Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für verteilte Systeme
Fachgebiet Wissensbasierte Systeme
Forschungszentrum L3S

Diploma Thesis

Unsupervised Post-Correction of OCR Errors

Author:
Kai Niklas

Examiner:
Prof. Dr. techn. Wolfgang Nejdl

Second Examiner:
Prof. Dr. Rainer Parchmann

Supervisors:
M.Sc. Nina Tahmasebi
Dr. Thomas Risse

Hanover, 11th June 2010

Abstract

The trend to digitize (historic) paper-based archives has emerged in the last years. The advantages of digital archives are easy access, searchability and machine readability. These advantages can only be ensured if few or no OCR errors are present. These errors are the result of misrecognized characters during the OCR process. Large archives make it unreasonable to correct errors manually. Therefore, an unsupervised, fully-automatic approach for correcting OCR errors is proposed. The approach combines several methods for retrieving the best correction proposal for a misspelled word: A general spelling correction (Anagram Hash), a new OCR adapted method based on the shape of characters (OCR-Key) and context information (bigrams). A manual evaluation of the approach has been performed on The Times Archive of London, a collection of English newspaper articles spanning from 1785 to 1985. Error reduction rates up to 75% and F-Scores up to 88% could be achieved.

Zusammenfassung

Der Trend papierbasierte (historische) Archive zu digitalisieren hat in den letzten Jahren zugenommen. Die einfache Zugänglichkeit, Durchsuchbarkeit und Maschinenlesbarkeit sind die Vorteile digitaler Archive gegenüber papierbasierten. Diese Vorteile können nur sichergestellt werden, wenn nur wenige oder keine OCR Fehler enthalten sind. Solche Fehler werden durch falsch erkannte Zeichen während des OCR Prozesses verursacht. Da bei großen Archiven eine manuelle Korrektur nicht sinnvoll ist, wird in dieser Arbeit eine Methode vorgeschlagen, um OCR Fehler voll automatisch zu korrigieren. Die vorgeschlagene Methode verbindet dabei mehrere Methoden, um den besten Korrekturvorschlag für ein falsch geschriebenes Wort zu finden: Eine allgemeine Rechtschreibkorrektur (Anagram Hash), eine neue für OCR Fehler angepasste Methode, basierend auf der Form bzw. Gestalt von Zeichen und Kontextinformationen (Bigramme). Eine manuelle Evaluation der vorgeschlagenen Methode wurde auf dem Zeitungsarchiv (The Times, London) durchgeführt, einer Sammlung englischer Zeitungsartikel von 1785 bis 1985. Fehlerreduktionsraten bis zu 75% und F-Scores bis zu 88% konnten erzielt werden.

Contents

List of Figures

List of Tables

List of Examples

List of Algorithms

1 Introduction

1.1 Motivation

In the last years the trend to digitize (historic) paper based documents such as books and newspapers, has emerged. The aim is to preserve these documents and make them fully accessible, searchable and processable in digital form. Knowledge contained in paper based documents is more valuable for today's digital world when it is available in digital form.

The first step towards transforming a paper based archive into a digital archive is to scan the documents. The next step is to apply an OCR (Optical Character Recognition) process, meaning that the scanned image of each document will be translated into machine processable text. Due to the print quality of the documents and the error-prone pattern matching techniques of the OCR process, OCR errors occur. Modern OCR processors have character recognition rates up to 99% on high quality documents. Assuming an average word length of 5 characters, this still means that one out of 20 words is defect. Thus, at least 5% of all processed words will contain OCR errors. On historic documents this error rate will be even higher because the print quality is likely to be of lower quality.

After finishing the OCR process several post-processing steps are necessary depending on the application, e.g. tagging the documents with meta-data (author, year, etc.) or proof-reading the documents for correcting OCR errors and spelling mistakes. Data which contains spelling mistakes or OCR errors is difficult to process. For example, a standard full-text search will not retrieve misspelled versions of a query string. To fulfil application's demanding requirements toward zero errors, a post-processing step to correct these errors is a very important part of the post-processing chain.

A post-processing error correction system can be manual, semi-automatic or fully-automatic. A semi-automatic post-correction system detects errors automatically and proposes corrections to human correctors who then have to choose the correct proposal. A fully-automatic post-correction system does the detection and correction of errors by its own. Because semi-automatic or manual corrections require a

lot of human effort and time, fully-automatic systems become necessary to perform a full correction.

Spell checkers and post-processing correction systems are mostly dictionary based and work in two steps. The first step compares each token with a word list. If the token is not present in the list, it is likely misspelled and presented as a potential error. The second step generates correction proposals bases on the underlying dictionary. The proposals are then ranked with different similarity measures. Usually, these proposal lists are ranked by their likeliness to be the best matching correction proposal.

Obviously a fully-automatic system relies heavily on the 1-best correction proposal. One state-of-the-art system for human spelling mistakes is GNU Aspell 0.60.6 [Atk08], here on referenced to as Aspell. It is able to propose correction lists including an appropriate proposal in about 90% for a test file which contains 547 human spelling mistakes. Unfortunately, only every second misspelling can be corrected fully-automatically by Aspell using the 1-best proposal.

1.2 Objective of the Thesis

Using static dictionaries for correcting errors has several disadvantages when applying on historic documents. For example names or language variations are likely not included in today's dictionaries and may be detected as wrong or even miscorrected to a wrong word. To overcome such problems, high frequent words from the archive itself can be used to derive adequate word lists. Assuming that frequent words are more likely to be correct, the derived word list enables correcting names as well as domain specific words. Furthermore, using the archive for correcting errors is language independent and can tackle languages with less resource, e.g. historic languages. Despite the advantage of a corpus derived dictionary the method also offers some disadvantages. Questions like "When is a word frequent enough to be correct?" have to be investigated.

To find the best matching proposal, similarity measures are very important. Since OCR errors differ from human spelling mistakes they require a special handling. To enable a better correction of OCR errors, this thesis proposes a new similarity key technique, called OCR-Key. OCR-Key exploits the nature of OCR systems and their generated errors based on the shape of characters. OCR-Key classifies characters into equivalence classes which look similar, e.g. "e $\approx$ c". Additionally, the structure of words is exploited, e.g. $iii \rightarrow m$. This can simplify the retrieval

of adequate words and boost the 1-best proposal rate when used in combination with other correction techniques.

To find the best matching correction for an OCR error, the context can be used as another source of information. This can be done by using word pairs (bigrams on word level). Therefore, a pre-processing step is necessary to collect bigrams and their occurrence. For example the fragment *another sample sentence* can be split up into two bigrams: *another sample* and *sample sentence*. Assume that the word *sampe* of the fragment *another sampe sentence* should be corrected. The correction proposals will be *sample*, *same* or something else. But using the context bigram information the resulting correction of the fragment *another sampe sentence* will be *another sample sentence*.

The core correction method of this thesis is based on the Anagram Hash technique first introduced by Reynaert [Rey04]. It uses a bad hashing function which assigns the same number to all words which have the same characters in common, e.g. *with* and *wtih* have the same hash value. Using the hash values of single characters enables to add, subtract or both on word's hash values to find similarly spelled words, e.g. adding the hash value of the character *l* to the hash value of the word *sampe* will retrieve the word *sample*. The advantages of the anagram hash is the speed of retrieving corrections, the ability to compute on numbers instead of characters and a natural ranking which will be presented later. Furthermore, bigrams can be hashed, which makes it possible to split up misspelled words properly, e.g. adding the character hash values of *l* and *whitespace* corrects the token *sampesentence* to *sample sentence*.

This thesis will focus on describing, analysing, applying and evaluating a fully-automatic post-correction system on the collection of The Times of London [Tim]. The collection spans from 1785 to 1985 and contains over 7 billion white-space separated tokens and nearly 8 million articles which motivates the usage of a fully-automatic system. To achieve a good working system, this thesis will combine the Anagram Hash method which uses a corpus derived dictionary to overcome dictionary problems, a bigram approach on word level to consider the context and OCR-Key, a new method specialized in correcting OCR errors.

Due to the lack of ground-truth material of The Times Archive, this thesis cannot be evaluated automatically using word lists which compare correct with error-prone word forms. Thus, human assessors will evaluate the presented OCR correction system with the help of a tool especially designed for the task.

1.3 OCR Error Classification

Before errors can be corrected they have to be identified and classified. A proper classification is important in order to know which kind of errors occur. Once this is known it can help to identify methods which are able to correct them. In related work there is one main classification scheme widely spread which divides errors into two classes: non-word and real-word errors (e.g. [Kuk92]).

Non-word error: A non-word error occurs if the original word is not contained in any dictionary (e.g. *tho* instead of *the*). This is the common error most spell checkers attempt to correct.

Real-word error: A real-word error occurs if the original word is correctly spelled but incorrectly used (e.g. *peace of paper* instead of *piece of paper*). Those kind of errors are only correctable using context information.

This classification is obviously not sufficient with respect to OCR errors as it only divides errors into two groups. Furthermore, it is unclear what happens to words which are correct but not contained in any dictionary, e.g. names, out-dated terms or (historic) spelling variations. With respect to OCR errors, this classification also lacks of several OCR related aspects. Hence, a better classification is needed here to determine which kind of errors occur.

Segmentation errors. Different line, word or character spacings lead to misrecognitions of white-spaces, causing segmentation errors (e.g. *thisis* instead of *this is* or *depa rtmen t* instead of *department*).

Hyphenation errors. Tokens are split up at line breaks if they are too long, which increase the number of segmentation errors (e.g. *de- partment*).

Misrecognition of characters. Dirt and font-variations prevent an accurate recognition of characters which induce wrong recognitions of words (e.g. *souiid* instead of *sound* or *&-Bi1rd#!* instead of *Bird*).

Punctuation errors. Dirt causes misrecognitions of punctuation characters. This means points, commas, etc. occur more often in wrong places with missing or extra white-spaces etc.

Case sensitivity. Due to font variations, upper and lower case characters can be mixed up (e.g. *BrItaIn* or *BRITAIN*).

Changed word meaning. Misrecognized characters can lead to new words which are often wrong in context but spelled correctly (e.g. *mad* instead of *sad*).

Error class	recognized word	correct word
Segmentation (missing space)	`thisis`	`this is`
Segmentation (split word)	`depa rtme nt`	`department`
Hyphenation error	`de- partment`	`department`
Character misrecognition	`souiid`	`sound`
Number substitution	`Opporunity`	`Opportunity`
Special char insertion	`electi'on`	`election`
Changed word meaning	`mad`	`sad`
Case sensitive	`BrItaIn`	`Britain`
Punctuation	`this.is`	`this is`
Destruction	`NI.I II I`	`Minister`
Currencies	`?20`	`$20`

Table 1.1: OCR Error Examples

Some typical sample errors can be found in Table 1.1. Especially destroyed tokens occur often in OCR processed texts with low quality documents, e.g. `NI.I II I` instead of `Minister`.

1.4 Outline of the Thesis

This work is structured as follows: Chapter 2 gives a survey of related work dealing with different strategies on correcting (human) spelling mistakes and OCR errors. Advantages and disadvantages are highlighted. Methods which are used in this thesis are described in more details. Chapter 3 introduces a new similarity key technique for correcting OCR errors by exploiting the nature of OCR processors, mainly based on the shape of characters. Chapter 4 explains the correction strategy for spell-checking and retrieving correction proposals for misspelled words. The co-operation between methods presented in related work and the new similarity key technique are explained in detail. In Chapter 5, the proposed correction strategy is evaluated on the data of The Times Archive and compared with Aspell. A discussion about the results is also given. Finally, the thesis concludes in Chapter 6.1 and gives an outlook on future work.

2 Related Work & Used Methods

There has been much effort in the field of correcting (human) spelling mistakes and OCR errors. Hence this chapter can only give a small overview of approaches available. The first section presents mainly general correction methods for spelling mistakes and some specially adapted methods for human spelling mistakes. Section 2.2 presents some works which have applied general and adapted correction methods on OCR errors. Methods which are used in this thesis are presented in depth in Section 2.3.

2.1 Correcting Spelling Mistakes

In Section 1.3 two main types of spelling mistakes are presented: non-word errors and real-word errors. A non-word error occurs if a word is not contained in any dictionary, e.g. *tho* instead of *the*. A real-word error occurs if a word is used in wrong context, e.g. *peace of paper* instead of *piece of paper*. Both error types can be tackled with different methods presented next.

Non-Word Errors

The general set-up for spell-checking words is to use a dictionary which contains correctly spelled words. Words are then checked against the dictionary. If they are not contained in the dictionary, similar words are retrieved from the dictionary as correction candidates (also called correction proposals). The similarity of two words can be measured in several ways. Three similarity measures are presented in the following:

1. Levenshtein-Distance: Minimum number of insertions, deletions and substitutions of characters which are necessary in order to transform a string x into y [Lev66]. More details about the Levenshtein-Distance can be found in Section 2.3.

2. Damerau-Levenshtein-Distance: This is an extension of the Levenshtein-Distance. It allows transpositions of two adjacent characters additionally, Damerau [Dam64]. With respect to OCR errors this extension is likely not meaningful because characters are not transposed by an OCR processor. This is more typical human spelling mistakes.

3. n-grams on character level: A character n-gram is a subsequence of n characters of a word. The fraction of n-grams which both words have in common and unique n-grams which both produce, can be used as similarity measure. A correction method based on n-grams has been used for example by Ahmet et al. [ALN07].

Depending on the implementation, many similar words are retrieved from the dictionary which have to be compared with the misspelled word. This slows down the system. The naive way is to simply compare every word of a dictionary. Similarity key techniques avoids the frequent computation of similarity measures. Similarity key techniques are based on mapping words to keys in such a way that similarly (spelled) words have an identical or similar key. Hence, computing the key of a misspelled word will point to all similarly spelled words in the dictionary, which constitutes the correction candidates.

One popular approach which uses a similarity key technique is the Soundex algorithm patented by Odell and Rusell in 1918 [OM18]. Words are mapped to keys based on the sound of characters. An improved version of the Soundex algorithm is the Double Metaphone algorithm from Philips [Phi00] which claims to retrieve better similar words for misspellings due to better rules for building the key. Both methods are specialized for correcting human spelling mistakes because they exploit the pronunciation of words. Both methods are mainly focusing on English language. But with some adaptations, other languages can be tackled as well.

Another similarity key technique, which exploits the structure of words, has been proposed by Pollock and Zamora [PZ84]. The similarity key is in this case a condensed form of the original word which only contains the most important characters containing structural information. Depending on the corpus, 56% up to 77% of all errors could be corrected using the described technique in combination with other methods.

A different approach for correcting spelling mistakes is to learn from mistakes first and then use the learned information to propose correction candidates. The requirement for learning is to have error-prone words and their corresponding correct word forms (often called ground-truth material). Neural nets or probabilistic methods are used for learning. One popular approach which uses probability methods has been presented by Brill and Moore [BM00]. This approach is also called an

improved noisy channel model. It first learns the probability of substituting character n-grams. These probabilities are then used for retrieving the most probable correction candidates from a dictionary. Additionally, a language model with context information is used to find the best correction candidate. An error reduction of 74% could be achieved on 10000 sample words of which 80% have been used for learning and 20% for evaluating. With respect to OCR errors, this approach is very interesting if enough ground-truth material is available. Because this thesis lacks ground-truth material, it is not a suitable algorithm.

Real-Word Errors

For correcting real-word errors several techniques have been proposed. A very important technique for correcting real-word errors are n-gram models on word level. n-grams on word level contain sub-sequences of n words of a sentence (or more generally, any text). The n-gram model usually contains frequency information about n-grams which are derived from a large collection.

Mays et al. [MDM91] have proposed a simple trigram (3-gram) approach for correcting real-word errors. The idea using trigrams is to derive the probability of a sentence being correct. Beforehand, trigrams have to be collected and stored with frequency information. By chunking each sentence into trigrams and using the frequency information, a probability which states the likelihood of having a correct sentence can be computed. The sentence is then modified by replacing a word with a spelling variation. The probability of the modified sentence is calculated again. This procedure is repeated for all (likely) spelling variations. The sentence with the highest probability indicates the correct sentence. This approach was later evaluated and compared to other approaches by Wilcox-O'Hearn et al. [WOHB08].

The approach by Mays et al. is very general and allows very many replacements of words. Therefore, this approach tend to be slow. Additionally, using n-gram on word level requires a large amount of data to overcome data-sparseness problems. If a trigram is not present or is under-represented, wrong corrections will be applied. Therefore, Islam and Inkpen [II09] have proposed to use the n-grams contained in the Google Web 1T data containing about one billion trigrams [Goo]. The described approach first tries to find adequate correction candidates by frequency information based on Google's trigrams and then chooses the best candidate by similarity. The similarity is measured by a modification of the longest common subsequence algorithm. The F-Score is slightly better in comparison with Wilcox-O'Hearn et al. [WOHB08]. The F-Score is about 60% for detecting real-word errors and 50% for correcting them.

For overcoming data-sparseness problems of n-grams another direction has been proposed by Fossati and Di Eugenio [FDE07], [FE08]. Their proposal consists of a mixed trigram model which uses information of a POS (part of speech) tagger. A POS tagger determines the grammatical category of each word of a sentence. A mixed trigram consists of words and/ or grammatical categories, e.g. ("the", noun, verb). Therefore, far fewer trigrams are necessary. Each word of a sentence is examined to determine the grammatical order of the sentence based on mixed trigrams. The examined word has several similar words (from a confusion set) which have to be checked. The candidate of the confusion set, which matches best in the sentence according to the mixed trigrams, is then chosen as correction. For setting up the confusion sets all words in a dictionary are examined beforehand. Similar words according to the Levenshtein-Distance, length of word and phonetic distance such as soundex or metaphone are added to the confusion sets. An F-Score of maximum 46% could be achieved. Using a POS tagger on damaged OCR material and historic texts is difficult. Therefore, this thesis will rely on only n-grams for taking the context into account.

For correcting words in context it is also possible to use word sense disambiguation algorithms (WSD). WSD algorithms identify the meaning of a word which has several different senses using the context. Such methods can help detecting words which are used in wrong context and help finding a better word. For example Ginter et al. [GBJS04] present a technique which uses WSD in a biologic domain for correcting real-word errors.

One problem is to decide which correction proposal is the most adequate. Wick et al. [WRLM07] have proposed a method which exploits the topic of a document for enabling more precise correction proposal retrievals. If the topic is known then correction candidates which are semantical related to the topic can be ranked higher. This can be useful with respect to real-word errors as well as non-word errors. Using the topic makes it necessary to have an adequate dictionary. Because many domain specific words are not contained in standard dictionaries further sources have to be used. Strohmaier [Str04] has proposed a method for deriving "tailored" dictionaries from websites which deal with concrete topics. Unfortunately, using websites is not an efficient method for correcting many different articles of an archive. Crawling can become very time consuming.

To achieve better results, several methods mentioned above can be combined to multi-level feature frameworks. One very rich composition has been described by Schaback and Li [SL07]. This approach uses methods such as Soundex on phonetic level, a noisy channel model on character level, a 2-gram model on word level, a POS tagger on syntactic level and co-occurrences on semantic level. Every level returns a feature vector and finally a support vector machine determines the best

feature vector and therefore the best correction proposal. This approach corrects real-word and non-word errors. Using a POS tagger and phonetic algorithms aims at correcting human spelling mistakes. A recall and precision of 90% could be achieved using the top ranked proposal for correction.

There are many more approaches dealing with correcting spelling mistakes. The above mentioned seem to be the most suitable to correct OCR specific errors. More techniques and details about correcting spelling mistakes can be found in Kukich [Kuk92].

2.2 Correcting OCR Based Errors

The correction of OCR errors can be divided into three main areas:

1. Using and improving the scanned image during the OCR process. For example Hong [HDH+95] has proposed improved visual and linguistic techniques.
2. Combining several OCR engines and using a voting system to select the best recognition. For example the work of Strohmaier [Str04] has proposed among other methods such a combination.
3. Using post-correction systems which rely only on the machine processable text after applying an OCR processor.

Because the original images are not available for this thesis, this section focuses on presenting methods related to the third area presented above.

Kukich [Kuk92] has stated, that the number of non-word errors outperforms the number of real-word errors present in documents after applying an OCR processor. Hence, OCR correcting systems should consider this fact. Methods for correcting non-word errors were given in the last section (Section 2.1) and most are adaptable for correcting OCR based errors. Especially, machine learning techniques and approaches which use similarity measures.

Some works which focus on OCR errors are given in the following. These methods can be split up into two categories, according to Perez-Cortes et al. [PCAAL00]:

Deterministic. OCR errors are corrected by utilizing only words contained in a dictionary. If the desired correct word form is not present in the dictionary then no proper correction is possible.

Non-Deterministic. OCR errors can be corrected using the probability of character confusions caused by OCR processors. Words which are probable correct are generated and proposed. These generated words are not necessarily present in a dictionary. That means, such methods do not need a dictionary and produce corrections which are closer to the correct word form or even the exact correct word form.

As described in Section 2.1, noisy channel models are important for correcting non-word errors. Using OCR errors as training material enables the correction of OCR errors. Kolak and Resnik have described such a noisy channel model for OCR errors in [KBR03]. The model is implemented as a finite state machine and needs to be trained. The approach achieves an error reduction up to 80% using the word error rate (WER) as measure. That means, the corrected words are not necessarily correct but closer to the correct word form using the WER. Kolak and Resnik have modified their model for languages with less resources in [KR05], no dictionary is needed but still some training. Because this thesis focuses on OCR archives which have no ground-truth material for training, this approach is not followed.

Tong and Evans [TE96] have proposed a statistical approach using character n-grams, word bigrams and a mechanism for learning confusion probabilities of characters. Character n-grams are used for retrieving correction candidates from a dictionary. Only candidates which have enough common n-grams with the original token are retrieved. The retrieved candidates are then ranked using a weighted Levenshtein-Distance measure. The weights are character confusion probabilities, initially set to an equal probability for all characters. For each sentence the Viterbi algorithm is applied to get the optimal path of tokens respectively correction candidates [FJ73]. This version of the Viterbi algorithm considers the probability of word bigrams (occurrence in pre-processed text) and the ranking of each correction candidate. After correcting a certain amount of documents, the used corrections are used to adapt the confusion probabilities (weights of the Levenshtein-Distance algorithm). The presented method achieves an error reduction of about 60%. The problem using a learning mechanism is the learning of wrong confusion probabilities. If the underlying method for correcting errors is not good enough, the learning strategy can fail. Therefore, a good and reliable correction mechanism has to be developed at first.

Perez-Cortes et al. [PCAAL00] have proposed a stochastic finite state machine (SFSM) which is able to correct OCR errors by finding an optimal path in the SFSM. The path finding is achieved by using an extended Viterbi algorithm. The machine can be used deterministically and non-deterministically in the sense given above. Using the SFSM in a deterministic way is the same as searching adequate

correction candidates from the whole dictionary using the weighted Levenshtein-Distance, but without computing the distance for each word explicitly. This work focuses on handwritten names and has training samples for that task. The samples are used to determine the weights for the SFSM. Furthermore, large dictionaries with names are used. The approach is able to reduce about 95% of misrecognized names using the SFSM deterministically and a large amount of misspelled names for training. This approach is very promising on restricted domains. It is unclear how such an approach behaves on contiguous texts in OCRed archives.

Besides handwritten document, OCR tasks are often applied to historic documents. Historic documents tend to vary in language style, used words or names as well as in used fonts. Hence, a contemporary dictionary is not applicable. Therefore, Hauser [Hau07] has proposed to combine the noisy channel model presented by Brill and Moore [BM00] with several dictionaries in order to cover and correct historic language. Sample pairs for training the noisy channel model are taken from a database which contains spelling variations of historic texts. This approach tackles especially old German texts, nevertheless most aspects are language independent. Preliminary experiments showed a small error reductions ($< 5\%$) on a sample document. Historic texts are very hard to handle, because of their heavy spelling variations.

For using a noisy channel model or weights for the weighted Levenshtein-Distance, training samples have to be available. Because such training samples are reare, Hauser and Schulz [HS07] have proposed an unsupervised training algorithm. This algorithm uses a dictionary and the corpus itself to obtain training samples. For each corpus word, the dictionary retrieves similar words based on the Levenshtein-Distance. Words which lay in a certain Levenshtein-Distance are then used for training. They state to achieve better results using weights derived from the samples than using the standard Levenshtein-Distance, but worse than using hand generated training data.

Myka and Güntzer [MG96] have presented several methods for performing full-text searches in document archives which contain OCR errors. They compare an exact search with several fuzzy string matching techniques. Among these methods, character classifications are used. Characters in strings are replaced by their canonical element and stored in a dictionary which maps to the original word. Such a method is a similarity key technique described earlier. The approach presented in Chapter 3 also uses a character classification, but the generated similarity key is based on different assumptions and a broader classification of characters. Such a classification achieves promising results if the confusions are known.

A semi-automatic OCR correction system has been proposed by Taghva and Stofsky [TS01], called OCRspell. OCRspell is especially designed as a semi-automatic approach. That means besides common techniques for correcting OCR errors, a learning mechanism is used based on the corrections applied by the user. By comparing error-prone token and the manual replacement by the user, dynamic mappings are derived, e.g. *iiiount@in* $\rightarrow$ *mountain* the mappings *iii* $\rightarrow$ *m* and @ $\rightarrow$ *a* are derived. Mappings which occur more often are weighted stronger. With an increasing number of mappings the system slows down, because many combinations have to be tested for generating appropriate correction proposals. Additionally, static mappings can be set manually at start-up. With respect to the similarity key technique presented in Chapter 3 (OCR-Key), static mappings are included implicitly among others by using character classifications. What this means will be pointed out in Chapter 3. The computational effort is far lower using OCR-Key but may not represent every possible rule.

Another work, which uses the archive itself for correcting OCR errors, is presented by Reynaert [Rey08b]. This work focuses on large sets of OCRed Dutch and English historic documents. Reynaert proposes a method which uses high frequent words derived from the corpus which should be cleaned from OCR errors. Instead of correcting errors in contiguous text, this approach gathers all typographical variants for any particular focus word that lie within a pre-defined Levenshtein-Distance. Such a focus word is either a high frequent word from the corpus or a word from a valid dictionary. The core algorithm for finding similar words, first presented by Reynaert [Rey04], is extended in this work for coping with OCR errors. The method achieves good results on the evaluated test set. Up to a Levenshtein-Distance of 2, the author states that the approach can be used fully-automatically because it achieves a high F-Score ($> 95\%$).

This thesis will use the algorithm for finding similar words presented by Reynaert [Rey04]. Instead of using the extended algorithm for tackling OCR errors proposed by Reynaert in [Rey08b], a similarity key approach will be used. This promises an enhanced and faster retrieval of similar words since the extended algorithm requires a higher computational effort. Further details about the algorithm used in this thesis can be found in Section 2.3.2. In contrast to Reynaert [Rey08b] this thesis will apply the algorithm on contiguous text.

2.3 Used Method in Detail

From the presented methods, this thesis uses the Levenshtein-Distance as similarity measure and the Anagram Hash method as core algorithm. Therefore, in this section, these methods are described in more detail.

2.3.1 Levenshtein Distance

To measure the similarity of two strings the *Levenshtein-Distance* (henceforth: LD) can be used [Lev66]. It gives the minimum number of insertions, deletions and substitutions of single characters that are necessary in order to transform a string $x = x_1 \dots x_n$ into another string $y = y_1 \dots y_m$.

To compute the Levenshtein-Distance for two strings $x := x_1 \dots x_n$ and $y := y_1 \dots y_m$ efficiently, the Wagner-Fischer algorithm [WF74] can be used which uses dynamic programming. The algorithm uses a $(n+1) \times (m+1)$ distance matrix D. Let $-$ be the neutral sign (no character), then each matrix entry $D_{i,j}$ can be computed recursively in the following way:

$$
\begin{aligned}
D_{0,0} &= 0 \\
D_{i,0} &= D_{i-1,0} + cost(x_i, -),\ 1 \leq i \leq n \\
D_{0,j} &= D_{0,j-1} + cost(-, y_i),\ 1 \leq j \leq m \\
D_{i,j} &= \min \begin{cases} d_{i-1,j} + \text{cost}(x_i, -) \\ d_{i,j-1} + \text{cost}(-, y_i) \\ d_{i-1,j-1} + \text{cost}(x_i, y_i) \end{cases}
\end{aligned}
$$

For the classic LD the following cost function is used:

$$
\text{cost}(x_i, y_i) := \begin{cases} 0, \text{if } x_i = y_j \\ 1, \text{otherwise} \end{cases} \tag{2.1}
$$

The distance between two strings x and y is the matrix entry $D_{i,j}$. The computational effort is $O(nm)$.

x \ y		c	o	n	u	t	e	e	r
	0	1	2	3	4	5	6	7	8
c	1	0	1	2	3	4	5	6	7
o	2	1	0	1	2	3	4	5	6
m	3	2	1	1	2	3	4	5	6
p	4	3	2	2	2	3	4	5	6
u	5	4	3	3	2	3	4	5	6
t	6	5	4	4	3	2	3	4	5
e	7	6	5	5	4	3	2	3	4
r	8	7	6	6	5	4	3	3	3

Table 2.1: Levenshtein-Distance Matrix: The distance between the words *computer* and *conuteer* is 3.

Example 2.1 (Levenshtein-Distance Computation)
This example illustrates the computation of the LD of the two strings $x = computer$ and $y = conuteer$. A distance matrix D has to be computed which is shown in Table 2.1. The element $D_{8,8} = 3$ of the distance matrix is the actual LD of x and y.

Weighted Levenshtein-Distance

The simple cost function for computing the LD (Formula 2.1) can be adapted for coping with several different error types. Every character substitution as well as insertion and deletion can have a different cost values. With respect to OCR errors complex cost functions are an interesting direction, e.g. substitutions of the character e and c cost less, because they are very similar to an OCR processor. But this direction is not further investigated in this thesis.

Character	ISO Latin-1 code value
`A...Z`	65...90
`a...z`	97...122
␣ (whitespace)	32
’ (apostrophe)	39
- (hyphen)	45

Table 2.2: ISO Latin-1 Character Value Samples

2.3.2 Anagram Hashing

The Anagram Hashing method is used as the core correction mechanism in this thesis. It was first introduced by Reynaert [Rey04] and further explained in [Rey06]. In this section the theoretical main idea from [Rey04] will be explained in a more general fashion than it was done in the original work.

The Anagram Hash method uses mainly a hash table and a “bad” hash function. The hash function is designed to produce a large number in order to identify words which have the same characters in common. Such words are called anagrams. In general an *anagram* is a word which is formed out of another word by only permuting the characters. For example anagrams of the word *tiger* are *tigre*, *regit*, *riget* and all other words which can be formed out of those 5 characters.

To calculate a numerical value for a word, each character c_i of a word $w = c_1 \cdots c_{|w|}$ with a word length of $|w|$ needs a corresponding numerical value. Therefore, the Anagram Hash method uses the ISO Latin-1 code values for each character. Some sample values of the ISO Latin-1 code can be found in Table 2.2. For example the character *a* has the value 97. The function, which maps a character to its corresponding numerical value (ISO Latin-1 code), is denoted as *int*. To obtain the hash value for a given word, each character is raised to a power of n and summed up. Formally, the hash function is defined as follows:

$$\text{hash}(w) := \sum_{i=1}^{|w|} \text{int}(c_i)^n, \quad \text{for } w = c_i \dots c_{|w|} \tag{2.2}$$

This bad hash function should only produce the same value for words which have the same characters in common. Therefore, the power n is set empirically to 5.

Lower powers do not have the desired properties according to Reynaert [Rey04]. Because this hashing function produces the same number for every word with the same characters in common, the name *Anagram Hash* is justified. The calculated hash for a word is called *anagram key*. This name is used in the original work. In this thesis the expression *anagram key* and *anagram hash* are used interchangeably as they reference to the same thing, namely the hash value of the hash function defined in Formula 2.2.

The Anagram Hash Dictionary

In a pre-processing step, a special dictionary has to be set up to work with the Anagram Hash method. Potential words for the dictionary are word lists from common spell checkers or high frequent corpus-derived words. The dictionary has to be implemented as a hash table using the hash function defined in Formula 2.2. Words which have the same anagram key can be chained.

Anagram Hash Alphabet Definitions

To explain how the Anagram Hash method works in general, some definitions are needed. This section formalizes the idea introduced in [Rey04].

The *alphabet* Σ_W^k contains all character n-grams with $1 \leq n \leq k$ of characters and signs present in the word-list W concatenated with a heading and tailing white-space:

$$\Sigma_W^k := \{n\text{-grams of all } sws \mid w \in W, s \text{ whitespace}, 1 \leq n \leq k\} \tag{2.3}$$

The *strict alphabet* Σ_W^{k-} contains all character n-grams with $1 \leq n \leq k$ of characters and signs present in the word-list W without concatenating a heading and tailing white-space:

$$\Sigma_W^{k-} := \{n\text{-grams of all } w \mid w \in W, 1 \leq n \leq k\} \tag{2.4}$$

For calculations on anagram keys two more alphabets are necessary. The *confusion alphabet* Π_D^k contains all anagram keys of character n-grams with $1 \leq n \leq k$ contained in the underlying word-list W:

$$\Pi_W^k := \{\text{hash}(\sigma) \mid \sigma \in \Sigma_W^k\} \tag{2.5}$$

The *focus word alphabet* Γ_w^k contains all anagram keys of character n-grams with $1 \leq n \leq k$ contained in the focus word w:

Note that the focus word alphabet uses the strict alphabet (2.4).

$$\Gamma_w^k := \{\text{hash}(\sigma) \mid \sigma \in \Sigma_{\{w\}}^{k-}\} \tag{2.6}$$

A more practical example will help in understanding the definitions given above.

Example 2.2 (Anagram Hash Definition)
To keep this example simple the dictionary only contains two words:

$$D = \{prime, minister\}$$

Depending on the application an appropriate k has to be chosen for the alphabet Σ_D^k. For this simple example $k = 2$ is a good choice. This leads to the following alphabet Σ_D^2 with 10 unigrams and 15 bigrams on character level:

$$\Sigma_D^2 = \{\text{␣}, p, r, i, m, e, n, s, t, r, \text{␣}p, pr, ri, im, me, e\text{␣}, \text{␣}m, mi, in, ni, is, st, te, er, r\text{␣}\}$$

The confusion alphabet Π_D^2 therefore contains all hashes of characters appearing in the alphabet Σ_D^2. Note that the confusion alphabet contains 2 hashes less than n-grams present in the alphabet, due to the fact that $\text{hash}(im) = \text{hash}(mi)$ and $\text{hash}(in) = \text{hash}(ni)$.

The focus word alphabet may use a different k as the confusion alphabet. For this example k is set to 3. Thus, for the misspelled focus word $w = pime$ the strict alphabet $\Sigma_{\{w\}}^{3-}$ contains 4 unigrams, 3 bigrams and 2 trigrams on character level:

$$\Sigma_{\{w\}}^3 = \{p, i, m, e, pi, im, me, pim, ime\}$$

Hence, the corresponding confusion alphabet $\Gamma_{\{w\}}^3$ contains all hashes of n-grams present in the strict alphabet $\Sigma_{\{w\}}^{3-}$.

Anagram key based correction

The main task of the Anagram Hashing method is to retrieve similarly (spelled) words for a given word. Similar words can be generated by inserting, deleting, substituting or transposing characters [Dam64].

The main characteristic of the anagram hash function is that each character or character sequence of a word $w = c_1 \cdots c_{|w|}$ can be calculated separately:

$$\begin{aligned} \text{hash}(w) &= \sum_{i=1}^{|w|} \text{int}(c_i)^n \\ &= \text{hash}(c_1 \cdots c_{k-1}) + \text{hash}(c_k) + \text{hash}(c_{k+1} \cdots c_{|w|}),\ \forall\, 1 \leq k \leq |w| \\ &= \sum_{i=1}^{|w|} \text{hash}(c_i) \end{aligned} \tag{2.7}$$

This characteristic enables a systematic calculation for retrieving similar words for a word w in the following way:

Transpositions. To retrieve all transpositions, which are present in the dictionary, only a single lookup for the current anagram key of the word is necessary. Example:

$$\text{hash}(\textit{tiger}) = \text{hash}(\textit{tigre}) = \text{hash}(\textit{regit}) = \ldots$$

Deletions. To delete a character or character sequence, each character or character sequence of the focus word alphabet Γ has to be subtracted from the original anagram key. Single deletion example:

$$\text{hash}(\textit{tigger}) - \text{hash}(g) = \text{hash}(\textit{tiger})$$

Insertions. To insert a character or character sequence, each character or character sequence of the confusion alphabet Π has to be added. Single insertion example:

$$\text{hash}(\textit{tigr}) + \text{hash}(e) = \text{hash}(\textit{tiger})$$

Substitutions. To substitute a character or character sequence each character or character sequence of the focus word alphabet Γ has to be subtracted and each character or character sequence of the confusion alphabet Π has to be added. Single substitution example:

$$\text{hash}(tigar) - \text{hash}(a) + \text{hash}(e) = \text{hash}(tiger)$$

Algorithm 2.1 summarizes the calculation of similar words for a given word based on the Anagram Hash method. Practically, the algorithm uses only the substitution formula to handle all four edit operations. For substituting, a value from the confusion alphabet is added and a value from the focus word alphabet is subtracted. For deletions zero is added and a value from the focus word alphabet is subtracted. For insertions a value from the confusion alphabet is added and zero is subtracted. For transpositions nothing needs to be added or subtracted. Retrieved words are checked against their LD to limit the retrieval.

Obviously this algorithm can be extended by a more sophisticated LD handling to speed-up the algorithm and to gain more control over the word retrieval. For example the initial work [Rey04] uses a LD limit based on the word's length.

If the underlying dictionary contains only single words, the confusion alphabet does not need to contain white-space characters. Therefore, the confusion alphabet can use the strict alphabet. If bigrams are included, white-space characters are important to split up words, e.g. *myball* $\rightarrow$ *my ball*. Because this thesis will use bigrams, the alphabet containing white-spaces will be used for the confusion alphabet.

Computational Characteristics

The design of the Anagram Hash method has three interesting characteristics.

Systematic errors. Values of character or character sequences in a word using the anagram hash function are static. For example the character z has the anagram key $122^5 = 27'027'081'632$, whereas the character s has the anagram key $115^5 = 20'113'571'875$. The difference between s and z is $6'913'509'757$. When comparing spelling variations such as *summarize/ summarise* or *advertize/ advertise* the anagram key difference between these words is always the same and equals the difference between s and z. Due to Formula 2.7 this characteristic holds for every systematic error and can enable finding and correcting typical systematic errors.

Algorithm 2.1 Find similar words using the Anagram Hash method

Require: Focus word w
Require: Levenshtein-Distance limit *ldLimit*
Require: Confusion Alphabet Π
Require: Focus Word Alphabet Γ

```
W ← ∅
focusWordHash ← hash(w);
for all γ ∈ Γ do
  for all π ∈ Π do
    simulatedWordHash ← focusWordHash + π − γ;
    W′ ← getWords(simulatedWordHash);
    for all w′ ∈ W′ do
      if LD(w, w′) ≤ ldLimit then
        W ← W ∪ w′;
      end if
    end for
  end for
end for
return W;
```

Natural ranking. In general, Algorithm 2.1 adds and subtracts characters. Every time a new simulated word hash is generated, the hash table is queried and similar words are retrieved. Some similar words are retrieved more often than others. Counting how often a word is retrieved gives a natural ranking. The more often a word is retrieved, the more related it is to the original, potential misspelled word. For further details see Example 2.3.

Retrieval of adequate similar words. In contrast to a naive approach which compares each word in the dictionary with the misspelled word, only adequate words are retrieved from the dictionary respectively the hash table. A query against the hash table is performed in constant time, $O(1)$. However, the size of the confusion alphabet and focus word alphabet determines the amount of queries which are necessary. Large alphabets will therefore perform worse than smaller alphabets.

Especially the natural ranking characteristic is very interesting for this thesis. Therefore, a more sophisticates example is given here.

Example 2.3 (Anagram Hash Method: Natural ranking)
For demonstrating the natural ranking of the Anagram Hash method let *tiiger* be the misspelled focus word. To simplify the computation of similar words only some sample additions and deletions are performed on character level instead of using anagram keys (numbers):

1. Subtract every unigram: *iiger* (-t), *tiger* (-i), *tiger* (-i), *tiier* (-g), *tiigr* (-e), *tiige* (-r). This first step produces the word *tiger* twice and some other words which are not interesting here.
2. Adding unigrams from the confusion alphabet or substituting focus word unigrams does not return any *tiger*.
3. Subtracting or adding bigrams does not lead to the word *tiger*.
4. Substituting unigrams/ bigrams by adding and subtracting unigrams/ bigrams: *tiger* (+i, -ii), *tiger* (+t, -ti), *tiger* (+g, -ig), and so on.

All in all, the algorithm retrieved the correction *tiger* several times for the misspelling *tiiger*. The more n-grams are used, the more often the correctly spelled word *tiger* can be retrieved.

3 OCR-Key: A new OCR error correction proposal

OCR errors can be corrected using various methods from Chapter 2, including correction approaches for human spelling mistakes. But, among OCR related correction approaches, usually a strong requirement for ground-truth material emerges. In general ground-truth material consist of lists which contain error-prone words and their corresponding correct word forms. Furthermore, ground-truth material needs to have a certain coverage to be applicable. If no ground-truth material is available these methods do not work properly.

Having no ground-truth material makes it necessary to either generate own ground-truth material or to follow a method which does not need such material. To find a good method, the idea of exploiting the nature of OCR processors and their generated errors seems promising. Finding such a method eliminates the need for ground-truth material and is still adapted to correct OCR errors.

In this chapter a new algorithm is proposed which uses character classifications, word frequencies and the Levenshtein-Distance for retrieving correction proposals for misspelled words. The method is called OCR-Key.

3.1 Introduction & Motivation for OCR-Key

Many OCR related correction methods require ground-truth material. Although the proposed method in this chapter will not rely on ground-truth material, the generation of ground-truth material and the caused problems are explained firstly. This helps understanding why the presented method in this chapter is interesting. Generating ground-truth material can mainly be done in three ways:

1. **Manual generation**. Manual correction of documents containing OCR errors requires a lot of effort. Additionally, large scaled archives tend to change in paper quality, print quality, fonts and language (spelling, grammar) over time. Such changes can be divided into several periods. For example,

a changing font face can introduce a new period of the archive, e.g. an *s* is detected as an *f*. That means the generated ground-truth material for the earlier period is not longer suitable for the new period. Hence, manual corrections have to be applied for each period of the archive, in order to cope with changing conditions. Manual generation of ground-truth material is very time consuming.

2. **Automatic generation using an OCR processor**. Digital test documents can be generated and printed out in order to scan them again and to apply the same OCR processor used for the archive. By comparing the original document with its scanned and OCR processed version, errors can be identified. The detected errors can be opposed with its error-prone version in lists. Three things have to be considered using this method:

   - The used OCR processor needs to be known and available.
   - The paper and print quality should be similar to the ones used in the archive for achieving adequate results.
   - The used font should be similar to the one used in the archive to cope with font variation problems etc.

3. **Archive derived**. Several methods can be applied to derive ground-truth material from an archive. One method is to use a valid dictionary and gather similarly spelled words from the archive. These words can be used as training material for algorithms which rely on ground-truth material. But the produced samples have a lower quality material than the methods described above (see [HS07]).

These problems are not always resolvable and lead to a search for alternative methods. In literature there are several algorithms which exploit the nature of human spelling mistakes, e.g. soundex [OM18], speedcop [PZ84]. But OCR errors are not caused by humans, but by OCR processors. Therefore, one interesting idea is to exploit the nature of OCR processors and their generated errors. For example a classification of characters [MG96]. Because OCR processor rely on pattern matching techniques, this leads to typical OCR errors. For example the character *e* is often recognized as an *c*. Some OCR errors which typically occur in The Times Archive can be found in Table 3.1. For example the error-prone word *verv* can be corrected to the word *very* by substituting the last *v* with *y*.

In order to correct particular OCR errors, rule based systems can be used (see Taghva and Stofsky [TS01]). Some rules for sample errors can be found in Table 3.1. For example the misspelling *tiine* can be corrected by substituting the characters

Error	Correction	Rule
IBritain	Britain	$I \rightarrow \emptyset$
tiine	time	$in \rightarrow m$
sanie	same	$ni \rightarrow m$
Oueen	Queen	$O \rightarrow Q$
thc	the	$c \rightarrow e$
verv	very	$v \rightarrow y$
vvere	were	$vv \rightarrow w$

Table 3.1: Samples of OCR Errors from The Times Archive

in by *m*. Usually, a rule based system does not know which rules have to be applied in order to retrieve the correct word form, therefore every possible rule has to be applied. Afterwards each generated word has to be verified by looking it up in a dictionary. This leads to heavy computations and slow retrieval.

Example 3.1 (Disadvantage of Correcting OCR Errors Using Rules)
Applying the rule $c \rightarrow e$ on the word *thc* will retrieve the desired word *the*. Here only one position for the rule is applicable. Applying the same rule on the word *circlc* will lead to seven possible rule applications and seven dictionary look-ups to verify the transformation. Using more rules will worsen this behaviour.

To overcome the computational problems, rules can be generalized in such a way that similarity keys for words can be computed. This can be achieved by classifying characters into equivalence classes of similar characters which leads to the category of similarity key techniques.

Similarity key techniques are based on mapping words to keys in such a way that similarly (spelled) words have an identical or similar key. One possible way to enable the retrieval of correction candidates is to implement the underlying dictionary as an hash table which uses the similarity key computation as hash function.

In order to retrieve similar words, the similarity key of the given word has to be computed. All words in the dictionary corresponding to the computed similarity key are the similar words.

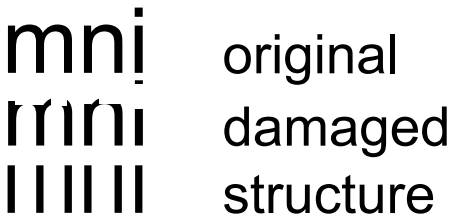

Figure 3.1: Structure of Characters Recognized by OCR Processor

3.2 Requirements for OCR-Key

Because the algorithm belongs to the category of similarity key techniques, a similarity key has to be computed. The basic idea and requirement for computing the similarity key are given in this section. The concrete algorithm can be found in Section 3.3.1.

Computing a similarity key for a word, is based on the idea of classifying characters by their similarity with respect to the underlying OCR processor. That means, characters which are likely to be confused during the OCR process fall into the same equivalence class. Due to different OCR processors and used fonts, these equivalence classes vary. Based on the errors of The Times Archive, the following equivalence classes are used in this thesis: i, o, c, v, s, z, a. An overview over all equivalence classes and their contained characters is given in Table 3.2. For example the character n belongs to the equivalence class i and has the cardinality 2. The cardinality value is explained in the following. Classifying characters such as B or D as i is motivated by the observation of having character insertions in front of the characters, e.g. *IB* or *ID*.

A closer look at common OCR errors helps finding an interesting characteristic. For example the shape of the character m is similar to iii (missing only the connections between each i). More formally, each character of the class i can be characterized by the amount of vertical dashes it includes. This characteristic can be seen in Figure 3.1. The original text got damaged over the years but the structure (the dashes) is still recognizable. For example the printed n misses only a small part of the connection between the dashes. An OCR processor has difficulties to recognize such damaged characters properly. The processor tend to recognize the damaged n as ii. This means each character belonging to the equivalence class i needs to have a certain cardinality which represents the structure of the character (here: the amount of dashes).

In general the cardinality states the number of *atomic parts* each character can be decomposed. For the equivalence class i the atomic part is a horizontal dash

Class	Member	Cardinality
i	f, i, j, k, l, r, t, B, D, E, F, I, J, K, L, P, R, T, 1, !	1
i	n, h, u, H, N, U	2
i	m, M	3
o	a, b, d, g, o, p, q, O, Q, 6, 9, 0	1
c	e, c, C, G	1
v	v, x, y, V, Y, X	1
v	w, W	2
s	s, S, 5	1
z	z, Z	1
a	A	1

Table 3.2: Character Classification by Similarity

as shown in Figure 3.1. The atomic part of the class v is a v. This is motivated by the fact that a w can be decomposed into vv. The cardinality categorization is only important for the equivalence classes i and v. All other equivalence classes do not need a cardinality, but for unification reasons they get the cardinality 1.

Example 3.2 (Equivalence Class and Cardinality)
The character m has the cardinality 3. Valid decompositions are all character sequences which are from of the same equivalence class and whose cardinality sum equals 3, e.g. in, nI or iii have the same cardinality and are therefore valid decompositions of m. For the word *tiine* this means that the characters in can be exchanged by m to get the correction *time*.

The character w lays in the same equivalence class as v and has the same cardinality as vv. Therefore, the characters vv of the word *vvhich* can be exchanged by the character w in order to achieve the correction *which*.

3.3 The OCR-Key Algorithm

After introducing the idea of similarity key techniques and the used classification of characters, this section will present a formal and detailed description of the algorithm to compute the similarity key, called *OCR-Key*. In order to find similar words based on OCR-Key a special dictionary is necessary. The generation of an adequate dictionary for the OCR-Key method is described in Section 3.3.2. Section 3.3.3 will present the main algorithm for retrieving similar words based on OCR-Key which will be used in Chapter 4, where the correction strategy is presented. The proposed method retrieves several similar words which are all possible correction candidates. In order to find the best matching correction candidate, a ranking is necessary. Therefore, a simple ranking is presented in Section 3.3.4.

The proposed similarity key (OCR-Key) and the corresponding algorithms presented in this section are new to the author's knowledge. A similar approach which uses character classifications for performing a full-text search on an OCR archive has been described by Myka and Güntzer [MG96].

3.3.1 Computing OCR-Key

The similarity key for a word w is computed as shown in Algorithm A.1 (page 92). It is the basis for finding similar words using the algorithm presented in the next section. Informally, the algorithm computes a similarity key in the following way:

1. Start at the first character of the word.
2. Sum up the cardinality of the current character and go to the next character.
3. If the current character belongs to the same equivalence class as the last one, go to step 2, otherwise continue.
4. Append the representative of the previous character to the output string.
5. Append the computed cardinality sum to the output string and reset the sum to 0.
6. If all characters of the word have been processed, stop and return the output string. Otherwise continue at step 2.

Note that characters which are not included in any equivalence class will be omitted. This can be interesting for special signs like apostrophes (') or hyphens (-) because they are often not of interest.

In the following the function "key" will be used to denote a similarity key computation based on the OCR-Key.

Example 3.3 (Computation of OCR-Keys)
This example illustrates how similarity keys based on the OCR-Key algorithm are computed. Given the word *saturday*:

$$saturday \rightarrow \underbrace{s}_{s1}\underbrace{a}_{o1}\underbrace{tur}_{i4}\underbrace{da}_{o2}\underbrace{v}_{v1} \rightarrow s1o1i4o2v1$$

The first character belongs to the equivalence class s. The following characters belong to another class. The cardinality of s is 1. Therefore, $s1$ has to be appended to the output string. The next three characters *tur* belong to the same equivalence class with representative i. Their cardinalities sum up to 4. Therefore, $i4$ has to be appended to the output string. The following two characters *da* belong to the equivalence class o and their cardinalities sum up to 2. Therefore, $o2$ has to be appended to the output string. The last character v produces $v1$. Hence, the final OCR-Key for *saturday* is $s1o1i4o2v1$.

Note that the format of the OCR-Key is not regular in every aspect. Although, equivalence class characters and cardinality digits alternate, the character does always have the length of one, whereas the digit can have the length of one or more. Example:

$$\text{key}(minimize) = i11z1c1$$

3.3.2 Generating a Dictionary for OCR-Key

In order to retrieve similar spelled words from the dictionary, two steps are necessary, assuming that the dictionary is implemented as hash table.

1. The dictionary has to be initialized. This can be done by using word-lists of common spell checkers or corpus derived word-lists. The hash function determines where words are stored in the table. Hash collision can be resolved by chaining a new word to the last inserted word with the same hash value. That means, each hash value points to a linked list. A collision occurs, if words have the same hash value in common.

2. In order to retrieve similar words, the hash value of the word has to be computed. Then the dictionary is queried and retrieves the linked list which corresponds to the calculated hash value. This list contains the similar (spelled) words.

Two characteristics of the OCR-Key method have to be considered when generating the dictionary:

1. Words which contain capital letters, or can be spelled with capital letters, e.g. at the start of a sentence, need a special handling. All possible (valid) spelling variations with respect to lower-case and upper-case characters need to be added to the dictionary. This is necessary because the following characters are not in the same equivalence class as their lower-case version: A, B, D, E, G. Example:

$$\text{key}(\textit{Britain}) \neq \text{key}(\textit{britain})$$

2. No OCR errors should be included. If OCR errors are included, the likelihood to retrieve misspelled words increases, because the map is designed to retrieve similarly spelled words with respect to OCR errors.

3.3.3 Finding Similar Words Using OCR-Key

The main task of OCR-Key is to enable the retrieval of similar words. Before similar words can be retrieved, a dictionary based on OCR-Key has to be set up. Setting up an adequate dictionary is described in the last section. The concrete retrieval of similar words using such a dictionary is described in the following.

Algorithm 3.1 (page 31) shows one way to implement the retrieval of similar words using OCR-Key as similarity key. Initially the algorithm uses the OCR-Key of the given word and retrieves all words which have the same OCR-Key in the dictionary. To retrieve more similar words, the cardinality of each word is increased and decreased by a given number δ. But the cardinality is always greater than 0. For each cardinality modification the dictionary is queried again to retrieve more words. In order to extend the control over retrieved words, the Levenshtein-Distance is used additionally to filter out words. This can be very important because the similarity key technique may retrieve words which are structural similar, with respect to the OCR processor, but still no adequate similar word. For example the words *minimum* and *untruthful* have the same OCR-Key *i15* but are definitely not very similar.

Algorithm 3.1 Find similar words using the OCR-Key method

Require: Word w
Require: $\delta \geq 0$
Require: Levensthein-Distance (LD) ldLimit

```
// initialize
S ← null
W ← null

// key = e_1 c_1 ··· e_n c_n, e equ. class, c cardinality, n number of equ. classes
key ← ocrKey(w)

// query similar words w.r.t cardinalities
for all i = 1 to n do
  for k = 1 to n do
    c̃_k ← c_k
  end for
  for j = −δ to δ do
    c̃_i ← max(c_i + j, 1)
    newKey ← e_1 c̃_1 ··· e_n c̃_n
    S ← S ∪ getWords(newKey)
  end for
end for

// filter out words w.r.t. Levenshtein-Distance
for all s ∈ S do
  if LD(w, s) ≤ ldLimit then
    W ← W ∪ {s}
  end if
end for
return W
```

In order to understand the mechanism of Algorithm 3.1, the OCR-Key method can be compared to the classic assumption that misspelled words have character insertions, deletions, substitutions and/ or transpositions [Dam64]. The OCR-Key method handles these four assumptions in the following way:

Transposition are handled by the fact that equivalent characters are grouped. This means transpositions can only appear inside a group of similar characters.

$$\text{key}(time) = \text{key}(tmie)$$

Substitution are handled by the equivalence classes which allow characters to be used interchangeably if they belong to the same equivalence class and satisfy the given cardinality. This constraint limits the substitutions in a positive way, because it only allows to substitute characters which are similar to the OCR processor.

$$\text{key}(times) = \text{key}(timcs) \neq \text{key}(timas)$$

Note that the cardinality limits the substitution of characters. For example the character n with a cardinality of 2 can not be replaced by the character m which has the cardinality of 3. But substituting such characters is necessary. Therefore, the cardinality has to be increased respectively decreased as shown in Algorithm 3.1.

Insertion & Deletions can only occur among characters of the same equivalence class. Insertions and deletions can only be handled if the cardinality is increased respectively decreased. To insert a character the cardinality has to be increased, whereas the cardinality has to be decreased when deleting a character.

Example 3.4 (Retrieving Similar Words Using OCR-Key)
This example illustrates how the retrieval of similar words works using algorithm 3.1. Three kind of situations can occur in order to find a correct similar word:

1. No cardinality changes.

$$\text{key}(tinie) = i5c1 = \text{key}(time)$$

2. Cardinalities have to be increased.

$$\text{key}(tine) = i4c1 \rightarrow i5c1 = \text{key}(time)$$

3. Cardinalities have to be decreased.

$$\text{key}(tiime) = i6c1 \rightarrow i5c1 = \text{key}(time)$$

All examples are taken from The Times Archive, year 1985. The frequencies of the words are: `time (32.758), tine (233), tinie (54), tiime (15)`.

3.3.4 Ranking Retrieved Similar Words

In most cases Algorithm 3.1 retrieves several similar words. In order to find the best matching candidate for the original, potential misspelled word, a ranking mechanism has to be applied. Three indicators can be used for ranking retrieved correction proposals of the OCR-Key method:

Frequency. By assuming that frequent words are more likely to be correct than less frequent words, the frequency can be used for ranking retrieved correction proposals. Note that this only works if the correct word is the most, or one of the most frequent words. The frequency for a word w is denoted as $f(w)$.

Levenshtein-Distance. The Levenshtein-Distance (LD) can be used to determine the similarity between two words with respect to character insertions, deletions and substitutions. Hence, a small LD indicates that two words are more similar than a larger distance. The more similar correction proposal and original word are, the more likely it is that the correction proposal is an adequate proposal.

Cardinality difference. The algorithm increases and decreases cardinalities of the OCR-Key. The cardinality difference between the initial computed OCR-Key for the original word and the modified one for the retrieved word can indicate how similar both words are. The assumption is that a lower difference indicates a greater similarity. The cardinality difference is denoted as c_c.

The following formula is used in order to rank the retrieved correction proposals c for a given word w and its length $|w|$:

$$\text{score}_{ocr}(c, w) = \ln(f(c)) \cdot (|w| - \text{LD}(w, c) - c_c)$$

The correction proposal with the highest score is assumed to be the most appropriate correction proposal.

3.4 Examples of The Times Archive

The randomly chosen word *Saturday* serves as example in the following to show which word forms can be tackled by the OCR-Key approach and where the approach fails. The dictionary was pre-processed on the data of The Times Archive for the whole year 1985. The algorithm is able to retrieve 69 distinct word forms besides the original form. All 70 retrieved words have an total occurrence count of 8102. Word forms with a frequency higher than 4 are shown here:

```
Saturday (7405), Saturdav (339), SaturdaY (108),
saturday (12), Salurday (21), Saturdy (22), Satuday
(17), Saurday (12), Satirday (12), Saiurday (7),
Saturdayv (9), Satutday (6), Satuiday (6), Satrday (14),
Satturday (8), Satuirday (7), Sattrday (7), Saturay (7),
Saturdax (5), Saturdaw (6), Saiturday (5)
```

After examining the retrieved word forms it is very likely, that they are all misspellings of the word *Saturday*. Thus, $8102 - 7405 - 12 = 685$ words could be theoretically tackled by the OCR-Key method. Some more word forms can be found in the dictionary, which can not be handled by the OCR-Key method:

```
SaturdaI (1), SaturdaN (4), SaturdaA (1), SaturdaT (1),
Saturdai (7), Saturdaj (1), Saturdak (1), Saturdac (1),
Saturdag (3)
```

These words can not be handled by the system because their last character does not belong to the equivalence class v which is necessary in order to retrieve the correct word *saturday*.

3.5 Discussion of OCR-Key

In this chapter a new method for correcting OCR errors based on the shape of characters was presented. Character classifications are used based on the propable confusion of the underlying OCR processor. The Levenshtein-Distance and word frequencies are used to rank correction proposals retrieved by the OCR-Key method.

Because the OCR-Key equivalence classes are inspired by the errors found in The Times Archive, these classes have to be adapted for different OCR processors.

The used character classification of Table 3.2 may be incomplete due to lack of knowledge. The more errors are known, the better equivalence classes can be chosen.

Using equivalence classes can only cover the systematic errors. As described above, the word *Saturday* has several error-prone word variants in the archive, which can not be covered by equivalence class approaches. These errors are absolutely unsystematic. That means, the OCR-Key approach can only be used to correct the most common systematic errors. In order to correct errors in a more general way, it is necessary to apply an additional approach.

The OCR-Key approach is language independent and relies only on the dictionary generated in a pre-processing step (see Section 3.3.2) and the used classification of characters.

Possible Improvements

Algorithm 3.1 is only able to find similar words by increasing or decreasing *single* equivalence classes. In order to enhance the algorithm, all possible cardinality variations could be computed. Of course these computations come at a cost.

Another possible improvement is to add a better control over increasing and decreasing cardinalities. For example, an i is probably added or deleted more often than a g which belongs to the equivalence class o. Therefore, Algorithm 3.1 should restrict the increasing and decreasing of cardinalities for the equivalence class o because insertions and deletions of characters of that equivalence class is less likely. But for adjusting these values, more insights of occurring errors are necessary.

The presented similarity computation using the OCR-Key method uses only a few equivalence classes. Especially, the handling of upper-case and lower-case characters is not optimal. This leads to a rough structure of words and may retrieve similar words which may not be optimal. Therefore, more equivalence classes could be used in order to represent a better structure of words. For example the character b is very similar to the character h.

4 Correction Strategy

This chapter presents the necessary steps for correcting OCR errors fully-automatically. This is done by using mainly the archive itself. The used approaches and the co-operation between them are discussed in detail.

The presented correction strategy can be seen as a processing pipeline. The processing pipeline is divided into 3 processing steps: pre-processing, main-processing and post-processing. Figure 4.1 gives an overview over these steps. In the pre-processing step all necessary data-structures are generated using mainly the archive itself. These data-structures are needed by correction algorithms in the subsequent processing steps for finding adequate correction proposals. In the main-processing step, correction proposals are computed and saved for the post-processing in XML files. The main-processing step uses the Anagram Hash and OCR-Key method for retrieving correction proposals. Correction proposals are generated for each document in the archive. In the post-processing step, a new version of the original document is generated. In this document misspelled words are exchanged by its best correction proposal which fulfil certain criteria. The whole correction strategy is implemented in Java.

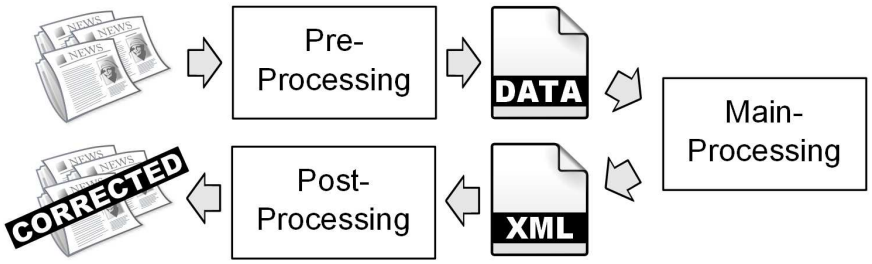

Figure 4.1: Correction Strategy: Overview of the necessary steps for correcting OCR errors contained in an archive.

To correct documents of The Times Archive, the correction strategy is applied for each year of the archive. That means, correcting a particular year of the archive requires the pre-processing of all documents for that particular year.

4.1 Pre-Processing

The first step towards finding and correcting OCR errors is to generate all necessary data-structures. The later described methods use these data-structures for retrieving adequate correction proposals. In contrast to classic dictionary-based systems, which use word-lists of contemporary language without frequency information, this pre-processing step derives "tailored" dictionaries based on the frequency of words present in the archive. Hence, such tailored dictionaries contain rare words, names or domain specific words which classic word-lists usually not contain. Assuming that frequent words are more likely to be correct, such tailored dictionaries enable detecting and correcting proper and person names as well as domain specific words. Note that an adequate correction is only possible if the correct word is frequent enough.

Figure 4.2 summarizes the generation of data-structures in the pre-processing step. In general tokens are collected, counted and stored in occurrence maps (henceforth: OMs). OMs map tokens to its occurrence count. Based on these OMs, all necessary data-structures for the correction algorithms in the main-processing step are generated. The details can be found in the following sections.

4.1.1 Generating Occurrence Maps

The first step of the pre-processing chain is to collect and count as many single tokens and token pairs as possible from the archive. Single tokens are unigrams on word level and token pairs are bigrams on word level. The tokens are stored in occurrence maps (OMs).

Generating the Mixed-Case Occurrence Map

In general the mixed-case OM contains single tokens and their corresponding token count. In order to generate the mixed-case OM all occurring tokens of the archive need to be collected and counted. But simply counting all occurring white-space separated tokens is not enough, first punctuation marks and other unwanted characters have to be handled. Furthermore, hyphenation errors can be reduced using a heuristic. Therefore, these three important improvements are used to enhance the counting results over using white-space separated tokens only:

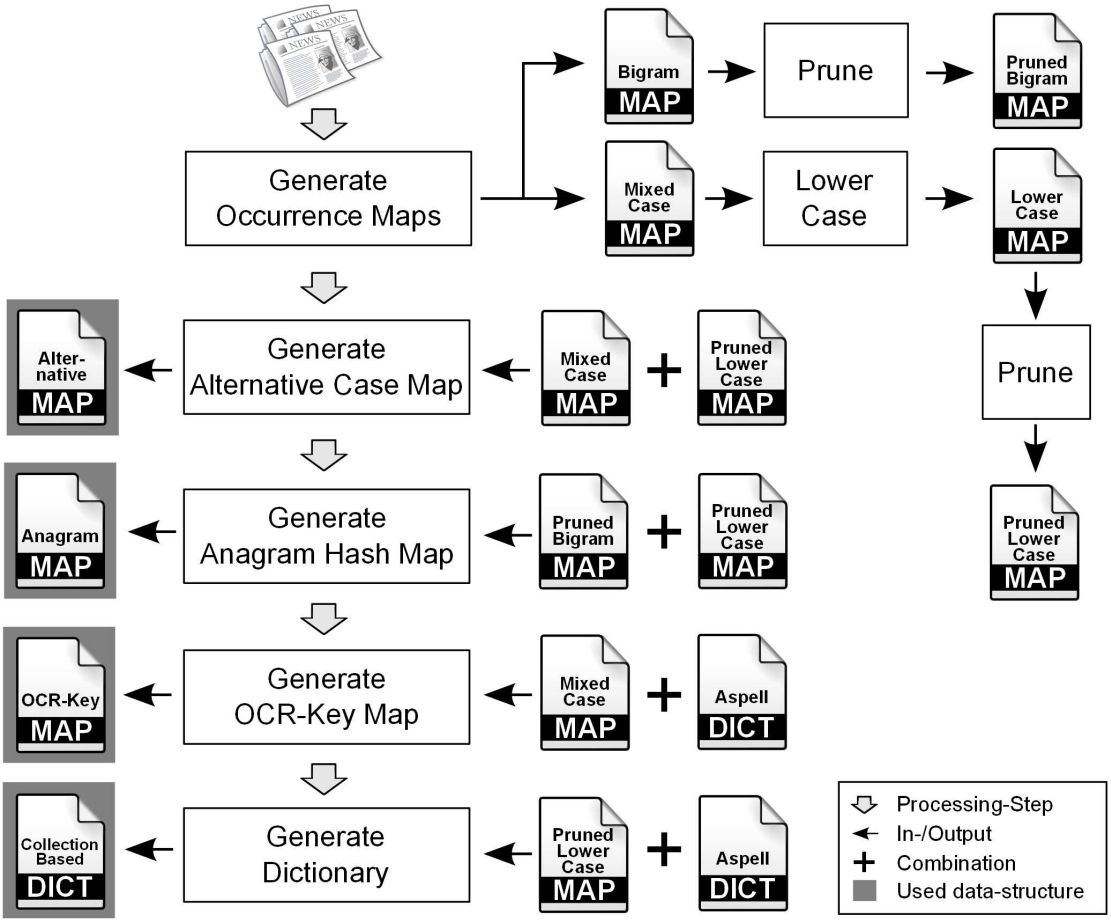

Figure 4.2: Correction Strategy: Pre-Processing. Overview of generated data-structures. Grey highlighted data-structures are used in the main-processing step.

Merging hyphenation errors. Hyphenation errors influence the count in a negative way. Because this type of error can be handled with a heuristic, potential hyphenation errors are remedied by merging split terms. The details of the used heuristic can be found in Section 4.2.1.

Enhanced tokenization. OCR processors tend to omit white-spaces. That means two words can be accidentally merged by the OCR processor if the white-space is not detected properly, e.g. *thisis* instead of *this is*. Such a fault can not be remedied in the pre-processing step where only words are collected. But omitted white-spaces around punctuation marks can be handled by splitting tokens at punctuation marks. This can lead to better token counts, although not all punctuation marks are really word separators. For example the word *read:er* has a false punctuation mark.

Preliminary experiments showed that it is worth splitting tokens on the following punctuation marks: `,.;:()\"&[]?!´‘{}/+#=<>%`. Note that there is no splitting at hyphens (`-`) and normal apostrophes (`'`) because these information can be important, e.g. *middle-aged* instead of *middle aged* or *cat's* instead of *cats*.

Cleaning. After tokenization, each token is cleaned by removing heading and tailing non alphabetic characters. For example the token *12bi1rds.%* becomes *bi1rds* after cleaning. Removing these characters enables focusing on only words without heading and tailing numbers and punctuation marks. For example dirty tokens such as *cats.*, *cats;* and *5cats* do all have the same word in common and therefore increase the occurrence count of *cats*.

To give a short outlook, the mixed-case OM serves as basis for the lower-case OM, alternative-case map, OCR-Key map and anagram-hash map.

Generating the Lower-Case Occurrence Map

It is favourable to lower-case the mixed-case OM to reduce the contained upper- and lower-case spelling variations. The resulting lower-case OM contains therefore only lower-cased token forms. Each token maps to the summed occurrence count, e.g. the occurrence count of *time* contains the count of *Time*, *tIme*, *TIME*, etc. The reduction has several advantages over using the mixed-case OM:

Smaller search space. Using only lower-cased characters simplifies the computation of similar words. Using only lower-cased characters enables the to compute on 26 characters less ($A - Z$). This decreases the computational

effort and the amount of retrieved tokens. The correct word form with respect to upper- and lower-case characters can be calculated later based on the alternative-case OM (see Section 4.1.2).

Summed occurrence counts. Summing up the occurrence counts of the same tokens with different spellings (with respect to upper- and lower-case characters only) can increases the occurrence count. This can enable a better separation of potential OCR errors. For example the occurrence counts of the tokens *time*, *Time*, *tIme* are summed up and only increase the count of the lower-cased word form *time*.

Generating the Pruned Lower-Case Occurrence Map

Following the assumption, that high frequent tokens are more likely to be spelled correctly, infrequent tokens need to be removed form the lower-case OM. Removing infrequent tokens is called *pruning* in this thesis. The second reason for pruning is to restrict the search space. Fewer words contained in the dictionary prevent the retrieval of these words and reduce the computational effort.

Removing infrequent tokens can be done in several ways. This thesis uses a simple frequency based pruning which removes all tokens which occur less than 8 times from the lower-case OM. Preliminary experiments showed that 8 might be a good number. Finding a good pruning strategy is very difficult which the following example illustrates.

Example 4.1 (Frequency Pruning Problematic)
This example illustrates the problematic of finding a proper way to prune OMs. Therefore, OCR errors of the token *which* are examined. The samples are extracted from the lower-case OM of the year 1985 of The Times Archive. Only some of the samples are listed here with its corresponding occurrence count:

```
which (101,733), Which (945), wvhich (1,016), vhich
(449), whichi (316), whieh (63), whIch (60), wlhich
(234)
```

Obviously, the frequencies of misspelled words are very high. In contrast, many rare words do not have such high occurrence counts, e.g. names. Hence, it is hard to find a proper pruning strategy.

Because the frequency is later used to rank similar retrieved words, the pruning problematic is a minor problem. However, a better pruning strategy is desirable and should be considered in future work.

Generating the Bigram Occurrence Map

The bigram OM contains word pairs (bigrams) and their corresponding occurrence count. For generating the bigram OM a similar procedure as for the lower-case OM has to be applied. The aim is to collect as many as possible bigrams.

The first step is equal to the step explained for generating the lower-case OM. The text is tokenized, possible hyphenation errors are removed by merging tokens and the tokens are cleaned on the sides. All bigrams which are added to the bigram OM are lower-cased beforehand to increase the occurrence count of the same token without respecting case-sensitive spelling variations. Case-sensitive spelling variations are not interesting for this map, but a high occurrence count. Two more things have to be considered:

1. Occurrence maps are the basis for generating the anagram-hash map. Therefore, it is not important in which order bigrams are stored, because they have an equal anagram key (see Section 2.3.2). For example the word pairs *my cat* and *cat my* have the same anagram key and are therefore the same here. Hence, only the first occurrence of a word pair needs to be stored in the map. If a twisted word pair occurs, the occurrence count of the first stored word pair is increased.

2. This bigram approach does not consider punctuation marks. That means word pairs are formed regardless of sentence delimiting punctuation marks such as points, commas, semicolons or double points. Usually, word pairs which are formed over sentence boundaries are not likely to be contextual. But with respect to OCR errors it makes sense to not follow this assumption here, because punctuation marks may not be detected reliably and errors in words may decrease the bigram count. It makes more sense to collect as many bigrams as possible to overcome data sparseness problems, when filtering out infrequent bigrams.

   Furthermore, bigrams are used to split up tokens which were wrongly merged by the OCR processor. Because no other technique is used in this thesis to split up wrongly merged tokens, is is absolutely necessary to have as many bigrams as possible.

Generating the Pruned Bigram Occurrence Map

Following the assumption, that frequent bigrams are likely to have less errors, the bigram OM has to be pruned. The same pruning problematic as for the lower-case OM emerges. However, a simple pruning strategy is applied which removes all bigrams which occur less than 3 times. Additionally, all bigrams, which contain a word that is only a single character, are removed, except the character *a*, e.g. *cat s* is removed and *a cat* is kept. This is necessary to overcome unwanted word splitting problems in the later process, such as:

- *IBritain* $\rightarrow$ *I Britain* or *Britainn* $\rightarrow$ *Britain n*

Providing no undesired bigrams, which allow unwanted splits, will prevent their generation in the main-processing step. Additionally, it decreases the search space which increases the speed of the algorithm. In future work a more sophisticated pruning strategy should be developed for the bigram OM.

4.1.2 Generating the Alternative-Case Map

This correction strategy focuses on computing and retrieving lower-cased correction proposals. Hence, *all* generated correction proposals are spelled with lower-cased characters. It is desirable to compute an adequate spelling variation with respect to lower-case and upper-case characters at the end of the process. For example correction proposals which contain proper or person names should be spelled with an initial capital letter. For computing these spelling variations which only differ in lower-case and upper-case characters, the *alternative-case map* is used for. The alternative-case map contains all words of the pruned lower-case OM and maps each word to the three most frequent spelling variations present in the mixed-case OM with corresponding occurrence counts.

The motivation for using the three most frequent word variants is derived from observations. These three different types of spelling variations are likely to occur:

1. All characters are lower-cased, e.g. *application*.
2. The first character is capitalized, e.g. *Application*. This mainly occurs at sentence beginnings or if the word is a name.
3. All characters are capitalized, e.g. *APPLICATION*. This mainly occurs if a word is emphasized or an abbreviation.

Note, that the three assumed word forms are not always present in the archive. Therefore, the map may contain less than three variants or some corrupt versions, e.g. *fullY* instead of *fully*. Usually corrupt word forms are less frequent than correct word forms. Hence, corrupt word forms should not affect the retrieval of correct variants.

4.1.3 Generating the Anagram-Hash Map

The anagram-hash map is the most important data-structure for computing correction proposals. It is implemented as hash map and uses the anagram key computation as hash function (see Section 2.3.2). It maps anagram keys to tokens and their corresponding occurrence count.

The anagram-hash map is generated by using the pruned lower-cased OM and pruned bigram OM. Every unigram and bigram is stored with its corresponding occurrence count in the anagram-hash map. To be more specific, each anagram key maps to a list containing all tokens with the same anagram key.

4.1.4 Generating the OCR-Key Map

The OCR-Key map is the data-structure for retrieving correction proposals for misspelled words based on the OCR-Key method. As described in Chapter 3 this map is a hash map and uses the OCR-Key computation as hash function.

The Times Archive has a font variation problem, especially in the earlier years (see Section 5.1). The character *s* is sometimes recognized as an *f*. In order to tackle these font variation problems, the equivalence classes of Table 3.2 (page 27) were modified. Instead of classifying an *f* as an *i*, an *f* is classified as an *s* with cardinality 1.

For generating the OCR-Key map two requirements have to be considered in more detail:

1. The map needs to contain upper-case and lower-case spelling variations of the same word.
2. No OCR errors should be included.

The first requirement can be fulfilled using the mixed-case OM to generate the OCR-Key map. It contains lower-case and upper-case words forms. Each token of the map is stored to its corresponding OCR-Key in lower-cased form with its

corresponding occurrence count. Lower-casing the token has the advantage to group words with the same key, which only differ in lower-case and upper-case characters. For example *Britain*, *BrItain*, *BrItaIn* have all the same OCR-Key. Their occurrence count can be summed up and stored as one single token in the map. Note that the word *britain* has a different OCR-Key and will be stored separately from the three words given above with a different occurrence count.

Fulfilling the second requirement, OCR errors have to be removed from the mixed-case OM. This has to be done before words are inserted into the OCR-Key map. Unfortunately, pruning strategies are a trade-of between removing as many OCR errors as possible and keeping words which are infrequent but correct, such as names and domain specific words. This means a pruned map is likely to contain too many OCR errors to generate the OCR-Key map directly. To overcome this problem, the dictionary of Aspell is used additionally. Only tokens of the mixed-case OM are added to the OCR-Key map if they appear in the Aspell dictionary.

Using a dictionary to decide which words can be added to the map limits the amount of words which can be corrected, but is a good way to control the generation until a better pruning strategy is available.

4.1.5 Generating the Collection-Based Dictionary

Checking and trying to correct *every* single word will slow down the system dramatically. Therefore, a dictionary check can be applied before generating correction proposals. This can filter out correct words which should not be handled by the system.

The collection-based dictionary for performing a dictionary check is build using the lower-case OM and the Aspell dictionary in the following way: All words which are contained in both are added to the dictionary.

If no external dictionary should be used, only very high frequent words of the lower-case OM could be added to the dictionary. Unfortunately, the lower-case OM contains many high frequent words with OCR errors which make it difficult to distinguish between misspelled and correctly spelled words.

The increased speed comes at a cost. The more words a dictionary contains, the more likely it is that wrongly spelled words are omitted. For example the misspelled word *thc* instead of *the* is detected as correct and will not be corrected using the Aspell dictionary since *thc* is a correct term. Because this error occurs frequently, the word *thc* is removed from the dictionary in this thesis.

Example 4.2 (Collection-Based Dictionary for The Times Archive in 1985)
This example illustrates the impact of using the Aspell dictionary to filter out unwanted words. Initially the lower-case OM contains roughly 2 million tokens. After removing all tokens which occur less than 8 times the map still contains roughly 110k tokens. Generating the dictionary using the lower-case OM and the Aspell dictionary with about 138k tokens results in a dictionary with about 79k tokens. That means 110k $-$79k $=$ 31k words are not included in the collection-based dictionary, although they are told to be frequent. It is very likely, that most of these 31k words are OCR errors. But on the other hand names and domain-specific words may also be removed by Aspell.

4.2 Main Processing: Generating Corrections

This section describes the spell checking and generation of correction proposals. Therefore, several steps are necessary. An overview over this processing step is depicted in Figure 4.3 and described in the following. Each step is explained in more detail later:

1. All white-space separated tokens of a document are read in and stored into a token list.
2. A heuristic is applied to detect and merge hyphenation errors.
3. A general cleaning of each token is applied to get rid of punctuation marks and other dirt characters. Therefore, heading and tailing "dirt" characters of each token are removed, i.e. all characters which are not $a-z$ and $A-Z$ are removed from the sides of each token.
4. Two rules for filtering out tokens which are too short or contained in the collection-based dictionary are applied.
5. Tokens which are not filtered out are now tackled by the Anagram Hash method. Correction proposals are stored in a separate list for each token.
6. Each token is then passed to the OCR-Key method, which generates and stores its correction proposals in a second proposal list for each token.
7. Both generated proposal lists are merged to one final list. The merging has to be applied with respect to the ranking of correction proposals.
8. Three rules are applied:

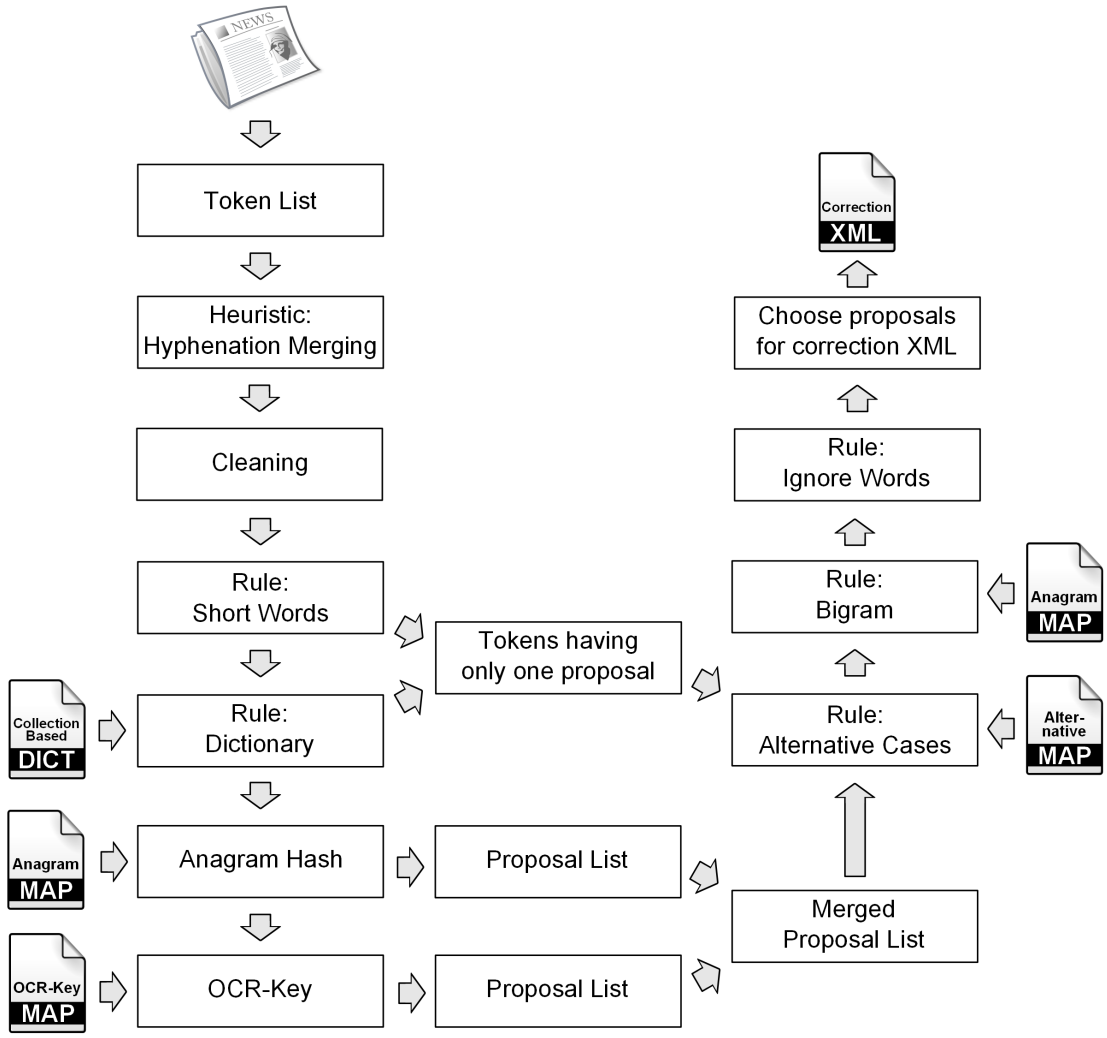

Figure 4.3: Correction-Strategy: Main-Processing. Necessary correction steps for generating correction proposals for a single document.

- The bigram rule, which allows to retrieve a better sorted proposal list based on co-occurring words.
- The alternative case rule, which computes the best matching lower- and upper-case variation of a word.
- The ignore word rule, which filters out word for which no proposals should be proposed.

9. The five best correction proposals for relevant misspelled words are stored into an XML file for each processed document. The correction proposals are used in the post-processing step for exchanging misspelled words.

To rank the computed correction proposals a simple scoring system is used, which will be explained at the end of this section.

4.2.1 Heuristic: Merging Hyphenation Errors

Scanned OCR documents often contain structural information about positions of words. Such information can help correcting hyphenation errors (see 1.3).

This approach uses a heuristic which requires that the following 4 conditions are fulfilled in order to merge two words:

1. A line-break or column-break has to be between both words.
2. The token before the break has to end with an hyphen after removing all tailing characters which are not alphabetic. For example "1.depart-," becomes "1.depart-" and is appropriated, whereas "1.depart-d," becomes "1.depart-d" and is not appropriated because it ends with a character instead of a hyphen.
3. The token before the break occurs, has to have a lower-case alphabetic character in front of the hyphen.
4. The first character of the word after the break has to be a lower-case character after removing all heading characters which are not alphabetic. For example ".ment.?" becomes "ment.?" and is appropriated, whereas ".Ment.?" becomes "Ment.?" and is not appropriated because it starts does not start with a lower-case character.

OCR errors sometimes hinder a proper recognition of hyphenation occurrences, because dirt or vertical lines to separate text segments on both borders of the text can produce additional characters such as punctuation marks, e.g. *de-.* instead

of *de-* or *,partment* instead of *partment*. Naturally, more characters besides punctuation marks can occur on the sides, such as alphabetic characters or digits, e.g. *IBritain* instead of *Britain*. But those can or should not be removed because it is not clear if they are a part of the original token. Therefore, only punctuation marks can be removed from the sides of the tokens.

If digits or upper-case characters occur around a hyphen the merge will not be performed, because it is likely to be a false merge, e.g. *Hyde-* $\backslash n$ *Park*.

It has to be pointed out, that this is only a heuristic and may lead to wrong merges. An evaluation of the heuristic is given in Chapter 5. Wrong merges can be caused by two things:

1. Due to dirt on the right side of a document, a hyphen can be added to a token accidentally by the OCR processor, e.g. *as-* $\backslash n$ *the* becomes *asthe*.
2. Words which are naturally spelled with a hyphen can be split up at line breaks and therefore be merged wrongly, e.g. *full-* $\backslash n$ *scale* becomes *fullscale*.

Wrongly merged words can sometimes be recovered by the anagram-hash correction algorithm. For example correction proposals for the wrongly merged token *fullscale* could be *full-scale* or *full scale*.

Detecting Hyphenation Errors in The Times Archive

Given the following two XML tags of The Times Archive[1]:

```
<wd pos="626,2608,797,2653">procure-,</wd>
<wd pos="139,2640,255,2686">ment</wd>
```

The *pos* attribute of the tag *wd* contains information about the position of the top left corner (x_1, y_1) and the bottom right corner (x_2, y_2). A hyphenation can occur at line-breaks or column-breaks. In Figure 4.4 depicts a line-break and a column-break example.

Given the token t with position x_1, y_1, x_2, y_2 and token $\tilde{t}$ with positions $\tilde{x}_1, \tilde{y}_1, \tilde{x}_2, \tilde{y}_2$. Then line-breaks and column-breaks can be detected as follows:

[1]Example taken from: The Times Archive 1985-02-04, page 12, article 1

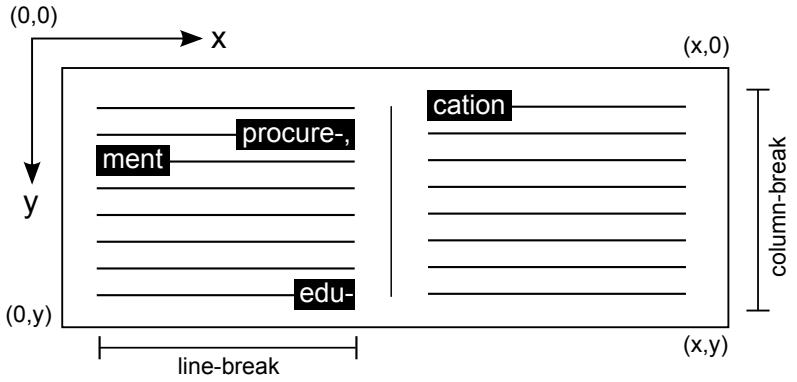

Figure 4.4: Hyphenation Errors: Line-Breaks and Column-Breaks. A schematic illustration of occurring line-breaks and column-breaks in The Times Archive.

Line-Break A line-break occurs if $x_1 > \tilde{x}_1$. No further information is necessary.

Column-Break A column break occurs if $y_1 > \tilde{y}_1 + \delta$. The tolerance value δ is necessary for compensating alignment problems while scanning documents. Usually two neighbour tokens are not scanned absolutely horizontal. Lines of text tend to move up or down. If they move up, then y_1 is greater than $\tilde{y}_1$ and a false column-break can be detected. Using a tolerance value δ prevents the detection of false column-breaks. Based on preliminary experiments δ is set to 20.

4.2.2 Filtering Rules

Applying the Anagram Hash and OCR-Key method to all words present in the text dramatically slows down the system. Therefore, two rules are applied to filter out tokens which should not be corrected. The rules are applied in the order they are mentioned.

Short Word Rule

Words with a length ≤ 2 are filtered out in the following process. This is done because it is very hard to correct words which consist of only one or two characters. Due to OCR errors short words can be dirt, error-prone, subsequence of other words which were separated accidentally or even correct.

Dictionary Rule

The dictionary rule uses the collection-based dictionary (from the pre-processing step in Section 4.1.5) to filter out correct words. Omitting these words has two advantages. Firstly, potential correct words can not accidentally be corrected to a wrong word. Secondly, the speed of the system increases.

The dictionary rule applies a second, more sophisticated clean of each token. Until now the token is cleaned only on the sides. But sometimes punctuation marks or other "dirt" characters are included in the token. Therefore, every character which is not a digit or alphabetic character is removed. The resulting token is checked again against the dictionary. If it is contained in the dictionary, the token is filtered out and the cleaned version of the token is added to the correction proposal list which can be used later as a proper correction candidate.

4.2.3 Anagram Hash Correction

The general Anagram Hash method for retrieving correction proposals for misspelled words is described in Section 2.3.2. The concrete implementation details are given here.

Implementation Details

The underlying anagram-hash-map was already generated in the pre-processing step (Section 4.1.3). It includes frequent lower-cased tokens and token pairs.

The size of the alphabet Σ_W^k is limited to $k = 2$. The word-list W for defining the alphabet contains all words of the pruned lower-case OM. The choice of k is a trade-off between computational effort and ability to find more similar words.

Because the anagram-hash map still contains OCR errors, the alphabet can not be generated by simply applying the definition of Σ_W^2. The alphabet should only contain characters which are necessary to spell words.

Because this approach computes only on lower-cased words, the following characters are necessary to spell a word: $a - z$, a hyphen (-), an apostrophe (') and a white-space (␣). Therefore, the alphabet Σ_W^2 contains the following character n-grams:

unigram: Characters $a - z$, a hyphen (-), an apostrophe (') and a white-space ($\sqcup$).

bigram: All character bigrams which can be formed out of sws with $w \in W$ and s white-space. Bigrams which contain characters that are not necessary to spell a word are removed, e.g. *a(* or *a$*.

The confusion alphabet Π^2_W therefore contains all anagram keys of the alphabet defined above. The focus word alphabet $\Gamma^2_{\{w\}}$ contains all anagram keys of character unigrams and bigrams of the focus word using the strict alphabet. Note that the focus word alphabet is not restricted as the confusion alphabet. This is important to be able to delete and/ or substitute all appearing character n-grams of the focus word.

Remembering the computation of correction proposals using the Anagram Hash Method. The computation uses the following formula:

$$simulatedWordHash \leftarrow focusWordHash + \pi - \gamma, \quad \pi \in \Pi, \gamma \in \Gamma$$

This formula requires that $\pi = 0$ to delete a character and $\gamma = 0$ to add a character. Therefore, the zeros are added to the confusion alphabet Π^2_W and focus word alphabet $\Gamma^2_{\{w\}}$.

A Levenshtein-Distance (LD) limit is used to limit the retrieval of correction proposals. Therefore, the LD between the original word in the text and the retrieved word has to be computed. If the retrieved word exceed the limit, it will be discarded. This implementation uses a LD limit of 3 for all words. That means only words which have a limit ≤ 3 will be retrieved. The retrieved words are added to the anagram proposal list. Note that only lower-cased words are retrieved. For computing the LD the implementation of the Java Open Source Spell Checker Jazzy [Idz10] is used.

Because the Anagram Hash method retrieves the same correction proposal several times, the LD computation is only necessary for words which were not already retrieved. This increases the speed of the algorithm.

Impact of Bigrams

Because the anagram-hash map includes bigrams on word level, and the confusion alphabet a white-space and hyphen, the algorithm is able to split up words, e.g. *inBritain* $\rightarrow$ *in Britain* or *fullscale* $\rightarrow$ *full-scale*. The first splitting using a white-space is only possible if enough bigrams on word level are present in the anagram-hash map. Therefore, as many bigrams as possible have to be collected in the pre-processing step.

4.2.4 OCR-Key Correction

The OCR-Key method was extensively explained and discussed in Chapter 3. Using the OCR-Key map generated in the pre-processing step (Section 4.1.4) three more implementation details are necessary for applying the OCR-Key method.

1. Algorithm 3.1 (page 31) modifies the cardinalities of each equivalence class by adding an subtracting a δ. This δ is set to 2.
2. Retrieved words are limited by a Levenshtein-Distance limit of ≤ 2. The retrieved words are added to the OCR-Key proposal list. Note that only lower-cased words are retrieved.
3. Because of the s/f-problem the OCR-Key map uses a different classification of characters as described in Section 4.1.4.

4.2.5 Generating the Final Proposal List

After computing correction proposals for potential misspelled words, two proposal lists are present for a single token. Both lists have to be merged to a final proposal list. This is done in three steps:

1. Use the anagram-hash proposal list as final list.
2. The five highest ranked proposals of the OCR-Key proposal list are used to boost the ranking of already existing equal proposals in the final list. Details are explained in Section 4.2.8 about the score computation (ocrBoost).
3. If not already present, the ten highest ranked proposals of the OCR-Key proposal list are additionally added to the final proposal list.

4.2.6 Correction Proposal Enhancement Rules

After processing every token, each token should have one or more proposals. If the anagram hash method and the OCR-Key method did not find any correction proposal the token itself is set as correction proposal. In order to enhance the found correction proposals of tokens, which were not filtered out, three rules are applied. The first rule uses context information of bigrams to influence the ranking of the five top-most proposals. Because all correction proposals are lower-cased, the second rule computes for each correction proposal a case-sensitive version which can be used later. The third rule is used to filter out some more tokens which should not be written to the XML correction files.

Bigram Rule

In addition of using bigrams to correct words, these bigrams can also be used to influence the ranking of the five top-most correction proposals of the final proposal list.

Let w_i be the i-th word of the text. Further, let c_j^i be the j-th correction proposal of the word w_i and $|c^i|$ the number of correction proposals. Then there exists two ways to form a bigram including the word w_i.

1. The bigram b_{jk}^{i-}, which is formed using the proposals of the word w_i and w_{i-1}.
2. The bigram b_{jk}^{i+} which is formed using the proposals of the word w_i and w_{i+1}.

Formally, the bigrams are defined as follows, using $\llcorner$ as white-space:

1. $b_{jk}^{i-} = c_j^{i-1} \circ \llcorner \circ c_k^i,\ 1 \leq j \leq \min(5, |c^{i-1}|),\ 1 \leq k \leq \min(5, |c^i|)$
2. $b_{jk}^{i+} = c_k^i \circ \llcorner \circ c_j^{i+1},\ 1 \leq k \leq \min(5, |c^i|),\ 1 \leq j \leq \min(5, |c^{i+1}|)$

To respect the context, bigrams are only generated if they are not blocked by punctuation marks. If w_{i-1} end with a punctuation mark no bigram between w_{i-1} and w_i will be formed. If w_i ends with a punctuation mark no bigram between w_i and w_{i+1} is formed. As punctuation mark all non-digit and non-alphabetic characters are considered.

The bigrams are now queried against the anagram-hash map to retrieve their occurrence count. The frequency of the bigram b in the anagram-hash map is denoted as $f(b)$. The resulting $bigramBoost_k^i$ for the correction proposal c_k^i (of the word w_i) is computed as follows:

$$bigramBoost_k^i = \sum_{j=1}^{\min(5,|c^{i-1}|)} f(b_{jk}^{i-}) + \sum_{j=1}^{\min(5,|c^{i+1}|)} f(b_{jk}^{i+}),\quad 1 \leq k \leq \min(5, |c^i|) \quad (4.1)$$

This *bigramBoost* value is later used for computing the score.

Alternative Case Rule

The anagram-hash and the OCR-Key method generate and retrieve lower-cased correction proposals only. The correction proposals can be enhanced by adding a spelling variation with respect to lower-case and upper-case characters. For example a name should be spelled with an initial upper-case letter. Therefore, the alternative-case map can be used (see Section 4.1.2) which contains the three most frequent spelling variations of tokens occurring in the pruned lower-case OM.

The best spelling variation with respect to lower-case and upper-case characters using the alternative-case map is computed in the following way:

1. A token (in the text) starts with a capital letter and contains more than two capital characters. Then the most frequent spelling variation which starts with a capital letter and which has more than two capital characters is used.

   Example: The token *AppLicAtioN* will likely be transformed to *APPLICATION* if it is present in the alternative-case map.

2. A token (in the text) starts with a capital letter and contains ≤ 2 capital characters. Then the most frequent spelling variation which starts with a capital letter and contains ≤ 2 capital characters is used.

   Example: The token *ApplIcation* will likely be transformed to *Application* if it is present in the alternative-case map.

3. If no initial capital letter is recognized or no matching token could be found, then the most frequent spelling variation is used.

   Example: The most frequent spelling variation for the token *britain* is *Britain*.

The capital character threshold of two is chosen because of the fact that some names usually contain more than one capitalized character, e.g. *McChrystal*. Furthermore, OCR errors may insert additional capital characters, e.g. *ApplIcation*. These words shall not use their complete capitalized spelling variation.

Ignore Word Rule

Because the correction strategy should be able to correct misspellings fully-automatically, it is necessary to make sure that few false corrections are applied. Therefore, a special rule is used to ignore some words.

If the best correction proposal equals the original token in the text, no correction is necessary. Therefore, these tokens can be ignored when generating the correction XML file.

Names are often hard to correct. For example similar names such as *Miller* and *Miler* could be both correct, or *Miler* is a misspelling of *Miller* or the other way around. If the initial letter of a potential misspelled token starts with a capital letter it can be a name. Then the top five proposals are checked against the original token with respect to upper- and lower-case characters. If the original token is among the top five proposals with exact the same case-sensitive spelling, the token will be ignored when generating the correction XML file. Note that proposals got a case-sensitive spelling variation from the alternative-case rule (Section 4.2.6).

4.2.7 Generate Correction XML File

For the post-processing step, the computed correction proposals of each document are stored into an XML file. The exact structure can be found in Appendix A.

In this correction XML file only tokens which have valid correction proposals are saved, e.g. short words or words that fulfil the ignore word rule are ignored. Merged tokens are also saved because they contain information for the post-processing step.

After determination which correction proposals need to be stored, only the five best ranked proposals are stored with its normalized score. The ranking mechanism using a scoring system will be explained in the next section in detail.

4.2.8 Score: Ranking of Correction Proposals

The Anagram Hash method and the OCR-Key method both produce correction proposals. Usually, they produce several proposals. In order to say which correction proposal is the best one, they have to be ranked. For this a simple scoring system is used.

A scoring function assigns a number to a correction proposal which states how suitable a proposal is. Because the Anagram Hash and the OCR-Key method rely on different numbers and behave differently, they have different score functions. For ranking the OCR-Key method the scoring function proposed in Section 3.3.4 is used. For ranking proposals of the Anagram Hash method a similar scoring function is used.

To compute the score for a correction candidate, there are initially two general influencing variables:

Frequency. Because the underlying dictionary, which is used to retrieve correction candidates, is derived from the archive, it still contains OCR errors. Hence, the correction algorithm will likely retrieve error-prone candidates among others. By assuming that frequent tokens are more likely to be correct, the frequency can be used to rank the retrieved correction proposals.

Levenshtein-Distance The Levenshtein-Distance (LD) can be used to determine the similarity of two words with respect to character insertions, deletions and substitutions. Hence, a small LD indicates that two words are similar. The more similar correction proposal and original word are, the more likely, that the correction proposal is an adequate proposal.

Ranking Anagram Hash Proposal

Let w a word with the length $|w|$ for chich correction proposals should be proposed. A correction proposal is denoted as c and its frequency as $f(c)$. Internally, a correction proposal can be retrieved several times by the Anagram Hash method (see Section 2.3.2). The amount of internal retrievals is denoted as c_r. The following scoring function is used for the Anagram Hash method:

$$\text{score}_{ana}(c, w) = \ln(f(c)) \cdot (|w| - \text{LD}(w, c)) \cdot c_r$$

In order to restrict the influence of the frequency, the natural logarithm of the frequency is used here. To put the LDinto relation with the length of the word, the LDis subtracted from the token length of the original token.

Ranking OCR-Key Proposals

Additional to the definitions above, let c_c be the cardinality difference which lays between the OCR-Key of the focus word w and the OCR-Key of c (see Section 3.3.4). Then the score function for the OCR-Key method is computed as follows:

$$\text{score}_{ocr}(c, w) = \ln(f(c)) \cdot (|w| - \text{LD}(w, c) - c_c)$$

Ranking the Final Proposal List

The anagram-hash proposal list and the OCR-Key proposal list are merged to the final proposal list at the end of the main-processing step. In order to rank this final proposal list a unified formula needs to be applied. Because two further factors influence the ranking computation, a normalization is not computed before all steps of the correction strategy are finished.

The first factor is the *ocrBoost* variable which occurs while merging the anagram-hash proposal list and the OCR-Key proposal list (see Section 4.2.5). The second factor is the *bigramBoost* variable which occurs while applying the bigram rule (see Section 4.2.6).

The final proposal list uses the following unified scoring function for ranking the computed correction proposals:

$$\text{score}_{final}(c, w) = \ln(f(c)) \cdot (|w| - \text{LD}(w, c)) \cdot c_r \cdot ocrBoost \cdot \ln(bigramBoost)$$

The variable *bigramBoost* is initially set to 2, because of the logarithm. The variable *ocrBoost* is initially set to 1. This makes sense in order to keep the initial computed scores as much as possible. The variables for each correction proposal are computed as follows:

ocrBoost. This variable is computed while merging anagram and OCR-Key proposal lists to the final list.

Before OCR-Key proposals are added, the five best ranked OCR-Key proposals are compared with the ones in the final list. The *ocrBoost* value of a proposal which is also in the final list is set to the score of the proposal in the OCR-Key list.

Then, the ten top ranked proposals of the OCR-Key proposal list, which are not included in the final proposal list, are added to the final proposal list. Because the score function differs from the OCR-Key score function, the additional value $c_r = 1$ is used. The value c_c is discarded.

bigramBoost. The *bigramBoost* of each proposal comes from the computation in Section 4.2.6 (Formula 4.1).

The correction XML file contains only the top five correction proposals for each token and the corresponding scores which are normalized with the sum of all five scores to give a value between 0 and 1. The score represents the confidence for each proposal.

4.3 Post-Processing

This section describes how correction proposals are chosen for automatic replacement of misspelled words. The first step determines if the correction proposals are good enough based on the score. In a second step they are replaced.

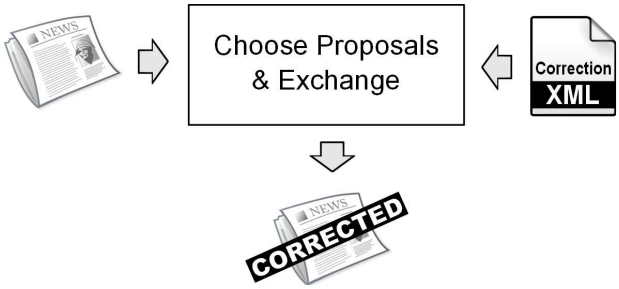

Figure 4.5: Correction-Strategy: Post-Processing.

4.3.1 Choose Adequate Correction Proposals

Each correction proposal has a score between 0 and 1. The assumption is that a proposal with a high score is more likely to be a correct correction proposal than a proposal with a low score. Thus, the best correction proposal is the one with the highest score. Following this assumption, every misspelling can be replaced by its best correction proposal.

Experiments showed that the system can produce correction proposals with two characteristics. Firstly, the scores may not be high enough to be significant. Secondly, the best score may be very close to the second best score. To prevent automatic corrections in these two situations thresholds can be constructed and applied. If these thresholds are not satisfied, the system will not apply an automatic replacement:

Minimum Threshold. The score of the first best correction proposal needs to exceed a minimum threshold t_{min}:

$$\text{score}_{\text{1-best}} \geq t_{min}$$

Logarithmic Threshold. The second best proposal's score has to be set into relation with the first best proposal's score. The natural logarithm of this relation has to exceed a certain threshold t'_{min}:

$$\ln\left(\frac{\text{score}_{\text{1-best}}}{\text{score}_{\text{2-best}}}\right) \geq t'_{min}$$

Another possible threshold for preventing automatic correction is to limit the Levenshtein-Distance:

$$\mathrm{LD}_{\text{1-best}} \leq \mathrm{LDLimit}$$

4.3.2 Applying the Correction Proposals

After determining if the correction proposals are good enough, they can be replaced. Two things have to be considered in order to replace the proposals correctly:

1. Heading and tailing "dirt" characters were removed during the correction process. When exchanging the original word of the text with the correction proposal these characters have to be kept. They contain structural information, such as points and commas.
2. Hyphenation errors were merged and saved as single correction proposals. Therefore, two tokens in the text have to be replaced by a single correction proposal.

4.4 Discussion

This chapter described the generation of correction proposals in three processing steps and the cooperation between the Anagram Hash and the OCR-Key method. In the pre-processing step occurrence maps (OMs) were generated by collecting words and word pairs from the archive. In order to remove potential OCR errors from these OMs a simple pruning strategy was applied. In order to remove more OCR errors an improved pruning strategy should be developed and applied in future work.

Besides OMs, the OCR-Key map was generated in the pre-processing step. This map uses a different classification than proposed in Chapter 3 in order to cope with the s/f spelling variation problem. It has to be investigated in future work where these spelling variation occur and if it is worth to differ from the proposed classification.

The used frequency counts in the OCR-Key map can be further improved. The map uses separate counts for tokens which have the same spelling but different OCR-Keys due to case-sensitive spelling variations. For example the token *Britain* has another OCR-Key than *britain*. The occurrence count of the token *Britain*, *BrItaIn* or *BrItain* are summed up and stored as occurrence count for the token *Britain*, but the token *britain* does only have the occurrence count of *britain*,

brItain or *brItaIn*. That means the tokens *Britain* and *britain* have different occurrence counts in the OCR-Key map. This behaviour can be desirable because tokens with different OCR-Keys should not be similar, but on the other hand they refer to the same token and could therefore have the same occurrence count.

The used Anagram Hash method retrieves correction proposals by using character unigrams and bigrams. A further improvement could be to use also character trigrams in the confusion and focus word alphabet to expand the search for alternative spellings. This can increase the retrieval of adequate words but will also increase the computational effort. Furthermore, the Levenshtein-Distance limit could be increased for the Anagram Hash as well as for the OCR-Key method to retrieve more words. The current limit is a trade-off between retrieving adequate correction proposals and preventing a mis-correction by proposing wrong proposals.

Because the proposed correction strategy uses only words which are contained in the archive itself limits the retrieval of adequate correction proposals in the following sense. A word which is not contained in the archive, because it is never recognized properly (or is not frequent enough), cannot be retrieved as correction proposal. Proper corrections are especially not possible in years where font variations problems occur. For example, the misspelled word *plcefe* can at least be corrected to *pleafe* in years where an *s* is spelled as an *f*.

5 Evaluation

This chapter deals with the evaluation of the correction strategy described in Chapter 4. The correction strategy is evaluated on The Times Archive. Therefore, some background information about the archive can help putting the right emphasis on the correction strategy and evaluation. Hence, Section 5.1 first introduces the used dataset and gives an overview of the data, the quality and some special characteristics. The next section introduces and explains the used evaluation methods and gives the final results in Section 5.3. The chapter concludes with a discussion and a comparison with related work.

5.1 The Dataset (The Times Archive)

The OCR processed newspaper articles from The Times Archive in London, spanning from 1785 to 1985. The OCR quality will be investigated in Section 5.1.1 in order to give an approximation of contained OCR errors in The Times Archive. Section 5.1.2 analyses the contained article categories. In Section 5.1.3 some special characteristics of the archive are presented.

In general The Times Archive contains 201 years of data with roughly 7.1 billion white-space separated tokens and about 7.8 million articles. In average each year contains about 35 million white-space separated tokens. The number of tokens ranges from about 3.9 to almost 68 million white-space separated tokens.

Figure 5.1 illustrates the size of The Times Archive with respect to contained articles and white-space separated tokens. The archive starts with about 4400 articles and almost 3.9 million tokens. The number of articles and tokens increases steadily. Especially in the early 20th century the number of articles and tokens reaches the archive's peak with about 68 million token and nearly 92.000 articles. In 1940 the number of articles and tokens decreases about 50%.

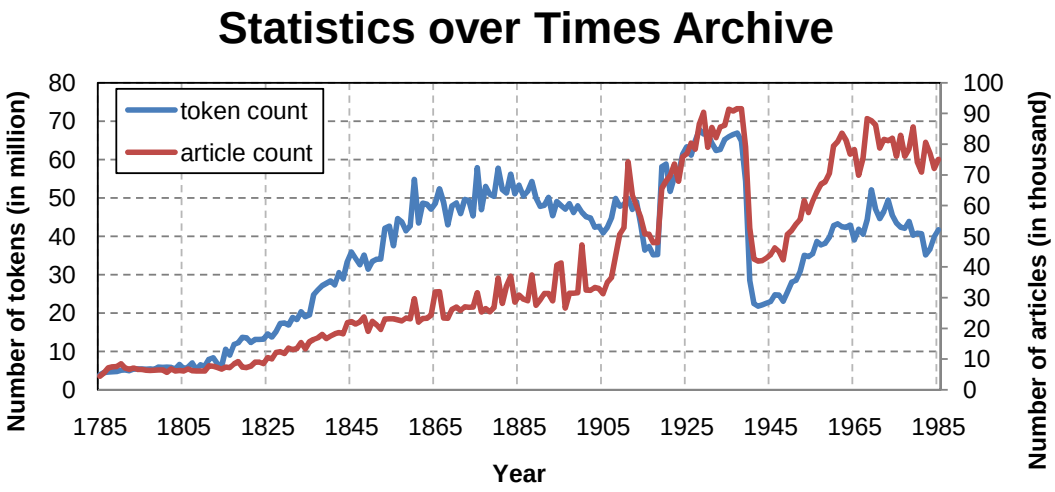

Figure 5.1: Statistics over Times Archive: Number of articles and white-space separated tokens.

5.1.1 OCR Quality

The amount of OCR errors in The Times Archive is approximated by measuring the dictionary recognition rate. The dictionary recognition rate measures the amount of tokens from the archive which also appear in a dictionary. Hence, the approximate amount of OCR errors is

$$OCRerrors \approx 1 - f_D(t) \tag{5.1}$$

where $f_D(t)$ is the dictionary recognition rate for the dictionary D and the year $t \in [1785, 1985]$.

Using a dictionary recognition rate is an approximation of contained OCR errors because contemporary dictionaries usually do not contain out-dated terms, names or domain specific words. Besides the lack of these words, some error types are counted wrongly. For example, segmentation errors such as *de pa rt ment* should be counted as a single error, but a dictionary does not know, that the character segments belong together. Depending on the used dictionary, zero to four errors could be counted for the split word *de pa rt ment*. Hence, the dictionary recognition rate can only be a rough approximation of OCR errors.

In order to spell-check the text, all single white-space separated tokens are cleaned from heading and tailing dirt before they are checked against a dictionary. That means, all characters which are non alphabetic are removed from the sides of a

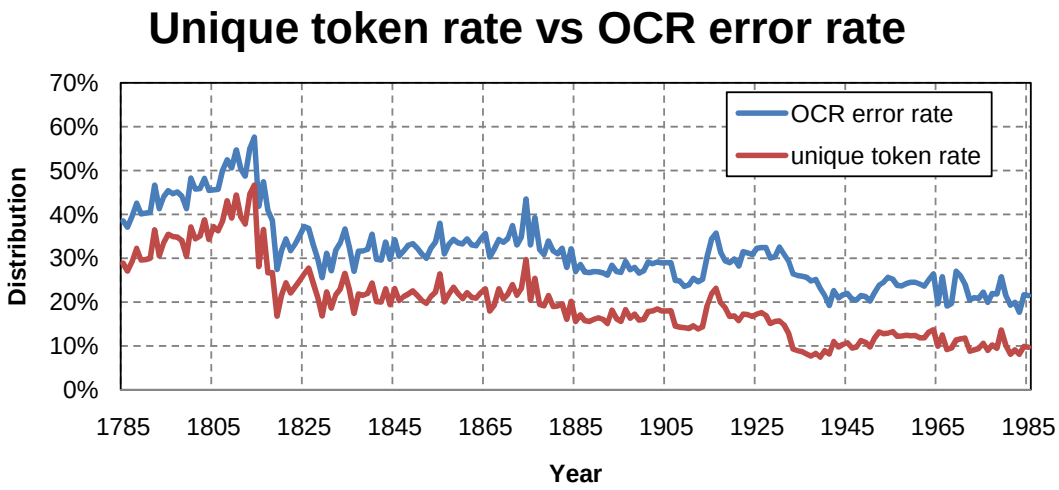

Figure 5.2: Unique Token Rate vs OCR Error Rate: The rate of OCR errors is approximated using 1 minus the Aspell dictionary recognition rate (see Equation 5.1).

token until an alphabetic character occurs. For example the token *12bi1rds.%* becomes *bi1rds* after cleaning. All tokens are checked against the dictionary in lower-cased form.

The used dictionary is extracted from GNU Aspell 0.60.6 [Atk08]. The Aspell dictionary contains about 138k single tokens. All tokens are converted to a lower-cased form. The extracted Aspell dictionary contains lemmas, names and irregular word forms. Therefore, no lemmatization is necessary in order to spell-check all tokens properly.

In Figure 5.2 the unique token rate is depicted against the OCR error rate, computed by Formula 5.1, using the Aspell dictionary. With respect to the OCR error rate, the archive can roughly be divided into three periods. The first period from 1785 to 1814 has a relatively high and steadily increasing error rate. The second period from 1815 to 1932 and the third from 1933 to 1985 have both relatively stable errors rates but the third has a lower error rate.

Nearly 39% of the cleaned tokens cannot be recognized by the Aspell dictionary in the first year. The approximated error rate increases in the following years until the year 1814 where the error rate drops about 20%. The decreased error rate can be explained by the increasing print quality produced by steam presses which were introduced in November 1814 [Tim14]. In the following years the error rate

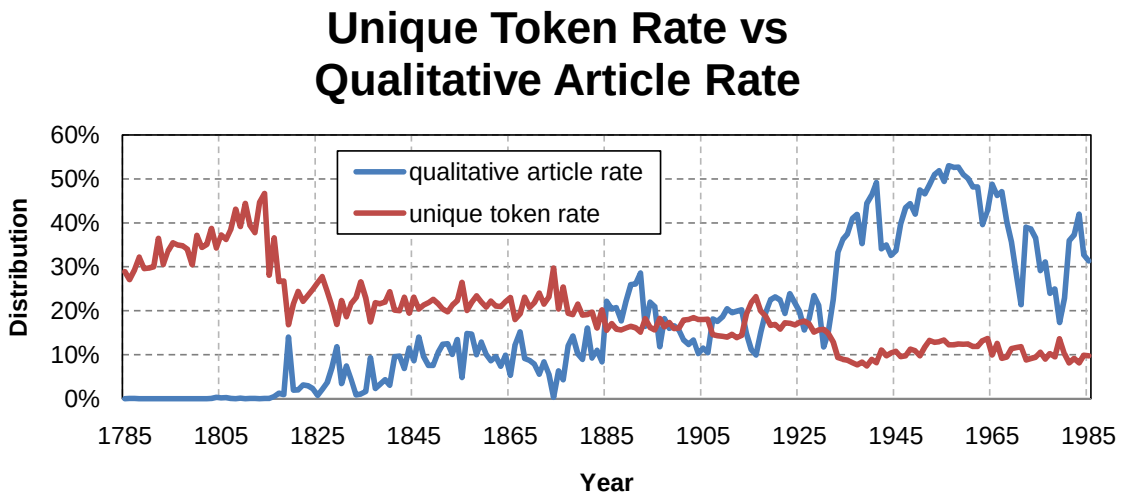

Figure 5.3: Unique Token Rate vs Qualitative Article Rate: The fraction of qualitative articles is plotted against the fraction of unique tokens. The dictionary recognition rate of qualitative articles is higher than 90% using Aspell (see Equation 5.1).

is relatively constant and decreases again between 1930 and 1940. Until 1985 the rate is again relatively stable.

The unique token rate (percentage of unique tokens with respect to all tokens) behaves very similar as the error rate. This is clear because OCR errors produce several spelling variations of the same word, e.g. *time*, *t1me*, *tine*, which are usually OCR errors. That means, if the rate of unique tokens is higher, then the rate of OCR errors is higher, and vice versa.

The correlation between unique tokens and OCR quality can also be seen in Figure 5.3. The figure plots the unique token rate against the qualitative article rate. Qualitative articles are articles which achieve a dictionary recognition rate higher than 90% using Aspell (see Equation 5.1) and contain more than 20 tokens after cleaning. Until 1816 only few qualitative articles are present in the archive and the unique token rate is relatively high. Between 1816 and 1880 more qualitative articles can be found. The rate of qualitative articles is relatively stable in these years. Between 1880 and 1930 the qualitative article rate is higher than in the previous period. The corresponding unique token rate is slightly lower. From 1930 the qualitative article rate increases dramatically and is more than twice as high

Category	$\frac{\#token}{\#article}$	$\frac{\#token}{\#allToken}$ [%]	$\frac{\#article}{\#allArticle}$ [%]
Arts and Entertainment	591	2.5	3.9
BIRTHS;MARRIAGES AND DEATHS;	541	2.1	3.6
BUSINESS AND FINANCE	670	8.5	11.7
COURT AND SOCIAL	370	0.8	2.0
Classified Advertising	4681	24.6	4.8
Display Advertising	239	1.8	7.0
Editorials/Leaders	936	2.8	2.7
Feature Articles (aka Opinion)	343	0.2	0.4
Law	861	5.1	5.5
Letters to the Editor	449	3.0	6.0
NEWS	565	21.7	35.2
Obituaries	306	0.6	1.8
Official Appointments and Notices	661	1.7	2.4
Politics and Parliament	3048	5.2	1.6
Property	3856	9.5	2.2
Sport	791	4.9	5.6
Stock Exchange Tables	1296	4.6	3.3
Weather	683	0.4	0.5

Table 5.1: Statistics for Times Archive Categories: Average length and portion with respect to tokens and articles

between 1930 and 1985 as in the period before. The unique token rate decreases in the last period. The high qualitative article rate in the last period is likely caused by the higher quality of documents which allows more articles exceed the 90% threshold.

5.1.2 Categories

The Times Archive is categorized into 18 categories when digitizing the archive. Some statistics about the average length of articles and the portion of categories with respect to the amount of tokens and articles can be found in Table 5.1. The two most dominating categories with respect to the amount of tokens are *Classified*

Advertising with 24.6% and *News* with a portion of 21.7%. The two most dominating categories with respect to the amount of articles are *News* with 35.2% and *Business and Finance* with 11.7%. The longest articles can be found in the category *Classified Advertising* with an average length of 4681 white-space separated tokens per article.

5.1.3 Special Characteristics

The Times Archive has several characteristics. Two of these characteristics are pointed out in the following: The used character set and the font variation problem.

Used Character Set

In order to correct OCR errors it is important to know which character set is used in the archive. Depending on the used language, different character sets are meaningful. The following characters can be found in The Times Archive:

- Alphabetic: a-z, A-Z
- Numbers: 0-9
- Punctuation marks: `'-!#$%&()*,./:;?@[]^_‘|+<=>{}~\`
- Special characters: Â, Ã

All standard alphabetic characters and digits are present. Furthermore, nearly all punctuation marks are present. Likely the special characters Â and Ã are OCR errors, since these characters do not appear naturally in English texts.

It is interesting that the archive lacks of currency symbols. Although it is an British newspaper no £-symbol is present in the archive, only the $-symbol.

During World War II German names such as *Göring* occur frequently. The problem is, that such names cannot be represented, because the character “ö” is not present in the character set. Another example is for the German word “Börse” (stock market). The word “Börse” is for example recognized as “B&rse” and cannot be corrected using only the archive itself since important characters are missing.

The lack of characters hinders a proper recognition of some words. With regard to the mainly used language in the archive not many words are affected. Hence, a reduced character set prevents the formation of OCR errors. Dirt can cause undesired misrecognitions of several characters easily, e.g. *o* and *ö* or *u* and *ü*.

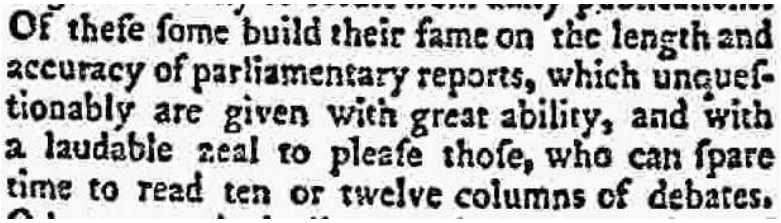
Of theſe ſome build their fame on the length and accuracy of parliamentary reports, which unqueſtionably are given with great ability, and with a laudable zeal to pleaſe thoſe, who can ſpare time to read ten or twelve columns of debates.

Figure 5.4: Font Variation Problem: s/f. The character *s* is misrecognized by the OCR processor because it looks like an *f*[1].

Font Variation

The lower dictionary recognition rate in the earlier years is not only caused by bad quality of documents but also by a font variation problem. The character *s* is often recognized as an *f*. This problem is further referenced to as the *s/f-problem*. A sample document which contains the s/f-problem can be found in Figure 5.4[1]. In this short sentence, six words use the character *s* which looks like an *f*. These words are: *these*, *some*, *unquestionably*, *please*, *those*, *spare*. All these words are recognized with an *f* instead of an *s*.

The s/f-problem mainly occurs in the earlier years, especially before 1804. Because the s/f-problem also occurs a few times in 1985, in future work it should be investigated where and how frequently the problem occurs.

5.2 Evaluation Method

This section describes the method used for evaluating the correction strategy proposed in Chapter 4.

Because the archive itself is used to generate correction proposals, enough qualitative data has to be available. Due to the heavy s/f-problems before the year 1804 it is not meaningful to use any data of these years for evaluation. Other evaluations on The Times Archive used a specific 10% sample such as Tahmasebi [Tah09] and Tahmasebi et al. [TNTR10]. The subset used in this thesis for evaluation are the following eight years of The Times Archive: 1835, 1838, 1885, 1888, 1935, 1938, 1982, 1985.

[1]Example taken from: The Times Archive 1985-01-02, page 25, article 2. Copyright of The Times, Times Newspapers Limited.

Out of these 8 years, sample documents were randomly chosen and correction proposals were generated. Human assessors counted the errors of each chosen document. Then they chose adequate correction proposals generated by Aspell as well as the proposed correction strategy presented in Chapter 4, here on referenced to as OCRpp. Based on these data the evaluation of OCRpp could be applied and compared to the performance of the Aspell spell-checker.

Because no specialized OCR error correction system was available as baseline, Aspell was chosen. It claims to perform very well on correcting human spelling mistakes and is easy to run parallel to OCRpp.

5.2.1 Choosing Documents and Generating Correction Proposals

From each chosen sample year, four documents were randomly chosen which fulfilled the following two criteria:

1. The document is classified as *News*.
2. The document contains between 500 and 1000 white-space separated tokens.

The News category has been chosen because it is one of the most dominating category with valuable content. For having comparable articles, the length of each article was limited to a lower bound of 500 tokens and an upper bound of 1000 tokens. The chosen documents can be found in Appendix A.

In order to retrieve similar token counts for each evaluated year, all documents were shortened. Therefore, the first occurring full stop after 500 tokens was chosen as a cut off and the rest of the text was removed. Using a sentence separator instead of a fixed number can preserve the context of the last sentence and make it easier for human assessors to read and use the documents.

On these samples the described correction strategy was applied yearly (see Chapter 4). That means, each four documents were corrected using the pre-processed data of the corresponding year.

In order to compare the quality of the generated correction proposals by OCRpp the spell-checker Aspell version 0.60.3 was applied in parallel as base-line. Furthermore, the ignore word rule was not applied if Aspell detects an error and returns any correction proposals. Additionally, Aspell uses the heuristic for merging hyphenation errors. Both modifications are necessary for evaluating both OCRpp and Aspell, properly. Preliminary experiments showed that it is reasonable to tackle only tokens of the length > 2. Hence, both system ignore short tokens.

Because the evaluation is done manually, the generation of correction proposals was limited to five. Having more than five proposals for each potential misspelling is not manageable for human assessors because it causes too high effort. Therefore, only a 1-best and 5-best evaluation is possible. 1-best evaluations consider only the top ranked proposal. 5-best evaluations consider all five proposals.

5.2.2 Counting the Errors

For evaluation, knowing the number of errors of the chosen documents is necessary. Because no fully corrected documents of The Times Archive are available, human assessors had the task to mark and count errors on printed documents. In order to ease this task, the OpenOffice spell checker version 3.2 marked potential misspellings beforehand [Ope10]. Additionally, the assessors were allowed to use the internet, e.g. online dictionaries to check unknown words.

Because the OpenOffice spell checker was used to mark potential errors, the assessors were told not to focus on these marks only. This was only thought to ease the job. Furthermore, they were told to mark and count the marks, to get more reliable counts.

The assessors were told to find the following error types: segmentation errors, hyphenation errors and misrecognized characters. The following three error types are not interesting in this thesis and should therefore not be counted: punctuation errors, case sensitivity errors and changed word meaning errors. (See Section 1.3 for more information on these error types)

Because this approach focuses on correcting words, one misspelled word counts as one error. That means, if a segmentation error occurs, each involved word counts as one error. For example *de par t ment* does only count as one error because it contains the word *department*. The error *thiswasa* counts as three errors, as it includes three words *this was a*.

Each assessor got four documents each from one of these four periods: 1835/ 1938, 1885/ 1888, 1935/ 1938, 1982/ 1985. This was done to divide the work equally, because the earlier documents contain more errors.

5.2.3 Choosing proposals

After counting all errors in each document, the assessors were provided a tool, especially designed for choosing adequate correction proposals. This tool is called *OCR Correction GUI* in the following. The assessors used the tool for evaluating the same documents which they already processed by hand.

The OCR Correction GUI is depicted in Figure 5.5. It contains the full article on the left hand side. Errors found by Aspell and OCRpp are marked yellow in the text field. Evaluated errors are marked green. Correction proposals and other choices for a token are displayed on the right hand side. The navigation is underneath the correction proposal lists. The next button cannot be chosen unless a choice has been made for the previous error. At the bottom, information about the open document can be found including a text field where the counted errors can be filled in.

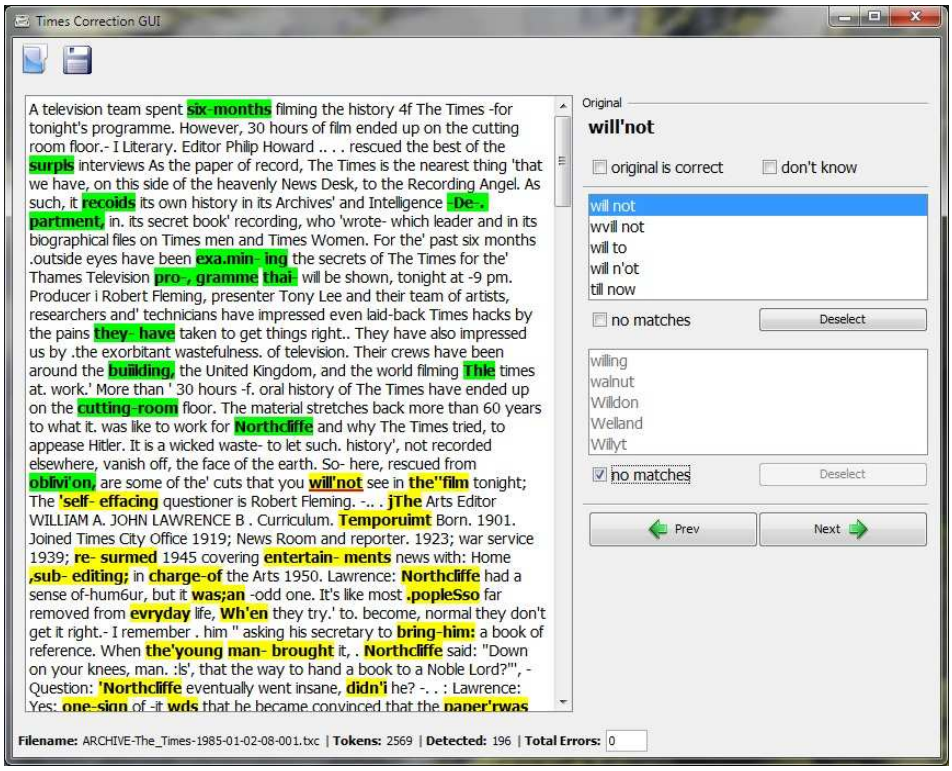

Figure 5.5: OCR Correction GUI. A GUI for evaluating correction proposals generated by OCRpp and Aspell.

For choosing the correct proposals, the following guidelines should be followed:

1. Proposals were allowed to be chosen case insensitive, e.g. for the original word *IBritain* the correction proposals *Britain*, *britain* or *BRITAIN* are all correct and could be chosen. If proposals only differ in their case then the best matching proposal should be chosen (e.g. for *IBritain* the best matching proposal is *Britain*).
2. A proposal without a hyphen is still correct, e.g. *well-organized* and *well organized* are both correct.

Although case-insensitive word forms and missing hyphens are not absolutely correct, the proposed words are correctly spelled. For this evaluation the correct spelling was more important than the appearance of tokens.

It has to be pointed out, that a correction proposal is only counted as correct, if its spelling matches exactly. For example, singular and plural word forms are differentiated. The proposal for the misspelled word *lcvels* has to be *levels* and not *level*.

For each token the following choices had to be made:

Token is correct. This option should be selected if the token was correctly spelled.

Don't know. If the assessor was not sure what the correction should be, this option had to be chosen.

No matches. This option should be chosen if none of the displayed proposals matched (for each proposal list).

Select proposals. If none of the above choices were made a proposal for Aspell and OCRpp should be selected.

In order to enable a proper evaluation, the OCR Correction GUI has the following features:

1. Aspell and OCRpp proposals are separately displayed in two proposal lists. Not only the proposals in the lists are randomly ordered but also the separated lists. That means the assessor does not know in which list which proposals are displayed and which are the most likely to be correct.
2. The assessor always has to make a proper choice in order to continue choosing proposals.

5.2.4 Measuring the Performance

The task of an automatic spell checking and correction system (henceforth: SCC) is to detect and correct misspelled words. In general, four possible alternatives can occur when applying an SCC:

1. correct → correct: A correct word form is still correct after applying an SCC. This is a true negative (TN).
2. correct → wrong: A correct word form word is "corrected" to a wrongly spelled word after applying an SCC. This is a false positive (FP).
3. wrong → correct: A wrongly spelled word is corrected by an SCC. This is a true positive (TP).
4. wrong → wrong: A wrongly spelled word is still wrong after applying an SCC. This is a false negative (FN).

From the FP, TP and FN the measures precision and recall can be derived and combined to the harmonic mean, the F-Score [MS03, pages 268-269].

$$\text{Recall} = R = \frac{TP}{TP + FN} \tag{5.2}$$

$$\text{Precision} = P = \frac{TP}{TP + FP} \tag{5.3}$$

$$\text{F}-\text{Score} = F = \frac{2 \cdot R \cdot P}{R + P} \tag{5.4}$$

Using these measures for evaluating an SCC has been proposed and discussed by Reynaert [Rey08a] and are used here. The recall measures the ability of an SCC to correct errors. More corrections (TP) and fewer false corrections (FN) increase the recall. The precision measures the ability of correcting errors only (distinguish between correctly and wrongly spelled tokens). More corrections (TP) and fewer introduced errors (FP) increase the precision. The F-Score combines both measures to one.

Because Aspell and OCRpp propose five correction proposals a distinction between the 1-best proposal and the 5-best proposal can be made. Using only the 1-best correction proposal gives insights on the performance of the system if used fully-automatically. Using the 5-best correction proposals enables to measure the

potential of the approach and the room for improvement with respect to the proposal ranking system.

In order to apply the stated measures, the amount of TP, FP and FN have to be determined. Because this approach is interested in correcting words, the number of involved words is essential for counting TP, FP and FN. That means, each word which is involved in a correction proposal has to be counted. The same counting as in Section 5.2.2 is applied.

5-best Evaluation

For determining the number of TP the selected proposals in the OCR Correction GUI are examined. The 5-best evaluation requires only one selected proposal among the 5 proposed corrections.

For unification, Aspell uses the heuristic for merging hyphenation errors which belongs to OCRpp. By default Aspell does not have the ability to merge tokens. Therefore, the number of TP which Aspell achieves using the heuristic are subtracted from Aspell's TP count.

The number of FP can be determined using tokens which are marked as correct. If no proposal of such a token equals the original token in the text, then this is a FP for the 5-best evaluation.

Because Aspell and OCRpp do not tackle all errors, the number of FN can only be computed by using the counted errors. The sum of TP and FN equals the total number of errors. Hence, the number of FN can be computed as follows:

$$FN = \#errors - TP \tag{5.5}$$

1-best Evaluation

For the 1-best evaluation the counting of TP and FP is done by considering only the top ranked correction proposal in the Correction GUI. The FN count stays the same for the 1-best evaluation.

Year	#token	#errors	error rate	cat. deviation [percentage points] all	news
1835	2045	505	24.7	−7.8	1.2
1838	2075	407	19.6	−12.1	−1.6
1885	2028	145	7.1	−7.8	−5.0
1888	2032	231	11.4	−19.8	−3.3
1935	2047	68	3.3	−15.4	−6.3
1938	2058	81	3.9	−22.6	−6.5
1982	2038	258	12.7	−7.3	1.3
1985	2067	152	7.4	−14.1	−4.1
$\sum$	16390	1847	11.3	−13.4	−3.0

Table 5.2: Statistics for Chosen Samples for Evaluation: Number of tokens, errors and error rate in comparison with the mean Aspell error rate for all categories and the news category

5.3 Evaluation Results

This section presents the evaluation results. These results give insights about the performance of OCRpp in comparison to Aspell. An overview of the chosen samples is given first. Furthermore, the impact of DON'T KNOWS and hyphenation errors is examined. An error reduction rate is defined in Section 5.3.5 which considers corrected and introduced errors. Randomly chosen example correction of OCRpp are given in Section 5.3.6

5.3.1 Chosen Samples Overview

Some statistics about the chosen samples used for evaluation can be found in Table 5.2. The average token count lays slightly above 2000 per year as desired in the choosing process. The total number of evaluated tokens is 16390. The total amount of counted errors is 1847. This leads to an error rate of 11.3%. This error rate can be compared to the dictionary recognition rate using the Aspell dictionary. The error rate of the chosen samples is 13.4 percent points higher than the average error rate of all categories and 3 percentage points higher than the

average error rate of the news category. The fact, that fewer errors occur in the chosen documents can be caused by four factors:

1. Human assessors did not find all errors.
2. The chosen samples have fewer errors.
3. The Aspell dictionary does not contain all correct words.
4. Human assessors counted errors in a different way than it is done using the dictionary recognition rate. For example the number of errors for wrongly segmented tokens are counted differently. Hence, the dictionary recognition rate is not very exact for measuring the amount of errors.

Though all four factors are possible, the third and fourth factor are most plausible for the observed difference in error rates.

Using a student t-test with $\alpha = 5\%$ shows that the mean number of errors in the years 1935 and 1938 is lower than the mean number of errors in the remaining years.

5.3.2 Performance of OCRpp and Aspell

As described in Section 5.2.4, the performance of OCRpp and Aspell can be measured using recall, precision and F-Score.

Figure 5.6 illustrates the performance by computing the F-Score. Using the 1-best proposal, OCRpp achieved an F-Score ranging from about 57% to 88%, whereas Aspell ranges from 24% to 49%. That shows that Aspell is vastly outperformed by OCRpp for correcting OCR errors.

On average, the 1-best F-Score is about 33% for Aspell and about 72% for OCRpp. That means OCRpp performs more than twice as good as Aspell for the top ranked proposal.

With respect to the 5-best evaluation, the F-Scores are higher for both systems. The average F-Score rises from 33% to 51% for Aspell and from 72% to 79% for OCRpp. Obviously, Aspell's performance increases more than the performance of OCRpp which indicates, that the ranking system of Aspell is not as well adapted to OCR errors as the ranking system of OCRpp.

The 5-best OCRpp F-Score is on average 7 percent points above the F-Score for 1-best OCRpp. This shows that the ranking system for OCRpp has room for improvements.

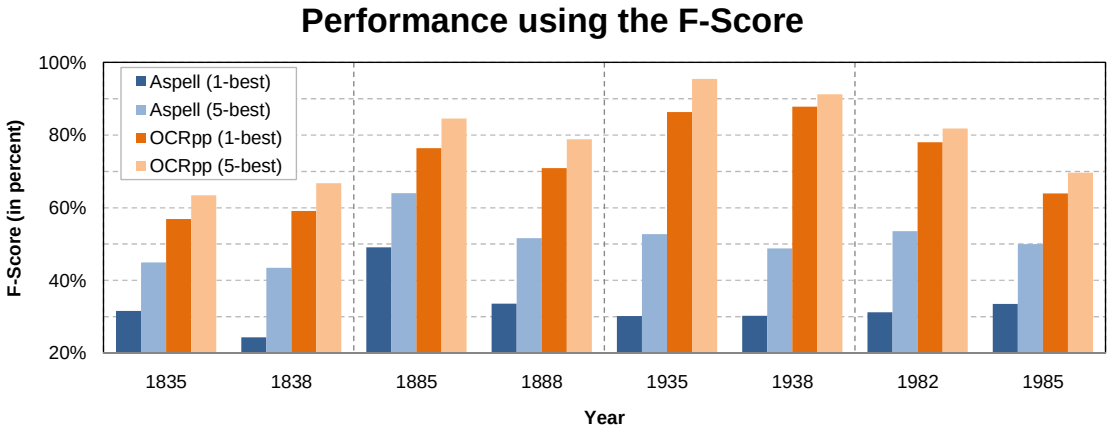

Figure 5.6: Evaluation F-Score: Comparison between Aspell and OCRpp with respect to 1-best and 5-best proposals.

Comparing the 1-best OCRpp and 5-best Aspell F-Score shows that Aspell is outperformed in all years, even with five proposals, compared to OCRpp's top ranked proposal.

In years with more errors, both systems have a lower F-Score than in years with fewer errors. This is mainly caused by a high DON'T KNOW rate which decreases the amount of possible TP and therefore increases the amount of FN (see Formula 5.5). Hence, a high DON'T KNOW rate hinders a proper evaluation.

Although the OCR error rate in the archive decreases over time, the F-Score in the years 1983 and 1985 is lower than in the years 1935 and 1938. This is caused by the high error rate of the chosen samples and their contained DON'T KNOWS (Section 5.3.1). Note, that the quality of chosen articles is very high in 1935 and 1938 (Section 5.3.1).

Detailed information about precision, recall and F-Score can be found in Table A.1, Table A.2 and Table A.3 in the appendix. One can see that the precision is significantly higher than the recall. This is clear, because the high number of DON'T KNOWS increases the number of FN which decreases the recall. Furthermore, short words which contain errors are not considered by the correction methods and increase the number of FN additionally.

The average precision of OCRpp using the 1-best proposal is about 92%, whereas the precision of Aspell is 72%. This indicates, that only few errors are introduced by OCRpp but far more by Aspell. A reason for this behaviour of OCRpp can

	Year	Destroyed	Names	Language	$\sum$	Percentage [%]
	1835	146	41	13	200	40.6
	1838	48	13	22	83	23.7
	1885	12	13	9	34	19.9
	1888	42	15	15	72	29.9
	1935	1	11	2	14	13.2
	1938	1	11	1	13	10.9
	1982	27	18	3	48	17.0
	1985	22	11	2	35	24.0
$\sum$		299	133	67	499	26.1

Table 5.3: Categorization of DON'T KNOWS found in chosen samples into destroyed, names and language. The percentage is the fraction of DON'T KNOWS and detected tokens.

be the usage of an archive derived dictionary which contains names and domain specific words.

The average recall of OCRpp using the 1-best proposal is about 62% and is nearly three times as high as 22% for Aspell. This indicates the ability of actually correcting errors. Because of the high DON'T KNOW rate this value is likely higher for both than shown by the recall.

Giving Aspell the ability to correct hyphenation errors using the heuristic of OCRpp increases the F-Scores of Aspell. Still the 5-best F-Score is still lower than the 1-best F-Score of OCRpp. That means, OCRpp outperforms an enhanced version of Aspell using the top ranked proposal. This behaviour of Aspell can be seen in Figure A.1 (page 100).

5.3.3 Impact of don't knows

It is very difficult to measure the performance of correction systems if human assessor do not know what the proper correction of a misspelled word is. Therefore, the assessors were told to mark tokens as DON'T KNOW if they do not know the proper correction. After evaluation occurring DON'T KNOWS were classified into three categories:

1. Totally destroyed tokens. The token cannot be restored to a proper word even with the help of the context.
2. Names. The assessor is not sure if the name is spelled correctly.
3. Language problems. The assessor does not know enough about the language. Language problems mainly occur if out-dated terms from earlier periods or domain specific terms are used.

Table 5.3 shows which DON'T KNOW types are contained in the chosen samples for each year. In the earlier years, language problems occur more often than in the later years. Unknown names are distributed equally over the years, except in 1835. The articles from 1835 contain many names of persons, cities and countries. A relation between error rate (see Table 5.2) and occurring destroyed tokens can be drawn. In years with a high error rate the fraction of destroyed DON'T KNOWS is higher than in years with a lower error rate.

About 3% of all tokens and 26% of all detected tokens are marked as DON'T KNOWS. Detected tokens are those, for which Aspell or OCRpp retrieved correction proposals and the assessor had to make a choice. More than the half of DON'T KNOWS are destroyed tokens. That means about one fourth of detected errors have no chance to be categorized as TP in the evaluation. This is the main reason why the evaluation gives bad F-Scores in years with high DON'T KNOW rates.

5.3.4 Impact of Hyphenation Errors

Hyphenation errors are those errors which occur if a token is too long for a line and has to be split up (see Section 1.3 for definition). In order to evaluate the heuristic which merges possible hyphenation errors, two evaluations steps are necessary.

1. Evaluate *correctly-merged* rate:
   Two tokens can be merged correctly, although they are not spelled correctly. This is the most important indicator for evaluating the heuristic.
2. Evaluate *correctly-corrected* rate. After merging two tokens, the merged token is tackled by the correction algorithm. Misspelled tokens may get corrected. It could also happen that wrongly merged tokens may get slit and corrected in the correction step.

Tokens which were merged by the heuristic were categorized into wrongly and correctly merged tokens. Table 5.4 shows amongst others the ratio of correctly merged hyphenation errors per evaluated year. It can be seen, that the ratio is

Year	#merged	correctly merged [%]	merge (1-best) #TP	merge (1-best) rate [%]	merge (5-best) #TP	merge (5-best) rate [%]
1835	29	100.0	25	86.2	26	89.7
1838	26	92.3	19	73.1	19	73.1
1885	18	100.0	17	94.4	17	94.4
1888	21	90.5	19	90.5	21	100.0
1935	26	84.6	25	96.2	25	96.2
1938	30	100.0	30	100.0	30	100.0
1982	52	94.2	50	96.2	51	98.1
1985	25	84.0	20	80.0	20	80.0
$\sum$	227	93.4	205	90.3	209	92.1

Table 5.4: Merged Hyphenation Errors. Ratio of correctly merged tokens and correctly corrected merged tokens.

higher than 90% in most years and even reaches 100% in three years. Only two years have a ratio around 84%.

The hyphenation heuristic produced 227 merged tokens over all evaluated years. Out of these, 205 tokens could be corrected using the 1-best correction proposal which constitutes about 90%. Using the 5-best correction proposals, 209 tokens could be corrected, which amounts to 92%.

Wrongly merged tokens are mainly caused by words which are naturally spelled with an hyphen but merged, due to a line-break, e.g. *full \n scale* $\rightarrow$ *fullscale*. It should be pointed out that OCRpp can correct words which were wrongly merged by the heuristic. This can be seen in the year 1935 where the correctly merged ratio is around 84%, whereas the correction rate is about 96%. That means all but one merged token could be corrected properly.

5.3.5 Error Reduction Rate

Measuring the system using precision, recall and F-Score gives insights about strengths and weaknesses of correction systems but does not show the practical usage. One valuable question to answer is, "how many errors can be reduced when applying the correction algorithm".

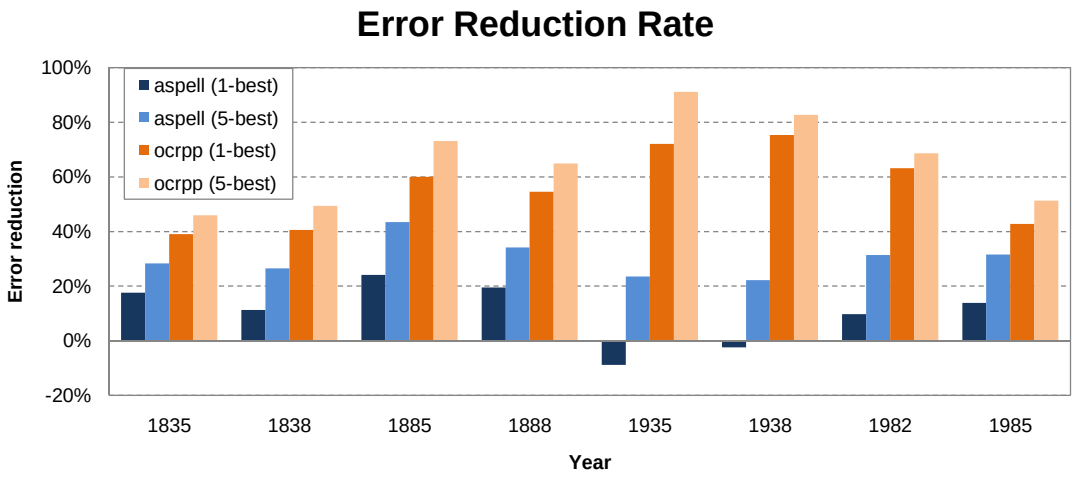

Figure 5.7: Error Reduction Rate: Using Aspell and OCRpp 1-best and 5-best correction proposals.

Measuring an error reduction rate for OCRpp can be achieved by computing the following fraction:

$$r_{ocrpp} = \frac{\#corrected - \#introduced}{\#errors} \tag{5.6}$$

The number of corrected tokens are all TP. The number of introduced errors equals the FP. The number of errors are the counted errors by the assessors.

Because Aspell is not able to merge hyphenation errors by default, the number of TP which Aspell produces using the hyphenation heuristic introduced in OCRpp is subtracted from Aspell's TP. Formally:

$$r_{aspell} = \frac{\#corrected - \#introduced - \#TP_{heuristic}}{\#errors} \tag{5.7}$$

The error reduction rate is depicted in Figure 5.7 for Aspell and OCRpp. The error reduction rate for the 1-best correction proposal ranges from 39% to 75% for OCRpp and −9% to 24% for Aspell. That means in the years 1935 and 1938 Aspell introduces more errors than it corrects. In contrast, OCRpp performs best in those years. This is because about one third of occurring errors are hyphenation errors. These errors can be handled well by OCRpp.

The error reduction rate for both systems is better, when considering the 5-best correction proposals. The average error reduction rate rises from about 50% to 58% for OCRpp, and from about 14% to 30% for Aspell. The big difference between

	1-best				5-best			
	Aspell		OCRpp		Aspell		OCRpp	
Year	TP	FP	TP	FP	TP	FP	TP	FP
1835	96	7	203	6	148	5	237	5
1838	58	12	175	10	115	7	207	6
1885	53	18	94	7	73	10	107	1
1888	47	2	128	2	81	2	151	1
1935	16	22	60	11	29	13	63	1
1938	18	20	72	11	30	12	73	6
1982	53	28	169	6	102	21	182	5
1985	33	12	77	12	52	4	85	7
$\sum$	374	121	978	65	630	74	1105	32

Table 5.5: Number of TP and FP (1-best) of Aspell and OCRpp on chosen samples.

1-best and 5-best indicates again, that Aspell's proposal ranking system is not appropriate for correcting OCR errors. The difference between 1-best and 5-best is not as big as for OCRpp but large enough to indicate that the ranking system of OCRpp can and should be improved.

The best error reduction rates are archived using OCRpp and the 5-best proposals with about 91% error reduction in 1935. The best value for 5-best Aspell is 43% in the year 1885.

Note, that Figure 5.7 only considers corrected and introduced errors. Errors which are present and corrected wrongly are not considered.

In Table 5.5 TP and FP are given for Aspell and OCRpp with respect to 1-best and 5-best evaluation. It can be seen, that the number of FP for OCRpp is on average lower than for Aspell. This can be explained by the usage of a corpus derived dictionary. More names and domain specific words can be detected using such a dictionary.

If Aspell uses the hyphenation heuristic of OCRpp, the error reduction rate of Aspell increases. In Figure A.2 (page 100, appendix) the increased error reduction rate is depicted. Nevertheless, 1-best OCRpp is still better than the improved 5-best Aspell, except in 1985 where both achieve equal correction rates. This shows

again the impact of hyphenation errors which increase the error reduction rate of Aspell by about 10 percentage points.

5.3.6 Sample Correction Proposals of OCRpp

Some sample correction proposals are given in the following for TP, FP, DON'T KNOWs and FN.

Table A.4 lists 20 randomly chosen tokens which could be corrected using the first best correction proposal of OCRpp (TP). It can be seen that simple character misrecognitions, hyphenation errors and segmentation errors can be corrected using OCRpp.

Ten randomly chosen sample tokens which are marked as correct but wrongly corrected by applying the top ranked correction proposal of OCRpp can be found in Table A.5 (FP). For example the name *Pattinson* is marked to be a correct word. But using the first best correction proposal of OCRpp mis-corrects the word. Therefore, the ignore word rule was introduced which would not correct the word, because the second best correction proposal equals the original string. But for evaluation, this rule was deactivated for having more insights and for being comparable to Aspell. FP are mainly caused by rare words which are removed when pruning the occurrence maps. This indicates again, the need for a better pruning strategy which also respects rare words.

15 randomly chosen DON'T KNOWs are listed in Table A.6 with corresponding correction proposals generated by OCRpp. For each DON'T KNOW category five samples are listed. For example the name *Niorrison* could be a misspelling. The top ranked proposal of OCRpp is *Morrison* which sounds more familiar. But also *Niorrison* is proposed. However, it is a DON'T KNOW because it is not clear what is correct. For destroyed tokens it is even harder to make a correct guess. For example *orry* could be anything, even with context information such an error is hardly to correct. But the word *rcla4ses* is probably the word *relapses* which is the second best correction proposal of OCRpp. That means, not all destroyed DON'T KNOWs are necessarily destroyed, although human assessors marked them. For correcting language problems more informations about the domain or language are needed in general. For example, the word *gerumin* is likely neither destroyed nor a name.

24 tokens which could not be corrected using the first correction proposals of OCRpp are listed in Table A.7, 3 for each evaluated year (FN). It can be seen, that the quality of correction proposals in earlier years is worse than in later years. This is caused by a high amount of OCR errors contained in the anagram-hash map.

Reducing OCR errors in the map using a better pruning strategy can enhance the results. It can be seen that some FN are only FN for the top ranked correction proposal, e.g. *Recretary* which should be Secretary using the second best proposal. Some tokens are not corrected using the top ranked correction proposal, e.g. *beten*. That means, the error is still in the document but not corrected to a different wrong word. FN are also caused by the restricted character set of The Times Archive as described in Section 5.1.3, e.g. the German word *B&rsen-Zeitung* lacks of the character ö instead of *&*.

Among all proposed correction proposals it can be seen, that OCRpp proposes many error-prone words. They are sometimes close to a correct word form but still not correct. This indicates, that the derived dictionaries from the archive needs to be enhanced by filtering out OCR errors in order to achieve better corrections, e.g. by applying a better pruning strategy of OMs.

5.4 Discussion

This chapter described the evaluation method and gave the results of the evaluation. A 1-best and 5-best evaluation was performed with respect to all errors contained in the chosen documents. The different results showed the importance of examining spell checking and correction systems from different angles. The average F-Score for OCRpp with 72% is more than twice as high than for Aspell for the 1-best evaluation. The F-Score using the 5-best correction proposals increases the F-Score to 79% for OCRpp and 51% for Aspell. An error reduction rate up to 75% could be achieved using OCRpp which is nearly three times as high as the best rate for Aspell.

One difficulty of the evaluation was the fact that human assessors evaluated the generated correction proposals. This introduced an uncertainty about the evaluation results. Due to the amount of errors in the older documents, it is not clear if these errors were counted properly. Distinguishing between errors and counting them is a hard task with respect to the used counting guidelines. Furthermore, different correctors behave differently. Some marked names more often as DON'T KNOWS than others did. Some had a better knowledge about the used language and could therefore correct errors better. But all in all the DON'T KNOW rate of about 26% is very high and worsen the evaluation results.

The high DON'T KNOW rate also has an impact on the error reduction rate. The overall error reduction rate is about 50% for OCRpp and 14% for Aspell, where the

DON'T KNOW rate is about 26%. Hence, a higher reduction rate can be expected if fewer DON'T KNOWS occur.

It can be seen in Figure 5.6 and Figure 5.7 that the 5-best OCRpp evaluation achieves better results than the 1-best OCRpp evaluation. That means, the ranking system has room and need for improvements. Hence, investigating and improving the ranking system is one possible way to improve the performance of OCRpp.

Comparison with Related Work

A comparison with related work is difficult because of the quality differences of datasets and different evaluation methods. Still some rough comparisons can be given.

Tong and Evans [TE96] state that their approach for correcting OCR errors is able to reduce about 60% of all errors. That is more than the average error reduction rate of OCRpp but less than the highest achieved rates of OCRpp on higher quality sample documents.

Kolak and Resnik [KBR03] stated an error reduction up to 80% using the word error rate (WER) as measure. This measure states how near a correction is in relation to the correct word using the Levenshtein-Distance. Unfortunately, this cannot be measured using the proposed evaluation method in this thesis, because the WER requires all correct and error-prone word pairs. But it is likely that OCRpp has an higher error reduction rate using the WER as measure than the absolute numbers of corrected words, because generated correction proposals are often close to the correct word.

OCRpp behaves worse than highly domain adapted approaches which consider only single misspelled words. For example Perez-Cortes et al. [PCAAL00] achieve error reduction rates up to 95% when correcting the OCR output of handwritten names. This can be explained by training a high amount of domain specific data and using "complete" dictionaries. Another approach which is only applied on single tokens is for example Reynaert [Rey06] which claimed cumulative F-Scores up to 95% for Levenshtein-Distances up to two. It is not clear how both methods behave on contiguous text.

6 Conclusion and Future Work

6.1 Conclusion

This thesis presented an unsupervised post-correction system for OCR errors which uses mainly the archive itself for correcting OCR errors. The thesis showed that archive derived frequency information about words and word pairs are useful for correcting OCR errors. High frequent words can be used as correction proposals and enable a correction of names and domain specific words. Removing infrequent tokens from the collected tokens was solved using a simple pruning strategy. All tokens which occurred less than a certain number were removed.

The proposed correction strategy achieved error reduction rates up to 75% on test samples using only the first best correction proposal. Considering all sample documents, the error reduction rate was on average 50%. Considering, that about 26% of all detected tokens could not be evaluated because they were marked as DON'T KNOW, this is a relative good value. The high precision of about 92% indicates that the algorithm introduced only few new errors (FP).

For correcting hyphenation errors, a heuristic was applied. The heuristic used the positions of tokens in the original scanned documents. More than 93% of tokens of all sample documents could be merged correctly.

Due to the size of many archives, it is necessary to perform a clean-up fully-automatically. In order to finish this task in a meaningful amount of time, the proposed system is a trade-off between generating high qualitative correction proposals and computational effort. For example a collection based dictionary was used to speed up the system by filtering out words which are likely correctly spelled and should not be corrected. Short words were also not corrected. Both leads to a decreased number of corrected words.

Although OCRpp achieves relatively high F-Scores, the amount of introduced and wrongly corrected errors is too large for using OCRpp fully-automatically. Instead the 5-best OCRpp method could be used semi-automatically.

The newly proposed OCR-Key method is especially useful for tackling systematic OCR errors, where the Anagram Hash method is able to tackle all kind of errors. Using both methods in combination enabled a better ranking of generated correction proposals. Taking the context into account by using collected bigrams from the archive enhanced the ranking of correction proposals further.

The strong Levenshtein-Distance limitation of both systems showed advantages and disadvantages. The limit hindered the retrieval of words which are less likely to be correct. Rare tokens could therefore be kept which were otherwise wrongly corrected. But on the other hand, fewer correction proposals could be proposed. Especially the OCR-Key method could not correct words for which it was originally designed for, e.g. *minimiiumii* $\rightarrow$ *minimum*.

6.2 Future Work

In contrast to related works, this thesis used no ground-truth material for training special algorithms. Nevertheless, the system showed a good performance. Therefore, the proposed correction method could use generated correction proposals to train weights for a weighted Levenshtein-Distance which can improve the retrieval and ranking of adequate correction proposals.

The used Levenshtein-Distance limit could be increased to retrieve more words. The current limitation hinders the retrieval of correctly spelled tokens which are more damaged. Furthermore, the Anagram Hash Method could be used using a more sophisticated algorithm which is able to insert, delete or substitute more than character bigrams. Although this increases the computational effort, it could be worth to achieve better results.

One important error which was only tackled partly in this thesis is segmentation errors. Two tokens can be merged at line breaks (hyphenation errors) but not in a sentence, e.g. *de part ment* $\rightarrow$ *department*. Tokens can only be split up at most into two tokens using the Anagram Hash and bigrams. Therefore, a more sophisticated splitting and merging strategy could be added in future work.

For ranking correction proposals a simple scoring system was proposed. The system worked well in many cases but has room for improvement. The combination of many different factors in different computation steps is difficult to handle. For example the weighting of methods is difficult. Therefore, each method should assign a normalized score to its generated correction proposals which can be combined using weights at the end of the process. The improvements of the scoring system including the search for optimal weights is left as future work.

Bibliography

[ALN07] Farag Ahmed, Ernesto William De Luca, and Andreas Nürnberger. MultiSpell: an N-Gram Based Language-Independent Spell Checker. In *Proceedings of Eighth International Conference on Intelligent Text Processing and Computational Linguistics (CICLing-2007)*, Mexico City, Mexico, 2007.

[Atk08] Kevin Atkinson. GNU Aspell version 0.60.6, Released under the GNU LGPL license in April, 2008. http://aspell.net/.

[BM00] Eric Brill and Robert C. Moore. An Improved Error Model for Noisy Channel Spelling Correction. In *ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 286–293, Morristown, NJ, USA, 2000. Association for Computational Linguistics.

[Dam64] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964.

[FDE07] Davide Fossati and Barbara Di Eugenio. A Mixed Trigrams Approach for Context Sensitive Spell Checking. In *CICLing '07: Proceedings of the 8th International Conference on Computational Linguistics and Intelligent Text Processing*, pages 623–633, Berlin, Heidelberg, 2007. Springer-Verlag.

[FE08] D. Fossati and B. Di Eugenio. I saw TREE trees in the park: How to correct real-word spelling mistakes. In *Proceedings of the 6th International Conference on Language Resources and Evaluation*, pages 896–901, 2008.

[FJ73] G. D. Forney Jr. The Viterbi Algorithm. In *Proceedings of the IEEE*, volume 61, pages 268–278, March 1973.

[GBJS04] Filip Ginter, Jorma Boberg, Jouni Järvinen, and Tapio Salakoski. New Techniques for Disambiguation in Natural Language and Their Application to Biological Text. *J. Mach. Learn. Res.*, 5:605–621, 2004.

[Goo] Google Web 1T Data. http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006T13.

[Hau07] Andreas W. Hauser. OCR-Postcorrection of Historical Texts. Master's thesis, University of Munich, 2007.

[HDH+95] Tao Hong, Chairman Dr, Jonathan J. Hull, Members Dr, Sargur N. Srihari, Dr. Deborah, K. Walters, Outside Reader, Dr. Henry, and S. Baird. Degraded Text Recognition Using Visual And Linguistic Context, 1995.

[HS07] Andreas W. Hauser and Klaus U. Schulz. Unsupervised Learning of Edit Distance Weights for Retrieving Historical Spelling Variations. In *Proceedings of the First Workshop on Finite-State Techniques and Approximate Search*, pages 1–6, Borovets, Bulgaria, 2007.

[Idz10] Mindaugas Idzelis. Open Source Spell Checker Jazzy version 0.5.2, 2010. http://jazzy.sourceforge.net/.

[II09] A. Islam and D. Inkpen. Real-word spelling correction using Google Web 1T 3-grams. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 1241–1249, 2009.

[KBR03] Okan Kolak, Willian Byrne, and Philip Resnik. A Generative Probabilistic OCR Model for NLP Applications. In *HLT-NAACL*, pages 55–62, 2003.

[KR05] Okan Kolak and Philip Resnik. OCR Post-Processing for Low Density Languages. In *HLT '05: Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 867–874, Morristown, NJ, USA, 2005. Association for Computational Linguistics.

[Kuk92] Karen Kukich. Techniques for Automatically Correcting Words in Text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[Lev66] V.I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviets Physics Doklady*, 10(8):707–710, 1966.

[MDM91] Eric Mays, Fred J. Damerau, and Robert L. Mercer. Context Based Spelling Correction. *Inf. Process. Manage.*, 27(5):517–522, 1991.

[MG96] Andreas Myka and Ulrich Güntzer. Fuzzy Full-Text Searches in OCR Databases. In *ADL '95: Selected Papers from the Digital Libraries, Research and Technology Advances*, pages 131–145, London, UK, 1996. Springer-Verlag.

[MS03] Christopher D. Manning and Hinrich Schuetze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 6 edition, 2003.

[OM18] Rusell R.C. Odell M.K. Patent Numbers, 1,261,167 (1918) and 1,435,663 (1922). *U.S. Patent Office*, 1918.

[Ope10] OpenOffice.org 3.2, Released under the LGPL license in February, 2010. http://openoffice.org/.

[PCAAL00] Juan C. Perez-Cortes, Juan C. Amengual, Joaquim Arlandis, and Rafael Llobet. Stochastic Error-Correcting Parsing for OCR Post-Processing. In *ICPR '00: Proceedings of the International Conference on Pattern Recognition*, page 4405, Washington, DC, USA, 2000. IEEE Computer Society.

[Phi00] Lawrence Philips. The Double Metaphone Search Algorithm. *C/C++ Users J.*, 18(6):38–43, 2000.

[PZ84] Joseph J. Pollock and Antonio Zamora. Automatic Spelling Correction in Scientific and Scholarly Text. *Commun. ACM*, 27(4):358–368, 1984.

[Rey04] Martin Reynaert. Text Induced Spelling Correction. In *COLING '04: Proceedings of the 20th international conference on Computational Linguistics*, page 834, Morristown, NJ, USA, 2004. Association for Computational Linguistics.

[Rey06] Martin Reynaert. Corpus-Induced Corpus Clean-up. In *LREC 2006: Fifth International Conference on Language Resources and Evaluation*, 2006.

[Rey08a] Martin Reynaert. All, and only, the Errors: more Complete and Consistent Spelling and OCR-Error Correction Evaluation. In *6th International Conference on Language Resources and Evaluation*, pages 1867–1872, 2008.

[Rey08b] Martin Reynaert. Non-interactive OCR Post-correction for Giga-Scale Digitization Projects. In *Computational Linguistics and Intelligent Text Processing*, pages 617–630, 2008.

[SL07] Johannes Schaback and Fang Li. Multi-Level Feature Extraction for Spelling Correction. In *IJCAI-2007 Workshop on Analytics for Noisy Unstructured Text Data*, pages 79–86, Hyderabad, India, 2007.

[Str04] Christian M. Strohmaier. Methoden der lexikalischen Nachkorrektur OCR-erfasster Dokumente, 2004.

[Tah09] Nina Tahmasebi. Automatic Detection of Terminology Evolution. In Robert Meersman, Pilar Herrero, and Tharam S. Dillon, editors, *OTM Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 769–778. Springer, 2009.

[TE96] Xian Tong and David A. Evans. A Statistical Approach to Automatic OCR Error Correction In Context. In *Proceedings of the Fourth Workshop on Very Large Corpora (WVLC-4)*, pages 88–100, 1996.

[Tim] The Times of London. http://archive.timesonline.co.uk/tol/archive/.

[Tim14] The Times of London, November 29, 1814. http://archive.timesonline.co.uk/tol/viewArticle.arc?articleId=ARCHIVE-The_Times-1814-11-29-03-003&pageId=ARCHIVE-The_Times-1814-11-29-03.

[TNTR10] N. Tahmasebi, K. Niklas, T. Theuerkauf, and T. Risse. Using Word Sense Discrimination on Historic Document Collections. In *(to appear in) 10th ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, Surfers Paradise, Gold Coast, Australia, June 21–25, 2010.

[TS01] Kazem Taghva and Eric Stofsky. OCRSpell: an interactive spelling correction system for OCR errors in text. *International Journal of Document Analysis and Recognition*, 3:2001, 2001.

[WF74] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. *J. ACM*, 21(1):168–173, 1974.

[WOHB08] L. Amber Wilcox-O'Hearn, Graeme Hirst, and Alexander Budanitsky. Real-Word Spelling Correction with Trigrams: A Reconsideration of the Mays, Damerau, and Mercer Model. In *CICLing*, volume 4919 of *Lecture Notes in Computer Science*, pages 605–616. Springer, 2008.

[WRLM07] M. Wick, M. Ross, and E. Learned-Miller. Context-Sensitive Error Correction: Using Topic Models to Improve OCR. In *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*, pages 1168–1172, Washington, DC, USA, 2007. IEEE Computer Society.

A Appendix

Algorithm for computing the OCR-Key

Algorithm A.1 Computing the OCR-Key for a given word w

Require: Word w

```
// initialize
ocrKey ← null
lastRrepresentative ← null
counter ← 0

for i = 1 to |w| do
  representative ← classRepresentative(w_i)

  // character has no equivalence class
  if representative = null then
    continue
  end if

  // character of different equivalence class detected
  if representative ≠ lastRepresentative then
    if counter ≠ 0 then
      append counter as string to ocrKey
    end if
    counter ← 0
    append representative as string to ocrKey
  end if

  counter ← counter + classCardinality(w_i)
  lastRepresentative ← representative
end for

append counter as string to ocrKey
return ocrKey
```

Correction XML File Structure

General Structure

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<document>
  <articleid></articleid>
  <correction>
    <term pos="">
      <merge pos="" />
      <proposals>
        <proposal score=""></proposal>
      </proposals>
    </term>
  </correction>
</document>
```

Evaluation Example Proposals

```
<term pos="28">
  <merge pos="29" />
  <proposals>
    <proposal score="0.9086044">Tonypandy's</proposal>
    <proposal score="0.09178387">Tonypandy</proposal>
  </proposals>
  <aspell>
    <proposal score="1.0">Tenant's</proposal>
    <proposal score="2.0">Townsend's</proposal>
    <proposal score="3.0">Downland's</proposal>
    <proposal score="4.0">Tenpins's</proposal>
    <proposal score="5.0">Tenpin's</proposal>
  </aspell>
</term>
```

Documents Used for Evaluation

- ARCHIVE-The_Times-1835-01-15-07-015.xml
  ARCHIVE-The_Times-1835-02-13-03-005.xml
  ARCHIVE-The_Times-1835-04-20-01-004.xml
  ARCHIVE-The_Times-1835-04-30-05-013.xml
- ARCHIVE-The_Times-1838-02-08-04-007.xml
  ARCHIVE-The_Times-1838-02-26-03-006.xml
  ARCHIVE-The_Times-1838-09-14-03-008.xml
  ARCHIVE-The_Times-1838-10-19-04-013.xml
- ARCHIVE-The_Times-1885-02-06-05-013.xml
  ARCHIVE-The_Times-1885-05-29-05-014.xml
  ARCHIVE-The_Times-1885-08-27-04-006.xml
  ARCHIVE-The_Times-1885-12-21-05-002.xml
- ARCHIVE-The_Times-1888-02-14-04-014.xml
  ARCHIVE-The_Times-1888-08-04-05-012.xml
  ARCHIVE-The_Times-1888-08-14-05-017.xml
  ARCHIVE-The_Times-1888-08-20-06-012.xml
- ARCHIVE-The_Times-1935-02-12-14-002.xml
  ARCHIVE-The_Times-1935-03-13-08-001.xml
  ARCHIVE-The_Times-1935-06-01-17-010.xml
  ARCHIVE-The_Times-1935-12-14-07-011.xml
- ARCHIVE-The_Times-1938-07-08-08-001.xml
  ARCHIVE-The_Times-1938-10-14-14-002.xml
  ARCHIVE-The_Times-1938-12-10-12-017.xml
  ARCHIVE-The_Times-1938-12-12-08-006.xml
- ARCHIVE-The_Times-1982-01-19-01-003.xml
  ARCHIVE-The_Times-1982-02-19-06-002.xml
  ARCHIVE-The_Times-1982-07-15-06-002.xml
  ARCHIVE-The_Times-1982-10-05-26-001.xml
- ARCHIVE-The_Times-1985-01-18-15-005.xml
  ARCHIVE-The_Times-1985-02-04-12-001.xml
  ARCHIVE-The_Times-1985-09-19-14-006.xml
  ARCHIVE-The_Times-1985-09-23-04-005.xml

Detailed Evaluation Results

	1-best Aspell			1-best OCRpp		
Year	Recall	Precision	F-Score	Recall	Precision	F-Score
1835	19.0	93.2	31.6	40.2	97.1	56.9
1838	14.3	82.9	24.3	43.0	94.6	59.1
1885	36.6	74.6	49.1	64.8	93.1	76.4
1888	20.3	95.9	33.6	55.4	98.5	70.9
1935	23.5	42.1	30.2	88.2	84.5	86.3
1938	22.2	47.4	30.3	88.9	86.7	87.8
1982	20.5	65.4	31.3	65.5	96.6	78.1
1985	21.7	73.3	33.5	50.7	86.5	63.9
$\bar{x}$	22.3	71.9	33.0	62.1	92.2	72.4

Table A.1: Evaluation Results (1-best): Recall, precision and F-Score for Aspell and OCRpp using the 1-best correction proposal.

	5-best Aspell			5-best OCRpp		
Year	Recall	Precision	F-Score	Recall	Precision	F-Score
1835	29.3	96.7	45.0	46.9	97.9	63.5
1838	28.3	94.3	43.5	50.9	97.2	66.8
1885	50.3	88.0	64.0	73.8	99.1	84.6
1888	35.1	97.6	51.6	65.4	99.3	78.9
1935	42.6	69.0	52.7	92.6	98.4	95.5
1938	37.0	71.4	48.8	90.1	92.4	91.3
1982	39.5	82.9	53.5	70.5	97.3	81.8
1985	34.2	92.9	50.0	55.9	92.4	69.7
$\bar{x}$	37.1	86.6	51.1	68.3	96.8	79.0

Table A.2: Evaluation Results (5-best): Recall, precision and F-Score for Aspell and OCRpp using the 5-best correction proposals.

	1-best Aspell (strict)			5-best Aspell (strict)		
Year	Recall	Precision	F-Score	Recall	Precision	F-Score
1835	22.5	93.2	36.3	32.4	96.7	48.5
1838	16.7	82.9	27.8	30.6	94.3	46.2
1885	43.2	74.6	54.7	55.8	88.0	68.3
1888	26.1	95.9	41.0	40.0	97.6	56.7
1935	42.9	42.1	42.5	57.1	69.0	62.5
1938	42.2	47.4	44.6	53.2	71.4	61.0
1982	32.8	65.4	43.7	49.0	82.9	61.6
1985	29.6	73.3	42.2	40.8	92.9	56.7
$\bar{x}$	32.0	71.9	41.6	44.9	86.6	57.7

Table A.3: Evaluation Results for Aspell + Heuristic: Recall, precision and F-Score for Aspell using the hyphenation heuristic.

Sample Correction Proposals of OCRpp

original string	proposals by OCRpp
cxecutlon	[execution, cxecution, exccution]
renweed	[renewed, entered, rendered, renowned, we need]
c!ected	[elected, erected, ejected, clected, crected]
sinall	[small, snail, stall, sinall, shall]
repalrswas	[repeal was]
Parlialmet	[Parliament, Plarliament, Parliamnent, Parliaments, Parlianment]
AmertTan	[American, Ameriean, Aderman, tert an, age tman]
eari-est	[earliest, earnest, nearest, dearest, fairest]
foundtiton	[foundation, foundations]
retsoled	[resolved, resorted, restored, resulted, Rtesolved]
interpreters,and	[interpreter and]
com- btatits	[combatants]
aihould	[should, ahould, shouild, ihould, shiould]
charaactr	[character, charactcr, charactr, characetr, charatctr]
dc- lincated	[delineated]
lIumanity	[humanity, inhumanity, humnanity]
l:riend	[friend, friends, friendl, frilend, triend]
LTurope	[Europe, Troupe, Lurope, Letrop, urope]
generaUy	[generally, generaUy, generaly, generaUly, generaDy]
Coffee.-August	[coffee august]

Table A.4: Samples of true positives (TP) using the top ranked correction proposal of OCRpp.

original string	proposals by OCRpp
Pattinson	[Parkinson, Pattinson, Pattison, Pattisson, Mattinson]
Ruddock's	[Ruddock]
Anile	[Annie, Anne, Aline, Arnie, Lanie]
tuciled	[tickled, tucked, LUnited, quilted, LUmited]
bodyguarding	[bodyguard in, bodyguards in]
I'lle	[Lille, Elle, I'le, I'lie, Ille]
Canning-a	[Canning, cunning a]
EAton-street	[Dean-street, East-street, ton-street, Gate-street, Water-street]
peasticks	[east hicks, practices, pastimes, penaltics]
A.R.A.	[ARA]

Table A.5: Samples of false positives (FP) using OCRpp.

original string	proposals by OCRpp	type
Malet	[Market, Mallet, Malet, Mater, Motet]	name
GARGETI	[ARGENTI, gardeti, GARREIT, GARAGE, garagte]	name
Niorrison	[Morrison, Niorrison, niorris on, Miorrison, NMorrison]	name
Kurier	[Currie, Kuiper, Kourie, Trier, Kruger]	name
Rohinisoni	[Robinison]	name
orry	[only, sorry, orly, airy, arty]	destroyed
moeth	[mo the, month, mother, mothe, mouth]	destroyed
MuanUi	[Manu, Anum, Anui, Muan, Manual]	destroyed
ciqlue	[clique, Cinque, clue, calUed, cluse]	destroyed
rcla4ses	[clarges, relapses]	destroyed
desiat	[design, desist, Dasent, die at, desite]	language
protcisors	[protectors, proctors]	language
occultation	[occupation, consultation, occupations, occuipation]	language
bainbars	[brain as, in bars]	language
gerumin	[Germian, resuming, retuming, Germain, ferming]	language

Table A.6: Samples of marked DON'T KNOWSwhile evaluation, categorized into name, destroyed and language.

original string	proposals by OCRpp	year
beten	[beten, beent, betn, beaten, beetn]	1835
c4vil	[evil, civil, cvil, cavil, cvils]	1835
pre2ent	[prent, prevent, present, preent, pretend]	1835
mlonIths	[monthis, moniths, mionths, montlhs, monthls]	1838
prosecutoread	[prosecutor and, prosecutor had, prosecuter and, prosecutor he]	1838
Btelief	[Relief, Belief, Rtelief, beleft, but lie]	1838
lwhich	[wlhich, lwhich, whichl, whlich, whiclh]	1885
mnimbers	[numbers, mnembers, nuimbers, menibers, niembers]	1885
Inan7interview	[Inan7interview]	1885
Y09IK	[YORIK, YOIK, YOIRK, by ik]	1888
Decembar	[Decembar, Decamber, Decemaber, December, Decomber]	1888
otfcring	[coTering, of tring, storing, of ting, to ring]	1888
thic	[the, tic, thic, Eric, tihc]	1935
Recretary	[Sceretary, Secretary, ECRETARY, VECRETARY, CECRETARY]	1935
Ofliccrs	[Ofliccrs, Oflicers, Officcrs, Oflicccs, Olficers]	1935
Ms4r	[Mfr, Mst, MSr, Mrs, MSrs]	1938
iisttitLition	[Instittition]	1938
B&rsen-Zeitung	[B&rsen-Zeitung]	1938
univer	[driver, UNIVER, Univers, unider, under]	1982
otlihes	[outlines, hotlines, othiers, Hotline, otlhers]	1982
Palestiniianl	[palestinian ol, Palestiniani]	1982
lhand	[land, hand, Bland, brand, Handl]	1985
LonAon	[loan on, Nolan, Landon, Lonaon, London]	1985
Shodyguard	[Bodyguards, Bodyguard, body guard]	1985

Table A.7: Samples of false negatives (FN) with corresponding correction proposal of OCRpp.

Additional Diagrams for Evaluation

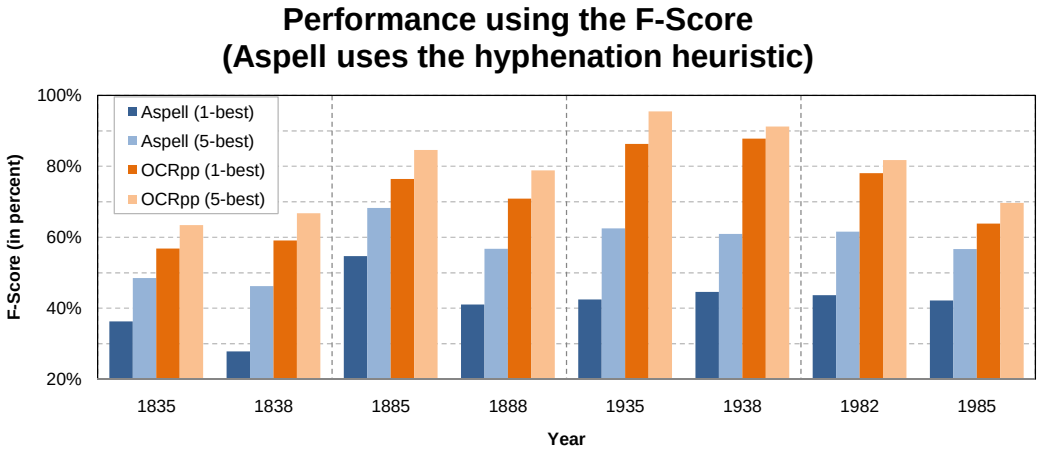

Figure A.1: F-Score: Comparison between Aspell and OCRpp with respect to 1-best and 5-best proposals. Aspell uses the hyphenation heuristic of OCRpp.

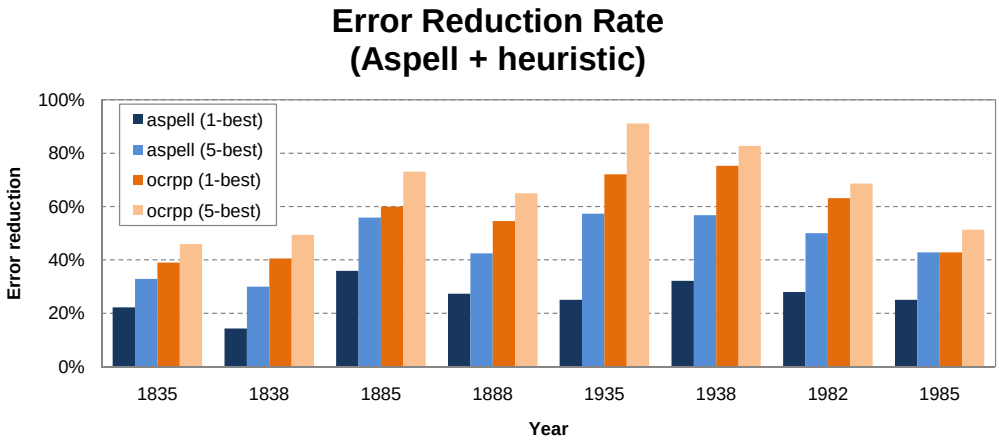

Figure A.2: Error Reduction Rate: Using Aspell and OCRpp 1-best and 5-best correction proposals. Aspell uses the hyphenation heuristic of OCRpp.

Acknowledgement

I would like to thank all persons who supported me during the time of my thesis. Special thanks go to my supervisor Nina Tahmasebi for her great support and spent time on helping me correcting and improving my written English. Great thanks go also to Thomas Theuerkauf who pushed me forward while writing his own diploma thesis at the same institute. He supported and motivated me and was always there as a friend. Furthermore, I would like to thank my assessors who participated in my evaluation. The biggest thanks go to my parents who made my studies possible and supported me during my whole studies.

We would like to thank Times Newspapers Limited for providing the archive of The Times for our research.

Thesis Declaration

I hereby certify that this thesis is my own and original work using the sources and methods stated therein.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe.

Kai Niklas Hannover, 11. Juni 2010