3365	0.3420	0.2855	0.2980
NDC	**0.3620**	0.3340	0.2880	0.3160
NDD	0.3545	0.3420	0.2875	0.3205
RF	0.3425	0.3405	0.2790	0.2865
NB	0.2525	0.2190	0.1815	0.1930
MCC1			0.2620	0.2805
MCC2			0.2900	0.3295
Baseline		0.2465		
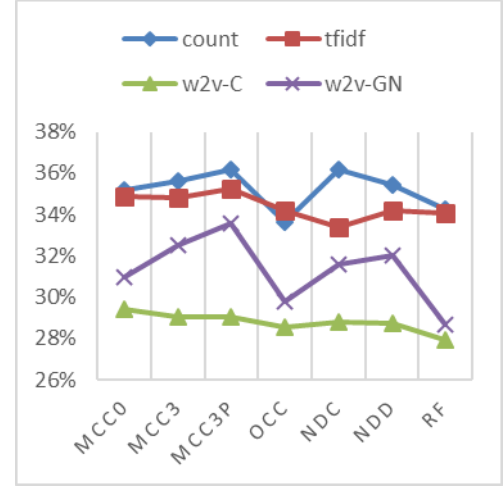
Table 21: Accuracies for position annotation problem (7 classes)

Figure 22: Comparison of accuracies for position annotation problem (7 classes)

Overall, the results of the five tables and figures show that the count-based model is the best, followed by the TF-IDF-based model and the chess word embedding is the worst. Among the classifiers, there was no clear trend which is best. These observations are also confirmed by the other two data sets. The count-based model has the highest average accuracy holding 13 of the 15 best accuracy values (see table 22). The other two best values are reached by the TF-IDF-based model, which is partially superior to the count-based model in the data set with long comments. The nested dichotomies classifiers achieve the best values overall, whereby the accuracies are only slightly above those of the other classifiers and therefore no conclusion can be drawn.

However, it is noticeable that the data set with the short comments achieves significantly better accuracies than the mixed-length data set. The only exception is the move vs. position annotation problem. The accuracies for the data set with long comments are on average slightly lower than for the mixed-length data set. These values suggest that either the short comments are generally more meaningful or that the condition that at least three English words must occur in the comment filters out qualitatively inferior short comments, while most qualitatively inferior long comments remain in the data set.

Data set	Position vs. Move	Move (2 classes)	Move (6 classes)	Position (3 Classes)	Position (7 Classes)
mixed	0.7825	0.7690	0.5490	0.5680	0.3620
short	0.7495	0.8134	0.5670	0.5985	0.4210
long	0.7813	0.7535	0.5275	0.5890	0.3535

(a) Accuracies

Data set	Position vs. Move	Move (2 classes)	Move (6 classes)	Position (3 Classes)	Position (7 Classes)
mixed	Count + RF	Count + ND	Count + MCC0	Count + MCC3P	Count + NDC/MCC3P
short	Count + ND	Count + ND	Count + NDD	Count + RF	Count + NDC
long	TF-IDF + ND	Count + ND	Count + MCC0	TF-IDF + NDD	Count + NDD

(b) Model + Classifier

Table 22: Comparison of the best achieved accuracies

The classification results obtained in the previous section have so far only been evaluated according to the accuracy value. In this section, the misclassifications are examined and it is checked whether the previously optimal results remain optimal even after weighting the misclassifications with costs. First, the confusion matrices are used to analyze which types of misclassifications frequently occur. The confusion matrices of the ordinal class classifier in table 23 are considered as examples; the matrices of other data sets and other classifications show similar patterns. The binary classification problems already result in the same values as with direct application of random forest.

predicted		
	1	2
1	1692	308
2	562	1438

(a) Move vs. position

predicted		
	1	2
1	1222	94
2	388	296

(b) Move (2 classes)

predicted			
	1	2	3
1	951	113	46
2	326	131	13
3	336	52	32

(c) Position (3 classes)

predicted					
	1	2	3	4	5	6
1	0	18	1	0	1	0
2	0	798	49	21	18	0
3	0	260	119	20	11	0
4	0	180	35	49	39	1
5	0	159	19	31	129	4
6	0	14	1	2	21	0

(d) Move (6 classes)

predicted						
	1	2	3	4	5	6	7
1	88	52	32	37	4	3	14
2	37	96	152	83	6	10	3
3	15	92	222	143	13	5	3
4	14	68	135	239	7	7	0
5	14	19	57	53	10	1	1
6	9	33	43	39	2	6	3
7	38	26	24	26	2	2	12

(e) Position (7 classes)

Table 23: Confusion matrices of ordinal class classifier

As desired, the confusion matrix of the move vs. position annotation problem has the highest values on the main diagonal, which is also determined by the equilibrium of the instances per class. With the move annotation problem with two classes it looks different; here class 2 (poor move) is rarer than class 1 and is predicted much less often. This even leads to the fact that class 1 is predicted more often than class 2 by the instances of class 2. The same happens with the other three problems: With the move annotation problem with six classes class 2 dominates, with the position annotation problem with three classes class 1 and with the position annotation problem with seven classes class 3 and 4. From this it can be concluded that the classifier for many comments does not find a sufficiently strong decision for any class and therefore chooses the most frequently occurring class.

However, a closer look at the misclassified instances revealed that some simple mistakes were made in addition to comments that were difficult to classify. Table 24 shows six examples in tokenized and lemmatized form that were misclassified by the random forest classifier in all four models. The comments have no particular complications and error-free class assignment is a prerequisite for using the classification model for non-test purposes. However, none of the classification models fulfils this requirement, so that further improvements in preprocessing, model building or classification are necessary.

Problem	Instance	Actual class	Predicted class
Move vs. position annotation	with the threat rf6 and re4	1	2
	black find himself in quiet passive position	2	1
Move (2 classes)	lead to a decisive advantage .	1	2
	now he also lose pair of bishop .	2	1
Position (3 classes)	and i don't see more for white than a draw .	2	1
	with small advantage for black . e.g.	3	1

Table 24: Simple misclassified instances

As already mentioned in section 4.6, binary classification problems do not need to be re-examined for optimality if the cost matrix is symmetrical and contains equal values on the main diagonal. For the other three problems, the best two

configurations of the accuracies are compared using the absolute and squared costs as defined in section 2.5. The results are shown in table 25, the lower costs of both configurations are marked in bold.

Data set	Configuration	Absolute cost	Squared cost
mixed	Count - MCC0	1565	3367
	Count - OCC	**1541**	**3259**
short	Count - NDD	**1463**	**3137**
	Count - MCC3P	1488	3232
long	Count - MCC0	**1672**	3668
	Count - NDC	1684	**3658**

(a) Move (6 classes)

Data set	Configuration	Absolute cost	Squared cost
mixed	Count - MCC3P	**1244**	**2004**
	Count - MCC0	1253	2027
short	Count - RF	**1207**	**2015**
	Count - NDD	1214	2030
long	TF-IDF - NDD	**1135**	**1761**
	TF-IDF - MCC0	1154	1812

(b) Position (3 classes)

Data set	Configuration	Absolute cost	Squared cost
mixed	Count - NDC	**2518**	7178
	Count - MCC3P	2523	**7153**
short	Count - NDC	2786	9758
	Count - MCC3P	**2743**	**9365**
long	Count - NDD	2281	5475
	Count - MCC3P	**2243**	**5247**

(c) Position (7 classes)

Table 25: Comparison of absolute and squared costs

In three out of nine cases, the second configuration is better than the first in absolute costs, and in squared costs it is in even five out of nine cases. Of these, all five cost improvements have occurred in problems with more than three classes. This suggests that for these problems the optimization of accuracy occurs at the expense of cost. It also would have been expected that the ordinal class classifier beats the other classifiers taking into account the costs, as is the case for the mixed-length data set with the move annotation problem. It is the only classifier that takes into account the order of classes and should therefore be less likely to confuse classes that are further apart. However, this expectation could not be confirmed in general, as can also be seen in the confusion matrices of table 23.

6 Summary and Outlook

In this thesis, a text mining process in the field of chess annotations is introduced. An overview of the structure of a noted chess game and the different types of comments, NAG and symbols it contains is given. After preprocessing by NLTK, four different feature extraction models were created and compared using different classifiers, data sets and problems. The count-based model clearly performed best, followed by the TF-IDF-based model. Of the two word embeddings, the pre-trained Google News word embedding achieved the better accuracy values. Nevertheless, it could be shown that the word embedding learned on the chess comments also uses a meaningful assignment of vectors, whereby the similarities between words from chess vocabulary are even better and higher than in the Google News word embedding. Among the classifiers, there was no clear trend which would provide the best accuracies. A pattern was found when comparing the accuracies on the different data sets. Significantly higher accuracies could be achieved with the short comments than with mixed-length or long comments. This can be explained by the fact that, due to the initial filtering of comments for short comments, the proportion of English-language tokens is at least one third.

Overall, it can be stated that the achieved accuracies are not sufficient to make reliable classifications. Only for the two binary classification problems the accuracies are between 75% and 80%, which is still too error-prone to use these models for assigning annotation symbols to comments without annotation symbol yet. In the classification of the move comments into six classes, about 55% accuracy was achieved, in the classification of the position comments into three classes about 60% and in the classification of the position comments into seven classes only about 40% accuracy. It also had to be noted that supposedly simple comments were misclassified in all models. The confusion matrices also indicated that for many instances only the most likely class was assigned because no criteria were found for a particular class. In the following, improvements in data selection, preprocessing, model generation and classification are presented that could solve these problems.

There is already a lot of room for improvement during the extraction process. A complex, but promising further development would be the parsing of the PGN files. This would make it possible to correctly recognize divided comments or comments preceding the annotation symbol. Conversely, comments incorrectly assigned to the preceding annotation symbol could be ignored. This would lead to a significant increase in data quality, since many cross-references to other games, which thus do not refer to the current move or position, would not be included in the data set. Alternatively, such statistical comments could also be filtered out by selecting a minimum proportion of English words instead of a minimum number of occurrences. An even bigger advantage achieved by parsing would be knowing which player made the last move. This would allow move comments, which describe the change on the board from an observer's point of view, to be transformed into consequences for the last player. Similarly, position comments could be better classified if the description from the point of view of the last player could be transformed to the consequences on the board for white and black.

In addition to the extraction process, optimizations can also be performed during preprocessing and feature selection. The removal of English stopwords turned out to be disadvantageous during the first tests, but a stopwords list created especially for chess games could bring an improvement. It could also be investigated whether lemmatizing is actually better than stemming or no normalization at all. Categorizations are also possible, e.g. all chess moves could be grouped into one category or the equivalent phrases "better is", "#C4" and "$142". Such measures lead to a quantitative deterioration, but a qualitative improvement of the features. Instead, besides the already included comment length, other interesting features could be added to the model, e.g. the final result of the chess game or the elo ratings of the chess players. In general, attributes of different models, e.g. count and TF-IDF values, can be combined with each other if no redundancies arise.

As mentioned above, the better results of the self-created word embedding based on the chess comments in comparison to the pretrained model via Google News are not reflected in the achieved accuracies in the classification. A possible reason has already been discussed with the problems of classification with terms that are too similar. An improvement could be achieved by learning word embedding on additional data such as comments without annotation symbols which of course could also lead to improvements in the count-based and TF-IDF-based model. Another possibility for improvement, which can be applied to both word embeddings, is the definition of a better function for calculating the comment vector. Especially with long comments many unimportant words can lead to the fact that the vectors of the decisive words hardly influence the comment vector. An interesting alternative model that uses tree structures to determine a final value for a comment from individual word sentiment values was developed by Socher et al. [Soc+13].

The classification process also offers some possibilities for further development. If larger computing capacities are available, an attribute selection only on the basis of the term frequencies could be avoided and be built into the classification process instead. Weka offers an AttributeSelectedClassifier, which selects a suitable subset of attributes

only based on the training data. This procedure has already been applied to the preselected attributes, but could not lead to any further improvements. With the formation of ensembles from different classifiers, the accuracy values could also be increased.

Finally, the target classes must also be questioned. A strict subdivision into move and position comments is questionable; in this case a multilabel classification might be conceivable. For the move and position comments it would have to be checked whether a fine partition into six or seven classes is possible at all or whether the additional classes overlap too much. Otherwise the reduction to two or three classes would remain, or the adjustment of the costs to a very low value, so that a choice of the "neighbor" class would be punished only slightly. In addition to the final evaluation, these cost values can also be included in the learning process. Weka offers the CostSensitiveClassifier for this purpose, which, however, did not lead to any significant improvement in costs during initial tests.

References

[AS16]   Vishal A. Kharde and S.S. Sonawane. "Sentiment Analysis of Twitter Data: A Survey of Techniques". In: *International Journal of Computer Applications* 139.11 (Apr. 2016), pp. 5–15. ISSN: 0975-8887. DOI: `10.5120/ijca2016908625`.

[AZ12]   Charu C Aggarwal and ChengXiang Zhai. "A survey of text classification algorithms". In: *Mining text data*. Springer, June 2012, pp. 163–222.

[Aly05]   Mohamed Aly. "Survey on multiclass classification methods". In: *Neural networks* 19 (2005), pp. 1–9.

[AC10]   Sylvain Arlot and Alain Celisse. "A survey of cross-validation procedures for model selection". In: *Statist. Surv.* 4 (2010), pp. 40–79. DOI: `10.1214/09-SS054`.

[BKL09]   Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[Bis06]   Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[DB94]   Thomas G Dietterich and Ghulum Bakiri. "Solving multiclass learning problems via error-correcting output codes". In: *Journal of artificial intelligence research* 2 (1994), pp. 263–286.

[DFK05]   Lin Dong, Eibe Frank, and Stefan Kramer. "Ensembles of Balanced Nested Dichotomies for Multi-class Problems". In: *Knowledge Discovery in Databases: PKDD 2005*. Ed. by Alípio Mário Jorge et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 84–95. ISBN: 978-3-540-31665-7.

[Edw94]   Steven J. Edwards. *Standard: Portable Game Notation Specification and Implementation Guide*. available at http://www.saremba.de/chessgml/standards/pgn/pgn-complete.htm, accessed 19.03.2019. 1994.

[FH01]   Eibe Frank and Mark Hall. "A Simple Approach to Ordinal Classification". In: *Machine Learning: ECML 2001*. Ed. by Luc De Raedt and Peter Flach. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 145–156.

[HM18]   Alrence Halibas and Varusai Mohamed. "Application of Text Classification and Clustering of Twitter Data for Business Analytics". In: Mar. 2018. DOI: `10.1109/MINTC.2018.8363162`.

[HT01]   David J. Hand and Robert J. Till. "A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems". In: *Machine Learning* 45.2 (Nov. 2001), pp. 171–186. ISSN: 1573-0565. DOI: `10.1023/A:1010920819831`.

[Har15]   Frank E Harrell Jr. *Regression Modeling Strategies : With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*. 2015. URL: `http://dx.doi.org/10.1007/978-3-319-19425-7`.

[Joa97]   Thorsten Joachims. "A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization". In: *proceedings of the 14th International Conference on Machine Learning, 1997*. Jan. 1997, pp. 143–151.

[KP04]   Sotiris B. Kotsiantis and Panagiotis E. Pintelas. "A Cost Sensitive Technique for Ordinal Classification Problems". In: *Methods and Applications of Artificial Intelligence*. Ed. by George A. Vouros and Themistoklis Panayiotopoulos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 220–229.

[Kus+15] Matt J. Kusner et al. "From Word Embeddings to Document Distances". In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 957–966.

[Kwa17]   Ted Kwartler. *Text mining in practice with R*. John Wiley & Sons, 2017.

[Liu12]   Bing Liu. "Sentiment Analysis and Opinion Mining". In: *Synthesis Lectures on Human Language Technologies* 5.1 (2012), pp. 1–167. DOI: `10.2200/S00416ED1V01Y201204HLT016`. eprint: `https://doi.org/10.2200/S00416ED1V01Y201204HLT016`.

[MRS08]   Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521865719, 9780521865715.

[Mik+13] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *Proceedings of Workshop at ICLR* 2013 (Jan. 2013).

[Mit97]   Thomas M. Mitchell. *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997. ISBN: 0070428077, 9780070428072.

[PL08]   Bo Pang and Lillian Lee. "Opinion Mining and Sentiment Analysis". In: *Foundations and Trends® in Information Retrieval* 2.1–2 (2008), pp. 1–135. ISSN: 1554-0669. DOI: `10.1561/1500000011`.

[PSM14]   Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543.

[Plu80]   Robert Plutchik. "A general psychoevolutionary theory of emotion". In: *Theories of emotion*. Elsevier, 1980, pp. 3–33.

[Qin+10]   Zhenxing Qin et al. "Cost Sensitive Classification in Data Mining". In: *Advanced Data Mining and Applications*. Ed. by Longbing Cao, Yong Feng, and Jiang Zhong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–11. ISBN: 978-3-642-17316-5.

[RLT14]   Yu-Xun Ruan, Hsuan-Tien Lin, and Ming-Feng Tsai. "Improving ranking performance with cost-sensitive ordinal classification via regression". In: *Information Retrieval* 17.1 (Feb. 2014), pp. 1–20. ISSN: 1573-7659. DOI: `10.1007/s10791-013-9219-2`.

[SH14]   Andreas Schieber and Andreas Hilbert. *Entwicklung eines generischen Vorgehensmodells für Text Mining*. Technische Universität Dresden, Fakultät Wirtschaftswissenschaften, 2014.

[Soc+13]   Richard Socher et al. "Recursive deep models for semantic compositionality over a sentiment treebank". In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1631–1642.

[TRB10]   Joseph Turian, Lev Ratinov, and Yoshua Bengio. "Word Representations: A Simple and General Method for Semi-supervised Learning". In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. ACL '10. Uppsala, Sweden: Association for Computational Linguistics, 2010, pp. 384–394.

[Tur02]   Peter D Turney. "Types of cost in inductive concept learning". In: *arXiv preprint cs/0212034* (2002).

[WZ10]   Deqing Wang and Hui Zhang. "Inverse-Category-Frequency Based Supervised Term Weighting Schemes for Text Categorization". In: *Computing Research Repository - CORR* 29 (Dec. 2010).

[WF15]   Christian Wirth and Johannes Fürnkranz. "On Learning From Game Annotations". In: *IEEE Trans. Comput. Intellig. and AI in Games* 7.3 (2015), pp. 304–316. DOI: `10.1109/TCIAIG.2014.2332442`.

[WF14]   Christian Wirth and Johannes Fürnkranz. "Preference Learning from Annotated Game Databases". In: *Proceedings of the 16th LWA Workshops: KDML, IR and FGWM, Aachen, Germany, September 8-10, 2014*. 2014, pp. 57–68.

[WF05]   Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 0120884070.

[WLW04]   Ting-Fan Wu, Chih-Jen Lin, and Ruby C. Weng. "Probability Estimates for Multi-class Classification by Pairwise Coupling". In: *J. Mach. Learn. Res.* 5 (Dec. 2004), pp. 975–1005. ISSN: 1532-4435.