Third Year Project:
A context-sensitive spell checker using trigrams and confusion Sets

By
Tsz Ching Sam Chan

Supervisor: John Mcnaught

School of Computer Science,
University of Manchester

Submission Date: 2/5/2016
Table of Contents

1. Abstract
2. Introduction
   2.1 Definition
      2.1.1 A basic spell checker: Dictionary check
      2.1.2 Problem with dictionary check
   2.3 Context-Sensitive Spell Checker
   2.4 Project Goal
   2.5 Summary
3. Problem Analysis: Spelling error
   3.1 Keyboard Typing error (Typo)
   3.2 Phonetic Spelling
   3.3 Uncorrectable spelling error
   3.4 Summary
4. Background
   4.1 Machine learning approach
      4.1.1 Bayesian Hybrid
      4.1.2 Winnow-based
   4.2 N-gram approach
      4.2.1 5-gram approach
      4.2.2 Noisy Channel Model
   4.3 Summary
5. Design
   5.1 Approach on building context-sensitive spell checker
      5.1.1 Spelling Detection
         5.1.1.1 Non-word Spelling Error
         5.1.1.2 Real-word Spelling Error
      5.1.2 Spelling Correction
         5.1.2.1 Word distance to the original word
         5.1.2.2 Word permutation
         5.1.2.3 Word phoneme
         5.1.2.4 Additional confusion set
   5.2 Development
      5.2.1 GUI
      5.2.1.1 Requirement
      5.2.2 Development Process
      5.2.3 Programming Tool
      5.2.4 Software Architecture
         5.2.4.1 Model-View-Controller
         5.2.4.2 Design Model
   5.3 Summary
6. Implementation and Testing
6.1 Word Processing Tools
   6.1.1 DocumentPreprocessor
   6.1.2 PTBTokenizer
   6.1.3 MaxentTagger

6.2 Data Structure
   6.2.1 Storing dictionary and trigrams
       6.2.1.1 Dictionary
       6.2.1.2 Word trigram
       6.2.1.3 Mixed POS trigram
   6.2.2 Storing confusion sets
       6.2.1 Word distance and word permutation
       6.2.2 Word Phoneme and addition confusion sets
   6.2.3 Java data structure

6.3 Attributes and Methods

6.4 Testing

6.5 Summary

7. Evaluation of results
   7.1 User's Interface
   7.2 Performance
   7.3 Results
       7.3.1 Standards
           7.3.1.1 Spelling Detection
           7.3.1.2 Spelling Correction
       7.3.2 Test Data
       7.3.3 Final Result
   7.3 Summary

8. Conclusion and Reflection
   8.1 Outcome
   8.2 Development process
   8.3 Future Development

Appendix A - Levenshtein Distance Algorithm
Appendix B - Penn Treebank tag set
Appendix C - Phoneme and Self-defined Confusion Sets
Appendix D - Sample Database Structure
Appendix E - Confusion Matrices

Reference
Acknowledgement

I would like to express my deepest appreciation to all who provide me the possibility to complete the project. Special thanks to my supervisor, John Mcnaught who contribute the project by suggestions and encouragement, helping me to finish the project as well as this report. Furthermore a thanks to a PhD student in NLP and human computer interaction, Matthew Shardlow who provided me inspirations on initial project design and data source. Finally, I appreciate to the supervisor, Kung Kiu Lau who has given guidance in project presentation and demonstration, helping me to improve the presentation skills as well as the comments on my project.

1. Abstract

This project addresses on the problem occurs in modern spell checking: real-word error detection and suggestion. This project has taken a classic word trigram approach and a mixed part-of-speech trigram approach as real-word error detection method. The suggestion process is done by confusion sets constructed by phonemes, word distances, word permutation and some self-defined confusion sets.
2. Introduction

2.1 Definition

What is spell checking? Date back to 1980s, a spell checker is more like a “verifier”[1]. It has no corresponding suggestions to the spelling error detected. As many of the readers are using word processor nowadays, a spell checker will first mark a word as mistaken(Detection) and give a list of replacement of word(Suggestion). Therefore the definition of spell checking involve more than only checking, it is the process of detecting misspelled words in a document/sentence and suggest with a suitable word in the context. Therefore, to construct a spell checker, it needs to have the following features:

1. Spelling Detection: the ability to detect a word error
2. Spelling Suggestion(Correction): the ability to suggest a suitable word to users which matches their need in context

2.1.1 A basic spell checker: Dictionary check

In many classic approach, a spell checker implements a simple dictionary check structure. The overall implementation involves simple dictionary check. A diagram demonstrating this algorithm is showed in Figure 1.

This method is nice and easy, also requires a low level of programming. Developer can simply define a sets of dictionary words for spelling detection and suggestion. If an spelling error occurs, do binary search on the dictionary list and generates a number of corrections. To improve accuracy, simply expanding the dictionary size and it can detect more words.

![Diagram of simple lexicon spell checker](image)

**Fig. 1**: A diagram of simple lexicon spell checker
2.1.2 Problem with dictionary check

The major problem of the basic spell checker is about the spell detection stage. It is designed in the assumption that all word errors are the word that are NOT in the dictionary. These are classified as non-word spelling error. However, there are cases where spelling error is not simply a “spelling error”, imagine the following case:

\[ \text{I would like a peace of cake as desert.} \]

By simply looking at the words on the sentence above, all of them are fine in terms of spelling. However, errors still occur as the word “peace” and “desert” are not suitable to the context. They are called real-word spelling errors. In a spell checker that uses dictionary check, this kind of error will go undetected and proceed. It is clear that dictionary check is not a optimal spelling detection method.

In addition, there is a problem on spell suggestion as well. In the basic spell checker above, all the spelling suggestions are generated from a match to the dictionary. However, can all spelling suggestions be done by just analysing a single word? It is easier to think this problem with a case on search engines. In search engines, imagine a search on turkey is be processed. It is impossible for a search engine to classify if “Turkey - the animal” or “Turkey – the country” is intended. There is a problem derived: single word has insufficient power on searching. Similarly, in spell checking, a word on its own provide insufficient information for analysis. Look at the example below:

\[ \text{Is tehre a solution to tehre problem for when tehre travelling?} \]

Three non-word spelling errors occur in the context above. For a basic spell checker, those error will be classified correctly. However when user wanted to get a list of suggestions, the same suggestions will be given to all three words, regardless of the algorithm used in suggestion stage. Sometimes it will even suggest a wrong word to the context. Hence, matching to dictionary is not the best approach for a spelling correction.

Figure 2 shows how the above sentence applies to the Google spell checker.

![Fig. 2](image)

Fig. 2: an example of Google context sensitive spell checker. [2]
Because of the inability to detect a real-word error and to give a suitable suggestion by only analysing the word alone, more information are needed to tackle real-word spelling errors. Hence, an advanced spell checker often observes a group of contexts. It is also called a context-sensitive spell checker.

2.3 Context-Sensitive Spell Checker

In a context-sensitive spell checker, more information in the overall context are reviewed. It is often a sentence-by-sentence checking rather than a word-by-word checking in the checking stage. When generating suggestions, it will analyse the surrounding context and gives the most suitable words depending on the context instead of the lexicon. Suggestions are often generated from confusion sets, sets of words that are easy to confuse with one another. For example, {peace, piece} is a confusion set as they often confuse to each other. Different types of spell checker has different approach in context extraction and it affects the checking process and confusion set generation. More about those approaches will be mentioned in Chapter 4.

2.4 Project Goal

In this project, the main aim is to develop a context-sensitive spell checker to solve the real-word spelling errors. For real-word spell checking, the spell checker will take a mixed part-of-speech trigram approach for spelling detection and uses confusion sets for spelling corrections. The spell checker will also attempt to combine the base spell checker with the context-sensitive spell checker hence it has both non-word and real-word spell error detections along with corrections. More about the system design will be mentioned in Chapter 5. The performance will be compared with the Google context-sensitive spell checker to measure the outcome of the program, which will be mentioned in Chapter 7.

The ultimate goal of the project is to create a spell checker which can detect and suggest on all type of real-word error made. In the evaluation, the performance will be analysed by the following question:
1. In what extent the real-word typing errors were detected?
2. In what extent a suitable suggestion was given to an user for each type of spelling error?
3. In what extent the performance of the context-sensitive spell checker improved from the base spell checker

2.5 Summary

In this chapter, two kinds of spelling error are introduced and readers can understand why a basic spell checker cannot solve the real-word spelling error. Concept about context-sensitive spell checker is introduced and a glance on the project goal is described. In the next chapter, there will be an analysis on how a spelling error is caused by user.
3. Problem Analysis: Spelling error

Before mentioning the techniques of context-sensitive spell checker, it is crucial to analyse how a spelling error is made by users. A spelling error is often caused by human errors. It can be either a typing error (or in short, typo) or simply lack of knowledge of the correct spelling. Understanding how a spelling error occurs will be beneficial to design a suitable algorithm for a spell checker. In this chapter, each type of spelling error will be analysed and some simple solution will be accessed.

3.1 Keyboard Typing error (Typo)

In most computer application, word input is done by keyboard. When a spelling error occurs, keyboard is one of the major cause. Especially many documentation in office requires a fast typing speed, making typing error on keyboards become more common. The term **Fat Finger** refers to one of the common cause which make typing mistake in the keyboard typing process. It becomes a more common problem nowadays with touch screen on smart phones. Other cause might due to fast typing, causing some spellings are not checked before proceeding to the next word. The following shows some example taken from List of Common Misspellings published by Wikipedia[3].

1. "abd" (and)
2. "heigth" (height)
3. "higer" (higher)
4. "reccommend" (recommend)

In the first case, a word error occurs with a erroneous letter replaced to the expected letter. Note the letter b is adjacent to another letter n on the keyboard position, which is an example of the Fat Finger problem mentioned above. This is classified as a **substitution** error.

In the second one, two letters' position from the original word are changed. The letter h is nearby the other letter t in keyboard and they have a sequence order in position. It is also easy to be confused especially when typing fast. This is classified as a **permutation** error.

In the last two case, there is a letter missing and added. Those are classified as **deletion** and **insertion** error respectively.

All the spelling errors distributed above only has one letter of error from the intended word. They are called a **simple error.** In fact, a research in 1964 shows over 80% of spelling errors fell into those four classes of simple errors[4]. A solution to that is called Damerau-Levenshtein Distance (also know as minimum edition distance).

The algorithm is described in detail in Appendix A.

The Levenshtein Distance algorithm has good time and space complexity $O(w^*v)$ which w and v are the size of each word. Therefore, this is widely adopted in many spell checkers and become a mainstream approach to correct non-word spelling errors.

However, the only limitation is that the **permutation** error can not be solved using this algorithm. Therefore, there is a need for an alternative approach for that.
3.2 Phonetic Spelling

Other possible cause of spelling error related to the homophones among words. To remembering a spelling of a word, the pronunciation of the word is often used as an aid. Conversely, when someone starts writing a sentence, the sentence may be “spoken in head” before it is written.

Phonetic Spelling error[5] distributes a kind of spelling error made when the erroneous word has similar phoneme to the intended word. For example, one might spell *acknowlege* when *acknowledge* is intended because the letter *d* has low use on pronouncing the word and be omitted easily. Another example, the word *niece* may be spelled as *neice* because the blur on similar sound on *ne* and *ni* lead to confuse on letter positioning. These example above have small distance (minimum edition distance) to the original word and can be fixed by the algorithm described in the previous section.

A more interesting case happens when two word with multiple distances but both have homophones. Examples such as: {allowed, aloud}, {bait, bate}, {ceiling, sealing}, {raw, roar}. Each pair above has homophones however cannot be detected with the distance algorithm. Hence, phonetic spelling is also one of the possible cause to real-word spelling error.

In a study in 1987 (Mitton, 1987), Mitton states that 44% of the spelling error study made were caused by homophone substitution. To solve these kind of real-word error, there are also researches on phoneme approach on spell checker.

3.3 Uncorrectable spelling error

Although some causes of spelling error, there are still spelling errors that are likely uncorrectable.

An early study (Wing and Baddeley’s, 1980) mentioned a type of handwriting error called “convention errors”. It is an error made when a user has lack of knowledge of the conventional spelling. It is considered uncorrectable as no one but the writer understand what the spelling error referring to. Some of errors made can be phonetic or graphemic (Differ from phonetic spelling, these type of errors are made when the word has similar shade to the intended ones. Example: {fisl, fish}, {valy, valley}, {pirite, private} [6]). Some are just total nonsense words (Example of 12-year-old spelling mistakes (from study in [5]): {zates, latest}, {usterand, understand}). Some of those can be corrected by some phonemic/graphemic standards, however a portion of these type of error can not be recovered.

3.4 Summary

In this chapter, some possible causes for a user to make a spelling mistake are identified. Some of them are correctable and some of them are not. There are researches focus on how to solve each kind of spelling error (will be mentioned in detail in the next chapter) and it is crucial to decide which kind of error can be solved and which cannot when designing a spell checker.
4. Background

Over the years, there are different attempts on developing a context-sensitive spell checker. Each has different performance on various kind of real-word errors, however none of them can provide perfect accuracy. In this chapter, some of the important approaches will be mentioned and how do they extract meaningful information from the context in order to do real-word analysis.

4.1 Machine learning approach

One attempt to a context-sensitive spelling correction is Machine learning based. It define a set of corpus which a spell checker follows and train the spell checker with sample data hence also called a feature-based method. There are two famous approaches: Bayesian Hybrid (Golding, 1995) and Winnow-based (Golding and Roth, 1996).

4.1.1 Bayesian Hybrid

This method has make use of two kinds of features: context words and collocations in the training process. The former checks if a word appear ±i word positions to the target word; latter analyse a pattern of j word and/or part-of-speech tag combinations taken from the context. For example, consider the confusion set {peace, piece} mentioned above. A context word feature could be: word appear in ±3 word positions. Collocation to the same case would be: ___ of cake. Both context word and collocation point toward the word piece hence the real-word error can be detected and suggested.

4.1.2 Winnow-based

This method uses the winnow method in machine learning. Unlike the method above, the feature size is not limited and the accuracy increases when more features are included most of the time. There is a threshold and two labels(1 = true and 0 = false). The threshold serves as a line drawn between two labels: if a value $x > \text{threshold}$, it is true, false otherwise. In the training stage, the value of threshold varies when more training data and their corresponding labels are reviewed. Each features have different proportion (e.g. feature 1 occupies 13% in threshold, feature 2 occupies 25%...etc) when calculating the threshold and hence the value threshold is depended on different features which make it reliable.

The winnow-based approach gives a promising result on which it is able to recognize about 96% of real-word errors[8] according to the developer.

4.2 N-gram approach

Other approach is by analysing a set of n-grams derived from the context. A n-gram language model is widely used in natural language processing. N-gram means a set of n things, can be letter, words, symbols.... in which word n-gram is used in context-sensitive spell checker. For example, the sentence “I want to be a guy” will derived word trigrams(3-gram): {I, want, to}, {want, to, be}, {be, a, guy}. The use of this model often predicts the probability of item(word) i occurs in the item set j, the probability formula derived as follow[9]:

```math
P(i | j) = \frac{n_i \cdot \Pi_{k=1}^{n-1} p_k}{\Pi_{k=1}^{n} p_k}
```

where $n_i$ is the number of times the ith word appears in the jth n-gram and $p_k$ is the probability of the kth word in the n-gram.
In an n-gram model, the probability \( P(w_1, \ldots, w_m) \) of observing the sentence \( w_1, \ldots, w_m \) is approximated as
\[
P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i | w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i | w_{i-(n-1)}, \ldots, w_{i-1})
\]

The formula varies with different n-gram (change of parameter n inside the formula) and it is the basis of many n-gram approach in context-sensitive spell checker.

4.2.1 5-gram approach

In 2006, Google published a Web 1T 5-gram striped from their web crawling data. There are subsequence approaches based on that piece of data (e.g., [10], [11]). In general, the method involves matching the words in 5-gram descending to 1-gram, and suggests the suitable correction based on pre-defined confusion set. The conditional probability served as a measure if a word is suitable for the context, also the ranking of error suggestions. This method shows a good accuracy (over 90% average accuracy).

4.2.2 Noisy Channel Model

Church and Hale (1991) suggested a noisy channel to predict corrections for a real-word error. This model treats the spelling errors as a distorted form derived from a noisy transmission channel and it start guessing what the original word is by altering the words along with the n-gram. This model make use of minimum edition distance mainly, create a range of “noisy” words and choose the one with the best probability. For each suggestion \( s \) for word \( w \), \( P(w | s) \ast P(s) \) is calculated to get \( P(s | w) \). Similar to n-gram language model, but noisy model does the match conversely: by altering the erroneous word instead of finding suitable words in the database. This saves the dictionary search time especially when the n-gram in suspect has large possible word size, e.g. generate a list of possible words after “is” and “a”, size will be exponential. The context-sensitive spell checker by Google also uses the Noisy Channel Model in spelling detection[12].

4.3 Summary

In this chapter, different approaches to a context-sensitive spell checker are reviewed. Machine learning techniques involve training to the spell checker and n-gram approach focus more in probabilistic measure of a word in a sentence along with its confusion set. Two main approaches have their own advantage on different kind of errors. There is also an attempt on a combined approach on both ([13]) and shows a level of success. Those approaches served as an inspiration on how to extract and analyse meaningful information from the context.
5. Design

From previous chapters, the problem of real-word spelling error, the goal of the project, and some analysis on spelling error are described. In this chapter, more details on the high level design will be discussed. The aim of the project's design is to find an optimal model which provide a great accuracy on spelling detection and correction to solve real-word error also guarantees the best use of resource. The design described in the chapter will serve as a model in implementation.

5.1 Approach on building context-sensitive spell checker

The context-sensitive spell checker for this project has adopted trigram language model (Mays et al, 1991) and concepts from mixed trigrams approach[14]. It involves calculating the conditional probability for (the part-of-speech of) an word given the surrounding trigrams and verifies with the confusion sets defined.

When a text is inputted, it is preprocessed by a string tokenizer along with a part-of-speech tagger. Spelling detection started with verifying the part-of-speech(POS) of each word to the mixed-trigram. If a real-word spelling error is found, the word will be passed to the spelling correction stage. In this stage, the relative suggestions are generated based on surrounding trigrams and confusion sets of erroneous word. Each suggestions are ranked upon a set of constraints so the most suitable suggestion will be shown on top.

The main reason to adopt trigram approach is that the space required relatively small. In some approaches mentioned before, e.g. 5-gram approach, are memory-based and are not very efficient in both space and time to analyse and store. Secondly developing a context-sensitive spell checker with trigram is the original requirement in the project title, the change of approach will involve the change of project title. Base on those motivations, the spell checker has adopted a trigram approach (5.1.1) along with a confusion set (5.1.2).

A word trigram spell checker is designed and implemented in the early stage of the development and kept in order to compare performance to the mixed trigram approach. The major difference among two approaches are the spelling detection method, which will be mentioned in 5.1.1.

A simple lexicon spell checker is also developed in this project, it implements a simple approach described in 2.1.1. Hence the approach of that is not discussed in this part.
This is a peace of cake.

String Tokenizer

This, is, a, peace, of, cake

Derive word trigrams

{This, is, a}
{is, a, peace}...etc.

Verify on word trigrams

This is a peace of cake.

Fig. 3: A diagram of an spelling detection flow using word trigram approach

POS tagger

This/Dt is/VBZ a/Dt peace/NN of/IN cake/NN .

Derive Mixed POS trigrams

{DT, is, a}
{This, VBZ, a}...etc.

Verify Mixed POS trigrams

Derive word trigrams

{This, is, a}
{is, a, peace}...etc.

Non-word spell checking on dictionaries

Recheck on word trigrams

This/Dt is/VBZ a/Dt peace/NN of/IN cake/NN .

Fig. 4: A diagram of an spelling detection flow using mixed pos trigram approach
5.1.1 Spelling Detection

Figure 4 demonstrated the system flow for spelling detection.

5.1.1.1 Non-word Spelling Error

A non-word spelling error is detected using a simple dictionary technique: check if it is in the dictionary, it is correct; incorrect otherwise. Non-word spelling error detection is crucial for this project. First, this is a base type of spelling error hence it needs to be corrected before any real-word spelling error detection. Secondly, the accuracy of the context-sensitive spell checker relies on the completeness of the surrounding context. If one or more word in the sentence is incorrect, some potential real-word spelling error cannot be detected as there are less useful context extracted. For example, in the n-gram approach, a n-gram is valid if it contains one or more misspelled words.

5.1.1.2 Real-word Spelling Error

For real-word spelling error detection, mixed POS trigram is used mainly to verify words. It is a trigram with a mixture of word and the part-of-speech of the word. To construct a mixed POS trigram, words will be tagged with their suitable part-of-speech label using a part-of-speech tagger before the spell checking begin. In this approach, the mixed POS trigram contains one part-of-speech tag and two words. An example of trigram construction is shown below:

Sentence: My dog also likes eating sausage.

Tagged Sentence: My/PRP$ dog/NN also/RB likes/VBZ eating/VBG sausage/NN ./.

Mixed POS trigram derived:
PRP$, dog, also
{My, NN, also}
{My, dog, RB}
{NN, also, likes}
dog, RB, likes
dog, also, VBZ
RB, likes, eating
also, VBZ, eating
also, likes, VBG
{VBZ, eating, sausage}
{likes, VBG, sausage}
{likes, eating, NN}

(note: this data is tagged by Penn Treebank POS tag set, please see appendix B for more details)

When a word spelling is being checked, the probability of the word part-of-speech being in specific position is calculated using the formula specified in 4.2. For example, if the word “likes” is being checked, the mixed pos trigrams {dog, also, VBZ}, {also, VBZ, eating}, {VBZ, eating, sausage} are used to determine if “likes” is suitable to the context.
**Word trigram vs Mixed part-of-speech trigram**

Why did POS trigram is chosen instead of word trigram in (the first) spelling error detection? In fact, word trigram can also be used as verification in spell checker and also my initial design of the spell checker (Figure 3). The major reason is to provide a level of allowance in the context. In any n-gram approach, data sparseness is an inevitable problem. N-gram need to cover as many contexts as possible else a correct word is easily marked as incorrect (false positive) due to the insufficient of data. Also, it need to be updated constantly as there are more contexts produced in everyday basis. The accuracy of n-gram relies highly on the completeness of n-gram data. Conversely, mixed POS trigram provides a more abstract checking on words: as long as it is the same part-of-speech, it is fine. This is beneficial especially when a word which is not observed by word trigram but in correct part-of-speech in position. A common example is to search a noun after {is, a}, as most of the noun is suitable in the context. If detection stage is implemented by word trigram, almost all possible noun are included and the size will be exponential. Instead, storing a mixed part-of-speech trigram {is, a, NN} save both in time and space required to process the error detection.

**Rechecking on word trigram**

The mixed trigram detection on words provide great efficiency when erroneous words are in different part-of-speech. However, if it is in the same part-of-speech, the word error will go undetected. An example is the confusion set {peace, piece}: peace and piece both share the same part-of-speech (noun) and they will go undetected in method stated. To solve this problem, there is a rechecking stage after the first stage has passed. In this stage, a lookup on word trigram is done in order to find if there is any better replacement on the original word (which marked as “correct” in the previous stage). A word from trigram will be classified as “better” from the original word if:

1. It applies to the same word trigrams surrounded the original word
2. It has a better probability than the original word
3. It lies within the confusion set of the original word (more will be mentioned in 5.1.2)

If there exist 1 or more better words, the word will be marked as “uncertain” because it is not always a word error or a correct word. A list of suggestion will still be shown but user will have to design if they take the suggestion or not.

**Constructing Word trigram and Mixed part-of-speech trigram**

Corpus of Contemporary American English(COCA)[15] is the source of trigram for this project. It is the largest free source of English. It analyse 20 million words each year from 1990-2015 from spoken, fiction, popular magazines, newspaper and academic texts. This is the source of the word trigram as it provided a high level of accuracy. It also tagged the trigram with their relative part-of-speech tags (which google n-gram data has no tags), hence mixed pos trigrams can be derived from that data. A sample data is shown in figure 5.

```
<table>
<thead>
<tr>
<th></th>
<th>B.A.</th>
<th>degree</th>
<th>at1</th>
<th>nn1</th>
<th>nn1</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B.A.</td>
<td>in</td>
<td>at1</td>
<td>nn1</td>
<td>ii</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>B.S.</td>
<td>in</td>
<td>at1</td>
<td>np1</td>
<td>ii</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>BA</td>
<td>in</td>
<td>at1</td>
<td>nn1</td>
<td>ii</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>babble</td>
<td>of</td>
<td>at1</td>
<td>nn1</td>
<td>io</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>babe</td>
<td>in</td>
<td>at1</td>
<td>nn1</td>
<td>ii</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>baby</td>
<td>and</td>
<td>at1</td>
<td>nn1</td>
<td>cc</td>
<td>308</td>
</tr>
<tr>
<td></td>
<td>baby</td>
<td>at</td>
<td>at1</td>
<td>nn1</td>
<td>ii</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>baby</td>
<td>bird</td>
<td>at1</td>
<td>nn1</td>
<td>nn1</td>
<td>50</td>
</tr>
</tbody>
</table>
```

Fig. 5: Sample trigram data stripped from the data set
5.1.2 Spelling Correction

Figure 6 shows a system flow in spelling correction. For spelling correction, confusion set is a crucial element as it contains all spelling suggestion to the spelling error. Not limited to real-word spelling error, it needs to provide suitable suggestions for non-word spelling error based on the surrounding context. Therefore, it need to be well constructed to provide a great error correction rate. Different spell checker approaches suggests different methods to construct a confusion set hence there are no standard way to construct a confusion set. In this project, the confusion sets are designed in order to solve different kind of errors mentioned in Chapter 3. A confusion set of a word is constructed by the following sectors.

5.1.2.1 Word distance to the original word
First, a word will be consider as a confusion if it has a close distance to another word. As mentioned in Chapter 3, this is a classic technique to solve typing error (typo). Not limit to only non-word spelling error, typo is one of the main cause which real error is made. For example, the word *click and clock* are a common typing error made personally because the adjacent keyboard position among two letters; click and clock will then be put in the same confusion set as they often confuse to each other. The algorithm uses the levenshtein distance algorithm described in 3.1.1. By applying this algorithm, the insertion, deletion, substitution errors inside confusion set can be corrected.

5.1.2.2 Word permutation
As described in 3.1.1, permutation error is not solved using levenshtein distance algorithm. Therefore, an algorithm is designed to solve permutation error. Like the Noisy Channel algorithm (4.2.2), it constructs a set of noisy words by permuting the words. The pseudocode is shown in figure 7.
output = list of valid words to output

for all character i in word w,
  w* = w with i and i-1 swapped
  If dictionary contains w*,
    Add w* to output

return output

Fig. 7: a pseudo code of permutation algorithm

5.1.2.3 Word phoneme

Word phoneme need to be considered as well. Many confusing words are the words with the same pronunciation. By analysing the phoneme among words, some of them can be added to the confusion set as they are easily confused with each other. The method to this is to pair up words with similar/same phoneme using phoneme dictionary. In this project, CMU Pronouncing Dictionary[16] is used as the source. The example on finding phoneme matching for piece is shown as follows:

1. Extract the phoneme for the word being viewed
   PIECE       P IY1 S
2. Do check on each word until a match with the same phoneme is found
   PEACE       P IY1 S
3. Add the found word into the confusion set of the original word
   PIECE    {PEACE}
4. Repeat step 2 until all words are viewed

Because of the complexity of searching on words, the phoneme confusion sets are pre-generated hence only one training stage is needed. The program code to generate confusion set is shown in Appendix C.

5.1.2.4 Additional confusion set

Although the two conditions above provides checks on quite a wide range of easy confused words, there are some words still not yet be captured. A classic example is the confusion set {borrow, lend}, this is an example of confusing words with homogeneous meaning. Those will be included in a self-defined confusion set. The details on the self-defined confusion sets can be found in Appendix C.

All the real-word spelling errors are correctable if the intended word is inside the confusion set of the original word. On the other hand, word errors are uncorrectable if the intended word is not in the confusion set. In this case, it can still be detected but no useful suggestion is shown due to the insufficient of data covered in confusion set.
5.2 Development

5.2.1.2 GUI

The GUI skelth of the spell checker is shown in figure 8. In the input pane, it allows users to input text they wish to do spell checking on. After that, there is a check button which start processing the user input when pressed. The feedback of the user input will be shown on the output pane which all the incorrect words will be highlighted in red. User can get the immediate feedback on spelling detection. Spelling suggestion are shown in the suggestion pane with a next button to switch between different suggestions for different words. In this part, user will be able to see which kind of error they have made (non-word or real-word) and the suitable suggestions of each. The suggestions should be ranked which the most preferable suggestion will be shown on top. In addition, the selection list is able to let user to switch between a basic spell checker and a context-sensitive one.
5.2.1 Requirement

To meet the goal of the project, sets of requirement of the system are defined. The functional requirements are shown below:

1. The system needs to take user's input correctly
2. The system should highlight non-word spelling errors
3. The system should highlight real-word spelling errors
4. The system should be able to generate a list of suggestions
5. The users should be able to view the system feedback based on their input
6. When all spelling errors are corrected by the user, the system should stop highlighting any word in the sentence as all the words should be correct

The non-functional requirements as follow:

1. The context-sensitive spell checking can be turn off by users
2. Users can correct their suggestion with a press of button

When the system is complete, it needs to meet all the functional requirements.
Non-functional requirements can be implemented to enhance user experience.

5.2.2 Development Process

In the project development, the agile model was adopted. This development process is both iterative and incremental. The project development is splitted into multiple iterations. Each iteration is used to develop a prototype of the program and implements more functionality than the previous prototype. As the development cycle is repeated in each iteration, the risk of development is hence reduced. Once there is an error in the implementation, simply roll back to the previous prototype of the system. Also it can split the requirements to different stage hence each requirement can be checked and corrected in early stage. The follow shows the brief event on each iteration:

**Iteration 1**: Build a normal spell checker
**Iteration 2**: Build a context-sensitive spell checker with word trigram approach
**Iteration 3**: Build a context-sensitive spell checker with mixed POS trigram approach

Iterations are split upon the difficulty in implementation. In each iteration, the prototype inherit a level of design from the previous iteration along with increment based on requirement. The programmer also got more familiar with the system based the previous iteration.

5.2.3 Programming Tool

This project is developed with Java along with data stored in MySQL based database. Java is an easy programming tool to use as it provides a number of useful default functions which can aid the system development. Also, the external package used in the project, Stanford NLP tool, is written in Java which provides tokenizer and POS tagger and both plays major role in text processing in the implementation. In addition, this project involves analysis on big size of text, all data are stored in the database. MySQL is a useful language to extract data from the database. It provides an effective search in the database hence saves time for searching and matching. It is also compatible with Java easily with connectors[17].
5.2.4 Software Architecture

The spell checker in this project consist of two part: lexicon spell checker, word trigram spell checker, mixed POS trigram spell checker. In order to allow users to switch between different spell checker using the same GUI. A software architecture called Model-View-Controller(MVC)[18] is adopted.

5.2.4.1 Model-View-Controller

The model view controller means the complete separation of model(Backend System model) and view(GUI). A controller is used to make interpretations among two of them. When users try to interact with the system, they will interact with the controller using keyboard/mouse. Then, the controller informs the model to operation. After getting the feedback from model, update the view to show users the feedback of the input.

Fig. 9: MVC structure in the project

Figure 9 shows how MVC structure applies to the project. In this project, the controller (selection list and check button) is a component of view(GUI). It serves as a switch among different models(spell checkers). The model then updates the view with associate result: each type of spell checker highlights different words on the output pane in GUI. The users can access different kind of spell checkers by changing the selection and the controller will change the mode based on user’s interaction. This software architecture model is beneficial in terms of the separation between view and model: the change in one of them will not result the update in another. Providing an efficient way to implement the software system.
5.2.4.2 Design Model

From the requirement gathered from previous chapter, a set of use case are identified:

1. Normal Spellcheck: Spell Verification
2. Normal Spellcheck: Spell Suggestion
3. Word Trigram Spellcheck: Spell Verification
4. Word Trigram Spellcheck: Spell Suggestion
5. POS Trigram Spellcheck: Spell Verification
6. POS Trigram Spellcheck: Spell Suggestion

A use case diagram is shown on Figure 10.

User inputs the text into the controller to interact with the system, hence user is always participate in the use case of spelling verification. To verify and correct a word for normal spell checker, dictionary database is accessed. In a word trigram spell checker, it interact with the word trigram database to verify and also generate suggestions. However, in the mixed POS trigram approach, although mixed POS trigram database plays a major role in spell verification, word trigram also accessed in order to do rechecking. Also, to generate suggestions with suitable position and part-of-speech, both word trigram and mixed pos trigram database are accessed.

As confusion sets are either generated in the system(word distance, word permutation) or are stored in external text files (phoneme, addition set). They do not play a role as an actor.

As different actor plays different role in the use case, the design model for each kind of spell checker are different(Figure 11, 12, 13). When normal and word trigram only interact with one database, the mixed pos trigram approach has interaction with two database. All three database differ from entries and structures, which provide aid in different approach.
Fig. 11: a high-level design model for normal spell checker

Fig. 12: a high-level design model for word trigram spell checker

Fig. 13: a high-level design model for mixed POS trigram spell checker
5.3 Summary

In this chapter, details about the spell checker approach on detections and corrections are mentioned. A simple view about the development process is viewed also included some design models based on use cases. In the next chapter, the actual implementation on the design will be mentioned.
6. Implementation and Testing

Fig. 14: A system model for implementation
In this chapter, the details about implementation on the project will be discussed, including different tools and system methods used to construct the project. Figure 14 gives an overview in the system architecture of the program in implementation. Each of the classes will be described in this chapter.

6.1 Word Processing Tools

All the text preprocessing in this project are handled by Stanford NLP Tools[19]. It is a open source written in java and contains a set of useful tools for language processing. The two classes I used for this project are DocumentPreprocessor, PTBTokenizer and MaxentTagger respectively. The first one is a document preprocessor, The second one is a string tokenizer and the third one is a part-of-speech tagger. All the input text passed from the controller will be processed by those tools before doing any spelling checking analysis.

6.1.1 DocumentPreprocessor

The DocumentPreprocessor is used in all context-sensitive spell checkers developed in this project. It takes a plain text and derives a list of sentences. This method is particularly useful as trigrams are derived from a sentence in this project, as a sentence gives the most accurate information about the context instead of cross sentences. Also a part-of-speech tagger can only tag words from the same sentence, which a sentence need to be passed into the method.

6.1.2 PTBTokenizer

The PTBTokenizer takes a string as input, output with a list of tokens derived. This is similar to the StringTokenizer method in the Java default class. However, instead of only separating sentences by spaces, it also identify the symbols and separate them as well. A simple diagram showing the differences among two of them included in figure 15. This method is used to derive word trigrams as well as checking spelling on each token. Hence this tokenizer is used in lexicon spell checker and word trigram spell checker.

![Diagram showing the differences between Stanford NLP tokenizer and Java build-in tokenizer](image)

Fig. 15: a diagram showing the difference between Stanford NLP tokenizer and Java build-in tokenizer
6.1.3 MaxentTagger

This MaxentTagger is used to tag words with part-of-speech tags from a sentence. It has similar functionality to PTBTokenizer but output with a list of tokens with part-of-speech tag. Therefore this method is used only in mixed POS trigram spell checker. A sample version of this tagger is available online[20]. However, this tagger only tag the words in Penn Treebank tagset (Appendix B) which 36 tags are outlined. The trigram data was tagged in CLAWS7 tagset[21]. There is a need to convert two of them, more about the relative method will be described in latter section this chapter.

6.2 Data Structure

6.2.1 Storing dictionary and trigrams

The dictionary, word trigram and mixed pos trigram are the most crucial part in the program as they store most of the data used for spell checking. There is a need to store them in a secure place and provide fast access in search as the data are accessed frequently. For the purpose of that, MySQL database (phpMyAdmin[22]) is used as a storage method. The table structure varies for different purpose for processing, the details about those will be described in this section.

6.2.1.1 Dictionary

Dictionary is stored in one string entry called word. As it is mainly used in non-word spelling error detection, there is no need to include any addition information above that (e.g. frequency). This only field is the primary key of itself. For the purpose of accuracy, the dictionary table is a combination of external source [23] and unigram derived from COCA source. It can guarantee a non-word spelling error is not misdetected before the real-word spell checking begin. Also provides word check allowance for every words which are not covered in the word trigram database.

6.2.1.2 Word trigram

Word trigrams are stored in four fields called word1, word2, word3 and frequency. The first three fields store the trigram. As each column is not a unique identifier itself, the primary key is assigned to all three fields. In a database, primary key can greatly speeds up queries, searches and sort requests[24]. Correctly identifying them can provide MySQL operations work more efficient. The word trigrams stored is Case Sensitive. It is beneficial to provide case sensitive spelling detection and suggestion: the confusion set {america, America} can be detected since they are treated as different records in the database. The frequency count is included in order to calculate the probability of occurrence for each word. Also, it serves as a ranking corpus in spelling suggestion result.

6.2.1.3 Mixed POS trigram

Mixed POS trigrams are stored in different tables based on different position of part-of-speech. 1stPos table stores all the mix trigrams started with part-of-speech tag pos and followed by other two words, 2ndPos table stores all the mix trigrams with
part-of-speech tag in the second word position, etc. The purpose of storing them in separate
tables is to illuminate confusions between part-of-speech tag and word, some word can be a
part-of-speech tag itself and will treated as repeat entries in the database. The primary key
is the part-of-speech tag with the other two words and applies all three table, as neither any
of the field can identify itself.

6.2.2 Storing confusion sets
Differ from trigrams, the amount of data stored in confusion set is fairly small. Some of them
can even be derived in runtime, therefore, there is no need to store a confusion set in the
database. As mentioned in the previous chapter, different kind of confusion sets are stored
differently. The storing method of each of them are designed in order to provide best
efficiency in process.

6.2.1 Word distance and word permutation
These two confusion sets stores the words derived from word distance and word
permutation respectively. It is not a good idea to store them in local storage. The main
reason is that the variation of a word change one letter to become another is too high. For
example, “a” is a valid word. The words derived from word distance are all the two letter
words with “a” inside, which the size is pretty huge. If suggestion is being generated,
analysing the large set of variate words and selecting the suitable one is not a good idea in
terms of time efficiency. Therefore, two confusion sets are generated upon run time in
following steps:
1. When a suggestion list need to be generated, get access to all the possible words in
   the intersection of surrounding trigrams
2. From all possible words, select the word which meets the word distance/permutation
   confusion set criteria.
By apply this step, the memory needed to store the confusion set reduce to “all possible
words in surrounding trigrams”.

6.2.2 Word Phoneme and addition confusion sets
Two of them stores the word phoneme and self-defined confusion sets respectively. Those
are stored in the local storage. Unlike word distance and permutation, they cannot be
applied to any algorithm which can derive confusion sets by simply comparison among two
words. In addition, the size of data is not too large so there is no need to stored in the
database. They are stored in the text file as input file, which gives both a good space and
time efficiency.

6.2.3 Java data structure
When sets of data (dictionary, trigrams or confusion sets) are loaded into Java class for
processing, they are stored in a object class called ArrayList. This in-build object class
provide some powerful functions for array processing. For example, trigrams are updated
each iteration by removing the first word and insert a new word. ArrayList can handle the
pop operation easily by removing the first item (e.g. trigram.remove(0)). For some dictionary
check for non-word spelling mistake, a string can be checked easily with one line of code
(e.g. dictionary.contains(word)).
6.3 Attributes and Methods

All main attributes and methods are stored in GUI class in the development. A small description on each attributes and method mentioned below:

**Attributes**
- **wordDict** - word dictionary
- **unigram** - unigram derived from same source to the trigram data
- **confusion** - a confusion set storing all confusions made in phoneme and extra confusion sets
- **posConfusion** - a confusion set among part-of-speeches
- **posConvert** - a convert list between Penn Treebank and CLAWS7 tagset
- **errorWord** - a list of words that is highlighted “false” in spell checkers
- **errorTriMap** - a hash map storing surrounding trigram of the error word
- **wdSuggestionList** - suggestion list from word distance
- **perSuggestionList** - suggestion list from word permutation
- **pSuggestionList** - suggestion list from phoneme and additional confusion sets

**Methods**
- **loadPOSconverter** - initialize attribute **posConvert**
- **loadPOSConfusion** - initialize attribute **posConfusion**
- **loadConfusion** - initialize attribute **confusion**
- **phonemeCheck** - method to find words with phoneme confusions
- **charsSwapped** - method to find words constructed by word permutation, implements the algorithm in figure 7.
- **makeDistanceSuggestions** - method to find words constructed by word distance, calls comparison done by distance generated from **minDistance** method
- **minDistance** - implements the Levenshtein Distance Algorithm.
- **doContextCheck** - method for word trigram checker detection
- **doPosCheck** - method for mixed POS trigram checker detection
- **doWordCheck** - method for lexicon spell checker detection
- **initialize** - the controller method, pass text and switch between different spell checkers. Also updates the suggestion pane after generating suitable suggestions.

In the system, spelling detections are done in different method (doContextCheck, doPosCheck, doWordCheck) because of the variation in approach. The design of each was described in the previous chapter. They will then update the attribute **errorWord** for any spelling error. In both trigram approach, **errorTriMap** is also updated to store the surrounding trigram for context sensitive suggestions.

The controller method, **initialize** will process the suggestion based on **errorWord** highlighted from spell checkers. The suggestions are generated and ranked from three methods: **phonemeCheck**, **charsSwapped**, **makeDistanceSuggestions** which updates **pSuggestionList**, **perSuggestionList** and **wdSuggestionList** respectively. If any of the suggestion list is not empty, means the word is correctable and its suggestion will be shown.
6.4 Testing
In order to test the correctness of implementation, JUnit is used as a testing method which verifies if a function behave as expected. These testing can check if the design is correctly implemented by the program. Some examples are included below:

**testDatabase** class - Check if all the database is correctly connected
**testPhoneme** class - Check if a phoneme confusion set is correctly constructed
**testWordTrigram** class - Check if trigrams are correctly constructed

In addition, known data are used for testing. Each test cases are constructed with known trigrams and confusion sets, in order to check the core functionality of the system. Some examples as below:

**testPOSDetection** class - check if a trigram with part-of-speech error is correctly identified
**testWdSuggestion** class - check if a word distance error is correctly suggested
**testPhonemeSuggestion** class - check if a phoneme error is correctly suggested

By doing testing on those repeatedly, the correctness of each object and method can be preserved. It is very useful especially the programming intensity of this project (about 3000 lines of code).

6.5 Summary
In this chapter, the implementation level of the system is mentioned. Including the storage methods, attributes and methods in the system. In the next chapter, the evaluation of the result will be mentioned.
7. Evaluation of results

In this chapter, the evaluation of the result will be mentioned. First, an overview on how the program interact with users will be evaluated. It will then introduce a set of standards to compare the result with modern spell checkers and also the efficiency of the program in terms of time and memory usage.

7.1 User's Interface

As shown in figure 14, users can do switch between different models by selecting different item on the list and click “Check”. In addition, when there is no suggestions available, the “Next” button will be disabled.

Fig. 14: Demonstrates a change in spell checker model

When there is more than one suggestions available, users can switch between suggestions by clicking “Next” button (Figure 15).

Fig. 15: Switch in suggestions is available

In suggestions, common confusions included the phoneme confusion and self-defined confusion sets. Permutation and word distance means the confusion sets made with the name itself. Users can know how they made an mistake on. Also, there is a tag near the mistaken word stating the cause of mistake: part-of-speech error, spelling mistake or “Do you mean?”, which are the set of words derived from word trigram rechecking, letting user to design to take the suggestions or not.
After the correction is made, the output pane stop highlighting the word. (Figure 16)

7.2 Performance

To measure the performance of the system, time efficiency and space efficiency is used. Due to the scope of the project, it is difficult to measure with big O parameters. Hence, the time taken to run a set of data is measured.

In this testing, 20 sample sentences with length 5 to 11 words are applied to all three spell checkers. Each of the sentence contains one non-word spelling error and one real-word spelling error. All the tests are performed in Linux configured with Intel i5-2400 CPU @ 3.10GHz and 8GB of Memory. Table 1 demonstrates the result.

<table>
<thead>
<tr>
<th>Spell Checker</th>
<th>Average Memory Used</th>
<th>stdev</th>
<th>Average Time Taken</th>
<th>stdev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Spell Checker</td>
<td>69.1 MB</td>
<td>0.32</td>
<td>0.097 seconds</td>
<td>0.004</td>
</tr>
<tr>
<td>Word Trigram Spell Checker</td>
<td>70.8 MB</td>
<td>0.92</td>
<td>1.94 seconds</td>
<td>1.17</td>
</tr>
<tr>
<td>Mixed Trigram Spell Checker</td>
<td>206.3 MB</td>
<td>2.16</td>
<td>3.14 seconds</td>
<td>1.26</td>
</tr>
</tbody>
</table>

Table 1: Average memory and time taken with standard deviation for three spell checker approach

The memory usage and time taken is relatively small in the normal spell checker, as it only need to do check each word in dictionary.

The memory increase slightly in word trigram spell checker as more information such as external confusion sets are stored. The memory grows about three times more in mixed trigram approach, the major reason for this is because of more data are used in part-of-speech analysis, e.g. additional trigrams needed for word trigram rechecking.
From normal spell checker to mixed trigram spell checker, the time taken increase in ascending order. The **word trigram approach** involves more matching on trigram hence the process time increase. The algorithm for **mixed trigram spell checker** involves large preprocessing of data, in fact about 1.2 sec in average are used to load part-of-speech tagger from Stanford NLP tool. Also plus additional time needed to do rechecking on trigram, the time taken is significant larger than previous two approaches.

In terms of the time and memory used in this project, it is lower than expected thanks to the efficient search function in MySQL database. There is no need to load huge sets of word trigrams (about 800k records, 50MB of data) and part-of-speech trigrams (about 550k records each POS position = 60MB) during runtime. The search time is also very fast in MySQL using the distributing computing technique: all the commands are passed to the server and returned when finish. The process speed relies on the server not the local host, hence the process time is improved significantly.

### 7.3 Results

The accuracy of the spell checker approaches will be mentioned in this part. It is an important measure which states how effective is the spell checker implemented in this project.

<table>
<thead>
<tr>
<th>Actual Class (True Label)</th>
<th>Predicted Class (Labeled by Spell Checker)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>True Positive</td>
<td>No</td>
</tr>
<tr>
<td>False Negative</td>
<td>False Positive</td>
<td>True Negative</td>
</tr>
<tr>
<td>False Positive</td>
<td>False Positive</td>
<td>True Negative</td>
</tr>
</tbody>
</table>

Table 2: A table of confusion matrix

#### 7.3.1 Standards

##### 7.3.1.1 Spelling Detection

The standard used to measure both the performance of detection rate is **Confusion Matrix**. It is widely used in development of any classifiers. A confusion matrix comprised of four elements, as shown in Table 2. Each elements related to different possible results during detection/correction testing, as shown below:

- **True Positive** - A correct word is correctly identified
- **False Negative** - A correct word is falsely identified to be spelling error
- **False Positive** - A spelling error is not detected
- **True Negative** - A spelling error is correctly identified

True Positive and True Negative are preferred attributes and the other two are less preferred as they indicates a mistake.

In use of the confusion matrix, a couple of measures derived which will be taken as measures on detection:

\[
\text{Accuracy} = \frac{\Sigma \text{True Positive} + \Sigma \text{True Negative}}{\text{Total}}
\]
This measures the overall accuracy of the spell checker: how often is it correct in total basic.

**Sensitivity** = \[ \frac{\sum \text{True Positive}}{\sum \text{True Positive} + \sum \text{False Negative}} \]

This measures how often a correct spelling is correctly identified

**Detection Rate** = \[ \frac{\sum \text{True Negative}}{\sum \text{True Negative} + \sum \text{False Negative}} \]

This measures the degree of spelling error detected

By calculating the measure above, the overall accuracy of the spelling detection can be accessed precisely.

### 7.3.1.2 Spelling Correction

The spelling correction rate used a more simple approach, correction rate. It is calculated by how often a correct suggestion is shown after a spelling error is correctly identified:

\[ \frac{\text{Case with correct suggestions}}{\sum \text{True Negative}} \]

This measure will shows the accuracy of the spelling correction of each approaches.

### 7.3.2 Test Data

The test data used in measure comprise of the following two types in order to perform a comprehensive test:

- **Text with no spelling errors**
- **Text with real-word spelling errors**
  - This comprised of five classes
    - Text with word distance errors
    - Text with word permutation errors
    - Text with word phoneme errors
    - Text with self-defined confusion errors

All the texts are derived from a random basis from **news and journal websites** (e.g. BBC, CNN, WSJ…) which may be covered in the word trigram or not. The authenticity of the context is very high since they are reliable text sources. The test involves checking random sentences extracted from different contexts, this cross validation can provide a more reliable test by adding randomness to the test data. Test sentences constructed by about a thousand words in total. real-word spelling errors are inserted randomly to the context which comprises of five categories defined above.

When testing with mixed part-of-speech approach, it is possible to show a “Do you mean” error means a blurred line between correct and misspelled words. For the testing purpose, those will be automatically classified as spelling mistakes.
7.3.3 Final Result

As the tests are aimed to solve real-word spelling errors, the performance of the lexicon spell checker will not be accessed. The performance of the lexicon spell checker can be predicted as: 100% on non-word spelling error, 0% on real-word spelling error. The main test is done on word trigram spell checker and mixed part-of-speech trigram spell checker, using the test measure mentioned in 7.3.1. Also, the data also applied on Google’s context-sensitive spell checker to compare the performance. The evaluated result on spelling detection is shown in following tables:

<table>
<thead>
<tr>
<th>Spell Checker</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Trigram Approach</td>
<td>84.3%</td>
</tr>
<tr>
<td>Mixed POS Trigram Approach</td>
<td>85.6%</td>
</tr>
<tr>
<td>Google Spell Checker</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

**Table 3:** Result for texts with no spelling errors detection

<table>
<thead>
<tr>
<th>Spell Checker</th>
<th>Accuracy</th>
<th>Sensitivity</th>
<th>Detection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Trigram Approach</td>
<td>77.3%</td>
<td>95.4%</td>
<td>65.4%</td>
</tr>
<tr>
<td>Mixed POS Trigram Approach</td>
<td>78.1%</td>
<td>95.7%</td>
<td>65.4%</td>
</tr>
<tr>
<td>Google Spell Checker</td>
<td>95.6%</td>
<td>88.0%</td>
<td>71.2%</td>
</tr>
</tbody>
</table>

**Table 4:** Result for texts with real-word spelling errors detection

Break down on confusion matrix for two spell checkers is available on Appendix E.

In the sample texts with no spelling error detection, there are some words which are misclassified as a spelling error. For word trigram approach, it has an accuracy of 84.3%. For mixed pos trigram approach, it has an improvement of 85.6%. In general, it is very difficult to get to 100% accuracy since there are not enough trigrams stored in the database, causing a classic data sparseness problem.

For sample texts with random real-word spelling errors, the overall accuracy is lower than before (77.3%, 78.1%) since there are random real-word errors added in the context which make variance of data. The sensitivity is well in both spell checkers in comparison to Google Spell Checker which means only a small portion of spelling errors are falsely identified as correct. For all the spelling mistakes included in the context, more than 65% of them are detected which Google does not have perfect accuracy as well.

The high sensitivity is possibly a result of the huge number in false negative which result in smaller number of true label (true positive + false positive) identified by the spell checker. The data sparseness problem grows larger when real-word error is applied. The trigram data are not enough to cover all English context which causes a large amount of misclassified words (false negatives).
Overall, mixed POS approach is better than word trigrams on spelling detection, in terms of overall accuracy and sensitivity though overall detection rate are the same. In another measure not included in the table, Specificity, word trigram and pos trigram has 25.2% and 26.3% respectively. This means mixed pos approach has less misdetected word than word trigram approach as it provides allowance on word input. However, it still suffers from a degree of data sparseness problem since all part-of-speech trigrams are derived from word trigram, causing the low specificity rate.

<table>
<thead>
<tr>
<th></th>
<th>Word Trigram</th>
<th>Mixed POS trigram</th>
<th>Google Spell Checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Distance</td>
<td>45.4%</td>
<td>45.4%</td>
<td>100%</td>
</tr>
<tr>
<td>Word Permutation</td>
<td>57.1%</td>
<td>57.1%</td>
<td>100%</td>
</tr>
<tr>
<td>Word Phoneme</td>
<td>50.0%</td>
<td>50.0%</td>
<td>100%</td>
</tr>
<tr>
<td>Self-defined</td>
<td>75.0%</td>
<td>75.0%</td>
<td>100%</td>
</tr>
<tr>
<td>Confusion sets</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Result of spell correction on each approach

The correction analysis is shown in table 5. For all the error which are correctly identified, only about half of them or below are generated with suitable suggestions for word distance, word permutation and word phoneme confusion sets. Although a spelling error is correctly identified, the trigram with correct word is not always observed sometimes. The confusion set with self-defined confusion sets generally done well. It might due to most of the trigrams used self-defined confusion sets are common trigrams: it contains some common words which as is, are, was, were hence most of them are correctly corrected.

7.3 Summary
In this chapter, the evaluations on GUI, performance and system accuracy are described. In the final result, a degree of data sparseness problem in the trigram data is observed. In the next chapter, the conclusion on the project and some reflections will be mentioned.
8. Conclusion and Reflection

8.1 Outcome

Feedback on the questions defined in 2.4:

1. In what extent the real-word typing errors were detected?
   In this program, the overall accuracy is about 78%. For the detection rate of the overall spelling error is about 65%.

2. In what extent a suitable suggestion was given to an user for each type of spelling error?
   In terms of spelling correction, word distance correction performance is the worst, only 45% is corrected. Self-defined set performances the best which solves 75% of the error identified.

3. In what extent the performance of the context-sensitive spell checker improved from the base spell checker?
   The non-word spelling error detection on both spell checker are perfect. In terms of real-word spelling error, the context-sensitive spell checker raised the detection rate from 0% to 65%.

In terms of the overall outcome of the program, mixed part-of-speech approach does have improvement on checking to unknown words compare to the word trigram approach. However, the system does suffer from insufficient data on spell checker analysis. In overall sense, a level of real-word spelling errors are solved using the spell checker developed in this project which can be considered as a success.

There is an argument made on the performance and result analysis. Although there is a performance increase from word trigram to mixed POS trigram, there is a cost on memory usage, process time and the increase in performance is not very high. Also, both approaches have the same correction rate. Therefore it's hard to say whether mixed POS trigram or word trigram is a better approach.

In many modern spell checker (e.g. Google spell checker), the steps of spell checking are not done locally. Input texts are often passed to external server which has higher computational power. Hence the overall response time can be minimized. Therefore, another argument would be to choose an approach with the best accuracy instead of the one with better space and time efficiency.

8.2 Development process

A context-sensitive spell checker should check and solve real-word spelling errors. In this project, the problem analysis is done in an early stage (Chapter 3) which provides help on constructing confusion sets. The iteration stage was initially planned to be only two iteration: normal spell checker implementation and word trigram implementation. However, the limitation on word trigram on being too strict to words causing false negatives is observed and hence the mixed part-of-speech approach is designed in the third iteration. Not much big modification on the overall system when a new spell checker is designed, showing the benefit from agile software development and model-view-controller system architecture.
From the initial plan, there was a training stage which trigrams should be derived from a large set of raw data. However, the programmer has failed to request additional disk quota in the computer and the training process can not be started. A compromise to that is to use pre-exist trigram data. The source of the trigram used in this project is a lot smaller than the google data (trigram source used in this project: 1 million entries, google 1T 5-gram corpus: 977 million entries. The use of google corpus was in the original plan however the data is not freely available.

8.3 Reflection
In this project, I have enhanced my knowledge about natural language processing. It is a fascinating area which I found a lot of excitement in. In addition, I have gain from experience on how to develop a project, including the researches, requirement gathering, design …etc. In here, thanks for every supervision again by all the supervisors.

8.4 Future Ideas
In this project, there are a number of improvement can be made which may result in a better accuracy/performance. This can serve as an idea to people who wish to take the related project and may present this project in a higher achievement.

Training with raw data: This is an original approach in this project as mentioned before. Though there are free source of trigram data, it does not provide a complete corpus of trigram data. In fact, many modern attempts use web crawling technique to receive n-gram data. It can provide an up-to-date and comprehensive data.

Spellcheck along with unigrams and bigrams: The major problem occurs in trigram approach is that when an error occurs, it is hard to find which of the word is mistaken in the trigram. In this project, it is solved by trying to select the else two words and check if there’s any result. If not, there is probably an error in somewhere else inside the trigram and the word in suspect is mark as correct. However, this is not always the case and causes some false negatives as well. In addition, each trigram only allows one mistake using trigram approach, which sometimes multiple errors go undetected. Bigram and unigram can provide a more refined check on context hence the position of error can be found.

Trigram approach - limitation: There are certain limits on trigram approach. For instance, it cannot check a spelling error which occurs in the start or end of sentence because the insufficient contexts. This is also the case for other n-gram approach as well, e.g. Google Spell Checker cannot determine some of the start-of-sentence errors during testing stage. If there are future researches on this area, it is a problem which is highly recommended to solve.

Alternative Part-of-speech tagger: The pos tagger used in this project, Stanford POS Tagger can certainly tag the word with their corresponsive part-of-speech tag. However, the tag set used is pretty small (Penn Treebank tagset, Appendix B). It is not very refined compared to CLAWS7 tagset. If anyone want to adopt the approach mentioned in this project, it is recommended to use a CLAWS tagger[25].
Appendix A - Levenshtein Distance Algorithm

This algorithm measures the minimum distance among two words with deletion, insertion, substitution. A table of demonstration is shown below.

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>n</th>
<th>k</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>p</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>k</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

The algorithm started from **top left** ended in **bottom right** which is the minimum distance computed. Two red cells are corresponding to empty string which allow insertion/substitution from the start of each word. The number in each cell represents the distance from top row characters to left column character. For example, the green cell is the distance between \{empty\} and \{empty, p\} which is 1 (with insertion of p). The algorithm process as follow:

1. Filling the top left cell to 0 and the top row and left column to 1,2,3,4
2. Calculate the distance in the rest of the cells by the formula:
   \[
   \text{Current distance among two corresponding characters (0 = same, 1 = different)} + \text{Minimum distance in top, top left and left cells.}
   \]

For example, look at the distance 1 in the highlighted text, it represents the distance between \{empty, i, n\} and \{empty, p, i, n\}. The distance 1 is calculated by:

- Distance between the letter from top -> n and letter from left -> n = 0
- The minimum distance from previous cells: \min(2, 1, 2) = 1.

By filling the rest of the cells, the minimum distance among two words is calculated in the bottom right corner.
# Appendix B - Penn Treebank tag set

(From [https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html](https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html))

**Alphabetical list of part-of-speech tags used in the Penn Treebank Project:**

<table>
<thead>
<tr>
<th>Number</th>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>CC</td>
<td>Coordinating conjunction</td>
</tr>
<tr>
<td>2.</td>
<td>CD</td>
<td>Cardinal number</td>
</tr>
<tr>
<td>3.</td>
<td>DT</td>
<td>Determiner</td>
</tr>
<tr>
<td>4.</td>
<td>EX</td>
<td>Existential there</td>
</tr>
<tr>
<td>5.</td>
<td>FW</td>
<td>Foreign word</td>
</tr>
<tr>
<td>6.</td>
<td>IN</td>
<td>Preposition or subordinating conj.</td>
</tr>
<tr>
<td>7.</td>
<td>JJ</td>
<td>Adjective</td>
</tr>
<tr>
<td>8.</td>
<td>JJR</td>
<td>Adjective, comparative</td>
</tr>
<tr>
<td>9.</td>
<td>JJJS</td>
<td>Adjective, superlative</td>
</tr>
<tr>
<td>10.</td>
<td>LS</td>
<td>List item marker</td>
</tr>
<tr>
<td>11.</td>
<td>MD</td>
<td>Modal</td>
</tr>
<tr>
<td>12.</td>
<td>NN</td>
<td>Noun, singular or mass</td>
</tr>
<tr>
<td>13.</td>
<td>NNS</td>
<td>Noun, plural</td>
</tr>
<tr>
<td>14.</td>
<td>NNP</td>
<td>Proper noun, singular</td>
</tr>
<tr>
<td>15.</td>
<td>NNPS</td>
<td>Proper noun, plural</td>
</tr>
<tr>
<td>16.</td>
<td>PDT</td>
<td>Predeterminer</td>
</tr>
<tr>
<td>17.</td>
<td>POS</td>
<td>Possessive ending</td>
</tr>
<tr>
<td>18.</td>
<td>PRP</td>
<td>Personal pronoun</td>
</tr>
<tr>
<td>19.</td>
<td>PRPS</td>
<td>Possessive pronoun</td>
</tr>
<tr>
<td>20.</td>
<td>RB</td>
<td>Adverb</td>
</tr>
<tr>
<td>21.</td>
<td>RBR</td>
<td>Adverb, comparative</td>
</tr>
<tr>
<td>22.</td>
<td>RBS</td>
<td>Adverb, superlative</td>
</tr>
<tr>
<td>23.</td>
<td>RP</td>
<td>Particle</td>
</tr>
<tr>
<td>24.</td>
<td>SYM</td>
<td>Symbol</td>
</tr>
<tr>
<td>25.</td>
<td>TO</td>
<td>to</td>
</tr>
<tr>
<td>26.</td>
<td>UH</td>
<td>Interjection</td>
</tr>
<tr>
<td>27.</td>
<td>VB</td>
<td>Verb, base form</td>
</tr>
<tr>
<td>28.</td>
<td>VBD</td>
<td>Verb, past tense</td>
</tr>
<tr>
<td>29.</td>
<td>VBG</td>
<td>Verb, gerund or present participle</td>
</tr>
<tr>
<td>30.</td>
<td>VBN</td>
<td>Verb, past participle</td>
</tr>
<tr>
<td>31.</td>
<td>VBP</td>
<td>Verb, non-3rd person singular present</td>
</tr>
<tr>
<td>32.</td>
<td>VBZ</td>
<td>Verb, 3rd person singular present</td>
</tr>
<tr>
<td>33.</td>
<td>WDT</td>
<td>Wh-determiner</td>
</tr>
<tr>
<td>34.</td>
<td>WP</td>
<td>Wh-pronoun</td>
</tr>
<tr>
<td>35.</td>
<td>WP$S$</td>
<td>Possessive wh-pronoun</td>
</tr>
<tr>
<td>36.</td>
<td>WRB</td>
<td>Wh-adverb</td>
</tr>
</tbody>
</table>

*Fig. 17: A table of Penn Treebank tag set*
Appendix C - Phoneme and Self-defined Confusion Sets

The algorithm and self-defined confusion sets mentioned in 5.1.2

```java
public static void main(String[] args) throws Exception {
    String line;
    Map<String, ArrayList<String>> phonemeMap = new HashMap<String, ArrayList<String>>();
    Map<String, ArrayList<String>> confusionMap = new HashMap<String, ArrayList<String>>();
    InputStream fis = new FileInputStream("input/phoneme.txt");
    InputStreamReader isr = new InputStreamReader(fis, Charset.forName("UTF-8"));
    BufferedReader br = new BufferedReader(isr);
    while ((line = br.readLine()) != null) {
        String[] parts = line.split(" ", 2);
        String right = parts[0];
        String left = parts[1];
        if (right.endsWith("\""))
            right = right.substring(0, right.length() - 3);
        if (!phonemeMap.containsKey(left))
            ArrayList<String> inList = new ArrayList<String>();
            inList.add(right);
            phonemeMap.put(left, inList);
        else
            ArrayList<String> inList = phonemeMap.get(left);
            inList.add(right);
            phonemeMap.put(left, inList);
    }
    br.close();
    PrintWriter writer = new PrintWriter("input/pconfusion.txt", "UTF-8");
    for (String current : phonemeMap.keySet())
        ArrayList<String> inList = phonemeMap.get(current);
        if (inList.size() > 1)
            for (String word : inList)
                ArrayList<String> wordList = new ArrayList<String>(inList);
                wordList.remove(wordList.indexOf(word));
                if (!confusionMap.containsKey(word))
                    confusionMap.put(word, wordList);
                else
                    ArrayList<String> thisList = confusionMap.get(word);
                    for (String temp : wordList)
                        if (!thisList.contains(temp))
                            thisList.add(temp);
                            confusionMap.put(word, thisList);
    }

    Map<String, ArrayList<String>> treeMap = new TreeMap<String, ArrayList<String>>(confusionMap);
    for (String key : treeMap.keySet())
        writer.println(key.toUpperCase() + " ");
        int count = 1;
        for (String words : treeMap.get(key))
            writer.println(words.toUpperCase());
            if (count < treeMap.get(key).size())
                writer.print(" ");
                count++;
        writer.println();
    writer.close();
}
```

Fig. 18: A program code which creates confusion sets based on word phoneme
Fig. 19: A list of self-defined confusion set

am are, is
are am, is
is am, are
was were
were was
borrow lend
lend borrow
say tell
tell say
can may
may can
accept except
except accept
begin being
being begin
among between
between among|
Appendix D - Sample Database Structure

The sample database structure described in 6.2

### Fig. 20: Database Structure of dictionary database

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Type</th>
<th>Collation</th>
<th>Attributes</th>
<th>Null</th>
<th>Default</th>
<th>Comments</th>
<th>Extra</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>word</td>
<td>varchar[255]</td>
<td>utf8_unicode_ci</td>
<td>No</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Fig. 21: Database Structure of word trigram database

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Type</th>
<th>Collation</th>
<th>Attributes</th>
<th>Null</th>
<th>Default</th>
<th>Comments</th>
<th>Extra</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>word1</td>
<td>varchar(200)</td>
<td>ascii_bin</td>
<td>No</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>word2</td>
<td>varchar(200)</td>
<td>ascii_bin</td>
<td>No</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>word3</td>
<td>varchar(200)</td>
<td>ascii_bin</td>
<td>No</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>frequency</td>
<td>int(11)</td>
<td></td>
<td>No</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Fig. 22: Database Structure of 1st Pos database

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Type</th>
<th>Collation</th>
<th>Attributes</th>
<th>Null</th>
<th>Default</th>
<th>Comments</th>
<th>Extra</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pos</td>
<td>varchar[255]</td>
<td>utf8_unicode_ci</td>
<td>No</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>word2</td>
<td>varchar[255]</td>
<td>ascii_bin</td>
<td>No</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>word3</td>
<td>varchar[255]</td>
<td>ascii_bin</td>
<td>No</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>frequency</td>
<td>int(16)</td>
<td></td>
<td>No</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix E - Confusion Matrices

The following shows the confusion matrix result on text with no spelling:

<table>
<thead>
<tr>
<th>Actual Class (True Label)</th>
<th>Predicted Class (Labeled by Spell Checker)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>305</td>
</tr>
<tr>
<td>No</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6: Confusion Matrix - Word trigram approach

The following shows the confusion matrix result on text with real-word spelling error:

<table>
<thead>
<tr>
<th>Actual Class (True Label)</th>
<th>Predicted Class (Labeled by Spell Checker)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>372</td>
</tr>
<tr>
<td>No</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 8: Confusion Matrix - Word trigram approach

<table>
<thead>
<tr>
<th>Actual Class (True Label)</th>
<th>Predicted Class (Labeled by Spell Checker)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>375</td>
</tr>
<tr>
<td>No</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 9: Confusion Matrix - Mixed POS trigram approach
Break down for real-word spelling test on word trigram approach:

<table>
<thead>
<tr>
<th>Total: 159 words</th>
<th>Predicted Class (Labeled by Spell Checker)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Actual Class (True Label)</td>
<td>Yes</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>8</td>
</tr>
</tbody>
</table>

**Table 10**: Confusion Matrix - Word Distance test

<table>
<thead>
<tr>
<th>Total: 136 words</th>
<th>Predicted Class (Labeled by Spell Checker)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Actual Class (True Label)</td>
<td>Yes</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 11**: Confusion Matrix - Word Permutation test

<table>
<thead>
<tr>
<th>Total: 98 words</th>
<th>Predicted Class (Labeled by Spell Checker)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Actual Class (True Label)</td>
<td>Yes</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table 12**: Confusion Matrix - Phoneme test

<table>
<thead>
<tr>
<th>Total: 132 words</th>
<th>Predicted Class (Labeled by Spell Checker)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Actual Class (True Label)</td>
<td>Yes</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 13**: Confusion Matrix - Self-defined set test
Break down for real-word spelling test on Mixed POS trigram approach:

<table>
<thead>
<tr>
<th>Actual Class (True Label)</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>117</td>
<td>23</td>
</tr>
<tr>
<td>No</td>
<td>8</td>
<td>11 (5 corrected)</td>
</tr>
</tbody>
</table>

Table 14: Confusion Matrix - Word Distance test

<table>
<thead>
<tr>
<th>Actual Class (True Label)</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>103</td>
<td>22</td>
</tr>
<tr>
<td>No</td>
<td>4</td>
<td>7 (4 corrected)</td>
</tr>
</tbody>
</table>

Table 15: Confusion Matrix - Word Permutation test

<table>
<thead>
<tr>
<th>Actual Class (True Label)</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>70</td>
<td>18</td>
</tr>
<tr>
<td>No</td>
<td>2</td>
<td>8 (4 corrected)</td>
</tr>
</tbody>
</table>

Table 16: Confusion Matrix - Phoneme test

<table>
<thead>
<tr>
<th>Actual Class (True Label)</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>85</td>
<td>35</td>
</tr>
<tr>
<td>No</td>
<td>4</td>
<td>8 (6 corrected)</td>
</tr>
</tbody>
</table>

Table 17: Confusion Matrix - Self-defined set test
Reference


[4]: Damerau, F.J. (1964) A technique for computer detection and correction of spelling errors.


[7]: Andrew R. Golding (1995), A Bayesian hybrid method for context-sensitive spelling correction


[12]: C. Whitelaw, B. Hutchinson, GY Chung, G. Ellis (2009), Using the Web for Language Independent Spellchecking and Autocorrection


