Classification of historical death and occupation coding

Mengyan Zhang

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Advanced Computing (Honours) at
Research School of Computer Science
Australian National University

May 2018

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work.

Mengyan Zhang
17 May 2018

Acknowledgements

I would like to thank my supervisors Prof Peter Christen (Research School of Computer Science, Australian National University) and Dr Timothy Graham (Research School of Computer Science, Australian National University) for their support and guidance throughout my honours year, with their expertise and patience. I could not have imagined having better supervisors for my research and thesis writing.

I would like to thank ANU and Research School of Computer Science for teaching and supporting over my undergraduate career. I have learnt a way to learn knowledge and been inspired by many outstanding researchers.

Many thanks to my family for their love and support throughout my life. My parents, Zheng Zhang and Shuqin Wang, have been encouraging and and understanding all the time.

I would like to thank my friend Tianyu Wang for his loving support throughout the year.

Abstract

This is my honours thesis, I summary the key points here, see detailed approaches in the following chapters if interested.

1. Describe the most significant research I have worked on.
   A growing number of digitised historical data collections provide important opportunities for social scientists to study historical records and generalise rules and patterns about society. Automatic coding historical records, for example historical occupations or causes of death, into standard systems can help social researchers to reduce a significant amount of manual work and study historical populations in more detail. The goal of this work is to develop text classification techniques that can deal with sparse and rare data, and evaluate these techniques on real-world data collections.

2. How I approached key technical challenges.
   This thesis focuses on three main steps of automatically classifying historical coding systems. First, due to the low data quality and sparse data, a variety of data pre-processing techniques are explored, including data cleaning methods, feature generation schemes, and dimensionality reduction techniques. Second, for the multi-class and multi-label classification tasks, we train one classifier for each class using classification techniques including naive Bayes, logistic regression, support vector machines and decision trees. Third, different evaluation techniques and metrics are investigated for their applicability within this imbalanced classification problem.

   We use two datasets collected from Scotland from two domains. The cause of death dataset contains 23,564 records with 591 unique death codes; while the occupation dataset contains 64,063 records with 418 unique occupation codes. We achieve a best classification performance with 0.78 precision, 0.66 recall and 0.65 F-measure on the cause of death dataset; and 0.93 precision, 0.94 recall and 0.85 F-measure on the occupation dataset.

   We implement our algorithms in Python.

3. What I gained from the experience.
   Compared with previous work, our approaches have achieved a significant improvement in terms of classification performance. Furthermore, our approaches can be successfully applied in the two data domains, which shows the methods we have developed can be successfully generalised.

   By doing this research project, I become familiarised with the various stages of a real-world supervised machine learning project, and am able to identify and

apply existing tools (e.g., sklearn) to classify text data using a variety of established algorithms (e.g., SVM, decision trees), as well develop novel data preprocessing and feature construction methods. Besides, I am able to implement string matching and data wrangling techniques to transform text data in order to improve classification performance (e.g., accuracy).

Contents

List of Figures

List of Tables

Introduction

1.1 Problem Statement

Social scientists are provided with an increasing number of research opportunities based on a growing number of digitised historical data collections. Coding historical records, such as causes of death or occupations, to standard classification systems can help researchers to study the relationships between different historical factors and produce multi-generational studies [Connelly et al. 2016]. This leads to an increasing need for studying the original historical records using advanced methods.

Researchers in Scotland aim to classify 8.3 million causes of death records and 50 million occupation titles into the International Classification of Disease (ICD-10 classification) [WHO 1990] and Historical International Standard Classification of Occupations (HISCO) [Van Leeuwen et al. 2002] respectively. The collected data was handwritten and manually entered into databases. The samples of collected cause of death and occupation data in Scotland are shown in Figures 1.1 and 1.2.

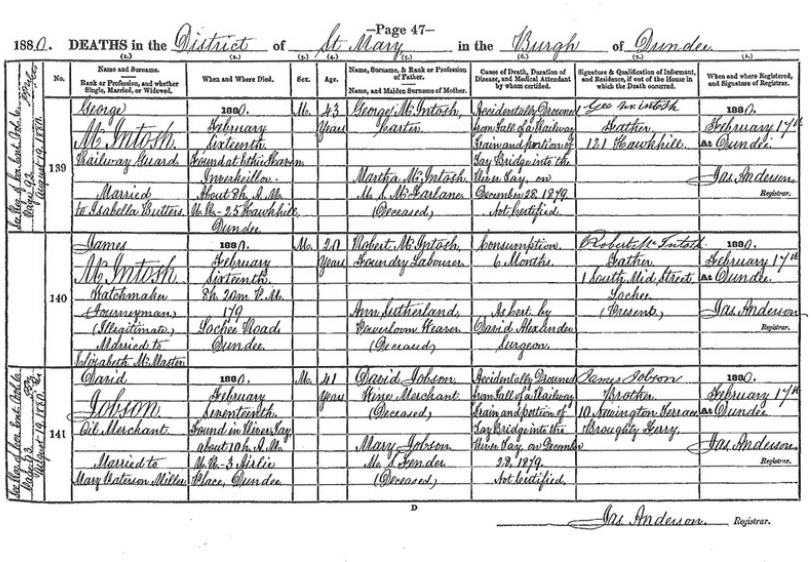


Figure 1.1: Sample image of cause of death records (third column from the right) [National Records of Scotland 1980].

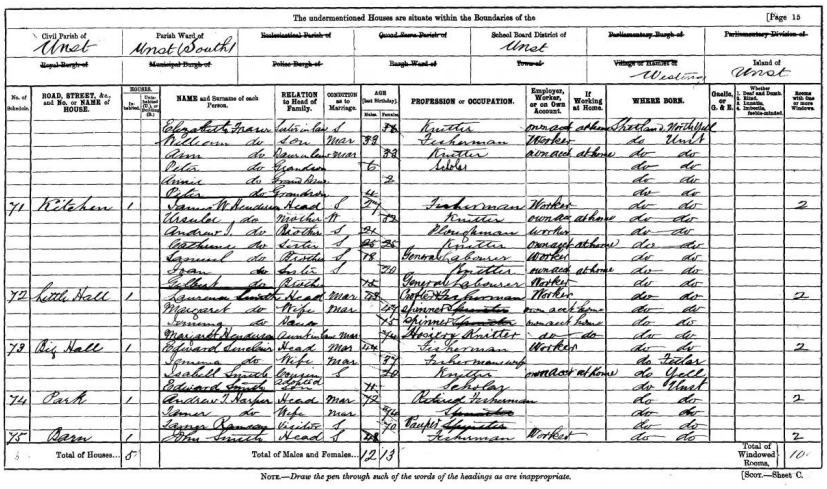


Figure 1.2: Sample image of occupation records (middle column) [National Records of Scotland 1901].

For our experiments in this project, we chose two subsets of the mentioned cause of death and occupation datasets used by Scottish researchers. We explore different methodologies for their suitability on both of these datasets. Our methodologies can then be used by the Scottish researchers and help them to accurately and efficiently classify their large datasets mentioned above.

This problem of automatically classifying historical records into modern standard classification systems is challenging for the following reasons:

- The historical data is noisy. The text descriptions are not standardised, and include many abbreviations and narrative descriptions, as shown in Table 1.1. Furthermore, the scanned copies of handwritings (e.g. Figures 1.1 and 1.2) are not clear enough and not easy to read, which leads to many typographical errors in the digitised historical datasets.

- Data sparsity is caused by highly skewed distributions of classes and text length. For some classes, only a few training records are available (e.g. for the cause of death dataset, there are 92 classes (codes) with only 1 training record), which makes it difficult to apply certain machine learning approaches for our problem.

- We face the human coding inconsistency problem, where the same text record might be manually classified into different classes by different domain experts.

- Since our approaches may eventually be applied to a large dataset in Scotland, classification algorithms should be efficient and scalable to large datasets.

Applying machine learning methods to the historical administrative data is novel and only a few previous relevant publications are available. For example, Carson et al. [2013] and Kirby et al. [2015] explored data pre-processing and machine learning methods on the same datasets as we used, and they found that naive Bayes has the

best performance. For further details about their work see Section 2.3. However, their approaches only achieved 0.4 F-measure for the cause of death dataset and around 0.6 precision/recall for the occupation titles dataset, which are not good enough for accurately classifying the historical records. This promotes us to explore better techniques for data pre-processing and classification for historical administrative records.

1.2 Data Description

The target datasets used in this research contain historical descriptions recorded in Scotland from two domains: causes of death and occupation titles. We will use the abbreviation *COD* for the cause of death dataset and *OCC* for the occupation dataset throughout this thesis. All datasets have been coded into standard classification schemes by expert historians [Carson et al. 2013; Kirby et al. 2015]. A cause of death string may contain more than one causes, and we consider the first one in the list to be the primary cause. Personal details are not available in our dataset. We briefly introduce the two datasets we used in following and will describe them in more detail in Section 4.1.

Cause of death dataset (COD):

This dataset contains a total of 23,564 records, split into a training dataset (18,877 records) and a testing dataset (4,687 records), from Kilmarnock in Scotland and covers the period from 1861 to 1901. See [Carson et al. 2013] for more information about this dataset. In the training dataset one record is assigned to one ICD code while in the testing dataset one record is assigned to between 2 to 4 ICD codes. Tables 1.1 and 1.2 show some example training and testing records from this dataset.

ID	Cause of death	Code (ICD-10)
785	hc	A37.90
7425	cancer of stomack	C16.90
14832	saw the child just before he died and am of opinion he died from fracture of base of the skull	Y34.01

Table 1.1: Example *COD* training records.

ID	Cause of death	Code_1	Code_2	Code_3	Code_4
2848	1. disease of prostate gland 2. retention of urine & disease of bladder 3. irritation fever	N42.90	R34.02	N32.90	
13027	both bron & asthma, childbirth & enteritis	J40.00	J45.90	O95.00	A09.01
18950	both flu; syncope	J11.10	R55.00		

Table 1.2: Example *COD* testing records.

Occupation dataset (OCC):

Sourced from the Cambridge Family History Study for the period from 1714 to 1995 [Bottero and Prandy 2001], this dataset contains 243,669 occupational titles that were originally coded to SOCN, which is an extension of the SOC (Standard Occupational Classification) coding system [US Bureau of Labor Statistics 2010]. 64,063 of these records are manually coded to HISCO [Van Leeuwen et al. 2002] according to a fixed SOCN-HISCO mapping. See [Kirby et al. 2015] for more information about this dataset. We use these 64,063 records as our experimental dataset. One dataset is given for occupation titles, where each record is assigned to one HISCO code. Table 1.3 shows examples of occupation records.

ID	Occupation Titles	Code (HISCO)
1176	mp	2-0
1463	governor (rear admiral rn) of naval hospital	2-0
2123	scientific assistant, meteorological office, air m	1-30.00
4431	master mariner/ship owner	4-12.50

Table 1.3: Example *OCC* records.

1.3 Objectives

The objectives of this project are to use the real historical death and occupation datasets described in the previous section to develop text classification techniques that can deal with sparse and rare data, and evaluate these techniques on large real-world data collections. Specifically, due to low data quality and data sparsity, a variety of data preprocessing and feature generation schemes need to be explored for their suitability to classifying such data, and because of possible multiple causes in a single death or occupation description, multi-class as well as multi-label classification techniques are employed.

1.4 Research Methodology

We use the research methodology shown in Figure 1.3 in our research. The detailed approaches in each step are as follows:

1. Preliminary Study: Learn about basic techniques for text processing and classification. Understand the basic social science background for our research.

2. Set Research Goals: Based on the preliminary study and literature review, we formulate research problems and set primary goals. These research goals may change during the research process.

3. Literature review: This step includes two parts: one is to review concepts and techniques used to solve research problems, including data pre-processing and

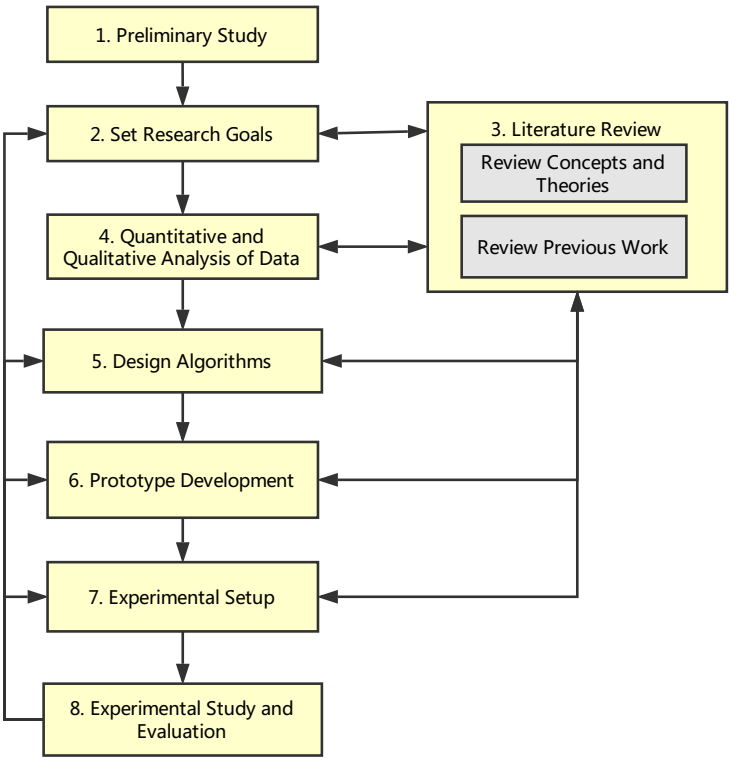


Figure 1.3: The research methodology used in this project.

machine learning techniques; while the second part is to review previous related research publications. We mainly review [Carson et al. 2013] and [Kirby et al. 2015] that performed the classification task on the same *COD* and *OCC* datasets.

4. Quantitative and Qualitative Analysis of Data: We analyse the two datasets in terms of the number of records, codes, and the quality of data items, which determines which further methods we can use for our research. The dataset characteristics are shown in Section 4.1.

5. Design Algorithms: Based on the conducted data analysis and literature review, we develop algorithms that suit the research data.

6. Prototype Development: We design the processes of research, including data pre-processing, classification and evaluation (described in detail in Chapter 3).

7. Experimental Setup: Experiments are conducted using Python 2.7.14 including the Numpy [1], Pandas [2], Scikit-learn [3], and Matplotlib [4] libraries.

8. Experimental Study and Evaluation: We use precision, recall and the F-measure for evaluation (see Chapter 4 for details). Based on the obtained evaluation results, algorithms and prototype are re-designed.

1.5 Thesis outline

In this thesis, we will show the background of this work in Chapter 2, which includes the general social science background for historical coding and the state-of-art of classification techniques. Chapter 3 describes the methodologies we developed for historical coding classification, including the data pre-processing techniques, classifier constructions and evaluation methods. Chapter 4 discusses the dataset characteristics and experiments on the evaluation of those classification techniques. Conclusion and future work are provided in Chapter 5.

[1] http://www.numpy.org/
[2] https://pandas.pydata.org/
[3] http://scikit-learn.org/stable/
[4] https://matplotlib.org/

Chapter 2

Background

This chapter provides an overview of the general social science background and the current state-of-art approaches to text classification. Relevant literatures for historical cause of death (*COD*) and occupation (*OCC*) dataset are closely reviewed.

2.1 Social Science Background

Social scientists have been studying historical administrative records for a long time, where administrative data is defined as information collected for the purposes of registration, transaction and record keeping [Woollard 2014]. For example, registers' information, such as notifications of birth, deaths, marriage, education and occupation, can be a type of administrative records. The administrative data can be large and complex quantitative information and primarily be generated for a purpose other than research [Connelly et al. 2016]. Thus for research purposes, the historical records need to be assigned into standard categories, but much of the work involved in assigning those records into categories has been manual.

In recent years we have witnessed the application of machine learning and other computer science methods to address these kinds of problems [Reid et al. 2015]. For example, administrative social science data are likely to be more complex than social survey data which researchers may be familiar with [Connelly et al. 2016]. The data can be noisy, so data cleaning approaches and management techniques can be applied to organise the data into a required format. Similarly, to analyse and examine patterns in administrative data, the original data usually need to be categorized into a required coding system or be clustered into several groups. The machine learning classification or clustering techniques can be explored for the suitability for these kinds of data.

These techniques allow social scientists to automate some parts of their research and methodologies, and of course to also use computational techniques for analysis (e.g. data science). Re-using historical records can afford exciting new opportunities for social science research, although the combination of data science and administrative data analysis have been under-appreciated by the research community [Connelly et al. 2016]. Social scientists are now increasingly interested in how computer science methods can be well integrated and used in their field.

2.2 Encoding Historical Classification System

In this section, we will briefly introduce the historical classification coding systems used in our work: the cause of death coding system (ICD-10), and occupation coding system (HISCO).

2.2.1 Cause of Death: ICD-10

To produce a comparable cause of death statistic, the disease classification system needs to be developed to assign morbid entities according to the established criteria. In 1853, the first International Classification of Diseases (ICD) was first introduced at the First International Statistical Congress, with the aim of permitting systematic recording, analysis, interpretation and comparison of cause of death records collected from different regions and times [WHO. 2004].

The ICD has eleven revisions since it was first published, where the latest revision (ICD-11) was published at the beginning of 2018. The tenth revision (ICD-10) was published in 1989 and used from 1995 to 2017. The *COD* dataset was classified based on ICD-10, so we mainly introduce the characteristics of the tenth revision.

The ICD-10 is used to convert diseases from words into alphanumeric codes, which permits easy storage, retrieval and analysis of the dataset, and become the international standard diagnostic classification scheme [WHO. 2004]. The ICD-10 has a hierarchical classification scheme, where the basic structure is as follows [WHO. 2004]:

- Chapters: The classification includes 22 chapters, where Chapters I - XVII relate to diseases and other morbid conditions, Chapter XVIII relates to injury, poisoning and other external causes, and other chapters cover the matters included in diagnostic data. One example of chapter coding is ”VI Diseases of the nervous system”.

- Three-character categories: The first character of the ICD code is a letter, followed by numbers. The three-character categories represent the main categories, for example, ”A00” (Cholera).

- Four-character categories: Most of the three-categories are subdivided with a fourth, numeric character after a decimal point. For example, ”A00.0” (Cholera due to Vibrio cholerae 01, biovar cholerae).

2.2.2 Occupation Titles: HISCO

Similarly to the *COD* data, the historical occupation titles (*OCC*) need a standard classification system to help historians perform comparative work. The Historical International Standard Classification of Occupation (HISCO) was published for this purpose, which makes the occupational classification consistent and easy to analysis [Van Leeuwen et al. 2002].

The HISCO was developed based on the existing scheme, ISCO68 ((1968 International Standard Classification of Occupations)) and three subsidiary classification

schemes. The HISCO structure is also hierarchical. A total of 1,881 occupation categories are grouped into:

- Major Group: a total of 10 major groups are in the most generalised level, represented by a single number. For example, the 4th group represents "Sales Workers".
- Minor Group: a total of 83 minor groups are the subgroups of major groups, represented by two numbers. For example, 4-1 represents "Working Proprietors".
- Unit Group: the unit group has two levels; the higher level is represented by three numbers, such as 4-10 ("Working Proprietors (Wholesale and Retail Trade)"); and the finer level is represented by more than three numbers, such as 410.20 ("Working Proprietor (Wholesale Trade)"). There are a total of 284 unit groups in the finest level.

2.3 Related Work

In this section, we focus on papers about developing a method for automatically coding the cause of death (*COD*) [Carson et al. 2013] and Scottish historical occupation titles (*OCC*) [Kirby et al. 2015] to standard classifications. Our research project engages with a similar research problem to these two papers.

The COD Dataset
Firstly, Carson et al. [2013] explored the *COD* dataset with three approaches:

1. Parsing using regular expressions: a simple parser was used to extract substrings equivalent to the desired coding. For example, an original narrative description "injury caused by being run over by a railway truck" can be parsed as "injury", which is corresponding to the standard historian coding "injury". However, this approach cannot deal with the cases where classification includes phrases not in the original record.
2. Natural language processing: More complex grammatical analysis was conducted using the OpenNLP toolkit [Apache Software Foundation 2010]. But it was hard to map parse trees to 'gold-standard' coding.
3. Machine learning: both individual and ensemble methods were tried.
   - Individual classifiers:
     - Stochastic Gradient Descent (SGD)
     - Naive Bayes
     - Complementary Naive Bayes
   - Ensemble Method [Dietterich 2000]: combine the decisions of individual classifiers

Although the parsing classifier turned out to give a rather low accuracy(44%), it supplies us with a way of data pre-processing. This kind of simple data cleaning of input data can improve the SGD classifier's performance. Before classification, it is crucial to study the original input data, figure out its features and then design an appropriate parser for it.

Carson et al. [2013] obtained accuracy from 72%-96% on several test sets. It turns out the ensemble with confidence-decided selection gave the best performance [Carson et al. 2013]. The result shows for the historical short text strings classification problem, parsing and natural language processing perform worse than machine learning techniques.

The researchers also conducted experiments to compare classification performance on unique record datasets with full datasets. As expected, unique record datasets returned a poorer result since the advantage of strengthening by redundant records in the training dataset would be lost. Carson speculated that performance on the unique data might be improved if the duplicate records were known [Carson et al. 2013].

However, we think there are some limitations in this work:

- The reason for choosing one classifier rather than others is not explained in sufficient detail. In this way, readers cannot have a clear mind why these classifiers are worth trying and why the performance of these classifiers should be better than others.

- Carson et al. chose accuracy (i.e. the percentage of correctly classified records among all the records) as the performance measurement. But the accuracy does not suit for evaluation on imbalanced classes. Simply classifying all records into the large proportion class can lead to a good accuracy. In the *COD* dataset, the class distribution is highly skewed and only a small proportion of records belonging to the positive class. Thus, other performance metrics should be considered, such as precision and recall [Marina and Guy 2009]. We will discuss these further in Section 2.4.3.

- In experiments, parsing pre-processing only marginally improved model performance. More data pre-processing methods should be explored, such as a variety of data cleaning, dimensionality reduction and feature selection approaches.

- It is also reasonable to evaluate some other classifiers which are suitable for text classification, like support vector machine (SVM) [Joachims 1998], decision tree [Breiman 2001; Baoxun et al. 2012].

Kirby et al. [2015] showed the following research results evaluated by precision, recall and F-measure, where on the Kilmarnock dataset, they achieved 0.84 precision and 0.4 recall.

The OCC Dataset

In subsequent research, Kirby et al. [2015] performed experiments on the *OCC* dataset. They first analysed occupation data and decided the techniques will be used based on the analysis. It would not be necessary to perform the cleaning before classification, as occupation descriptions only contain few words. Furthermore, human coding inconsistency, i.e. the same occupation might be classified into different categories, was dealt with by discarding inconsistent titles. Apart from that, due to the high dimensionality of the text records, Kirby et al. [2015] used feature selection method (X^2 statistic(CHI) [Yang and Pedersen 1997]) to reduce the dimensionality.

The same classification methods may have different performance on different datasets, so it is essential to analyse the content and structure of datasets and perform data pre-processing before classification. Without appropriate data pre-processing, we are unlikely to obtain a good classification result.

For the *OCC* dataset, Kirby et al. [2015] tried two main approaches:

1. Edit Distance Classifier: The idea is to select the occupation descriptions with the smallest distance from the standard class. But this approach performs badly for occupation titles which occur as a substring embedded in the definition string

2. Machine Learning Classifier:

    - Individual Classifiers: Logistic regression using stochastic gradient descent and naive Bayes.
    - Ensemble Approaches: Majority voting and confidence-weighted.

As a result, the best performance is achieved by individual naive Bayes classifier with cleaning multiple-coded descriptions. This yielded the 61% precision and 66% recall [Kirby et al. 2015]. This bad performance may be caused by lack of training records. Ensemble methods did not produce any improvement compared with individual Naive Bayes. The reason may be that the performance of Naive Bayes is better than other classifiers for most codes.

Kirby et al. [2015] also explored the 'potential of new occupational titles' in future research, such that whether their study are practical if there are more and more occupation titles being generated over time. This part of the paper highlights how the research is dedicated to a real-world problem, which makes the research more practical and attractive.

After studying the related project, in this project we will concentrate on the following aspects:

- Quantitatively and qualitatively analyse the original datasets and find out what kind of data pre-processing methods can be used to improve the performance;
- Experimentation with data cleaning measures, such as spelling correction;
- Try different data matching methods for measuring data similarity;

- Experimentation with feature selection methods and how they can influence the classification performance and computational time;
- Explore the suitability in terms of different feature types and classifiers;
- Evaluation on both the *COD* and *OCC* datasets to investigate the generalisation of the developed approaches.

2.4 Data mining Techniques Explored

In this section, we will introduce the general data mining techniques for text classification task. The main pipeline is data pre-processing, classification and evaluation, which are shown in Section 2.4.1 to 2.4.3.

2.4.1 Data Pre-processing

Before passing the text collections to a classifier, the original data should be represented to certain format since they cannot be directly interpreted a classifier-building algorithm. Data pre-processing covers mainly three aspects, namely data cleaning, feature extraction and dimensionality reduction [Sebastiani 2002; Christen 2012].

2.4.1.1 Data Cleaning

Data quality directly influences classification performance. The main aim of data cleaning is to improve consistency and quality of the original data. If perfect quality data is processed for data matching, we only need simple indexing and comparison methods to represent the data. Otherwise, sophisticated techniques may be needed to achieve a good evaluation result. The following aspects could be considered [Christen 2012, Chapter 3]:

- Improving data quality.

  We need to smooth the noisy data and correct the inconsistent data to improve the data quality. Noisy data does not supply useful information for the classification task and may interfere with it. Conventional methods to clean noisy data include the following:

  - Correct spelling errors. For example, in the *COD* dataset, "abscess" is misspelled as "absess". One method is to use dictionaries to detect misspelled words and correct them. If there are no proper dictionaries, for example, when targeting proper noun database, including names, addresses and special abbreviations, this issue can be dealt with through regression or clustering approaches that replace all similar strings in a group by the most frequent string or cluster centroid of this group. Mazeika and Bohlen [2006] use inverse strings together with sampling to compute the centre of a group of strings and the border of the group.

  - Remove unwanted characters and tokens. Unwanted characters and tokens refer to those elements which do not contain useful information for data matching and classification. For example, punctuation marks, stop words.
  - Identify and correct inconsistent values. Same text description may be manually coded into different classes. For example, human coding inconsistency appears in the *OCC* dataset, "fireman" is classified into "railway steam-engine fireman"(37%), "fire-fighters"(26%) and "boiler fireman"(13%) [Kirby et al. 2015]. The possible ways are to re-classify all "fireman" into the most popular one ("railway steam-engine fireman") or discard all related records. However, if we cannot make sure which one of those categories is the correct one, then any change of original data may introduce further mistakes rather than correct the data.

- Standardisation, tokenisation and stemming. Standardisation usually needs an extensive look-up table which supplies standardised values for tokens. Tokenisation assigns each token with one or more tags which specify the type of token. Stemming is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form [Willett 2006]. For example, a stemming algorithm reduces the words "fishing", "fished", and "fisher" to the root word, "fish". Although stemming can reduce the dimensionality and dependence between terms, it also has been reported to reduce model performance [Baker and McCallum 1998].

2.4.1.2 Feature Generation

A classification algorithm cannot directly process text documents, thus an indexing procedure is needed to map a document into a representation format. There are two typical choices for representation of documents: one is the meaningful units of text (lexical semantics), the other one is the meaningful natural language rules for the combination of these units (compositional semantics) [Sebastiani 2002]. More precisely, a document d_i is represented as a vector of feature weights $\vec{d_i} = < w_{i1}, ..., w_{in} >$, where n is the length of feature sets. Each feature occurs in at least one document. Different approaches for feature generation will depend on two aspects [Sebastiani 2002]:

1. Features types.

   A typical choice is the *bag of words* approach, where words are identified features. For example, a text description "heart disease" can generate two features based on the *bag of words* approach: "heart" and "disease".

   More generally, the n-gram based indexing is considered. An n-gram is a contiguous sequence of n items from a given text record. The items can be phonemes, syllables, letters words or base pairs [Christen 2012]. Common choices for n are n=1 (called *unigram*), n=2 (called *bigrams* or *digrams*) or n=3 (called *trigrams*) for word level, character level respectively. A sliding window approach is used to

extract n words or characters from string s at any position from 1 to the number of grams. The extracted terms are called *features*. For example, the trigrams for character level list for "mengyan" is ['men','eng','ngy','gya','yan'].

Other approaches such as soundex (generating grams based on American-English language pronunciation), canopy clustering (grouping similar records into the same cluster), and mapping based indexing (string comparison methods needed) are also valuable candidates for defining features [Christen 2012].

2. Term weights.

Different features have different level of importance in a textual corpus. The term weights are used to represent the degree of importance of a feature. There are several weighting methods commonly used.

- Binary. The binary weight of one feature is indicated by whether a feature appears in a record.
- Term Frequency(TF). Use the term frequency $tf_{t,d}$ to represent how many times a feature t appears in a record d.
- Document Frequency (DF). Use document frequency df_t to represent how many records containing a feature t. High document frequency indicates a feature is not important, like stop-words ("and", "but", etc.).
- Inverse document frequency(IDF). The IDF is the inverse of DF, which can be defined as follows:

$$idf_t = log\frac{N}{df_t}, \tag{2.1}$$

where N is the total number of records.

- TF-IDF. The weight TF-IDF combines TF and IDF, considering the degree of importance in terms of both the feature frequency in a records and in the whole corpus. The TF-IDF can be defined as follows:

$$tf - idf_{t,d} = tf_{t,d} \times idf_t. \tag{2.2}$$

2.4.1.3 Feature Extraction: Dimensionality Reduction

After generating features and weights according to the degree of importance from the textual corpus, feature selection and dimensionality reduction are generally considered to improve the classifier performance and provide faster and more cost-effective classifiers [Guyon and Elisseeff 2003].

One major problem of text classification is having a large number of features. On the one hand, some features contain irrelevant information that can mislead the classifier. On the other hand, large dimensionality can be high time and space complexity.

Recently in the area of text classification, researchers have explored feature selection methods to pick valuable features based on information gain, mutual information, and so on [Uuz 2011; Yang and Pedersen 1997], and dimensionality reduction

methods such as principal components analysis, projection pursuit, and independent component analysis [Fodor ; Jolliffe 1986].

We will introduce one particular dimensionality reduction method, principal components analysis (PCA), in more detail. The basic idea of PCA is to linearly project the records onto a lower dimensional subspace such that the variance of the projected data is maximized [Wold et al. 1987], where the variance measures how far a set of numbers are spread out from their average value. Given N observations $\mathbf{x_n} \in \mathbb{R}^D$, where n = 1,...,N; and some unit vector $\mathbf{u_1} \in \mathbb{R}^D$ satisfies $\mathbf{u_1}^T\mathbf{u_1} = 1$. Each data point $\mathbf{x_n}$ can be projected onto a scalar value $\mathbf{u_1}^T\mathbf{x_n}$. The variance of the projected data can be expressed as:

$$\frac{1}{N}\sum_{n=1}^{N}\left\{\mathbf{u_1}^T\mathbf{x_n} - \mathbf{u_1}^T\bar{\mathbf{x}}\right\}^2 = \mathbf{u_1}^T\mathbf{S}\mathbf{u_1}, \tag{2.3}$$

where $\bar{\mathbf{x}}$ is the sample mean and S is the covariance matrix.

If we maximize the Equation 2.3, we can find the optimal projectors. Assume the goal is to project N-dimensional data into M dimensions, the optimal projectors are M eigenvectors $\mathbf{u_1}, ..., \mathbf{u_M}$ of the covariance matrix S corresponding to the M largest eigenvalues [Wold et al. 1987].

2.4.2 Classification

Classification is a supervised machine learning task in which pairs of data and target labels are given. The goal of classification is to learn a mapping between the data and label which generalizes well to new data, more specifically, to assign input data to one of K discrete classes C_k where k = 1, ..., K. According to the number of classes K, the classification tasks can be categorized into *binary classification* (K = 2), *multi-class* and *multi-label classification* ($K > 2$).

For a text classification problem, a file containing text records with their labelled classes will be converted into a $N \times (D + m)$ data matrix X, where each row is a tuple $\{(\mathbf{x_i}, \mathbf{c_i}) \big|\ \mathbf{x_i} \in \mathcal{X}, \mathbf{c_i} \in \mathcal{Y}, 1 \leq \mathrm{i} \leq N\}$ with D dimensional *feature vector* and m *class labels*. $\mathcal{X}$ is the set of input *feature vectors* and $\mathcal{Y}$ is the set of *class labels* vectors. There are some explanations for related concepts:

- **record**: also called *data point* or *instance*. $\mathbf{x_i} = \{x_{i1}, x_{i2}, ..., x_{iD}\}$ in each row is called one *record*. The data matrix X totally has N *records*.
- **feature**: also called *attribute*. In machine learning, a *feature* is defined as "an individual measurable property or characteristic of phenomenon being observed" [Bishop 2006]. Each *record* has D dimensional *features* in the data matrix.
- **label**: also called *class* or *code*. Each *record* is assigned to m *labels* $\mathbf{c_i} = \{c_{i1}, c_{i2}, ..., c_{ij}..., c_{im} | c_{ij} \in \mathcal{C}\}$, where $\mathcal{C}$ is the set of *class labels*.

  For multi-class classification, each *record* corresponds with only one *label* (m = 1). For a *multi-label classification* problem, multiple labels can be assigned for one record ($m > 1$). The type of classification tasks and corresponding methods to solve these tasks are introduced in Section 2.4.2.1.

- **record-feature matrix**: $N \times D$ matrix where each row is a *record* with D dimensional *features*.

Generally, there are two phases for a classification problem: training phase and testing phases. If only one input dataset is given, it will be separated into two or more datasets for training and testing phases respectively. The separation methods will be introduced in detail in Section 2.4.3.

1. **Training phase:** The goal of the training phase is to construct a classification model, which can be viewed as a function f(**x**) that can predict the class label for unobserved *record* **x**. The model is commonly called a *classifier*.

2. **Testing phase:** The goal of the testing phase is to predict *labels* for test (unobserved) *records* using the trained model, then compare predicted labels with true labels for test *records* to estimate the accuracy of the model. If the accuracy is acceptable, we can use the model to classify new *records*.

2.4.2.1 Types of Classification Tasks

Classification tasks have different types in terms of the number of classes and how many classes one record is assigned to. Figure 2.1 shows the types of classification tasks.

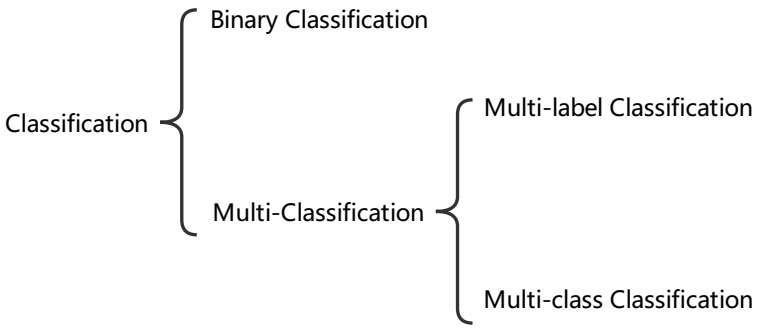


Figure 2.1: Type of classification.

According to the number of classes K, classification tasks include *binary classification* (K = 2), *multi-class* and *multi-label classification* ($K > 2$). When K $= 2$, training records are called *positive records* and *negative records* in general. When $K > 2$, if one record is assigned to only one class, the task is called *multi-class classification*; otherwise *multi-label classification*. We will introduce the standard methods for *multi-class* and *multi-label classification* in following.

For multi-class classification, there are three general strategies [Aly 2005]:

- Transformation to binary.

  We can reduce the multi-class classification problem to multiple binary classification problems. The main approaches include *one-vs.-rest* and *one-vs.-one*.

– *one-vs.-rest*: train K single classifiers, where one classifier for per class, with the records of that class as positive records and all other samples as negatives [Bishop 2006]. Let f_i be the ith classifier, classify with:

$$f(x) = \arg\max_i f_i(x), \tag{2.4}$$

– *one-vs.-one*: train $K(K-1)$ classifiers, where one classifier distinguishes each pair of class i and j [Bishop 2006]. Let f_{ij} be the classifier where records for class i are positive and records for class j are negative. We classify with:

$$f(x) = \arg\max_i (\sum_j f_{ij}(x)). \tag{2.5}$$

- Extension from binary.

  We can extend existing binary classifiers to multi-class classifiers. For example, naive Bayes, decision trees, support vector and neural networks have been developed to address multi-class classification problems [Rennie 2001; Takahashi and Abe 2002; Anand et al. 1995].

- Hierarchical classification.

  Multi-class classification can also be solved by dividing the output space into a tree structure. Each parent node is divided into multiple child nodes and the process is continued until each child node represents only one class [Kowsari et al. 2017].

For multi-label classification problems, the most common method is the transformation method. The baseline of this method is called *binary relevance method*, where one binary classifier is trained for each label independently. For an unobserved record, the combined model then predicts all labels for the record where the respective classifier predicts a positive result [Read et al. 2011]. Other approaches involve adapted algorithms, where the binary classification algorithms are adapted to the multi-label tasks [Chen et al. 2003].

In summary, no matter whether it is *multi-class* classification or *multi-label* classification, the strategies are based on the basic *binary* classification algorithms. In next section, we will introduce the popular *binary* (or extension *multi-class* version) classification algorithms for text classification problems.

2.4.2.2 Classifier Construction

The core aim of classification tasks is to build a robust classifier which can accurately predict unobserved records. We explore several classifier construction algorithms which are widely used in text classification [Joachims 1998; Kim et al. 2006; Lewis and Ringuette 1994; Genkin et al. 2007], namely naive Bayes, logistic regression, support vector machine and decision tree.

- **Naive Bayes:**

  A naive Bayes classifier is a probabilistic classifier which predicts the conditional probability of a given *record* belonging to a particular *class label*. We first define several concepts commonly used in Bayes' Theorem:

  - *Evidence* **x**: observed *record* which has D dimensions,
  - *Hypothesis H*: a *class label*,
  - *Prior P(H)*: the a priori probability (Hypothesis) of H,
  - *Likelihood P(***x***|H)*: the conditional probability of observing the *record* **x** given the hypothesis holds,
  - *Posterior P(H|***x***)*: the posteriori probability where the hypothesis holds given observed *record* **x**.

  According to Bayes' Theorem [Stuart 1994]:

$$P(H|\mathbf{x}) = \frac{P(\mathbf{x}|H) \times P(H)}{P(\mathbf{x})}, \tag{2.6}$$

  which can be explained as,

$$Posterior = \frac{Likelihood \times Prior}{Evidence}.$$

  The Bayes classifier predicts one *record* $\mathbf{x_i}$ belongs to one *class label* c_j if the posteriori probability $P(c_j|\mathbf{x_i})$ is the highest among all other $P(c_k|\mathbf{x_i})$ for all K classes, where k $\in$ {1,..., K}.

  To compute the *Likelihood P(***x***|H)*, the naive Bayes classifier assumes *class conditional independence*, which assumes each feature is conditionally independent from all other features. The assumption can be expressed as Equation 2.7. Although the *class conditional independence* is commonly not true since features tend to have correlations with each other, it greatly reduces the computation cost and the naive Bayes classifier can generally work well for text classification problem based on this assumption [Kim et al. 2006],

$$\begin{aligned} P(\mathbf{x_i}|c_j) &= \prod_{d=1}^{D} P(x_{id}|c_j) \\ &= P(x_{i1}|c_j) \times P(x_{i2}|c_j) \times ... \times P(x_{iD}|c_j). \end{aligned} \tag{2.7}$$

  For *discrete* features, each item $P(x_{id}|c_j)$ in Equation 2.7 can be calculated by the percentage of feature values' occurrence. For *continuous* features, a binning procedure can be used to make features discrete [Kotsiantis and Kanellopoulos 2005]. An alternative method is to assume feature distribution, where the common choices can be Gaussian distribution [John and Langley 1995], Multinomial distribution [Murphy 2006] and Bernoulli distribution [Murphy 2006].

The naive Bayes classifier is easy to implement and fast. It needs less training data compared to some more complicated algorithms like Neural Networks [Lai et al. 2015]. The classifier is also not sensitive to irrelevant features since the irrelevant features are likely to give similar conditional probability if we have enough data. However, the strong *class conditional independence* assumption may have a negative influence on the performance in some cases.

- **Logistic Regression:**

  Contrary to what the name suggests, logistic regression is not a regression problem but a probabilistic classification model which outputs the probability that a given input record belongs to a certain class.

  Equation 2.8 shows the *logistic function* (also called *sigmoid function*), which produces a logistic curve, which maps the numeric number into a value between 0 and 1. Figure 2.2 show the logistic function curve.

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e} \tag{2.8}$$

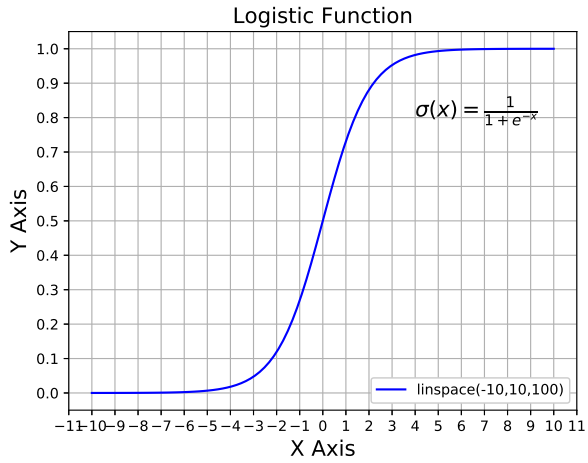


Figure 2.2: Logistic function curve. The horizontal axis sample numeric number, and the vertical axis is the corresponding logistic regression function value.

Equation 2.9 shows a linear model, where $\phi(\mathbf{x})$ is some fixed feature space mapping and b is bias. Training data are N input vectors $\mathbf{x_i}$ ($1 \leq i \leq N$) with corresponding targets c_j where $c_j \in \{0, 1\}$ for binary classification. $y(\mathbf{x})$ can be mapped to the range 0 to 1 using the logistic function by assigning $z = y(\mathbf{x})$.

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b \tag{2.9}$$

Thus, logistic regression maps a *record* $\mathbf{x}$ in D-dimensional feature space to a value in the range between 0 to 1. Generally, if the mapped value is close to 1,

the *record* $\mathbf{x}$ is predicted as $c_j = 1$, otherwise, it is predicted as $c_j = 0$.

To measure the performance of the prediction, the likelihood function can be used to show the probability that a random record is correctly classified:

$$p(\mathbf{c}|\mathbf{w}) = \prod_{i=1}^{N} {y_i}^{c_j}(1 - y_i)^{1-c_j}, \tag{2.10}$$

where $y_i = y(\mathbf{x_i})$.

So the task of a logistic regression classifier is to maximize the (log) likelihood, that is to maximize the probability that a new record will be correctly classified. This task can be handled by using gradient descent-based methods [Meier et al. 2008], where first order derivative gives the direction the search should take.

- **Support Vector Machine:**

  Support Vector Machine (SVM) is a classification method for both linear and non-linear data [Gunn et al. 1998]. Non-linear data can be linearly separable in a higher dimension, so a mapping method called *Kernel Method* can be used to transform the training data into a higher dimension if the data is non-linear.

  SVM then searches for a *hyperplane* (i.e. *decision boundary*) which can linearly separate classes and minimize the classification error on unobserved data. We define a *margin* as the smallest distance between the hyperplane and any of the records. SVM searches for the hyperplane with the largest margin. The *Support vectors* are defined as the training records that determine the largest margin hyperplane. Then for a binary classification task and the linear model defined as Equation 2.9, the training data are N input vectors $\mathbf{x_i}$ with corresponding label $c_i \in \{-1, +1\}$. We solve the maximum margin by Equation 2.11.

$$arg \max_{\mathbf{w},b} \{ \frac{1}{\|\mathbf{w}\|} \min_i [c_i(\mathbf{w}^T \phi(\mathbf{x_i}) + b)] \}. \tag{2.11}$$

  SVM is effective on high-dimensional data, since it is the number of support vectors rather than the dimensionality of the data that determines the complexity of a trained classifier. The support vectors alone can generate the same hyperplane without other training samples [Han et al. 2011]. However the training process can be slow since finding support vectors consumes a great amount of time.

- **Decision Tree:**

  A decision tree classifies the labelled data by applying a tree of logical tests on features that partition the data into finer sets [Han et al. 2011]. We start with the set of training records and split it according to some splitting rules at each non-leaf node. The splitting process constructs a tree-like structure, in which the root node and each internal node represent a test on one feature and each leaf node holds a class label. Figure 2.3 shows a simple example of a decision tree model.

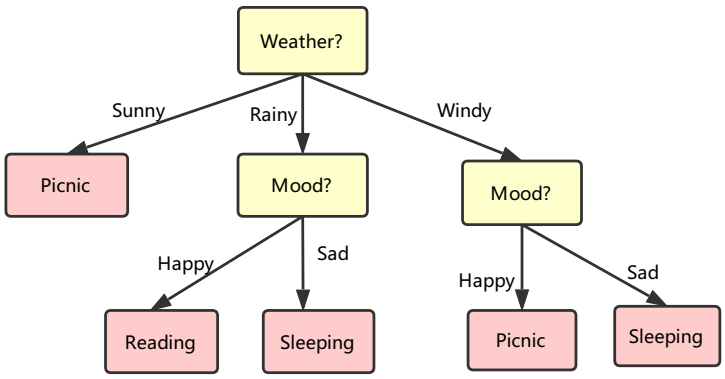


Figure 2.3: Example of decision tree model. Yellow nodes represent tests on the features *Weather, Mood*. Red nodes represent class labels *Picnic, Reading, Sleeping*.

The key point of decision tree model construction is the *feature selection method*, which decides how to split data. Generally, we need to decide the order of features to split on. Different splitting methods may result in different tree models. For example, in example shown in Figure 2.3, we can construct a different decision tree model by firstly splitting on "Weather" and then splitting on "Mood". Notably, if an attribute is continuous, then a splitting point also needs to be selected (normally using the midpoint).

The goal of *feature selection method* is to make each branch as pure as possible. Popular methods to select the feature includes using the largest *information gain* and smallest *gini impurity*, which will be described in more detail in Section 3.2.2.

Decision trees are a white box model and are easy to understand and interpret. We can extract logic rules from decision trees, for example, one rule of the decision tree shown in Figure 2.3 is "IF Rainy AND Happy THEN Reading". However, it is often possible to have an over-complex decision tree that doesn't generalize well to out-of-sample data, which is known as overfitting problem [Fox 2017]. Furthermore, a decision tree can be unstable since completely different trees can be created due to small variations in the data.

2.4.3 Evaluation

This section discusses how to evaluate classifiers. Firstly, several evaluation methods are introduced, including holdout methods, train-validation-test model and cross-validation. Following this, the evaluation metrics, such as precision and recall, are summarised and compared.

1. **Evaluation Method:** Evaluation measures cannot be tested on the training data because the classifier may be over-tuned for the training data. Thus when a dataset is given, it needs to be split for training and testing phases. There are several splitting methods:

   - Holdout method. Split a given dataset into two datasets: a training dataset for classifier construction and a testing dataset for evaluation.
   - Training/Validation/Test. A variation of the *holdout method* where an additional validation dataset is generated. The validation set is used to find the best parameter settings for the model.
   - Cross-Validation. The dataset is split into k mutually exclusive subsets $D_1, ...D_k$, where each subset approximately has equal size. At the i-th iteration, we use D_i for testing and the remainder for training. The overall performance is measured by the average performance of each model on its respective testing dataset.

2. **Evaluation Metrics:**

   Several evaluation metrics are commonly used for classification, namely *accuracy, error rate, sensitivity, specificity, precision, recall and F-measure* [Han et al. 2011]. To define them, we first introduce the *confusion matrix*, which illustrates how well a classifier can recognize records of different classes. Table 2.1 demonstrates the case for binary classification. Columns indicate the ground truth of class labels while rows represents the predicted class.

Total population	**Labeled Positive**	**Labeled Negative**
Predicted Positive	True Positive (TP)	False Positive (FP)
Predicted Negative	False Negative (FN)	True Negative (TN)

Table 2.1: Contingency table for binary classification

   - True Positive (TP): the number of records which actually belong to the positive class and are classified correctly into the positive class.
   - False Positive (FP): the number of records which actually belong to the negative class and are classified wrongly into the positive class.
   - False Negative (FN): the number of records which actually belong to the positive class and are classified wrongly into the negative class.
   - True Negative (TN): the number of records which actually belong to the negative class and are classified correctly into the negative class.

Table 2.2 shows possible evaluation metrics and corresponding formula.

Measure	**Formula**
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$
Error rate	$\frac{FP+FN}{TP+TN+FP+FN}$
Sensitivity/Recall	$\frac{TP}{TP+FN}$
Specificity	$\frac{TN}{TN+FP}$
Precision	$\frac{TP}{TP+FP}$
F-measure, where $\beta \in [0,1]$	$\frac{1}{\beta\frac{1}{precision}+(1-\beta)\frac{1}{recall}}$
F_1	$2 \cdot \frac{precision \cdot recall}{precision+recall}$

Table 2.2: Evaluation metrics

Accuracy indicates the percentage of correctly predicted records among all the records. However, *accuracy* is not suitable for imbalanced class problems. Simply predicting all records to the majority class label can lead to a good *accuracy*. *Error rate* is the opposite case of *accuracy*, showing the percentage of wrongly predicted records among all the records. For the same reason, *Error rate* is not suitable for imbalanced class problems. In this case, the *sensitivity* and *specificity* can be used to measure the correctly predicted records for the positive class and negative class respectively.

Precision and *recall* were originally developed for information retrieval [Larson 2009] and have been widely used in classification evaluation. *Precision* is a way to measure exactness, which is the percentage of records classified as belonging to the positive class that actually belong to the positive class. *Recall* is a way to measure completeness, which is the percentage of records labelled as positive class that are classified as positive.

F-measure is the trade-off between *precision* and *recall*. For the special case when $\beta = 0.5$, F_1 is the harmonic mean of *precision* and *recall*.

2.5 Chapter Summary

In this chapter, we first introduced the general social science background for historical administrative data classification in Section 2.1. Following this, the structure and characteristics of historical coding systems for both of the *COD* and *OCC* datasets were demonstrated in Section 2.2.

Two related works ([Carson et al. 2013] and [Kirby et al. 2015]) were closely reviewed in Section 2.3. Based on critically analysing these studies, we identified several key directions for our research. For example, Carson et al. [2013] showed the machine learning approaches were more suitable for the historical text classification problem

than other approaches, which provided a rationale and motivation to further explore machine learning approaches in this research project.

Following this, the general data mining techniques for text classification were discussed in Section 2.4. The main pipeline includes data pre-processing, classification and evaluation. Four classifiers were studied in detail, namely naive Bayes, logistic regression, support vector machine and decision tree.

In the next chapter, we will introduce the methodologies we used for the historical coding classification problem.

Chapter 3

Methodology

The main processes of classification of a historical coding system are data pre-processing, training a classifier and evaluation, which is shown in Figure 3.1. The blue boxes show detailed sub-steps of each part and we will describe the used methodologies in this chapter and evaluate them in Chapter 4.

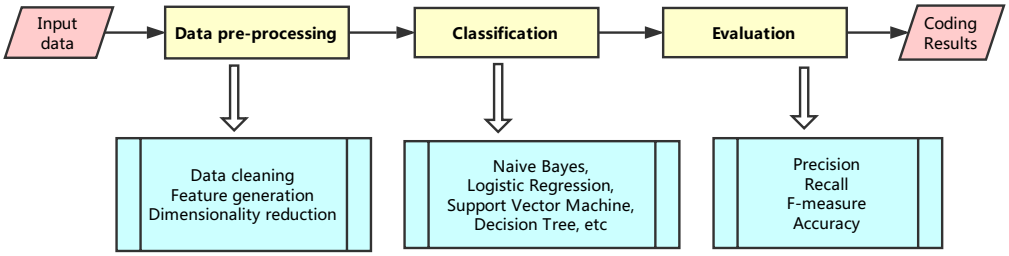


Figure 3.1: Historical coding classification process

3.1 Data pre-processing

Text data collected from real-world applications is often non-standard, inconsistent, sparse and contains errors, which largely influences the classification performance. Data pre-processing plays an indispensable role in solving such issues, where the aim is to clean the raw data and transform it into a matrix format that can be used to train a classifier.

The data pre-processing pipeline in this thesis includes three main Steps: data cleaning, feature generation (character and word n-grams) and feature extraction (dimensionality reduction). The historical *COD* and *OCC* datasets collected from Scotland have low data quality, including misspellings, inconsistency and non-standard descriptions. The data cleaning approach, including general cleaning, spelling correction and inconsistency correction, can increase the data quality to some extent. The

cleaned datasets can then be used for generating features, and a *record-feature matrix* can be built based on those generated features, where rows represent text records and columns are corresponded to features. Due to the sparseness of the data and variations of features, the generated *record-feature matrix* have a high dimension. Dimensionality reduction methods can be used for such situations to project useful features into some lower dimensional space, which may play a role in reducing computational time and increase classification performance. Each step of data pre-processing will be described in more detail in Sections 3.1.1 to 3.1.3.

3.1.1 Data cleaning

The goal of data cleaning is to improve the data accuracy and quality. A classifier learns the characteristic of classes based on the input data, and low quality data can directly lead to bad classification performance. So effective data cleaning is the foundation of the whole classification process.

Based on the study of previous work on data cleaning [Christen 2012; Mazeika and Bohlen 2006], which is discovered in Section 2.4.1.1, we choose to clean our historical text data by applying general cleaning (removing unwanted characters, stemming, etc.), spelling corrections, and inconsistency corrections.

1. **General Cleaning.**
   The purpose of general cleaning for raw data is to remove useless information. These noisy information can interfere the classification process and decrease a classifier's performance accuracy. We consider the following steps to conduct general cleaning:

   - Removing punctuation:

     Punctuation (such as comma, semicolon, etc.) generally do not contain any useful information, so we use a white-space to replace them.

     But a particular case for our data is that some punctuation like "&" and ";" can be used as indicators for separating several causes of death. For example, "&" in the *COD* description "british cholera & infantile diarrhoea" can separate the two causes "british cholera" and "infantile diarrhoea". If we just remove those punctuation we will mix up multiple causes' information and never be able to separate them if we need to. One alternative way is to use "and" to replace "&".

   - Handling white-spaces:

     Unlike punctuation, white-spaces might be useful since we can use white-spaces to split input strings into tokens. For example, we can split "heart disease" by white-space into ["heart","disease"].

     However, if there are multiple white-spaces between tokens or at the beginning or ending of a string, these white-spaces will also be extracted in features and provide useless information. For example, if one record is "␣␣heart␣␣disease" (where "␣" represents one white-space, that is there are

two leading spaces and two spaces between tokens), the generated bigram character level features can be ["_ _ ", "_ h", "he", "ea", "ar", "rt","t _", "_ _ ","_ d", ...], where white-spaces in "_ _ " and "_ h" would be noise information.

So instead of removing all white-spaces, we remove all leading and tailing white-spaces and double and triple white-spaces between tokens.

- Removing stop-words:

  Stop-words are defined as commonly and frequently used words that a classifier can ignore [Leskovec et al. 2014], such as "the", "that", "of", etc. These kinds of words are included in many text records and thus do not contain useful information expressing the characteristic of classes. Removing stop-words can decrease the degree of data noisy.

  Besides using a list supplied by the Natural Language Toolkit (NLTK) in Python [Loper and Bird 2002], we also tried a frequency-based approach, where if a token occurs in more than a certain percentage of all unique records, then we will regard the token as a stop-word. Since we think some frequently occurring words which are not in the NLTK stop-words list may also be noisy, such as "disease" for the *COD* dataset.

  However, how to decide whether a frequency word can be removed tends to be tricky. For example, removing frequently used words may not be suitable in a situation where text records are only several nouns rather than narrative descriptions. In the *OCC* dataset, "clerk", "labourer" occurs more frequently than words like "of". It is inappropriate to regard those words as stop-words and remove them since they may be the only word to describe a certain occupation class in one record.

- Stemming:

  Stemming is the process of converting words to their stem, base or root form by removing the derivation affixes [Willett 2006]. The returned stem is not necessarily to be the morphological root of a word. For example, "ponies" will be converted to "poni" rather than "pony".

  In our dataset, text descriptions use different forms of a word, such as "disease" and "diseases", which will become different features if we use "bag of words" to generate features, which is inappropriate since they provide the same information. Stemming can tackle this problem by unifying the different forms into their root form "disease" and reduce the number of unique features.

2. **Spelling correction.**

   After the general cleaning process, text records are now free of unwanted characters, including punctuation, stop-words and useless spaces. But there are still some misspellings which also have a negative influence on the classification performance.

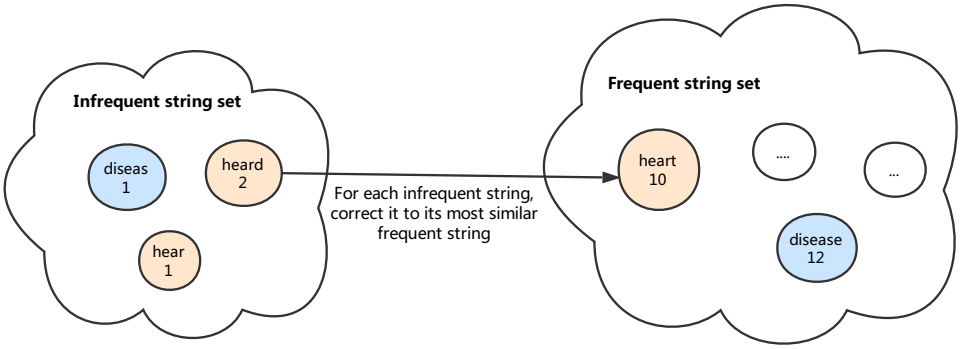


Figure 3.2: Frequent and infrequent set example. Strings are given with the frequency count with a frequency threshold 5.

Although spelling correction has been widely used for automatically writing check [Kukich 1992; Gail et al. 2016], we still face challenges dealing with cleaning misspellings for our historical dataset. On the one hand, we expect a spelling correction algorithm can detect which word is misspelled and then correct it without any manual feedback. On the other hand, since historical datasets, like our *COD* dataset, include many non-standard abbreviations and semantical phrases, a dictionary-based approach is not suitable. For example, if ”heart disease” is misspelled as ”heard disease”, it could be challenging to detect the misspelled word ”heard” because it is itself a meaningfully correct word.

To deal with such issues, we use a similarity-based approach to replace potential misspellings into their most similar string [Wagner and Fischer 1974]. We assume the potential misspellings tend to occur less often than the correct spellings. We can separate the input strings into a *frequent set* and an *infrequent set* according to a user-defined minimum *frequency threshold*. The strings in the *infrequent set* are potential misspellings while the strings in *frequent set* are assumed to be correctly spelled ones. Then for each word in the *infrequent set*, we correct it to its most similar word in the *frequent set*, if the similarity between them is larger than a user-defined *similarity threshold*. The parameter choices of *similarity* and *frequency* threshold are explored in Section 4.2.

Figure 3.2 shows the example for frequent and infrequent set. The world ”heard” only occurs twice since it is misspelled by accident, and we put it into the *infrequent set* due to its low frequency. After calculating the similarity between ”heard” and all others strings in the *frequent set*, the word ”heart” turns out to be the most similar one and their similarity is larger than the *similarity threshold* so that we can correct ”heard” to ”heart”.

The next major question then is how we can calculate the similarity between strings. There are many existing string matching techniques that can be used to calculate the similarity between strings [Christen 2006]. We choose two of them which have been shown to work well for matching short strings [Kukich 1992; Schulz and Mihov 2002]:

- Edit Distance:
  The edit distance measures the dissimilarity of two strings by counting the minimum number of operations (insertion, deletion and substitutions) required to transform one string into the other [Wagner and Fischer 1974]. The similarity of two strings then can be calculated by

$$sim(s_1, s_2) = 1.0 - \frac{dist(s1, s2)}{max(|s_1|, |s_2|)}, \tag{3.1}$$

  where $dist(s_1, s_2)$ is the returned value by the edit distance function with $0 \leq dist(s_1, s_2) \leq max(|s_1|, |s_2|)$
- Longest common sub-string (LCS):
  LCS repeatedly finds and removes the longest common sub-string in the two compared strings, up to a minimum length [Friedman and Sideli 1992]. For example, if we set minimum common sub-string length to 2 characters, ”heart disease” and ”heard disease” have a longest common sub-string ” disease” (including the white-space). After it is removed, the two new strings are ”heard” and ”heart”. In the second iteration, two strings have common sub-string ”hear” and it is removed, leaving ”d” and ”t”. The total common sub-string length is 12 (also counting the white-space). The similarity of two strings can be normalized by,

$$sim(s_1, s_2) = \frac{LCS(s1, s2)}{ave(|s_1|, |s_2|)}, \tag{3.2}$$

  where the denominator can also be the minimum or maximum length of two original strings [Christen 2006].

The algorithm of spelling correction is showed in next page as Algorithm 1. The parameter choices of similarity and frequency threshold can be crucial for the correction accuracy, we will show experiments for different parameter choices in Section 4.2.

Algorithm 1 Spelling correction.

Input: **T** is the input text file general cleaned;
f_{min} is the frequency threshold used for separating input strings into two sets;
s_{min} is the similarity threshold used for determining whether an infrequent string can be corrected to its most similar frequent string;
sim_m (s,s') is the function that returns a similarity score between string s and s' using string matching methods m.

Output: A new file where text strings are cleaned.

Initialize frequent set $S_F = \{\}$, infrequent set $S_I = \{\}$
Read **T** and initialize a set S, for all unique strings s and their counts c, $(s, c) \in S$
for $(s, c) \in S$ **do**
    if $c > f_{min}$ **then** $S_F = S_F \bigcup \{s\}$
    else $S_I = S_I \bigcup \{s\}$
    end if
end for
for $s_i \in S_I$ **do**
    $s_{max} = 0, s_{replace} = ""$
    for $s_f \in S_F$ **do**
        if $sim_m(s_i, s_f) > s_{max}$ **then** $s_{max} = sim_m(s_i, s_f), s_{replace} = s_f$
        end if
    end for
    if $s_{max} > s_{min}$ **then** correct all occurring s_i with $s_{replace}$
    end if
end for
Generate *New_File* with spelling correction
return *New_File*;

3. **Inconsistency correction.**

   In both of the *COD* and *OCC* datasets, the same text description in different records may be labelled as different class labels (codes) due to the human coding inconsistency. For the inconsistency coding example refer to Section 4.1.

   Coding inconsistency brings noise into the classification. We expect one cause of death or occupation title can be coded into one standard class if it does not belong to multiple meaningfully different classes. When we observe an inconsistent coding in our dataset, we find that the different labels assigned to each text descriptions have very similar meanings. From example, records with occupation title "dairyman" are 75% coded into "Livestock Worker, Specialisation Unknown" (6-24.00) and 21% coded into "Livestock Farmer" (6-12.40). The two class labels are very similar and it is unreasonable to keep both labels for one occupation title.

   We adopt the approach to alter all labels for multiple-coded descriptions to the most frequent one. For example, for all records with "dairyman", we re-code them into the most frequent label "Livestock Worker, Specialisation Unknown"

(6-24.00). The rationale is we assume that the most frequent coding is the correct one and the others are human coding mistakes.

We only do the inconsistency correction for the *OCC* dataset. One reason is that in the *COD* dataset, the multiple-coding descriptions are rare (0.03% of all unique descriptions). Another reason is that, the structure of the *COD* testing dataset and *OCC* dataset is different (refer to Section 1.2 for detail). In the *COD* testing dataset, one record is coded into multiple labels, which means multiple coding for unique descriptions may not influence the classification accuracy. Furthermore, multiple coding may be reasonable for *COD* descriptions. Since one cause of death could belong to different medical categories, especially when the coders have additional context available. But occupation titles, on the opposite, are more likely to belong to one category.

So far, the raw input data has been cleaned from unwanted characters, misspellings and inconsistent values. The next step is to generate features from the cleaned text dataset, which will be introduced in the next section.

3.1.2 Feature Generation

The aim of feature generation is to produce a *record-feature matrix* which can be used by the classification process, where for each input record a list of features is generated and where weights of features are represented in columns. Three steps for the feature generation processes are applied, namely generate features, calculate feature weights and build *record-feature* matrix. We will show these three steps in detail in the following.

1. **Generate features.**

   This step aims to generate features from the cleaned input text. The *n-gram* (also called *q-gram*) based approach is explored in our project, which uses a sliding window approach to extract sub-strings of a certain size. According to the type of substrings, the *n-grams* approach can be at *word-level* and *character level*. According to the length of features, there are unigrams (n=1), bigrams (n=2) and trigrams (n=3) as common choices. Apart from the basic *n-grams* approach, We also applied the *skip-grams* approach [Guthrie et al. 2006]. As its name suggests, features are generated by skipping some characters. For our experiments, we only use skip grams for character level bigrams.

   We will show some examples to explain how features are generated by the *n-gram* based approach. Using the text record ”heart disease” as an example, word level unigram generates the features [”heart”, ”disease”], character level trigrams generates the features [”hea”, ”ear”, ”art”, ”rt_”,”t_d”, ”_di”, ”dis”, ”ise”, ”sea”, ”eas”, ”ase”], and 1-skip character level bigrams are [”ha”,”er”,”at”,”r_”, ”td”,”_i”,”ds”,”ie”,”sa”,”es”, ”ae”].

Type	Explanation	Example: ”gen deb and old age”
w1	Word level unigram	[”gen”, ”deb”, ”and”, ”old”, ”age”]
w2	Word level bigrams	[”gen deb”, ”deb and”, ”and old”, ”old age”]
w3	Word level trigram	[”gen deb and”, ”deb and old”, ”and old age”]
q1	Char level unigram	[”g”, ”e”, ”n”, ”d”, ”b”, ”a”, ”o”, ”l”]
q2	Char level bigrams	[”ge”, ”en”, ”n_”, ”_d”, ”de”, ”eb”, ”b_”, ”_a”, ”an”, ”nd”, ”d_”, ”_o”, ”ol”, ”ld”, ”ag”]
q3	Char level trigrams	[”gen”, ”en_”, ”n_d”, ”_de”, ”deb”, ”eb_”, ”b_a”, ”_an”, ”and”, ”nd_”, ”d_”]
s1	Skip one character	[”gn”, ”e_”, ”nd”, ”_e”, ”db”, ”ba”, ”_n”, ”ad”, ”n_”, ”do”, ”_l”, ”od”, ”l_”, ”da”, ”_g” ”ae”]
s2	Skip two characters	[”g_”, ”ed”, ”ne”, ”_b”, ”d_”, ”ea”, ”bn”, ”_d”, ”a_”, ”no”, ”dl”, ”o_”, ”la”, ”dg”, ”_e”]

Table 3.1: Individual feature types and examples. Empty feature (white-space generated by ”q1”) is deleted. Skip grams are based on character level bigrams.

There are two cases where empty features or feature lists can be generated. First, if ”q1” (character level unigram) feature type is used, white-space between tokens can be generated as one empty feature. Second, if the feature length is longer than the string length, then the feature list for the string is empty. For example, if one string only has two words and word level trigram is used, then feature list for the string is empty. For both cases, we delete the empty feature or feature list, since it will not supply any useful information for classification.

For the *character level* approach, we also *pad* strings before generating features by adding (*n*-1) special characters to the start and end of the strings. For example, when using bigrams, ”heart” can be padded to ”*heart*”, resulting in features [”*h”, ”he”, ”ea”, ”ar”, ”rt”, ”t*”]. The padding approach allows the beginning and ending characters to be included as features. Empirical results [Keskustalo et al. 2003] showed that the padding approach could increase string matching quality.

For experimental evaluation, we experiment with individual feature types and their combinations. Table 3.1 describes the individual feature choices. Combinations can be any number of any features' combination (e.g. [w1,q2]).

The intuition for testing different features includes two main aspects. One is that we want to figure out whether there are patterns between feature types and code frequencies/text lengths. For example, it is naturally to think that word level features may lead to better class performance for long text length while character level features may be better suited short text lengths. The second intuition is that we want to find out which feature types lead to better performance for our historical datasets (and more generally for short text and imbalance class classification problem).

ID	Text	Features (TF)
685	old old age	old (2), age (1)
1048	bron	bron (1)
6073	age	age (1)
6105	con	con (1)
6110	con	con (1)
11735	gen deb	gen (1), deb (1)
11476	deb birth	deb (1), birth (1)
14492	old age deb	old (1), age (1), deb (1)

Table 3.2: Sample *COD* records with term frequency. The second column shows cleaned *COD* records, the third column shows the extracted word level unigram features and corresponding term frequency (feature count in the record).

2. **Calculate feature weights.**

   Different features have different the degree of importance of expressing class characteristics. Generally, features that occur more often in record *r* but less often in all other records tend to have the higher degree of importance for record *r*. As mentioned in Section 2.4.1.2, we can use feature weights to represent the degree of importance, which mainly includes binary, term frequency (count), document frequency and TF-IDF. Refer to Section 2.4.1.2 for details. For our experiments we choose to use *TF-IDF* to weight features since it has been shown to work better than simple *term frequency* or *binary* approach [Joachims 1996].

   To demonstrate an example for the TF-IDF weight of word level unigram features, we pick a set of sample *COD* records and calculate the *term frequency* (TF) and *inverse document frequency* (IDF) respectively, based on Equations 2.1. Table 3.2 shows the cleaned text records with the word level unigram features and their corresponding *term frequency*. Table 3.3 shows the *inverse document frequency* for each feature extracted from the sample records. Then TF-IDF for each feature in each record can be calculated by multiplying TF and IDF (Equation 2.2), which is demonstrated in Table 3.4.

   From the above example, we can see the TF-IDF weight measures the importance of a feature according to two aspects:

   - How often does a feature occur in one record? The more often A feature occurs in the record, the more important the feature is for this record.
   - How often does a feature occur in all records? The less often a feature occurs in all records, the more important the feature is for all records including it.

Feature	DF	IDF
old	2	0.602
age	3	0.426
bron	1	0.903
con	2	0.602
gen	1	0.903
deb	3	0.426
birth	1	0.903

Table 3.3: Document frequency (DF) and inverse document frequency (IDF) example. The first column is each feature extracted from the sample records in Table 3.2. DF refers to document frequency (total count of one feature in all records). IDF represents inverse document frequency, which is calculated by $\log_{10} \frac{N}{DF}$, where N is the total number of records. We keep three digits for IDF weights.

ID	Text	Features (TF-IDF)
685	old age	old (0.124), age (0.426)
1048	bron	bron (0.903)
6073	age	age (0.426)
6105	con	con (0.602)
6110	con	con (0.602)
11735	gen deb	gen (0.903), deb (0.426)
11476	deb birth	deb (0.426), birth (0.903)
14492	old age deb	old (0.602), age (0.426), deb (0.426)

Table 3.4: TF-IDF example. The second column shows cleaned *COD* records, the third column shows the extracted word level unigram features and corresponding TF-IDF (calculated by multiplying TF and IDF).

3. **Build *record-feature* matrix.**

   Once we get features and their corresponding weights from an input dataset, we can build an $N \times D$ *record-feature* matrix. Each row of the *record-feature* matrix is a feature vector where each dimension corresponds to a separate feature and represented by a feature weight.

   Using the sample record shown in Table 3.2, we show the *record-feature* matrix in Table 3.5. As mentioned, the *record-feature matrix* is based on word level unigram features and TF-IDF weights.

   We can see that the matrix is sparse (many feature weights are zeros) since most of the words only occur in a small number of records. In this case, a large number of records may generate a very high dimensional and sparse matrix. In the next section, we will discuss the problems high dimensional data brings and how we use dimensionality reduction techniques to address these problems.

ID	old	age	bron	con	gen	deb	birth
685	0.602	0.426	0.0	0.0	0.0	0.0	0.0
1048	0.0	0.0	0.903	0.0	0.0	0.0	0.0
6073	0.0	0.426	0.0	0.0	0.0	0.0	0.0
6105	0.0	0.0	0.0	0.602	0.0	0.0	0.0
6110	0.0	0.0	0.0	0.602	0.0	0.0	0.0
11735	0.0	0.0	0.0	0.0	0.903	0.426	0.0
11476	0.0	0.0	0.0	0.0	0.0	0.426	0.903
14492	0.602	0.426	0.0	0.0	0.0	0.426	0.0

Table 3.5: 8×7 Record-feature matrix, where each row represents each record (identified by ID), and columns are features. The numeric number for each row and column shows the TF-IDF weight for each feature in each record.

3.1.3 Dimensionality Reduction: Principal Component Analysis

Experimental results in Section 4.2 show that introducing more feature types tend to increase the performance accuracy of classifiers for our short text data. Among all individual and combination of feature types, the combination "q2, q3, w2, w3, s1" gives the best performance for 21.6% of all codes in *COD* dataset, which is much higher than individual feature types. The reason behind this may be that more features can supply more information about the records, especially when the text description is short.

However, a combination of many feature types may result in a high dimensional *record-feature* matrix. For example, the combination "q2, q3, w2, w3, s1" generates 9,902 unique features for the *COD* dataset, which is much higher than only using the "w1" feature type (1,345 dimensions).

High dimensional data can bring two main problems for the classification task. First, we may face the *curse of dimensionality* problem, which describes "the problem caused by the exponential increase in volume associated with adding extra dimensions to Euclidean space" [Bellman 2013]. Informally, the behaviour of data structures and algorithms in low dimensions may not generalize well in higher dimensional space [Keogh and Mueen 2017]. Second, when the dimension becomes higher, the computation costs are likely to be higher. For a large amount of data and real-world applications, computation time can be crucial.

To address these problems, we use the principal component analysis (PCA) algorithm to reduce dimensions. As introduced in Section 2.4.1.3, PCA projects high dimensional data into a lower dimension space where the variance of the projected data is maximized [Wold et al. 1987]. PCA selects the largest eigenvalues of the covariance matrix of an input matrix as *components*. As the name suggests, PCA allows us to analysis only principal components, which explained most of the variance of the input data, rather than analysing the whole dataset. This allows useful information to be expressed and computed in a more compact way.

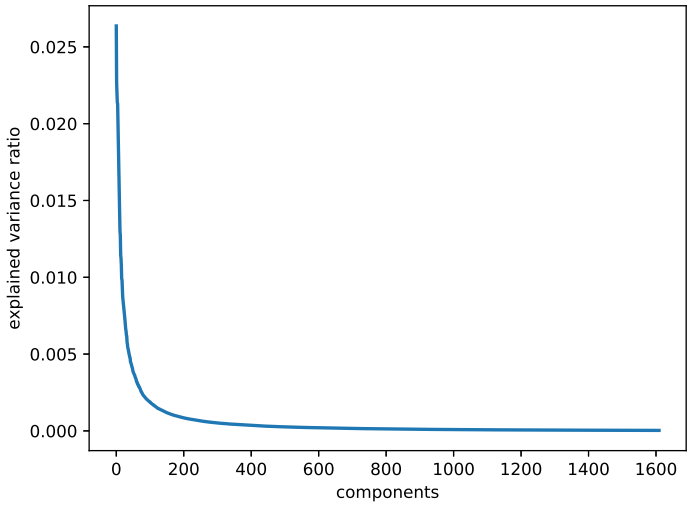


Figure 3.3: Explained variance ratio for PCA (*COD* dataset, using "q2, q3, w2, w3, s1" features and TF-IDF weights). The horizontal axis is the selected number of components, the vertical axis is the percentage of explained variance. The whole selected components (around 1,600) can explain 99% of the amount of variance for the original data. The first 200 selected components explained the most of variance.

Figure 3.3 shows the percentage of variance expressed by each of the selected components. The figure shows information about the number of components such that 99% percent of the amount of variance is explained. From the figure we can see that around 1,600 components can explain 99% percentage of the amount of variance of a 9,000-dimensional data space, among which, most of the variance is explained by first 200 components. This suggests that we can reduce a large number of dimensions without losing much variance expressed by the original data.

Using PCA to pre-processing data may have different influences on the classification performance in terms of the nature of the data and classifiers. PCA emphasizes valuable features and ignores noise, which can help a classifier to concentrate on useful information and increase the accuracy to some extent. However, when projecting data to lower dimensions, some original information can be lost or distorted, which may also have a negative influence on the classification performance. So whether the PCA increases or decreases the performance depends on the nature of the data and the noise-tolerance of the classifier.

One of the major questions for implementing PCA is how to select the number of components to get a trade-off between accuracy and computation cost. Generally, the fewer components are selected (i.e. fewer remaining dimensions), the lower the computational costs, but at the same time, the lower the accuracy. We conduct an experiment to compare the performance and computational time when different numbers of components are selected and describe how to make a best choice for the number of components, which will be demonstrated in Section 4.2.

3.2 Classification

In Section 3.1, we illustrate how we pre-process the noisy, sparse text input records into a cleaned, compact *record-feature* matrix. The next step is to construct a classification model which can learn from the *record-feature* matrix and predict unobserved records accurately. In this section, we will firstly introduce how we deal with the multi-class and multi-label classification problems. Then we will illustrate the classifier algorithms we used.

3.2.1 Multi-class and Multi-label Classification

Before we construct classifiers, it is essential to figure out what type of classification algorithm suits the problem we need to solve with. In this section, we will briefly analyse the structure of our historical datasets and then introduce the strategies we adopt to deal with the classification task.

For both COD and OCC dataset, we have more than two classes (codes), which means our tasks of classifying historical datasets are multi-class or multi-label classification problems. The type of classification tasks and corresponding strategies have been introduced in Section 2.4.2.1.

Before showing the strategies we adopted to tackle the multi-class and multi-label classification tasks, we first introduce the *record-code matrix* generated by the training and testing records. Similar as the *record-feature* matrix, we build a $N \times K$ *record-code* matrix, where N is the total number of records and K is the number of classes (codes). Each row of the matrix represents one record and each column represents one class label. Label all classes assigned for each record as 1 in the matrix, otherwise label them as 0. Figure 3.4 shows an example of building the *record-code* matrix using a sample *COD* training dataset.

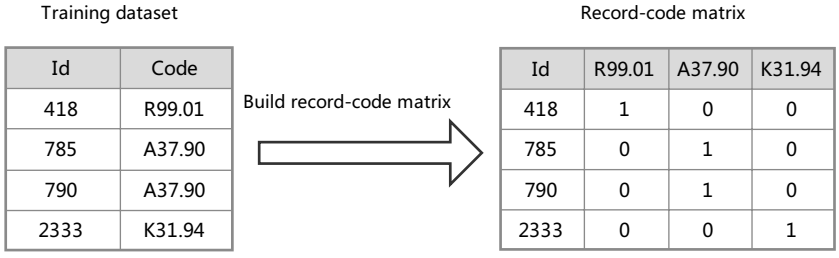


Figure 3.4: Example of record-code matrix. The left table shows a sample extracted from the *COD* training dataset (for simplicity, only ID and Code two columns are shown). The right table shows the record-code matrix generated from the left table. Binary values are used to show whether a code is labelled for one record.

- **The *COD* dataset:**

  The nature of classifying cause of death task is multi-label classification. In the training dataset, each record is assigned one label, while in the testing dataset each record is assigned more than one (up to four) labels. Furthermore, in the training dataset, the cause of death description usually contains only one cause. For example, record 785 is one cause "hc" with code "A37.90". In the testing dataset, cause descriptions usually contain multiples causes, separated by number, & or ";", where each cause corresponds to one labelled code. For example, record 2848 is "both flu; syncope" with code "J11.10" and "R55.00". So the task is to classify unobserved records with multiple causes and labels, given training records with single cause and label.

  We adopt the transformation method for classifying *COD* records, which converts the multi-label classification into multiple binary classifications. Considering our *COD* training and testing datasets have different structures, where training dataset is one-labelled and the testing dataset is multi-labelled, we trained one classifier for each code and predict all labels for each unobserved record with every classifier predicts a positive result.

  Algorithm 2 shows the process of training classifiers and evaluation, where Steps 4 to 23 train one classifier for one code and evaluate it on testing file. For one code *c*, Step 5 trains the classification model using the training *record-feature* matrix and a code vector which represents whether each training record belongs to *c* or not, Step 6 returns a predicted code vector showing whether each testing record belongs to *c* or not, Steps 8 to 20 evaluate the predicted results for *c* on each testing record. As a result, each testing record can be predicted to multiple labels by multiple classifiers. The evaluation methods used in this project will be introduced in more detail in Section 3.3.

- **The *OCC* Dataset:**

  The nature of classifying occupation titles is a multi-class classification. For the *OCC* dataset, only one dataset is given, in which each record is manually coded into one label. For example, record 4849 could be "bricklayer" with code "9-51.20". We split the dataset into training and testing dataset after doing the data pre-processing.

  We adopt one transformation strategies called "*one vs. rest*", which train one classifier for one class and classify the unobserved record with the maximum score of all classifiers. The algorithm for classifying *OCC* dataset (shown in Algorithm 3) is similar as Algorithm 2. The difference is that, instead of directly predicting label for testing records using each classifier, we predict class log-probabilities (Step 6). Then among the log-probabilities for record r predicted by all classifiers, we pick the label with the maximum log-probability as the predicted label for test records (Steps 10 to 19). So as a result, each record is predicted to one label only.

Algorithm 2 Classification for *COD*.

Input: *Training File*, containing three columns: "id", "cause", and "code".
Testing File, containing six columns: "id", "cause", "code1", "code2", "code3", and "code4".
Classifier C with *train*() and *predict*() functions.
Evaluation Function evaluate() to get precision P, recall R, F-measure F for each code.

Output: Evaluation result for each code and overall performance for *testing file*.

1: Generate *train feature matrix* (M_{trf}), *train code matrix* (M_{trc}), and *train code id Set* S_{tr} from training file.
2: Generate *test feature matrix* (M_{tef}), *test code matrix* (M_{tec}), and *test code id Set* (S_{te}) from testing file.
3: Initialise array for precision, recall and F-measure A_p = [ ], A_r = [ ], A_f = [ ]
4: **for** $cid \in S_{tr} \bigcap S_{te}$ **do**
5: C.train(M_{trf}, $M_{trc}[:, cid]$)
6: Predict_list = C.predict(M_{tef})
7: Initialise True Positive TP=0, True Negative TN=0, False Positive FP=0, False Negative FN=0.
8: **for** record id *rid* in test file **do**
9: Predict_label = Predict_list[rid]
10: True_label = $M_{tec}[rid, cid]$
11: **if** Predict_label == 1 and True_label == 1 **then**
12: $TP = TP + 1$
13: **else if** Predict_label == 1 and True_label == 0 **then**
14: $FP = FP + 1$
15: **else if** Predict_label == 0 and True_label == 1 **then**
16: $FN = FN + 1$
17: **else if** Predict_label == 0 and True_label == 0 **then**
18: $TN = TN + 1$
19: **end if**
20: **end for**
21: $P, R, F = evaluate(TP, FP, FN, TN)$
22: $A_p.append(P)$, $A_r.append(R)$, $A_f.append(F)$
23: **end for**
24: Calculate mean μ and standard deviation σ for A_p, A_r, A_f
25: **return** A_p, A_r, A_f, μ, σ

Algorithm 3 Classification for *OCC*.

Input: *Training File*, *Testing File* split from one input file which contains "id", "occupation title", "code" (three columns).
Classifier C with *train*() and *predict*() functions.
Evaluation Function evaluate() to get precision P, recall R, F-measure F for each code.

Output: Evaluation for each code and overall results for *testing file*.

Generate *train feature matrix* (M_{trf}), *train code matrix* (M_{trc}), and *train code id Set* (S_{tr}) from training file.
Generate *test feature matrix* (M_{tef}), *test code matrix* (M_{tec}), and *test code id Set* (S_{te}) from testing file.
Initialise array for precision, recall and F-measure A_p = [ ], A_r = [ ], A_f = [ ]; hash table H_p for predicted class log probability.
for $cid \in S_{tr} \bigcap S_{te}$ **do**
    C.train(M_{trf}, $M_{trc}[:, cid]$)
    Predict_prob_list = C.predict(M_{tef})
    H_p.insert(cid, Predict_prob_list)
end for
Initialise array for predict label $A_{predict} = []$.
for record id *rid* in test file **do**
    Max_{pro} = 0, $Max_{cid} = 0$
    for cid in $H_p.keys()$ **do**
        if $H_p.get(cid)[rid] > Max_{pro}$ **then**
            $Max_{pro} = H_p.get(cid)[rid]$
            $Max_{cid} = cid$
        end if
    end for
    $A_{predict}$.append(cid)
end for
for $cid \in S_{tr} \bigcap S_{te}$ **do**
    Initialise True Positive TP=0, True Negative TN=0, False Positive FP=0, False Negative FN=0.
    for record id *rid* in test file **do**
        Predict_label = $A_{predict}[rid]$
        True_label = $M_{tec}[rid, cid]$
        if Predict_label == 1 and True_label == 1 **then** $TP = TP + 1$
        else if Predict_label == 1 and True_label == 0 **then** $FP = FP + 1$
        else if Predict_label == 0 and True_label == 1 **then** $FN = FN + 1$
        else if Predict_label == 0 and True_label == 0 **then** $TN = TN + 1$
        end if
    end for
    $P, R, F = evaluate(TP, FP, FN, TN)$
    $A_p.append(P)$, $A_r.append(R)$, $A_f.append(F)$
end for
Calculate mean μ and standard deviation σ for A_p, A_r, A_f
return $A_p, A_r, A_f, \mu, \sigma$

3.2.2 Classifier construction

One of the most important part of Algorithms 2 and 3 is how to build the classifier C. In this section, we will introduce the choices of classifiers in our project and corresponding parameter choices. The detailed explanation for each classifier is mentioned in Section 2.4.2.2. We implement the classification algorithms using Scikit-learn library in Python [Pedregosa et al. 2011].

- **Naive Bayes**: we use multinomial naive Bayes, which is one of the classic Bayes variants used in text classification. The distribution is parametrized by vector $\theta_c = (\theta_{c1}, ..., \theta_{cD})$ for each class c, where D is the number of features and θ_{ci} is the probability $P(x_i|c)$ of feature i appearing in a record belonging to class c [Pedregosa et al. 2011].

  A Laplace smoothing is used for calculating θ_{ci},

  $$\hat{\theta_{ci}} = \frac{N_{ci} + \alpha}{N_c + \alpha n}, \tag{3.3}$$

  where N_{ci} is the number of times feature i occurs in a record assigned to class c and N_c is the total count of N_{ci} for all features for c.

  α is the smoothing prior, adding the features not presented in training records to both numerator and denominator. Smoothing prevents zero probabilities in further computations.

- **Logistic Regression**: Regularized logistic regression is implemented, which can handle the sparse input. The inverse of regularization strength is specified by parameter C (positive float), smaller C specifies stronger regularization.

- **Support Vector Machines:** SVMs is implemented with different kernel methods, the choices of which include [Elisseeff and Weston 2002]:

  - linear: $\langle x, x' \rangle$
  - polynomial: $(\gamma \langle x, x' \rangle + r)^d$
  - RBF (radial basis function): $exp(-\gamma \|x - x'\|^2)$
  - sigmoid:$(tanh(\gamma \langle x, x' \rangle + r))$

  We implement C-support vector classification based on libsvm [Chang and Lin 2011]. Penalty parameter C indicates the error term.

- **Decision Tree:** we use information gain to measure the quality for splitting branches, which is defined by Equation 3.6.

  Let p_i be the probability that an arbitrary record belonging to class c_i of K classes. Then the expected information (called *entropy*) needed to classify a record in dataset D is

  $$Info = -\sum_{i=1}^{K} p_i \log_2(p_i), \tag{3.4}$$

If we use feature F to split the dataset D into v partitions, the information needed to classify the whole dataset is reduced to

$$Info_F = -\sum_{j=1}^{v} \times \frac{D_j}{D} Info(D_j), \tag{3.5}$$

The information gained by branching on feature F is

$$Gain(F) = Info(D) - Info_F(D). \tag{3.6}$$

3.3 Evaluation

Once we build a classification model which can predict labels, we need to evaluate the performance of the model for unobserved records. The evaluation methods and metrics used in this project are introduced in this section.

For the *COD* dataset, the training file and testing file are given with different structures. Thus, we directly train classifier models on training dataset and test on testing dataset.

For the *OCC* dataset, one gold standard file is given. We perform the data preprocessing on the whole dataset and split the dataset into the training dataset (80%) and the testing dataset (20%). Considering the imbalance classes in our dataset, we use the *stratified shuffle split*, where relative class frequencies are approximately preserved in training and testing datasets. Before splitting, we delete the codes that only occur once in the dataset, since after splitting there cannot be at least one record for the code in both training and testing dataset.

It could be better if we can use *cross-validation* [Kohavi et al. 1995] to make full use of the dataset and get the average performance for multiple models. However, it would be more time consuming for experimental tests considering the size of data. Due to the time limit, we use *hold-out* evaluation method for this project. For future work, *cross-validation* model can be tested, as discussed in Section 5.2

Although we are dealing with the multi-class and multi-label classification problem, we transformed the problem into multiple binary classification problems (shown in Algorithms 2 and 3). Therefore, we can apply the evaluation metrics for the binary case as well.

To demonstrate the meaning of the evaluation metrics for our historical dataset, we use a contingency table for binary classification, which is shown in Table 2.1 (on Page 22), where columns are ground truth and rows are predicted labels. For each trained classifier for code c, we treat the records belonging to c as positive records and all other records as negative records.

Informally, the metrics are defined for historical dataset as follows:

- True Positive (TP): the number of records which actually belong to the code c and are classified correctly into c.
- False Positive (FP): the number of records which actually not belong to the code c and are classified wrongly into c.
- False Negative (FN): the number of records which actually belong to the code c and are classified wrongly not to belong to c.
- True Negative (TN): the number of records which actually not belong to the code c and are classified correctly not to belong to c.

  In our project, we use precision, recall and F-measure to evaluate each code and overall performance. The definition of these concepts is given in Section 2.4.3. To explain why we choose these evaluation metrics, we first show what these metrics mean for historical dataset:

  - **Precision:** the percentage of correct classification among historical records that are classified as belonging to code c.
  - **Recall:** the percentage of correct classification among all historical records actually belong to code c.
  - **F-measure:** a trade-off version of precision and recall, showing both how precise the classifier is (how many instances it classifies correctly) and how robust it is (it doesn't miss a significant number of instances). We set parameter $\beta = 0.5$ in our project, which is the harmonic mean of precision and recall.

  These measures evaluate the model for the percentage of correct classification in terms of the ground truth and predicted labels, rather than all the records. For imbalanced classes, evaluating on all records (i.e. accuracy) can always achieve a good result if we just simply classify all records into the class with majority records.

  Precision and recall can better reflect the model with imbalance classes. For example, if we totally have 100 unobserved records, where only 3 records are assigned as belonging to code c and others not belonging to c. Then if we simply classify all records as not belonging to c, then the accuracy is 0.97, which is extremely high, although the model is not working at all. However, it will lead to a bad precision (0) since no record is predicted to code c and a bad recall (0) since records truly belonging to c are all wrongly classified.

  F-measure can give a balanced version for the observed model. By themselves, a bad precision or recall cannot imply a bad model. Precision and recall supply different versions to observe the performance of the model. The choice between precision and recall is based on what characteristic (precise or robust) of the

ID	Label	Predict
1	c	c
2	c	c
3	c	c
4	not c	c
5	not c	c
6	not c	c
7	not c	not c
...	...	...
100	not c	not c

Table 3.6: Example for predicted labels. We have 100 records with imbalanced classes, where only first three records belong to c. We predict first six records belonging to c and all other records not belonging to c.

model the user emphasizes. For example, in Table 3.6, the recall is 1 since all records belonging to c are correctly predicted; precision is only 0.5, since only half of records predicted to c are correct. However, either precision and recall doesn't evaluate the model well. The recall indicates the model is perfect and the precision indicates the model only randomly guessing. In this case, we can use F-measure to evaluate the model, which supplies a trade-off version of the both metrics and can give a more reasonable score (0.67).

As shown in Algorithms 2 and 3, we calculate precision, recall and F-measure for each code respectively and then get the overall performance by taking mean of those measure values for all codes.

3.4 Chapter Summary

In this section, we summarised the historical coding classification processes and showed the algorithm pipeline in Figure 3.5, where each step is summarised as follows:

1. **Load dataset**: load the input dataset and record the information for historical text descriptions and codes.

2. **Clean text descriptions**: data cleaning includes three main steps:

   - General cleaning: includes removing unwanted characters (i.e. punctuation, extra white-spaces and stop-words) and stemming.
   - Spelling correction: a similarity-based approach is applied to correct potential misspellings.
   - Inconsistency correction: the same text description may have multiple labels. We replace inconsistency labels to the most frequent label assigned to the same description.

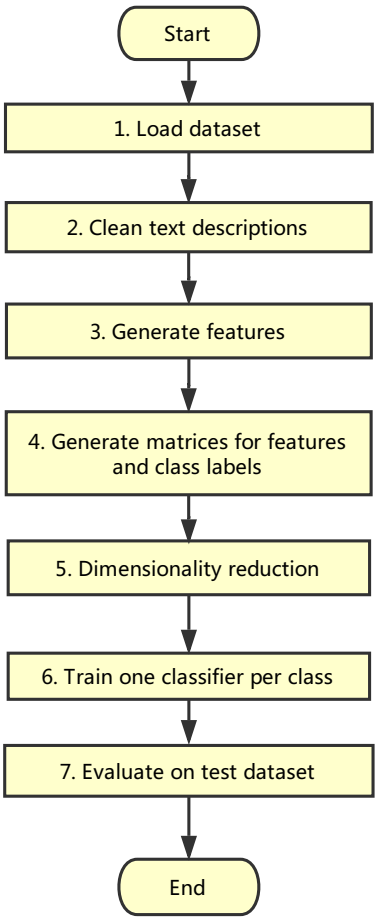


Figure 3.5: Historical coding classification algorithm pipeline

3. **Generate features:** we use individual and combination n-grams as feature types and TF-IDF as feature weight. This step transforms input text descriptions into feature vectors.

4. **Generate metrics:** we build $N \times D$ *record-feature* matrix based on the generated feature vectors and $N \times K$ *record-code* matrix, where N is the number of records, D is the number of feature dimensions and K is the number of classes.

5. **Dimensionality reduction:** we use principal component analysis (PCA) to reduce feature dimensions, where the number of components is carefully selected to balance the classification performance and computational cost.

6. **Train one classifier per class:** we use the transformation strategies to address the multi-class and multi-label classification problems, converting them into multiple binary classification tasks. The classification methods for the *COD* and *OCC* datasets are slightly different due to the different structure of each dataset. The classification algorithms include naive Bayes, logistic regression, support vector machine and decision tree.

7. **Evaluate on test dataset:** we use the *hold-out* method, training the model on the training dataset and testing it on the testing dataset. Evaluation metrics are precision, recall and F-measure, which are calculated for each code respectively. The overall results are calculated by taking the mean of scores of individual codes.

Also note that for the *COD* dataset, the training and testing datasets are separately given, thus Steps 1 to 5 are processed separately on the training and testing datasets. For the *OCC* dataset, we split the given dataset after dimensionality reduction (Step 5). In the next chapter, we will show the characteristics of the datasets and the evaluations results for both of the datasets.

Chapter 4

Evaluation

In this chapter, we will introduce the characteristics of the two datasets in Section 4.1, namely the cause of death (*COD*) and occupation (*OCC*) datasets. Then in Section 4.2, we will show the experiments we designed to evaluate our techniques as well as the corresponding results.

4.1 Dataset Description

In this section, we will firstly introduce the standard coding system we adopted in our work. Then we will demonstrate the characteristics of our datasets.

- **Standard Coding System:**

  As described in Section 2.2, the standard coding system used in the *COD* and *OCC* datasets are International Classification of Disease (ICD-10) and Historical International Standard Classification of Occupation (HISCO) respectively. We will show some examples in the datasets and methods we pre-processed them.

  The COD dataset

  The *COD* dataset is based on the International Classification of Disease (ICD-10) classification scheme [WHO 1990]. Table 4.1 shows a sample of the ICD-10 classification scheme. The ICD-10 classification scheme is hierarchical. We call the three-character categories as the *main code* and the four-character categories as the *full code*. For example, A00 is a *main code* and A00.10 is a *full code*.

Code Structure	Standard Code	Standard Disease Name
Main code	A00	Cholera
Full code	A00.00	Cholera due to Vibrio cholerae 01, biovar cholerae (Classical cholera)
	A00.10	Cholera due to Vibrio cholerae 01, biovar eltor (Cholera eltor)
	A00.90	Cholera, unspecified

Table 4.1: ICD-10 Classification scheme examples

In the *COD* dataset, the given codes are all *full code*, from where we extract the *main code*. Some extraction examples are shown in Figure 4.1. Records with different *full codes* can have the same *main code* after extraction. For example, A00.90 and A00.91 both becomes the main code A00. As a result, each code of the *main code* will have more training records compared with *full code*.

The number of records with *main* and *full codes* in the *COD* dataset is shown in Table 4.2. Some codes only occur in the training or testing dataset. Codes only occurring in the training dataset are trained without valid evaluation, while codes only occurring in the testing dataset cannot be trained due to lack of solid training. Therefore, we only consider those codes that occur in both the training and testing datasets.

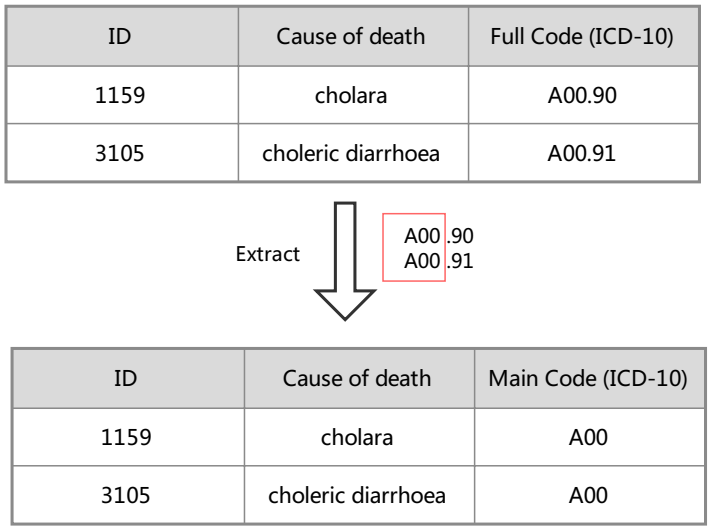


Figure 4.1: Examples of the main code extraction process

	Main Code	**Full Code**
Codes in training dataset	290	484
Codes in testing dataset	311	533
Codes only in training dataset	55	128
Codes only in testing dataset	76	177
Codes in both training and testing dataset	235	356

Table 4.2: The *COD* dataset unique code counts

The OCC dataset

The *OCC* dataset is coded to the Historical International Standard Classification of Occupations (HISCO) [Van Leeuwen et al. 2002]. Table 4.3 shows the example HISCO hierarchical scheme. The *major code* is a top-level occupation hierarchy with single number, while the *minor* and *unit codes* are more detailed hierarchies with two or more characters. For example, 0 is a *major code*, 0-1 is a *minor code* and 0-11.10 is a *unit code*.

Code Structure	**Standard Code**	**Standard Occupation titles**
Major Groups	0	Professional, Technical and Related Workers
Minor Groups	0-1	Physical Scientists and Related Technicians
Unit Groups	0-11	Chemists
	0-11.20	Organic Chemist
	0-11.90	Other Chemists

Table 4.3: HISCO scheme examples

In the *OCC* dataset, the given codes include all three types. Similar as shown in Figure 4.1, we also extract higher hierarchical codes from lower hierarchies. Table 4.4 shows the number of different types of codes in the *OCC* dataset.

	Major Code	**Minor Code**	**Unit Code**
Input Dataset	10	76	332

Table 4.4: The *OCC* dataset unique code counts

- **Dataset Characteristic:**

  After demonstrating the basic classification schemes, we now describe some of the characteristics of our datasets. We use the *COD* training dataset and the *OCC* dataset as examples.

  - **Skewed distribution of classes:**

    We define the frequency of code c as the number of records with c in a dataset. A common problem with our historical *COD* and *OCC* datasets is the unbalanced distribution of class labels. In the *COD* training dataset the code frequency ranges from 1 to 2,834; in the *OCC* whole dataset the code frequency ranges from 2 to 7,270.

    Table 4.5 shows the five least and most frequent codes (both *main code* and *full code*) in the *COD* training dataset. 347 codes have no more than 10 training records, and 92 codes only have 1 training record.

	Code	Count
Least Frequent	G20	1
	P95	1
	K14.00	1
	C34.90	1
	Y19.04	1
Most Frequent	I51	952
	J20.91	985
	J20	1245
	A16.96	1255
	A16	2834

Table 4.5: Five least and most frequent codes in the *COD* Dataset

Figures 4.2 and 4.3 show the code frequency distribution for the *COD* and *OCC* datasets. The left figures show small code frequency between 0 to 100, and the right figures show the code frequency larger than 100. From the figures we can see that both the datasets have many low frequency codes and distribution of code frequencies are quite skewed.

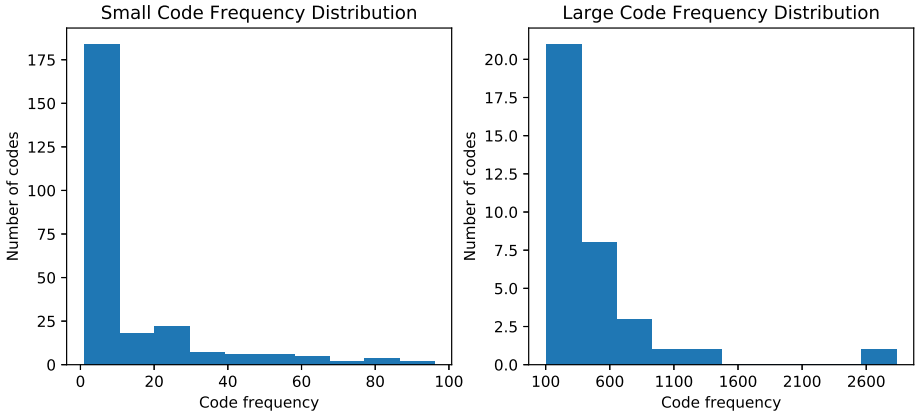


Figure 4.2: Code frequency distribution in *COD* training dataset. The horizontal axis is code frequency. The vertical axis is the number of codes for a certain code frequency.

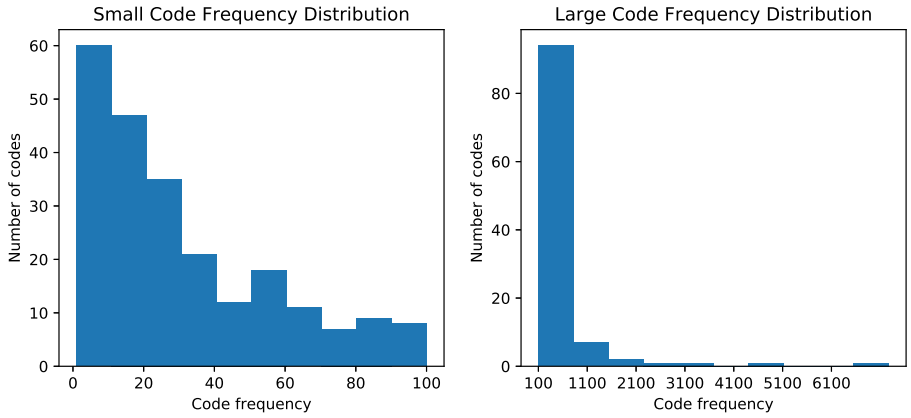


Figure 4.3: Code frequency distribution in *OCC* training dataset. The horizontal axis is code frequency. The vertical axis is the number of codes for a certain code frequency.

– **Non-standard descriptions with different length:**

Medical personnel writes the cause of death descriptions with abbreviations and narrative descriptions. There are more than one thousand records with abbreviations, such as ”hc”, ”sf”, which stands for ”whooping cough” and ”Scarlet fever”. This kind of descriptions can be very short and difficult to interpret to non-experts. There are also long narrative descriptions providing detailed explanations of death causes, for example, one record is ”died suddenly probably from heart disease but can”t certify the cause as life was exinct before being seen by any medical man”.

The lack of consistency between descriptions and abbreviations makes the data sparse and hard to be correctly classified. Figure 4.4 and 4.5 show the *COD* and *OCC* description length distribution respectively, measured as the number of characters (left plots) and words (right plots) in a textual. For example, the length of string ”cancer of liver” is 15 and it has 3 words.

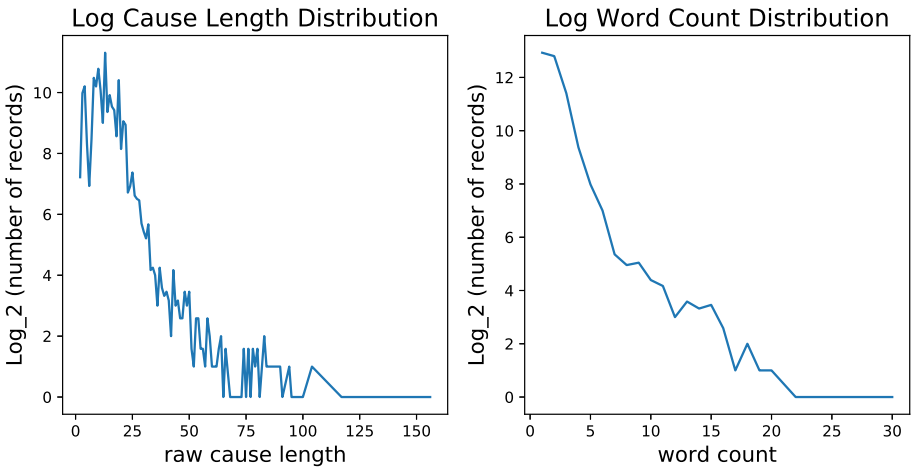


Figure 4.4: Text length distribution in *COD* training dataset. The horizontal axis shows the cause character/word length. The vertical axis shows the log of number of records with a certain text length.

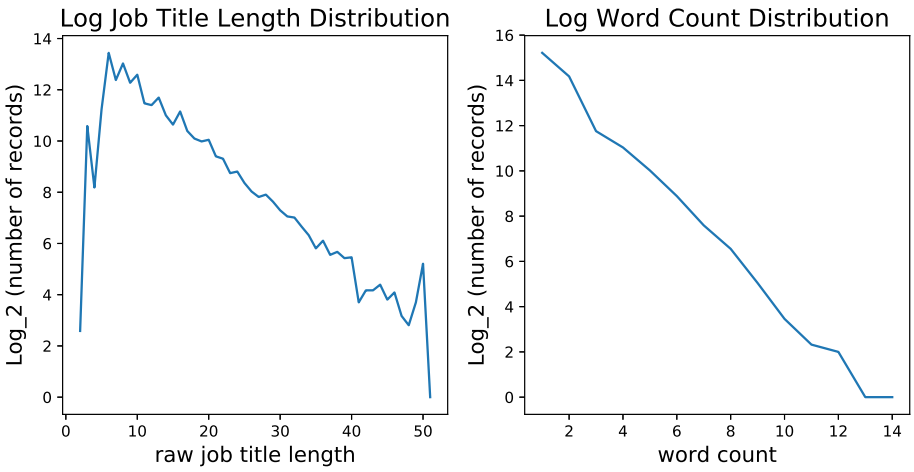


Figure 4.5: Text length distribution in *OCC* training dataset. The horizontal axis shows character/word length of the occupation titles. The vertical axis shows log of the number of records with a certain text length.

– **Misspellings:**

Both the *COD* and *OCC* descriptions have many misspellings. The datasets are originally collected in handwritten format, as shown in Figures 1.1 and 1.2. The scanned copies of handwritings are not clear enough and not easy to read, which leads to some typographical errors in the historical datasets. Table 4.6 and 4.7 show examples of records containing misspellings.

Id	Cause of death	Code (IDC-10)	Misspelling Correction
2022	caner of thigh	C76.50	cancer
6637	injuriy of knee & chest	Y34.07	injury
1230	hoopen cough (spelling?)	A37.90	whooping, swelling

Table 4.6: Misspelling examples in the *COD* dataset

Id	occupation Title	Code (HISCO)	Misspelling Correction
1465	sargeant major	5-83.00	sergeant
1796	electrical enginneer	8-55.00	engineer
2492	sales assistent, drapery	4-51.00	assistant

Table 4.7: Misspelling examples in the *OCC* dataset

– **Human coding inconsistency:**

Records of text descriptions are manually coded into standard classification coding schemes. The same text description can be coded into different categories. From the 2,139 unique cause of death descriptions, 8 have been coded to more than one ICD-10 code; and from 9,771 unique occupation titles, 128 have been assigned to more than one HISCO code.

We give examples for multiple-coded descriptions below, where labels with a frequency smaller than 5% with regard to all label for the unique description is shown as "other". For example, in the *COD* training dataset, death description "bronchitis" is coded into:

* Acute bronchitis, unspecified (J20.91): 82.6%
* Simple chronic bronchitis (J40.01): 15.3%
* Other : 2.1%

In the *OCC* dataset, the occupation title "dairyman" is coded into:

* Livestock Worker, Specialisation Unknown (6-24.00): 75.0%
* Livestock Farmer (6-12.40): 20.8%
* Other : 4.2%

Taken together, these issues present considerable challenges for classification. Skewed class distribution and different length of descriptions leads to sparse data. The lack of training records makes it difficult to apply certain machine

learning approaches. Noisy descriptions including abbreviations, narrative sentences and misspellings will likely influence the accuracy in a negative way. In the next section, we will introduce the experiments designed to address these problems.

4.2 Experiments and Results

In this section, the model parameter details for experiments is explained first. Then we design experiments for the *COD* and *OCC* datasets to evaluate automatic classification algorithms, where the main experiments (classifier and features testing, spelling correction and PCA experiments) are performed on the *COD* dataset, and the best parameter choices are applied for the *OCC* dataset to validate the generalisation of our methods.

4.2.1 Experimental Setup

The experimental design is based on the algorithm pipeline shown previously in Figure 3.5. We introduce the options for parameters in the pipeline as follows:

- Input file: we used two historical datasets collected from Scotland from two domains, namely cause of death (*COD*) and occupation (*OCC*). See Section 1.2 for examples of the datasets.
  - *COD* dataset: the training and testing datasets are given separately. Each record is coded into one class in the training dataset and coded into multiple classes in the testing dataset.
  - *OCC* dataset: only one file is given and we split it into one training dataset and one testing dataset after dimensionality reduction. Each record is assigned to one label.
- Clean flag: To indicate whether to clean the text descriptions. The cleaning steps include remove unwanted characters, stemming, spelling correction and inconsistency correction (see Section 3.1.1 for details). Spelling correction algorithm is separately tested. For spelling correction, we have different choices for the following parameters:
  - Frequency threshold: a positive integer as threshold to distinguish correct words and potential misspellings.
  - Similarity threshold: a fraction as threshold to decide whether a potential misspelling will be replaced as its most similar word.
  - String matching techniques: measures to calculate the similarity between words. We use either *edit distance* or *longest common sub-string (LCS)*.

- Feature list: The list of feature types to use, including individual feature types and combinations, where individual feature types include:
  - w1: single words
  - w2: double words
  - w3: triple words
  - q1: character unigrams
  - q2: character bigrams
  - q3: character trigrams
  - s1: skip unigrams
  - s2: skip bigrams

  The possible combinations is the combination of any individual feature type (e.g. "w1,q2"). See Section 3.1.2 for examples.

- For dimensionality reduction, we use principal components analysis (PCA) (See Section 3.1.3 for details):
  - PCA Flag: to indicate whether to use PCA.
  - n components: the percentage of the amount of variance to be retrieved.

- Classifier types and corresponding parameters (See Section 3.2.2 for details):
  - Naive Bayes (NB): using multinomial NB, α (additive smoothing parameter: float)
  - Logistic regression (LR): C (inverse of regularization strength: positive float)
  - Decision Tree (DT): criterion (one of: gini, entropy)
  - Support vector machine (SVM): kernel (one of: linear, poly, rbf, sigmoid).

  The abbreviation showed in brackets are used to throughout this chapter to represent each classifier.

- According to code hierarchy, the code types can be:
  - For death code, use one or more of the *main code* or *full code*.
  - For occupation code, use one or more of the *major code* or *minor code* or *unit code*.

We summarise the above parameter choices in Table 4.8, where six main parameters are outlined. The spelling correction and PCA parameters are not included in the table.

Para	Input file(s)	Clean flag	Feature list	PCA flag	Classifier	Code Type
Choices	*COD* *OCC*	*True* *False*	w1,w2,w3, q1,q2,q3, s1,s2 combs	*True* *False*	NB (0.1) LR (1.0) SVM (linear, 1.0) DT (entropy)[a]	*main/ full* *major/ minor/ unit*

[a]NB: naive Bayes; LR: logistic regression; SVM: support vector machine; DT: decision tree

Table 4.8: Experimental parameters setup.

4.2.2 *COD* : Experimental Design and Results

In this section, we discuss the three main experiments we designed and the corresponding results on the *COD* dataset, which include:

- Classifier and Feature Experiment: all combinations of feature types and classifiers are evaluated in terms of individual classifier performance and overall performance. The patterns between classification performance and code frequency/text length are explored.
- Spelling Correction Experiment: parameters for the spelling correction algorithm demonstrated in Algorithm 1 (on page 30) are explored, namely *frequency threshold, similarity threshold and similarity method*. To observe how spelling correction influences the results, controlled experiments are designed, where we tune one parameter to observe changing trend and keep other parameters unchanged.
- PCA Experiment: this experiment is designed to explore how to achieve a balance between classification performance and computational cost by remaining different number of dimensions in the PCA algorithm. A variety of results with PCA on different classifiers are shown in terms of both classification performance and computational cost.

For each experiment, we will first show the design ideas and parameter settings. Followed this, detailed results and analysis will be shown. Finally, we will discuss the results in terms of the significance and limitations.

1. **Classifier and Feature Experiment**

   We design this experiment to explore how feature types and classifiers influence the classification performance. We iterate the training and evaluation process for different combinations of feature types and classifiers, and keep other parameters unchanged.

   In total, 1,024 combinations of classification and feature types have been explored in this experiment. Each of the individual feature types can be included in the feature type list or not, so we have $2^8 = 256$ feature types in the feature

Para	Input file(s)	Clean flag	Feature list	Feature weight	PCA flag	Classifier	Code Type
Setting	*COD*	True	all type of combinations[a]	TF-IDF	False	NB (0.1) LR (1.0) SVM (linear, 1.0) DT (entropy)[b]	*main full*

[a] combinations of w1, w2, w3, q1, q2, q3, s1, s2. Totally $2^8 = 256$ combinations for feature types.
[b] NB: naive Bayes; LR: logistic regression; SVM: support vector machine; DT: decision tree

Table 4.9: Setting for classifier and feature experiment

type list. We perform the experiment using four classifiers, namely naive Bayes (NB), logistic regression (LR), support vector machines (SVMs) and decision tree (DT). Therefore, there are 256 $\times$ 4 = 1,024 combinations in total. Table 4.9 shows the parameter settings for this experiment.

For evaluation, we calculate the precision, recall and F-measure separately for each code, and taking the corresponding means of these evaluation scores, as shown in Algorithm 2 and 3 (on Page 39). In this experiment, we analyse the overall performance and individual classifier performance for each code respectively.

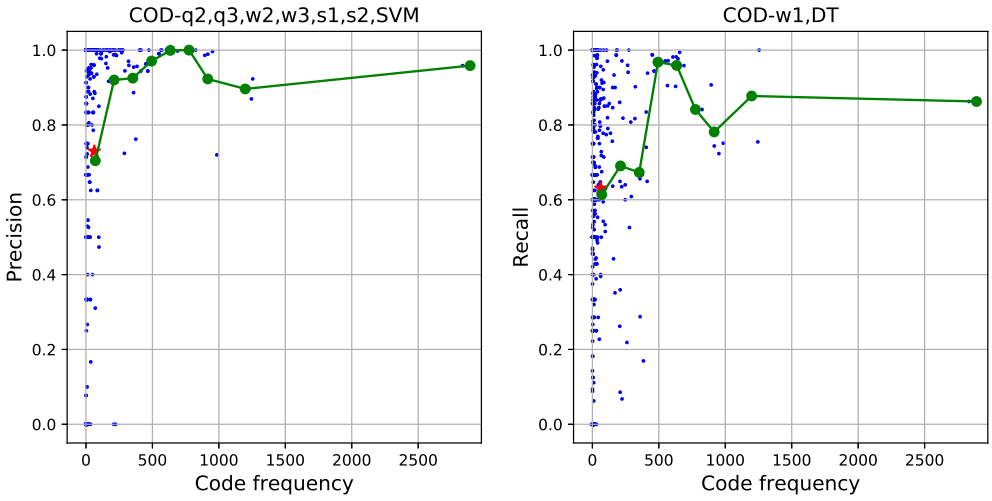
Overall Performance

For overall performance (precision, recall and F-measure means), we pose the following questions:

- Which combination(s) give the best overall performance?
- Which classifier(s) tend to give a better overall comparative performance?
- Which kind of feature types performance better? Long combinations of features or individual ones?

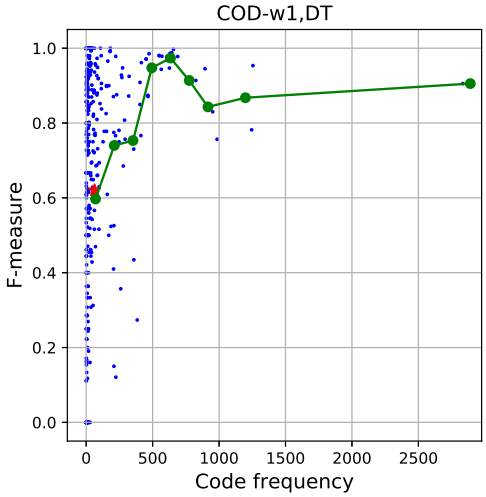
First, we rank the overall performance score for all combinations. Figure 4.6 shows the combinations which achieve the best performance in terms of each evaluation metric. The figure shows the performance with different code frequencies, where we define the code frequency for code c as the number of training records for c.

As shown in Figure 4.6, the best precision is achieved by the combination "q2, q3, w2, w3, s1, s2" and SVM. The best recall and F-measure score is returned by "w1" and DT. The overall performance scores are not very high because they are greatly influenced by the codes with low code frequency, where there are many code with a score below 0.5.



(a) Best Precision: 0.75

(b) Best Recall: 0.63



(c) Best F-measure: 0.63

Figure 4.6: Overall best performance (*COD*). The horizontal axis shows the code frequency, which is the number of records for a code. The vertical axis shows the precision/recall/F-measure score. We scatter performance scores of each code as blue points and use a binning approach to show the performance trend (green line with marker). The red star represents the average value of both code frequency and performance. The higher the evaluation score (closer to 1), the better the performance.

We use the F-measure score to do further evaluation of model preference. For the combinations that achieved top 100 F-measure scores (≥ 0.55), 96% of them include SVM classifier, 4% include DT classifier and there were none for NB or LR. This result is interesting since for the *COD* dataset, SVM performs stably better than other classifiers. Another interesting fact is that SVM performances better with long feature combination lists (more than 3 individual feature types" combination, like "q2, q3, w2, w3, s1") than with short ones. And for DT, the four best combinations are "w1", "w1, w2", "w1, w3" and "w1, w2, w3", amongst which "w1" achieves the best F-measure score compared with all combinations.

The results show that among the four classifiers, SVM and DT on average perform better in terms of the F-measure score. DT is feature-sensitive, only performing well when it combines with word level feature types. And the *bag of words* (word level unigram) is the best combination choice for DT. SVM is less feature-sensitive compared with DT, performing well with a variety of features and it generally achieving better results when combining more features rather than less features.

The different preferences for features of SVM and DT can be explained by how the classifiers work. SVM depends on the support vectors that define the maximum margin rather than all of the input features. Thus SVM is less feature-sensitive. More input features may take SVM more time to find the support vectors, but they also help SVM to find the more accurate margin to separate different classes. So SVM with long feature combinations can achieve better performance compared with individual feature types. DT generates a tree-like structure to predict labels, splitting branches on each feature. Word level features tend to give the DT algorithm a clearer way to generate a tree, since each feature itself has semantic meaning and it is sensible to be treated as an individual splitting point. The character features are less independent and may lead to an over-complex and over-fitting tree.

Performance for each classifier

From Figure 4.6, we can see that the performance differs in terms of different code frequencies. The overall performance is greatly influenced by low frequency codes and may not accurately reflect the true performance of the classifier. Thus it is worthwhile to analyse the individual code performance and explore patterns behind the performance. More precisely, we can pose this in terms of the following questions:

- Which combinations of classifiers or feature lists appear most frequently as the best combinations for each code?
- How is performance influenced by the code frequency? Is there any pattern between the performance of different classifiers/features and the code frequency?
- How is performance influenced by the length of text descriptions (number of words)?

To answer the above questions, we design three sub-experiments:

(1) Frequent Best Combinations

Similar to the overall performance analysis, we first find the combinations that achieve the best F-measure for each code (we call them *best combinations*). For example, for code "A17.01", the maximum F-measure is 0.98, which is returned by "w2, w3, DT" and "w2, DT" combinations. Note that there are 21 codes with maximum F-measure (max_f) equalling to 0. Table 4.10 shows the relationship between the code frequency and the number of codes with 0 maximum F-measure (# max_f). We can see most of the zero scores come from the code with only one training record (71.4 %) and all zero scores are from the codes with a low frequency (≤ 6), which shows the challenge brought about by a lack of training records.

From the best combinations for each code, we extract the features and classifiers, and calculate their frequency to achieve the best F-measure. For example, from a combination "q2, q3, w2, w3, s1, SVM", the feature combination "q2, q3, w2, w3, s1" and the classifier "SVM" are extracted. The top 5 best combinations, features and classifiers along with their frequency (count divided by total code number) are shown in Table 4.11. "q2, q3, w2, w3, s1" and SVM are the most frequent feature list and classifier of being included in the best combinations.

Code Frequency	#max_f = 0
1	15
2	3
4	1
6	2
Total	21

Table 4.10: Poor performance codes. The first column shows the code frequency, the second column shows the number of codes with maximum F-measure equalling 0.

Rank	Combination		Feature		Classifier	
	name	freq	name	freq	name	freq
1	q2_q3_w2_w3_s1, SVM	0.2166	q2_q3_w2_w3_s1	0.2166	SVM	0.5415
2	q2_q3_w3_s2, DT	0.0457	q2_q3_w3_s2	0.0457	DT	0.3503
3	q1_w1_w3_s1, SVM	0.0423	q1_w1_w3_s1	0.0423	NB	0.0609
4	w3, DT	0.0355	w3	0.0355	LR	0.0118
5	w2_w3, DT	0.0338	w2_w3	0.0338		

Table 4.11: Top 5 rank best combinations. The "name" column shows the name of combinations/features/classifiers, the "freq" column shows the frequency for each item being included in the best combinations.

(2) Code Frequency

To figure out how performance is influenced by the code frequency, we first sort all the codes (591 codes for both *main code* and *full code*) in terms of the code frequency and then bin them into 5 code frequency groups (CFG), where each group has approximately the same number of codes (around 118). The bound of CFGs can be overlapped since codes with the same code frequency can be binned into different groups. Then each CFG with its corresponding code frequency interval is:

- CFG 1: [1,2]
- CFG 2: [2,4]
- CFG 3: [4,12]
- CFG 4: [12,42]
- CFG 5: [42,2834]

For each CFG, we calculate the frequency of classifiers/features present within in the best combinations, where the frequency for classifier/feature m in CFG n is calculated as a relative frequency count:

$$F_{m,n} = \frac{C_m}{N_n}, \tag{4.1}$$

where C_m is the count of m extracted from all the best combinations which achieve maximum F-measure score for each code, N_n is the total number of codes in CGF n.

Then for each CGF, we rank the best combinations, features and classifiers as we did in Table 4.11. We show the rank results (Top 5) in Appendix. To clearly compare the frequency of classifiers/features between CGFs, we draw lines plots to show the frequency of each classifier/individual feature in terms of different CGFs, which is shown in Figures 4.7 and 4.8.

Classifier Frequency

Figure 4.7 shows the frequency of classifiers in terms of code frequency. It is clear that as the code frequency increases, the percentage of best performance from SVM grows and the percentage of DT decreases. NB and LR are less competitive compared with SVM and DT; their frequencies are on average below 0.1 for all CFGs.

When the code frequency is low, many features tend to appear with low weight since they only appear a few times. SVM may find an inaccurate maximum margin hyperplane due to the limited number of features. More training records can provide more features and the valuable features tend to have higher weights, which helps SVM to find support vectors more accurately. So when the code

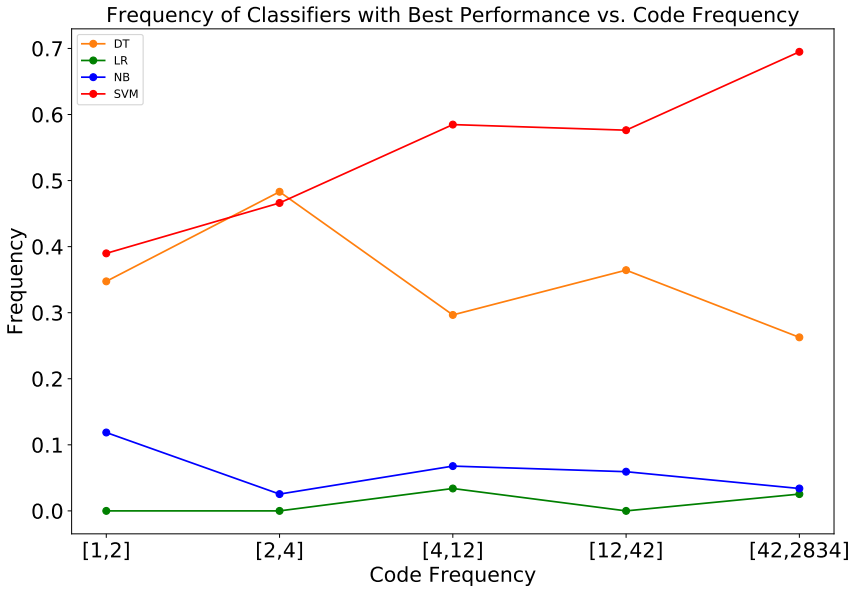


Figure 4.7: Frequency of best classifiers for five CFGs (*COD*). The horizontal axis represents the code frequency for 5 code frequency groups (CFGs). The vertical axis shows the frequency of classifiers that achieve a maximum F-measure score for codes in each group. The higher the frequency, the better the performance.

frequency increases, the SVM takes a larger percentage of the best performance combinations. However, for DT, more features may lead the classifier to generate an over-fitting tree, which fits the training model well but is poor at generalising to out-of-sample data.

Feature Frequency

From the best feature combinations, we extract the frequency of each individual feature. Figure 4.8 shows the frequency of individual features in terms of code frequency. Word level feature types (w1, w2, w3) are represented in green, character level feature types (q1, q2, q3) are represented in red, and skip features are shown in blue.

It can be seen from Figure 4.8 that when the code frequency increases, the frequency of word level features increases while the frequency of the character level and skip features decreases. In general, when code frequency is relatively low (1st-4th CFGs), "q2" and "q3" have a good (with around 0.4 -0.6 frequency) and stable (with small changes) performance, and when there are enough training records (5th CFG), word level features can provide better performance. The "q1" feature type is less competitive compared with other feature types, with frequency in the range 0.1-0.2 for all CFGs.

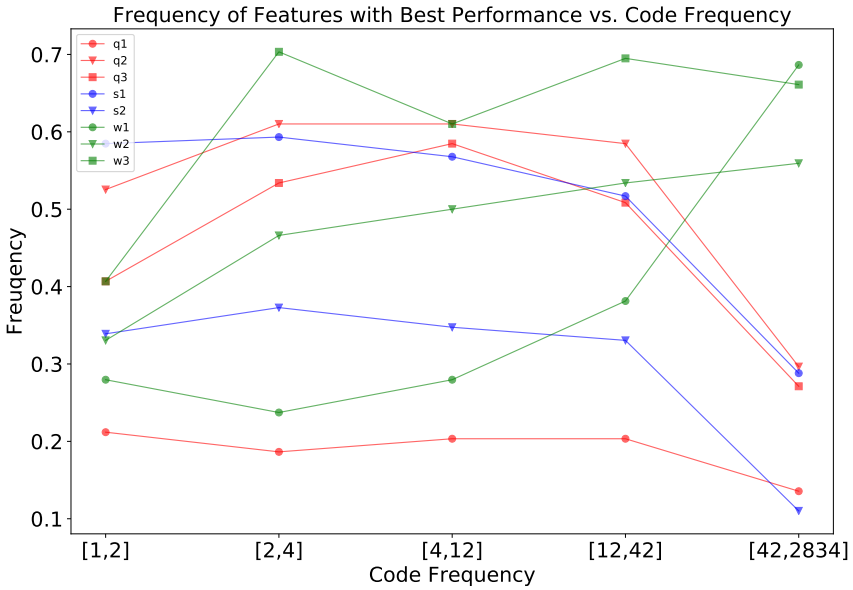


Figure 4.8: Frequency of best features for five CFGs (*COD*). The horizontal axis represents the code frequency for 5 CFGs. The vertical axis shows the frequency of individual feature types that achieve a maximum F-measure score for codes in each group. The higher the frequency, the better the performance.

When the number of training of records is very small, the input records cannot provide enough features and valid feature weights for model training. In this case, character level feature types can provide more information (features) to the classifiers compared with the word level feature types. Thus when code frequency is low, the character level features perform better.

When there are enough training records, we can have a large number of word level features, which can provide sufficient information for the model to classify the unobserved records. Furthermore, each feature generated from word level types can be more independent and meaningful compared with character level features. The character level features may interfere with the classification, appearing as noise in some cases. Thus when the code frequency is high, word level features perform better.

In conclusion, code frequency significantly influences the performance of classifiers. There are obvious patterns between classifiers/feature types and the best performance in terms of different code frequencies. For low code frequencies, DT and character level feature types tend to perform better; while for high code frequencies, SVM and word level feature types tend to perform better.

(3) Text length

Define the word length of a code c as the average number of words in all training records for c. We use the word length to indicate the text length for codes. Similarly to the binning code frequency groups, we first sort all the codes in terms of the text length and then bin them into 3 groups, where each group has approximately the same number of codes (around 197). The reason we choose 3 groups rather than the same number as CFGs (5 groups) is that we want to make the text length difference between groups more obvious. The group bounds can be overlapped since codes with the same text length can be binned into different groups. It follows that each word length group (WLG) with its corresponding word length interval is:

- WLG 1: [1,2]
- WLG 2: [2,3]
- WLG 3: [3,11]

For each WLG, we did the similar analysis as for CFGs. We first rank the best combinations, features and classifiers and show the top 5 rank results in Appendix. Then we show the line plots for the frequency of classifiers in Figure 4.9. The frequency of features does not change too much in terms of different WLGs, so we will not show the line plots for frequency of features.

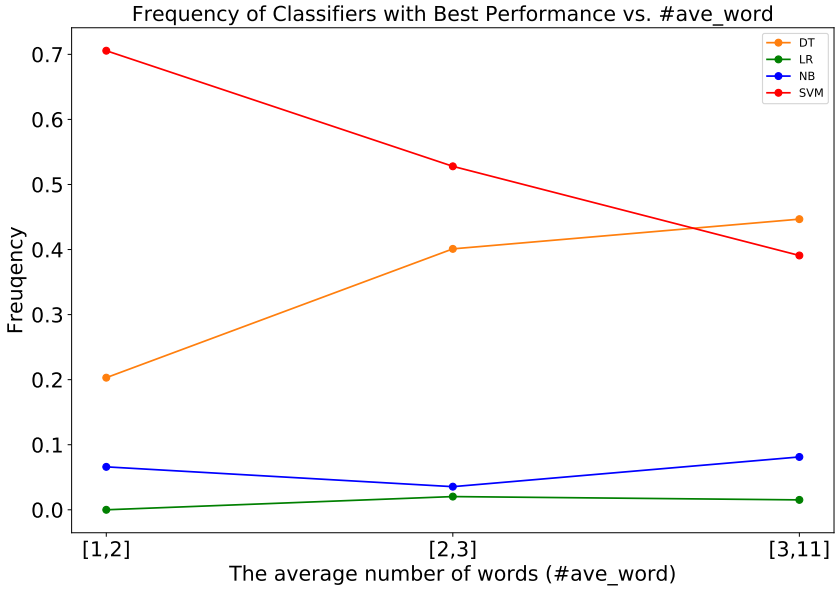


Figure 4.9: Frequency of best classifiers for three WLGs (*COD*). The horizontal axis shows the the average word length. The vertical axis shows the frequency of classifiers in the best combinations. The higher the frequency, the better the performance.

Classifier Frequency

Figure 4.9 shows the frequency of classifiers achieving maximum F-measure score in terms of text length. The frequency can be calculated by Equation 4.1. It is clear that when the text length increases, the percentage of SVM achieving best result decreases and the percentage of DT increases. NB and LR performance worse than SVM and DT for all groups.

We have identified that word level features suit the DT classifier well when doing the overall performance analysis; a larger number of words provides the DT more valid information about the code and the DT tends to generate a more precise tree which describes the code in a more detailed way. Thus when the word length for codes increases, the DT classifier performs better.

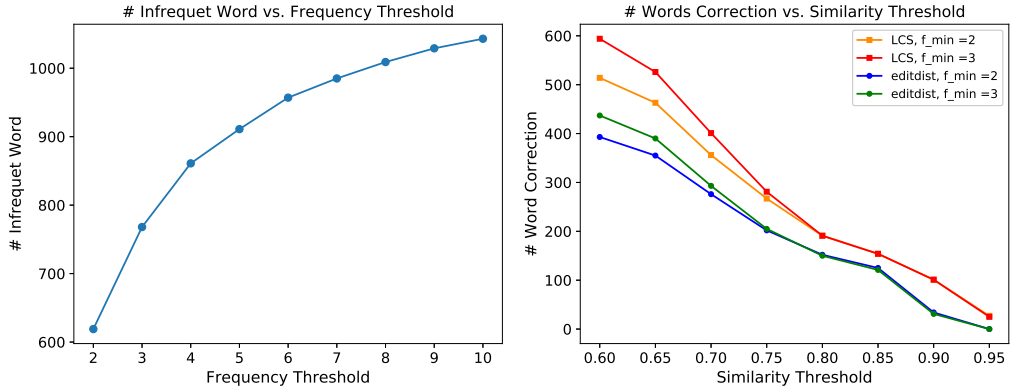
However, for SVM, more features included for one code means higher dimensional data needs to be dealt with. From the length distribution in Figure 4.4, we know that longer word length records tend to have a smaller quantity, which means when the code word length increases, the records provide a high variety of features which only appear once or twice. It makes it more difficult for SVM to linearly separate training records into different classes and find the accurate maximum margin. Thus the SVM algorithm performs worse when the code word length is longer.

In conclusion, word length also influence the performance of classifiers to some extent. There are obvious patterns between classifiers and best performance in terms of different word length. When world length is big, DT performs better than SVM; otherwise, SVM performs better. The frequency of features is largely invariant with different text length, meaning that there is no evident pattern between best feature types and the number of words in records.

2. **Spelling Correction Experiment**

   For the previous experiment, we already performed general cleaning (including removing punctuations, handling white-spaces, removing stop-words and stemming) in the data pre-processing process. Moreover, we found that the general cleaning greatly improves overall model performance. For example, if we use DT and w1, the general cleaning process can improve the F-measure from 0.50 to 0.63.

   Aside from the general cleaning process, we also need to correct misspellings. We use a similarity-based algorithm to automatically detect potential misspellings, which is described in Section 3.1.1.



(a) f_{min} Experiment (b) S_{min} Experiment

Figure 4.10: Spelling correction parameter experiments. Figure (a) shows the number of words separated to infrequent set in terms of different frequency thresholds. Figure (b) shows the number of words being corrected in terms of different similarity thresholds, frequency thresholds and similarity methods

For the spelling correction algorithm, we have three parameters:

- Frequency Threshold f_{min}: specifies the threshold for separating strings into a frequent set and an infrequent set. We choose $f\_min \in$ [2,10], where f_min is an integer.
- Similarity Threshold s_{min}: specifies the threshold for determining whether correct infrequent string into its most similar frequent one. We choose $s\_min \in$ [0.6,0.95] by every 0.05, where s_min is a numeric number.
- Similarity Method $sim_m(s, s'')$: specifies the algorithm m used for detecting similarity between two string s, s''. Algorithm m can be *edit distance* and *longest common sub-string (LCS)*.

Figure 4.10 (a) shows the experiment for frequency threshold f_{min}. f_{min} determines the percentage of words to be in the infrequent set, which are regarded as potential misspellings. When f_{min} becomes larger, the number of words being regarded as infrequent increases. One intuition is that the percentage of potential misspellings should have a relatively small proportion of the total number of unique words (1282). Thus we pick $f_{min} \in \{2, 3\}$ to do further experiments on similarity thresholds and methods.

Figure 4.10 (b) shows the experiment for similarity detection by observing the number of words being corrected. When s_{min} becomes bigger, the number of words corrected decreases. Two similarity methods *edit distance, longest common sub-string (LCS)* are explored. Refer to Section 3.1.1 for a detailed explanation of these two methods. We can see that the *edit distance* algorithm calculates a relatively lower similarity between two strings compared with LCS, which leads to a smaller number of words being corrected.

Origin	Corrected	Similarity
whistle	white	0.7143
iron	bron	0.7500
one	bone	0.7500
caner	cancer	0.8333
spelling	swelling	0.8750
purpuria	purpura	0.8750
injuriy	injury	0.8751
turbercular	tubercular	0.9091
consumtion	consumption	0.9091

Table 4.12: Examples for Spelling Correction. The "origin" column shows the potential misspellings detected by our algorithm. The "corrected" column shows the most similar words that are used to correct the misspellings. The "similarity" column shows the similarity calculated by *edit distance* between the word in "origin" column and "corrected" column.

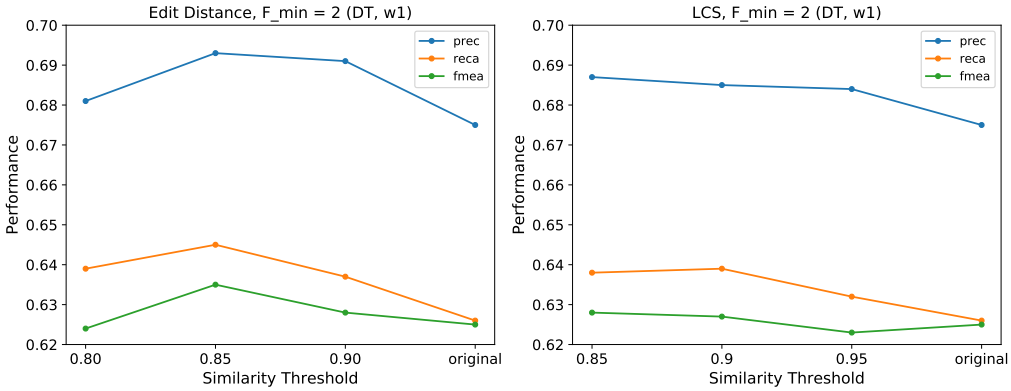
Type	f_min	s_min	sim_m (s, s')	# word correction	other
Spelling Correction	2	0.8	editdist	152	COD DT(entropy) w1 both code type no PCA
		0.85		125	
		0.9		34	
		0.85	LCS	154	
		0.9		101	
		0.95		27	
Original	–	–	–	–	

Table 4.13: Setting for spelling correction experiments.

We hypothesise that the true misspellings in input records will not be larger than 1/8 of the total unique words (around 160), that is, *edit distance* with similarity above 0.8 and LCS with similarity above 0.85. To verify our hypothesis, we observe the words being corrected with different similarities. Table 4.12 shows some examples for words corrected by "editdist, f_min =2" (where editdist represent *edit distance*). We can see that the first three rows are wrongly corrected and the others are rightfully corrected. The similarity below to 0.8 tends to lead a wrong word correction.

Finally, we pick six groups with word correction (where $s_{min} \geq 0.8$ and # word correction ≤ 160) and one group without word correction to conduct a controlled experiment. The parameter settings are specified in Table 4.13. We show the experiment results in Figure 4.11.

We found that spelling correction with suitable parameters can increase the overall performance. From Figure 4.11 we can see that the cleaned data with $s_{min} = 0.85$, $f_{min} = 2$, *edit distance* similarity method gives the best performance, increasing the F-measure without spelling correction by around 3%.



(a) Edit Distance (b) LCS

Figure 4.11: Spelling correction performance experiments. The experiments are based on "DT, w1" combinations. The horizontal axis shows similarity threshold choices for *edit distance* (a) and *LCS* (b); the vertical axis represents the performance. The higher the performance score is, the better the performance.

The spelling correction algorithm may also decrease the performance of classifiers for some codes, since the algorithm may replace some correct words with incorrect ones. From Figure 4.10, we know that in the *COD* dataset, most of the words (around 50 %) only occur once or twice, so finding potential misspellings according to the frequency of words may not be precise enough for the *COD* dataset. If a correct word only occurs once and is very similar to the other frequent words, then it will be mis-replaced.

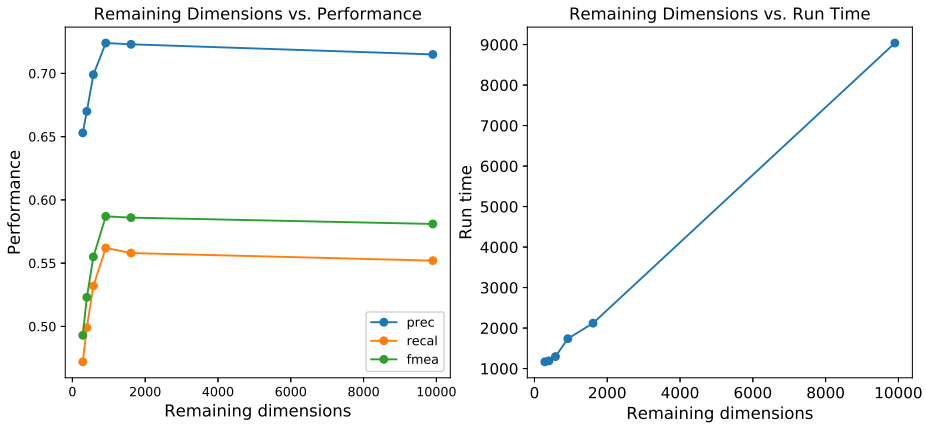
3. **PCA Experiment**

   We use the top ranking combinations selected from the previous "Classifier and Feature Experiment" to conduct our PCA experiment. As mentioned in Section 3.1.3, one of the major problems of implementing PCA is to select the suitable number of components which can achieve a balance between classification performance and computation time.

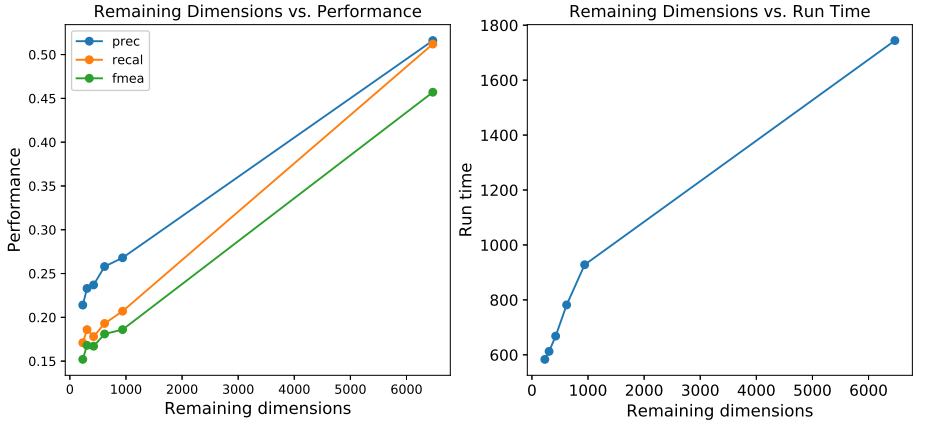
   We conduct our experiment on different numbers of components to observe the changing trend on performance and computation time. The experimental setting is specified in Table 4.14, where *n_components* specifies the percentage of the amount of variance being explained by the remaining dimensions. We only do experiments on SVM and DT since NB and LR were not found to perform well in our previous experiments. Experimental results for SVM and DT are shown in Figure 4.12.

Type	PCA Flag	PCA Para	Other
PCA	True	n_components ∈ {0.8, 0.85, 0.9, 0.95, 0.99}	COD {q2_q3_w2_w3_s1_SVM, q2_q3_w3_s2_DT} Full code
Original	False	–	

Table 4.14: Setting for PCA experiments.



(a) q2, q3, w2, w3, s1 - SVM



(b) q2, q3, w3, s2 - DT

Figure 4.12: PCA experiment results. The horizontal axis shows the remaining dimensions and the vertical axis shows the performance (left figures) or run time (right figures). The run time includes pre-processing (including conducting PCA), training and testing time and the unit of time is specified by seconds. The points on lines indicate the remaining dimensions corresponding with the *n_components* parameter settings and the original dimension, where the original dimension is the largest one.

From Figure 4.12 (a), we can see that when around 1,000 dimensions (*n_components* = 0.95) are retained after conducting PCA, the performance of SVM is *slightly better* than on the original data (9902 dimensions). However, if we continue to choose a smaller *n_components* (< 0.95), the performance scores decrease dramatically. The computational time is approximately proportional to the remaining dimensions. When a lower dimension is used, less run time is needed. The result shows that we can use PCA to achieve a similar or better result with a great amount of computation time reduction when we use an SVM classifier.

Figure 4.12 (b) shows that for DT, when fewer dimensions retained, the performance and computational time decrease with a similar rate. Since the computational time with the original dataset (1800s) for DT is relatively small compared with SVM (9000s), it is not worthwhile to sacrifice the classification accuracy to save running time with DT.

The reason that Figure 4.12 (a) and (b) show different performance trends likely relate to the nature of the classifier. The nature of SVM is to find support vectors from all input features. PCA can help SVM project valuable features into lower dimensions and may also help SVM exclude noise, which may lead SVM to find a more accurate maximum margin. However, DT splits on features and generate logic rules for classification. When PCA projects features on low dimensions, it mixes up the original features together, which may break the original valid splitting rules.

The above experiments demonstrate that the PCA and spelling correction algorithm can help to increase the performance with suitable parameter settings. Therefore, we conduct PCA and spelling correction on the top combinations of parameters identified previously in the Classifier and Feature Experiment. As a result, for the overall performance, the best performance and corresponding parameter settings are shown in Table 4.15, which improves by approximately 4% compared with results in Classifier and Feature Experiment.

Metric	Value	Combination	Run Time
Precision	0.78	[q2, q3, w2, w3, s1, s2] [SVM, linear] [PCA, 95%] [SC, 0.85, 2, editdist]	2128.04
Recall	0.66	[w1] [DT, entropy] [SC, 0.85, 2, editdist]	918.17
F-measure	0.65		

Table 4.15: Best performance summary for *COD* Dataset. Run time includes data preprocessing, model training and evaluation, which is measured by seconds.

4.2.3 *OCC* : Experimental Design and Results

We have explored the data pre-processing methods and classification models for *COD* in the previous section. To explore whether the methods can be generalised to other datasets well and not over-fit the *COD* dataset, we apply these methods for the *OCC* dataset. The *OCC* dataset has similar characteristics to the *COD* dataset, but is from another domain (see Section 4.1).

For the *OCC* dataset, we design two main experiments. One is the classifier and feature experiment, where we use the top ranking best combinations from the *OCC* datasets to train the *OCC* dataset. Another experiment is to apply the spelling correction and PCA algorithm with parameters selected from the *COD* dataset, to observe the classification performance and run time.

1. **Classifier and Feature Experiment**:

   We use similar parameter settings as the *COD* "Classifier and Feature Experiment", except we use the subset of the best feature combinations for *COD* dataset, rather than all combinations of possible individual feature types. More specifically, for each classifier (NB, LR, SVM and DT), we choose 20 feature combinations that give the top F-measure scores.

   For the *OCC* dataset, we are interested in the following questions:

   - Will the performance for the *OCC* dataset be similar to the *COD* dataset?
   - Which combination(s) give the best overall performance?
   - Which classifier(s) tend to give a better overall performance compared with others? Are they consistent with the results for the *COD* dataset?
   - How can we find a balance between accuracy and computational cost, considering the size of the *OCC* dataset is three times bigger than *COD* dataset?

   To evaluate the overall performance, we rank the scores for each evaluation metric. The combinations that achieved the best precision, recall and F-measure respectively are shown in Table 4.16 with the corresponding evaluation scores.

Metric	Combination	Precision	Recall	F-measure
Best Recall	[q2, q3, w1] [NB, 0.1]	0.17	**0.94**	0.24
Best Precision	[q1, q2, q3, w1, w2] [SVM, linear]	**0.91**	0.81	**0.84**
Best F-measure				

Table 4.16: Best performance for *OCC* dataset (without spelling correction and PCA)

The best precision and F-measure is achieved by a long feature type combination and SVM classifier, which is consistency to the results we achieved in the previous *COD* experiments. However, the performance score is much higher than the score for the *COD* dataset, where precision is improved from 0.75 to 0.91 and the

F-measure is improved from 0.63 to 0.84. The improvement can be attributed to the difference between the datasets. Compared with the *COD* dataset, the *OCC* dataset has shorter text descriptions and larger code frequency (see Section 4.1).

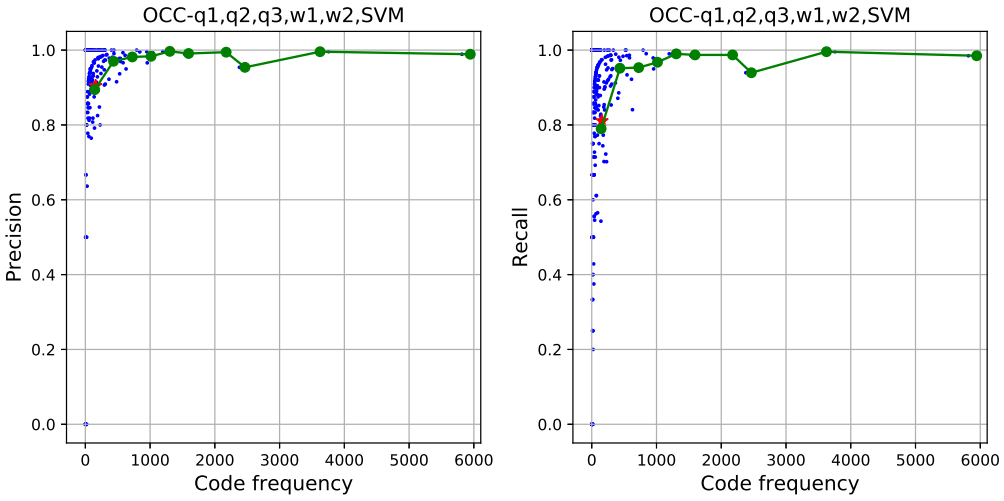
The best recall is achieved by the "q2, q3, w1, NB" combinations, which is quite different from the *COD* dataset. For the *COD* dataset, the combination "q2, q3" gives the best recall (0.54) among all feature combinations for NB classifier, which is much lower than "w1, DT" combinations (0.63 recall). In the *OCC* dataset, the NB classifier outperforms others in terms of recall. However, it is worth noting that the "q2, q3, w1, NB" combination returns very poor precision (0.17) and F-measure (0.24), which means among the records classified as belonging to one code, the correction classification rate is quite low. This may occur when NB classifies most of the records as belonging to one code, then the records which actually belong to the code are more likely to be correctly classified, resulting in a high recall.

To further analyse the performance of the two best combinations, we show their precision and recall performance in terms of code frequency in Figure 4.13. Figure 4.13 (a) and (b) shows the precision and recall performance for the combination "q1, q2, q3, w1, w2, SVM". Compared with the *COD* best performance shown in Figure 4.10, we find that the SVM algorithm performs much better at classification when the code frequency is low. Although the SVM combination does not achieve the best recall, it actually does perform well in terms of recall, aside from a few low frequency codes that skew the result.

Figure 4.13 (c) and (d) show the precision and recall performance for the combination "q2, q3, w1, NB". The precision for NB is highly influenced by the low frequency codes, where most of the codes only achieve around 0.2 precision. The recall is less influenced by code frequency, staying relatively high for all code frequencies. Overall, the NB combination does not suit a situation where we expect the correction rate among all classified positive labels to be as high as possible. But if we would like the ground truth positive label to be correctly classified as much as possible, NB may be a good choice.

To find a balance between classification performance and computational cost, we pick the best combination for each classifier, namely "q2, q3, w1, NB", "q3, LR", "q1, q2, q3, w1, w2, SVM" and "w1, DT", to undertake a comparison between performance and computational cost. The results are shown in Figure 4.14, from which we can see that the NB combination achieves the best recall, but with a low precision and F-measure. The other three classifiers have similar performance in terms of three evaluation metrics. SVM slightly out-performs LR and DT, but is more costly in terms of time.

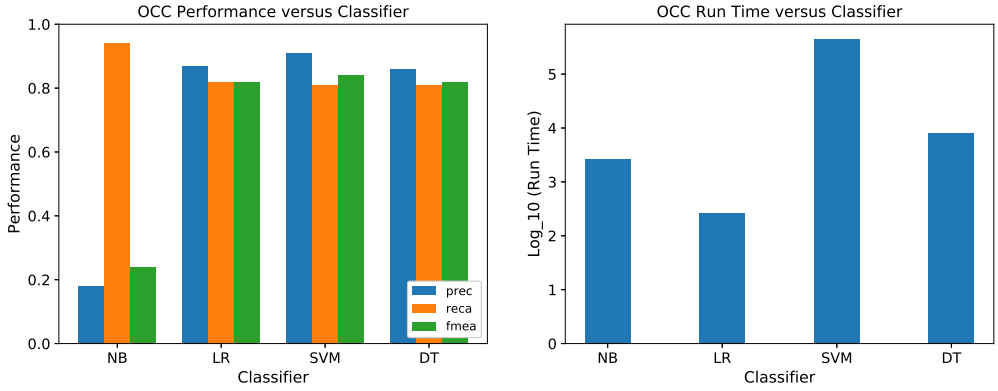
In conclusion, for the *OCC* dataset, LR, SVM and DT have reasonably good performance, while NB has the best recall but low precision. If time matters, LR can be a better choice than SVM, where LR can achieve similar performance with a 99% reduction in computational time.



(a) SVM Precision (b) SVM Recall

(c) NB Precision (d) NB Recall

Figure 4.13: *OCC* performance versus code frequency. The horizontal axis shows the code frequency, which is the number of records for a code. The vertical axis shows the precision/recall. We plot performance scores of each code as blue points and use a binning approach to show the performance trend (green line with marker). The red star represents the average value of both code frequency and performance. The higher the evaluation score (closer to 1), the better the performance.



(a) Performance Comparison (b) Run Time Comparison

Figure 4.14: *OCC* performance and run time comparison.

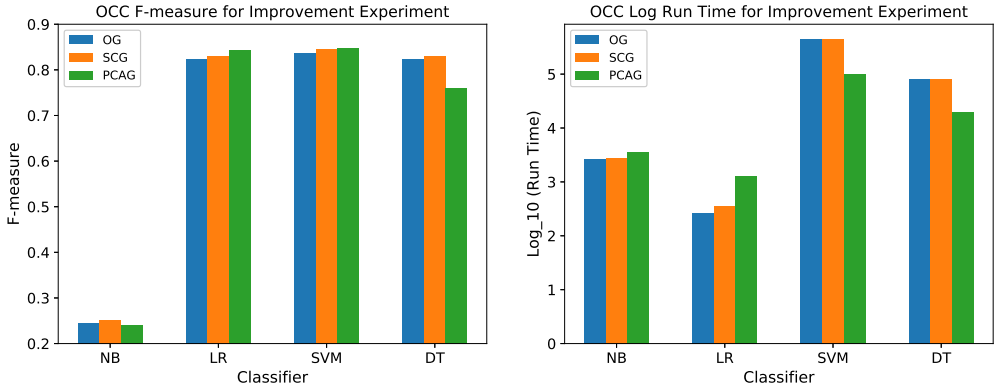
2. **Improvement Experiment**:

   We conduct spelling correction and PCA algorithm on the combinations used above, namely "q2, q3, w1, NB", "q3, LR", "q1, q2, q3, w1, w2, SVM" and "w1, DT". These four combinations are used to represent each classifier"s best performance. We design three experimental groups to explore whether the spelling correction and PCA algorithm and selected parameters can improve the performance of the *OCC* dataset as they did for the *COD* dataset:

   - Original Group (OG): raw performance without spelling correction and PCA. However, the general cleaning process (removing unwanted characters, stemming, etc.) has been applied.
   - Spelling Correction Group (SCG): apply spelling correction algorithm on OG. The parameter choices are the same as the selected ones in the *COD* dataset: similarity threshold = 0.85, frequency threshold = 2, similarity method is *edit distance*.
   - PCA Group (PCAG): apply PCA on OG with 95% n_components.

   The performance and run time for each group are shown in Figure 4.15. For simplicity, we only use F-measure to show the performance change, and the run time includes the time spent computing spelling correction or PCA.

   For spelling correction, the classification performance (F-measure) for all classifier combinations is slightly improved, which shows that the spelling correction algorithm and selected parameters used in the previous experiments are also suitable for the *OCC* dataset. The run time for each combination also slightly increases.



(a) Performance Comparison (b) Run Time Comparison

Figure 4.15: *OCC* performance and run time comparison for Improvement Experiment. Three groups (OG, SCG and PCAG) performance and run time are compared.

For PCA, the F-measure score increase for LR and SVM and decreases for NB and DT. The run time dramatically decreases for SVM and DT. However, since NB and DT combinations originally consume a small amount of time and running PCA on a large dataset consumes a long time, the run time obviously increases for NB and LR.

In conclusion, the overall performance shows consistency for the two datasets in terms of spelling correction and PCA algorithm and selected parameters. The spelling correction method can stably improve the classification performance without consuming too much time. PCA performs well on SVM in terms of both F-measure and runtime. The best performance with spelling correction and PCA are shown in Table 4.17. In the Combination column, the first row shows the feature combination and the second row shows the classifier and corresponding parameters. PCA and SC represents conducting PCA and spelling correction respectively. Run time includes data pre-processing, model training and evaluation, which is measured in seconds.

Metric	**Value**	**Combination**	**Run Time**
Recall	0.94	[q2, q3, w1] [NB, 0.1] [SC, 0.85, 2, editdist]	2620.35
Precision	0.93	[q1,q2,q3,w1,w2] [SVM, linear] [PCA, 0.95] [SC, 0.85, 2, editdist]	94085.03
F-measure	0.85		

Table 4.17: Best performance summary for *OCC* dataset.

The results show that our approaches have great generalisation. The approaches perform well on the *COD* dataset also achieve good performance on the *OCC* dataset. Actually, the results shown in Table 4.17 are much better than the results for the *COD* dataset, where only 0.78 precision, 0.66 recall and 0.65 F-measure is achieved. The reason we have achieved a better results on the *OCC* dataset can be the occupation descriptions tend to be shorter and have less variety compared with the cause of death descriptions. The *OCC* dataset also has less codes with low code frequencies (see Figure 4.2 and 4.3).

4.2.4 Chapter Summary

In this chapter, we analysed the characteristics of the *COD* and *OCC* datasets and discussed the experimental results on both datasets.

In Section 4.1, we first showed the examples of the standard coding systems, ICD-10 and HISCO. Then we summarised the main characteristics of our historical datasets, namely: skewed distribution of classes; non-standard descriptions with different length; misspellings; and and human coding inconsistency. These characteristics provide additional challenges for the classification task.

In Section 4.2, we designed experiments to evaluate the methodologies in Chapter 3. For the *COD* dataset, we designed three main experiments, namely the classifier and feature experiment, spelling correction experiment, and PCA experiment, where the patterns between best combinations and code frequency/text length were explored.

For the *OCC* dataset, we designed two experiments using the parameters that achieved top performance in the *COD* dataset. The results for the *OCC* dataset shows consistency with the *COD* dataset, suggesting that our approaches achieve good generalisation.

In the next chapter, we will conclude this thesis. The overall best classification performance will be summarised and compared with the results of previous research.

Chapter 5

Conclusion

This thesis set out to develop a method for automatically classifying historical descriptions into standard classification systems. Two datasets collected from Scotland were used, namely the cause of death (*COD*) and the occupation (*OCC*) datasets. In both datasets, the historical descriptions and corresponding standard codes are given. The *COD* dataset contains 23,564 records and 591 codes, while the *COD* dataset contains 64,063 records and 418 codes. The characteristics of each dataset were described in Section 4.1.

We developed several methods to address the challenging problems brought by the characteristics of the *OCC* and *COD* datasets. Firstly, the data are noisy (e.g., they include misspellings and narrative descriptions) and inconsistent (i.e. the same text descriptions can be coded into different classes). We developed a general cleaning process (removing unwanted characters, stemming, etc.) and spelling corrections to smooth noisy data. To address the problem of inconsistent coding, we altered all labels for multiple-coded descriptions to the most frequent codes.

Secondly, the skewed distribution of codes and description lengths lead to data sparsity. The low frequency of many codes also makes it difficult to use machine learning methods for classification. We explored different classifier and feature combinations, and the experimental results showed that long feature combinations could improve the performance of codes with low frequency. Furthermore, we conducted PCA to reduce the sparsity and computational time. And after applying PCA, the classification performance was also lightly improved for some classifiers (e.g. SVM).

In conclusion, three main processes were utilised in this project to achieve the classification goal. First, data pre-processing was conducted to smooth noisy historical descriptions, including data cleaning, feature generation and dimensionality reduction. This step is crucial for classification accuracy since the data collected from the real world are noisy and non-standard. Second, we built classifiers for multiple classes. A *one-versus-the-rest* transformation strategy was used to convert the multi-class and multi-label classification problem into multiple binary classification problems. We trained one classifier for each code. Naive Bayes (NB), logistic regression (LR), support vector machine (SVM) and decision tree (DT) were explored as classifier algorithms. Third, the trained models were evaluated. The *hold-out* method was adopted for evaluation, where the given dataset was split into a training dataset and a testing dataset. We split the dataset in a balanced way, where relative class frequencies were

Data	Metric	Combination	Prec	Reca	Fmea	Time (s)
COD	Best Precision	[q2,q3,w2,w3,s1,s2] [SVM, linear] [PCA, 0.95]	**0.78**	0.60	0.62	2128
	Best Recall	[w1] [DT, entropy]	0.70	**0.66**	**0.65**	918
	Best F-measure					
OCC	Best Recall	[q2, q3, w1] [NB, 0.1]	0.18	**0.94**	0.25	2620
	Best Precision	[q1, q2, q3, w1, w2] [SVM, linear] [PCA, 0.95]	**0.93**	0.82	**0.85**	94085
	Best F-measure					

Table 5.1: Overall best performance. In Combination column, the first row shows the features we used, where "q" represents character level features, "w" represents world level features, "s" represents skip grams; and the number followed by character represents the length of grams. For example, "q2" represents character level bigrams. Refer to Section 3.1.2 for further details. The second row represents classifiers, and the last row shows whether PCA was applied. The time is specified by the unit of second.

Data	Best Precision	Best Recall	Best F-measure
COD	0.84	0.40	0.40
OCC	0.61	0.65	–

Table 5.2: Classification results from previous work

approximately preserved in the training and testing datasets. The metrics *precision, recall* and *F-measure* were used for evaluating each classifier's performance on each code. The overall performance was evaluated by the mean value of each classifier's score.

Experiments were conducted on both historical datasets (*COD* and *OCC*), with the main experiments conducted on the *COD* dataset. The *OCC* dataset was used as a validation process for developing the methods, that is, testing whether the developed methods can be generalised to other datasets from other domains which also contain short, noisy text descriptions and an imbalanced class distribution.

The best performance achieved on both datasets is summarised in Table 5.1. For simplicity, the combination column in Table 5.1 only shows the main parameters (features, classifiers, PCA). All experiments were conducted with cleaning process, including both general cleaning and spelling correction. The best scores are highlighted in Table 5.1. For the *COD* dataset, the best recall and F-measure are achieved by the same combination; while for the *OCC* dataset, the best precision and F-measure are achieved by the same combination.

For comparison, we present the results achieved by previous research [Carson et al. 2013; Kirby et al. 2015] in Table 5.2, where they used the same datasets but different pre-processing and classification approaches. For the *OCC* dataset, Kirby et al. [2015] did not include the F-measure as an evaluation measurement. For detailed information about the previous work see Section 2.3.

Compared to the results of previous research (shown in Table 5.2), our approach and methods significantly improved the classification performance, especially for the *OCC* dataset. For the *COD* dataset, the precision of our approaches is slightly lower than the previous result, while the recall and F-measure scores were improved around 50%.

To provide a more detailed overview of the results of this study, we now summarise the key findings of the experiments. For the *COD* dataset, three main experiments were conducted.

1. Classifier and Feature Experiment

   - The most frequent combination that achieved the best F-measure for each code was "q2, q3, w2, w3, s1, SVM". The most frequent feature combination achieved best F-measure was "q2, q3, w2, w3, s1" and the most frequent classifier was SVM. NB and LR performed much worse than SVM and DT in most of cases.
   - SVM achieved better performance when combined with long feature type combinations (for example, "q2, q3, w2, w3, s1, s2") than short combinations; while DT achieved better performance when combined with word level feature types than character level feature types.
   - When code frequency was low, DT and character level feature types performed better; on the other hand, when code frequency was high, SVM and word level feature types performed better.
   - We also found that long feature combinations performed better than short feature combinations when code frequency was low. That is, using more features can improve the classification performance if there are only a few training data records.
   - For codes that had corresponding training records with shorter text length, we found that SVM performed better; when codes had longer records, DT performed better. The feature types' performance did not show a clear pattern in terms of different text lengths.

2. Spelling Correction Experiment

   The best performance was achieved by the spelling correction algorithm (Algorithm 1 see Page 30) with the following parameters: similarity threshold of 0.85; frequency threshold of 2; and where *edit distance* was used as the string matching method.

3. PCA experiment

    - When 95% of the amount of variance was explained by PCA components, SVM achieved the best performance, which was around 1% better than the result (F-measure) achieved without conducting PCA and around 70% computational time was reduced.
    - For DT, the performance and computational time were both proportional to the remaining dimensions. PCA reduced the computational time (around 56%), but at the same time, it greatly reduced the performance in terms of accuracy (around 50%).

For the *OCC* dataset, two comparison experiments were conducted:

1. Classifier and Feature Experiment

    - The performance for the *OCC* dataset was better compared to the *COD* dataset. Similarly as the *COD* dataset, long feature combinations and SVM achieved the best precision; but in contrast to the *OCC* dataset, NB achieved a much higher recall.
    - The LR and DT also achieved reasonably good performance with around 0.8 precision and recall, which is slightly worse than SVM. At the same time, these two classifiers consumed less computational time compared to SVM.

2. Spelling Correction and PCA experiment:

    - The spelling correction algorithm, with frequency threshold set to 2, similarity threshold set to 0.85 and using the *edit distance* method, improved the F-measure by around 1%.
    - PCA slightly improved the performance of SVM (around 1%) and LR (around 2%) and greatly reduced computational time for SVM (around 77%) and DT (around 67%).

5.1 Limitations

Although the results achieved in this study made a substantial improvement over the results of previous work, it still has a number of limitations. In this way, there are three limitations of this study, which also serve to provide a basis for future work.

First, the overall classification performance is negatively influenced by the codes with low frequencies (i.e. with only a few training records). We have found several possible ways to deal with this problem, for example, use character level features and long feature combinations for codes with low frequencies. However, we have not applied these rules to improve the classification performance due to the limit of time. Furthermore, even with the help of these rules, the performance of codes with low frequencies is still much worse than codes with high frequencies.

Second, although the spelling correction algorithm we proposed can improve the classification performance, it also happens that the algorithm sometimes wrongly replaces the correct words with their similar words. As mentioned, in our datasets, many words only appear once or twice, and therefore detecting potential misspellings via term frequency can be inaccurate for our datasets.

Third, the PCA algorithm shows varied results for influencing the performance of different classifiers. For example, PCA can improve the performance of SVM and LR, but worsen the performance of NB and DT. We have not performed a sufficient number of experiments to further explore and verify the reason behind it.

Taking these limitations as a point of departure, we now pose several potential directions for future research.

5.2 Future Work

There are several directions for further development of this work:

- The rules and patterns we found in this project could be applied into a classification process to conduct a rule-based classifier. For example, we could first use character level feature types when the code frequency is low and use word level feature types when the code frequency is high.
- Explore methods to classify unobserved records with codes appearing only a few times in the training dataset. In our work, the overall performance is negatively influenced by those low frequency codes.
- Explore methods to detect misspellings and correct them, especially when the word frequencies are averagely low (i.e. occurring frequency cannot accurately detect the misspellings).
- Analyse the reason that principal component analysis (PCA) has different influences on the performance of different classifiers, and based on that, investigate ways to improve the performance of classification with PCA.
- Hierarchical classification could be considered, given that both historical coding systems are hierarchical. For some codes we lack training records, therefore classifying records into a higher hierarchy first and then classifying them into refined classes may increase the classification accuracy.
- Investigate the applicability of this work to short text classification in other domains, for example, classifying mentions of companies on Twitter into standard industry classification schemes, such as the Global Industry Classification Standard [1].

[1] https://www.msci.com/gics

Appendix

A.1 *COD* Classifiers and Features Experiment

In this section, we demonstrate the top best combinations in terms of five code frequency groups (CFGs) and three world length groups (WLGs), along with the frequency included in a best combination. The best combination is defined as a combination which achieved the best F-measure score for a code among the scores return by all models. The detailed information is introduced in Section 4.2.3 (Classifier and Features Experiment, performance for each classifier).

We show the top 5 combinations, features and classifiers for each CFG in Tables A.1 to A.5, and for each WLG in Tables A.6 to A.8.

Rank	**Combination**		**Feature**		**Classifier**	
	name	freq	name	freq	name	freq
1	q2_q3_w2_w3_s1, SVM	0.2034	q2_q3_w2_w3_s1	0.2034	SVM	0.3898
2	q1_w1_w3_s1, SVM	0.0508	q2_s1	0.0508	DT	0.3475
3	w2_w3, DT	0.0254	q1_w1_w3_s1	0.0508	NB	0.1186
4	w2_s2, SVM	0.0254	w1	0.0339		
5	q2_s1, NB	0.0254	q1_w1_s1_s2	0.0339		

Table A.1: Rank results for CFG 1 (with code frequency [1,2])

Rank	**Combination**		**Feature**		**Classifier**	
	name	freq	name	freq	name	freq
1	q2_q3_w2_w3_s1, SVM	0.2712	q2_q3_w2_w3_s1	0.2712	DT	0.4831
2	q2_q3_w3_s2, DT	0.1102	q2_q3_w3_s2	0.1102	SVM	0.4661
3	w3, DT	0.0678	w3	0.0678	NB	0.0254
4	w2_w3, DT	0.0593	w2_w3	0.0593		
5	s1_s2, SVM	0.0424	s1_s2	0.0508		

Table A.2: Rank results for CFG 2 (with code frequency [2,4])

Rank	Combination		Feature		Classifier	
	name	freq	name	freq	name	freq
1	q2_q3_w2_w3_s1, SVM	0.2712	q2_q3_w2_w3_s1	0.2712	SVM	0.5847
2	w3, DT	0.0508	w3	0.0508	DT	0.2966
3	q1_w1_w3_s1, SVM	0.0508	q1_w1_w3_s1	0.0508	NB	0.0678
4	q2_q3_w3_s2, DT	0.0339	q2_q3_w3_s2	0.0339	LR	0.0339
5	q2_q3_w2_s2, SVM	0.0339	q2_q3_w2_s2	0.0339		

Table A.3: Rank results for CFG 3 (with code frequency [4,12])

Rank	Combination		Feature		Classifier	
	name	freq	name	freq	name	freq
1	q2_q3_w2_w3_s1, SVM	0.2288	q2_q3_w2_w3_s1	0.2288	SVM	0.5763
2	q1_w1_w3_s1, SVM	0.0508	q1_w1_w3_s1	0.0508	DT	0.3644
3	w3, DT	0.0424	w3	0.0424	NB	0.0593
4	q2_q3_w3_s2, DT	0.0424	w1_w2_w3	0.0424		
5	w2_w3, DT	0.0339	q2_q3_w3_s2	0.0424		

Table A.4: Rank results for CFG 4 (with code frequency [12,42])

Rank	Combination		Feature		Classifier	
	name	freq	name	freq	name	freq
1	q2_q3_w2_w3_s1, SVM	0.1102	w1_w3	0.1102	SVM	0.6949
2	w1_w3, SVM	0.1017	w1_w2_w3	0.1102	DT	0.2627
3	w1_w2_w3, SVM	0.0847	q2_q3_w2_w3_s1	0.1102	NB	0.0339
4	w1, SVM	0.0678	w1_w2	0.1017	LR	0.0254
5	w1_w2, SVM	0.0593	w1	0.0763		

Table A.5: Rank results for CFG 5 (with code frequency [42,2834])

Rank	Combination		Feature		Classifier	
	name	freq	name	freq	name	freq
1	q2_q3_w2_w3_s1, SVM	0.264	q2_q3_w2_w3_s1	0.264	SVM	0.7056
2	w1, SVM	0.0609	w1	0.0711	DT	0.203
3	w1_w2_w3, SVM	0.0508	w1_w2_w3	0.066	NB	0.066
4	w1_w3, SVM	0.0457	w1_w3	0.0457		
5	q2_q3_w3_s2, DT	0.0406	q2_q3_w3_s2	0.0406		

Table A.6: Rank results for WLG 1 (with world length [1,2])

Rank	**Combination**		**Feature**		**Classifier**	
	name	freq	name	freq	name	freq
1	q2_q3_w2_w3_s1, SVM	0.2538	q2_q3_w2_w3_s1	0.2538	SVM	0.5279
2	w3, DT	0.0609	w3	0.0609	DT	0.401
3	w2_w3, DT	0.0609	w2_w3	0.0609	NB	0.0355
4	q1_w1_w3_s1, SVM	0.0609	q1_w1_w3_s1	0.0609	LG	0.0203
5	q2_q3_w3_s2, DT	0.0457	q2_q3_w3_s2	0.0457		

Table A.7: Rank results for WLG 2 (with world length [2,3])

Rank	**Combination**		**Feature**		**Classifier**	
	name	freq	name	freq	name	freq
1	q2_q3_w2_w3_s1, SVM	0.132	q2_q3_w2_w3_s1	0.132	DT	0.4467
2	q2_q3_w3_s2, DT	0.0508	q2_q3_w3_s2	0.0508	SVM	0.3909
3	w3, DT	0.0457	w3	0.0457	NB	0.0812
4	q1_w1_w3_s1, SVM	0.0355	q1_w1_w3_s1	0.0355	LG	0.0152
5	w2_w3, DT	0.0305	w2_w3	0.0305		

Table A.8: Rank results for WLG 3 (with world length [3,11])

Bibliography

ALY, M. 2005. Survey on multiclass classification methods. *Neural Netw 19*, 1–9.

ANAND, R., MEHROTRA, K., MOHAN, C. K., AND RANKA, S. 1995. Efficient classification for multiclass problems using modular neural networks. *IEEE Transactions on Neural Networks 6*, 1, 117–124.

APACHE SOFTWARE FOUNDATION. 2010. Apache OpenNLP. Available at: http://opennlp.apache.org/, Accessed January 29, 2013.

BAKER, L. D. AND MCCALLUM, A. K. 1998. Distributional clustering of words for text classification. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98 (New York, NY, USA, 1998), pp. 96–103. ACM.

BAOXUN, X., XIUFENG, G., YUNMING, Y., AND JIEFENG, C. 2012. An improved random forest classifier for text categorization. *JOURNAL OF COMPUTERS 7*, 12 (Dec), 2913–2920.

BELLMAN, R. 2013. *Dynamic programming*. Courier Corporation.

BISHOP, C. M. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

BOTTERO, W. AND PRANDY, K. 2001. Women's occupations and the social order in nineteenth century britain. *Sociological Research Online 6*, 2, 1–17.

BREIMAN, L. 2001. Random forests. *Machine Learning 45*, 1 (Oct), 5–32.

CARSON, J., KIRBY, G., DEARLE, A., WILLIAMSON, L., GARRETT, E., REID, A., AND DIBBEN, C. 2013. *Exploiting historical registers: Automatic methods for coding c19th and c20th cause of death descriptions to standard classifications*, pp. 598–607. Eurostat.

CHANG, C.-C. AND LIN, C.-J. 2011. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST) 2*, 3, 27.

CHEN, Y.-L., HSU, C.-L., AND CHOU, S.-C. 2003. Constructing a multi-valued and multi-labeled decision tree. *Expert Systems with Applications 25*, 2, 199 – 209.

CHRISTEN, P. 2006. A comparison of personal name matching: Techniques and practical issues. In *Sixth IEEE International Conference on Data Mining - Workshops (ICDMW'06)* (Dec 2006), pp. 290–294.

CHRISTEN, P. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Data-Centric Systems and Applications. Springer Berlin Heidelberg.

CONNELLY, R., PLAYFORD, C. J., GAYLE, V., AND DIBBEN, C. 2016. The role of administrative data in the big data revolution in social science research. *Social Science Research 59*, 1 – 12. Special issue on Big Data in the Social Sciences.

DIETTERICH, T. G. 2000. *Ensemble Methods in Machine Learning*. In Multiple Classifier Systems. Berlin, Heidelberg: Springer (Vol. 1857, pp. 115).

ELISSEEFF, A. AND WESTON, J. 2002. A kernel method for multi-labelled classification. In *Advances in neural information processing systems* (2002), pp. 681–687.

FODOR, I. K. A survey of dimension reduction techniques.

FOX, E. 2017. Decision trees: Overfitting.

FRIEDMAN, C. AND SIDELI, R. 1992. Tolerating spelling errors during patient validation. *Computers and Biomedical Research 25*, 5, 486 – 509.

GAIL, H. R., HANTLER, S. L., LAKER, M. M., LENCHNER, J., AND MILCH, D. 2016. Method and apparatus for automatic detection of spelling errors in one or more documents. US Patent 9,465,791.

GENKIN, A., LEWIS, D. D., AND MADIGAN, D. 2007. Large-scale bayesian logistic regression for text categorization. *Technometrics 49*, 3, 291–304.

GUNN, S. R. ET AL. 1998. Support vector machines for classification and regression. *ISIS technical report 14*, 1, 5–16.

GUTHRIE, D., ALLISON, B., LIU, W., GUTHRIE, L., AND WILKS, Y. 2006. A closer look at skip-gram modelling. In *Proceedings of the 5th international Conference on Language Resources and Evaluation (LREC-2006)* (2006), pp. 1–4. sn.

GUYON, I. AND ELISSEEFF, A. 2003. An introduction to variable and feature selection. *J. Mach. Learn. Res. 3*, 1157–1182.

HAN, J., KAMBER, M., AND PEI, J. 2011. *Data Mining: Concepts and Techniques* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

JOACHIMS, T. 1996. A probabilistic analysis of the rocchio algorithm with tfidf for text categorization. Technical report, Carnegie-mellon univ pittsburgh pa dept of computer science.

JOACHIMS, T. 1998. *Text categorization with Support Vector Machines: Learning with many relevant features*, pp. 137–142. Springer Berlin Heidelberg, Berlin, Heidelberg.

JOHN, G. H. AND LANGLEY, P. 1995. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, UAI'95 (San Francisco, CA, USA, 1995), pp. 338–345. Morgan Kaufmann Publishers Inc.

JOLLIFFE, I. T. 1986. *Principal Component Analysis and Factor Analysis*, pp. 115–128. Springer New York, New York, NY.

KEOGH, E. AND MUEEN, A. 2017. Curse of dimensionality. In *Encyclopedia of Machine Learning and Data Mining*, pp. 314–315. Springer.

KESKUSTALO, H., PIRKOLA, A., VISALA, K., LEPPÄNEN, E., AND JÄRVELIN, K. 2003. Non-adjacent digrams improve matching of cross-lingual spelling variants. In M. A. NASCIMENTO, E. S. DE MOURA, AND A. L. OLIVEIRA Eds., *String Processing and Information Retrieval* (Berlin, Heidelberg, 2003), pp. 252–265. Springer Berlin Heidelberg.

KIM, S.-B., HAN, K.-S., RIM, H.-C., AND MYAENG, S. H. 2006. Some effective techniques for naive bayes text classification. *IEEE Transactions on Knowledge and Data Engineering 18*, 11 (Nov), 1457–1466.

KIRBY, G., CARSON, J., DUNLOP, F., DIBBEN, C., DEARLE, A., WILLIAMSON, L., GARRETT, E., AND REID, A. 2015. *Automatic Methods for Coding Historical Occupation Descriptions to Standard Classifications*, pp. 43–60. Springer International Publishing, Cham.

KIRBY, G., HAJIARABDERKANI, M., DEARLE, A., CARSON, J., DUNLOP, F., DIBBEN, C., AND WILLIAMSON, L. 2015. Automatic extraction of multiple underlying causes from textual death records (8 2015).

KOHAVI, R. ET AL. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, Volume 14 (1995), pp. 1137–1145. Montreal, Canada.

KOTSIANTIS, S. AND KANELLOPOULOS, D. 2005. Discretization techniques: A recent survey. *32*, 47–58.

KOWSARI, K., BROWN, D. E., HEIDARYSAFA, M., MEIMANDI, K. J., GERBER, M. S., AND BARNES, L. E. 2017. Hdltex: Hierarchical deep learning for text classification. *arXiv preprint arXiv:1709.08267*.

KUKICH, K. 1992. Techniques for automatically correcting words in text. *ACM Comput. Surv. 24*, 377–439.

LAI, S., XU, L., LIU, K., AND ZHAO, J. 2015. Recurrent convolutional neural networks for text classification. In *AAAI*, Volume 333 (2015), pp. 2267–2273.

LARSON, R. R. 2009. Introduction to information retrieval. *Journal of the American Society for Information Science and Technology 61*, 4, 852–853.

LESKOVEC, J., RAJARAMAN, A., AND ULLMAN, J. D. 2014. *Mining of massive datasets*. Cambridge university press.

LEWIS, D. D. AND RINGUETTE, M. 1994. A comparison of two learning algorithms for text categorization. In *In Third Annual Symposium on Document Analysis and Information Retrieval* (1994), pp. 81–93.

LOPER, E. AND BIRD, S. 2002. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02 (Stroudsburg, PA, USA, 2002), pp. 63–70. Association for Computational Linguistics.

MARINA, S. AND GUY, L. 2009. A systematic analysis of performance measures for classification tasks. *Information Processing Management 45*, 4, 427 – 437.

MAZEIKA, A. AND BOHLEN, M. H. 2006. Cleansing databases of misspelled proper nouns. In *In CleanDB Workshop* (2006), pp. 63–70.

MEIER, L., VAN DE GEER, S., AND BÜHLMANN, P. 2008. The group lasso for logistic regression. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 70*, 1, 53–71.

MURPHY, K. P. 2006. Naive bayes classifiers. *University of British Columbia 18*.

NATIONAL RECORDS OF SCOTLAND. 1901. National records of 1901 census in scotland.

NATIONAL RECORDS OF SCOTLAND. 1980. Scottish birth, marriage and death records. *Who Do You Think You Are? Magazine*.

PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12*, 2825–2830.

READ, J., PFAHRINGER, B., HOLMES, G., AND FRANK, E. 2011. Classifier chains for multi-label classification. *Machine Learning 85*, 3 (Jun), 333.

REID, A., GARRETT, E., DIBBEN, C., AND WILLIAMSON, L. 2015. a confession of ignorance: deaths from old age and deciphering cause-of-death statistics in scotland, 18551949. *The History of the Family 20*, 3, 320–344. PMID: 26900320.

RENNIE, J. D. 2001. Improving multi-class text classification with naive bayes.

SCHULZ, K. U. AND MIHOV, S. 2002. Fast string correction with levenshtein automata. *International Journal on Document Analysis and Recognition 5*, 1 (Nov), 67–85.

SEBASTIANI, F. 2002. Machine learning in automated text categorization. *ACM Comput. Surv. 34*, 1 (March), 1–47.

STUART, K., A.; ORD. 1994. *Kendall's Advanced Theory of Statistics: Volume IDistribution Theory*. Edward Arnold.

TAKAHASHI, F. AND ABE, S. 2002. Decision-tree-based multiclass support vector machines. In *Neural Information Processing, 2002. ICONIP'02. Proceedings of the 9th International Conference on*, Volume 3 (2002), pp. 1418–1422. IEEE.

US BUREAU OF LABOR STATISTICS. 2010. Standard occupational classification (soc) system. Available at: http://www.bls.gov/soc/ Accessed 2014.

UUZ, H. 2011. A two-stage feature selection method for text categorization by using information gain, principal component analysis and genetic algorithm. *Knowledge-Based Systems 24*, 7, 1024 – 1032.

VAN LEEUWEN, M. H., MAAS, I., AND MILES, A. 2002. *HISCO: Historical international standard classification of occupations*. Leuven Univ Pr.

WAGNER, R. A. AND FISCHER, M. J. 1974. The string-to-string correction problem. *J. ACM 21*, 1 (Jan.), 168–173.

WHO. 1990. Who international classification of diseases(icd-10). *WHO. World Health Organization*. Available at: http://www.who.int/classifications/icd/en/ Accessed 2016.

WHO. 2004. *ICD-10 : international statistical classification of diseases and related health problems / World Health Organization* (10th revision, 2nd ed. ed.). World Health Organization Geneva.

WILLETT, P. 2006. The porter stemming algorithm: then and now. *Program 40*, 3, 219–223.

WOLD, S., ESBENSEN, K., AND GELADI, P. 1987. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems 2*, 1, 37 – 52. Proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists.

WOOLLARD, M. 2014. 3.1 administrative data: Problems and benefits. a perspective from the united kingdom1. *Facing the Future: European Research Infrastructures for the Humanities and Social Sciences*, 49.

YANG, Y. AND PEDERSEN, J. O. 1997. A Comparative Study on Feature Selection in Text Categorization.