

Multi-Label Classification of Movie Genres using Text-based Features and WordNet Hypernyms

Bachelor's Thesis in Artificial Intelligence

Radboud University Nijmegen

Faculty of Social Sciences

Department of Artificial Intelligence

Supervisor: Franc Grootjen

Sam van der Meer, s4344839

June 18, 2019

Abstract

Text categorization techniques have become increasingly more important in the past decade. Whereas many approaches rely on video or audio features for classifying digital media, text-based features provide a considerable amount of information and are computationally inexpensive to process. In this thesis we present a large movie subtitle database of data in natural language, which will be used to predict genre labels in a multi-label classification problem. We provide methods to extract text-based features and reduce attribute dimensionality effectively. We also demonstrate the generation of a second dataset using WordNet, where all words from the original subtitles are replaced by their direct hypernyms. A final distinction is made within datasets to include TF-IDF-transformations or not. We hypothesize that the dataset containing hypernyms will outperform the original dataset of text-based features. Furthermore, we hypothesize that TF-IDF-transformation has a positive effect on classification accuracy. A selection of multi-label classification techniques were tested on their performance using the four conditions. Results show very good scores on classification performance but no significant difference between the four experimental conditions.

Contents

1	Introduction	3
1.1	Related work	4
1.2	Terminology	6
1.2.1	Domain specification	6
1.2.2	The complexity of movie genres	7
1.2.3	The categorization of genres: a multi-label classification problem	7
1.3	Main Research	8
2	Methods	10
2.1	Experimental setup	10
2.2	First steps towards a suitable subtitle database	11
2.3	Data acquisition	12
2.3.1	Creating the database	12
2.3.2	Building IMDbLink	13
2.3.3	Frequency analysis	15
2.4	Data pre-processing	17
2.4.1	Data extraction and filtering	17
2.4.2	Transformation to feature vectors	19
2.4.3	Attribute dimensionality reduction	22
2.5	WordNet: A lexical database	24
2.6	Classification algorithms	27
2.7	Evaluation Measures	29
3	Results	31
4	Conclusions	35
5	Discussion	37

Chapter 1

Introduction

The amount of original content appearing in digital media faced an exponential growth on the internet and on other platforms over the past two decades. The digital revolution also had a significant impact on new movie releases. Compared to traditional ways of capturing, distributing and promoting a movie, the internet has dramatically changed the effort and cost associated with movie making. Not only has the production cost decreased drastically, filmmakers now have an array of options available to share their ideas, to find funding for their projects and to keep in touch with their prime audience, producers or investors (Waldfogel, 2017).

These changes are reflected in the amount of annual movie releases. The largest online movie database, IMDb (www.imdb.com) showed a steady growth of new movie releases over the past two decades. Around four thousand movies were created in the year 2000, ten years later this number had risen to 7500 and more than 11.000 new titles were published only last year. More than five thousand of those had their origin in the United States, which dominates the film industry. An even larger magnitude of other video material has appeared online in the form of smaller videos or exclusive movies, which have seen an explosive growth over the past couple of years with video streaming services such as Netflix and YouTube increasing in popularity.

One major downside of the uncontrollable amount of videos that are published every day is that it has become infeasible for humans to supervise all content manually. Because of this, automatic classification tools have shown to provide an effective solution to monitor the content appearing in multimedia in a structured way. For example, movies containing profanity or violence must be classified separately so that they can be hidden from sensitive user groups, such as children. Another application of automatic

categorization are recommender systems. If all labels are known for videos in a large corpus, one can recommend a selection of similar videos according to the user’s preferences.

1.1 Related work

Recent literature has shown that genre classification can be accomplished through (a combination of) three modalities: visual, audio and text-based. Brezeale and Cook (2008) conducted a literature survey of the video classification literature available at that time. It has become apparent that the majority of the literature focuses on single modalities only, with the most research being conducted in the visual domain. Although the visual domain has been studied most, it also suffers from computational complexity the most. Generally, movies are filmed with 20-60 frames per second. Each of these frames contains more than a million pixels and the majority of them differ between frames. Analysis of an average two-hour movie requires a lot of computing power and is not easily scalable to a general classification problem, which requires hundreds or thousands of instances to work properly. As a result, most studies compare various visual features of short scenes or trailers, in order to reveal information about the content of the movie itself. These can then be used to infer a more general selection of genres (e.g. Rasheed and Shah (2005)). Studies that analysed auditory features recorded them over very small ranges (a few seconds) in order to infer a selection of movie genres. Additionally, various auditory stimuli appeared to be hard to isolate from a spectrum of the signal.

Compared to the visual and auditory modalities, text-based approaches are far more efficient in terms of computational resources, which caters to the problem at hand. Many movies will be able to be analyzed in a short time period, due to the properties and the structure of textual data. Text-based data in the domain of movies are most easily extracted from the corresponding subtitle. Other information sources include transcripts or closed captioning. These data structures are nothing more than a semi-structured sequence of text, which is enclosed in part-of-speech. Text strings can be efficiently coded with a relative low amount of bits. Whereas computations in the visual domain often include a few gigabytes of data, a single subtitle file of a full-length movie is sized between a mere 2-200kB. These text-based strings contain linguistic information which can be analyzed with natural language processing techniques. A vast body of research has been conducted on information retrieval and text categorization techniques (Sebastiani, 2002).

While the amount of studies that consider movie genre classification using text-based features is limited compared to other domains, we will give a short overview of related work. Brezeale and Cook (2006) used closed captioning in parallel with an approach from the visual domain to classify movie genres and user ratings. They mention that some movies had multiple genre labels, but reduced the classification problem to a single-classification problem as they used a support vector machine (SVM) classifier only. The two approaches were found to perform equally well, but movie genre classification (class accuracy 87-90%) significantly outperformed the prediction of user ratings. Chao and Sirmorya (2016) presented a novel system for movie genre classification using probabilistic topic modeling (LDA) and movie scripts as text-based source. They used the cosine similarity measure between feature vectors to create a ranking of movie genres and simply took the top five or top six genres. Baseline classification algorithms included the majority label algorithm (simply takes top-x occurring labels), SVM and Random Forest (RF). The LDA approach was able to classify genres with lower precision but higher recall scores than baseline algorithms, resulting in a moderate F_1 -score of 49% (harmonic mean between precision and recall). An analysis of multiple text-based feature types in Dutch movies and TV-shows from the SUBTIEL-corpus showed that the SVM and RF classifiers proved most accurate in classifying genres, with a maximum F_1 score of 88% (Lee, 2017). Another study categorized news categories instead of movie genres (Zhu, Toklu, & Liou, 2001). They used a weighted voting method and Bayes’ decision metric. Their confusion matrix showed that different categories scored precision and recall rates with a high variance. For categories that scored poor, this was due the overlap with other categories, insufficient training samples or the lack of unique language features. The average prediction accuracy worked out to be 75-80% based on the decision method.

Katsioulis, Tsetsos, and Hadjiefthymiades (2007) used subtitles of documentaries in conjunction with WordNet domains to train a classifier. Using keyword extraction and word sense disambiguation, they made a mapping between the corresponding WordNet domains and category labels. They achieved a classification accuracy of 89% with the J4.8 classifier.

Many evaluation metrics for multi-label classification problems seem to exist (e.g. Maimon and Rokach (2010), Sebastiani (2002) and Sorower (2010)). In the literature, the most appearing evaluation metrics are accuracy, precision, recall, F_1 -score, area under the ROC-curve and Hamming-loss. How a problem should be evaluated is said to depend on the evaluation target: partitioning into classes, label rankings or label hierarchy.

1.2 Terminology

1.2.1 Domain specification

Many studies limit the scope of video material that is being analysed. News bulletins, TV-shorts and trailers are among the most popular ones, which only last for a few minutes. The meta-collection of videos that appears on IMDb reflect this wide array of video formats. In advanced title search, one can filter between feature films, TV movies, -series, -episodes, -specials and shorts, mini-series, documentaries, short films, video games and video.

For this thesis, we limit ourselves to one specific class of video, namely feature films. Also called theatrical film, motion picture or movie (equivalent terms), the first productions date back to the early 1900s. This type of video material is characterized to have a running time long enough to fill a program. Most institutes determined that a feature film should run for at least 45 minutes, while others increase this number to 75 (British Film Institute and Screen Actors Guild respectively). Most movies are between 75 minutes and three hours long. Moreover, movies don't require themselves to be shown in theatres necessarily. Stand-alone titles are seen more and more the past couple of years, which are often released in smaller communities or uploaded to the internet. Recently, many video streaming services such as Netflix or HBO have been producing platform-specific content as well. For the thousands of movies that are released every year, only a small proportion (around 500 (IMDb)) are released in theatres.

One advantage of feature films (further denoted as movies) is that the text-based component in the form of subtitles or closed captioning are almost always encapsulated in a separate file accompanying a movie, which means they are easy to isolate and to be stored in a database.

One last important thing to mention is that we limit ourselves to UK/US-movies only, which are almost always produced in English. Although a lot of new releases are in foreign languages, this would complicate our analysis as different words are used for the same concept across languages. In natural language processing, which underlies the classification of textual information, one should limit their research to one specific language. This choice is also supported by the fact that the majority of new films originate from the United States, with Hollywood being the home of the film industry worldwide. We decisively did not choose to use translated subtitle files from foreign movies, as certain part-of-speech elements cannot be effectively translated to English and they often contain errors.

1.2.2 The complexity of movie genres

A varying range of genres exist in movie classification. IMDb enumerates *popular* movie genres as follows: comedy, sci-fi, horror, romance, action, thriller, drama, mystery, crime, animation, adventure, fantasy, comedy-romance, action-comedy and superhero. Other genres that are not mentioned here include biographies, documentaries, family films, fantasy, film noir, history, musical... (the list goes on). Stephen Follows, an established film data analyst for many English newspapers, has conducted experiments to determine the most occurring genres overall (Follows, 2018). Over half of all films made worldwide are dramas (51,6%), followed by comedies (28,4%). Other popular genres, presented in descending order, are thrillers (12,4%), romance (11,6%) and action (11,2%) (Note that these percentages don't add up to 100%, as movies can have multiple genre classifications).

IMDb provides genre labels for each movie in their database. Up to 28 unambiguous genre labels exist and each movie can contain up to three of them. Potential conjunctions of genres, like romcoms, would be classified with the two genres 'romance' and 'comedy' respectively. For this research, a number of high-frequency genres will have to be selected in order to train the classifiers on. We will come back to this point in a later stage.

1.2.3 The categorization of genres: a multi-label classification problem

On average, movies have 1,7 genre classifications (Follows, 2018). For example, *The Incredibles* (2004) belongs to the genres animation, action, adventure and family, while *The Wolf of Wall Street* (2013) belongs to the genres biography, crime and drama. Analysing documents of text and grouping them into one or more predefined categories belongs to the domain of text categorization (Sebastiani (2002) and Korde and Mahender (2012)). In this thesis, the set of documents will be a set of subtitles and the predefined categories will be movie genres. We will take the IMDb genre classifications as ground truth labels for our own classification problem. Nevertheless, these can contain up to three genres.

Because movies can be classified with multiple genres, we are dealing with a *multi-label* classification problem. In contrary to single-label or multi-class classification problems, multi-label problems are concerned with assigning a set of target labels to each instance. Because multi-label classification adds a certain degree of uncertainty to the sample space, a wide range of unique algorithms and evaluation metrics exist to manage the problem complexity

(Maimon & Rokach, 2010). The multi-label algorithms can be grouped in two categories: problem transformation methods and algorithmic adaptation methods. The first group transforms the problem to a set of single-label classification tasks, for which a wide range of classifiers exist (a couple were mentioned in section 1.1). The second group contains algorithms that are specifically built to handle multi-labeled data.

1.3 Main Research

We have seen that relatively little research has been conducted on text-based video classifiers in the domain of movie genres. The literature that exists provides promising numbers in terms of classification accuracy. However, most of them are based on smaller video fragments and make use of single-label classification techniques. Text-based classification analysis is important because it is both computationally efficient and useful for many purposes in document categorization, summarization and recommender systems.

Research questions

In this thesis we present a large movie subtitle database of data in natural language, which will be used to predict genre labels in a multi-label classification problem. We provide methods to extract text-based features and reduce attribute dimensionality effectively. We also demonstrate the generation of a second dataset using WordNet, where all words from the original subtitles are replaced by their direct hypernyms. A final distinction is made within datasets to include TF-IDF-transformations or not. Hypernyms are semantical generalisations of terms and TF-IDF is a measure for word relevance (more details about these notions will be clarified in chapter 2).

These four experimental conditions, along with a selection of multi-label classification algorithms, are used to classify movies into a set of genres using text-based features only. We propose two research questions to support our classification problem:

1. To what extent do WordNet or TF-IDF-transformations on feature vectors of subtitle words contribute to the multi-label classification performance of movie genres?
2. What combination of multi-label algorithms and base classifiers is best suited in the setting of text-based subtitle classification?

Hypotheses

First, we hypothesize that the dataset containing WordNet hypernyms will outperform the original dataset of text-based features. Because all terms are replaced by their direct parent terms, words that relate to the same parent term will be grouped together, thus making it clearer for a classifier which groups of words relate to which genre labels. Furthermore, we hypothesize that TF-IDF-transformation has a positive effect on classification accuracy. After all, the more relevant a word is in a specific document, the more value it gains for discriminating the corresponding genre. A selection of multi-label classification techniques were tested to see which combination maximizes various classification evaluation methods.

Thesis overview

The rest of this thesis will be organized as follows. Chapter 2 will first discuss the experimental setup and the acquisition of a suitable subtitle database, including data pre-processing. Next, we will discuss how we extracted WordNet hypernyms to generate the second dataset. After that we will explain our choices of multi-label classification algorithms, base classifiers and evaluation methods. Results will be displayed in chapter 3 and chapter 4 and 5 will provide conclusion and discussion, respectively. The final pages will be subject to the references and appendices, which include the raw data results and all code that was written for data pre-processing.

Chapter 2

Methods

2.1 Experimental setup

In order to test our hypotheses, we require a suitable database of subtitle files. We explain our data acquisition process, including the search for the corresponding genre labels of each subtitle, in sections 2.2 and 2.3. After the database has been completed, the distribution of genres will be analysed. A selection of genres are very uncommon, which leads to class imbalance and an increased complexity for label classification. Based on these observations, we make a substantiated decision to discard a certain number of instances.

From there, the instances that remain will be pre-processed. This process allows us to isolate subtitle strings and discard irrelevant (non part-of-speech based) information, which is a prerequisite for effective natural language processing. Next, we make a transformation to feature vectors, which is a more machine-friendly data structure compared to raw words in natural language. The final step in the pre-processing phase includes attribute dimensionality reduction, which limits the amount of attributes in each vector by discarding attributes that do not contribute to the information gain. All these processes are explained in section 2.4. We also generate a second dataset where each word in the subtitle files is replaced by its direct WordNet hypernym, which is described in section 2.5. Finally, the choice of classification algorithms and evaluation measures are described in sections 2.6 and 2.7 respectively.

The classification algorithms are tested across four experimental conditions. First, we make a distinction between the original dataset and the dataset generated with WordNet hypernyms. Second, the feature vectors of these datasets were created with and without TF-IDF-transformation. The conditions will be tested for three multi-label classification algorithms com-

bined with three base classifiers. The combination of these dimensions will result in a total of 36 test classifications. All tests were executed using 10-fold cross-validation and will be further discussed in chapter 3.

2.2 First steps towards a suitable subtitle database

Prior to training the classification algorithms, we first need to acquire a database that consists of movie subtitle files. Unfortunately, such an open-source corpus is not readily available on the internet due to copyright issues. Instead, F. Grootjen was able to provide me with a very large subtitle database ($N = 12083$). The collection of subtitle files was composed of many different video sources and was used in a former bachelor’s thesis project. In addition to the database came software that was written by him in Java, named `CleanUpSubtitles` (Grootjen, 2018). The program made an attempt to dispose as much irrelevant information from the subtitle filename, while maintaining the movie title and production year (see figure 2.1).

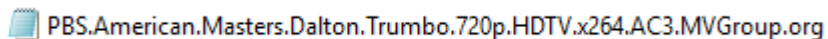


Figure 2.1: Example of a downloaded subtitle file, where much unnecessary information needs to be filtered out correctly in order to reduce this title to *Trumbo* (2007).

From there, the most probable IMDb-movie title was linked with each subtitle filename using the Levenshtein similarity measure. Each match was also accompanied with an unique IMDb identification number. This ID is a very informative source for this research, as it contains valuable information about the corresponding movie of the subtitle (including production year, runtime and most importantly, up to three genre classifications). This allows for an automatic provision of classification labels, provided that the subtitle is linked with the correct movie.

A couple of issues arose with the performance analysis of Grootjen’s program, which are unique for this research. First, the program takes a very long time to match all subtitle files with the corresponding IMDb-ID (up to four hours), as each reduced subtitle filename has to be compared to all IDs that exist (over five million). Second, though the subtitle database is very large, the majority of subtitles in it originate from television series, which don’t belong in the category feature films. The subtitle database also contains many duplicates and subtitles in a range of different languages. Third, while the program was able to correctly link over 50% of subtitles

with their corresponding ID, there were still plenty of errors being made. Consequently, this led to thousands of obligatory manual checks. Because the scope of this research limits us to movies in English only, a large proportion of the database had to be discarded, and less than a thousand unique titles remained. In the end, we decided it would be beneficial for our research if we built our own database of movie subtitles, accompanied by a personalized linking algorithm. The source code of `CleanUpSubtitles` provided some very useful insights in how to accomplish this.

2.3 Data acquisition

2.3.1 Creating the database

Our goal was to build a database containing a couple of thousand unique English movie subtitles, in order to guarantee plenty of data to train our classifiers with. A few subtitle extensions exist to represent the data. For this research, we limited ourselves to the `.srt`-extension (SubRip). This extension is text-based, which is useful for natural language processing later on. Furthermore, the file sizes are really small (under 100kB on average) and other than the timestamps, the files don't include any meta-data.

In contrary to subtitle databases, individual movie subtitles are readily available on a various number of websites. We decided to use <https://www.opensubtitles.org> as our source of subtitle files. Per production year we were able to generate a list of the most popular movie subtitles in `.srt`-format, which were easily downloadable from there. Movies made between 1960 and 2019 were included in our database, resulting in a total number of $N = 4736$ unique movie titles. One major advantage of using `opensubtitles.org` over other sites was that the directory layout of each downloaded subtitle file was very convenient (see figure 2.2). Each name of the downloaded (compressed) folder was structured the same, which proved to be very useful when trying to extract movie title and production year.

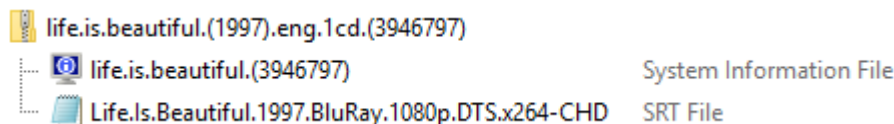


Figure 2.2: Directory layout of a random movie subtitle file.

For every compressed folder download, the filename starts with the title of the movie. It is followed by the production year of the movie in between brackets. Then the language of the subtitle is appended, which in this research was always `.eng`. The language is followed by a `x`-number of subtitle files (`xcd`) that appear inside the compressed folder. Finally, each filename is terminated with a number in between brackets, which indicates some kind of subtitle version (different from the IMDb-ID). Hence this last number is not relevant for this research. Every attribute is separated by dots, as well as whitespaces in the movie title.

These features were always structured like this, with the (possibly noisy) subtitle filename(s) inside the folder. Because the parent folder name contains no additional irrelevant information (in comparison with the subtitle-file itself), it is very convenient to use that one for linking the child subtitle-file with its corresponding IMDb-ID. For this reason, the database is composed of extracted folders for each individual movie, where the subtitle files themselves appear inside ($N = 4775$). Please be aware of the fact that some subtitle files are inherently subject to vulgar or discriminating language and remain unfiltered to maintain data integrity.

2.3.2 Building IMDbLink

With the subtitle database ready to go, we programmed our own software equivalent of `CleanUpSubtitles` and named it `IMDbLink`. The program was coded in Java. The classes and code can be found in Appendix B.

Two input files are required to run the program. First, we included the subtitle database from section 2.3.1 inside a compressed folder in our working directory. This enables our program to decompress, read and write subtitle files. Second, we included a thorough database of every single existing IMDb-ID (up to may 2019), along with the relevant movie information. This database was obtained from the IMDb website (<https://datasets.imdbws.com>), which is offered as open-source material. The compressed folder (`imdbData.zip`) contains one data file, `data.tsv`, which our program is able to read.

The IMDb-data file contains an useful number of attributes. First, the IMDb-ID number appears, represented as a number preceded by `'tt'` in string format. It is followed by two title attributes, which are primary title and original title respectively. A boolean `isAdult` indicates whether the title is an adult title or not. The production year and runtime (in minutes) are also included. Finally, the associated genres are represented as string array (up to three genre slots available per title), which is what we are after.

The class `Entry.java` represents a single ID with associated attributes. Everything is stored in this data structure, but only ID, title, year and genres are used for further analysis. The main class `IMDbLink.java` makes instantiations of all other classes, including `ReduceDatabase.java`. This class reads the IMDb data file and reduces it according to certain criteria. It saves the entries of all movies that were made after 1960 and have at least one genre classification, along with our subtitle requirements. Everything else is discarded. The reduced IMDb-database is written in a new file named `reducedIMDbData.csv`. The original database contains (as of may 2019) over 5,8 million entries and we were able to reduce it to 477k entries, specific to our problem description. This reduction of 92% is the main reason of a significant runtime improvement compared to the `CleanUpSubtitles` program.

Next, the `FetchSubtitleNames.java`-class unzips the subtitle input file and moves all `.srt`-files to a specific output folder in the project directory. The parent folders and system information files are discarded, but the partial filenames of the parent folders are extracted and transferred to substitute the filename of the subtitle files (with the `.srt`-extension).

Finally, the `MatchIMDbTitles.java`-class compares all subtitle files with each entry in the reduced IMDb database file by computing the Levenshtein distance between the two. The Levenshtein distance is a similarity measure between two strings (lower value is more similar), given by formula (1):

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise} \end{cases} \quad (1)$$

Where a and b are the character strings and i and j denote the i_{th} and j_{th} character respectively. Additionally, a normalized similarity score as a function of title length is computed to express the success rate in terms of a percentage. For each entry, the original relevant data (ID, title, year and genres) are printed to `compactReport.csv`. Additionally, the presumable corresponding IMDb movie title and the similarity score are printed in `extensiveReport.csv` for finding false associations. After opening the extensive report in a spreadsheet viewer, the relative low percentages ($< 75\%$, arbitrarily chosen) were highlighted in red, manually checked and removed from the subtitle database if the titles didn't match. This was mainly the case for non-movie titles that were accidentally downloaded with the subtitle database. Finally, the program renames all raw subtitle files by appending the corresponding IMDb-ID in front.

2.3.3 Frequency analysis

IMDb discriminates 28 stand-alone genres, which are action, adult, adventure, animation, biography, comedy, crime, documentary, drama, family, fantasy, film-noir, game-show, history, horror, music, musical, mystery, news, reality-tv, romance, sci-fi, short, sport, talk-show, thriller, war and western. Because the attributes are encoded non-orthogonally this way, they are classified separately.

In a previous version of the class `MatchIMDbTitles.java`, a method was included that kept track of all associated genre classifications that appeared. After matching all subtitles with the corresponding IMDb entries using our program, we gained some useful insights in the frequency distribution of genres in our database. There are a couple of genres that are not mentioned once across all subtitles, which are adult (mainly pornography), film-noir (smaller genre popular in '40s and '50s movies), game-show, news, reality-tv, short and talk-show (not movie related). Furthermore, it is clear that the genres are not equally distributed, which is in line with our findings in the introduction. The most occurring genres in our dataset are drama, comedy, action, adventure, crime, thriller and romance. Other genres decrease in frequency as the categories become more and more subject specific (for example sci-fi > history > western). The frequency distribution of the remaining 21 genres in our dataset is shown in figure 2.3. A total of 4775 subtitle files were analyzed and combined they contained 12332 genre classifications. On average, each movie was classified with 2,58 unique genre categories ($\sigma = 0,644$).

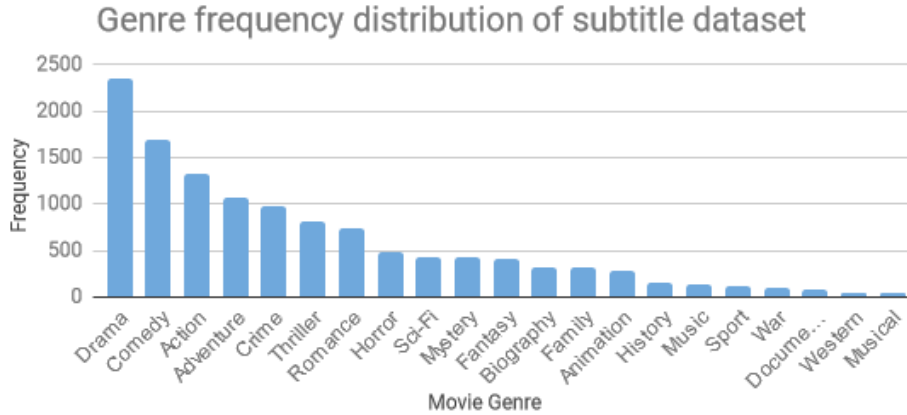


Figure 2.3: Frequency distribution of movie genres in the original database of subtitles ($N = 4775$)

For this research, our multi-label classification problem should be able to accurately classify up to three genres for each subtitle file. As more genres enter the classification problem, so increases its complexity. Including more lower frequently occurring genres in the classification problem leads to a greater class imbalance, which in turn leads to under- or over-sampling. Therefore, we decided to take the seven most occurring genres to train our classifiers with. Only movies with one or more of the seven most occurring genres were kept for further analysis. Hence, all IMDb genre classifications that are not along those seven (including their corresponding subtitles) were removed from the data set.

This resulted in a new subtitle database that contains a reduced amount of titles. 2111 subtitles remained (a reduction of 56%), with a total number of 4996 genre classifications. On average, each movie is classified with 2,37 unique genre categories ($\sigma = 0,732$). The new frequency distribution of the seven most occurring genres can be found in figure 2.4. Furthermore, it should be noted that there is a wide array of combinations between different genres. A co-occurrence matrix is shown in table 1. It becomes apparent that thrillers don't often co-occur with comedy and romance genres, the adventure and crime genres don't occur very often together and the romance genre is most of the time accompanied by comedy or drama genres only. The adventure genre occurred as fourth most frequent in the original dataset, but is ranked a mere seventh in the reduced dataset. This is because the adventure genre often appeared in conjunction with other genres that were filtered out in the previous stage.

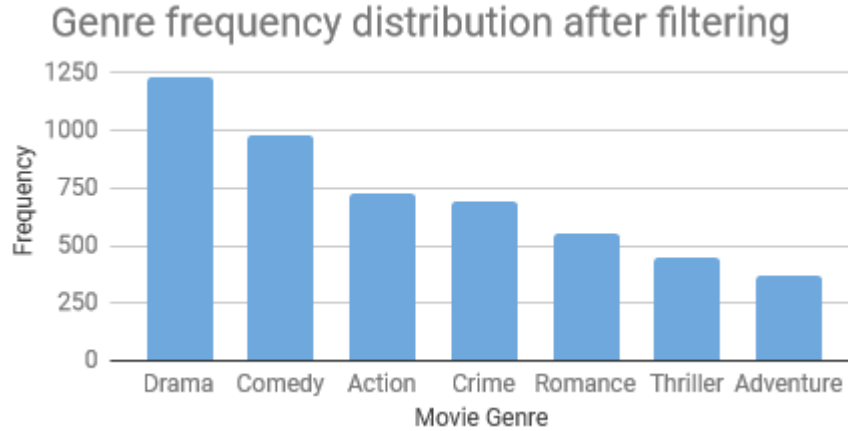


Figure 2.4: Frequency distribution of movie genres in the reduced database of subtitles ($N = 2111$)

	Action	Adventure	Comedy	Crime	Drama	Romance
Adventure	255					
Comedy	193	155				
Crime	355	56	208			
Drama	289	140	387	375		
Romance	27	24	335	30	389	
Thriller	224	67	16	224	215	15

Table 1: Co-occurrence matrix of the seven genres in the reduced database of subtitles ($N = 2111$)

2.4 Data pre-processing

2.4.1 Data extraction and filtering

Though the subtitle files are compact in size, they all contain some structural elements which have to be removed first, such that only the subtitles remain. The subtitles themselves are subject to a certain amount of noise, which has to be removed. Then, the data in natural language must be transformed into a machine-friendly dataset before classifier analysis can begin. These steps are involved in data pre-processing.

The subtitles files in `.srt`-format are all structured the same (see figure 2.5 for reference). Every movie subtitle contains a finite set of caption frames, which are sequential blocks of subtitle lines along with some accompanying data. Each caption frame begins with a sequence number on a single line, which is an integer starting from one at the beginning of the file. It increments every subsequent caption frame. Next, on a new line the timestamp is given, which gives an indication of the time the subtitle should be shown on-screen. It is always formatted with two timestamps in `hour:minute:second,millisecond`-format separated by an arrow, indicating start and ending times of the corresponding subtitle respectively.

Next, the subtitle for that particular caption frame appears on one or two lines below. All caption frames are separated by an empty line. The sequence number, timestamp and newlines are easily removable, but there also exists a variety of noise in the subtitles themselves. First, a couple of punctuation marks exist that do not contribute to the meaning of the subtitles, and should be removed. Examples are quotation marks (`"`), the dash-symbol (`'`), frequently used to indicate another person speaking in dialogue) or ellipsis marks (`'...'`). Also line terminating symbols (dots, exclamation and question

```

131
00:08:35,958 --> 00:08:38,876
<i>I just wanna know who he did it with</i>
<i>and get the pictures.</i>

132
00:08:38,999 --> 00:08:40,035
BARTENDER: <i>Yeah, I think I remember her.</i>

133
00:08:40,158 --> 00:08:41,955
Amelia, right?

134
00:08:42,877 --> 00:08:44,197
Was in here three or four nights ago.

```

Figure 2.5: Example of a few caption frames that appear in the subtitle file of the movie 'The Nice Guys' (2016).

marks) are deleted, such that text-only data remains.

Furthermore, a selection of very basic text formatting-tags exist, including `<i>...</i>`, `<u>...</u>` and `...`, for cursive, underlined and bold text segments respectively. Another one that appears is `...` for changing text colors. These obviously have to be removed as well. For subtitles that include closed captions, everything between brackets `((...))`, `[...]`, `{...}` is also discarded. That is because they often contain expressions, subjects or actions that do not relate directly to relevant dialogue (for example, '(gasps)'). Another manifestation appears when a subject is directly indicated when speaking dialogue (for example, 'MAN: Please don't panic!'). For all occurrences, the subject is discarded, but the line itself is kept. At the beginning and/or end of the subtitle files, subtitle makers often add some kind of source tag or trademark, which don't belong to the movie subtitles either and should therefore be discarded as well. Finally, everything is converted to lower case letters. This was done to prevent elements with different capitalization being mapped to different words, while in fact they are equal.

From a grammatical viewpoint, every text in natural language uses different word forms for varying grammatical contexts. For instance, verbs often change their suffixes according to the corresponding subject or grammatical context. Every instance of related words can be reduced to the dictionary

form of a word, which is called the *lemma*. For example, the verbs 'walk', 'walks', 'walked' or 'walking' all project to the lemma 'walk'. Additionally, there are certain families of derivationally related words with similar meanings, such as 'multiplication' and 'multiplier' pointing to 'multiple'. The process of reducing morphological variation is called *lemmatization* and will also be applied to reduce and combine the variability of word frequencies. Because the classifiers rely on the frequency distribution of words, it is important that variability of the same word will be handled to avoid some word forms being marked as irrelevant. We chose lemmatization over stemming because it considers grammatical context, which often depends on part of speech that frequently appears in subtitle dialogue. Stemming operates on single words independent of context and will not be able to discriminate between words with different meanings.

We have written a small java-project called `SubtitlePreprocessing` to deal with these basic issues. The program was coded in Java and can be found in Appendix C. It contains a `SubReader.java`-class which reads all lines of all subtitle files that were successfully selected by `IMDbLink`. Each reduced subtitle string, along with the IMDb-ID, are stored in a list of two-string mappings. ID numbers are included because we need to make a connection to the movie genres (attribute labels) when writing the output file. Sequence numbers, timestamps and empty lines are discarded automatically. For the lines that remain, we used a `Trimmer.java`-class to trim subtitle lines of all characteristics that were mentioned above. Next, the reduced string is lemmatized. For this we used Stanford's CoreNLP pipeline (Manning et al., 2014), which was included as a single class named `StanfordLemmatizer.java` in our project. It uses different `.jar`-files of the CoreNLP package, which were included in our project. After lemmatization, the class returns a string of words separated by whitespaces. Finally, the `SubWriter.java`-class writes a new file in `.arff`-format (see section 2.4.2) with each string on a newline, preceded by binary genre attribute labels. After these operations, our data only contains natural language and can be used for further analysis.

2.4.2 Transformation to feature vectors

For the remainder of this subsection, we utilized the WEKA extension to the Java programming environment (Frank, Hall, & Witten, 2016). WEKA is a collection of many tools and machine learning algorithms used for various classification and data mining purposes, designed by the University of Waikato in New-Zealand.

The WEKA environment requires an `.arff`-file as input for further analysis. The acronym stands for *Attribute-Relation File Format*, which is a simple ASCII-text file that describes a list of instances sharing a set of attributes. Although WEKA provides a toolkit for many pre-processing purposes, multi-label classification algorithms are not yet supported. That is why we also used MEKA (Read, Reutemann, Pfahringer, & Holmes, 2016), which is a multi-label extension to WEKA. It contains all functionalities of WEKA, but also adds methods and algorithms for multi-label classification tasks.

Because both programs weren't able to transform our `.txt` subtitle files to a single `.arff`-file, we implemented the creation of the `.arff`-file ourselves in the project `SubtitlePreprocessing`. Before presenting the subtitle texts, a header is inserted which contains a few lines that provide information about the nature and structure of the data. At the top of the file, it expects a relation declaration, which defines the relation name and the number of labels. Next, each genre is represented as a single binary attribute. In multi-label datasets, binary label attributes are the only kind of target attributes. The sequence of genres is defined as they appear in figure 2.6. They appear as a sequence preceding each subtitle string, separated by commas. For example, if a subtitle is preceded by `'0,0,1,0,0,1,0,'`, the corresponding text is labeled with the genres Comedy and Romance. The lemmatized subtitle text in natural language is defined as the final eighth attribute with type string and is enclosed in quotation marks. The header is terminated with the `@data`-line, which denotes the start of the data segment in the file.

```
@relation 'subtitles: -C 7'

@attribute Action {1,0}
@attribute Adventure {1,0}
@attribute Comedy {1,0}
@attribute Crime {1,0}
@attribute Drama {1,0}
@attribute Romance {1,0}
@attribute Thriller {1,0}
@attribute subtitle string

@data

0,0,1,0,1,1,0,"on november 1 1959 the population
```

Figure 2.6: Header of the data file and a small part of the first instance.

At this moment, our subtitle attribute is nothing but a mere random text value. By applying an alphabetic tokenizer (only keep words with contiguous alphabetic characters) on the text data in WEKA, there appears to exist more than 184k unique words between all subtitle strings. Moreover, these appear in varying frequencies for different subtitle text lengths. It should also be noted that the majority of these words do not at all appear in a dictionary, as many words simply contain typos, contain any leftover unfiltered text or encode screams, grunts or other emotional expressions. Overall, this data manifestation is not very suitable to train a classifier with.

In order to learn a classifier distinguish between movie genres, we need to select useful features from the data and represent them with vectors in the numerical domain. This transformation to a more machine-friendly dataset is called *feature selection* and the `StringToWordVector` filter in WEKA allows us to do just that. This class is able to convert string attributes into a set of attributes representing word occurrence depending on a number of settings. This filter contains a couple of settings that allow us to do some final pre-processing on all string attributes, while one other setting will be treated as variable to determine which feature vectors produce optimal performance results.

In everyday language, there exists an array of words that appear so frequently that they don't hold any information for classification purposes. Also known as stopwords, a few examples are 'a', 'the', or 'for'. The removal of stopwords significantly helps for decreasing the feature set of words (Sharma & Jain, 2015). Although an official list of stop words does not exist, we used the NLTK's list of English stopwords (Bird, Klein, & Loper, 2009), which contains 127 entries. Additionally, we included the most popular American first names (Kantrowitz, 1994) (both male and female) to the stopword list, such that they are not included in the feature vectors. Names are not genre dependent and have a high chance at appearing in the feature vector because they're generally mentioned a lot. That is why they were excluded. The stopword list was included in WEKA's `stopwordsHandler`. An alphabetic tokenizer was also included as filter (`tokenizer-setting`), in order to remove any elements that contained numbers or other unrecognized characters. An arbitrarily maximum number of 5000 words were set to keep (`wordsToKeep`), in order to control the program computational feasibility. Finally, WEKA's `outputWordCounts` was set to `True` to output word frequencies (numeric) instead of word occurrences (binary). The former contains more information compared to the latter. As a result of one movie having more dialogue than another, the raw data is composed of string attributes with varying lengths. This is reflected in the file sizes of the raw subtitle files (2kB-200kB). Because

of this variance, it is wise to normalise the data. The re-scaling of the data was done using WEKA’s `normaliseData`.

One other setting was varied across datasets, namely TF-IDF. Another way of displaying word frequencies is by applying the TF-IDF-metric (see formula 2). TF-IDF is a very popular weighting metric, with 70% of text-based recommender systems in digital libraries using it (Beel, Breitinger, Gipp, & Langer, 2016). It is a numerical statistic which reflects the importance or relevance of words in a document.

$$w_{i,j} = tf_{i,j} * \log\left(\frac{N}{df_i}\right) \quad (2)$$

For term T_j in document D_i where N is the total number of documents in the collection. TF-IDF is the result of the product of two statistics, namely TF (term frequency) and IDF (inverse document frequency). How many times a certain word appears in a document is captured by the term frequency. If a word also appears in a large corpus of other documents, then the document frequency is large. The inverse of document frequency is multiplied with the term frequency, because words that appear very often all the time are probably not that informative. On the other hand, words that appear frequent in a certain document but not in others are probably relevant for that document. As we are looking for the most predictive words in classifying each genre, it is assumed that the TF-IDF measure provides a significant contribution to our classifier performance.

2.4.3 Attribute dimensionality reduction

WEKA’s `StringToWordVector`-filter determines the top 5000 words depending on word frequencies and class labels over all instances. All subtitle attribute instances are transformed into numeric values for each of the selected 5000 words. In other words, each of these selected words is treated as a new numeric attribute and every instance scores a value for each of these new attributes. The value varies with the word occurrence in that instance. Conclusively, the eight attributes and 2111 instances of data with type string are now transformed into a dataset with dimensions 5007x2111 and type numeric. Each instance now contains feature vectors with word frequencies.

As the dimensionality of each instance is very large, we can limit the number of attributes effectively by evaluating the relevance of each word for classifying one or more genre labels. Some words are very specific for predicting a certain genre and are therefore very useful to train the classifiers with. WEKA’s `AttributeSelection`-filter is a flexible supervised evaluative

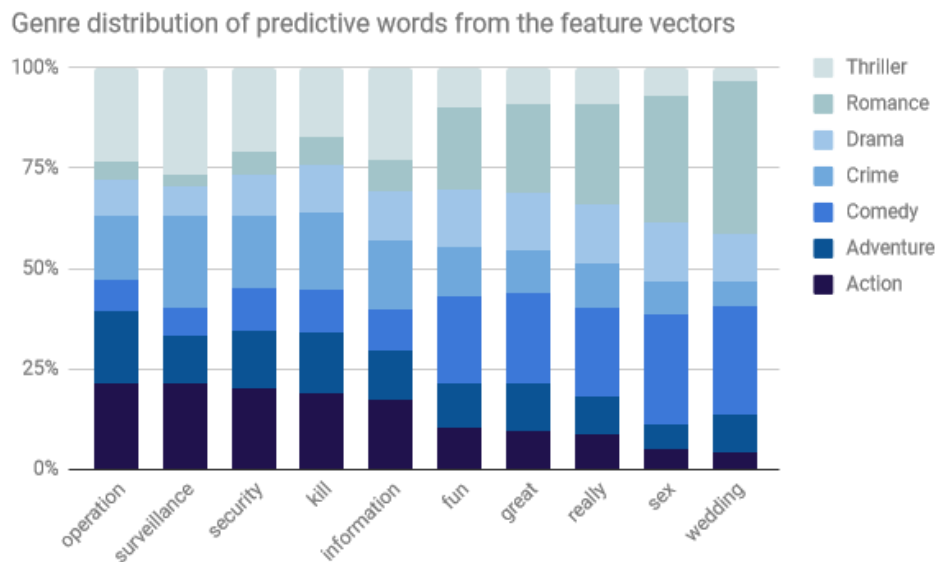


Figure 2.8: Genre distribution of a selection of words from the final feature vectors. Some words are predictive for a few genres, but rarely for others.

2.5 WordNet: A lexical database

The dataset obtained so far will be used in one experimental condition. The other one will be generated differently, namely by utilising WordNet (Miller, 1995). WordNet is a large lexical database of the English language. It is more than an ordinary dictionary can provide, because it also displays semantic relationships between words in tree-like structures. WordNet groups similar words together with an abstract concept called *synsets*, which is an abbreviation of synonym sets. A word usually has multiple synsets, which indicate different meaning or linguistic uses of the word. The semantic relationships are encoded by using hyponyms and hypernyms. From the viewpoint of a certain word, these indicate more specific synsets and more general synsets respectively. For example, a blackbird is a hyponym of a bird, which in turn has a hypernym 'animal' (see figure 2.9). This way, different words with equal linguistic meaning meet each other at higher levels in the semantic tree. We will be using hypernyms to generate our second dataset.

WordNet is offered as a Java API with the `extJWNL`-package (Extended Java WordNet Library) (Autayeu, 2000-2016). This package allowed us to access the WordNet dictionary from our Java programming environment. We

- [S: \(n\) bird](#) (warm-blooded egg-laying vertebrates characterized by feathers and forelimbs modified as wings)
 - [direct hyponym](#) / [full hyponym](#)
 - [part meronym](#)
 - [member holonym](#)
 - [direct hypernym](#) / [inherited hypernym](#) / [sister term](#)
 - [S: \(n\) vertebrate, craniate](#) (animals having a bony or cartilaginous skeleton with a segmented spinal column and a large brain enclosed in a skull or cranium)
 - [S: \(n\) chordate](#) (any animal of the phylum Chordata having a notochord or spinal column)
 - [S: \(n\) animal, animate being, beast, brute, creature, fauna](#) (a living organism characterized by voluntary movement)
 - [S: \(n\) organism, being](#) (a living thing that has (or can develop) the ability to act or function independently)
 - [S: \(n\) living thing, animate thing](#) (a living (or once living) entity)
 - [S: \(n\) whole, unit](#) (an assemblage of parts that is regarded as a single entity) "*how big is that part compared to the whole?*"; "*the team is a unit*"
 - [S: \(n\) object, physical object](#) (a tangible and visible entity; an entity that can cast a shadow) "*it was full of rackets, balls and other objects*"
 - [S: \(n\) physical entity](#) (an entity that has physical existence)
 - [S: \(n\) entity](#) (that which is perceived or known or inferred to have its own distinct existence (living or nonliving))

Figure 2.9: Semantical relations of the synset 'bird' in WordNet

included the library to our workspace and created the `WordNetLink.java`-class to handle the subtitle data. The code can be found in Appendix D.

Some words in our subtitle files serve more than one role in part of speech. For example, a single word can both be considered as a verb or a noun, depending on the context. The word 'attack' is a verb in the phrase '*The soldiers attack the man*', but a noun in the phrase '*The man had an asthma attack*'. The English language knows a couple of hundred words that act this way. We need to know the grammatical role of each word if we want to select its correct synset. A POS(Part of Speech)-Tagger is designed to analyse phrase structure and is able to associate each word with its syntactic role. Because a POS-Tagger works on phrases, we extended our code from `SubtitlePreprocessing` to include POS-tagging. We utilized the POS-Tagger made by the Stanford Natural Language Processing Group (Toutanova, Klein, Manning, & Singer, 2003).

After the lemmatization in the pre-processing phase, we pass the lem-

matized string to the `StanfordPOSTagger.java`-class. Here, the Stanford POS-Tagger assigns the POS-labels to each word by appending the tag in all-caps to the end of the word. A selection of 36 labels exists to effectively label all things that can possibly appear in a text correctly. For our problem, we are only interested in a few. We only included words from the subtitle file with the tags as appearing in table 2. We limited ourselves to these tags, because they retain the most information, which are helpful to train a classifier with. Other tags that exist include (but are not limited to) pronouns, foreign words or wh-determiners, which are usually not very informative.

| Class | POS-Tag | Meaning | Example |
|------------|-------------|--------------------------------------|---------------------------|
| | JJ | Adjective | third, participatory |
| JJ* | JJR | Adjective, comparative | braver, calmer |
| | JJS | Adjective, superlative | bravest, calmest |
| | NN | Noun, singular | cabbage, humour |
| NN* | NNS | Noun, plural | coasts, bodyguards |
| | NNP | Proper noun, singular | London, Trump |
| | NNPS | Proper noun, plural | Antilles, Americans |
| | RB | Adverb | occasionally, prominently |
| RB* | RBR | Adverb, comparative | larger, higher |
| | RBS | Adverb, superlative | largest, highest |
| | VB | Verb, base form | ask, believe |
| VB* | VBD | Verb, past tense | speculated, registered |
| | VBG | Verb, gerund or present participle | focusing, traveling |
| | VCN | Verb, past principle | experimented, desired |
| | VBP | Verb, non3rd person singular present | comprise, emphasize |
| | VBZ | Verb, 3rd person singular present | stretches, seduces |

Table 2: POS-tag labels that were kept during pre-processing.

The resulting `.arff`-data file is added to the `extJWNL`-project. This class contains a file reader, a file writer and all the tools `extJWNL` provides to handle WordNet data structures. It copies the data file header and labels of each instance to a new file. For all the tagged words that appear in the subtitle attribute, the program will match it with the first synset of the corresponding syntactic category. We made this choice because the first one usually covers the intention of the word best, often being the most used definition. From this synset, all the direct hypernyms are selected. These words are appended to the correct instance in the new data file. All hypernyms that consist of multiple words will be concatenated by an underline symbol such that they become one term. Otherwise, the data pre-processing tools would consider them as individual words. To prepare the second dataset for classification, the same steps that were discussed in sections 2.4.2 and 2.4.3 were executed.

2.6 Classification algorithms

Binary Relevance

In multi-label classification, binary relevance is the most intuitive classification algorithm (Zhang, Li, Liu, & Geng, 2018). It reduces the multi-label problem to a single-label one by training a set of single-label classifiers, one for each class (genre label). Each of these classifiers predicts the absence or presence of a genre label and returns the union of all these decisions as multi-label output. One major downside of binary relevance is that it ignores label correlations that appear in the train data during the transformation process. However, binary relevance has low computational complexity compared to other methods, scaling linearly with the size of the label set. Aside from the low computational complexity, it is argued that binary relevance methods are competitive in performance and efficacy compared to other state-of-the-art classification methods in machine learning (Luaces, Díez, Barranquero, del Coz, & Bahamonde, 2012). Binary Relevance is often used as benchmark to test the performance of other classification algorithms.

Probabilistic Classifier Chains

Classifier Chains are built on the basic principle of binary relevance methods to eliminate their correlation problem, while keeping an acceptable computational complexity for large-scale problems, designed by Read, Pfahringer, Holmes, and Frank (2009). The model contains as many binary classifiers as in binary relevance (one for each label) and are linked along a chain $(C_1, \dots, C_{|L|})$. Instead of solely solving the single-label classification problem at a current position in the chain, it also considers all decisions that were made in the preceding stages of the same chain. Hence, the classification progress begins at the first classifier and propagates along the chain, where each subsequent classifier computes conditional probabilities augmented by the previous classifiers. The algorithm keeps the label correlations into account, while having just a negligible worse runtime complexity compared to binary relevance methods. Dembczynski et al. suggested a novel method in multi-label classification, which was designed from a probabilistic perspective on classifier chains (Cheng, Hüllermeier, & Dembczynski, 2010). Their Probabilistic Classifier Chains (PCCs) estimate the entire joint distribution of labels to make better classification decisions. While they proved that PCCs outperform regular CCs, it should be noted that these require $2^{|L|}$ paths and are therefore computationally infeasible for problems with a large number of labels. Because we only consider seven labels ($|L| = 7$), the com-

putational complexity will not be an issue. Hence, we think that the PCC classifier will provide better results than regular CCs.

Label Powerset

Label Powerset (LP) is a classification method that transforms the multi-label classification problem into a single multi-class classification problem. Note that this is different from a multi-label classification problem; a multi-class problem assigns one class to each instance from a set of three or more classes. It assigns each unique possible set of labels to an individual class. That means that there will be $2^{|L|}$ classes, with $|L|$ being the number of original labels. This poses complexity issues for a multi-label problem with many labels, but for our problem it will be manageable. For each test instance, the LP algorithm will output the most probable class from a probability distribution over all classes, which is actually a set of labels. The label powerset algorithm will also take label correlations into account. One downside to this algorithm is that a few set of labels will occur very infrequently, because not all combinations of labels occur in our problem (also see chapter 5).

Base Classifiers

All these multi-label classification algorithm use a base classifier to base decisions on. The Naive Bayes classifier appears most in the literature as baseline for text-based data. This probabilistic classifier is based on Bayes' theorem with independence assumptions between features. It has linear time complexity and is able to weigh up against other base classifiers in the domain of textual data. It is a popular classifier because it only requires one sweep through the training data, therefore it is very fast. However, it has modest accuracy compared to other classifiers (Godbole & Sarawagi, 2004).

Another base classifier that will be tested is RandomForest. Instead of splitting nodes based on some most important feature, this base classifier constructs multiple decision trees during runtime based on some randomness and searches for the best feature among those (Breiman, 2001). The diversity this brings usually promises a very good classification evaluation. The Random Forest algorithm is robust against the general decision tree's tendency of overfitting to training data (Friedman, Hastie, & Tibshirani, 2008).

The final base classifier that will be used are based on Support Vector Machines (SVMs). SVMs use structural risk minimization by finding a hypothesis h for which can be guaranteed to make the lowest true error

(probability that h will make an error on a randomly selected test sample). SVMs have the property to be independent of the dimensionality of the feature space. They were proved to be very robust, eliminating the need for expensive parameter tuning (Joachims, 1998). SVMs have quadratic complexity in terms of training examples (Godbole & Sarawagi, 2004). Because of all these properties, just as we have seen in the literature research, SVMs were found to work very well on text categorization problems.

2.7 Evaluation Measures

The evaluation measures that are used in single-label classification cannot be easily transferred to multi-label problems. In fact, misclassifications in the multi-label domain are not simply about a hard right or wrong; predictions containing a subset of the ground truth labels should be evaluated better than predictions with no correct labels at all. Many different evaluation measures exist to express different aspects of classification performance (Maimon & Rokach, 2010), (Sebastiani, 2002), (Sorower, 2010).

For our problem description, we decided to use the micro-averaged F_1 evaluation measure. This measure is adapted for a multi-label classification problem and takes both precision and recall scores in consideration. There are four outcomes for a certain decision, which are *true positive* (TP, positive label is correctly classified), *true negative* (TN, negative label is correctly classified), *false positive* (FP, Type I-error, negative label is faulty predicted positive) and *false negative* (FN, Type II-error, positive label is faulty predicted negative). Precision is the ratio of correct positive predictions to the total predicted positives, averaged over all instances (see formula 3).

$$P_{avg} = \frac{TP_{avg}}{TP_{avg} + FP_{avg}} \quad (3)$$

Recall is the ratio of correct positive predictions to the total amount of positives, averaged over all instances (see formula 4). In the output files, micro/macro precision and -recall values are reported, but also as precision and recall per label. We prefer to discuss these values as they display any potential under-/overfitting of certain labels.

$$R_{avg} = \frac{TP_{avg}}{TP_{avg} + FN_{avg}} \quad (4)$$

The F_1 -score is then computed by taking the harmonic average of both precision and recall evaluation measures (see formula 5). It provides a balanced score between precision and recall. A micro-averaging metric is chosen

because it will aggregate the contributions of all classes to compute the average, while a macro-averaging metric just takes the average over all classes. For our problem micro-averaging is preferred, as classes are not balanced perfectly (not all genre classes are distributed equally).

$$F_{1,micro} = 2 * \frac{P_{avg} * R_{avg}}{P_{avg} + R_{avg}} \quad (5)$$

Finally, we decided on reporting two more evaluation measures that tell us something about the amount of (in)correctly classified labels. The *exact match* score is considered a very harsh evaluation, because it shows the percentage of label sets that are completely correctly classified with respect to the ground truth labels (see formula 6). Here, $I(true) = 1$, $I(false) = 0$ and n denotes the amount of labels. The hamming loss is also frequently reported in the literature, which defines the percentage of total incorrectly classified labels (see formula 7), where Δ is the symmetric difference of label sets. Both (6) and (7) are example based, and are averaged over all test instances.

$$MR = \frac{1}{n} \sum_{i=1}^n I(Y_i = Z_i) \quad (6)$$

$$HL = \frac{1}{n} \sum_{i=1}^n \frac{Y_i \Delta Z_i}{N} \quad (7)$$

Chapter 3

Results

Our four experimental conditions are split between the original dataset of normalized word frequencies, with or without TF-IDF-transformation and the WordNet dataset of direct hypernyms, also with or without TF-IDF-transformation.

They were tested across nine multi-label classification algorithm x base classifier combinations. The multi-label classification algorithms are Binary Relevance (BR), Probabilistic Classifier Chains (PCC) and Label Powerset (LP). The base classifiers are Naive Bayes (Bayes), Support Vector Machines (SVM) and Random Forest (RF). To evaluate the performance of the classification process, we report F_1 -score, hamming-loss and exact match. All precision and recall scores per label for each test condition, including run times, are fully listed in appendix A. The results are listed in table 3.

| | | IDF0-WN0 (A=791) | | | IDF1-WN0 (A=728) | | | IDF0-WN1 (A=703) | | | IDF1-WN1 (A=685) | | |
|-----|-------|------------------|--------------|--------------|------------------|--------------|--------------|------------------|--------------|--------------|------------------|--------------|--------------|
| | | F1 | EM | HL | F1 | EM | HL | F1 | EM | HL | F1 | EM | HL |
| BR | Bayes | 0,784 | 0,127 | 0,268 | 0,796 | 0,138 | 0,256 | 0,765 | 0,124 | 0,286 | 0,790 | 0,129 | 0,261 |
| | SVM | 0,856 | 0,233 | 0,192 | 0,842 | 0,206 | 0,210 | 0,849 | 0,226 | 0,201 | 0,846 | 0,210 | 0,204 |
| | RF | 0,855 | 0,188 | 0,204 | 0,857 | 0,196 | 0,201 | 0,854 | 0,183 | 0,205 | 0,856 | 0,191 | 0,203 |
| PCC | Bayes | 0,785 | 0,128 | 0,267 | 0,796 | 0,137 | 0,256 | 0,765 | 0,124 | 0,286 | 0,789 | 0,129 | 0,261 |
| | SVM | 0,853 | 0,243 | 0,195 | 0,837 | 0,210 | 0,216 | 0,846 | 0,249 | 0,205 | 0,843 | 0,215 | 0,208 |
| | RF | 0,857 | 0,215 | 0,200 | 0,860 | 0,221 | 0,196 | 0,855 | 0,196 | 0,203 | 0,852 | 0,196 | 0,207 |
| LP | Bayes | 0,820 | 0,207 | 0,240 | 0,809 | 0,186 | 0,251 | 0,803 | 0,164 | 0,261 | 0,789 | 0,155 | 0,267 |
| | SVM | 0,854 | 0,289 | 0,193 | 0,858 | 0,292 | 0,188 | 0,841 | 0,268 | 0,210 | 0,847 | 0,271 | 0,203 |
| | RF | 0,826 | 0,246 | 0,229 | 0,832 | 0,248 | 0,223 | 0,823 | 0,223 | 0,233 | 0,824 | 0,227 | 0,232 |

Table 3: Classification performance scores for the four experimental conditions. The first two columns represent the original dataset, without and with TF-IDF-transformation respectively, the last two columns represent the WordNet dataset, without and with TF-IDF-transformation respectively. The variable A next to the headers indicates attribute vector lengths. The best scores for each evaluation measure are shown in bold.

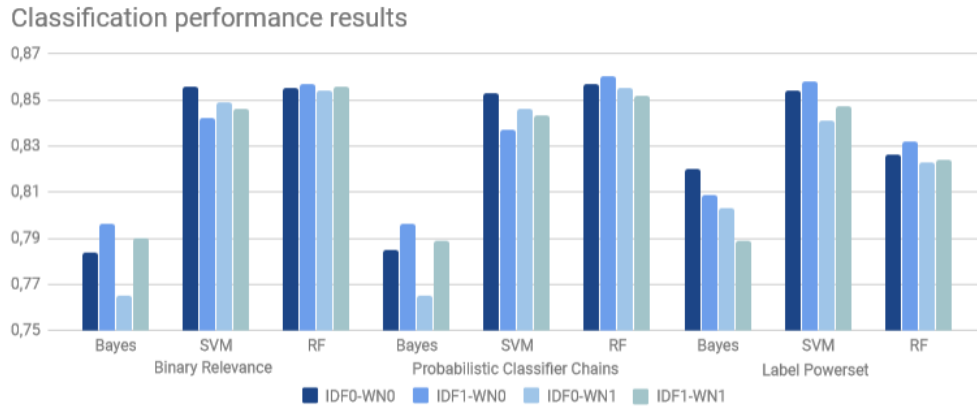


Figure 3.1: Performance results (F_1 -scores) of all classification algorithms. The four datasets are reported in different shades of blue (see legend).

The overall classification results are shown in figure 3.1. Everything was tested using 10-fold cross-validation. The probabilistic classifier chains in combination with the Random Forest base classifier performed best across three datasets. The original dataset composed of normalized word frequencies scored 85,6% and the equivalent dataset with TF-IDF-transformation scored a F_1 -score of 86,0%. This was also the highest performance score overall. For the WordNet datasets, the one composed of normalized word frequencies scored 85,5% with the Probabilistic Classifier Chains and RF base classifier, matching the other dataset without TF-IDF-transformation. The other WordNet dataset (with TF-IDF-transformation) scored an equivalent score of 85,6% with the Binary Relevance algorithm and RF base classifier.

The Naive Bayes classifier had the lowest F_1 -score for all multi-label classification algorithms. The scores are similar on the Binary Relevance algorithm and on PCCs, but offers a higher baseline on the Label Powerset algorithm (except for the WordNet dataset with TF-IDF-transformation, where all three multi-label classification algorithms perform the same). The Random Forest base classifier outperforms the Support Vector Machines on the Binary Relevance algorithm and PCCs, although the effect difference is marginal. For the label powerset algorithm SVM base classifier performs better. However, the Random Forest base classifier in that condition cannot match the performance of the other two multi-label classifiers.

Averaged across datasets, the three best scoring algorithmic combinations were BR/RF (85,55%), PCC/RF (85,60%) and LP/SVM (85,00%). Averaged across algorithms, the dataset with normalized word frequencies scored 83,22% and the same dataset with TF-IDF-transformation scored 83,19%. The WordNet without and with TF-IDF-transformations scored 82,23% and 82,62% on their F_1 -scores respectively.

The F_1 -scores were computed by taking the harmonic mean between precision and recall scores. To visualize how these scores were established, we give the distribution of precision/recall scores for each individual label. Because the distribution of precision/recall scores were equivalent over dataset tests, we provide the average over all instances in figure 3.2.

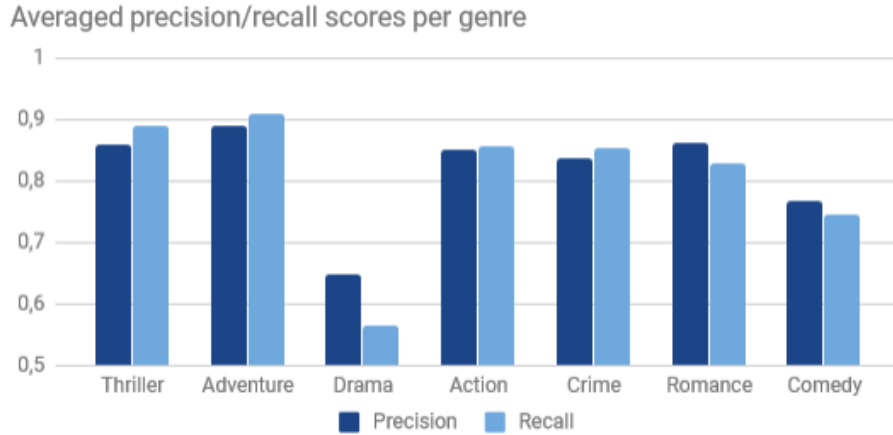


Figure 3.2: Precision/recall-scores, averaged over all classification algorithms and experimental conditions.

The thriller and adventure genres scored best on precision and recall. The action and crime genres scored slightly lower. These four genres had higher recall than precision. The romance genre scored similar to the action and crime genres, but had higher precision than recall. Comedy scored below-average, but the drama genre evidently scored worst on precision and recall.

Scores that stand out are the recall scores from the RF base classifier on six genres, which were significantly higher than recall scores from other algorithmic combinations. The effect was strongest with the Binary Relevance and PCC algorithms, but also appeared in the Label Powerset algorithm to a lower extent. On the contrary, the drama genre experienced the lowest recall scores across all tests with RF. The precision scores were also lowest for the drama genre, but they were higher than the recall rates.

The final evaluation metrics measured were the exact match ratio and hamming loss. The distribution of these scores can be seen in figure 3.3.

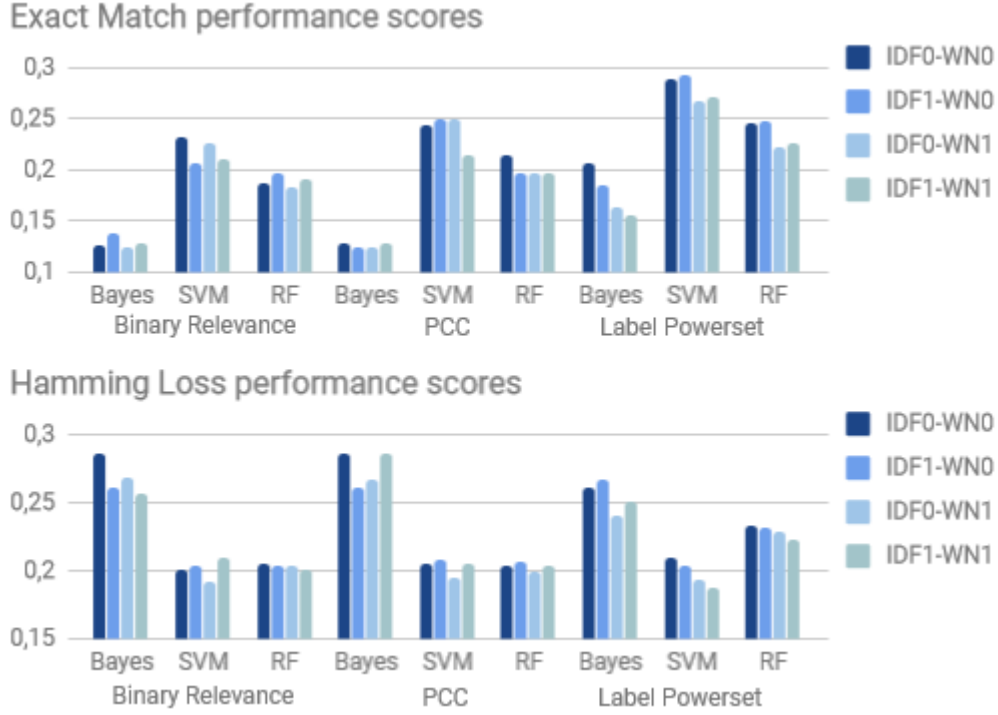


Figure 3.3: Performance scores for exact match ratio (higher is better) and hamming loss (lower is better) respectively.

The Naive Bayes classifier acted as baseline for the tests, as can be seen in figure 3.3. The average exact match ratios are 12,95% and 12,63% for BR and PCC algorithms respectively. It scores slightly better on the Label Powerset tests (17,80%). The same trend for the Naive Bayes classifier can be seen for the hamming loss metric (26,78%, 27,50% and 25,48% respectively).

Support Vector Machine base classifiers seem to outperform Random Forest classifiers on the exact match ratio across tests. The best performance was seen for the SVM/LP algorithmic combination with the original TF-IDF-transformed dataset, with an exact match ratio of 29,20%. SVM and RF classifiers scored equal hamming-loss scores for BR and PCC multi-label algorithms, but SVM scored better with the Label Powerset algorithm than RF, with an minimal (optimal) score of 18,80% on hamming-loss overall.

Chapter 4

Conclusions

1. The maximized classification performance (F_1 -scores) for each experimental condition were consistent and did not differ from each other.

The results show that text-based features can be effectively used for the multi-label classification problem of movie genres. Everything from data extraction, pre-processing and classification takes up to a few seconds, exploiting the computational simplicity of analysing text-based data. The most efficient classification algorithms scored between 85,5% and 86% on classification performance across our four experimental conditions, which are numbers we are very content with.

2. The *drama* genre was significantly more difficult to classify than others.

Although the classification performance results were good, they also fell short by the performance of the drama genre. While some genre labels managed to score precision/recall scores over 90%, the classification algorithms were not able to capture the word variation in the drama genre effectively, with an averaged precision/recall score of 60%. Some algorithms even scored below chance for this movie genre.

3. Neither WordNet nor TF-IDF-transformation on the feature vectors of subtitle words had a significant effect on classification performance.

Although our classification algorithms performed very well for determining genre labels, we have to conclude that there exists no significant difference in performance between the TF-IDF-transformed datasets and the regular and WordNet datasets. They did not lead to an improved or reduced classification accuracy. Instead, the classification algorithms maximizing performance did not perform differently overall.

4. Based on F_1 -performance, PCCs with RF scored best for most datasets.

Probabilistic Classifier Chains in combination with the Random Forest base classifier proved to be most accurate in three out of four datasets, the Random Forest in conjunction with the Binary Relevance algorithm scored similar results in the fourth dataset.

5. Based on the performance over all evaluation measures, including single label precision/recall scores, hamming-loss and exact match scores, LP with SVMs were most effective.

The Support Vector Machine classifiers performed better than the Random Forest classifier on maximizing the exact match score and minimizing the hamming-loss score. Overall, the most useful classification algorithm and base classifier combination are SVMs in combination with the Label Power-set algorithm for multi-label classification. While it wasn't the absolute top scoring classifier in F_1 -score, the performance was still very good (84-86%, depending on the dataset). This trade-off in F_1 -score led to a very significant increase in exact-match and hamming-loss performance, which no other classifiers could beat. This translates to a bigger proportion of genres being correctly classified relative to the proportion of genres being incorrectly classified.

Chapter 5

Discussion

In the previous chapter, we concluded that Random Forests achieved the best scores for the F_1 -measure, but that SVMs would provide a better classifier in general considering exact-match and hamming-loss evaluation measures. The observation that SVM outperforms RF is also reflected in the recall scores for both classifiers, which might even give us an explanation for the under-performance of the recall scores of the drama genre in general.

The Random Forest classifier scored an average of 48,54% on recall for the drama genre, while the SVM classifier scored an average of 63,88% on recall for the drama genre. From the literature (Breiman (2001) and Friedman et al. (2008)) we learned that RF is robust to overfitting. But what about underfitting? It might be the case that RF is not complex enough to capture the trends in the data of the drama genre. After all, SVMs perform significantly better and are not based on decision trees. Another observation is that the most informative words in the drama genre are not representative for the genre. If we take another look at figure 2.8, we see that a lot of these words are predictive for one or more genres, but the words shown all appear with roughly the same frequency in movies with the drama genre. This is an intrinsic property of the genre itself, as there are not many words that could indicate the appearance of the drama genre without also making an appearance in others.

The binary relevance algorithm also captured the classification performance well, although the use of it is discouraged as label correlations are not taken into account (Zhang et al., 2018). Probabilistic Chain Classifier were created to overcome this issue, but both algorithms show similar results in our research. This observation can be interpreted as our labels not having any (or not many) correlations between genres. Although some genres

co-occur more frequently with certain others (see table 1 for reference), each instance is accompanied with a stand-alone set of genres. Another thing that should be mentioned is that we have chosen to follow the IMDb genre classifications as ground-truth labels, but other genre classifications might exist that differ from IMDb's interpretation.

The WordNet dataset adaptation did not offer the expected results. Instead, the performance did not change compared to other datasets. In hindsight we think that the direct hypernyms did not offer the distinctive power as hypothesised, because the direct hypernyms might be a step too short to observe matches from other words. Higher-order levels of hypernyms or WordNet domains might be the solution for improved classification performance, but unfortunately did not reach the scope of our research.

A reason for the lack of effect between TF-IDF conditions is that WEKA's attribute selection filter is an effective measure for relevance too, which has proven to work both on datasets with and without TF-IDF-transformation.

Our research is subject to a number of limitations. First, the scalability of our results cannot be easily determined. Different results might arise when following the same methodology with smaller or larger datasets. The Label Powerset algorithm becomes more of a problem computationally when the number of labels is increased. Another limitation of this algorithm, is that it considers all possible combinations of labels - while some of those combinations do not exist in the dataset of our multi-label classification problem. Second, the genres were not equally distributed across the dataset, which lead to majority and minority classes. Because the magnitude of differences between classes was not that great, we have not balanced any classes. The classification performance might improve without class imbalance. Third, we did not experiment with parameter tweaking of the classification algorithms thoroughly, which might also lead to a classification performance boost.

Future research might explore the effects of parameter tweaking, extended WordNet domain or higher-order hypernym transformation on classification performance. The classification of text-based features could also be extended to other forms of digital video. One could attempt to build a recommender system based on predicted genre labels, or wrap all functionalities described in this thesis into one application.

From our discussion it is understood that there is no single method or classification algorithm that best fits a collection of problems in machine learning. Algorithms and classifiers perform differently depending on size, structure and nature of data it is given. However, the Support Vector Machine classifiers can be applied successfully to a large number of problems in the domain of textual classification tasks, one of which is genre classification.

References

- Autayeu, A. (2000-2016). *extJWNL: Extended Java WordNet Library*. (Retrieved from: <http://extjwnl.sourceforge.net/>)
- Beel, J., Breiteringer, C., Gipp, B., & Langer, S. (2016). Research-paper recommender systems: a literature survey. *International Journal on Digital Libraries*, 17(4), 305–338.
- Bird, S., Klein, E., & Loper, E. (2009). *Natural Language Processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc. (Retrieved from: <https://gist.github.com/sebleier/554280>)
- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5–32.
- Brezeale, D., & Cook, D. (2006). Using closed captions and visual features to classify movies by genre. *Poster session of the seventh international workshop on Multimedia Data Mining (MDM/KDD2006)*, 1–5.
- Brezeale, D., & Cook, D. (2008). Automatic video classification: A survey of the literature. *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, 38(3), 416–430.
- Chao, B., & Sirmorya, A. (2016). Automated movie genre classification with LDA-based topic modeling. *International Journal of Computer Applications*, 145(13), 1–5.
- Cheng, W., Hüllermeier, E., & Dembczynski, K. (2010). Bayes optimal multilabel classification via probabilistic classifier chains. In (pp. 279–286).
- Follows, S. (2018). *Genre trends in global film production*. Retrieved from <https://stephenfollows.com/genre-trends-global-film-production>
- Frank, E., Hall, M., & Witten, I. (2016). *The WEKA workbench. Online appendix for "Data Mining: Practical Machine Learning Tools and Techniques"* (fourth ed.). Morgan Kaufmann, Burlington, Massachusetts. (Retrieved from: <https://www.cs.waikato.ac.nz/ml/weka>)
- Friedman, J., Hastie, T., & Tibshirani, R. (2008). The elements of statistical

- learning: Data mining, inference and prediction. In (second ed., pp. 587–588). New York: Springer.
- Godbole, S., & Sarawagi, S. (2004). Discriminative methods for multi-labeled classification. In *Pacific-asia conference on knowledge discovery and data mining* (pp. 22–30). Springer, Berlin, Heidelberg.
- Grootjen, F. (2018). *CleanUpSubtitles*. (Written in Java)
- Joachims, T. (1998). Text categorization with support vector machines: Learning with many relevant features. In (pp. 137–142). Springer, Berlin, Heidelberg.
- Kantrowitz, M. (1994). *Name corpus: List of male, female and pet names*. (Version 1.3., retrieved from: <http://www.cs.cmu.edu/Groups/AI/util/areas/nlp/corpora/names/>)
- Katsioulis, P., Tsetsos, V., & Hadjiefthymiades, S. (2007). Semantic video classification based on subtitles and domain terminologies. In *KAMC*.
- Korde, V., & Mahender, C. (2012). Text classification and classifiers: a survey. *International Journal of Artificial Intelligence & Applications*, 3(2), 85–99.
- Lee, C. (2017). *Text-based video genre classification using multiple feature categories and categorization methods* (Master’s thesis). Departement of Communication and Information Sciences, Tilburg University.
- Luaces, O., Díez, J., Barranquero, J., del Coz, J., & Bahamonde, A. (2012). Binary relevance efficacy for multilabel classification. *Progress in Artificial Intelligence*, 1(4), 303–313.
- Maimon, O., & Rokach, L. (2010). Data mining and knowledge discovery handbook. In (2nd ed., p. 667–685). Springer, New York.
- Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., & McClosky, D. (2014). The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the 52nd annual meeting of the association for computational linguistics: System demonstrations* (pp. 55–60).
- Miller, G. (1995). WordNet: a lexical database for english. *Communications of the ACM*, 38(11), 39–41.
- Rasheed, S. Y., Z., & Shah, M. (2005). On the use of computable features for film classification. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(1), 52–64.
- Read, J., Pfahringer, B., Holmes, G., & Frank, E. (2009). Classifier chains for multi-label classification. In *Joint european conference on machine learning and knowledge discovery in databases* (pp. 254–269).
- Read, J., Reutemann, P., Pfahringer, B., & Holmes, G. (2016). MEKA: A multi-label/multi-target extension to WEKA. *Journal of Machine*

- Learning Research*, 17(21), 1–5. Retrieved from <http://jmlr.org/papers/v17/12-164.html>
- Sebastiani, F. (2002). Machine learning in automated text categorization. *ACM Computing Surveys (CSUR)*, 34(1), 1–47.
- Sharma, D., & Jain, S. (2015). Evaluation of stemming and stop word techniques on text classification problem. *International Journal of Scientific Research in Computer Science and Engineering*, 3(2), 1–4.
- Sorower, M. (2010). A literature survey on algorithms for multi-label learning. *Oregon State University, Corvallis*, 18, 1–25.
- Toutanova, K., Klein, D., Manning, C., & Singer, Y. (2003). Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 conference of the north american chapter of the association for computational linguistics on human language technology-volume 1* (pp. 173–180).
- Waldfoegel, J. (2017). How digitization has created a golden age of music, movies, books, and television. *Journal of Economic Perspectives*, 31(3), 195–214.
- Yang, Y., & Pedersen, J. (1997). A comparative study on feature selection in text categorization. *International Conference on Machine Learning*, 97(35), 412–420.
- Zhang, M., Li, Y., Liu, X., & Geng, X. (2018). Binary relevance for multi-label learning: an overview. *Frontiers of Computer Science*, 12(2), 191–202.
- Zhu, W., Toklu, C., & Liou, S. (2001). Automatic news video segmentation and categorization based on closed-captioned text. In *ICME* (pp. 1–4).

Appendices

Appendix A: Raw data (results)

Abbreviations:

- Datasets:
 - $IDF\{0/1\}$: TF-IDF-transformation no/yes.
 - $WN\{0/1\}$: WordNet hypernym dataset no/yes.
- Classification algorithms:
 - BR: Binary Relevance.
 - PCC: Probabilistic Classifier Chains.
 - LP: Label Powerset.
 - BAYES: Naive Bayes base classifier.
 - SVM: Support Vector Machine base classifier.
 - RF: Random Forest base classifier.
- Evaluation methods:
 - TT: Total runtime (averaged across folds, in seconds)
 - EM: Exact Match (higher is better)
 - HL: Hamming Loss (lower is better)
 - F1: F_1 -score (higher is better)

| | TT | EM | HL | F1 | THRILLER | ADVENTURE | DRAMA | ACTION | CRIME | ROMANCE | COMEDY |
|--------------------|------|-------|-------|-------|----------|-----------|-------|--------|-------|---------|--------|
| IDF0-WN0-BR-BAYES | 3,8 | 0,127 | 0,268 | 0,784 | 0,901 | 0,914 | 0,564 | 0,841 | 0,843 | 0,919 | 0,743 |
| IDF0-WN0-BR-SVM | 29,4 | 0,233 | 0,192 | 0,856 | 0,819 | 0,82 | 0,505 | 0,823 | 0,759 | 0,683 | 0,595 |
| IDF0-WN0-BR-RF | 29,1 | 0,188 | 0,204 | 0,855 | 0,865 | 0,909 | 0,691 | 0,867 | 0,86 | 0,859 | 0,803 |
| IDF0-WN0-PCC-BAYES | 65,7 | 0,128 | 0,267 | 0,785 | 0,903 | 0,932 | 0,66 | 0,886 | 0,867 | 0,878 | 0,795 |
| IDF0-WN0-PCC-SVM | 35,2 | 0,243 | 0,195 | 0,853 | 0,809 | 0,841 | 0,737 | 0,822 | 0,809 | 0,803 | 0,762 |
| IDF0-WN0-PCC-RF | 57,8 | 0,215 | 0,2 | 0,857 | 0,982 | 0,997 | 0,493 | 0,924 | 0,945 | 0,969 | 0,853 |
| IDF0-WN0-LP-BAYES | 3,3 | 0,207 | 0,24 | 0,82 | 0,901 | 0,914 | 0,564 | 0,841 | 0,843 | 0,918 | 0,743 |
| IDF0-WN0-LP-SVM | 13,4 | 0,289 | 0,193 | 0,854 | 0,819 | 0,82 | 0,505 | 0,823 | 0,76 | 0,685 | 0,597 |
| IDF0-WN0-LP-RF | 7,9 | 0,246 | 0,229 | 0,826 | 0,865 | 0,911 | 0,689 | 0,865 | 0,868 | 0,865 | 0,797 |
| IDF1-WN0-BR-BAYES | 3,4 | 0,138 | 0,256 | 0,796 | 0,903 | 0,93 | 0,647 | 0,88 | 0,871 | 0,855 | 0,781 |
| IDF1-WN0-BR-SVM | 38,8 | 0,206 | 0,21 | 0,842 | 0,814 | 0,847 | 0,765 | 0,823 | 0,813 | 0,817 | 0,77 |
| IDF1-WN0-BR-RF | 28,3 | 0,196 | 0,201 | 0,857 | 0,976 | 0,994 | 0,493 | 0,923 | 0,94 | 0,96 | 0,843 |
| IDF1-WN0-PCC-BAYES | 60,9 | 0,124 | 0,286 | 0,765 | 0,882 | 0,897 | 0,533 | 0,871 | 0,809 | 0,849 | 0,753 |
| IDF1-WN0-PCC-SVM | 41,4 | 0,249 | 0,205 | 0,846 | 0,868 | 0,899 | 0,632 | 0,816 | 0,87 | 0,851 | 0,702 |
| IDF1-WN0-PCC-RF | 54,2 | 0,196 | 0,203 | 0,855 | 0,869 | 0,92 | 0,687 | 0,882 | 0,854 | 0,878 | 0,799 |
| IDF1-WN0-LP-BAYES | 2,9 | 0,186 | 0,251 | 0,809 | 0,884 | 0,936 | 0,63 | 0,861 | 0,868 | 0,863 | 0,806 |
| IDF1-WN0-LP-SVM | 13,1 | 0,292 | 0,188 | 0,858 | 0,817 | 0,874 | 0,646 | 0,861 | 0,846 | 0,897 | 0,755 |
| IDF1-WN0-LP-RF | 7,5 | 0,248 | 0,223 | 0,832 | 0,934 | 0,969 | 0,435 | 0,827 | 0,862 | 0,744 | 0,769 |
| | | | | | 0,906 | 0,92 | 0,587 | 0,862 | 0,846 | 0,9 | 0,776 |
| | | | | | 0,813 | 0,796 | 0,63 | 0,807 | 0,776 | 0,701 | 0,68 |
| | | | | | 0,861 | 0,914 | 0,66 | 0,858 | 0,838 | 0,861 | 0,787 |
| | | | | | 0,871 | 0,911 | 0,666 | 0,86 | 0,84 | 0,87 | 0,784 |
| | | | | | 0,807 | 0,843 | 0,74 | 0,82 | 0,808 | 0,81 | 0,771 |
| | | | | | 0,984 | 0,998 | 0,521 | 0,929 | 0,94 | 0,958 | 0,862 |
| | | | | | 0,903 | 0,91 | 0,58 | 0,851 | 0,85 | 0,882 | 0,741 |
| | | | | | 0,765 | 0,791 | 0,503 | 0,789 | 0,716 | 0,66 | 0,578 |
| | | | | | 0,855 | 0,898 | 0,684 | 0,847 | 0,859 | 0,875 | 0,775 |
| | | | | | 0,898 | 0,936 | 0,612 | 0,88 | 0,883 | 0,845 | 0,749 |
| | | | | | 0,813 | 0,849 | 0,749 | 0,83 | 0,808 | 0,816 | 0,755 |
| | | | | | 0,978 | 0,994 | 0,483 | 0,928 | 0,93 | 0,956 | 0,833 |
| | | | | | 0,871 | 0,886 | 0,548 | 0,834 | 0,817 | 0,838 | 0,781 |
| | | | | | 0,883 | 0,875 | 0,623 | 0,826 | 0,812 | 0,816 | 0,675 |
| | | | | | 0,873 | 0,923 | 0,672 | 0,881 | 0,871 | 0,869 | 0,807 |
| | | | | | 0,886 | 0,941 | 0,639 | 0,869 | 0,891 | 0,867 | 0,803 |
| | | | | | 0,824 | 0,874 | 0,644 | 0,858 | 0,845 | 0,888 | 0,775 |
| | | | | | 0,944 | 0,964 | 0,459 | 0,821 | 0,864 | 0,773 | 0,789 |

Table 4a: Results (raw data). For each experimental condition, the first row is precision per label, second row is recall per label

| | TT | EM | HL | F1 | THRILLER | ADVENTURE | DRAMA | ACTION | CRIME | ROMANCE | COMEDY |
|--------------------|------|-------|-------|-------|----------|-----------|-------|--------|-------|---------|--------|
| IDF0-WN1-BR-BAYES | 3,5 | 0,124 | 0,286 | 0,765 | 0,903 | 0,91 | 0,58 | 0,852 | 0,849 | 0,882 | 0,74 |
| IDF0-WN1-BR-SVM | 36 | 0,226 | 0,201 | 0,849 | 0,765 | 0,791 | 0,502 | 0,791 | 0,715 | 0,658 | 0,574 |
| IDF0-WN1-BR-RF | 28,8 | 0,183 | 0,205 | 0,854 | 0,855 | 0,897 | 0,689 | 0,854 | 0,855 | 0,861 | 0,774 |
| IDF0-WN1-PCC-BAYES | 60,9 | 0,124 | 0,286 | 0,765 | 0,898 | 0,94 | 0,619 | 0,878 | 0,889 | 0,867 | 0,784 |
| IDF0-WN1-PCC-SVM | 41,4 | 0,249 | 0,205 | 0,846 | 0,807 | 0,848 | 0,724 | 0,824 | 0,811 | 0,807 | 0,749 |
| IDF0-WN1-PCC-RF | 54,2 | 0,196 | 0,203 | 0,855 | 0,981 | 0,993 | 0,509 | 0,927 | 0,937 | 0,961 | 0,852 |
| IDF0-WN1-LP-BAYES | 3 | 0,164 | 0,261 | 0,803 | 0,903 | 0,91 | 0,58 | 0,851 | 0,85 | 0,882 | 0,741 |
| IDF0-WN1-LP-SVM | 13,8 | 0,268 | 0,21 | 0,841 | 0,765 | 0,791 | 0,503 | 0,789 | 0,716 | 0,66 | 0,578 |
| IDF0-WN1-LP-RF | 7,8 | 0,223 | 0,233 | 0,823 | 0,855 | 0,898 | 0,684 | 0,847 | 0,859 | 0,875 | 0,775 |
| IDF1-WN1-BR-BAYES | 3,4 | 0,129 | 0,261 | 0,79 | 0,898 | 0,936 | 0,612 | 0,88 | 0,883 | 0,845 | 0,749 |
| IDF1-WN1-BR-SVM | 61,7 | 0,21 | 0,204 | 0,846 | 0,813 | 0,849 | 0,749 | 0,83 | 0,808 | 0,816 | 0,755 |
| IDF1-WN1-BR-RF | 27,9 | 0,191 | 0,203 | 0,856 | 0,978 | 0,994 | 0,483 | 0,928 | 0,93 | 0,956 | 0,833 |
| IDF1-WN1-PCC-BAYES | 59,9 | 0,129 | 0,261 | 0,789 | 0,873 | 0,892 | 0,523 | 0,852 | 0,805 | 0,822 | 0,723 |
| IDF1-WN1-PCC-SVM | 60 | 0,215 | 0,208 | 0,843 | 0,865 | 0,879 | 0,607 | 0,804 | 0,804 | 0,849 | 0,679 |
| IDF1-WN1-PCC-RF | 56,5 | 0,196 | 0,207 | 0,852 | 0,862 | 0,899 | 0,649 | 0,869 | 0,851 | 0,875 | 0,776 |
| IDF1-WN1-LP-BAYES | 2,9 | 0,155 | 0,267 | 0,798 | 0,888 | 0,917 | 0,598 | 0,858 | 0,871 | 0,858 | 0,751 |
| IDF1-WN1-LP-SVM | 13,8 | 0,271 | 0,203 | 0,847 | 0,817 | 0,877 | 0,615 | 0,853 | 0,845 | 0,893 | 0,739 |
| IDF1-WN1-LP-RF | 7,7 | 0,227 | 0,232 | 0,824 | 0,927 | 0,959 | 0,428 | 0,826 | 0,874 | 0,762 | 0,751 |
| | | | | | 0,919 | 0,92 | 0,589 | 0,866 | 0,854 | 0,905 | 0,784 |
| | | | | | 0,775 | 0,759 | 0,61 | 0,798 | 0,789 | 0,675 | 0,709 |
| | | | | | 0,88 | 0,918 | 0,667 | 0,872 | 0,859 | 0,871 | 0,78 |
| | | | | | 0,805 | 0,85 | 0,72 | 0,819 | 0,819 | 0,803 | 0,759 |
| | | | | | 0,986 | 0,994 | 0,549 | 0,931 | 0,937 | 0,963 | 0,839 |
| | | | | | 0,919 | 0,92 | 0,589 | 0,865 | 0,853 | 0,906 | 0,781 |
| | | | | | 0,775 | 0,759 | 0,611 | 0,797 | 0,787 | 0,677 | 0,704 |
| | | | | | 0,862 | 0,905 | 0,674 | 0,863 | 0,848 | 0,863 | 0,771 |
| | | | | | 0,88 | 0,913 | 0,669 | 0,866 | 0,856 | 0,876 | 0,747 |
| | | | | | 0,813 | 0,85 | 0,722 | 0,816 | 0,817 | 0,815 | 0,756 |
| | | | | | 0,979 | 0,991 | 0,489 | 0,919 | 0,932 | 0,949 | 0,816 |
| | | | | | 0,882 | 0,875 | 0,543 | 0,83 | 0,786 | 0,83 | 0,764 |
| | | | | | 0,838 | 0,826 | 0,653 | 0,82 | 0,769 | 0,839 | 0,729 |
| | | | | | 0,871 | 0,909 | 0,654 | 0,868 | 0,866 | 0,862 | 0,784 |
| | | | | | 0,877 | 0,928 | 0,646 | 0,87 | 0,878 | 0,864 | 0,747 |
| | | | | | 0,82 | 0,88 | 0,613 | 0,85 | 0,834 | 0,892 | 0,756 |
| | | | | | 0,932 | 0,957 | 0,483 | 0,82 | 0,873 | 0,772 | 0,731 |

Table 4b: Results (raw data). For each experimental condition, the first row is precision per label, second row is recall per label

Appendix B: IMDbLink

B.1 IMDbLink.java

```
import java.io.IOException;
import java.util.ArrayList;

/**
 * Links subtitle file names with corresponding IMDb entries.
 * Note: currently only works with movies, produced after 1960, for which one or
 * multiple genres are readily available. Filter criteria can be easily adjusted
 * in ReduceDatabase class.
 * @author Sam van der Meer
 * @author sparky
 */
public class IMDbLink {

    //Zipped folder contains IMDb-data file.
    private static final String IMDB_DATABASE_ZIP_FILE = "imdbData.zip";

    //Raw IMDb-data file, as it appears in the zipped folder.
    private static final String IMDB_DATABASE_FILE = "data.tsv";

    //Zipped folder containing folders with subtitle-files (.srt) for each
    //movie.
    private static final String SUBTITLE_FILE = "subtitles.zip";

    //Folder where output subtitle files are stored.
    private static final String OUTPUT_FOLDER = "src/output/subtitles/";

    //Three data files that store the reduced IMDb-database and two for
    //matched IDs, respectively.
    private static final String REDUCED_DATA_FILE = "reducedIMDbData.csv";
    private static final String EXTENSIVE_REPORT_FILE =
        "extensiveReport.csv";
    private static final String COMPACT_REPORT_FILE = "compactReport.csv";

    private String imdbDatabaseZipFile, imdbDatabaseFile, subtitleFile;
    private String outputFolder, reducedIMDbData, extensiveReport,
        compactReport;

    public IMDbLink(String databaseZipFile, String databaseFile,
        String subtitleFile, String outputFolder,
        String reducedIMDbData, String extensiveReport,
        String compactReport) {
        this.imdbDatabaseZipFile = databaseZipFile;
        this.imdbDatabaseFile = databaseFile;
        this.subtitleFile = subtitleFile;
        this.outputFolder = outputFolder;
        this.reducedIMDbData = reducedIMDbData;
        this.extensiveReport = extensiveReport;
        this.compactReport = compactReport;
    }

    /**
     * Main executable function
     * @param args: arguments
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        new IMDbLink(IMDB_DATABASE_ZIP_FILE, IMDB_DATABASE_FILE,
            SUBTITLE_FILE,
            OUTPUT_FOLDER, REDUCED_DATA_FILE, EXTENSIVE_REPORT_FILE,
            COMPACT_REPORT_FILE).run();
    }

    /**
     * Call to other classes
     * @throws IOException
     */
    private void run() throws IOException {
        ArrayList<Entry> entries = new
            ReduceDatabase(imdbDatabaseZipFile,
                imdbDatabaseFile, reducedIMDbData).run();
    }
}
```

```

        ArrayList<String> names = new FetchSubtitleNames(subtitleFile,
            outputFolder).run();
        MatchIMDbTitles matcher = new MatchIMDbTitles
            (entries, names, outputFolder, extensiveReport,
            compactReport);
        matcher.match();
    }
}

```

B.2 Entry.java

```

/**
 * Represents a single IMDb-entry.
 * @author Sam van der Meer
 * @author sparky
 */
public class Entry {

    private String id, type, title;
    private boolean isAdult;
    private int year, length;
    private String[] genres;

    public Entry(String id, String type, String title, boolean isAdult,
        int year, int length, String[] genres) {
        this.id = id;
        this.type = type;
        this.title = title;
        this.isAdult = isAdult;
        this.year = year;
        this.length = length;
        this.genres = genres;
    }

    //Type of the entry. Possible types: movie, short, tvMiniSeries,
    //tvSeries, tvMovie, tvEpisode, tvSpecial, video, videoGame.
    public String getType() {
        return type;
    }

    //Entry title
    public String getTitle() {
        return title;
    }

    //(Non)-Adult entry, true or false.
    public boolean isAdult() {
        return isAdult;
    }

    //IMDb-ID (as appears in URL, preceded by 'tt')
    public String getId() {
        return id;
    }

    //Publication year
    public int getYear() {
        return year;
    }

    //Runtime of entry, in minutes.
    public int getLength() {
        return length;
    }

    //List of associated genres (up to three).
    //Genres: list of associated genres, up to three. Possible types:
    // Action, Adult, Adventure, Animation, Biography, Comedy, Crime,
    // Documentary, Drama, Family, Fantasy, Film-Noir, Game-Show, History,
    // Horror, Music, Musical, Mystery, News, Reality-TV, Romance, Sci-Fi,
    // Short, Sport, Talk-Show, Thriller, War or Western.
    public String[] getGenres() {
        return genres;
    }
}

```

```

/**
 * Changes the list of genres according to new String array.
 * @param newGenres: list of new genres (up to three).
 */
public void setGenres(String[] newGenres) {
    this.genres = newGenres;
}

/**
 * Not all attributes are used here to create the database
 * due to relevancy for the thesis. Can be easily adjusted, however.
 */
@Override
public String toString() {
    StringBuilder string = new StringBuilder();
    string.append(id + ";" + title + ";" + year + ";");
    for (String genre : genres)
        string.append(genre + ",");
    string.deleteCharAt(string.length() - 1);
    return string.toString();
}
}

```

B.3 ReduceDatabase.java

```

import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.zip.ZipEntry;
import java.util.zip.ZipFile;

/**
 * Takes the IMDb data and only takes the entries that meet certain criteria.
 * Writes the new, reduced database to reducedData.csv. Entries that contain
 * multiple movie titles will be written as two separate entries.
 * @author Sam van der Meer
 * @author sparky
 */
public class ReduceDatabase {

    private final String imdbDatabaseZipFile;
    private final String imdbDatabaseFile;
    private final String reducedDataFile;
    private ZipFile zip;
    private Scanner scanner;
    private int entriesAdded;

    /**
     * Database attributes (title.basics.tsv.gz, retrieved from
     * https://datasets.imdbws.com):
     * 0: Unique IMDb-ID (as appears in URL for each entry, preceded by 'tt')
     * 1: Type of the entry. Possible types: movie, short, tvMiniSeries,
     *    tvSeries, tvMovie, tvEpisode, tvSpecial, video or videoGame.
     * 2: Primary title
     * 3: Secondary title
     * 4: (Non)-Adult entry, 0 (no) or 1 (yes)
     * 5: StartYear: integer, indicating starting date (one-time releases)
     * 6: EndYear: integer, indicating ending date (series)
     * 7: Runtime: length of the title in minutes
     * 8: Genres: list of associated genres, up to three. Possible types:
     *    Action, Adult, Adventure, Animation, Biography, Comedy, Crime,
     *    Documentary, Drama, Family, Fantasy, Film-Noir, Game-Show, History,
     *    Horror, Music, Musical, Mystery, News, Reality-TV, Romance, Sci-Fi,
     *    Short, Sport, Talk-Show, Thriller, War or Western.
     * (!) Missing fields are indicated with "\N", entries separated by
     *    newlines.
     */

    /**
     * Class constructor enables reading from database files.
     */
}

```



```

    * @param imdbDatabaseZipFile: Zipped data folder appearing in project
      directory.
    * @param imdbDatabaseFile: Raw IMDb data.tsv file inside zip file.
    * @param reducedDataFile: reducedData.csv stores a reduced version of
      the IMDb-database.
    */
    public ReduceDatabase(String databaseZipFile, String databaseFile,
        String reducedDataFile) {
        this.imdbDatabaseZipFile = databaseZipFile;
        this.imdbDatabaseFile = databaseFile;
        this.reducedDataFile = reducedDataFile;
    }

    /**
     * Filters entries based on certain criteria, which can be changed
       below. Here,
     * we only want (tv-)movie entries, produced on or after 1960, with some
       genre
     * classification.
     * @param attributes: fields for each entry.
     * @return boolean indicating whether the entry should be kept or not.
     */
    private boolean meetsCriteria(String[] attributes) {
        int year = (!attributes[5].equals("\\N")) ?
            Integer.parseInt(attributes[5]) : -1;
        return ((attributes[1].equals("movie") ||
            attributes[1].equals("tvMovie"))
            && (year >= 1960) && !(attributes[8].equals("\\N")));
    }

    /**
     * Declares reader and writer, runs the algorithm.
     * @return the reduced arrayList of entries (also appears as
       reducedData.csv in directory).
     * @throws IOException
     */
    public ArrayList<Entry> run() throws IOException {
        FileWriter csvWriter = new FileWriter(reducedDataFile);
        zip = new ZipFile(imdbDatabaseZipFile);
        ZipEntry database = zip.getEntry(imdbDatabaseFile);
        InputStream reader = zip.getInputStream(database);
        System.out.println("Reading and reducing IMDb titles...");
        return reduce(reader, csvWriter);
    }

    /**
     * Reduces original IMDb database according to certain criteria.
     * Writes new reduced database to reducedData.csv (appears in directory).
     * @param reader: for reading the original database file.
     * @param csvWriter: for writing new reduced database file.
     * @return ArrayList of reduced entries.
     * @throws IOException
     */
    private ArrayList<Entry> reduce(InputStream reader, FileWriter
        csvWriter) throws IOException {
        String line;
        int linesRead = 0;
        ArrayList<Entry> entries = new ArrayList<Entry>();
        scanner = new Scanner(reader);
        scanner.useDelimiter("\\n");
        scanner.nextLine();
        while (scanner.hasNext()) {
            line = scanner.nextLine();
            linesRead++;
            String[] attributes = line.split("\\t");
            if (meetsCriteria(attributes)) {
                if (attributes.length != 9)
                    throw new RuntimeException("Invalid input line");
                addEntry(attributes, entries, csvWriter);
            }
            if (linesRead%200000==0)
                System.out.print("*");
        }
        int percentage = 100-(entriesAdded*100/linesRead);
        System.out.println("\\n" + linesRead + " Lines read, which were
            reduced to " +

```

```

        entriesAdded + " titles (" + percentage + "%
            reduction).\n");
    csvWriter.flush();
    csvWriter.close();
    reader.close();
    return entries;
}

/**
 * Copies an existing entry to the new database and entries data
 * structure.
 * If there is a secondary title and it is different from the primary
 * title,
 * then an additional entry is created in order to correctly match the
 * subtitle files.
 * @param attributes: data fields of each entry
 * @param entries: arrayList containing reduced entries
 * @param csvWriter: for writing new reduced database
 * @throws IOException
 */
private void addEntry(String[] attributes, ArrayList<Entry> entries,
    FileWriter csvWriter) throws IOException {
    String primaryTitle = attributes[2];
    String secondaryTitle = attributes[3];
    if (!primaryTitle.equals("\\N")) {
        entries.add(getEntry(attributes, primaryTitle));
        csvWriter.append(entries.get(entries.size()-1).toString() +
            "\\n");
        entriesAdded++;
    }
    if (!primaryTitle.equals(secondaryTitle) &&
        !secondaryTitle.equals("\\N")) {
        entries.add(getEntry(attributes, secondaryTitle));
        csvWriter.append(entries.get(entries.size()-1).toString() +
            "\\n");
        entriesAdded++;
    }
}

/**
 * Creates new entry object
 * @param attributes: data fields of each entry
 * @param title: correct title for this entry
 * @return entry object
 */
private Entry getEntry(String[] attributes, String title) {
    boolean isAdult = attributes[4].equals("1");
    int year = (!attributes[5].equals("\\N")) ?
        Integer.parseInt(attributes[5]) : -1;
    int length = (!attributes[7].equals("\\N")) ?
        Integer.parseInt(attributes[7]) : -1;
    String[] genres = attributes[8].split(",");
    return new Entry(attributes[0], attributes[1], title, isAdult, year,
        length, genres);
}
}

```

B.4 FetchSubtitleNames.java

```

import java.io.BufferedReader;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;

/**
 * Unzips the subtitle data file and copies the .srt-files to the input folder.

```

```

* Overwrites every filename with the movie title and production year to allow
  easier IMDb linkage.
* @author Sam van der Meer
* @author sparky
*/
public class FetchSubtitleNames {

    private final String subtitleFile;
    private final String outputFolder;

    /**
     * Class constructor, enables reading and writing to directory files.
     * @param subtitleFile: zipped file containing movie folders.
     * @param outputFolder: output folder to store .srt-files.
     */
    public FetchSubtitleNames(String subtitleFile, String outputFolder) {
        this.subtitleFile = subtitleFile;
        this.outputFolder = outputFolder;
    }

    /**
     * Declares readers and writers, runs main algorithm.
     * Only takes title and production year to replace subtitle file name.
     * @return list of new subtitle file names (as they appear in input
     *         folder).
     * @throws IOException
     */
    public ArrayList<String> run() throws IOException {
        System.out.println("Moving and renaming subtitle files...");
        int filesRead = 0, subtitlesAdded = 0;
        ArrayList<String> fileNames = new ArrayList<String>();
        try (FileInputStream fis = new FileInputStream(subtitleFile);
            BufferedInputStream bis = new BufferedInputStream(fis);
            ZipInputStream stream = new ZipInputStream(bis)) {
            File folder = new File(outputFolder);
            checkFolder(folder);
            ZipEntry entry;
            while((entry = stream.getNextEntry()) != null) {
                String name = entry.getName();
                if (name.endsWith(".srt")) {
                    //Only take title and production
                    year
                    String folderName =
                        name.substring(10, name.indexOf(".eng."));
                    ByteArrayOutputStream buffer = new
                        ByteArrayOutputStream();
                    copy(stream, buffer);
                    //Check if multiple subtitle files for a
                    title exist.
                    //Then, combine information into a new
                    subtitle file.
                    FileOutputStream fileOutputStream = new
                        FileOutputStream(new File
                            (folder, folderName + multipleCDs(fileNames,
                                folderName, name) + ".srt"));
                    ByteArrayInputStream inputStream = new
                        ByteArrayInputStream
                            (buffer.toByteArray());
                    copy(inputStream, fileOutputStream);
                    fileNames.add(folderName);
                    subtitlesAdded++;
                    inputStream.close();
                    fileOutputStream.close();
                }
                filesRead++;
            }
            if (filesRead % 500 == 0)
                System.out.print("*");
        }
        System.out.println("\n" + filesRead + " Files
            read, "
            + subtitlesAdded + " subtitle files
            added.");
        fis.close();
        bis.close();
    }
    return fileNames;
}

```

```

    }

    /**
     * If output folder does not yet exists , makes one.
     * Otherwise it will delete all previous output data first.
     * @param folder: output folder
     */
    private void checkFolder(File folder) {
        if (!folder.exists())
            folder.mkdir();
        else
            for (File f : folder.listFiles()){
                if (!f.isDirectory())
                    f.delete();
            }
    }

    /**
     * Checks if a movie contains multiple subtitle files.
     * If there are, a simple string will be appended to the filename.
     * @param fileNames: list of replaced filenames
     * @param folderName: current filename to be considered.
     * @param name: name of directory path
     * @return empty string if movie is unique, "CD2" or "CD3" for multiple
     *         files.
     */
    private String multipleCDs(ArrayList<String> fileNames, String
        folderName, String name) {
        if (fileNames.size()>1 && !name.contains("1cd") &&
            fileNames.get(fileNames.size()-1).equals(folderName))
        {
            if
                (fileNames.get(fileNames.size()-2).equals(folderName))
            {
                return ".CD3";
            }
            return ".CD2";
        }
        else return "";
    }

    /**
     * Copies content of a subtitle file to a new file.
     * @param input: input file
     * @param output: output file
     * @throws IOException
     */
    private void copy(InputStream input, OutputStream output) throws
        IOException {
        byte[] buf=new byte[1024];
        while(true){
            int length=input.read(buf);
            if(length<0)
                break;
            output.write(buf,0,length);
        }
    }
}

```

B.5 MatchIMDbTitles.java

```

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * Matches subtitle files with IMDb IDs using Levenshtein distances.
 * Only keeps subtitles that appear along the seven most occurring genres.
 * Subtitle file names are adjusted with corresponding IMDb ID and original
 * title name.

```

```

    * @author Sam van der Meer
    * @author sparky
    */
public class MatchIMDbTitles {

    private ArrayList<Entry> imdbEntries;
    private ArrayList<String> subtitleFileNames;
    private int titlesAnalyzed, subtitleFilesWritten;
    private final String outputFolder;
    private final FileWriter extensiveWriter, compactWriter;

    //Used to keep the most frequently occurring genres only.
    private String[] frequentGenres =
        {"Action", "Adventure", "Comedy", "Crime", "Drama", "Romance", "Thriller"};

    /**
     * Class constructor, defines two file writers to write .csv-report
     * files.
     * @param imdbEntries: all IMDb-database entries that remained after
     * filtering.
     * @param subtitleFileNames: list of subtitle file names.
     * @param outputFolder: output folder file path.
     * @param extensiveReport: file name of the extensive report.
     * @param compactReport: file name of the compact report.
     * @throws IOException
     */
    public MatchIMDbTitles(ArrayList<Entry> imdbEntries, ArrayList<String>
        subtitleFileNames,
        String outputFolder, String extensiveReport, String
        compactReport) throws IOException {
        this.imdbEntries = imdbEntries;
        this.subtitleFileNames = subtitleFileNames;
        this.outputFolder = outputFolder;
        this.extensiveWriter = new FileWriter(extensiveReport);
        this.compactWriter = new FileWriter(compactReport);
    }

    /**
     * Main executable algorithm. Writes headers in file writers and informs
     * user
     * of execution progress. Reports can be found in extensiveReport.csv
     * and compactReport.csv.
     * @throws IOException
     */
    public void match() throws IOException {
        System.out.println("\nAttempting to match subtitles to IMDb
            IDs...");
        extensiveWriter.append("Matched IMDb title info;;; Matching
            report;;; \n");
        extensiveWriter.append("ID; Title; Year; Genres; Title original
            (again);"
            + "Title IMDb match; Score; Reduced genres \n");
        compactWriter.append("ID; Title; Year; Genres \n");
        getMatches();
        System.out.println("\nDone! Subtitles can be found in src/output
            folder,"
            + " report in \n\t extensiveReport.csv and
            compactReport.csv");

        printResults();
        extensiveWriter.flush();
        extensiveWriter.close();
        compactWriter.flush();
        compactWriter.close();
    }

    /**
     * Iterates through the list of all subtitle files, then checks which
     * IMDb-title
     * matches the subtitle file best. If the subtitle file being considered
     * solely
     * contains one or more of the most occurring genres (see frequentGenre
     * class
     * attribute), it will be added to the output folder and report files.
     * Otherwise, it will be deleted.
     * @throws IOException
     */
}

```

```

private void getMatches() throws IOException {
    File directory = new File(outputFolder);
    File[] files = directory.listFiles();
    for (String fileName : subtitleFileNames) {
        int bestScore = Integer.MAX_VALUE;
        Entry bestEntry = null;
        String name = fileName.substring(0,
            fileName.indexOf(".("));
        name = name.replace('.', ' ');
        int year = getFileNameYear(fileName);
        for (Entry entry : imdbEntries) {
            if (entry.getYear() == year) {
                int score =
                    levenshtein(name, entry.getTitle().toLowerCase());
                if (score < bestScore) {
                    bestScore = score;
                    bestEntry = entry;
                }
            }
        }
        if (hasOnlyHighFreqGenres(bestEntry))
            addSubtitleToFile(bestEntry, name, bestScore,
                files);
        else
            deleteSubtitleFile(files, bestEntry);
        titlesAnalyzed++;
        if (titlesAnalyzed % 165 == 0)
            System.out.print("*");
    }
}

/**
 * Adds subtitle file to the output folder and changes the filename by
 * including IMDb-ID.
 * @param bestEntry: IMDb-entry that was matched with the current
 * subtitle file.
 * @param name: title of subtitle file name excluding production year
 * and no. of cds.
 * @param bestScore: score from computing the levenshtein distance.
 * @param files: list of initial files in output folder.
 * @throws IOException
 */
private void addSubtitleToFile(Entry bestEntry, String name,
    int bestScore, File[] files) throws IOException {
    subtitleFilesWritten++;
    extensiveWriter.append(bestEntry.toString() + ";" + name + ";" +
        bestEntry.getTitle() + ";" + computeSimilarity(name,
            bestScore) + "%;" + Arrays.toString(bestEntry.getGenres()) +
            "\n");
    compactWriter.append(bestEntry.getId() + ";" +
        bestEntry.getTitle() + ";" + bestEntry.getYear() + ";" +
        Arrays.toString(bestEntry.getGenres()) + "\n");
    renameSubtitleFile(files, bestEntry);
}

/**
 * Computes a percentages of the similarity measure, compensating for
 * title
 * name length. This normalised score gives an rough indication whether a
 * file was matched correctly or not.
 * @param name: subtitle file name (without year).
 * @param bestScore: levenshtein similarity score.
 * @return percentage indicating normalised similarity.
 */
private int computeSimilarity(String name, int bestScore) {
    //scores < 75% should be checked for matching errors.
    double simScore = bestScore;
    double simLength = name.length();
    return (int) (100 - (simScore/simLength)*100);
}

/**
 * Renames a subtitle file name by appending corresponding IMDb ID.
 * @param files: list of subtitle files appearing in output folder.
 * @param bestEntry: current entry being adjusted.
 */

```

```

private void renameSubtitleFile(File[] files, Entry entry) {
    if (titlesAnalyzed < files.length) {
        File subtitleFile = files[titlesAnalyzed];
        File id = new File(outputFolder +
            entry.getId().toString() + "(" +
            subtitleFile.getName().substring(0,
            subtitleFile.getName().length()-4) +
            ").srt");
        subtitleFile.renameTo(id);
    }
}

/**
 * Deletes a subtitle file name out of the output folder.
 * @param files: list of subtitle files appearing in output folder.
 * @param bestEntry: current entry being removed.
 */
private void deleteSubtitleFile(File[] files, Entry bestEntry) {
    if (titlesAnalyzed < files.length) {
        File subtitleFile = files[titlesAnalyzed];
        subtitleFile.delete();
    }
}

/**
 * Pattern for detecting production years in subtitle file names.
 * @param fileName: the filename being considered.
 * @return the production year as integer in the subtitle file name
 * string.
 */
public int getFileNameYear(String fileName) {
    Pattern yearPattern =
        Pattern.compile("\\(19[0-9][0-9]\\\\)\\\\(20[01][0-9]\\\\)");
    Matcher matcher = yearPattern.matcher(fileName);
    matcher.find();
    String yearString =
        fileName.substring(matcher.start()+1,matcher.end()-1);
    return Integer.parseInt(yearString);
}

/**
 * Computes the Levenshtein distance between two strings.
 * Functions as similarity measure in order to select the most similar
 * one.
 * @param name: original file name string.
 * @param title: current IMDb-title string being considered.
 * @return distance as integer, lower is better (more similar).
 */
private int levenshtein(String name, String title) {
    int m = name.length()+1;
    int n = title.length()+1;
    int[][] distance = new int[m][n];
    for(int i=0; i < m; i++)
        distance[i][0]=i;
    for(int j=0; j < n; j++)
        distance[0][j]=j;
    for(int i=1; i < m; i++)
        for(int j=1; j < n; j++) {
            int cost;
            if(name.charAt(i-1) == title.charAt(j-1))
                cost = 0;
            else
                cost = 1;
            distance[i][j] = minimum(distance[i-1][j]+1,
            distance[i][j-1]+1,distance[i-1][j-1]+cost);
        }
    return distance[m-1][n-1];
}

/**
 * Returns the minimum between three integers.
 * @param i: integer1.
 * @param j: integer2.
 * @param k: integer3.
 * @return minimum of integer1, integer2 and integer3.
 */

```

```

private int minimum(int i, int j, int k) {
    return minimum(i, minimum(j, k));
}

/**
 * Returns the minimum between two integers.
 * @param j: integer1.
 * @param k: integer2.
 * @return minimum of integer1 and interger2.
 */
private int minimum(int j, int k) {
    if(j < k)
        return j;
    return k;
}

/**
 * Checks if the entry only contains genres listed in the private class
    attribute.
 * Most popular genres are Action, Adventure, Comedy, Crime, Drama,
    Romance, Thriller.
 * @param entry: entry to be considered, only genre-attribute is used.
 * @return boolean indicating if it only contains the most frequently
    occurring genres.
 */
private boolean hasOnlyHighFreqGenres(Entry entry) {
    for (String genre : entry.getGenres())
        if
            (!Arrays.stream(frequentGenres).parallel().anyMatch(genre::contains))
                return false;
    return true;
}

/**
 * Informs the user what proportion of the subtitles was kept.
 * @param numberMatched: the total number of successful IMDb-links made.
 */
private void printResults() {
    int percentage = subtitleFilesWritten*100/titlesAnalyzed;
    System.out.println("\n" + subtitleFilesWritten + "/" +
        titlesAnalyzed +
        " (" + percentage + "%) of the subtitle files are along
        the most " + "occurring \n\t genres " +
        Arrays.toString(frequentGenres));
}
}

```


Appendix C: SubtitlePreprocessing

C.1 SubtitlePreprocessing.java

```
import java.io.IOException;

/**
 * Serves various pre-processing purposes for natural language processing of
 * subtitle files.
 * Combines all subtitles, included with genre labels, into a single ARFF-file
 * for further analysis.
 * @author Sam van der Meer
 */
public class SubtitlePreprocessing {

    private static final String GENRE_DATA_FILE = "compactReport.csv";
    private static final String SUBTITLE_FILE = "subtitles.zip";
    private static final String OUTPUT_FILE = "data.arff";
    private String reportFile, subtitleFile, outputFileName;

    /**
     * Class constructor, initialising file locations.
     * @param reportFile: csv-data file with genre allocations.
     * @param subtitleFile: zip-file with database of subtitles.
     * @param outputFileName: output file name to write reduced subtitle
     * files to.
     */
    public SubtitlePreprocessing(String reportFile, String subtitleFile,
        String outputFileName) {
        this.reportFile = reportFile;
        this.subtitleFile = subtitleFile;
        this.outputFileName = outputFileName;
    }

    /**
     * Main executable function
     * @param args: arguments
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        new SubtitlePreprocessing(GENRE_DATA_FILE, SUBTITLE_FILE,
            OUTPUT_FILE).run();
    }

    /**
     * Call to other classes. Parameter settings for subreader:
     * False -> Strings will only get pre-processed and lemmatized.
     * True -> Strings will get pre-processed, lemmatized and POS-tagged.
     * @throws IOException
     */
    public void run() throws IOException {
        SubWriter writer = new SubWriter(outputFileName, reportFile);
        SubReader reader = new SubReader(writer, subtitleFile, true);
        reader.readSubtitleFiles();
    }
}
```

C.2 SubReader.java

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import java.util.zip.ZipEntry;
import java.util.zip.ZipFile;
```

```

import java.util.zip.ZipInputStream;

/**
 * Reads database of subtitle files one line at a time.
 * Makes calls to reduce data and write them to a new file.
 * @author Sam van der Meer
 */
public class SubReader {

    private final SubWriter writer;
    private final Trimmer trimmer;
    private final String subtitleFile;

    //If wordNetMode is activated, also applies POS-Tagger on the subtitles.
    private final boolean wordNetMode;

    /**
     * Class constructor initialises data, also makes new trimmer object.
     * @param writer: writer class to write new reduced subtitle files.
     * @param subtitleFile: location of zip-file containing subtitle
     * database.
     * @param wordNetMode: if true, strings will also get POS-tagged.
     */
    public SubReader(SubWriter writer, String subtitleFile, boolean
        wordNetMode) {
        this.writer = writer;
        this.subtitleFile = subtitleFile;
        this.wordNetMode = wordNetMode;
        this.trimmer = new Trimmer();
    }

    /**
     * Reads every subtitle file in the database, passes is onto
     * readSubtitleFile-method for
     * reading the files themselves. Passes results to the writer object for
     * output generation.
     * @throws IOException
     */
    public void readSubtitleFiles() throws IOException {
        printUserInformation();
        FileInputStream fis = new FileInputStream(subtitleFile);
        BufferedInputStream bis = new BufferedInputStream(fis);
        ZipInputStream stream = new ZipInputStream(bis);
        stream.getNextEntry();
        ZipEntry entry;
        ArrayList<Map<String, String>> subtitles = new
            ArrayList<Map<String, String>>();
        StanfordLemmatizer lem = new StanfordLemmatizer();
        StanfordPOSTagger pos = new StanfordPOSTagger();
        while((entry = stream.getNextEntry()) != null) {
            System.out.println("Analysing " +
                entry.getName().substring(20));
            Map<String, String> subEntry = new
                HashMap<String, String>();
            String subtitleID =
                entry.getName().subSequence(10,19).toString();
            String reducedString = readSubtitleFile(entry);
            String lemmatizedString = lem.lemmatize(reducedString);
            if (wordNetMode)
                subEntry.put(subtitleID,
                    pos.tag(lemmatizedString));
            else
                subEntry.put(subtitleID, lemmatizedString);
            subtitles.add(subEntry);
        }
        writer.write(subtitles);
        System.out.println("\nDone! Result can be found in data.arff.");
        stream.close();
    }

    /**
     * Prints an introduction to the console displaying the current mode.
     */
    private void printUserInformation() {
        if (wordNetMode)
            System.out.println("WordNetMode activated: POS-tagger

```

```

        will also be applied.");
    else
        System.out.println("WordNetMode deactivated: strings
        will get lemmatized only.");
    System.out.println("Pre-processing subtitlte data...\n");
}

/**
 * Reads a single subtitle file line by line. Once everything has been
 * read and reduced, it will call the writer object to create a new file.
 * @param entry: current subtitle file being read.
 * @throws IOException
 */
private String readSubtitleFile(ZipEntry entry) throws IOException {
    ZipFile zip = new ZipFile(new File(subtitleFile));
    InputStream in = zip.getInputStream(entry);
    Scanner subReader = new Scanner(in);
    StringBuilder lines = new StringBuilder();
    String line, lastLine = "";
    int linesRead = 0;
    while (subReader.hasNext()) {
        line = subReader.nextLine();
        linesRead++;
        boolean checkTrademark = (linesRead < 5);
        if (trimmer.isText(line, checkTrademark)) {
            lastLine = cleanUpLine(line);
            lines.append(lastLine);
        }
    }
    if (!trimmer.isText(lastLine, true))
        lines.delete(lines.length() - 1 - lastLine.length(),
            lines.length() - 1);
    zip.close();
    subReader.close();
    return lines.toString();
}

/**
 * Checks for common characteristics in each line, calls
 * the corresponding trimmer method(s) accordingly.
 * @param line: current line of text being considered.
 * @return the reduced line if there's anything left, the empty string
 * otherwise.
 */
private String cleanUpLine(String line) {
    String[] containsFormatting = {"<", "</"};
    String[] containsPunctuation = {"\\*", "\\\"", "-", "#", "'", ",", ";",
    String[] containsLineTerminators = {"!", "?", "."};
    String[] containsBrackets = {"(", ")", "{", "}", "[", "]"};
    line = trimmer.removeDots(line);
    if
        (Arrays.stream(containsFormatting).parallel().anyMatch(line::contains))
        line = trimmer.trimFormatting(line);
    if
        (Arrays.stream(containsPunctuation).parallel().anyMatch(line::contains))
        line = trimmer.trimPunctuation(line,
        containsPunctuation);
    if
        (Arrays.stream(containsLineTerminators).parallel().anyMatch(line::contains))
        line = trimmer.trimLineTerminators(line,
        containsLineTerminators);
    if
        (Arrays.stream(containsBrackets).parallel().anyMatch(line::contains))
        line = trimmer.trimBrackets(line);
    if (line.length() > 0) {
        Character first = line.charAt(0);
        if (Character.isWhitespace(first))
            line = trimmer.removeSpaces(line);
        else if (Character.isUpperCase(first))
            line = trimmer.removeDialogueSubject(line);
        if (!line.endsWith(" "))
            return (line.toLowerCase() + " ");
        else return line.toLowerCase();
    }
    else return "";
}

```

C.3 StanfordLemmatizer.java

```
import java.util.List;
import java.util.Properties;
import edu.stanford.nlp.ling.CoreAnnotations.LemmaAnnotation;
import edu.stanford.nlp.ling.CoreAnnotations.SentencesAnnotation;
import edu.stanford.nlp.ling.CoreAnnotations.TokensAnnotation;
import edu.stanford.nlp.ling.CoreLabel;
import edu.stanford.nlp.pipeline.Annotation;
import edu.stanford.nlp.pipeline.StanfordCoreNLP;
import edu.stanford.nlp.util.CoreMap;

/**
 * Lemmatization class for mapping variation in words to the corresponding lemma.
 * Adapted from Stanford CoreNLP package, version 3.9.2.
 * Retrieved from https://stanfordnlp.github.io/CoreNLP/index.html as seen in:
 * Manning, Christopher D., Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J.
 * Bethard,
 * and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing
 * Toolkit
 * In Proceedings of the 52nd Annual Meeting of the Association for
 * Computational Linguistics: System Demonstrations, pp. 55–60.
 * Class code retrieved from
 * https://stackoverflow.com/questions/1578062/lemmatization-java:
 * @author Tihamer (11-11-2013).
 */
public class StanfordLemmatizer {

    protected StanfordCoreNLP pipeline;

    public StanfordLemmatizer() {
        // Create StanfordCoreNLP object properties, with POS tagging
        // (required for lemmatization), and lemmatization
        Properties props;
        props = new Properties();
        props.put("annotators", "tokenize, ssplit, pos, lemma");

        /*
         * This is a pipeline that takes in a string and returns various
         * analyzed linguistic forms. The String is tokenized via a tokenizer
         * (such as PTBTokenizerAnnotator), and then other sequence model
         * style annotation can be used to add things like lemmas, POS tags,
         * and named entities. These are returned as a list of CoreLabels.
         * Other analysis components build and store parse trees, dependency
         * graphs, etc.
         * This class is designed to apply multiple Annotators to an Annotation.
         * The idea is that you first build up the pipeline by adding
         * Annotators, and then you take the objects you wish to annotate and
         * pass them in and get in return a fully annotated object.
         *
         * StanfordCoreNLP loads a lot of models, so you probably
         * only want to do this once per execution
         */
        this.pipeline = new StanfordCoreNLP(props);
    }

    public String lemmatize(String documentText) {
        StringBuilder lemmas = new StringBuilder();
        // Create an empty Annotation just with the given text
        Annotation document = new Annotation(documentText);
        // run all Annotators on this text
        this.pipeline.annotate(document);
        // Iterate over all of the sentences found
        List<CoreMap> sentences = document.get(SentencesAnnotation.class);
        for(CoreMap sentence: sentences) {
            // Iterate over all tokens in a sentence
            for (CoreLabel token: sentence.get(TokensAnnotation.class)) {
                // Retrieve and add the lemma for each word into the
                // string of lemmas
                lemmas.append(token.get(LemmaAnnotation.class)+" ");
            }
        }
        return lemmas.toString();
    }
}
```

C.4 StanfordPOSTagger.java

```
import edu.stanford.nlp.tagger.maxent.MaxentTagger;

/**
 * Applies POS-tagging on a subtitle string using the Stanford POS-tagger.
 * @author Sam van der Meer
 */
public class StanfordPOSTagger {

    private final MaxentTagger tagger;

    /**
     * Class constructor, initialises new tagger object once.
     */
    public StanfordPOSTagger() {
        this.tagger = new
            MaxentTagger("edu/stanford/nlp/models/pos-tagger/"
                + "english-left3words/english-left3words-distsim.tagger");
    }

    /**
     * Applies POS-tagging on a subtitle string.
     * @param lemmatizedString: the (lemmatized) subtitle string.
     * @return POS-tagged string reduced to a certain range of tags.
     */
    public String tag(String lemmatizedString) {
        String taggedString = tagger.tagString(lemmatizedString);
        return reduceString(taggedString);
    }

    /**
     * Reduces a POS-tagged string to a limited range of tags.
     * Only keeps adjectives (JJ*), nouns (NN*), verbs (VB*) and adverbs
     * (RB*, but not WRB, which are adverbs starting with wh-), discards the
     * rest.
     * @param taggedString: the POS-tagged subtitle string.
     * @return a reduced string of words with tags.
     */
    private String reduceString(String taggedString) {
        String[] words = taggedString.split(" ");
        StringBuilder reducedString = new StringBuilder();
        for (String word : words)
            if (word.contains("JJ") || word.contains("NN") ||
                word.contains("VB") ||
                ((word.contains("RB") &&
                    !word.contains("WRB"))))
                reducedString.append(word + " ");
        return reducedString.toString();
    }
}
```

C.5 Trimmer.java

```
import java.util.Arrays;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * Contains methods to trim down subtitle files based on certain features.
 * @author Sam van der Meer
 */
public class Trimmer {

    /**
     * Checks whether a line is natural language or not.
     * Discards sequence numbers, time stamps and newlines.
     * @param line: the current line being considered.
     * @param checkTrademark: tells the method if it should check on any
     * trademarks.
     * @return boolean indicating whether the line is natural language or
     * not.
     */
}
```

```

    */
    public boolean isText(String line, boolean checkTrademark) {
        if (checkTrademark && isTradeMark(line))
            return false;
        Pattern timeStamp = Pattern.compile
            ("\\d{2}:\\d{2}:\\d{2},\\d{3} →>
             \\d{2}:\\d{2}:\\d{2},\\d{3}");
        Matcher matcher = timeStamp.matcher(line);
        if (line.matches("\\d+") || matcher.find() || line.isEmpty())
            return false;
        else
            return true;
    }

    /**
     * Checks if the first or last string belong to the subtitles themselves.
     * Subtitle makers often add a source string at the beginning or end of
     * their files.
     * @param line: the current line being considered.
     * @return boolean indicating if the line is a trademark or not.
    */
    private boolean isTradeMark(String line) {
        String[] strings = {"www", ".com", "http", ".org", "fps", "sync
            by", "Subtitle", "subtitle", "subs",
            "Resync", "Sync", "Cleaned", "Ripped", "Produced", "Screenplay", "Production"};
        if (Arrays.stream(strings).parallel().anyMatch(line::contains))
            return true;
        else return false;
    }

    /**
     * Removes all elements of formatting in a line.
     * @param line: the current line being considered.
     * @return the line without any formatting elements.
    */
    public String trimFormatting(String line) {
        String trimmedLine = line;
        Pattern font = Pattern.compile("\\<font color=.{2,10}>");
        Matcher match = font.matcher(trimmedLine);
        if (match.find())
            trimmedLine = trimmedLine.replace(match.group(), "");
        String[] toBeReplaced =
            {"<b>", "</b>", "<i>", "</i>", "<u>", "</u>", "</font>"};
        for (String s : toBeReplaced)
            trimmedLine = trimmedLine.replaceAll(s, "");
        return trimmedLine;
    }

    /**
     * Removes all irrelevant pieces of punctuation in a line.
     * @param line: the current line being considered.
     * @param toBeReplaced: list of punctuation tokens that should be
     * removed.
     * @return the line without any punctuation markers.
    */
    public String trimPunctuation(String line, String[] punctuations) {
        for (String s : punctuations)
            line = line.replaceAll(s, "");
        return line;
    }

    /**
     * Replaces line ending punctuation marks (. ! ?) with a space.
     * @param line: the current line being considered.
     * @param lineTerminators: list of punctuation tokens that should be
     * replaced.
     * @return the line with the line terminator replaced by a space.
    */
    public String trimLineTerminators(String line, String[] lineTerminators)
    {
        for (String s : lineTerminators)
            if (line.endsWith(s))
                line = line.replace(s, " ");
        return line;
    }

```

```

public String removeDots(String line) {
    Pattern dots = Pattern.compile("\\.{3}");
    Matcher match = dots.matcher(line);
    return match.replaceAll("");
}

/**
 * Removes everything in between brackets that appears in a subtitle
 * file.
 * @param line: the current line being considered.
 * @return the line without any brackets and what was inside them.
 */
public String trimBrackets(String line) {
    Pattern brackets = Pattern.compile("\\{.*\\}|\\[.*\\]|\\(.*\\)");
    Matcher match = brackets.matcher(line);
    if (!match.find())
        return line;
    else {
        String inBrackets = match.group();
        return line.replace(inBrackets, "");
    }
}

/**
 * If any spaces appear at the beginning of a line, removes them.
 * @param line: the current line being considered.
 * @return the line with the spaces at the beginning removed.
 */
public String removeSpaces(String line){
    if (line.length() == 0)
        return line;
    if (!(line.charAt(0) == ' '))
        return line;
    else
        return removeSpaces(line.replaceFirst(" ", ""));
}

/**
 * Checks if a line starts with upper-case characters followed by a
 * colon,
 * which indicates a dialogue subject (line that is said by someone).
 * Removes the subject if it is detected, returns original string
 * otherwise.
 * @param line: the current line being considered.
 * @return the line without dialogue subject, if it was detected.
 */
public String removeDialogueSubject(String line) {
    for (int i=0; i<line.length(); i++) {
        if (Character.isLowerCase(line.charAt(i)))
            return line;
        else if (line.charAt(i) == ':') {
            return line.substring(i+1);
        }
    }
    return line;
}
}

```

C.6 SubWriter.java

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Map;
import java.util.Scanner;

/**
 * Writes all subtitles to a single .ARFF-file, which is used in Weka/Meka.
 * @author Sam van der Meer
 */
public class SubWriter {

    private final File file;
    private final String genreDataFile;
    private String[] validGenres =
        {"Action", "Adventure", "Comedy", "Crime", "Drama", "Romance",
         "Thriller"};

    /**
     * Class constructor, initialising output file.
     * @param outputFile: file where the output is written to.
     * @param genreDataFile: data file (.csv) containing genres for each
     * title.
     */
    public SubWriter(String outputFile, String genreDataFile) {
        this.genreDataFile = genreDataFile;
        File file = new File(outputFile);
        if (file.exists())
            file.delete();
        this.file = file;
    }

    /**
     * Writes all subtitles to a single .ARFF-file, separated by newlines,
     * by combining boolean values for each appearance of genre and the
     * subtitle string itself.
     * @param subtitles: all reduced subtitle files and the corresponding
     * IDs.
     * @throws IOException
     */
    public void write(ArrayList<Map<String, String>> subtitles) throws
        IOException {
        PrintWriter writer = new PrintWriter(new FileWriter(file, true));
        System.out.println("\nWriting subtitles (text-only) to output
            file ...");
        writer.write(getHeader());
        try {
            for (Map<String, String> subtitle : subtitles) {
                subtitle.forEach((key, value) -> {
                    try {
                        writer.write(getLabels(key));
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    writer.write("\n" + value + "\n\n");
                });
            }
        } finally {
            writer.close();
        }
    }

    /**
     * Print header at beginning of .ARFF-file, which declares attributes
     * for each entry.
     * @return string that prints the necessary information.
     */
    private String getHeader() {
        StringBuilder header = new StringBuilder();
        //Defines the relation and number of attribute labels
        header.append("@relation 'subtitles: -C 7' \n\n");
        for (String genre : validGenres)
```



```

        //For each genre, a new binary attribute is defined
        header.append("@attribute " + genre + " {1,0} \n");
    //For the subtitle string a separate attribute is defined
    header.append("@attribute subtitle string \n\n");
    //Denotes the start of the data segment in the file
    header.append("@data \n\n");
    return header.toString();
}

/**
 * Searches the genre-data file for a corresponding match using the
 * subtitle ID.
 * @param key: subtitle ID that is being considered.
 * @return string containing zeros and ones for all genres.
 * @throws Exception - if no match has been found in the database.
 */
private String getLabels(String key) throws Exception {
    Scanner reader = new Scanner(new FileReader(genreDataFile));
    reader.useDelimiter(";");
    while(reader.hasNext()) {
        String[] entry = reader.nextLine().split(";");
        if (key.equals(entry[0])) {
            String genres = entry[3];
            reader.close();
            return getLabelString(genres);
        }
    }
    reader.close();
    throw new Exception("Match not found in genre database!");
}

/**
 * Makes string of genre labels consisting of zeros and ones.
 * For example, "0,0,1,0,0,1,0" = [Comedy, Romance].
 * @param genres: genres as they appear in genre-data file for current
 * title.
 * @return string of seven zeros or ones, corresponding to appearing
 * genres.
 */
private String getLabelString(String genres) {
    StringBuilder labelString = new StringBuilder();
    for (String genre : validGenres)
        if (genres.contains(genre))
            labelString.append("1,");
        else
            labelString.append("0,");
    return labelString.toString();
}
}

```

Appendix D: WordNetLink.java

```
package net.sf.extjwnl.utilities;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import net.sf.extjwnl.JWNLEException;
import net.sf.extjwnl.data.IndexWord;
import net.sf.extjwnl.data.POS;
import net.sf.extjwnl.data.PointerUtils;
import net.sf.extjwnl.data.list.PointerTargetNodeList;
import net.sf.extjwnl.dictionary.Dictionary;

/**
 * Replaces words of pre-processed subtitles with corresponding hypernyms from
 * WordNet.
 * Writes results to a new .arff-file such that it can be analysed by Meka.
 * @author Sam van der Meer
 */
public class WordNetLink {

    private final static String FILENAME = "C:/Users/Sam/Desktop/data.arff";
    private final static String OUTPUTFILE = "data_wordnet.arff";
    private Dictionary dictionary = null;
    private final PrintWriter writer;
    private String[] validGenres =
        {"Action", "Adventure", "Comedy", "Crime", "Drama", "Romance",
         "Thriller"};

    /**
     * Class constructor, creates output file and writes the attribute
     * header.
     * @throws JWNLEException
     * @throws IOException
     */
    public WordNetLink() throws JWNLEException, IOException {
        this.dictionary = Dictionary.getDefaultResourceInstance();
        File file = new File(OUTPUTFILE);
        if (file.exists())
            file.delete();
        this.writer = new PrintWriter(new FileWriter(file, true));
        writeHeader();
    }

    public static void main(String[] args) throws JWNLEException, IOException
    {
        new WordNetLink().readSubtitles();
    }

    /**
     * Reads the file of pre-processed subtitles and calls the
     * getHypernymsFromSubtitle method for each instance.
     * @throws IOException
     */
    private void readSubtitles() throws IOException {
        System.out.println("Replacing subtitles with WordNet
            hypernyms...");
        BufferedReader reader = new BufferedReader(new
            FileReader(FILENAME));
        String currentLine = "";
        while (!currentLine.contains("@data"))
            currentLine = reader.readLine();
        reader.readLine();
        while ((currentLine = reader.readLine()) != null)
            getHypernymsFromSubtitle(currentLine);
        System.out.println("\nDone! Results can be found in
            data_wordnet.arff");
        reader.close();
        writer.close();
    }
}
```

```

/**
 * For each word in the subtitle instance, looks up the hypernyms (if
 * they exist)
 * and passes them to a method that writes them to the output file.
 * @param currentLine: current subtitle instance being considered.
 */
private void getHypernymsFromSubtitle(String currentLine) {
    writer.write(currentLine.substring(0,14)+"\n");
    String[] words = currentLine.substring(15).split(" ");
    for (String word : words) {
        POS syntacticCategory = getSyntacticCategory(word);
        try {
            if (word.equals("\n")) {
                writer.write("\n\n");
                break;
            }
            IndexWord iWord = dictionary.lookupIndexWord(
                (syntacticCategory, word.substring(0,
                    word.lastIndexOf('_'))));
            PointerTargetNodeList hypernyms =
                PointerUtils.getDirectHypernyms(iWord.getSenses().get(0));
            String descriptiveString = hypernyms.toString();
            //List of hypernyms may not empty
            if (descriptiveString.length() > 3) {
                String hypernymWordString =
                    descriptiveString.substring(
                        (descriptiveString.indexOf("Words:
                            ") + 7,
                            descriptiveString.indexOf("
                                --")));
                String[] hypernymWords =
                    hypernymWordString.split(",");
                writeWords(hypernymWords);
            }
        }
        //Word cannot be matched with database, continue to next
        word.
        catch (NullPointerException e) {}
        catch (JWNLEException e) {
            e.printStackTrace();
        }
    }
}

/**
 * Gets the syntactic category from the POS-tag concatenated to the word.
 * @param word: the current word being considered.
 * @return: the corresponding POS (Part-of-Speech) enumeration value.
 */
private POS getSyntacticCategory(String word) {
    if (word.contains("JJ"))
        return POS.ADJECTIVE;
    else if (word.contains("NN"))
        return POS.NOUN;
    else if (word.contains("RB"))
        return POS.ADVERB;
    else
        return POS.VERB;
}

/**
 * Print header at beginning of .ARFF-file, which declares attributes
 * for each entry.
 * @return string that prints the necessary information.
 */
private void writeHeader() {
    StringBuilder header = new StringBuilder();
    //Defines the relation and number of attribute labels
    header.append("@relation 'subtitles: -C 7' \n\n");
    for (String genre : validGenres)
        //For each genre, a new binary attribute is defined
        header.append("@attribute " + genre + " {1,0} \n");
    //For the subtitle string a separate attribute is defined
    header.append("@attribute subtitle string \n\n");
    //Denotes the start of the data segment in the file
    header.append("@data \n\n");
}

```

```

        writer.write(header.toString());
    }

    /**
     * Prints all the hypernoms that a word contains.
     * Hypernoms consisting of mulitple words are concatenated with '_' to a
     * single word.
     * @param hypernymWords: the string array of hypernoms of the current
     * word being considered.
     */
    private void writeWords(String[] hypernymWords) {
        for (String word : hypernymWords) {
            if (word.contains(" "))
                word = word.replace(" ", "_");
            if (word.charAt(0)=='_')
                word = word.substring(1);
            writer.write(" " + word.toLowerCase());
        }
    }
}

```