**4**

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 section~\ref{sec:evaluation\_of\_results}.

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 \citeauthor{Mitchell1997} presented in subsection~\ref{subsec:definition\_of\_the\_problem\_and\_goals}, we have:

\begin{itemize}[noitemsep]

\item Task $T:$ determine the correct standard code for a given chess annotation comment

\item Performance measure $P:$ percentage of correctly assigned codes in all code assignments (accuracy)

\item Experience $E:$ database with tuples of comments and correct codes

\end{itemize}

Before further specifying the input and output of the task $T$ in subsection~\ref{subsec:problem\_specification}, 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 subsection~\ref{subsec:data\_set\_extraction} and the performance measure $P$ will be adapted to the problem in subsection~\ref{subsec:evaluation\_methods}.

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 \autocite[Section~1]{Edwards1994}. A sample game in PGN notation is shown in figure~\ref{fig:sample\_pgn\_game}.

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.

\begin{quotation}

7. Bg5 a6.

\end{quotation}

Each move contains the piece by a single upper-case letter except of the pawn (see table~\ref{tab:basic\_chess\_notations}) followed by the square the piece is moved to (see figure~\ref{fig:square\_names}). 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:

\begin{quotation}

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

\end{quotation}

Additionally, there are some further moves with a special notation (see table~\ref{tab:basic\_chess\_notations}). 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 \autocite[Chapter~8]{Edwards1994}.

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 subsection~ref{subsec:nags}) 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~\ref{tab: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.

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:

\begin{itemize}[noitemsep]

\item NAG zero is used as a placeholder

\item NAGs with values from 1 to 9 annotate the move just played.

\item NAGs with values from 10 to 135 annotate the current position.

\item NAGs with values from 136 to 139 describe time pressure.

\end{itemize}

The NAGs with values from 140 to 255 are partially defined and used unofficially. The most common NAGs are listed in table~\ref{tab:meaning\_of\_nags} (see \autocite[Section~10]{Edwards1994}).

As shown in table~\ref{tab:meaning\_of\_nags}, 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.

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~\ref{tab:file\_statistics}) - 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:

\begin{itemize}

\item Classification into move and position annotations

As already seen in table~\ref{tab:meaning\_of\_nags}, 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:

\begin{quotation}

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

\end{quotation}

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

\item 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:

\begin{quotation}

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

\end{quotation}

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):

\begin{quotation}

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

\end{quotation}

\item 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:

\begin{quotation}

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

\end{quotation}

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):

\begin{quotation}

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

\end{quotation}

\end{itemize}

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

Data Set Extraction

As data sources a set of files\footnote{\url{http://www.angelfire.com/games3/smartbridge/}} in standard PGN format is used as well as a bundle of commented games that have been extracted from Mega Database 2012\footnote{\url{https://shop.chessbase.com/en/products/mega\_database\_2012}} 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~\ref{fig:sample\_pgn\_game}. 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 kit NLTK is used. \\\\

NLTK\footnote{\url{https://www.nltk.org/}} 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\ref{fig:nlp\_pipeline}. Note that the pictured steps of tokenization and normalization are not considered in this section, but in subsection~\ref{subsec:nltk\_preprocessing}.

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

where the variable \textit{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~\ref{tab:comment\_move\_combinations}, 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:

\begin{itemize}

\item NAGs immediately followed by a comment:

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

\item NAGs followed by another NAG and thereafter a comment:

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

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

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

\end{itemize}

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 subsection~\ref{subsec:problem\_specification}. 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~\ref{tab:class\_distributions} 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.

The last criterion quality will be discussed in subsection~\ref{subsec:nltk\_preprocessing}. 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. The limitation to certain minimum quality standards will lead to a reduction in the amount of data, whereby the quantity remains sufficiently high Besides, the reduction has the advantage that the training time is significantly shortened. \\\\

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~\ref{tab:file\_statistics}, 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.

To get the values of table~\ref{tab:file\_statistics}, 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 \textbackslash d\textbackslash+\*\textbackslash s\*[!\textbackslash?]+ (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.

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\textunderscore 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\textunderscore tokenize" for punctuation symbols is not appropriate for all cases. If move variants are presented in a comment or additional NAGs are used, "word\textunderscore 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 \autocite[Section~7]{Edwards1994} and some comments were examined manually for their structure. This leads to a regular expression based on the following cases:

\begin{itemize}

\setlength\itemsep{0.4em}

\item \makebox[7cm][l]{PGN non-standard codes} \#[\textbackslash w\textbackslash d]{2}

\item \makebox[7cm][l]{NAG non-standard codes} \textbackslash\$\textbackslash d+

\item \makebox[7cm][l]{Move-annotating symbols} [!\textbackslash?]+

\item \makebox[7cm][l]{Position-annotating symbols } [\textbackslash-\textbackslash+/=]+

\item \makebox[7cm][l]{Remis symbol} 1/2

\item \makebox[7cm][l]{Abbreviations} (?:\textbackslash w\textbackslash.)+

\item \makebox[7cm][l]{Multiple dots} \textbackslash.+

\item \makebox[7cm][l]{Words (including hyphens or apostrophes)} [\textbackslash w\textbackslash d\textbackslash-\textbackslash']+

\item \makebox[7cm][l]{Remaining non-whitespace characters} \textbackslash S

\end{itemize}

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 subsection~\ref{subsec:tokenizer\_evaluation}. \\\\

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. \\\\

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~\ref{fig:nltk\_preprocessing}.

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. \\\\

Attribute Extraction 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.

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.

\begin{itemize}

\item Count-based Model

The count-based model uses the CountVectorizer class of the machine learning library scikit-learn\footnote{\url{https://scikit-learn.org/stable/index.html}}. 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 unigrams, 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.

\item 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 unigrams, 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.

\item Own Word2Vec-Model

This model uses the word2vec module of the machine learning library gensim\footnote{\url{https://radimrehurek.com/gensim/models/word2vec.html}} 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.

\item 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.

\end{itemize}

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 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.

Conversion to ARFF

Weka\footnote{\url{https://www.cs.waikato.ac.nz/ml/weka/}}, 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 subsection~\ref{subsec:classification\_algorithms}. The algorithms are applied to data represented in Attribute-Relation File Format (ARFF)\footnote{\url{https://www.cs.waikato.ac.nz/ml/weka/arff.html}}. 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 like as shown in figure~\ref{fig:sample\_arff\_files}. 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.

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~\ref{fig:sample\_arff\_file\_complete}. 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~\ref{fig:sample\_arff\_file\_sparse}) 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. \\\\

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.

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~\ref{fig:weka\_classify}.

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 subsections~\ref{subsec:multiclass\_classification}~and~\ref{subsec:multiclass\_classification} - are used for the analysis:

\begin{itemize}

\item 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.

\item 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.

\item 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.

\item 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.

\item 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.

\item MultiClassClassifier, method 3 (MCC3)

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

\item 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.

\item OrdinalClassClassifier (OCC)

The ordinal class classifier can be installed separately and is based on the already introduced method of \citeauthor{Frank2001}, 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.

\item 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.

\item 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.

\item 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.

\end{itemize}

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. 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.

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 subsection~\ref{subsec:cost\_sensitive\_learning} are considered. An inclusion of these cost matrices in the classification process via cost-sensitive learning is not carried out.