Addis Ababa University,
College of Natural and Computational Sciences
School of Information Science

Amharic Spelling Error Detection and Correction System:

Morphology-based Approach

Mariawit Shimelis

A thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science in Information Science

Addis Ababa University

Sept, 2020

Addis Ababa University,
College of Natural and Computational Sciences
School of Information Science

Amharic Spelling Error Detection and Correction System:

Morphology-based Approach

Mariawit Shimelis

Name and signature of Members of the Examining Board

Michael Melese (PhD)	_____	_____
Advisor	Signature	Date
Martha Yifiru (PhD)	_____	_____
Examiner	Signature	Date
Solomon Teferra (PhD)	_____	_____
Examiner	Signature	Date

Declaration

This thesis is my original work and has not been submitted as a partial requirement for a degree in any university.

_____

Mariawit Shimelis Ibssa

The thesis has been submitted for examination with my approval as university advisor.

_____

Michael Melese (PhD)

In memory of my mom, Etew

… who has sacrificed her life to build mine

Acknowledgement

Before all, I would like to thank God and Virgin Mary for leading and following me in each steps, for giving me the strength when I get weak, for showing me the way when I get lost and for giving me grace to present my work.

Next, I would like to thank my advisor Dr. Michael Melese for supporting me from improving my title to reading my document line by line and giving me comments. I also I wish to show my gratitude to Dr. Melkamu for encouraging me to go forward.

I wish to express my deepest gratitude to my husband Ermy and my sister Abuti for supporting and encouraging me in my journey of choosing a topic to presenting my final defense. Not forgetting my dad and other family members who encouraged me to go on. I also want to thank my friend Yodit for her kind support.

List of Tables

List of Figures

List of Listings

Abstract

Nowadays, it is a common practice to notice spelling errors in typed Amharic documents. As Amharic is the official working language of Ethiopia and the second most spoken Semitic language in the world, the need to have an Amharic spelling error detection and correction system is evident.

Previous attempts are made to develop a spell checker for Amharic. These works have attempted to use existing approaches likes metaphone and edit distance algorithms. However, these approaches are designed for languages which have a simple morphology like English. Moreover, unlike Amharic, in other languages distance between words is not dependent on the family and order of characters. As a result, using these approaches for a language with complex morphology like Amharic will not give the anticipated result. Similarly, using existing morphological analyzers for computational morphology is attempted in other works. Though the analyzers were reported to work with reasonable accuracy for valid words, their output for misspelled words is not clear. Accordingly, this study attempts to investigate the possibility of using morphology-based approach to design and develop an Amharic typing error detection and correction system for non-word errors.

Design science methodology is employed in this study. It involves six activities namely, problem identification and motivation, defining objectives, design and development, demonstration, evaluation and communication. To carry out the experiment 717 morphological rules are defined, 2398 stem words selected from each category of root words are stored and the system has been tested with 1724 words selected from different derivational and inflectional categories. A prototype is developed using python programming language and it uses three knowledge bases which are stored in csv format. To evaluate the system, evaluation metrics precision, recall and predictive accuracy are used.

The experimental results show 96% Lexical Recall, 89% Error Recall, 99% Lexical Precision, 70% Error Precision, 95% Predictive Accuracy and 70% corrections are generated for correctly identified invalid words. This shows that the system has high accuracy in flagging words as valid/ invalid and needs some improvement in suggestion generation.

The system gives a good accuracy for selected words (with complex morphology) which are representative of words in the languages. Accordingly, it is concluded that the system is capable of detecting and correcting errors as long as the correct rule definition is defined and the corresponding stem is found inside the dictionary. It is also concluded that by applying the algorithms proposed for morphological analysis and distance calculation, morphology-based approach is suitable for Amharic spell checking than other approaches.

For future works, improving rule definitions by including word classes, handling exceptions, including additional spell checker functionalities, expanding the work to include real-word errors and applying the proposed architecture for Amharic-English translation systems are recommended.

***Keywords**: Spell Checker, Morphology-based approach, n-gram, Morphology*

Table of Contents

CHAPTER ONE

INTRODUCTION

1.1. Background

Communication is the exchange of information by speaking, writing, or using some other medium [1]. One of the methods of communication among humans is language. It is either spoken or written, consisting of the use of words in a structured and conventional way. Out of languages spoken in the world, over 85 of them (which are in different language branches) are spoken in Ethiopia [2]. Linguistic experts divide Ethiopian languages into Semitic, Cushitic, Omotic and Nilo-Saharan branches [3].

Among the Semitic languages, Amharic is the official working language of Ethiopia government and the second most-spoken Semitic language in the world, next to Arabic [4]. It is an Afro-Asiatic language of the Semitic branch and is a member of the Ethio-semitic group.

It is a common practice to use word processing applications to type Amharic texts. The focus of this research is in one of the integrated functionalities in word processing application called spell checking. Spell checker is one of the areas of natural language processing applications and it is a tool that checks if words are written in the correct spelling in a text file and suggests possible alternatives for wrongly spelled words [5].

Spell checking involves two steps, namely spelling error detection and spelling error correction. In the first step, a word is identified as correctly spelled or not. In the second step, corrections close to the misspelled word are given and replaced with the correct word [6][7]. One of the main tasks in spell checking is computational morphology. It is the study of morphological analysis and generation of words.

There are two general approaches in computational morphology: rule-based and corpus-based approaches [8] [9]. The rule-based approach involves storing lexicon and morphological rules of the language. In order for this approach to be effective, rules that define the morphology of the language have to be defined with the involvement of a linguistic expert [8]. This approach is very effective as long as all rules are defined and every lexicon is stored [9]. The main drawback with this approach is defining every rule, which is costly and time-consuming [8][9].

Corpus-based approach involves supervised or unsupervised machine learning from annotated or unannotated corpus respectively [8]. In this approach, linguistic knowledge is extracted from the given corpus and trained to the system [8]. Supervised approach learns by example from an annotated corpora while unsupervised approach learns from raw unannotated corpora [8] [9]. Some of the supervised machine learning approaches are Inductive Logic Reasoning, Support Vector Machine, Hidden Markov Model, Memory-based Learning [8].

Among the main techniques identified for spelling error detection, dictionary look-up involves checking whether a word is correct or not by searching it in a stored knowledge base of roots, derivations and inflections [10]. A word not found in the collection of words is flagged as invalid. The disadvantages with this technique arises because storing every word in the language is a challenging task, especially for complex languages like Amharic. In addition, searching a single word in large set of words becomes time-consuming [11]. To solve this problem, a technique called hashing is proposed. In hashing, each word is given an address or a key and instead of searching for the word, the system searches for a key related with the word inside a hash table. Even if this technique improves the efficiency in dictionary look-up technique, it still doesn't apply to Amharic because it has a very complex derivational and inflectional morphology [12].

Another error detection approach is n-gram analysis. It works by comparing sequence of n-letter subsequences with n-gram frequencies which are stored in an n-dimensional matrix [13]. If a unigram, bigram or trigram is missing, the word is flagged as invalid. This technique requires n-gram statistics generated from large corpora. This approach cannot be applied to Amharic because it needs a large corpus and such a large corpus is not currently available for morphological rich languages like Amharic [9].

Morphology-based approach is another spell checking approach [5] [12] [14]. It involves formulation of morphological analysis and generation rules and testing the given word against the list of rules. It uses morphological rules to remove affixes and reverse the word to its root [12]. In most cases, it uses a dictionary of root words (lexicon) to search for the resulting root after morphological analysis [12].

Among the approaches for spelling error correction, Edit Distance, n-gram-based Technique, Probabilistic technique, Rule-based Technique, Similarity Keys and Neural Networks can be mentioned.

Edit Distance Technique works by calculating the distance (difference between the words) by comparing the length, order of characters, first and last characters and other features of the words [10].

N-gram based technique can be used in three manners: error detection, candidate suggestion and rank similarity. N-gram approach helps to avoid the ambiguity that is occurred when two or more generated words have equal distance. It works by comparing the previous and next words in the given paragraph [10] while probabilistic technique works based on some statistical features of the language. There are different types of probabilistic techniques. For example, transition probabilities give the probability of a given letter or sequence of letters being followed by any other given letter [11].

Rule-based Approach for spelling correction involves defining a set of rules that capture common spelling and typographic errors and applying these rules to the misspelled word [11]. In similarity key technique, every word is assigned a key and similar words will have similar or the same keys. The key to the misspelled word is computed and words with similar key values to it will be generated as candidate suggestions for the misspelled word [7] [11]. Noisy Channel Model is a neural network model that works by treating the misspelled word as if a correctly spelled word had been misrepresented when it passes through noisy communication channel [15].

In this study, an attempt is made to study existing spell checking approaches, identify benefits and drawbacks and apply morphology-based approach for detection and correction of spelling errors in Amharic.

1.2. Motivation

It is a very common practice to see a lot of spelling errors on typed Amharic documents. Most people who type Amharic make mistakes and do not even notice it. In printing presses, they have a dedicated personnel to go through typed documents, identify wrongly spelled words and make corrections. Even after this process, it is still very common to see spelling errors on printed materials. This implies the need for an efficient and accurate spell checker that can be integrated with word processing applications.

Previous attempts are made for the development of an Amharic spell checker. One of the attempts was made using a hybrid approach [16]. The proposed system uses metaphone

algorithm for error detection and edit distance algorithm for error correction. Metaphone algorithm is specifically designed for languages like English [13]. Amharic has highly complex morphology compared to English. The huge difference in the nature of morphology of the two languages and the language branch they belong to is not considered in this study.

Edit distance algorithm is used for correction of invalid words in this research. Edit distance algorithm requires a large set of stems, inflected and derived words. This is difficult to achieve because storing all root and derived words is a cumbersome task. A single Amharic word can have up to 2000 derivations. The Amharic dictionary published by Ethiopian Language Studies contains 7127 verbs, 11,359 nouns, 4122 adjectives, and 239 adverbs [17]. That makes the number of root words more than 22,000. The application of such complex derivations on 22,000 root words would result in a huge number of derivations. That means one has to store millions of words in order for the error corrector to work. Searching a word closer to a single word in this large set of words is very time-consuming and inefficient. In addition, finding and collecting all words in Amharic is very challenging because the amount of Amharic corpus found online is very small compared to the whole set [9].

The same gap exists for another attempt that uses dictionary look-up approach with hashing to build a multi-lingual spell checker for five Ethiopian languages [18]. In this study, to detect and correct an invalid word, the process requires to search the dictionary of list of words twice i.e. to search for a matching word (detection) and to look for a closer word to the misspelled word (correction). Although the system proposes hashing for better performance, it is still a lot of work given the huge number of derivations of Amharic.

Another attempt uses morphological information of Amharic to detect Amharic spelling errors [12]. The study is focused on detection of spelling errors in Amharic verbs only. Correction of errors and spelling error detection in other parts of speech is out of the scope of this study. A morphology-based approach is also used for Afaan Oromo which is found in the Cushitic branch of languages [5]. The attempt uses morphology-based approach to detect errors. However, the same proposed architecture cannot be applied to Amharic because of the difference in the morphological nature of the two languages. Amharic is a Semitic language and Afaan Oromo is under Cushitic branch. Even if both exhibit a complex morphology, the fact that the two languages are in different language branches makes their morphology different. In addition, Latin characters are used to type text in Afaan Oromo. As a result, using edit distance for ranking suggestions may work well. This is not the case in Amharic because

it is written with Ge'ez script and using edit distance algorithm to calculate distance between two Amharic words does not give the intended result (Section 5.2.2).

This drawback with Levenshtein edit distance approach to generate suggestions is not taken into consideration in other two Amharic spell checking studies [13] [19] [20]. One of these attempts applies morphological analyzer tool for detection and edit distance for correction of errors [20]. This study does not consider words with more than one prefixes/ suffixes.

In this study, available morphological analyzers for Amharic are not employed. Though the analyzers were reported to work with reasonable accuracy for valid words, their outputs for invalid (misspelled) words are not clear. Assuming the analyzers show that a misspelled word cannot be analyzed and give a definite answer, the response will help only to identify the validity of the word. However, the spelling corrector requires the whole path of the analysis (not only the final outputs) to generate appropriate suggestions.

This study intends to solve problems and drawbacks with existing spell checking research works (discussed above) and propose morphology-based approach for detection and correction of Amharic typing errors. It uses a manually stored set of morphological rules to identify spelling errors and give closer suggestions for invalid words. It also intends to solve the problem with efficiency of morphology-based systems by applying an algorithm that filters suitable rules and searches stem words based on the POS it belongs to and the number of characters it contains. It also presents the problem with existing distance calculation algorithms and proposes a new distance calculating algorithm that is particularly designed for Amharic (Section 4.3 vi).

1.3. Problem Statement

Semitic languages like Amharic are characterized by their relatively complex morphology [21]. In addition to the complexity, the amount of resources about the language are minimal. Amharic is one of the morphologically complex languages with a root-pattern morphology. Root is defined as "a set of consonants, which has a basic lexical meaning" [8]. A single root word (verb, adjective, adverb or noun) can take prefix, suffix or affixes and result in a significant number of derivations (Appendix 5). The resulting derivations could be phrases or even sentences with a full meaning [8].

Amharic uses different affixes to create inflectional and derivational morpheme. Affixation can be prefix, infix, suffix, and circumfix [8]. Amharic verbs can take up to four prefixes and five suffixes, and the affixes exhibit a complex set of co-occurrence rules [22].

Derived verbs in Amharic reflect number, gender, tense, aspect and mood. Amharic derivations can be categorized into several classifications. Passive, causative, reciprocal, Affirmative/ negative, relativization are some of the classifications. Verbs can also take prepositions and conjunctions [22].

Example

Root word = ል-ም-ድ *(l-m-d)* , **Stem = ለመድ-** *(lämädä-)*

እየተለማመዳቸሁባቸውና = **እየ** + **ተ** + ለ*ማመ*ድ + **አቸሁ** + **ባቸው** + **ና**

əyä + tä + lämamäd + ačəhu + bačäwə + na

Prefixes **Suffixes**

In addition to the morphological complexity, Amharic language has several ambiguities in the writing system [13]. In Amharic writing, there are several phonetic redundancies. For instance, the character *ha* can be represented by: ሀ, ሃ, ሐ, ሓ, ኀ, ኃ, and ኻ; the character *sə* can be represented by ሰ, ሠ; the character '*a*' can be represented by አ, ኣ, ዐ, ዓ; the character *ṣä* can be represented by ጸ, ፀ. Amharic also has visually redundant characters like ው and ዉ, ኘ and ኙ, ፖ and ፓ, ዩ and ዪ, ጉ and ጕ, which are used interchangeably. There are occasions where there is a substitution of characters. For example, ም is substituted for ን when it comes before በ. For instance, አንበሳ */anbäsa/* is also written as አምበሳ */ambäsa/*. Moreover, on some occasions, prefix omissions occur when a consonant is followed by a vowel. For example, የአየር */yäayärə/* is also written as ያየር */yayärə/*.

Writers might use ge'ez characters in place of canonical Amharic characters. For example, one might type ኩሉ to type ሁሉ. Another misspelling occurs when writers use both Amharic and Ge'ez pluralization technique. For example, to change ቀለም to its plural form, one has to use ቀለሞች (Amharic form) or ቀለማት (Ge'ez format). However, people usually make a mistake of using both and writing ቀለማቶች. On top of this, sounds which are made with lips and softpalate (labiovelars) like ዋ could be written separated in words like እሷ which can also be written as እስዋ.

The same word can have different dialects and such words can be written in two or more different forms. e.g. ዓጤ Vs ዓፄ. In addition, words that are taken from foreign languages may not be written as the native speakers pronounce it. e.g. ካርቦን Vs ካርበን.

Amharic related systems are challenging to develop because of the above-mentioned facts.

Because of the complexity of the language, the main challenge in Amharic spell checking arises when it tries to detect errors in derived and inflected words [12]. The spell checker uses a dictionary of words at some point [12], but storing every word in the language is time consuming and costly task. Accordingly, finding a way to identify spelling errors for derived and inflected words and storing only stem words is a better option. As discussed in Section 1.2, previous research works tend to have gaps in some way. The study aims to fill the gap with previous works and propose a better approach for improvement.

The study attempts to apply and evaluate morphology-based approach (morphological rules of the language) for Amharic spelling error detection and correction. It evaluates the capability of calculating distance between Amharic words by experimenting on existing distance calculation algorithms. The study proposes an efficient way for rule filtering and searching stems. It also proposes a new distance calculator algorithm specially designed for Amharic.

This study answers the following research questions:

- To what extent spelling errors in Amharic can be detected and corrected using morphology-based approach?
- Is morphology-based approach more suitable for Amharic spelling error detection and correction than other spell checking approaches?

1.4. Objective

1.4.1. General Objective

The general objective of the study is to investigate the possibility of using morphology-based approach to design and develop an Amharic spelling[1] or typing error detection and correction system.

[1] *In this research, the term 'spelling' is used to refer to the order of characters that give a meaningful word. The author has used the word 'Spelling Error' for Amharic in the same context. However, since each sound in Amharic has its own dedicated character, it does not apply to Amharic. People who type Amharic make 'Typographic Error' not 'Spelling Error'. Due to its common use, the phrase 'Spelling Error' is used instead of 'Typing Error'.*

1.4.2. Specific Objectives

The specific objectives of the research are to:

- Review related research works to identify the state-of-the art in spell checking.
- Define rules that represent prepositions, prefixes, suffixes, conjunction and derivation patterns of inflected and derived words.
- Design an architecture to detect incorrectly spelled words and suggest closer corrections.
- Prepare test cases from different Amharic documents and texts.
- Develop an algorithm that identifies incorrectly spelled Amharic words as invalid, generates closer suggestions to the misspelled word and ranks suggested words.
- Develop a prototype that implements the designed architecture and the defined morphological rules.
- Test and evaluate the prototype against the test cases.

1.5. Significance of the Study

This study is significant in such a way that it fills the gap in using morphology-based approach for Amharic spell checking by proposing an algorithm that handles the complex morphology of Amharic and the complex nature of having multiple affixes in a single word. It also avoids the problem with suggestion generation by proposing a new distance calculator that works particularly for Amharic.

It also contributes as an input for further research works in the same area. The architecture proposed in this research can be reused or improved and used for future works. Moreover, it opens path for more advanced applications in which contextual checking would be included.

It is the author's strong belief that it would be a good input for other spell-checking researches and applications. It could also help as an input for translation studies from Amharic to other languages and vice versa. Amharic-English printed dictionaries give the translation of stem words only. Studies for translation of derived words can follow the morphological rules

presented in this study. Derived words which have the same morphological pattern would follow the same translation pattern. For instance, 'እየበላሁ ነው።።' can be translated to English as 'I am eating'. Similarly, a sentence which contains the same morphological rule like 'እየሰማሁ ነው።።' can be translated by the system as 'I am listening'.

1.6. Scope and Limitation of the Study

This study only covers detection and correction of Amharic non-word errors on a word processing application. It identifies correctly spelled words as valid and incorrectly spelled words as invalid and gives a closer suggestion to misspelled words. Real-word errors are out of the scope of this research. The research is mainly focused on non-word errors with common derivations. Words with unique and exceptional derivations are not considered here. Though one of the reasons to choose python for implementation is to integrate the system with word processing applications, the integration is not through due to time constraint. Due to the same reason, additional functionalities of a spell checker other than detection and correction of errors is not implemented. The prototype is a stand-alone application that has its own GUI to enter candidate words and generate suggestions for invalid ones.

All morphological rules and all stem words are not stored for this study. However, a total of 717 morphological rules which are representative of noun and verbal derivation patterns (Section 3.2 iv) are defined. These rules are representative of more than 13 complex derivations patterns defined by linguistic experts [23][24]. This can assure that a system tested with such rules will have a similar performance when more rules are added.

The researcher was not able to find a linguistic expert that can assist in defining rules. However, books and articles written by linguistic experts are referenced and used. As a result, the rules are defined from facts taken such books and articles [23] [24] [25]. There might be a possibility of making a mistake in the rule definitions because the expertise of the researcher is not in linguistic area.

There is no standard to write compound words in Amharic [26]. They can be written in three ways: (a) separated by a hyphen like ህገ-መንግስት and ስነ-ምግባር, (b) separated by a space like ትምህርት ቤት and ሆደ ሰፊ [24], (c) concatenated in one word like ብረትምጣድ ዓለምአቀፍ [23]. In this study, it is assumed that compound words are written either separated by a hyphen or as a single word. Compound words separated by a space are not considered.

The average processing time to spell check a word was not measured. Because, the main aim of this research is to evaluate the capability of morphology-based approach in detecting and correcting errors.

1.7. Organization of the Thesis

This research paper is organized in six different but interrelated chapters. The first chapter starts by discussing the background of the research work including Amharic language, spell checking approaches followed by motivation of the study, statement of the problem, objective of the study including general and specific, methodology, significance of the study and finally scope and limitation of the study are discussed in detail.

Chapter two discusses basic theoretical concepts about the Amharic language, its morphology, spelling error types, existing spell checking approaches and spell checking algorithms for Amharic and other languages. Chapter three discusses the research methodology employed, the process model which includes explanation of the preparation of test cases, preparation of the dictionary of stems and evaluation metrics used. Chapter four discusses the general architecture and design, the designed Amharic spelling error detection and correction algorithm and the rule definition structure designed as an input for the algorithm. Moreover, stored knowledge base components used by the prototype are discussed. Chapter five discusses preliminary experiment and analysis performed prior to the main experiment, development tools used, prototype used to test the system, evaluation and test results found from the main experiment. Chapter six presents conclusions and recommendations concluded from the experimental results.

CHAPTER TWO

LITERATURE REVIEW

2.1 Overview

This chapter discusses basic theoretical concepts about the Amharic language, its morphology, spelling error types, existing spell checking approaches and spell checking algorithms for Amharic and other languages. Section 2.2 discusses the origin of the Amharic language and its morphological nature. Section 2.3 discusses general spelling error types for other languages and Amharic. Section 2.4 explains the different spelling error detection and correction approaches. Section 2.5 presents existing algorithms developed for spell checking. Section 2.6 presents review of existing spell checking research works for Amharic, Afaan Oromo and Arabic. Morphological analysis works by other scholars is as well discussed as part of the related works section.

2.2 Amharic language

Amharic is an Afro-asiatic language of the Semitic branch and is a member of the Ethio-semitic group [7]. A research shows that studying the origin of Amharic indicates that it is a hybrid of Semitic, Cushitic, Omotic and Nilo-Saharan language, even though its Semitic nature dominates [27]. Looking at the history of the origin of Amharic, the birth of Amharic starts when Semitic-speaking people start to build South Arabic civilization in Eritrea and Axum in the first three centuries A.D.

In the middle of the fourth century, the mission reached the region that was later recognized as Amhara [27]. The military comprised of Semitic-speaking leaders, who speak pre-Amharic (originated from Ge'ez), large number of Agew, Omotic and Cushitic speaking military members and Nilo-Saharan speaking servants. This diverse combination of languages eventually creates a language with Semitic Lexicon and Cushomotic syntax namely Amharic.

Ge'ez is a purely Semitic language and its origin goes back to South Arabia [27]. As Ge'ez being a parent of Amharic, this relates Amharic to Arabic and Tigrigna. However, the syntax of Amharic resembles Oromo, Somali, Welamo and other languages of neighboring Cushitic and Omotic groups than Tigrinya, Ge'ez or Classical Arabic.

2.2.1 Morphology of Amharic

Semitic languages like Amharic are characterized by their complex morphology. It is related to other Semitic languages such as Hebrew, Arabic and Syrian. It is the second populous Semitic language, after Arabic [9].

Amharic is written using a script that grew out of the Ge'ez abugida called ‘Fidel' and ‘Abugida'. The character set contains 33 base characters vertically listed with seven columns, which gives 231 characters in total. About 310 characters are used to type Amharic [2]. Amharic is written from left to right separated by double dots (:) or white space [13].

Amharic language is characterized by its complex morphology and it exhibits a root-pattern morphology. Root comprises of a set of consonants with a basic lexical meaning [7]. Consonants /Radicals/ in Amharic are six-order characters on the Amharic writing system, ‘fidel’. Roots are divided into four basic root types [25].

	Root Type	**Example**
1	1-2-3	s-b-r-
2	1-1-2	s-s-t-
3	1-2-2	I-k'-k'-
4	1-2-1	w-r-w-

Table 2.1 Basic root types

The first type of root consists of three different sixth order character, whereas the second one comprises of two similar sixth order characters followed by another one. The third one comprises of a sixth order character followed by two similar sixth order characters. The forth one consists of a sixth order character followed by another character, again followed by a character which is the same as the first one.

Parts of speech in Amharic can be categorized into six: nouns, verbs, adjectives, pronouns, prepositions and conjunctions [24].

The first four categories give a sense or pictorial image to the receiver of the message by themselves, whereas prepositions and conjunctions do not give sense of meaning without combining with other parts of speech [24]. However, when used in combination with words that stand by themselves, they partake in functionalities like comparison. For example, the word እንደ /*endä*/ does not give meaning by itself. However, if the noun አንበሳ /*anbäsä*/ comes after it, it gives the meaning ‘like a lion’.

Words that stand by themselves have a tendency to change their shape and the others do not [24]. For example, appending the suffix -ው /-*w*/ at the end of አንበሳ /*anbäsä*/ gives አንበሳው /*anbäsäw*/ - 'the lion', which implies the speaker is talking about a specific lion. Such words can change their shape to show number, gender, aspect, tense, mood and person [24] [9].

Words that stand by themselves can change their shape and produce new words. For example, the word አንበሳ /anbäsä/ can take the suffix -ነት /nät/ and give አንበሳነት /anbäsänät/, which gives the meaning 'being a lion'.

Verbs

Verbs in Amharic are characterized by two special features [23]. First, it appears at the end of a sentence and second, it contains a bound morpheme that indicates the subject of the action. For Example, in the sentence ብርጭቆ ሰበረ, the verb ሰበረ /säbbärä/ = ሰበር-ኧ /säbbär- ä/ is located at the end of the sentence and the bound morpheme '- ä' shows that the subject of the action is 'he'. The following table shows bound morphemes that express the subject of a verb.

Word		**Suffix**	**Subject**
ሰበረች	säbbär-äč	-äč	She
ሰበረ	säbbär-ä	-ä	He
ሰበሩ	säbbär-u	-u	They
ሰበርን	säbbärr-n	-n	We
ሰበርሽ	säbbärr-š	-š	you /female/
ሰበርክ	säbbär-k	-k	you /male/

Table 2.2 Bound morphemes that express subject of a verb

As shown on Table 2.2, bound morphemes *-äč, -ä, -u, -n, -š* and *-k* express the subjects *She, He, They, We, You/female/* and *You/male/* respectively.

It is claimed that verbs can be divided into three based on the number of consonantal radicals in their stem: bi-radicals, tri-radicals and quadri-radicals [23]. On the contrary, another article claims the number of consonantal radicals ranges from one to six. An argument is presented to this claim and shows that all verbs are derivations of tri-radical verbs [25].

Apart from the disagreement of the linguistic experts, the following are the known/identified types of verbs based on the number of consonantal radicals. Amharic verbs can have up to six root consonants.

1) **Mono-radical Verbs** [25]
2) **Bi-radical verbs** – contain two characters after passing through the process of root-reduction from three-radical verbs [9] [25]
3) **Tri-radical verbs** – contain three characters and they are the basic types or the origins of other types of verbs [9] [25]
4) **Quadri-radical verbs** contain four characters and they are extended from three-radical verbs [9] [25]
5) **Quinqui-radical verbs** contain five characters and they are extended from three-radical verbs [25]
6) **Sexi-radicals** contain six characters and they are extended from three-radical verbs [25]

The following table shows examples of the different types of verbs.

Type of Verb	Example	Phonetic Representation	Translation
Mono-radical	ሻ-	-ša-	desire
Bi-radical	ተው- ሸጥ-	täw- š-t'-	leave sell
Tri-radical	ሰብር- ለምድ-	säbr- lämd-	break get used to
Quadri-radical	ገልበጥ- ሰልችት-	gälbɨt'- sälšɨt'-	turn over fed up
Quinqui-radical	ሸቅንጥር-	-šk'änät'ɨr-	through away violently
Sexi-radical			

Table 2.3 Types of Verbs based on their number of radicals (Adopted from [25])

Stem Formulation

Tri-radical verbs go through Internal extension (germination and reduplication) and external extension (epenthesis) processes and give stems with more than or less than three radicals[25].

I. Extension

Extension is a process of formation of stems from root verbs having more than three radicals [25]. The process is categorized into internal extension (germination and reduplication) and external extension (epenthesis) [25] [9].

a) Internal Extension

Internal extension (germination and reduplication) of radicals of tri-radical roots articulates verbal features like aspect (perfective) and adverbial features like manner (intensity, iterativity, and attenuativity) [25].

i) Gemination

Gemination is a kind of root extension that involves germination of the ultimate radical (attenuative) or both the ultimate and the penult radicals[2] (intensive) [25]. For example, in the root verb s-b-r, the ultimate radical –r- and the penult radical -b- change their form to create the intensive stem sɨbbɨrr-. Intensive stems insert the epenthetic vowel ɨ in the germination process. The ultimate radical –r- changed its form to create the attenuative stem säbärr-. In order to give a meaning by themselves, intensive and attenuative stems take the auxiliary verb *al-* to form compound forms like sɨbbɨrr alä, sɨbbɨrr- aläč, säbärr alä, säbärr aläč.

ii) Reduplication

Reduplication is a kind of root extension that indicates an iterative action. It is in turn divided into two [25].

Total reduplication is formed by reduplicating attenuative and intensive stems [25]. For instance, the intensive and attenuative stems sɨbbɨrr- and säbärr- can extend to sɨbbɨrr- sɨbbɨrr- and säbärr- säbärr- to form total reduplications. Such extensions combine with auxiliary verbs *aderreg-* and *al-* to form compound forms like sɨbbɨrr- sɨbbɨrr- aderregä and säbärr- säbärr- alä.

Partial reduplications reduplicate part of their radicals to show intensity of actions [25]. There are three types.

In *first degree partial reduplication*, the penult radical is reduplicated [25]. For example, s-b-r- goes through first degree partial reduplication to create the iterative perfective stem säbabbär- which gives the meaning ‘break repeatedly’.

In *second degree partial reduplication,* ultimate and penult radicals are reduplicated and penult radical is geminated [25]. For example, s-b-r- goes through second degree partial reduplication to create the iterative intensive stem sɨbɨrbɨrr- which gives the meaning ‘break into pieces

[2] Ultimate radicals are consonants that are found at the end of the root verb and penult radicals are situated before the ultimate radical.

completely'. Second degree reduplications combine with auxiliary verbs *aderreg-* and *al-* to form compound forms.

In *third degree partial reduplication*, radicals continue duplicating until the speaker feels they have expressed the intensity of the action [25]. For example, s-b-r- goes through third degree partial reduplication to create the iterative intensive stem sɨbɨrbɨrbɨrbɨrbɨrbɨrr- which gives the meaning 'break into pieces and pieces'.

b) External extension (epenthesis)

External extension involves inserting an additional radical just before the initial radical and reduplicating the penult radical [25]. For instance, the root k'-d-m- passes through internal extension to form k'-d-d-m- and the external extension –š-k'-d-d-m- is formed by inserting the radical š before the initial radical –k'-. Perfective stems like a-škädaddäm- can be derived from external extension stems.

II. Reduction

Reduction is a process of formation of stems from root verbs having less than three radicals [25]. It is the reduction of the consonantal radicals of laryngeals and glides also called gutturals or weak radicals (h, y, w, ?) and two additional weak radicals (b and r). When roots are reduced to verbals, the laryngeals and glides radicals are omitted and they reappear in their parallel deverbals (surface forms). The following table taken from [25] shows formation of stems by the reduction process. The characters highlighted are laryngeals, glides and weak radicals.

Roots	Verbals	Deverbals	Gloss
f-r-h	färra	fɨrh-at	fear
s-y-t'	šät'ä	šɨyyač'	sale
k'-w'-m	k'ʷomä	k'ɨwwame	opposition
l-?-k	lakä	lɨ?uk	delegation
r-?-y	ayy-	rɨ?yot ra?ɨy	
b-h-l	al-	bɨhil	

Table 2.4 Formation of stems by reduction process

Verb Derivations

Verbs can be derived from roots by intercalating vowels or by internal derivation, external derivation or combination of both [9]. Verbs derived using the first method are *simple verbs* and the later are called *derived verbs*. Internal derivation involves changing patterns of consonant-vowel patterns (E.g. derivation of -ggädäl- from g-d-l-). External derivation involves attaching affixes (prefix and/or suffix) on simple verbs (E.g. derivation of tä-ggädäl- from g-d-l-).

Verbs are derived from basic roots containing consonantal radicals. The structures of the stems that are derived from roots are divided into nine [24].

	Type of Verb	**Root**	**Derivation**
	Deragi	g-d-l-	-ggädäl-
Passive/ Reflexive	Tederagi	g-d-l-	tä-ggädäl-
Transitive/ Causative	Adragi	l-b-s-	a-läbäs-
Transitive/ Causative	Asderagi	g-d-l-	as-ggädäl-
	Daragi	g-d-l-	-ggadäl-
Reciprocal	Tedaragi	g-d-l-	tä-ggadäl-
Adjutative	Adaragi (1)	g-d-l-	a-ggadäl-
Adjutative	Adaragi (2)	s-b-r-	a-ssabär-
Reduplicative/ Repetitive	Deraragi	s-b-r-	-säbbabär-
Reciprocal	Tederaragi	g-d-l-	tä-ggädaddäl-
	Aderaragi (1)	g-d-l-	a-ggädaddäl-
	Aderaragi (2)	d-b-d-b-	a-tä-däbaddäb-
Jussive Passive		-n-b-b-	yɨ-t-näbäb-
Other		-s-b-r-	säbärr-

***Table 2.5** Types of derived verbs*

The verbs listed on Table 2.5 show repeated actions, actions which take place on one subject, by another one. All of them are derived from the stem -ggädäl-. Deragi verbs indicate the doer of the action. When the prefix tä- is attached in front of it, it shows the receiver of the action. When prefixes a- and as- are appended in front of it, it shows helping someone to do the action and making someone to do the action respectively. Daragi, reciprocal, Reduplicative/ Repetitive and adjutative verbs show that the action is done by more than two subjects. Jussive Passive shows a future action that is going to be done by someone other than the speaker.

Nouns

Nouns in Amharic are divided into two: primary and derived [23].

Primary nouns originally exist in the language and are not derived from any word [23]. These include names of people, mountains, countries e.g. *bet, mekina*

Pronouns are forms of primary nouns, which are inserted to show the subject of the action [23]. *ene, ante, anchi, egna, enante* are some of Amharic pronouns.

Derived nouns can be derived from verbs, adjectives, nouns, morphemes and combining nouns [23].

Nouns that are derived from verbs can have five types [23]. At least one noun can be derived from a single verb. For example, nouns like *mot, amuamuat, memot, memocha* and *mut* can be derived from the stem *mot-*. Nouns can also be derived from primary or derived adjectives. *chernet, lebo, bilt'et* are derived from the primary adjectives *cher, leba* and *biltt'* respectively. *sinfina, kifat, sekaramnet, ebdet, gilbiya* are derived from derived adjectives *senef, kifu, sekaram, ebd* and *galabi* respectively.

Nouns can also be derived other primary nouns by suffixing *-nät, -o, -ägna, -e* and *–egna* [23]. Nouns like *lijnet, ahyo, feresegna, gojamie* and *englizegna* are derived from nouns *lij, ahya, feres, gojam* and engliz respectively by adding the mentioned suffixes. There are nouns that are derived from derived nouns as well. For example, the noun *shumet* is derived from the derived noun *shum* by suffixing *-ät*.

Nouns like *gibrina, elbat* and *simet* are derived from morphemes [23]. They are formed by suffixing *-ena, -at* and *-et* on *gibir, elb* and *sim*.

Compound nouns are formed by combining two nouns, two verbs or verbals and adjectives [23]. *wetbet, hige-mengist, alem-ak'ef, wedo-zemach* and *sergo-geb* are examples of compound nouns.

Adjectives

Adjectives are parts of speech which can take the word *bät'am* /very greatly/ in front of nouns [23]. Adjectives can be primary or derived. Adjectives like *däg* are primary and *sänäf, tärarama* and *hodäsäfi* are derived from the stem verb *sänäf-*, the noun *tärara* and combination of derived nouns *hod* and *säfi*.

2.3 Spelling Errors

Spelling errors are misspelling made by users who type in word processing application. These errors can be classified as typographic, cognitive and phonetic errors [11]. They are also classified as non-word and real-word errors. The next section discusses the distinction between each types of error.

2.3.1 Types of Errors

Typing errors can be categorized into typographic, cognitive and phonetic errors based on the error patterns produced when typing words on a word processing application [11].

i. **Typographic errors** occur when the correct spelling of the word is known but the person who types unknowingly makes a mistake. Typographic errors can either be single errors or Multi-errors [6].

Single errors occur when the error is only in one character of a word and they could occur due to four operations:

Insertion – inserting an extra character in the word eg) Typing ትመምህርት for ትምህርት

Deletion - missing a single character from the word eg) Typing ምህርት for ትምህርት

Substitution – substituting a character with another one eg) Typing ትመህርት for ትምህርት

Transposition – swapping two adjacent letters in the word eg) Typing ትህምርት for ትምህርት

Multi-errors contain errors in more than one characters.

ii. **Cognitive Errors** occur when the correct spelling of the word is not known. For example, typing ‘beleive’ for ‘believe’. These types of errors do not apply to languages like Amharic because there is no such thing as spelling in Amharic. As a result, the person who types would always know how to spell the word as long as s/he knows the shape of the characters.

iii. **Phonetic Errors** occur when the user replaces a character with its phonetically equivalent character [14]. (e.g. Separate – Separete)

Spelling errors can also be categorized into two based on the meaning they give: Non-word and Real-word errors [19] [6]. **Non-word errors** are spelling errors that do not give any meaning, while **Real-word (semantic) errors** are morphologically valid word which give no sense in context. These errors could occur when the user types a different correctly spell word or when a non-word is mistakenly corrected to a wrong real-word by the spell checker. For

example, typing ፓስታ for ፖስታ. In this example, both words are valid, but the person who types mistakenly types ፓ instead of ፖ. Conventional spell checkers would not detect such errors.

Real-word errors are in turn classified into Wrong-word errors, Wrong-form-of-word errors and Word-division errors. Wrong-word errors occur when the word is grammatically and semantically incorrect, Wrong-form-of-word errors occur when the word is grammatically incorrect and Word-division errors occur as a result of omitting a space between two Words (run-on) or inserting a space in the middle of a word.

	Type of Error			Example
1	Non-word error	Simple Error		እህቴ ነገ ተመጣለች
		Multi-error		እህቴ ነገ ተመጣላች
2	Real-word Error	Wrong-word error		እህቴ ነገ በኋላ ይመጣል
		Wrong-form-of-word error		እህቴ ነገ ይመጣል
		Word-division error	Run-on	እህቴ ነገትመጣለች
			Split word	እህቴ ነገ ትመ ጣለች

Table 2.6 Non-word and real-word errors in Amharic

As shown on Table 2.6, simple errors are caused by making a mistake on a single character, whereas multi-errors are caused by making a mistake on two or more characters. Wrong-word errors are caused by using invalid combination of words in a sentence while wrong-form-of-word error is caused by invalid use of words that does not go with the sentence. Run-on errors are caused by omitting a space character between words whereas split word errors are caused by inserting a space character in the middle of a word.

2.3.2 Spelling Errors in Amharic

Most of the category of errors discussed in section 2.2.1 apply to Amharic except cognitive and phonetic errors. Amharic is different from other languages like English because vowels are found embedded in Amharic characters. For example, ሁ = ሀ + ኡ has the vowel ኡ embedded at the end. As a result, cognitive errors do not occur in Amharic texts. At the same time, one cannot replace a character with its phonetic equivalent because each sound is represented by its own character.

The existence of phonetic redundancies could create spelling errors. In Amharic, there are several phonetic redundancies [28], that is characters which have the same sound but represented with more than one character. For instance, the character ha can be represented by: ሀ, ሃ, ሐ, ሓ, ኀ, ኃ, and ኻ; the character sə can be represented by ሰ, ሠ; the character ‘a’ can be

represented by ኣ, ኣ, ዐ, ዓ; the character ṣä can be represented by ጸ, ፀ. There is a convention for the way some words are typed. For instance, the word ṣähay can be typed like ፀሐይ, ፀሀይ, ፀሃይ, ፀሓይ, ፀኀይ, ፀኃይ or ፀኻይ. All the mentioned words can be read as the intended word, however the convention insists to use the characters ፀ = ṣä and ሐ = ha in ṣähay. As a result, the other six forms of typing ṣähay can be counted as misspelled.

2.4 Spell Checking Approaches

Different spell checking approaches have been used for different languages. This section thoroughly describes the main techniques. Spell checking involves two steps. The first step involves detecting if typed words are correctly typed or not. The second step involves generating closer suggestions for misspelled words.

2.4.1 Spelling Error Detection Approaches

Spelling error detection is the first step in spell checking. It is the process of identifying if the given word is a non-word or not. Different error detection approaches are applied to language with different natures. Existing detection techniques can be generally classified as Dictionary Lookup and n-gram.

i. Dictionary Look-up

Dictionary look-up technique works by searching the given word in a dictionary of lexicon, a corpus or a combination of lexicons and corpora [7]. The dictionary is assumed to contain all words and their inflected forms plus a word of specific topics like Computer Science or Economics [7] [11]. A word which does not have a match in a dictionary (using an exact string matching technique) will be taken as misspelled. The disadvantage of this method is in the fact that it will be impossible to store all words and even if all are stored, it should be updated from time to time. Moreover, searching for a word in a large number of knowledge base would not be efficient [11]. To solve the problem of inefficiency, a technique called hashing is used. *Hashing* involves searching for the hashing address or key of a word in a hashing table. If the word in the hashing address and the input word are different, the word will be marked as incorrectly spelled.

Binary search tree predominantly *median split tree* is another method used in dictionary look-up approach. Median split tree involves looking for a word in a large collection of strings. It

avoids the time it takes to search for the word in uncommon words i.e. it only accesses common words.

Finite State Automata (FSA) is another technique used in dictionary look-up and it involves representing a language as a set of strings with a sequence of notations of some script.

ii. n-gram Analysis

n-gram analysis works by comparing sequence of n-letter subsequences. N-gram frequencies are stored in an n-dimensional matrix and the n-gram sequences of the input word are compared to the stored subsequences. If one n-gram is missing, the word will be considered misspelled. n could have values 1, 2 or 3 and the resulting n-grams are unigram, bigram and trigram respectively.

2.4.2 Spelling Error Correction Approaches

Following error detection, spelling error correction is the second step in spell checking. The error corrector generates closer words to the misspelled word, ranks the candidate words and replaces the misspelled word with the correct one [7].

There are three types of spelling correctors [13].

1. Spelling correctors that use the dictionary - look for the given word in a given list of correctly spelled words.
2. Spelling correctors that use the dictionary indirectly - create a table of all trigrams by splitting words in the dictionary into three letter sequences. Such spelling correctors work by splitting the given word into trigrams and comparing them with the tabulated trigrams list.
3. Spelling correctors that do not use the dictionary - work by dividing the text into trigrams storing them in a table. It then counts the frequency of occurrence of each trigram and calculates index of peculiarity for each word on the basis of the trigrams it contains. Words with high peculiarity level will be identified as misspelled.

Spelling error correctors can work in two forms. *Isolated term correction* can work by identifying spelling errors for each word in a text whereas *context-sensitive correction* checks if a word (spelled correctly or not) gives a meaning context-wise.

Spelling error correctors can be also classified as *Fully Automatic Spelling Error Detection System* and *Interactive Spelling* Error Detection System [6].

1. Fully Automatic Spelling Error Detection System automatically replaces the misspelled word with the most likely candidate.
2. Interactive Spelling Error Detection System generates candidate words, ranks them and suggests the closest candidates.

Some of the existing error correction techniques include minimum edit distance, similarity key, rule-based, n-gram-based, probabilistic, neural networks and noisy channel approaches [7] [11].

i) Minimum Edit Distance

Minimum edit distance refers to the number of operations (insertion, deletion or substitution) that needs to be performed to transform one string to another. From the way it was presented first, it was later improved and transposition (swapping) was introduced to the list of operations [7]. This technique is more useful for keyboard input errors that phonetic errors [11].

ii) Similarity Key Technique

In this technique, every word is assigned a key and similar words will have similar or the same keys [11]. The key to the misspelled word is computed and words with similar key values to it will be generated as candidate suggestions for the misspelled word [7] [11].

In this technique, similarity between words can depend on three factors [7]:

a. *Positional Similarity* – the degree of matching characters of two words being in the same position.

b. *Material Similarity* - the degree of two words having the same set of characters being in different order.

c. *Ordinal Similarity* – the degree of matching characters by two words being in the same order.

iii) Rule-based Technique

It works by defining set of rules that denote common spelling error patterns and steps (morphological information) to generate correct words from misspelled words [11]. If a misspelled word falls in the set of common spelling error patterns, candidate words will be generated by applying all rules that can be applied on the identified error pattern.

iv) n-gram-based Technique

n-gram-based technique can be used with or without a dictionary [11]. When used with a dictionary, the distance between words will be defined by n-grams and will be compared with words in the dictionary. When used without a dictionary, n-grams are used to find the position of the error in the misspelled word. It can be used in three manners: error detection, candidate suggestion and rank similarity.

v) Probabilistic Technique

Probabilistic technique is based on statistical characteristics of the language [11]. There are two kinds [7]:

a. *Transition or Markov probabilities* give the probability of a given character being followed by another one.

b. *Confusion or error probabilities* give the probability of a given character replacing another one in a misspelled word.

vi) Neural Networks

Neural networks use algorithms like back-propagation to train patterns of actual spelling errors to the machine [11]. A back-propagation network consists of an input layer for possible n-grams of a word, a hidden layer that indirectly connects input and output nodes and an output layer for all words in a dictionary. The trained network gives one of the nodes on the output layer as a correction for the misspelled word.

vii) Noisy Channel

Noisy Channel Model works by treating the misspelled word as if a correctly spelled word had been misrepresented when it passes through noisy communication channel [15].

2.4.3 Spell Checking Approaches in Semitic languages

This paper presents some of the spell checking approaches used for Semitic languages like Amharic and Arabic. In section 2.5, application of noisy channel and optimized metaphone algorithm, dictionary-based approach with hashing for Amharic, morphology-based approach for Afaan Oromo and Amharic and Levenshtein algorithm for Arabic are reviewed and discussed. Most of the works use edit distance algorithm to generate suggestions. In almost all studies, the researchers try to tackle the problem of handling the morphological complexity of Amharic.

Dictionary Look-up Approach for complex languages like Amharic is ruled-out because it involves listing out and storing every root and stem words and all their derivations. This would be a very cumbersome and time-consuming task. Even if one stores all root and derivations, then for the spell checker to go through this knowledge base and searching for a single word would take a lot of time and will in no way be efficient. However, this approach can be used as part of a morphology-based spell checker. The dictionary can be used to store only stem words of the language, which is significantly small compared to number of every words in the language. Once a word is reduced to its stem form by applying morphological rules, the resulting stem can be searched in this dictionary and the word can be labeled as valid or invalid.

Metaphone algorithm is also ruled out because it is designed for morphologically simple languages like English.

Rule-based approach for spelling correction involves defining all sets of rules that capture common spelling and typographic errors [11]. In this approach the defined rules represent the common error types that are mostly made, not morphological rules as in the case with morphology-based approach. This approach is ruled-out because identifying common error patterns needs a large corpus and defining them is costly and time-consuming [7]. In addition, as the morphology of the language gets complex, the common error patterns become complex as well.

Machine learning methodologies try to identify a pattern and create their own models from the given training set. Given the complexity of Amharic morphology, extracting a model from a given training set would not give a correct result. It is unclear how machine-learning systems work. As a result, have a problem of predicting an unexpected result [14]. The algorithm might focus on a part of the data that is not significant to predict the result. Prediction results that work in one test corpora might not be the same for the other. This implies relying entirely on such predictions is not strategic. In addition, machine-learning approaches need a very large corpus, which is not currently available for morphologically rich languages like Amharic. Moreover, Amharic corpus collected from the web contains documents written in a mix of Unicode and non-Unicode characters. Because of this fact, corpus-based machine-learning approach would be challenging. Amharic also suffers from out-of-vocabulary words problem [9] which is one of the reasons to rule-out machine-learning approaches for spelling error detection.

This indicates that the complex morphological nature of the language makes a morphology-based approach more fit for spelling error detection and correction. Though rule-based systems tend to have high performance in specific scenario and reduced performance when generalized, the designed algorithm will play a great role in making the system more cost-effective. The algorithm will work in a well-planned and organized manner so that the system will only access suitable rules. Details are discussed in chapter 4.

2.5 Spell Checking and Correction Algorithms

There are different spelling correction algorithms. String matching algorithms work by calculating the edit distance or the Levenshtein distance between two strings. Some of the open-source algorithms developed using this algorithm in combination with phonetic matching algorithms are Ispell, Aspell, Jazzy, Jortho and Suggester. They are designed for widely spoken European and Asian languages [13].

Phonetic matching algorithms work by mapping terms with the same sound to the same value. Such algorithms are called Soundex. They are later improved to metaphone algorithm to offer better performance. Metaphone algorithms work only with English language [13]. Some of the algorithms are discussed below.

SPELL

SPELL is a Unix spelling checker algorithm that was introduced by S.C. Johnson in 1979 and later improved by McIlroy. SPELL needs a small memory as it is designed for a machine with 64 KB address space memory [7].

SPELL takes an input file and reads and splits words in the file. Then duplicates are identified and removed. The split words are compared with a dictionary and those which are not found are reported as misspelled. The limitation of this algorithm is its small vocabulary coverage and slowness [7].

CORRECT

It is a spelling correction algorithm that uses edit distance and probabilistic approaches [7]. It takes words rejected by SPELL algorithm, produce candidate words and ranks them by probabilities. CORRECT assumes that spelling error occurs due to a single error caused by one of the operations: insertion, deletion, substitution and reversal. The misspelling can also be corrected by applying one of the four operations. For instance, an error caused by the insertion operation can be corrected by deletion and vice versa.

ASPELL

ASPELL uses metaphone algorithm created by Lawrence Philip in combination with ISPELL's near miss strategy which states applying insertion or substitution on a misspelled word gives a correct word [7]. It makes use of hashing with dictionary look-up. It is an open-source software which can work as a stand-alone application or it can be integrated as a library.

AURA

AURA (Advanced Uncertain Reasoning Architecture) is a spell checker and corrector that uses phonetic matching and Correlation Matrix Memories (CMMs) to correct single letter and phonetic spelling errors [7]. Due to CMM's fast retrieval capability, AURA is developed as a preprocessor for an information retrieval system.

2.6 Related Works

This section summarizes previous works done in spell checking. The first section presents seven spell checking research works for Amharic. In addition, related works for Afaan Oromo and Arabic are summarized and discussed. As morphological analysis and generation is the main backbone for spell checking researches, three related works on this topic are discussed as well.

2.6.1 Amharic Spell Checkers

a. Portable Spelling Corrector for a Less-Resourced Language: Amharic

In this research, a corpus-driven approach with noisy channel is used to correct spelling errors of Amharic and English [19]. Since the focus of this research is Amharic language, research work for Amharic will be reviewed. Damereau-Levenshten edit distance algorithm is used to measure nearness among generated closer words to the misspelled word. The research specifically focuses on correcting spelling for non-word errors. This approach is selected because the authors claim that it takes much effort to make a list of all language-dependent rules for spelling correction, given Amharic is morphologically-rich.

They built their own contemporary Amharic corpus (CACO) that is assembled from publicly archived Amharic newspapers, legal documents, magazines, fictions, historic novels and books, short novels, politics books, Amharic bible and children books. They have used HaBiT, a sizable text corpora that is created from automatically crawled pages, for comparison. Paragraphs are extracted and characters are transliterated to Latin-based characters. Following that, numbers are replaced by a placeholder, hyphenated words are split by removing the

hyphen and replacing it with space character and unique sentences are identified and extracted. From the identified sentences, words are tokenized based on separator characters two dots (:) and white space. Then, sentences which contain words that appear only once in the whole corpus are removed. This step is added by assuming that the word that is mentioned only once is probably incorrectly spelled.

The language models are trained using the KenLM language-modeling toolkit. The error model was adapted from the one created by Norvig (2009) on forty thousand spelling errors. Terms with missed white spaces are split on the basis of the most probable split candidates based on CACO language model and the corresponding term list.

The performance of their approach is evaluated based on Amharic test data and the results are compared with Aspell and Hunspell baseline systems.

The evaluation for Amharic spelling error detection resulted in 89.4% precision, 80.6% recall and 84.8% F1-score. Precision flags all misspellings, recall indicates the language coverage and F1 measure indicates the capability of detecting the spelling error. Evaluation metrics were also computed and compared for other baseline systems: HaBiT, Aspell and Hunspell. The results show that there was an improvement in spelling error detection (F1) on the proposed approach from HaBiT. Nevertheless, there was no improvement in the value of recall when the term list in HaBiT corpus is used.

Suggestion list were also evaluated and 77% of correct spellings appear in the top five suggestions list for the proposed system using CACO compared to 34% for Hunspell, 62% for Aspell and 75% for HaBiT. The proposed system scored 9%, 18% and 35% top first suggestions higher than HaBiT, Aspell and Hunspell respectively.

The authors concluded that the proposed system can be used for other written languages as long as they are typed using a QWERTY keyboard with direct mapping between keystrokes and characters. In addition, they will evaluate their approach for real-word spelling errors in future works.

b. Development of Amharic Spelling Corrector for Tolerant-Retrieval

In this research, a metaphone algorithm specially optimized for Amharic is used for Amharic spelling correction and Levensthein edit distance algorithm is used for suggestion [13]. The purpose of the research is to integrate the system into tolerant-retrieval search systems. It is the first spelling correction work for Amharic.

In this research, metaphone algorithm is optimized for Amharic by using the basic characteristics of the Amharic writing system. The optimization involves mapping homophones to their representative characters followed by removing all vowels from a word except an initial vowel character. A dictionary is developed from the Amharic bible and Amsalu Aklilu's Amharic-English dictionary. Spelling errors found in these documents are manually corrected.

The proposed algorithm works by creating a hash table and an Amharic metaphone code. The algorithm looks for the metaphone code in the hash table. Edit distance calculation is used to select relevant suggestions. It also creates more suggestions by deleting, inserting, swapping and replacing letters and splitting words.

The test result shows that 81.7% of cases are correct suggestions among its top-five responses. The author suggests to make the dictionary bigger, to use stemming methods and to assign different weights for the different operations of the edit distance algorithm.

c. Application of Morphological Information for Amharic Spelling Error Detection

In this research, three approaches for detection of Amharic spelling errors are proposed and evaluated [12]. The general objective of the study is "to find ways of enhancing the performance of Amharic spelling error detection by applying a stemming algorithm and derivational rules". The study uses a design science research methodology which involves problem identification, defining objective and design & development.

The first approach involves using dictionary look-up to detect errors. The flagged words in the first approach are passed to the second approach for further processing. The second approach involves application of stemming to check validity of words and to identify prefixes and affixes. The third approach accepts flagged words from the second approach and checks whether the flagged words are correct flags or not.

The system requires dictionary file which contains a total of 175,070 Amharic words collected from different sources. The error detector is tested with 33,535 words of different domains collected from two news channels and pre-edited documents. The system also uses an affix list with 165 affixes, an XML file which contains 239 rules and a file containing list of Amharic punctuation marks and special characters.

A prototype developed by Microsoft Visual Studio is developed. The prototype has five components. Tokenizer component converts an input stream of characters, splits it into words, removes special characters and returns tokens. Transliterator component changes words written

in Ethiopic script to their Latin equivalent. Error detection component looks for a given word inside the dictionary and sends it to the stemmer component if it not found. If it is found, the word is flagged as valid.

The stemmer component does a series of iterative procedures of affix removal and sends the word back to the error detector. If the word is still not found in the dictionary, it will flag the word as invalid. Pattern matcher component receives the words whose affixes are removed and flagged as invalid and looks for a matching rule inside the XML file that contains rule patterns. If no matches are found and if the affixes are valid, the word is flagged as valid, otherwise, it is flagged as invalid.

The prototype is tested with two test cases. The first test case contains 33,535 words collected from different sources and the second test case contains 1750 verb forms generated manually.

The experimentation result shows that the second approach works better than the first and the third approach works better than the second. The third approach tested with the first test cases shows values of 0.9, 0.95, 0.97 Predictive Accuracy for the first, second and third approaches respectively.

The testing with the second test case shows 0.42, 0.6 and 0.95 Predictive Accuracy for the first, second and third approaches respectively. The values show the progress shift from one approach to the next. The researcher concluded that predictive accuracy of Amharic spelling error detection can be improved by applying stemming, inflection/ derivation rules and following morphological information (affix list and patterns). The researcher recommends including inflectional behavior of other parts of speech and identifying the stem group in which a pair of prefix and suffix patterns occur at the same time to avoid flagging meaningless words as valid and flagging meaningful words with multiple concatenated prefixes/ suffixes as invalid.

d. Multilingual Spelling Checker for selected Ethiopian Languages

This research employs dictionary-based approach with hashing for detection and correction of misspelled words in five Ethiopian languages namely Amharic, Afan Oromo, Tigrigna, Hadiyyisa and Awngi [18].

The proposed system flags a word as correctly spelled if the candidate word matches the word at the hash address. If there is no match, then the word is flagged as invalid. The experiment uses a text corpus collected from different sources of equivalent amount of words for each of the languages.

The evaluation is measured using evaluation metrics precision, recall and F-measure. The results show precision, recall and F-measure values above 81%. It was concluded that the suggested approach is able to detect diverse classes of spelling errors in all mentioned languages.

e. Automatic Amharic Spelling Error Detection and Correction using Hybrid Approach

The study proposes using metaphone algorithm for error detection and edit distance algorithm for error detection of wrongly spelled Amharic words [16]. A prototype is developed using VB.NET and python programming language. The system follows three steps namely pre-processing, spell checking and correction. It accepts a candidate word, extracts the root word and required features and checks the validity of the words against the dictionary of words. If it is not found, that means the word is invalid and the system will generate closer words from the dictionary using edit distance algorithm.

The evaluation results show that the system is 98% effectiveness in its error detection and correction capability. From the results, it was concluded that the proposed system is effective and accurate. However, the speed of processing was not considered and the author recommends improving the processing time and integrating it with other Microsoft office systems in future works.

f. Automatic Spelling Checker for Amharic Language

This study attempts to design and develop an automatic Amharic spell checker integrated with Open office word processor [20]. The study focuses on non-word errors. It considers internal inflection of words, repeated words and compound words. It uses Hunspell tools for morphological analysis of words. Hunspell is a morphological analyzer library designed for languages with complex morphology. The spell checker has five components namely, input component, normalization, error detection, morphological analyzer, error correction and suggestion components.

The input component tokenizes and removes punctuations from words written or copied on Open office. Normalization component brings characters with syllographic redundancy into a common form. Error detection searches the candidate word inside dictionary of root words. If it is found, it is flagged as valid, otherwise, it will be passed to the next component. Morphological analyzer component receives words not found in dictionary from the previous component and splits it into stem and affixes. The last component, error correction and

suggestion receives stem and affixes from the morphological analyzer and uses Levenshtein edit distance to search closer words inside lexicon of words and ranks them.

To evaluate the system, five experiments are conducted. The experiments are conducted with documents taken from different sources. Table 2.7 shows the summary of experiments, the number of test cases and the evaluation results.

Experiment	Source	Total number of words	Invalid words	Precision	Recall
1	Amhara Science Technology and Information Communication Commission	199	5	100	99.4
2	Amhara Science Technology and Information Communication Commission	840	16	100	97.6
3	Afar Region ICT 2009 annual report	181	7	100	96.1
4	Harari region ICT 2009 annual report	94	9	100	89.8
5	Experiment 4 evaluated by language expert	94	7	97.75	92.55

***Table 2.7 Evaluation results for the experiment conducted for* [20]**

The evaluation result gives an average performance of 97.4%. To figure out the reason of incompleteness, a sixth experiment was made and the performance of the system is improved. Incompleteness was caused because affix rules are not exhaustively defined and dictionary of words are incomplete. The study recommends including real-word errors, considering additional sources of errors, comparing other correction techniques, integrating the work with other NLP applications.

g. Rule Based Amharic Spellchecker and Amharic-English Translation Aid

This was a self-initiated project to implement a spellchecker for Amharic which is highly morphologically complex language. Another objective of the project was to lay down a background for the development of a system for Amharic to English translation (direct or indirect translation). It was claimed that the existence of rule-based spellchecker could help a lot in implementing Amharic to English translation in such a way that the definition of the derivation rules can be linked to meaning of phrases and sentences.

The importance of having an Amharic spell checker was indicated by mentioning the challenge of typing Amharic which is written with more than 300 characters. It was also stated that spelling errors in Amharic are particularly typing errors as the notion of spelling in Amharic is non-relevant if there was no typing error. This is supported by the fact that each sound is represented by a single designated character in Amharic.

The project implementers stated that any spellchecker requires a knowledge base from which candidate words can be checked. During their research, they collected 22,000 common root words from two known Amharic dictionaries [17] [29]. They claimed that it would have been reasonable to create a dictionary consisting of these words to implement the spell checker if Amharic was not morphologically complex language. They have found out that thousands of words could be derived from a single root verb. Thus, they asserted that building a knowledge base consisting of root words and derivation rules is more efficient and can lead to building an accurate Amharic spellchecker than collecting millions of Amharic words, which could be root words, phrases or even sentences.

The project was started by identifying derivation rules and formulating a relationship between root words and derivation rules. The implementers explained that they were forced to do this as they could not find a research or a book that defines all derivation rules for Amharic. However, they have utilized the categories of root verbs according to [24] to define and group rules. The resulting knowledge base mainly consists of set of root words and derivation rules. Derivation Rules are defined with various parameters such as prefixes/suffixes, whether they use of both prefixes and suffixes, to what categories of root words they are applied to and other useful identifiers.

The system has three major modules namely Dictionary Lookup, Rule Evaluator and Suggestion Generator. The dictionary lookup implements an efficient searching to look up for words. The Rule Evaluator is applied to any word which was not found in the root words list. It is responsible for analyzing a word against the available rules to pick likely rules and assign testing priority to them. A candidate word is evaluated to identify if it probably fulfills the definition of a rule, and rules with strong probability are generally picked. Then, it tests the selected rules in the order of their testing priority to identify if they can produce a valid root word. It makes use of the dictionary lookup module. Testing a rule against a derived word involves attempting to reverse the word into a root word.

The suggestion generator finally acts up on identified invalid words. During the process of checking the validity of a candidate word, two major processes are carried out, which are looking up the word in the root words list and attempting to reverse the derived word to a root word by applying the selected rules. The word was obviously detected as invalid as none of these attempts were successful. The suggestion generator generates a list of suggestions from two sets. The first set will be all closer valid words from the root word list. The second set of suggestions is found by picking valid root words which are closer to the invalid root words

produced from the reversing process and applying the respective rules. The union of these two sets will be presented to the user after they are ranked.

The implementers reported that identifying a word as valid or invalid works 100% as long as the root word and the derivation rule exist in the knowledge base. However, they witnessed that exhaustively defining all rules was beyond their capacity and it requires the involvement of a linguistic expert.

The major limitation of this work was identified by the implementers. To improve the completeness and accuracy of the spellchecker, the knowledge base should be complete in the sense that it needs to include definition of all derivative rules in relation to the category of root words. Even though they believe that this is possible to do with the help of language experts, the size of the knowledge base affects the efficiency of the process and hence the response time. For such applications, response time is quite important as the system was integrated with Word processing software and used in real time. They suggested rules should be defined in a better way to reduce the processing time. And reversing the derived words back to root words should be made automatic. Their system needed to run code to do the reversing process, which took the majority of the processing time. They concluded that rule definition should be systematic and more distinct and should show the speech parts separately.

2.6.2 Afaan Oromo Spell Checker

a. Design and Implementation of Morphology-based Spell Checker

This research work is the first spell checking research for Afaan Oromo [5]. The research focuses on detection and correction of non-word errors. Afaan Oromo has a very rich inflectional morphology. Nouns are inflected for number and verbs are inflected to show gender, number, tense, voice, aspect and mood.

The proposed system uses a morphology-based spell checker that combines dictionary look-up with morphology rules. The suggested architecture contains Tokenizer, knowledge base, Error detection, Morphological analyzer, Error correction, Morphological generator, Suggestion ranker and Word Assembler components. The tokenizer component splits a block of text into individual words. The knowledge base component stores root words (lexicons), affixes and rules of the language. The error detection component receives a word from the word tokenizer and looks for the word inside the root word knowledge base. If it is found, the word will be taken as correctly spelled. If not, the word will pass to morphological analyzer component. The morphological analyzer breaks the word into root affixes and sends it back to the error detection

component. The error detection component looks for the word and affix in the knowledge base and if it exists and if the root has the affix flag, that means the word is correctly spelled. If not, it will pass to error correction component. The error correction component classifies the errors into classes and makes a closer correction based on their class.

Morphological analyzer component receives corrected morphemes from the error correction component and retrieves affixes based on the class of the root word. Then, possible candidate words will be generated from the retrieved affixes. These will be sorted based on keyboard layout character distance and Levenshtein Edit Distance algorithms. The Levenshtein distance between the misspelled word and the candidate words will be calculated by using insertion, deletion, substitution and transposition operations. If the resulting distance is equal for some of the candidates, character edit distance is calculated to rank them. Finally, the word assembler component maps the correct word with the misspelled word. The evaluation of the proposed system resulted in 88.62% lexical recall, 100% error recall and 28.62% precision. Lexical recall specifies the percentage of valid words correctly accepted, error recall indicates the percentage of words correctly flagged as invalid and precision shows the percentage of correctly detected invalid words.

The author concluded by suggesting that the research will be a worthy input for the development of Afaan Oromo spell checker.

2.6.3 Arabic Spell Checkers

a. For an Independent Spell-Checking System from the Arabic Language Vocabulary

The research implements an automatic spell checker that introduces morphological analysis concept in the Levenshtein algorithm and using small size dictionary that contains Arabic language stems [30]. They have compared and evaluated their method with Levenshtein approach. They have measured rate of correction by insertion, deletion and permutation operators. The average correction rate by the proposed system is 33.7% higher than Levenshtein distance. The average correction time (0.10 ms) is faster than that of Levenshtein distance (0.19 ms).

The author concluded that the proposed approach is much better than that of Levenshtein distance and it can be a way to go for the development of automatic spell checker.

2.6.4 Morphological Analysis

a. HornMorpho: a system for morphological processing of Amharic, Oromo and Tigrigna

The study presents a morphology analyzer and generator tool for Amharic, Oromo and Tigrigna [31]. Morphological analysis is breaking down a word into morphemes and classifying grammatical and lexical morphemes in their respective categories. The tool benefits developers who work in language related application, non-native speakers of Amharic who are learning the language and everyone.

HornMorpho uses an architecture that is composed of a layer of Finite State Transducers (FSTs) that represent alternative rules and morphotactics (stem, prefixes, suffixes). The program has morphological analysis and generation functions. The morphological analyzer has functions anal_word and anal_file for a single word and all words in a file respectively. The functions take input word and outputs a root, stem and morphological analysis of the word. In addition, seg_word and seg_file functions segment verbs into morphemes separated by hyphens. phon_word and phon_file functions take words given in orthographic format and change them to phonetic formats. The morphological generator component of the program has a function called gen that takes a stem or verb plus grammatical features and returns morphological derivations.

The morphological analyzer is evaluated with randomly selected word list from web crawled corpora. The test result gives 96% and 99% accuracy for Tigrigna and Amharic verbs respectively and 95.5% accuracy for Amharic nouns and adjectives.

The morphological generator is tested and gave a result of 100% accuracy for Amharic and 93% for Tigrigna. The authors are in the process of integrating Oromo language to HornMorpho and no evaluation results were presented. The weakness of the program is in the limited number of available roots and stems and in handling ambiguities.

b. Development of Amharic Morphological Analyzer using Memory-Based Learning

The research attempts to develop a morphological analyzer for Amharic using one of the supervised machine learning approaches known as memory-based learning [7].

The proposed system is trained with manually annotated sample verbs and nouns. The derivation pattern stems and morphemes follow is studied and used for annotation. The proposed architecture has two phases: training phase and morphological analysis. The first step of the training phase of the architecture involves morpheme annotation. To detect the

morpheme from a set of words, inflected words are identified and they are segmented to prefix, stem and suffix and each segment will be marked by their representative notations. Suffixes have seven features: plurality, derivation, relativazation, definiteness, negation, causative and conjection. These features are represented by their own representative letters. The annotated words are stored in a morphological database. Following morpheme annotation, instances of the annotated words are automatically extracted from the morphological database using windowing method. Then, a memory-based learning model is developed using a learning tool TiMBL. Classifier algorithms called IGtree and IB1 are used to construct databases in memory.

Once the learning model is developed, the morphological analysis phase continues. In this phase, a new word is analyzed based on previous training data. The word is segmented and represented as instances and they are compared with the training set. The word will be classified as the closest instance. If the inflected word is unknown, its morpheme is identified and the class that shares the most common features is inferred and predicted as the class of the new instance.

The next step is reconstructing the given word into meaningful units. The system searches for similar stem patterns from the stored training set. If it is not found, distance to the most nearest neighbor is calculated. The last step is extracting the root from the stem. This involves removing the vowel in stems for words with more than three characters. Exception is for stems which start with a vowel and for mono and bi radical verbs.

The system is tested with 10-fold cross-validation technique for IB1 and IGtree algorithms. The data is divided into ten. One is used as a training set and the other nine as test sets at a time. Each split data gets to be used as a training set. The researchers also used Leave-One-Out (LOO) method to evaluate IB1 algorithm. LOO method takes all available data except one as a training data.

The evaluation gives an accuracy of 96.40% and 93.59% for IB1 algorithm using LOO and 10-fold techniques respectively and 82.26% accuracy for IGtree algorithm. The LOO method becomes time consuming when used for IGtree algorithm, as a result, the experimentation results are not presented.

The authors concluded that the proposed system has a better accuracy from other similar attempts. A better performance will be achieved by increasing training data and embedding grammatical features in segmented morphemes.

c. Learning Morphological Rules for Amharic Verbs using Inductive Logic Programming

This research uses supervised machine learning approach for morphological analysis of Amharic verbs [22]. An Inductive Logic Programming system called CLOG is implemented. It learns from positive examples and it runs faster than other systems.

Three training experiments are conducted for stem extraction, root patterns and internal stem alternation rules. In experiment one, the system learns stem extraction by identifying prefix and suffix of a word and extracting the stem from the given input word. In experiment two, the system learns roots by separating vowels and consonants of a stem and relates the stem with the root formed from radicals. The third experiment involves learning stem internal alternations from a training data containing stems with such patterns. In addition to the three experiments, two predicates are defined to extract the valid pattern for the stem and to link affixes and root patterns.

The system is trained with 216 manually prepared Amharic verbs, 108 affix extracting rules, 18 root template extraction rules and 3 internal stem alternation rules. The system analyzes 86.99% of the given words correctly. The researchers recommend using more training data for better performance.

For the future, they are in the process of experimenting with genetic programming to learn predicates. The researchers concluded that ILP can be used to analyze morphology of complex languages like Amharic.

2.7 Summary of related works

Title	Author, Year	Purpose	Approach/ Methodology	Key Findings	Recommendation & Future Work
Portable Spelling Corrector for a Less-Resourced Language: Amharic [19]	Andargachew M., Andreas N., Binyam E., 2018	an automatic spelling corrector for non-word errors for Amharic & English	a data-driven (corpus-driven) approach with the noisy channel for spelling correction, Damerau-Levenshtein edit distance to measure nearness	The proposed systems has better results that existing algorithms HaBiT, Aspell and Hunspell	The proposed system can be used for other written languages as long as they are typed using a QWERTY keyboard with direct mapping between keystrokes and characters. They will evaluate their approach for real-word spelling errors in future works.
Development of Amharic Spelling Corrector for Tolerant-Retrieval [13]	Andargachew M., 2012	Spelling Corrector for tolerant-retrieval Amharic search systems	Amharic metaphone algorithm /metaphone algorithm optimized for Amharic/ Edit distance algorithms for Suggestion /Levenstein Distance Algorithm/	81.7% of cases are correct suggestions among its top-five responses	Future recommendation - to make the dictionary bigger, to use stemming methods and to assign different weights for the different operations of the edit distance algorithm.

Application of Morphological Information for Amharic Spelling Error Detection [12]	Mulualem T., 2019	Find ways to enhance the perfection of Amharic spelling error detection	stemming algorithm and derivation rules	Capacity of Amharic spelling error detection can be improved by applying stemming, inflection/ derivation rules and following morphological rules	Including inflectional behavior of other POS, Identifying the stem group in which a pair of prefix and suffix patterns occur at the same time
Multilingual Spelling Checker for selected Ethiopian Languages [18]	Wubetu B., 2020	Detection and correction of misspelled words in five Ethiopian languages	Dictionary-based approach with hashing	The system is able to identify diverse classes of spelling errors.	-
Automatic Amharic Spelling Error Detection and Correction using Hybrid Approach [16]	Getnet A., 2018	Amharic spelling error detection and correction	Metaphone algorithm for detection and edit distance for correction of misspelled words	The proposed system is effective in its error detection and correction functionality.	implementing the system in all Microsoft office products improving processing time
Automatic Spelling Checker for Amharic Language [20]	Melaku T., 2020	Automatic spell checker integrated with Open office word processor	Hunspell for morphological analysis, Edit distance for error correction	Exhaustively defining affix rules and completing dictionary words would give a better result	Including real word errors, considering additional sources of errors, comparing other correction techniques, integrating with other NLP applications
Design and Implementation of Morphology-based Spell Checker [5]	Gaddisa O., Dr. Dida M., 2014	Spelling error detection and spelling error correction for Afaan Oromo	Morphology-based approach	Percentage of valid words correctly accepted = 88.62% percentage of words	The research will be a worthy input for the development of Afaan Oromo spell checker.

				correctly flagged as invalid = 100% percentage of correctly detected invalid words = 28.62%	
For an Independent Spell-Checking System from the Arabic Language Vocabulary [30]	Bakkali H., Gueddah H., 2014	automatic spell checker for Arabic	morphological analysis with Levenshtein algorithm	The average correction rate by the proposed system is 33.7% higher than Levenshtein distance. The average correction time (0.10 ms) is faster than that of Levenshtein distance (0.19 ms).	the proposed approach is much better than that of Levenshtein distance and it can be a way to go for the development of automatic spell checker
HornMorpho: a system for morphological processing of Amharic, Oromo and Tigrigna [31]	Michael G., 2011	Morphological Analyzer for Amharic, Tigrigna and Oromo languages	Finite State Transducers	Promising result for Amharic and Tigrigna (100% accuracy for Amharic and 93% for Tigrigna), work in progress for Afaan Oromo.	more user-friendly interfaces for the general public
Development of Amharic Morphological Analyzer using	Mesfin A. and Yaregal A., 2014	Morphological Analyzer for Amharic	Memory-Based Learning	Accuracy of 96.40% and 93.59% for IB1 algorithm using LOO and 10-fold techniques respectively and	The proposed system has a better accuracy from other similar attempts. A better performance will be achieved by increasing

Memory-Based Learning [7]				82.26% accuracy for IGtree algorithm	training data and embedding grammatical features in segmented morphemes.
Learning Morphological Rules for Amharic Verbs using Inductive Logic Programming [22]	Wondwossen M. and Michael G., 2012	Morphological Analyzer for Amharic	Inductive Logic Programming	ILP can be used to analyze morphology of complex languages like Amharic	Experimenting with genetic programming to learn predicates

2.8 Executive Summary

Different spell checking and morphological analysis techniques are applied for different languages. For morphologically complex languages like Amharic, it was shown that different approaches including morphology-based approach have been researched. In some of these studies, existing spelling error detection and correction algorithms (like metaphone and edit distance) designed for languages like English are employed. In these works, the complex morphological nature of Amharic and the huge difference in the nature of the morphology of Amharic and other languages is not considered.

In attempts that use dictionary-based approach, it requires storing millions of words which makes the system inefficient. In addition, the small amount of corpus found online is not considered. Some attempts use existing morphological analyzers for computational morphology. However, morphological analyzers tend to give unintended outputs for invalid words. It is also noticed that basing morphology is the main process in all spell checking and morphological analysis researches reviewed. For instance, in [7] though the researcher mainly uses memory-based learning approach, morphology-based annotation is used in preparing the training set. The same holds true for most of the research works.

This study fills the gap mentioned above by applying morphology-based approach integrated with a specially designed algorithm to detect and correct errors.

CHAPTER THREE

METHODOLOGY

3.1. Overview

Choosing the right methodology is a crucial step in conducting a research. As a result, a research paradigm that can produce the intended solution to the problem is employed. In order to accomplish the research aims and objectives, provide valid and reliable results, a research methodology that employs methods and techniques which are the best fit for the research is chosen. The research encompasses designing a new artifact to solve observed problems, evaluating the artifact and presenting the results. Therefore, design science research methodology is a perfect fit for this research. This chapter discusses the research methodology employed and activities performed to analyze the problem and get to the solution. In Section 3.2, the selected methodology, the designed process model and its components (activities, procedures and techniques) are discussed.

3.2. Research Methodology

In this research, design science methodology is employed. Accordingly, an artifact, a prototype that employs an algorithm to detect and correct Amharic spelling errors, is rigorously designed and evaluated.

The research is focused on Design and Development centered approach depicted on Fig 3.1. The figure shows the research process that involves six activities [32]. The DSRM process model on Fig 3.1 is adopted and modified for this research [32].

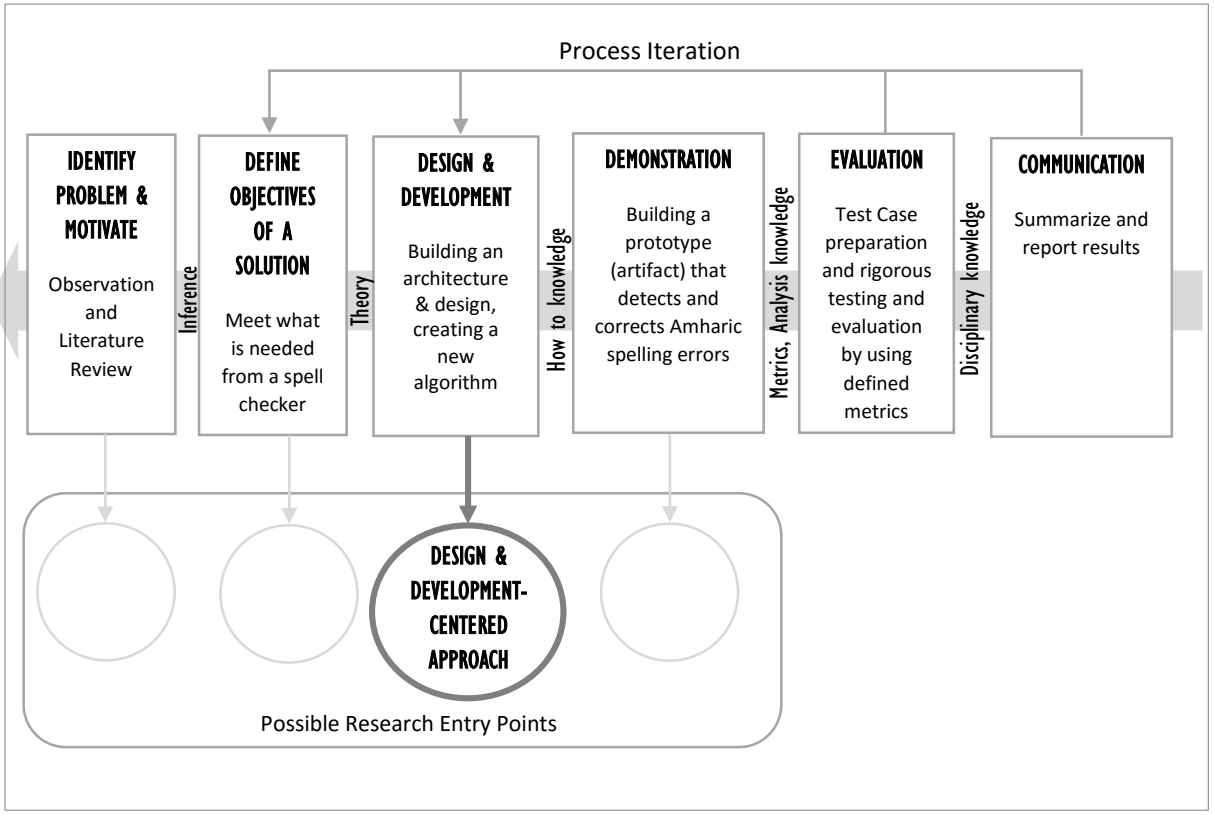


Fig. 3.1 DSRM Process Model (Adopted from [32])

i. Problem Identification and Motivation

In order to identify the problem, the researcher's prior observation and further investigation through literature review are applied. Journal articles, books, lecture notes, resources on the web are referenced and reviewed. Literature review is conducted to get a deeper understanding of Amharic morphology, to study previous approaches towards Amharic spell checking, and to identify improvement points. To understand more about the language, books written by linguistic experts are reviewed. They are referenced in every step of the research. To get a deeper understanding of the problem area, conceptual theories of research papers and books are reviewed.

ii. Definition of Objective

From the problem identified in the previous step and the solutions that were attempted by previous scholars, general and specific objectives of the solution to accomplish better results to the problem are inferred.

iii. Design and Development

The design and development process involves developing a design and architecture to detect and correct Amharic spelling errors. It is followed by designing an efficient algorithm that solves the problem defined on the first step and fulfills the objectives defined on the second step. The algorithm that is designed based on the designed architecture identifies incorrectly spelled Amharic words as invalid, generate closer suggestions to misspelled words and rank suggested words.

The components of the architecture are tokenizer, dictionary searcher, n-gram splitter, rule filter, stem extractor, distance calculator, suggestion generator and suggestion ranker. Details of each component are discussed in Chapter four.

iv. Demonstration

Demonstration involves implementation of the designed algorithm and building a prototype. At this stage, a prototype using python programming language is implemented.

a) Algorithm Development

The proposed methodology for spelling error detection and correction uses morphology-based approach for detecting misspelled words and generating suggestions. In order to make the system efficient, existing spell checking algorithms are studied.

Even though the algorithms discussed vary in the way they perform spelling error detection and correction, they share one common thing which is they all require a dictionary with the complete list of the lexicons. Generating or collecting the complete list of meaningful units for a resource limited and morphologically complex language like Amharic is a time consuming and costly task and if done results in a very large database. However, defining derivation rules, affixes, prepositions and conjunction and how they are related to the well-defined categories of stems [23] is evidently promising. Based on this justification, adopting these algorithms for Amharic spelling error detection and correction becomes unsound.

As a result, an algorithm that identifies probable error position, filters morphological rules and identifies the most likely rule that is applied to the candidate word, generates suggestions is developed. To rank suggested words, existing distance calculation algorithms are compared

and evaluated (Section 5.2.2). Based on the results found, a new distance calculator that gives a better result for Amharic is designed (Section 4.3 vi).

b) Prototype Development

A prototype that detects and corrects Amharic spelling errors is developed based on the developed algorithm and by referring to rule definitions and their relationships with root verbs. It is implemented in Python programming language.

c) Rule Definition

The prototype makes use of 717 morphological rules that are defined manually. Derivational and inflectional patterns discussed on [23][24][25] are studied and implemented.

The following is a sample rule to inflect the word **እንደተለማመዳችሁባቸውና** *(ənd*ätälämamädačəhubačäwəna*)* from the root word *l-m-d- (stem word lemed-).*

Position			4	3	2	1			1	2	3	4	5			
Rule Format	Rule Name	Pre	P4	P3	P2	P1	Stem	Derivation	S1	S2	S3	S4	S5	Conj	n	Verb / Noun
Sample1	Rule X	**እንደ**				**ተ**	**A1B1C1**	**A1B4B1C4**				**ችሁ**	**ባቸው**	**ና**	**3**	**Verb**

Table 3.1 Sample rule definition

In the above sample rule, the stem word pattern A1B1C1 shows that the stem word contains three characters each of which are in the first order. The derivation pattern A1B4B1C4 shows that the word is derived by taking the first character as it is, followed by the second character in fourth and first orders, then comes the last character in fourth order. The prefix '**ተ**' and the preposition '**እንደ**' are attached before the derivation pattern. The suffixes '**ችሁ**' and '**ባቸው**' follow the pattern. The conjunction '**ና**' can also be attached after the suffixes. Details about the slots on the rule definition structure is discussed in Chapter four.

Morphological rules are categorized into derived nouns, derived adjectives, derived verbs, derived pronouns and compound words [24]. In this study, most of the rules are focused on derived verbs because verbs exhibit a very complex morphology compared to other parts of speech [12]. To test the validity of the prototype on nouns, rules for derived nouns are included as well.

Derived Nouns

Nouns can be derived from verbs, adjectives, nouns, morphemes and by combining nouns. According to [24], there are twenty derivations patterns for nouns. In this study, 135 rules to change nouns to plural forms and other patterns are defined.

Derived Verbs

Verbs can be derived from root words by going through internal and external derivations. According to [24], there are twelve patterns of verb derivations that change the internal structure of stems and/or append affixes before and after the stem. In addition, rules that show situations and intensity of action are presented. Accordingly, 582 morphological rules that are representative of these patterns are defined. Table 3.2 shows the number of rules defined for each derivation style. Moreover, prefixes and suffixes that show the subject and doer of the action and the type of action (passive, causative, reciprocal, affirmative/ negative) [22] are also included in rule definitions. Prepositions and conjunctions that can go with the identified verb derivations are also part of the rule definitions. For instance, on the rule shown on Table 3.1, the pattern A1B4B1C4 shows a repeated action (ደራራጊ አዕማድ) [24]. The prefix ተ shows that the type of action is in passive form (ተደራጊ) [24]. The suffix ችሁ shows that the doer of the action is in third person (you /እናንተ/). The second suffix ባቸው shows that the receiver of the action is ‘they /እነሱ/. This rule can also take the conjunction ና which shows that there is a remaining action next to it. Words that follow this rule include እንደተለማመዳችሁባቸውና, እንደተለዋወጣችሁባቸውና, ተለዋወጣችሁባቸው, እንደተለማመዳችሁ and so on. Note that, the rule also works for words that fulfill some part of the rule definition.

Table 3.2 shows the distribution of morphological rules among the mentioned derivational and inflectional patterns.

	Verb and Noun Derivation Type	Amount
ደራጊ	Repeated action	10
ተደራጊ	Passive/ Reflexive	60
ተደራራጊ	Reciprocal	33
አድራጊ	Transitive/ Causative	22
አደራራጊ	Repeated Transitive/ Causative	17
አስደራጊ	Transitive/ Causative	40
አዳራጊ	Adjutative	14
ይሁንታዊ	Jussive passive	22
-	Negative action (Prefixes Al- and Ale-)	101
-	Other prepositions (Be-, Ende-, Ye-, Endemi-, Yemi-)	263
-	Plural form - Noun	15
-	Selected derivations - Nouns	120
-	*Verbal Nouns (Rules are embedded in preposition rules)*	*80 (included in preposition rules)*
		717

Table 3.2 Distribution of defined rules

v. Testing and Evaluation

The prototype development is followed by testing, analyzing and evaluating the results. This involves test case preparation, dictionary (knowledge base) preparation, annotation, testing and evaluation. In order to test the accuracy and performance of the system, two types of test cases were prepared. The test cases comprise of a total of 1724 words manually selected from an online available corpora [31] and taken from a chapter on the bible. Each word is manually checked and annotated as valid or invalid. The system compares the results found from the prototype with the annotations and reports words if they are correctly flagged or not.

a) Test Case preparation

In order to find out the reliability of the proposed approach, the system is tested on a test data manually selected from an online available corpora that is collected from different Amharic online documents and texts [33]. The test data contains combination of words spelled correctly and incorrectly. Test case taken from the Amharic bible is also used as an additional resource.

To decide the number of words the test cases contain, an analysis is made on the number of words used for testing in previous spell checking (detection and correction) researches. The number of test cases are summarized on Table 3.3.

Title	**Number of words in test cases**
Design And Implementation Of Morphology Based Spell Checker [5]	1464
Portable Spelling Corrector for a Less-Resourced Language: Amharic [19]	252
Development of an Amharic Spelling Corrector for Tolerant-Retrieval [13]	82
Morphology Based Spell Checker for Kafi Noonoo Language [14]	2743
Multilingual Spelling Checker for Selected Ethiopian Languages [18]	Not specified
Automatic spelling checker for Amharic Language [20]	1314
Automatic Amharic spelling error detection and correction using hybrid approach [34]	500
Average	**1059**

Table 3.3 Summary of number of test cases use in spell checking researches

As the table indicates, the number of test cases used by the researchers are not uniform. As a result, an average of the number of test cases is computed. Accordingly, two test cases which contain a total of 1724 words is used.

Test Case 1

Test case one was prepared by manually selecting different classes of words from an online corpus collected from the web [33]. It comprises of 1551 correct and 173 incorrect words which are selected based on the noun and verb derivation discussed on [24]. The selected words are manually flagged as valid and invalid. The test case contains words from more than 13

categories of derivational and inflectional patterns. The diversity of pattern in the test case preparation shows that the test case is representative of many of the word that can be found in a large corpus. Table 3.4 shows the number of distribution of categories in test case 1. In order to get a more accurate system and to thoroughly monitor the performance of the system, a group randomly selected words from the same source is also included.

		Total	**Valid**	**Invalid**
Test Case 1				
Passive Reflexive (Tederagi)		390	329	61
Transitive/ Causative (Adragi)		25	22	3
Repeated Transitive/ Causative (Aderaragi)		25	21	4
Transitive/ Causative (Asderagi)		89	75	14
Jussive Passive		14	14	0
Other derived verbs				
Negative action	Al-, Suffix = Combined	355	338	17
	Ale-, Suffix = Combined	159	156	3
Derived verbs with prepositions and combined suffixes	Be-, Suffix = Combined	72	53	19
	Ende-, Suffix = Combined	60	50	10
	Endemi-, Suffix = Combined	91	89	2
	Ye-, Suffix = Combined	24	21	3
	Yemi-, Suffix = Combined	111	102	9
Randomly selected		46	37	9
Nouns (Primary & Derived)		174	155	19
		1635	**1462**	**173**

Table 3.4 Disaggregation of verbs and nouns in Test Case One by derivation type

Test Case 2

To call a spell checker efficient, testing a whole paragraph from the same source is recommended [6]. Accordingly, three paragraphs from the Amharic bible have been taken and annotated as valid and invalid to form the second test case. Test case 2 comprises of 89 correct and no incorrect words in which 24 of them were derived from verbs and the rest from nouns, adjectives and adverbs. The words collected from the bible contain a total of 157 words. After removing duplicated words, it resulted in 88 unique words (Appendix 3).

b) Dictionary (Knowledge base) Preparation

The dictionary (list of words) used as one of the knowledge base for the prototype consists of sample stems from the major categories of Amharic parts of speech namely verbs, nouns,

adjectives and adverbs [24]. Simple random sampling technique is generally used to select root stems from what was collected from [17]. This technique is appropriate when the elements of data are homogenous and it makes sure that the probability of selecting each is the same and avoids human bias [35][36].

A total of 2398 stem words are stored in the dictionary and the composition of the data based on the category of the stem words is depicted in the following table (Table 3.5). Ten percent of the totally collected nouns, adjectives and adverbs were picked randomly as recommended in [37].

Category	**Total collected**	**Sample size**
Noun	11,250	1125
Adjective	3760	376
Adverb	190	19
Verb	2838	878
	Total	2398

Table 3.5 Composition of the dictionary based on the category of the stem words

The verbs were selected slightly in a different manner. Amharic verbs could further be broadly categorized as bi-radicals, tri-radicals and quadri-radicals [23]. Based on the order of the radicals, further categorization is presented in [23]. Thus, sample was taken from each subcategory, and sample size was determined based on the available collected data size. If the number of stem words in a given subcategory is greater than 100, ten percent of the words were randomly picked. Otherwise, all of them were stored in the dictionary [37]. Table 3.6 illustrates the selection of verbs as per their categories.

Category	**Sub category**	**Pattern**	**Total**	**Sample size**	**Example**
Bi-Radicals	1	A1B4	101	11	ነካ
	2	A1B4	22	22	ጠጣ
	3	A1B1	9	9	መሸ
	4	A1B1	12	12	አጨ
	5	A4B1	17	17	ዋኘ
	6	A4B1	86	86	ጣረ
	7	A4A4	12	12	ራራ
	8	A7B1	26	26	ሾመ
	9	A7B1	12	12	ቦካ
	10	A7B1	4	4	ቆየ
	11	A1B1	7	3	ሸጠ
	12	A1A1	1	1	ሸሸ
	13	A5B1	3	3	ሄደ
			Total	**218**	

Tri-radicals					
	1	A1B1C1	294	29	ቀበረ
	2	A1B1C1	572	58	ቀየጠ
	3	A1B1C1	15	15	ፈነጨ
	4	A1B1C4	60	60	በረታ
	5	A4B1C1	126	13	ዳለጠ
	6	A7B1C1	19	19	ቆሰለ
	7	A7B1C1	95	95	ሾፈረ
	8	A7B1C1	6	6	ቆነጀ
	9	A7B1C4	5	5	ጎመራ
				Total	**300**
Quadri-radicals	1	A1B1C1D1	231	23	ሰነዘረ
	2	A1B4C1D1	93	93	ቀባጠረ
	3	A7B1C1D1	41	41	ሞነጨረ
	4	A1B1A1B1	178	18	መነመነ
	5	A1B4C1D1	5	5	ወላወለ
	6	A7B1A7B1	44	44	ቆሰቆሰ
	7	A1B1C1C1	97	97	ሸለቀቀ
	8	A7B1C1C1	20	20	ቆነደደ
	9	A1B1A1D1	9	9	ከረከመ
	10	A7B1A7D1	8	8	ኮረኮመ
	11	A1B1C1D1	2	2	ፈለሰፈ
				Total	**360**

Table 3.6 Selection of verbs inside the dictionary as per their categories

c) Annotation

The words in the test cases are manually annotated as valid and invalid. Words which tend to follow a certain pattern but with no meaning are identified as invalid. The rest are annotated as valid. Once the system flags a word as valid or invalid, it compares the result with the given annotation and notifies whether the system gives the intended result or not.

d) Testing and Evaluation

The prototype is tested and evaluated against the prepared test cases. The performance of the prototype is measured by comparing the results with the manually annotated test cases.

The results found from the error detection experimentation phase of the proposed system are classified by using a classifier evaluation metrics called confusion metrics. “Given m classes, an entry, CMi,j in a confusion matrix indicates number of tuples in class i that were labeled by the classifier as class j” [38].

Actual class/ Predicted class	Predicted Correctly Spelled Words	Predicted Misspelled Words
Actual Correctly Spelled Words	True Positives (TP)	False Negatives (FN)
Actual Misspelled Words	False Positives (FP)	True Negatives (TN)

Table 3.7 A confusion matrix showing classification of results of the experiment

True positives (TP) – number of correctly spelled words that were correctly identified as valid

True negatives (TN) – number of misspelled words that were correctly identified as invalid

False positives (FP) – number of misspelled words that were incorrectly identified as valid

False negatives (FN) – number of correctly spelled words that were incorrectly identified as invalid

According to [39], to evaluate the performance of a spell checker, precision, recall and predictive accuracy measures are used. Precision measures what percentage of words the system label as valid are actually valid. Precision is related to accuracy of the system in flagging words. That is, it measures the accuracy of the system in identifying only correct words as valid and only incorrect words as invalid [39]. Precision can be divided into two: Lexical Precision and Error Precision.

Lexical Precision is calculated by dividing number of correctly spelled words that were correctly identified (True Positives /TP/) by the number of words that are identified as valid (the sum of True Positives /TP/ and False Positives /FP/) [39].

$$Lexical\ precision = \frac{TP}{TP + FP}$$

Error Precision is calculated by dividing number of wrongly spelled words that were correctly identified (True Negatives /TN/) by the number of words that are identified as invalid (the sum of True Negatives /TN/ and False Negatives /FN/) [39].

$$Error\ precision = \frac{TN}{TN + FN}$$

Recall measures the percentage of words the system correctly flags as valid or invalid. Lexical recall measures the percentage of correctly spelled words the system label as valid [6]. Error recall measures the percentage of incorrectly spelled words the system label as invalid.

Lexical Recall is calculated by dividing number of correctly spelled words that were correctly identified (True Positives /TP/) by the sum of correct words that are identified as valid (True Positives /TP/) and the number of correct words that are identified as invalid (False Negatives /FN/) [39].

$$Lexical\ recall = \frac{TP}{TP + FN}$$

Error Recall is calculated by dividing number of incorrectly spelled words that were identified as invalid (True Negatives /TN/) by the sum of incorrect words that are identified as invalid (True Negatives /TN/) and the number of incorrect words that are identified as valid (False Positives /FP/) [39].

$$Error\ recall = \frac{TN}{TN + FP}$$

Predictive Accuracy of a spelling checker shows the possibility of handling any candidate word accurately, whether the word is valid or invalid [39]. It is computed by the following formula:

$$Predictive\ Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

The error correction and suggestion capability of the system is measured by checking whether the intended correction is situated inside the list of closer words displayed by the system [19]. From the generated list, suggestions whose distance with the candidate word is above 0.65 are displayed. This threshold is computed from an analysis made by comparing the distance between the candidate word and the suggestions generated by the system. The analysis shows that the intended word is located in suggestions whose distance is above the mentioned value.

vi. Communication

The final step involves summarizing the test results and presenting conclusion and recommendation from the test results. Challenges during the DSRM process are also presented and reported at this stage.

CHAPTER FOUR

SYSTEM DESIGN AND ARCHITECTURE

4.1 Overview

This chapter discusses the architecture of the system, the designed Amharic spelling error detection and correction algorithm, the rule definition structure designed as an input for the algorithm. Moreover, stored knowledge base components used by the prototype are discussed.

Section 4.2 discusses the general architecture of the system. Components of the architecture and their functionalities are thoroughly discussed in Section 4.3. Section 4.4 discusses the design of the system and the flow of steps to detect and correct spelling errors in Amharic.

4.2 General Architecture

The spell checker consists of three main components namely Preprocessor, Error Detector and Error Corrector. The preprocessor component is responsible for accepting a block of text from the user, tokenizes each word, removes punctuation marks except the compound words concatenator hyphen '-' character, removes duplicates and passes candidate words to the next component. The Error Detector component accepts a candidate word from the preprocessor component and analyzes the word for spelling errors. To do so, it splits the word into unigrams, bigrams and trigrams. Next, it compares the n-grams with affixes on existing sets of rules. It filters rules with matching affixes. Each rule is analyzed till the probable stem of the word is identified. The derivation pattern of the stem is compared with the one on the rule. If the word fully matches the definition of one of the filtered rules, the word is flagged as valid. This component flags the word as invalid if none of the rule definitions match.

The Error Corrector component accepts invalid words, estimated location of errors and rule weights from the previous component. It uses this information to generate suggestions that are closer to the candidate word. It compares the generated suggestions with the word and takes and returns suggestions whose distance with the candidate word is above 0.65. The algorithm on Listing 4.1 shows the general steps followed to spell check an Amharic candidate word.

```
        BEGIN
1               Search WORD in Dictionary Lookup
2               if WORD is found
3                       result = Correctly Spelled
4                       Mark WORD as Correctly Spelled
5               else
6                       Divide WORD into n-grams
7                       Search matching Conjuctions, Preposition and Affixes
8                       if affixes match
9                               Compare preceding and following affixes
10                              Read derivation pattern
11                              Compare Derivation Pattern
12                              if derivation pattern matches
13                                      reverse word to stem form
14                                      search generated stem in dictionary lookup
15                                      if generated stem is found
16                                              result = Correctly Spelled
17                                      else
18                                              result = Incorrectly Spelled
19                                              Mark error position
20                              else
21                                      result = Incorrectly Spelled
22                                      Mark error position
23                      else
24                              result = Incorrectly Spelled
25                              Mark error position
26                      If result = Incorrectly Spelled
27                              Replace errors with probable correct char at error position
28                              Generate Suggestions
29                              Find closer stem words and generate suggestions
        END
```

Listing 4.1 General algorithm used by the spell checker

The spell checker accepts the candidate word and searches it inside the stored words knowledge base. If the word is found, it marks the word as correctly spelled *(Lines 1-4)*. If the word is not found, it divides the word into n-grams (unigrams, bigrams and trigrams) and searches the morphological rules database for matching affixes, conjunction and preposition *(Lines 5-7)*. If there are matching affixes, it continues to compare the preceding and following affixes on the matched rules *(Lines 8-9)*. When there are no more matching affixes, it generates the derivation pattern of the processed word and compares it with the rule in question. If the derivation pattern matches, it reverses the word to its stem form and searches it in dictionary lookup. If it is found, it marks the word as correctly spelled, if not it flags it as incorrectly spelled and marks error position *(Lines 10-19)*.

If all derivation patterns of the matched rules do not match, it marks the word as incorrectly spelled and marks the position of the error. Error position is marked in any occasion when mismatch in rule definition occurs. This is later used to generate corrections *(Lines 20-22)*. If affixes do not match or are not close to each other, it marks the error position and labels the

word as invalid *(Lines 23-25)*. If the result is incorrectly spelled, it replaces characters at error position with the probable correct characters and generate suggestions. It also searches for closer stem words and does the same replacement on closer words and generate suggestions *(Lines 26-29)*.

4.3 System Architecture and Algorithm Design

The spell checker works by accepting a block of text, splitting it into words and analyzing each word for spelling error. It then identifies the word as either valid or invalid and gives a closer suggestion for incorrect words. The spell checker follows three steps to accomplish this. These are Preprocessing, Error Detection and Error Correction. Fig. 4.1 shows the general architecture of the system.

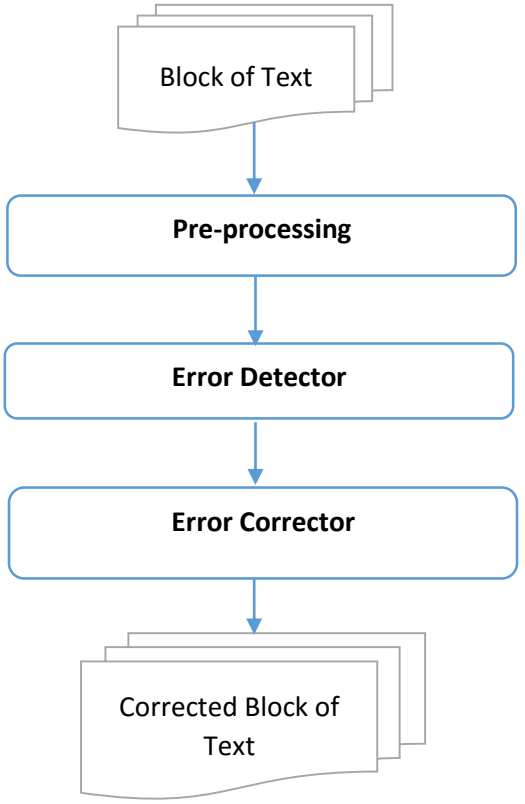


Fig. 4.1 General Architecture of the System

The system uses three stored databases in the background. These are Morphological rules, Dictionary lookup and Amharic characters database. It also uses Distance Calculator component which is part of Rule Filter, Stem Extractor and Suggestion Ranker components. Fig 4.2 shows the detailed architecture of the system. The functionalities of each component are discussed in the following section.

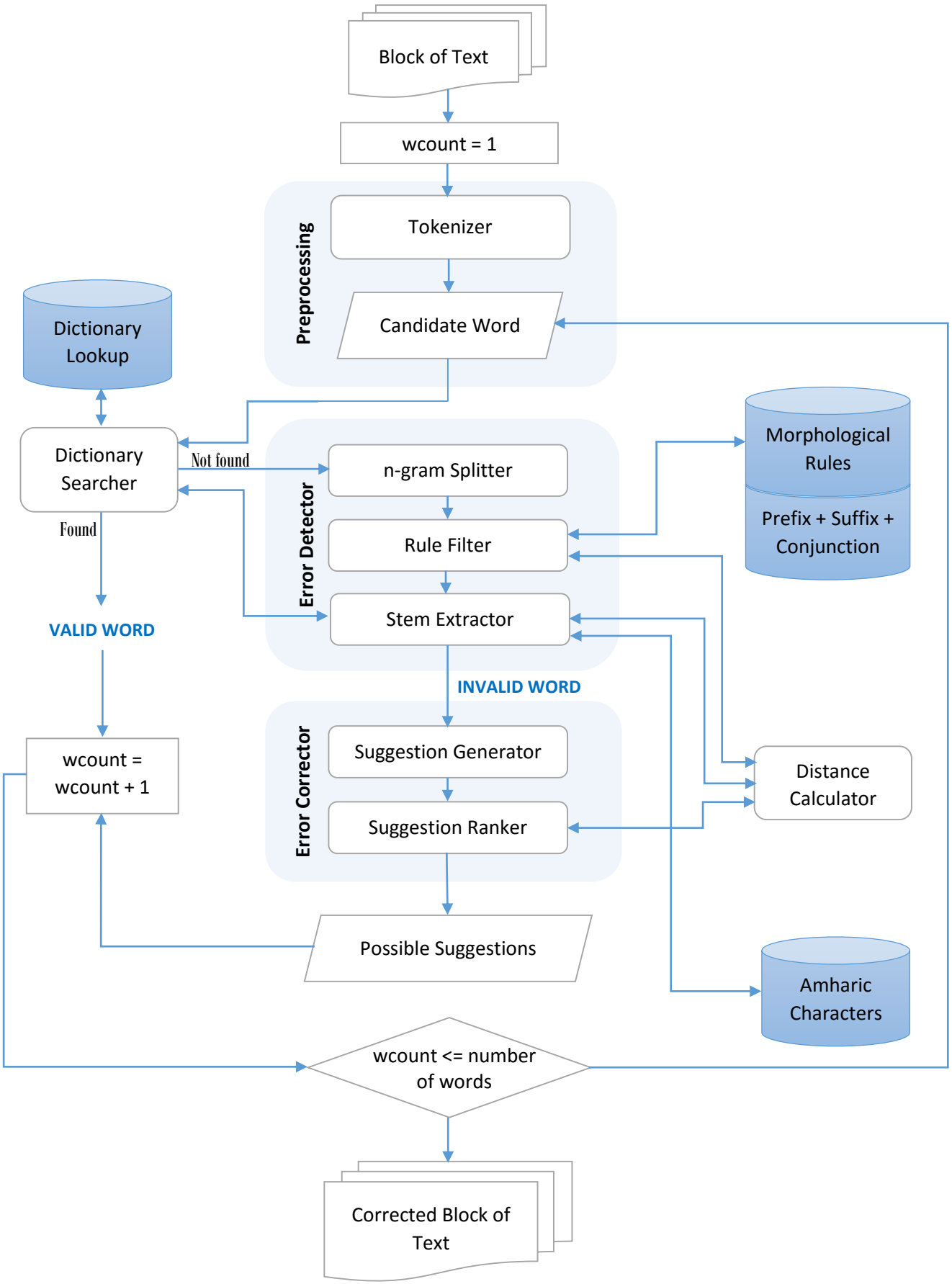


Fig. 4.2 Detailed Architecture of the System

i. Tokenizer

In the preprocessing step, this component is responsible to segment a given block of text into tokens for further processing [40]. The process involves splitting the block of text at the point where a separator character (in this case space) is detected. It also involves removing punctuation marks except hyphen '-' which is used to combine compound words [40]. In addition, it removes duplicated words and passes one instance of each repeated word. Each word is sequentially given to and processed by the spell checker. It works by looking for a separator character, in this case space and punctuation marks and splitting the text at the point where the separator is detected. Once the splitting is done, puncuation marks except the compound words concatenator are removed. Taking compound words into consideration, punctuation mark hyphen '-' is kept. Compound words are made of two words connected by the punctuation mark '-' and have meanings that is different from the words they are made of. Since there is no standad in writing compound words [26], in this research, it is assumed that compound words are written either separated by a hyphen '-' or as a single word. Compound words separated by a space are not considered. Listing 4.2 shows the algorithm used by the tokenizer.

```
    BEGIN
1       Read block of text
2       split point = 0
3       Do while end of document is reached
4           If the character is different from separator or punctuation mark
            except hyphen
5               Skip
6           Else
7               Mark as split point
8               previous split point = split point
9               split point  =  current position
10              Split the word from current to previous split point
11      Return word
```

Listing 4.2 Algorithm to tokenize block of text to words [40]

The algorithm to tokenize a block of text starts working by reading a block of text and initializing the first split point, which is at position zero of the block *(Lines 1 & 2)*. Until the end of the document is reached, it iterates through each character in the text till it finds a separator character or punctuation marks except hyphen character. Separator characters in Amharic can be space or punctuation marks represented by two dots (፡) or four dots (።). Other

punctuation marks include '።', '፤', '፧' and '!'. When it finds one of the specified characters, it marks the first split point as the previous split point and marks the current position as the split point. It then takes out the characters starting from the previous split point to the current one and returns the resulting word. This process is repeated till the end of the block of text is reached *(Lines 3 & 11)*.

ii. Dictionary Searcher

This component accepts a word from the *Tokenizer* or *Rule Filter* component and searches the *Dictionary Lookup* for the word. If the word is found, the candidate word will be marked as valid. If the dictionary searcher receives the word from the *tokenizer* component and if the word is not found, it sends the word to the *n-gram splitter* for further processing. If it receives the word from the *Rule Filter* component and if the word is not found in the dictionary, it is marked as invalid.

```
    BEGIN
1       Read candidate word, type
2       n = 0
3       found = false
4       next word = word[n]
5       word type = type[n]
6       Do While found = true OR end of dictionary = true
7           if candidate word = next word and type = word type
8               found = true
9               break
10          else
11              next word = word[n+1]
12      End While
13      Return found
    END
```

Listing 4.3 Algorithm to search the dictionary lookup component

The algorithm on Listing 4.3 reads the candidate word and word type, initializes variables that hold the position of a word in the dictionary, that holds the next word and type to be compared and a flag that tells if the candidate word is found in the dictionary or not *(Lines 1 – 6)*. It then goes through the words in the dictionary till the word is found or end of dictionary is reached. If a matching word and type is found, it changes the variable 'found' to 'True' and breaks the loop. Otherwise, the loop will end keeping the value 'False' inside the variable 'found'. It finally returns True if the word is found and False if not *(Lines 7 – 13)*.

iii. n-gram Splitter

This component receives a word from the *Tokenizer* component and splits it into unigrams, bi-grams and tri-grams. It passes the n-grams to the *Rule Filter* component. The length of Amharic affixes can range from one to three characters. That is why the number of characters in the n-grams is limited to three.

Candidate word - **እየሰባበረ**

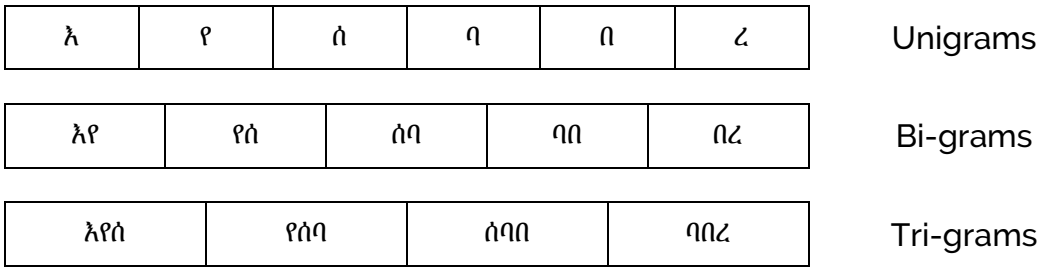


Fig. 4.3 n-grams generated from the word እየሰባበረ (əyäsäbabärä)

```
        BEGIN
1           Read candidate word
2           Let n = 1
3           While n <= 3
4               Split word into n number of sequences
5               if n = 1
6                   Store in unigrams
7               if n = 2
8                   Store in bigrams
9               if n = 3
10                  Store in trigrams
11              n = n + 1
12          End While
13          Return unigrams, bigrams, trigrams
        END
```

***Listing 4.4 Algorithm to split a given word into n-grams* [41]**

The n-gram splitter algorithm on Listing 4.4 reads the candidate word and initiates the variable n as one. n decides the number of n-grams *(Lines 1 & 2)*. Then, if n = 1, the given word is divided into a set of unigrams, if n = 2 and n = 3 into bigrams and trigrams respectively. The value of n is increased by one till n is less than or equal to three. It finally returns list of unigrams, bigrams and trigrams *(Lines 3 - 13)*.

iv. Rule Filter

This is the main part of the spell checker with fundamental functionalities. The *Rule Filter* component accepts splitted characters from the *n-gram splitter*. It searches for rules which have matching conjunctions, prepositions, prefixes and suffixes inside stored morphological rules.

The *Rule Filter* component accepts unigrams, bigrams and trigrams at the same time from the *n-gram splitter* and performs several actions. The rule filter does not process a rule for just one n-gram. A rule can contain more than one n-gram, therefore the rule filter processes rules for combinations of unigrams, bigrams and trigrams. These actions are performed by sub-components with distinct functionalities.

Search Conjunctions – searches morphological rules knowledge base for the unigram, bigram and trigram which is located at the end of the word. That is the last set of characters in the list received from the n-gram splitter *(Line 2)*.

Search Prepositions – Similarly, matching prepositions to unigram, bigram and trigram at the beginning of the word are searched and retrieved from the morphological rules knowledge base *(Line 3)*.

Search Prefixes – looks for matching prefixes inside Rules database. Similar to suffixes, only the prefix at position one is checked for matching at this stage *(Line 4)*.

Search Suffixes – searches character(s) received from the previous component at the fifth position of suffixes on defined rules. A word can have more than one suffix. However, at this stage, only the last suffix (position five) is checked for matching and the rest are handled in the steps that follow *(Line 5)*.

After matching rules are identified, common rules that fulfill more than one criteria are grouped together. i.e. The rule filter filters rules with prepositions, conjunctions or affixes with in the unigrams, bigrams and trigrams passed from the previous component. Once the Rule Filter searches for matching affixes, it drops the matches from the candidate word. Then, the resulting word is processed with the same rule definition. If the matches are suffix and conjunction, it checks to see if the character/s in the preceding column of the rule definition table is the same as that of the word. It successively cuts one, two and three characters from right to left and compares them with the suffixes on the preceding column of the rule. If one of them is a match, the character/s will be dropped from the processed word *(Lines 6 - 10)*.

```
BEGIN
1       Read candidate word, unigrams, bigrams, trigrams
2       Conjunction Rules = search matching conjunctions
3       Preposition Rules = search matching prepositions
4       Prefix Rules = search matching prefixes
5       Suffix Rules = search matching suffixes
6       if there are matching conjunction or suffix rules
7           Do While no preceding suffix and no closer preceding suffix
8               if there are preceding suffixes
9                   processed word = drop Suffix
10                  Process next suffix
11              else
12                  search closer preceding suffixes
13                  if there are closer preceding suffixes
14                      replace with correct suffix
15                      mark error position
16                      drop corrected suffix
17                      process next suffix
18                      add rule weight
19                  else
20                      pass processed word to Stem Extractor
21          End While
22      else if there are matching preposition or prefix rules
23          Do While no following prefix and no closer following prefix
24              if there are following prefixes
25                  processed word = drop prefix
26                  Process next prefix
27              else
28                  search closer following prefixes
29                  if there are closer following prefixes
30                      replace with correct prefix
31                      mark error position
32                      drop corrected prefix
33                      process next prefix
34                      add rule weight
35              else
36                  pass processed word to Stem Extractor
37          End While
38      Return Processed Word, error positions, error correction, rule weight,
        dropped affixes
END
```

Listing 4.5 Algorithm used to filter rules

If none of the preceding characters match, it searches for affixes which are closer to first, second and third characters taken from right to left. To call an affix closer, the distance between the affix and the n-gram should be above a threshold value 0.5 *(Line 12)*. If no affixes are close, it passes the full word to the stem extractor *(Line 19 & 20)*.

If there are any closer affixes, the system assumes that this is the position where the error occurred. It marks this position and later passes it to the *Suggestion Generator* component. It replaces the possibly wrong affix from the word with the closer affix from the rule definition and increments the weight of the rule. It then drops the corrected suffix and repeats the same process to the resulting word till there are no more matching or closer affixes *(Lines 13 - 18)*.

If the matching affixes are preposition and prefix, it checks if the following affixes match first, second and third characters of the word from left to right. The processes that follow are similar to the one discussed for matching suffixes *(Lines 22 - 37)*. In case of common rules, the steps explained are merged to process the word.

Rules with matching preceding or following affixes and rules with matching closer preceding or following affixes are given a higher rule weight. This helps to identify the most probable derivation rule used to derivate the given word, whether the word has correct spelling or not. The rule weight helps to rate identified rules and figure out the rule that is used to inflect the candidate word.

Finally, the processed word, the error positions marked, the error corrections identified, the dropped affixes and weight of each rule processed are returned and passed to the stem extractor component *(Line 38)*.

v. Stem Extractor

Once the *rule filter* gives up, it gives the extracted word to the *stem extractor*. This component is responsible to identify the probable stem word. First, it checks if the derivation pattern matches the one on the selected rule passed from the *Rule Filter* component. If it does, it reverses the word to its stem form. The stem pattern is also part of the selected rule. Once the stem is extracted, it will be sent to the dictionary searcher.

The Stem Extractor has two functionalities namely generating derivation pattern and reversing the pattern.

a) Generate Derivation Pattern – generates the derivation pattern of the word

```
    BEGIN
1       Read candidate word
2       Character Representations = [A, B, C, D, E, F, G, H . . . .],  n = 0
3       While n < length of word
4           Read character n
5           Char Repr = Search character in Amharic Characters Database
6           Order = find order in Amharic Characters Database
7           Concatenated = Char Repr[n] + Order
8           Derivation Pattern = Derivation Pattern + Concatenated
9       End While
10      Return Derivation Pattern
    END
```

Listing 4.6 Algorithm used to generate derivation pattern

The algorithm that generates the derivation pattern works by reading a candidate word and initializing English letters that represent each character in the word *(Lines 1&2)*. Till it reaches the end of the word, it reads each character from Amharic characters' database component and finds out the order. It then represents each character by an English letter followed by the order of the character. If there are one or more characters with the family, they will be represented by the same English letter. Finally, it returns the pattern generated *(Lines 3-10)*.

b) Reverse Derivation - reverses the word to its stem form

```
    BEGIN
1       Read Derivation Pattern
2       Read Word Pattern, word
3       Character Representations = [A, B, C, D, E, F, G, H]
4       n = 0
5       While n < length of derivation word
6           Read character n
7           Read order of character n
8           Do While character n is found in Word Pattern
9               Search character n
10              Retrieve order of character n
11              Reversed Char = Change derivation order to word order
12              Reversed Word = Reversed Word + Reversed Char
13      End While
14      Return Reversed Word
    END
```

Listing 4.7 Algorithm used to reverse derived word to its stem

The above algorithm reads the derivation pattern and the word pattern of the specified rule from stored rule definitions and the word from the system *(Lines 1&2)*. It then initializes variables that store English letters for character representation and the number of processed characters *(Lines 3&4)*. The derivation pattern passed to this method is the same as the given word's pattern. Keeping this in mind, it reads each character of the word and its order from the derivation pattern and changes it to the corresponding order on the word (stem) pattern. It appends the reversed character to the reversed word till the end of the word is reached. At last, it returns the reversed word *(Lines 5-14)*.

vi. Distance Calculator

This component is responsible to find the distance between two Amharic words. It uses an algorithm specially designed for Amharic characters. It is shown by a preliminary experiment done to compare distance calculation algorithms that none of the algorithms give the intended result to find the difference between two Amharic words (See Section 5.1.2).

The distance calculator uses the algorithm shown on Listing 4.8. The algorithm starts by finding matching blocks of character(s) between the two words. It also identifies un-matching blocks of text for both words. Leaving the matched blocks, it compares the distance between combination of characters of unmatched blocks in the same position and computes the average. Characters in the same row of an Amharic alphabet are given a distance of 0.7 because it is assumed that the user has most likely typed one of the characters correctly. This assumption is made because it takes to press the main key which holds the family followed by a vowel to change its order. For example, to type the character ሰ (*sä*) using Power Ge'ez, one has to press the 'S' key followed by 'E'. And, to type ሱ (*su*), 'S' followed by 'U'. So, if the user mistakenly typed *su* for *sä*, the difference between the two is very small resulting in a higher value. The distance calculator is designed in such a way that distance of one means, the compared words are exactly the same. That is, the value keeps decreasing to zero when the difference between the words keeps getting wider (Fig 4.5). As a result, in this algorithm, the higher the distance, the closer the words are.

Characters in the same column of the Amharic alphabet are given 0.5 distance. This is assuming that the user typed the order of the character correctly and mistyped the family. The distance is less from the first case because it is unlikely to make such mistake than the previous. This is confirmed by the analysis explained in Section 5.2.1. Similarly, characters with no matching family and order are given 0.3 distance. Substituting a character with another one from a

different family and order is also less likely (Section 5.2.1). Distance between any character and a missing character is given a distance of 0.1 because it is clear how far the two are. On the contrary, the same characters are given a distance of one. Fig 4.4 depicts the above explanation with sample characters.

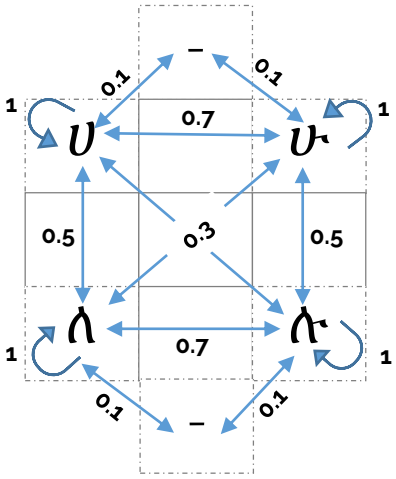


Fig. 4.4 Values used by the distance calculator

To identify the weights between Amharic characters of different family and order, an analysis is made on two words (ሰራው and ሰሩት) which contain a pair of the same characters, a pair of characters in the same family and a pair of characters in the same order.

First, the following assumptions are made:

i) Distance between two words ranges from 0 to 1 (0 < d <= 1). Value 1 shows the words are exactly the same.

ii) As the depiction on fig 4.5 shows, five conditions are identified and their value lies between 0 and 1.

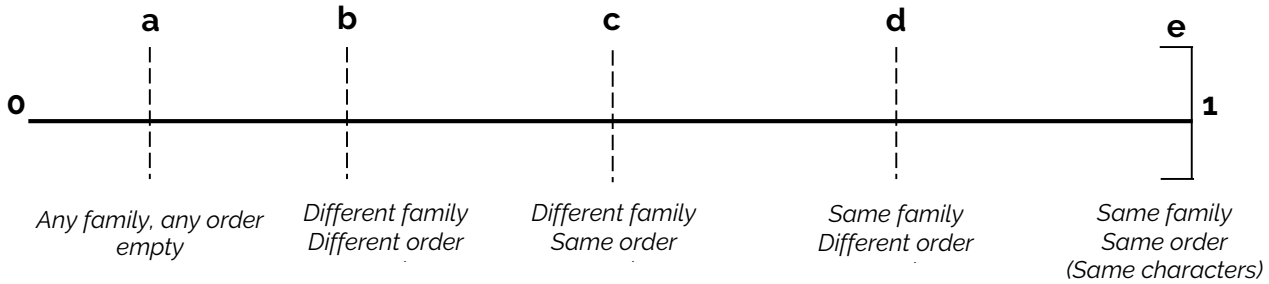


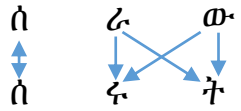
Fig. 4.5 Distance between characters by family and order

iii) Distance between characters of different family and order lies in the middle, that makes the distance 0.5

iv) Distance between characters of same family and different order is greater than the average and distance between characters of different family and different order, which

is in turn is less than the average value. Because, the probability of making the first mistake is higher than the second (Section 5.2.1)

In order to figure out the distance between characters on conditions b and d (Fig 4.5), distance between words (ሰራው and ሰሩት) is calculated for each decimal values 0.1, 0.2, 0.3, 0.4 for the first and 0.6, 0.7, 0.8 and 0.9 for the later.



D = d(ሰ, ሰ) + d(ራ, ሩ) + d(ራ, ት) + d(ው, ሩ) + d(ው, ት)

		Different family Different order			
		0.1	**0.2**	**0.3**	**0.4**
Same family Different order	**0.6**	0.46	0.5	0.54	0.58
	0.7	0.48	0.52	0.56	0.6
	0.8	0.5	0.54	0.58	0.62
	0.9	0.52	0.56	0.6	0.64
Sum		1.96	2.12	2.28	2.44

Table 4.1 Calculation to compute weights on distance calculator

$$Average\ distance = \frac{Sum\ of\ values\ computed\ for\ each\ combination}{Number\ of\ combinations}$$

$$Average\ distance = \frac{6.84}{16} = 0.55$$

As shown on Table 4.1, a value close to 0.55 is computed when values 0.7 and 0.3 are used. From this analysis, distance between characters of same family and different order is 0.7 and distance between characters of different family and same order is 0.3.

Distance between any character and an empty character is computed after doing the same analysis as above.

The algorithm on Listing 4.8 shows the steps used by the distance calculator. It starts working by reading the words that are going to be compared. It then initializes variables to store distance between characters and number of comparisons *(Lines 1 & 2)*. It then finds matching and unmatched blocks of text *(Line 3 & 4)*. For each un-matching blocks of character(s), it takes all the possible combinations of characters and compares their distance based on the conditions

and values discussed *(Line 5-19)*. It then adds distance between matched blocks i.e. length of matched block to the distance computed from the unmatched blocks *(Line 20-21)*. Finally, it computes the difference by dividing the distance by the number of comparisons and returns it *(Line 22-23)*.

```
    BEGIN
1       Read word1, word2
2       comparisons = 0, distance = 0
3       Find matching block of characters
4       Find unmatching block of characters
5       for n in unmatching blocks
6           for i in len(unmatching block)
7               char1 = charcter in first block
8               char2 = charcter in second block
9               comparisons += 1
10              if family(char1) = family(char2) and order(char1) = order(char2)
11                  distance += 1
12              elseif family(char1) = family(char2) and order(char1) != order(char2)
13                  distance += 0.7
14              if family(char1) != family(char2) and order(char1) = order(char2)
15                  distance += 0.5
16              if family(char1) != family(char2) and order(char1) != order(char2)
17                  distance += 0.3
18              if char1 = '' or char2 = ''
19                  distance += 0.1
20      for m in matching block
21          distance += len(matching block)
22      difference = distance / comparisons
23      return difference
    END
```

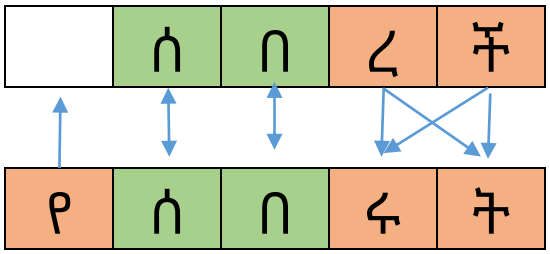
Listing 4.8 Algorithm to calculate distance between two words

The following demonstration (Fig 4.6) shows how the distance calculator is used to calculate the difference between *ሰበረች* and *የሰበሩት*.

Example 1

ሰ	በ	ረ	ች

የ	ሰ	በ	ሩ	ት



d(_, የ) = 0.1

d(ሰ, ሰ) = 1

d(በ, በ) = 1

d(ረ, ሩ) = 0.7

d(ረ, ት) = 0.3

d(ች, ሩ) = 0.3

d(ች, ት) = 0.5

Total distance = 0.1 + 1 + 1 + 0.7 + 0.3 + 0.3 + 0.5

= 3.9

Total number of comparisons = 7

$$\text{Difference} = \frac{\text{Total distance}}{\text{Total number of comparisons}}$$

= 3.9/7

= **0.56**

Fig. 4.6 Distance calculation to calculate the difference between ሰበረች (säbbäräč) and የሰበሩት (yäsäbbärut)

vii. Suggestion Generator

The *suggestion generator* accepts the most probable position and cause of the spelling error from the *rule filter* and *stem extractor*, use that to generate possible suggestions. It reverses back the error on the positions received from the previous steps. It also reattaches all the dropped affixes, conjunctions and preposition in the rule filter component. It accepts list and location of correct affixes, preposition, conjunction from the rule filter component and appends them to the processes derived stem. When it does that, affixes with errors are replaced with corrections passed from the *rule filter*. Stem words which are closer to the extracted stem word are also used to generate suggestions. The distance calculator finds closer stem words and the suggestion generator follows the same derivation style to change the new stem to the form the candidate word follows. Finally, it calculates the distance between the generated words and the candidate word and passes it to the next step.

viii. Suggestion Ranker

This component receives list of suggestions and their distance from the candidate word from the suggestion generator. From the generated list, suggestions whose distance with the

candidate word is above 0.65 are selected and displayed for the user. An analysis made by comparing the distance between the candidate word and the suggestions generated by the system shows that the intended word is located in suggestions whose distance is above the mentioned value.

Stored Knowledge bases

The architecture consists of three stored knowledge bases namely Dictionary Lookup, Morphological Rules and Amharic Characters database.

i. *Dictionary Lookup*

Dictionary lookup is a knowledge base used by the spellchecker which contains stored set of Amharic stem words (verbs, nouns, adjectives and adverbs). Stem verbs in Amharic are characterized by having a sixth order character at the end. However, the stem words used in this repository are slightly different. For ease of use, verbs showing actions by a masculine subject are stored (Appendix 2). The spell checker will look for the given word in this repository and if not found, it will jump to the next step.

ii. *Morphological Rules*

A morphology-based spellchecker works by using a stored set of rules defined manually. The morphological rules component contains rules to derivate and inflect an Amharic stem word to different forms. To make the system more efficient, the rules are stored in a csv file. The file contains fourteen columns that define the characteristics of the rule. Table 4.2 shows the structure used to manually define the rules.

Position			4	3	2	1			1	2	3	4	5			
Rule Format	Rule Name	Pre	P4	P3	P2	P1	Stem	Derivation	S1	S2	S3	S4	S5	Conj	n	POS
Sample 1	Rule 1	እንደ				ተ	A1B1C1	A1B4B1C4				ችሁ	ባቸው	ና	3	Verb
Sample 2	Rule 2	እንደ					@1_Z6	@1_Z7				ች			-	Noun
Sample 3	Rule 3						-	-	ዬ						-	Noun
		Preposition	*Prefixes*				*Derivation Pattern*		*Suffixes*					*Conjunction*	*Number of characters*	

Table 4.2 Structure of morphological generation rules

An Amharic word can have up to four prefixes and five suffixes [22]. Taking this into consideration, the rules are designed to have slots for the mentioned amount of affixes. In addition, preposition which comes in the beginning of a word and conjunction which comes at the end are given a different slot.

The derivation pattern defines the way the characters in the given stem change their order to give the inflected word. The number of characters in the stem is also stored to make the searching easier and efficient for the spell checker.

The first column *Rule_Name* shows the unique name given to each rule. Giving a unique name to rules makes it easier to identify and process them during error detection and correction. The next column is where the preposition used by the rule is stored. If the rule has no preposition, it stays empty. The next four columns specify the possible prefixes that follow the preposition. Four slots are dedicated because prefixes can range from one to four. The next two columns specify the stem and derivation patterns of the rule. As can be seen from the samples, these columns can be fully represented, partly represented or not represented at all.

On Table 4.2, on the first sample, each character is represented by English letters followed by the order of the character on the Amharic alphabet. For instance, Sample 1 shows the formation of ሰባበራ- (*säbabära-*) from ሰበረ (*säbärä*).

ሰ	በ	ረ
A	B	C
1	1	1
A1B1C1		

ሰ	ባ	በ	ራ-
A	B	B	C
1	4	1	4
A1B4B1C4			

***Fig. 4.7 Stem and derivation pattern to inflect the word ሰበረ** (säbärä) **to ሰባበራ-** (säbabära-)*

However, the derivation and stem pattern of the rules might be partly known. The second sample shows such scenario. For example, for nouns ending in six order character, adding the suffix ኦች could result in the rule specified on Sample 2.

አ	ገ	ሮ	ች
@		Z	
1		6	
@1_Z6			

አ	ገ	ሮ	ች
@		Z	
1		7	
@1_Z7			

***Fig. 4.8 Stem and derivation pattern to change the noun አገር** (agär) **to its plural form አገሮች** (agäroč)*

The third sample shows a rule whose inflection does not depend on the stem and derivation pattern. For example, derivation of ሁለመናዬ from ሁለመና.

The next five slots are dedicated for possible suffixes that follow the word derived by the derivation pattern. The number of suffixes can range from one to five. The next column is where conjunction that follows the suffix is stored. The columns next to last shows the number of characters of the stem word. This column helps not to make unnecessary search inside the repository of words. The last column is where we specify the word class the rule applies to (either noun or verb). This also helps in making the searching efficient.

iii. Amharic Characters Database

This is where all characters in Amharic are categorized and stored. The format follows the representation used by the Amharic alphabet set 'fidel' (Appendix 5).

4.4 System Design

Morphologically complex languages like Amharic are characterized by having a lot of complex derivation rules. Defining all morphological generation/ derivation rules and performance issue of going through all rules to detect spelling error are the drawbacks with morphology-based approach. The system uses an algorithm that solves this performance issue. Fig 4.9 shows a flowchart that depicts the workflow used by the system.

As the flowchart depicts, it tokenizes the given sentence and processes one word at a time. It reads a word and searches it inside the dictionary of stored stems. If it is not found, it divides the word to n-grams and starts comparing the n-grams with prefixes, suffixes, conjunctions and prepositions on rule definitions. It repeats this process till there are no more affixes. If there are no more affixes, it identifies the probable stem word. If the derivation and stem pattern of the probable stem word matches, it looks for the reduced form of the stem inside the dictionary. If the patterns do not match, it searches the dictionary for closer words which are above a defined threshold value. Following that, it applies the generation rule to generate a suggestion. Finally, it ranks the suggested words. It repeats the same process till all words in the sentence are spell checked. The example shown on Fig. 4.10 demonstrates the steps explained above.

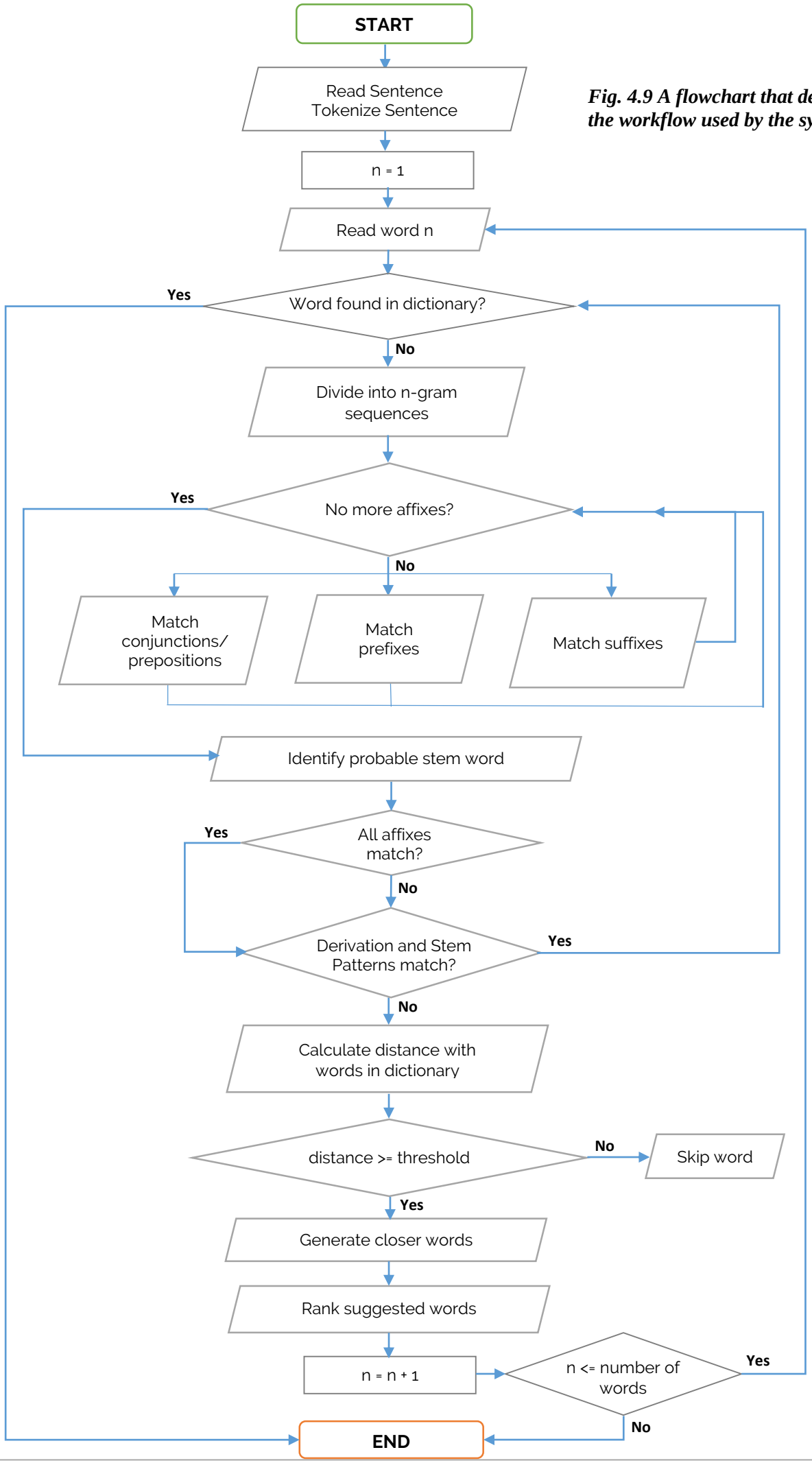


Fig. 4.9 A flowchart that depicts the workflow used by the system

Candidate word - **እንደተለመመዳችሁባቸውና**

- *Search WORD in Dictionary => Word not found*
- *Divide into n-grams*

እ	ን	ደ	ተ	ለ	መ	መ	ዳ	ች	ሁ	ባ	ቸ	ው	ና

unigrams

እን	ንደ	ደተ	ተለ	ለመ	መመ	መዳ	ዳች	ችሁ	ሁባ	ባቸ	ቸው	ውና

bigrams

እንደ	ንደተ	ደተለ	ተለመ	ለመመ	መመዳ	መዳች	ዳችሁ	ችሁባ	ሁባቸ	ባቸው	ቸውና

trigrams

- *Search matching rules*

	Pre	P4	P3	P2	P1	Stem	Derivation	S1	S2	S3	S4	S5	Conj	n
Rule 1	እንደ					A1B1C1	A1B4B1C1						ና	3
Rule 2	እንደ				ተ	A1B1C1	A1B4B1C4				ችሁ	ባቸው	ና	3
Rule 3	እንደ				ተ	A1B1C1	A1B4B1C6				ከ	ባቸው	ና	3
Rule 4	እንደ				ተ	A1B1C1	A1B4B1C6				ሽ	ላቸው	ና	3

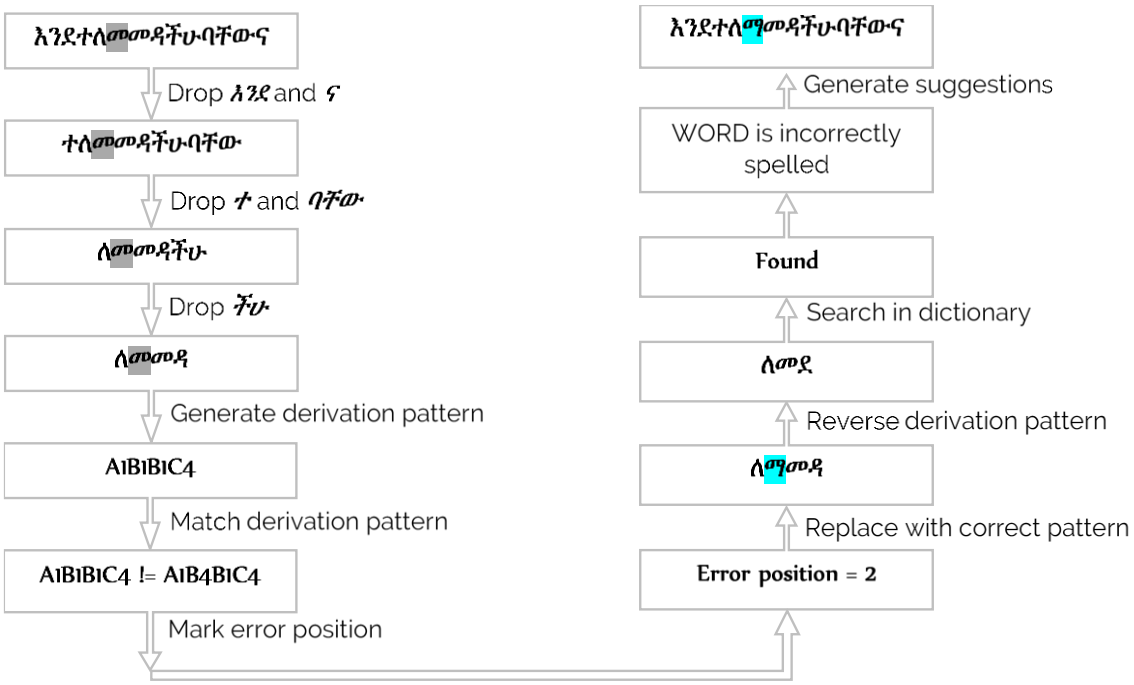


Fig. 4.10 Spell checking steps of the wrongly spelled word እንደተለመመዳችሁባቸውና

(əndätälämämädačəhubačäwəna)

CHAPTER FIVE

THE EXPERIMENT

5.1 Overview

This chapter discusses preliminary experiments performed prior to the main experiment, explanation of the preparation of test cases, development tools used, prototype used to test the system and evaluation and test results found from the main experiment. Section 5.2 discusses the findings from a preliminary experiment and an analysis made prior to the main experiment. The results found from the analysis and preliminary experiment are used as an input for the main experiment. Section 5.3 explains the prototype of the system, its components and the tools used to develop the prototype. Section 5.4 discusses the evaluation results, the findings of the experiment and summary of test results. Section 5.5 discusses the findings of the experiment by comparing the results with previous research works. Section 5.6 presents the challenges faced during the experiment and discusses the limitations occurred by them.

5.2 Preliminary Experiment and Analysis

5.2.1 Common Error Patterns in Amharic

A document randomly selected and downloaded from the web was manually checked for spelling errors. Out of 4364 words, 75 errors were identified. From the errors identified, majority (87%) of them were non-word errors, while the rest are real-word (Appendix 7). In addition, 96% of the errors were single errors i.e. the error is located only on one character of the word.

From the errors, the proportion of errors occurred on verbs and nouns is almost equal. This shows that people could make spelling mistakes equally on nouns and verbs. Analyzing the type of error shows that most of the errors (65%) are caused by substituting a character with another one. 16% of the errors are caused by deleting a character. The remaining 20% are caused by insertion, run-on, split word and substitution. Errors caused by transposition were not identified.

Interestingly, 67% of the substitution errors are caused by substituting a character with another character of the same family and the error occurred in the middle position.

Analyzing the errors caused by deletion shows that deletion is the next cause of spelling errors in Amharic and most of the cases show that the omission occurs at the end of the word.

The following table shows the number of spelling errors found, their type disaggregated by position, type of error and word class the error has occurred.

		Number of Errors
Occurrence of Error	Verbs	35
	Nouns	35
	Adjectives	3
	Others	2
Number of Errors	Single	72
	Multiple	3

Table 5.1 Results from analysis of common error patterns

Type of Error	**Total Number of Errors**	**Cause of Error**	**Location of Error**	**Number of Errors**
Non-word	65	Substitution	same family	45
		Substitution	same family, middle	33
		Substitution	same family, first	4
		Substitution	same family, last	7
		Substitution	same family, middle, last	1
		Substitution	Multiple	2
		Deletion	Middle	2
		Deletion	First	2
		Deletion	Last	7
		Insertion	-	5
		Transposition	-	0
		Mixed	-	3
Real-word	10	Run-on	-	4
		Split Word	-	3

Table 5.2 Results from analysis of common error patterns disaggregated by location & cause of errors

From the results found, this research is going to focus on non-word errors. In addition, the algorithm will first look for substitution errors in both verbs and nouns. If not found, it will move to looking for deletion errors. Then, insertion error will be considered based on the results found from the previous steps. This will help to make the algorithm more efficient.

5.2.2 Experiment - Distance Calculation Algorithms Comparison

In this experiment, seven distance similarity measure algorithms were compared for the verb ሰበረ (*säbärä*) and its inflections. The algorithms compared in this experiment are cosine similarity, Euclidean distance, manhattan distance, minkowski distance, jaccard similarity, python's sequence matcher and levenshtein distance. Table 5.3 shows the distance computed by each distance calculation algorithm for each pair of words.

	ሰበረ Vs							
	ሰበረ	**ሰበሩ**	**ሰብሮ**	**ሰብር**	**አሰበረ**	**ሰባሪው**	**የሰበሩት**	**ይሰበር**
Euclidean distance	0	1.414	1.414	1.414	1.414	1.414	1.414	1.414
Manhattan distance	0	2	2	2	2	2	2	2
Minkowski distance	0	1.26	1.26	1.26	1.26	1.26	1.26	1.26
Cosine similarity	1	0	0	0	0	0	0	0
Jaccard similarity	1	1	1	1	1	1	1	1
Python's SequenceMatcher	1	0.67	0.33	0	0.85	0	0.5	0.57
Levenshtein distance	0	1	2	3	1	4	3	2

Table 5.3 Distance computation between ሰበረ and its inflections with existing distance calculation algorithms

The results on the first column of table 5.3 shows that Euclidean distance, Manhattan distance, Minkowski distance and Levenshtein distance algorithms give a distance of zero when the words compared are the same. In the case of cosine similarity, jaccard similarity and python's SequenceMatcher, a distance value one implies that the words are the same.

The distance given by the first five algorithms is the same for the derivations/ inflections of the the verb ሰበረ. Though the derivations are closer to the stem word, the extent of closure is not the same. This shows that the Euclidean distance, Manhattan distance, Minkowski distance, cosine similarity and jaccard similarity are not suitable for word-level comparison of Amharic words. They are rather suitable to compare documents written in English or other similar languages.

The distances computed by python's SequenceMatcher has a better result than the preceding algorithms, however since it uses exact matching technique, it fails to identify the similarity with ሰባሪው and ሰብር even though these words are very close to the stem verb. On the other hand, Levenshtein distance is typical for string comparisons in language processing

applications. It works by counting the number of operations that has to be performed to change one word to the other. However, in the case of Amharic, it does not show the closure or relationship between characters in the same family and characters in the same order. The distances given for 'ሰባሪው', 'ሱበር' and 'የሰበሩት' are much higher than what is expected. This implies that SequenceMatcher and Levenshtein distance are also not fully suitable for Amharic string comparison.

Based on the results found, none of the algorithms are suitable for Amharic. Therefore, a new distance calculation algorithm is designed and implemented. The way the algorithm works is explained in Section 4.3 vi in detail. Table 5.4 shows the distance computed among the same pair of words with the new distance comparison algorithm. Note how it is able to identify closure between characters of the same family and characters in the same order.

	ሰበረ Vs							
	ሰበረ	**ሰበሩ**	**ሱበሮ**	**ሱበር**	**አሰበረ**	**ሰባሪው**	**የሰበሩት**	**ይሰበር**
Distance Calculator	1	0.9	0.6	0.433	0.6	0.4	0.62	0.7

Table 5.4 Distance computation between ሰበረ and its inflections with the distance calculator

5.3 Prototype of the System

The prototype of the system was developed to show how the system's architecture and design presented in Chapter 4 is implemented. It is developed by python programming language. The prototype uses three reference files stored in csv format. Python is selected as a development environment because python is known for its efficiency in the implementation of data processing applications, NLP being one of them. The knowledge base used by the system is stored in csv format because such formats are very light and pulling data from and saving data to them does not take much resources. Spell checker applications can benefit from these characteristics of the tools to be efficient and accurate.

The prototype works by accepting a word or list of words from the user. The latter option was implemented to make the testing easier for the researcher. The prototype has a GUI shown on fig 5.1 and 5.2. The user enters a word or a sentence and clicks on *CHECK* button. The system tokenizes the given sentence and processes each word and gives a closer suggestion for invalid words. In future works, the prototype will be integrated with word processing applications. When that happens, it is expected to have more functionalities like highlighting invalid words in a different color, replacing invalid words by suggestions given by the system, ignoring error

of words identified as invalid, adding new words to a dictionary and other functionalities as needed.

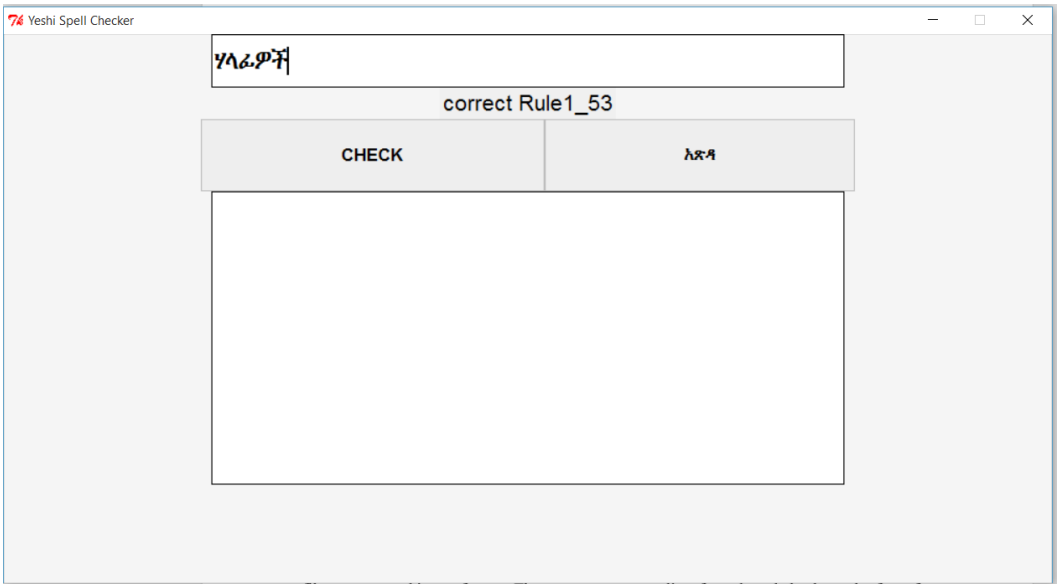


Fig. 5.1 GUI of the prototype – spell checking the valid word ሃላፊዎች *(hälafiwoč)*

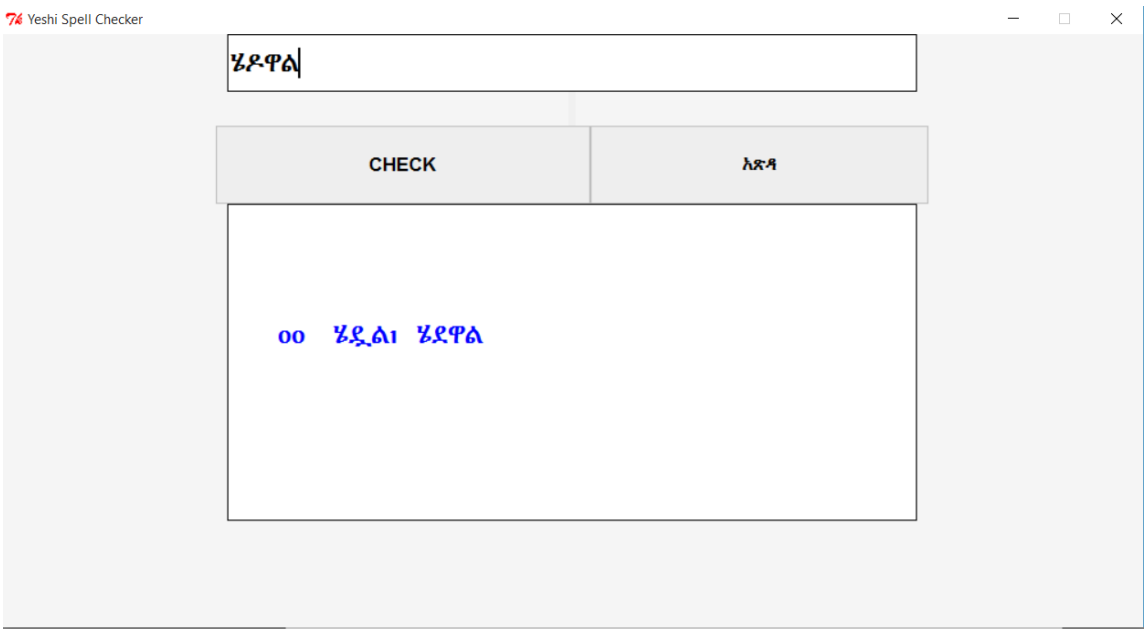


Fig. 5.2 GUI of the prototype – spell checking the invalid word ሄዶዋል *(hédowal)* ***and giving suggestions***

For testing purpose, the prototype accepts sentences from the system, tokenizes them, removes punctuation marks except the compound words concatenator '-' and saves the resulting words in separate rows in a csv file. This process is needed so that each word should be annotated as valid/invalid manually by the researcher. This is later used to evaluate the results given by the system. The system then takes this list, checks each word for spelling error and returns TRUE

if correctly flagged and FALSE if not. The results found are again saved in csv format, which is later used to calculate the performance of the system.

The prototype has seven components.

a) **Tokenizer** – This component is responsible to break a given sentence into separate words and removing punctuation marks except '-' on the way. If there are any repeat words, it removes these words and returns unique ones only.

Fig. 5.3 Tokenization of Test Case 2

b) **Search Dictionary** – searches a given word in the stored set of stem words and returns Valid if it is found. If not, it passes the word to the next component.

c) **n-gram Splitter** – splits the given word into unigrams, bigrams and trigrams and passes them to the next component.

d) **Rule Filter** – takes the first and last unigram, bigram and trigram and looks for matching rules that contain such n-grams as preposition, prefix, suffix or conjunction. Rules that contain multiple affixes are identified and processed first.

e) **Pre/Suf Checker** – the system accepts the filtered rules from the previous component and works by dropping the identified n-grams from the word and checking if the following or preceding affix matches with those on the defined rule. if it comes across an un-matching n-gram, it suspects it as an error and marks the error position, the probable correct character and the probable wrong character. It also calculates the rule weight, the number that tells the system the most likely rule to use when generating suggestions.

   Assuming there is an error, it replaces the suspected wrong character(s) with the one estimated to be correct and repeats the same procedure of comparing following and preceding character(s) till there are no more matching characters on both left and right sides. There is a lot more that this component does; it is explained in a general way.

f) **Stem Extractor** – the system generates the derivation pattern of the extracted word as explained in Section 3.2 iii. If the derivation pattern matches with the one given on the rule, the system reverses the word to its root form based on the word pattern defined on the rule. Then, it will search the word inside the dictionary. If it is found, then the word is valid, if no errors are previously suspected. If it is not found, the system keeps looking for more matching rules. If the derivation pattern does not match, the system suspects that the error might have occurred at this position and marks it as an error.

g) **Suggestion Generator and Ranker** – if the word is finally marked as invalid, the system generates closer words by going through all the error positions marked and replacing characters with the correct ones. Accordingly, it looks for closer stem words and applies the derivation pattern given by the rule. It also replaces suspected wrong prefixes and suffixes by the correct one. finally, it checks if the generated suggestion is a valid word.

The way the system works is depicted on the output taken partly from python anaconda editor on fig 5.4. It shows spell checking process for the word እያሰባሰቡን /*əyasäbasäbəŋəna*/.

```
========= CANDIDATE WORD =========
እያሰባሰብን
================================
SEARCHING IN DCITIONARY.............................
=> Word NOT Found in dictionary
DIVIDING WORD INTO N-GRAMS.............................
___________ UNIGRAMS ___________
[u'\u12a5', u'\u12eb', u'\u1230', u'\u1263', u'\u1230', u'\u1265', u'\u1295']
_____________________________________

SEARCHING FOR MATCHING RULES.............................
___________ CONJUCTION RULES ___________[]
___________ SUFFIX RULES ___________
['Rule1_471', 3, 'Rule1_323', 4, 'Rule1_338', 4, 'Rule1_394', 4, 'Rule1_399', 4,
'Rule1_120', 5, 'Rule1_131', 5, 'Rule1_146', 5, 'Rule1_170', 5, 'Rule1_175', 5,
'Rule1_185', 5, 'Rule1_198', 5, 'Rule1_200', 5, 'Rule1_219', 5, ...]
___________ PREPOSITION RULES ___________[]
___________ PREFIX RULES ___________[]
Unigrams
==========
[u'\u12a5', u'\u12eb', u'\u1230', u'\u1263', u'\u1230', u'\u1265', u'\u1295']
conjrules1       []
suffixrules1
['Rule1_471', 3, 'Rule1_323', 4, 'Rule1_338', 4, 'Rule1_394', 4, 'Rule1_399', 4,
'Rule1_120', 5, 'Rule1_131', 5, 'Rule1_146', 5, 'Rule1_170', 5, 'Rule1_175', 5,
'Rule1_185', 5, 'Rule1_198', 5, 'Rule1_200', 5, 'Rule1_219', 5 ...]
preprules1       []
prefixrules1     []
Bigrams
==========
[u'\u12a5\u12eb', u'\u12eb\u1230', u'\u1230\u1263', u'\u1263\u1230',
u'\u1230\u1265', u'\u1265\u1295']
conjrules2       []
suffixrules2
['Rule1_442', 4, 'Rule1_133', 5, 'Rule1_586', 5]
preprules2       []
prefixrules2
['Rule1_820', 1]
```

```
Trigrams
==========
[u'\u12a5\u12eb\u1230', u'\u12eb\u1230\u1263', u'\u1230\u1263\u1230',
u'\u1263\u1230\u1265', u'\u1230\u1265\u1295']
conjrules3       []
suffixrules3     []
preprules3       []
prefixrules3     []

FILTERING COMMON RULES.............................
___________ CONJUCTION RULES ___________   []
___________ SUFFIX RULES ___________
['Rule1_471', 3, 'Rule1_323', 4, 'Rule1_338', 4, 'Rule1_394', 4, 'Rule1_399', 4,
'Rule1_120', 5, 'Rule1_131', 5, 'Rule1_146', 5, 'Rule1_170', 5, 'Rule1_175', 5,
'Rule1_185', 5, 'Rule1_198', 5, 'Rule1_200', 5, 'Rule1_219', 5 ... ]
___________ PREPOSITION RULES ___________  []
___________ PREFIX RULES ___________       []
___________ COMMON RULES ___________
['Rule1_820', u'\u1295', 12, 'S', 'Rule1_820', u'\u12a5\u12eb', 1, 'R']
Closer words
[u'\u12a5\u1295\u12f0', u'\u12a5\u1295\u12f2\u1201\u121d']
PROCESSING COMMON RULES.............................
['Rule1_820', u'\u1295', 12, 'S', 'Rule1_820', u'\u12a5\u12eb', 1, 'R']
[u'\u12a5', u'\u12eb', u'\u1230', u'\u1263', u'\u1230', u'\u1265', u'\u1295']
___________ COMMON SUFFIX RULES ____________
Rule1_820
ብ can not preceed
ሰብ can not preceed
ባሰብ can not preceed
derivation
A6B4C1D4C1D6
Rule Derivation Pattern =  A1B4A1B6
Word Derivation Pattern =  A6B4C1D4C1D6
Closer words
[u'\u1230\u1260\u1230\u1260', u'\u12e8\u1265\u1235']
[u'\u12a5\u12eb', u'\u12eb\u1230', u'\u1230\u1263', u'\u1263\u1230',
u'\u1230\u1265', u'\u1265\u1295']
___________ COMMON PREFIX RULES ____________
Rule1_820
ሰ can not follow
ሰባ can not follow
ሰባሰ can not follow
Rule Derivation Pattern =  A1B4A1B6
Word Derivation Pattern =  A1B4A1B6

Spelling: correct
```

Fig. 5.4 Output of the spell checking procedures by the system

5.4 Evaluation

The system was first tested and evaluated for each class of rules and derivation types identified during test case preparation (Section 3.2v). Then, the overall performance of the system is evaluated. The system was evaluated for each group of words so that the researcher will be able to identify which type of words are challenging for the spell checker (Table 5.5 and 5.6).

	Total	**Valid**	**Invalid**	**TP**	**FN**	**TN**	**FP**
Test Case 1							
Passive Reflexive (Tederagi)	390	329	61	298	31	61	0
Transitive/ Causative (Adragi)	25	22	3	22	0	3	0
Repeated Transitive/ Causative (Aderaragi)	25	21	4	21	0	4	0
Transitive/ Causative (Asderagi)	89	75	14	71	4	11	3
Jussive Passive (Yihuntawi)	14	14	0	13	1	0	0
Negative action							
Al-, Suffix = Combined	355	338	17	332	6	17	0
Ale-, Suffix = Combined	159	156	3	150	6	2	1
Prepositions & combined suffix							
Be-, Suffix = Combined	72	53	19	48	5	17	2
Ende-, Suffix = Combined	60	50	10	48	2	5	5
Endemi-, Suffix = Combined	91	89	2	89	0	0	2
Ye-, Suffix = Combined	24	21	3	19	2	3	0
Yemi-, Suffix = Combined	111	102	9	99	3	8	1
Randomly selected	46	37	9	37	0	9	0
Nouns (Primary & Derived)	174	155	19	149	6	14	5
Test Case 2	89	89	0	89	0	0	0
	1724	**1551**	**173**	**1485**	**66**	**154**	**19**

Table 5.5 Evaluation results of Test Case One disaggregated by derivation style

	Recall	Precision	Lexical Recall	Error Recall	Lexical Precision	Error Precision	Predictive Accuracy
Test Case 1							
Passive Reflexive (Tederagi)	1	0.91	0.91	1	1	0.66	0.92
Transitive/ Causative (Adragi)	1	1	1	1	1	1	1
Repeated Transitive/ Causative (Aderaragi)	1	1	1	1	1	1	1
Transitive/ Causative (Asderagi)	0.96	0.95	0.95	0.79	0.96	0.73	0.92
Jussive Passive (Yihuntawi)	1	0.93	0.93	-	1	0	0.93
Negative action							
Al-, Suffix = Combined	1	0.98	0.98	1	1	0.74	0.98
Ale-, Suffix = Combined	0.99	0.96	0.96	0.67	0.99	0.25	0.96
Prepositions							
Be-, Suffix = Combined	0.96	0.91	0.91	0.89	0.96	0.77	0.9
Ende-, Suffix = Combined	0.91	0.96	0.96	0.5	0.91	0.71	0.88
Endemi-, Suffix = Combined	0.98	1	1	0	0.98	-	0.98
Ye-, Suffix = Combined	1	0.9	0.9	1	1	0.6	0.92
Yemi-, Suffix = Combined	0.99	0.97	0.97	0.89	0.99	0.73	0.96
Randomly selected	1	1	1	1	1	1	1
Nouns (Primary & Derived)	0.97	0.96	0.96	0.74	0.97	0.7	0.94
Test Case 2	1	1	1	-	1	-	1
	0.99	**0.96**	**0.96**	**0.89**	**0.99**	**0.7**	**0.95**

Table 5.6 Evaluation metric results of Test Case One disaggregated by derivation style

Accordingly, the results show that the system has equivalent capability of flagging valid and invalid for nouns and different groups of verbs. The system flags **95.7%** of the given valid words as valid. The challenges with the rest arises because Amharic has unique derivational rules that are exceptions to those we usually use (Section 5.6).

The system flags **89%** of the invalid words as invalid. From the given test cases, it flags **11%** of the invalid words as valid. This occurs because the system tends to follow a certain rule partly and assumes that the given word is valid (Section 5.6).

The system shows an average precision and recall of **96%** and **99%** respectively. Table 5.7 shows evaluation results of the experiment. From the figures shown on the table, Predictive Accuracy of **95%** shows that the system has highest possibility of handling any candidate word accurately, whether the word is valid or invalid. **99%** Recall shows that the system has the highest completeness to label correctly spelled words as valid. Lexical and Error Recall values of **96%** and **89%** designate the inclusiveness of the lexicon of the system [39]. It also shows the efficacy of morphological analysis performed by the spell checker [39].

Precision	**96%**
Recall	**99%**
Lexical Recall	**96%**
Error Recall	**89%**
Lexical Precision	**99%**
Error Precision	**70%**
Predictive Accuracy	**95%**

Table 5.7 Evaluation results

A precision of **96%** shows the exactness of the system i.e. the percentage of valid words from the words that are labeled as valid and the percentage of invalid words from the words that are labeled as invalid. In other words, it shows the system has a high capability to label only correct words as valid and only incorrect words as invalid [39]. Lexical and Error Precision values of **99%** and **70%** designate the ability of the spell checker to flag all invalid words [39].

The confusion matrix shown on Table 5.8 shows the following results:

True Positives (TP) = **1485** (number of correctly spelled words that were correctly identified as valid)

True Negatives (TN) = **154** (number of misspelled words that were correctly identified as invalid)

False Positives (FP) = **19** (number of misspelled words that were incorrectly identified as valid)

False Negatives (FN) = **66** (number of correctly spelled words that were incorrectly identified as invalid)

Actual class\Predicted class	*Predicted Correctly Spelled Words*	*Predicted Misspelled Words*
Actual Correctly Spelled Words	**1485**	**66**
Actual Misspelled Words	**19**	**154**
	1504	**220**
	(Total number of words predicted as valid)	(Total number of words predicted as invalid)

Table 5.8 Results of the experiment in a confusion matrix

The results on Table 5.9 show that the summary of suggestions given by the system for words correctly flagged as invalid. The table shows for how many words suggestion is provided, how many were correct and how many were incorrect, for how many words suggestion is not given at all. The result shows that the system gives the expected suggestion for 70% of the words correctly flagged as invalid.

Number of invalid words	173
Number of invalid words correctly flagged	154
Suggestion provided	122
Correct suggestion provided	109
Incorrect suggestion provided	13
Suggestion not provided	32

Table 5.9 Evaluation results for spelling corrector of the system

Some of the causes for the results achieved have been identified.

i. The system was unable to identify which rules can be applied for which words. i.e. the system may apply a rule on a stem without knowing if such derivation rule can be applied to that stem.
ii. The system makes a mistake in suspecting the position of the error.

Resolutions for the problems identified are discussed on Section 5.6.

5.5 Results and Discussion

The experimental results show value of 96% lexical recall, 89% error recall, 99% lexical precision, 70% error precision and correct suggestions were provided for 70% of the correctly flagged invalid words. This shows that the system has a high accuracy in flagging words as valid/invalid and needs some improvement in suggestion generation.

In an attempt that was made to understand Amharic spelling error patterns, it was found out that most errors occur in non-words and the majority of the errors are substitution errors. Interestingly enough, most of the substitution errors happen when one substitutes a character with another character of the same family. This could be the result of mistyping one of the keys (mostly the second which represents the vowels) after correctly typing the key that represents the consonant. This input was quite important for the development of spelling corrector.

The above fact led the researcher to examine the existing distance computing algorithms to examine if they correctly measure the similarity between two Amharic words. None of the seven algorithms were found to reflect the closeness of Amharic characters in the same family as compared to the distance algorithm developed in this research. One of the tested algorithm is Levenshtein Edit Distance used in [5] for Afaan Oromo typed with Latin. The newly designed and implemented algorithm in this research considers and implements the fact that Amharic characters are grouped as family of seven and each member in this set is given an order. Similarity between characters is measured in terms of two factors namely family and order. As far as the researcher knows, a distance computing algorithm with such consideration never exists. The accuracy of the distance computing algorithm affects the spelling corrector as it helps to generate closer suggestions.

The result of the second and major experiment on the spelling error detection extends the fact that morphology-based spelling error detection is a promising approach in achieving a better accuracy for Amharic as illustrated in [12]. Apart from defining affixes, derivation rules and stems separately, no attempt was stated to define any relationship to store what is applied to what in [12]. However, morphology rules in this study are defined in such a way that the relationship between prepositions, conjunctions, affixes, stem type and derivations are defined (Table 4.2). For example, አልሰበርክም is detected as correctly spelled word as there exists a rule that defines the pattern አል (preposition) + ሰበር (derivation of a verb) + ክም (suffix). On the other hand, አልሰባራክም is correctly detected as invalid. Even though the removal of the preposition(አል) and suffix (ክም) results in the word ሰባራ which is a meaningful Amharic noun, the word is considered as spelled incorrectly since no rule indicates the usage of the combination of these preposition and suffix with nouns. This avoids the possibility of seemingly correctly constructed words with no meaning to be considered spelled correctly.

The format of the rule dictionary is specifically designed to handle Amharic nature. It allows storing prepositions, enough number of affixes and conjunction while maintaining their

relationship between each other and stem types unlike adopting the Hunspell dictionary and affix file format as in [5]. This reflects Amharic morphology in a better manner.

Testing the spelling corrector also reveals that morphology-based technique combined with a distance computation algorithm tailored to Amharic is a promising approach towards solving the problem. Correct or expected suggestions were generated for more than half of the invalid words. The main cause of wrong suggestions was identified as picking the incorrect rule or applying to an incorrect verb type. This implies that rule definitions should further be expanded to make them more specific. One example of making rules more specific is considering the classes of verbs to relate them to the derivations and other speech parts instead of generalizing them to all verbs.

The overall experiment suggests that Amharic spelling errors can be detected and corrected as long as the morphology of the language is completely identified and clearly defined. It was also observed that effective algorithms need to be adopted or developed to be able to make the system more efficient.

The researcher observed that there are exceptions for the considered rules in this study. Those were evidently one source of error. No technique was implemented to handle the exceptions in this research.

Another limitation of this study is that rules defined for verbs are too broad. Amharic verbs' derivations are complex in nature. Such generalization results in erroneous result both in terms of detection and correction. Rules could be more specific and can be defined for specific classes of verbs.

The researcher also observed that when the number of rules is increasing, the system finds it difficult to pick the probable appropriate rules to reverse a word. This is because of a lot of ambiguities and similarities between the rules. This is open for improvement. Effort can be made to (1) make the rule definition more precise, (2) create a systematic way of reducing the number of rules or (3) improving the rule picker algorithm.

5.6 Challenges and Limitations

One of the challenges of the system is exceptions to the trends in Amharic derivations as mentioned in Section 5.4. The challenges to flag some of the valid words as invalid arises because Amharic has unique derivational rules that are exceptions to those we usually use. For example, to inflect the verb ከሰረ to show the subject 'I' as the doer of the action, we change the orders of the last two characters to get ከሰሬ. Most words follow the same rule of changing the order of their last or two last characters. But, there is an exception for verbs ending in ጠ, ደ, ነ and ለ. For example, applying the same derivation to the verbs 'ሸጠ', 'ሄደ', 'ለመነ', 'ሸለለ' results in 'ሸጬ', 'ሄጀ', 'ለምኜ', 'ሸልዬ' respectively. As can be seen, these words tend to change their last character to a different character in addition to the order. These and other exceptions in the language are the causes of the detection mistakes caused by the system. This can be resolved by adding such exceptions specifically to rule definitions and adding a component that scans given words for such exceptions. Another cause for flagging valid words as invalid occurs when the system could not find the resulting stem inside the dictionary of words, yet the word is a valid word. This can be resolved when the stored stem words are complete.

The other challenge is partly applying rules to words. The cause of this is due to the way rules are defined for the system. The system tends to follow a certain rule partly and if the given word fulfills the derivation and prefix/suffix given by the rule partly, it flags it as valid. For example, there is a rule that changes a noun to its plural form by changing the order of the last character of the noun to seventh order and adding the suffix 'ች'. Accordingly, when the word ህዝቦ is given to system, it only applies the first action (i.e. changing the order of the last character) and assumes that the word is valid. This can be solved by modifying the way Prefix/Suffix checker component works.

The third challenge is in generating suggestions. This can be resolved by including the word category to rule definitions. A second solution is improving the way a rule weight is calculated for each rule, so that the system would know which rules to focus on when generating suggestions.

CHAPTER SIX

CONCLUSION AND RECOMMENDATION

6.1 Overview

This chapter presents the conclusion and recommendation acquired from the evaluation results. Section 6.2 discusses the findings and the conclusion determined from the research findings. Section 6.3 discusses recommendations for future researchers who are interested to precede in the same or related research area.

6.2 Conclusion

Spell checker is an application that can be integrated with word processing applications, detects spelling errors and gives closer suggestions to invalid words with errors. In this research, an attempt was made to apply morphology-based approach for the implementation of Amharic spell checker. The system uses manually defined rules and was tested on words selected from a corpus collected from the web, books and newspapers.

The Amharic spell checker was implemented using python programming language. The system uses manually defined rules along with sets of stored stem words and Amharic characters organized in tabular format (Appendix 4). To predict the validity of a word, it divides the word into n-grams and finds matching preposition, prefix, suffix and conjunctions that match the defined rules. It also matches the derivation pattern of the stem word.

The system was evaluated by comparing the results found with manually annotated test cases. The performance of the system was evaluated with three common measures called precision (lexical and error precision), recall (lexical and error recall) and Predictive Accuracy. The performance shows 96% precision, 99% recall, 96% lexical recall, 89% error recall, 99% lexical precision, 70% error precision and 95% predictive accuracy. Out of the words correctly flagged as invalid, correct suggestions were given for 70% of them.

Based on the results of the study, the following conclusions are made:

- From the evaluation results found from the experiment, it was shown that morphology-based approach is capable of detecting and correcting errors. A predictive accuracy of

95% shows that the system is able to detect and correct spelling errors as long as it finds the correct rule definitions and the corresponding stem word inside the dictionary.

The experimental results also show that the system is able to cover detection and correction of complex derivation of verbs, which is the most complex morphology in Amharic [12]. It is capable of analyzing words with multiple prefixes and suffixes plus prepositions and conjunctions. The system is also tested on derived nouns and it was able to analyze and spell check those as well. This implies that it is able to spell check most of the complex words in Amharic. A system capable of handling complex words would handle simple derivations on other parts of speech (adjectives, adverbs) and compound-nouns. From this, the researcher concluded that the system is able to spell check majority of words in the language.

- The experiment made from test cases prepared from diverse classes of verb and noun derivations shows that the system is able to handle broad and complex categories of words in the language. The spell checking approaches discussed in section 2.1.3 have a problem of handling the complex morphology of Amharic. On the other hand, morphology-based approach in combination with the proposed algorithm and the distance calculator discussed in Chapter 4 is able to breakdown the complex morphology of words and reverse them to their stem form. Once, the stem is extracted, what is remaining is searching the extracted stem inside the stored dictionary of words. In addition, to avoid the problem of reduced performance in morphology-based spell checkers, several methods are proposed. One is a system to filter rules by n-grams located at the beginning and end of the candidate word is proposed, hence makes the system efficient. Moreover, rules are filtered by the part of speech they belong to and the number of characters in the stem. This improves the efficiency of the dictionary searcher component to search words with the specified POS and number of characters. For example, if the POS given for a rule is ‘verb’ and if the number of characters is 3, then the system will search for three letter verbs inside the dictionary once the stem extraction is completed.

  Besides the distance calculator proposed plays a great role in making the system more accurate. It does so by showing the closure of words which contain characters which are in the same family and in the same order. For instance, if a person mistakenly writes ሰባረ for ሰበረ, Levenshtein edit distance would only tell that one character should be substituted by another to give the intended word. However, the distance calculator

would show that the distance between the words is even closer, because it can recognize that the characters ባ and በ are in the same family. The spell checker would predict that the error is caused by replacing a character with another one from the same family (which is a common error pattern in Amharic as taken from the analysis made on common spelling error patterns) and corrects the error and generates suggestion accordingly. The distance calculator also makes the suggestion ranker more efficient by computing the accurate distance between the candidate word and generated suggestions.

Furthermore, the rule filter can work on multiple n-grams to filter most probable rules at the same time. It receives unigrams, bigrams and trigrams from the n-gram splitter and finds rules with the corresponding affixes. When it does that, it works by searching for rules which contain affixes from all sets of n-grams. For instance, if the word እየበላሁ is given, the rule filter searches for a rule with prefix እየ and suffix ሁ and finds the most likely rule. This also avoids the performance problem that arises in analyzing the affixes of a word.

In conclusion, morphology-based approach integrated with the proposed architecture and algorithms is more suitable for Amharic spelling error detection and correction. Applying the recommendations on Section 6.3 would make the approach even more suitable.

In conclusion, the study shows that using morphology-based approach for detection and correction of Amharic spelling errors can work very well. More improvements are needed on the implementation of morphology-based approach for correction of invalid words.

6.3 Recommendation

This study can be improved in future works. The following are recommended for future studies:

1. The most complex and common morphology is that of verbs and priority is given to define rules for common and complex noun and verb derivations. Due to time constraint, the remaining morphology patterns for other POS are not defined. Therefore, improving the definition of rules to include the possibly uncovered morphology of Amharic is one of the works that needs to be considered in future studies.

2. Designing a technique to define morphology based on Amharic verb classes as verbs exhibit the most complex derivations in Amharic. This will highly improve the performance of the spellchecker.
3. Relationship between morphology and verbs should also be defined by further considering the nature of verbs. For example, ደመርኩ is a derivation of ደመረ. On the other hand, the word ደበርኩ from ደበረ is either uncommon or meaningless and should especially be excluded from suggestion list. ደመረ and ደበረ are verbs from the same category given their number of consonants and their order. For future works, it is recommended to closely study Amharic verbs to determine and understand such discrepancy when defining morphology of the language.
4. Identifying and handling exceptions that deviate from the defined morphology rules can be also be a future work. Incorporating such exception would greatly enhance the performance of the spellchecker.
5. A future research could be made to determine the possibility of partially storing common derivations along with the root words in the dictionary to reduce the number of rules and improve the accuracy and performance of the spell checker. The effect of storing a certain part of the derived words in the dictionary in an attempt to reduce the number of rules and as a result to improve the performance of the spell checker could be studied.
6. Due to time constraint, the researcher was not able to include additional functionalities of a spell checker and integrate the prototype with word processing applications. In future works, the spell checker should be able to let the user store new words inside a dictionary, ignore some kinds of words and words written incorrectly on purpose and automatically correct misspellings. It should also be integrated with word processing application.
7. This study mainly focuses on non-word errors. It can be expanded to include detection and correction of real-word errors by applying context checking.
8. The research can be expanded to Amharic-English translation systems. The architecture and design in this research can be applied to design Amharic-English translation systems. Derived words which have the same morphological pattern would follow the same translation pattern. This can be applied to word and sentence translation. For instance, the derived noun በሬዎች shows the plural form of በሬ and it means ‘oxen’. Similarly, ጠረጴዛዎች can be translated as ‘tables’ by copying the translation pattern from the previous word. The same holds true for sentence translation. For example, ‘እየበላሁ ነው።።’ can be translated to English as ‘I am eating’. Similarly, a sentence which contains the same morphological

rule like ‘እየሰማሁ ነው።።’ can be translated by the system as ‘I am listening’. Further studies can be carried on to investigate the capability of application of morphology in Amharic-English translations.

9. To avoid the drawbacks of morphology-based approach, future researchers can consider applying other approaches in combination with morphology-based approach. To the best of the researcher’s knowledge, there are no research works that use morphology-based approach in hybrid with other approaches for detection of spelling errors. In chapter 2, a work that attempts to use a hybrid approach is reviewed [16]. However, this work makes use of metaphone algorithm for detection and edit distance for correction of misspelled words, not morphology-based approach.
10. Amharic is by nature morphologically complex. In this study, the researcher was not able to find a linguistic expert that can assist in understanding the morphology of the language and defining rules. However, it is advisable to involve linguistic experts in the study, especially when building a final product that can be usable by end users.

REFERENCES

[1] "Communication | Definition of Communication by Oxford Dictionary on Lexico.com also meaning of Communication", *Lexico Dictionaries | English, 2020*. [Online]. Available: https://www.lexico.com/en/definition/communication. [Accessed: 17- Sep- 2020].

[2] M. Kore and V. Goyal, "Machine Transliteration for English to Amharic Proper Nouns", *International Journal of Computer Science Trends and Technology (IJCST)*, vol. 5, no. 4, pp. 23-31, 2017.

[3] "Language." [Online]. Available: ethiopiantreatures.co.uk/pages/language.htm.

[4] "Amharic", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Amharic. [Accessed: 04- Jan- 2018].

[5] G. Olani and D. Midekso, "Design And Implementation Of Morphology Based Spell Checker", *International Journal of Scientific & Technology*, vol. 3, no. 12, pp. 118-125, 2014.

[6] Al-Jefri et al., "Arabic Spell Checking Technique." May 2015.

[7] H. Liang Lorraine, "Spell Checkers and Correctors - A Unified Treatment." Nov. 2008.

[8] M. Abate and Y. Assabie, "Development of Amharic Morphological Analyzer Using Memory-Based Learning", *Springer International Publishing Switzerland*, pp. 1-13, 2014.

[9] M. Yifiru, " Morphology-Based Language Modeling for Amharic", *Springer International Publishing Switzerland*, Aug. 2010.

[10] R. Kaur and J. Singh, "Spelling Error Detection and Correction System for Punjabi Unicode using Hybrid Approach", *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 5, no. 9, pp. 15002 – 15007, 2017.

[11] N. Gupta and P. Mathur, "Spell Checking Techniques in NLP: A Survey", *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 2, no. 12, pp. 217-221, 2012.

[12] M. Tesfaye, "Application of morphological information for Amharic spelling error detection.", Feb. 2019

[13] A. Mekonnen, “Development of an Amharic Spelling Corrector for Tolerant-Retrieval.” 2012. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2457276.2457281.

[14] F. B. Tafesse, “Morphology Based Spell Checker for Kafi Noonoo Language.” Oct. 2018. [Online]. Available: http://etd.aau.edu.et/handle/123456789/19584.

[15] D. Jurafsky and J. H. Martin, " Spelling Correction and the Noisy Channel", 2016. [Online]. Available: https://web.stanford.edu/~jurafsky/slp3/B.pdf.

[16] G. Assefa, "Automatic Amharic spelling error detection and correction using hybrid approach", *Journal of Emerging Technologies and Innovative Research (JETIR)*, vol. 5, no. 6, pp. 605 – 611, 2018.

[17] *ኣማርኛ መዝገበ ቃላት*. አዲስ አበባ: የኢትዮጵያ ቋንቋዎች ጥናትና ምርምር ማዕከል፣ አዲስ አበባ ዩኒቨርሲቲ ማተሚያ ቤት, 2009.

[18] W. B. Demilie, “Multilingual Spelling Checker for Selected Ethiopian Languages,” *Int. J. Adv. Sci. Technol.*, vol. 29, no. 7, p. 8, 2020.

[19] A. Mekonnen, A. Nürnberger and B.Ephrem, "Portable Spelling Corrector for a Less-Resourced Language: Amharic", *International Journal of Computer Science Trends and Technology (IJCST)*, 4127-4132, 2018.

[20] M. Tilahun, “Automatic spelling checker for Amharic language.” May 2020. [Online]. Available: http://hdl.handle.net/123456789/10846

[21] M. Gebreegziabher and L. Besacier, "Preliminary Experiments on English-Amharic Statistical Machine Translation", SLTU, 36-41, 2012.

[22] W. Mulugeta and M. Gasser, “Learning Morphological Rules for Amharic Verbs Using Inductive Logic Programming”, *Workshop on Language Technology for Normalisation of Less-Resourced Languages (SALTMIL8/AfLaT2012)*, pp. 7-12, 2012.

[23] F. Tariku, *ኣማርኛ ሁሉ ለሁሉ*. ZA Publishers, 2009.

[24] B. Yimam, *የኣማርኛ ሰዋሰው*, 2nd ed. Eleni PLC, 2008.

[25] B. Yimam, “Root Reductions and Extensions in Amharic,” *Ethiop. J. Lang. Lit.*, pp. 56–88, 1999.

[26] A. M. Gezmu, B. E. Seyoum, M. Gasser, and A. Nürnberger, “Contemporary Amharic Corpus: Automatically Morpho-Syntactically Tagged Amharic Corpus,” *Proceedings of the First Workshop on Linguistic Resources for Natural Language Processing*, pp. 65–70, Aug. 2018.

[27] M. Bender Lionel, “The Origin of Amharic.” pp. 41 -52.

[28] S. Yimam Muhie, "A Brief Analysis of Amharic NLP: From POS Tagging to Question Answering". Sep. 06, 2016.

[29] A. Aklilu, *Amharic English Dictionary*, 2nd ed. Kuraz Publishing Agency, 1987

[30] B. Hamza, Y. Abdellah, G. Hicham, and B. Mostafa, “For an Independent Spell-Checking System from the Arabic Language Vocabulary,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 5, no. 1, 2014, doi: 10.14569/IJACSA.2014.050115.

[31] M. Gasser, “HornMorpho: a system for morphological processing of Amharic, Oromo, and Tigrinya.” *Conference on Human Language Technology for Development, Alexandria, Egypt*, pp. 94 - 99, May 2011.

[32] K. Peffers, T. Tuunanen, M. Rothenberger A., and S. Chatterjee, “A Design Science Research Methodology for Information Systems Research.” *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45-78, 2007.

[33] B. Gebremichael, “Wordlist and Spell checking for Amharic and Tigrigna.” [Online]. Available: http://www.cs.ru.nl/~biniam/geez/crawl.php.

[34] “A General Approach to Preprocessing Text Data,” *https://www.kdnuggets.com/2017/12/general-approach-preprocessing-text-data.html&hl=en-ET*. Jan. 12, 2020.

[35] “Sampling.” [Online]. Available: http://www.stat.yale.edu/Courses/1997-98/101/sample.htm.

[36] “Simple random sampling.” [Online]. Available: http://dissertation.laerd.com/simple-random-sampling.php.

[37] “How to choose a sample size (for the statistically challenged).” [Online]. Available: http://www.tools4dev.org/resources/how-to-choose-a-sample-size/.

[38] M. Beyene, “Data Analytics,” 2018.

[39] G. B. van Huyssteen, R. Eiselen, and M. Puttkammer, “Evaluating Evaluation Metrics for Spelling Checker Evaluations.” [Online]. Available: http://www.puk.ac.za/opencms/export/PUK/html/fakulteite/lettere/ctext/Article.RES.VanHuyssteenxEiselenxPuttkammer2004.9.9.9.GBVH.2008-09-13.FinalxSmallx.pdf.

[40] J. E. Pentheroudakis, D. G. Bradlee and S. S. Knoll, “Tokenizer for a Natural Language Processing System.”, Aug. 15, 2006.

[41] “n-gram.” [Online]. Available: https://en.wikipedia.org/wiki/N-gram.

APPENDICES

Appendix 1 – Sample Rule Definitions

Rule Name	Pre	P4	P3	P2	P1	Stem	Derivation	S1	S2	S3	S4	S5	Conj	n	Verb/ Noun
Rule1_1	የ				መ	A1B4	A6B4				ት	ን	ም	2	Verb
Rule1_2	የ				መ	A1B4	A6B4				ቱ	ን	ም	2	Verb
Rule1_3	የ				መ	A1B4	A6B4				ታችን	ን	ም	2	Verb
Rule1_4	የ				መ	A1B4	A6B4				ህ	ን	ም	2	Verb
Rule1_5	የ				መ	A1B4	A6B4				ሽ	ን	ም	2	Verb
Rule1_6	የ				መ	A1B4	A6B4				ታችሁ	ን	ም	2	Verb
Rule1_7	የ				መ	A1B4	A6B4				ቱ	ን	ም	2	Verb
Rule1_8	የ				መ	A1B4	A6B4				ቷ	ን	ም	2	Verb
Rule1_9	የ				መ	A1B4	A6B4				ታቸው	ን	ም	2	Verb
Rule1_10	የ				መ	A1B1C1	A1B1C6				NULL	ን	ም	3	Verb
Rule1_11	የ				መ	A1B1C1	A1B1C5				NULL	ን	ም	3	Verb
Rule1_12	የ				መ	A1B1C1	A1B1C4				ችን	ን	ም	3	Verb
Rule1_13	የ				መ	A1B1C1	A1B1C6				ህ	ን	ም	3	Verb
Rule1_14	የ				መ	A1B1C1	A1B1C6				ሽ	ን	ም	3	Verb
Rule1_15	የ				መ	A1B1C1	A1B1C4				ችሁ	ን	ም	3	Verb
Rule1_16	የ				መ	A1B1C1	A1B1C2				NULL	ን	ም	3	Verb
Rule1_17	የ				መ	A1B1C1	A1B1C8				NULL	ን	ም	3	Verb
Rule1_18	የ				መ	A1B1C1	A1B1C4				ቸው	ን	ም	3	Verb
Rule1_19	የ			ለ	መ	A1B1C1	A1B1C6						ና	3	Verb
Rule1_20	የ			ለ	መ	A1B1C1	A1B1C2						ና	3	Verb

Appendix 2 – Sample Stem Words categorized by type

BI-RADICALS VERBS

ኑዳ	TWO_1
ገራ	TWO_1
ቀጣ	TWO_1
ረካ	TWO_1
ከካ	TWO_2
ቀባ	TWO_2
ሰካ	TWO_2
መሸ	TWO_3
ረጨ	TWO_3
ፈጀ	TWO_3
ፈጨ	TWO_3
አሸ	TWO_4
አለ	TWO_4
ጠጀ	TWO_4
ታየ	TWO_5
ጋየ	TWO_5
ዋሸ	TWO_5
ዋጠ	TWO_6
ዳጠ	TWO_6
ሾቀ	TWO_8
ሾረ	TWO_8
ሞቀ	TWO_8
ሮጠ	TWO_8
ኮራ	TWO_9
ጎፋ	TWO_9
ቆጨ	TWO_10
ጎሸ	TWO_10
ተወ	TWO_11
ቸረ	TWO_11
ቸከ	TWO_11
ጀለ	TWO_11
ጨሰ	TWO_11
ሸሸ	TWO_12
ሄደ	TWO_13
ጤሰ	TWO_13

TRI-RADICALS VERBS

ጨገገ	THREE_1
ገደደ	THREE_1
ነፈገ	THREE_1
ጠለፈ	THREE_1
ከሰረ	THREE_1
ሠቀቀ	THREE_2
በገረ	THREE_2
ደለመ	THREE_2
ቃቃረ	THREE_2
ሰለቸ	THREE_3
መነጨ	THREE_3
ገበየ	THREE_3
ፈነጨ	THREE_3
ጋፈጠ	THREE_4
ፈነዳ	THREE_4
አመራ	THREE_4
አነጋ	THREE_4
አነባ	THREE_4
ጋመረ	THREE_5
ታመቀ	THREE_5
ጎደለ	THREE_6
ጎበጠ	THREE_6
ቆሰለ	THREE_6
ቆነነ	THREE_7
ቆዘረ	THREE_7
ሞጨጨ	THREE_7
ጎፈየ	THREE_8
ቆረነ	THREE_8
ጎደራ	THREE_9
ቆረባ	THREE_9

QUADRI-RADICALS VERBS

ገረደፈ	FOUR_1
መረቀነ	FOUR_1
ሸበለቀ	FOUR_1
ነዳደፈ	FOUR_2
አሻመደ	FOUR_2
ጎደፈረ	FOUR_3
ቆነጸለ	FOUR_3
ጎረመደ	FOUR_3
ቆረፈደ	FOUR_3
ቦተረፈ	FOUR_3
ዘበዘበ	FOUR_4
ነፈነፈ	FOUR_4
ጨቀጨቀ	FOUR_4
ወላወለ	FOUR_5
ወላወለ	FOUR_5
ሞቀሞቀ	FOUR_6
ሞጨሞጨ	FOUR_6
ቆለቆለ	FOUR_6
ቆረቆረ	FOUR_6
ቆሰቆሰ	FOUR_6
መረኮዘ	FOUR_7
ጨመደደ	FOUR_7
በለጠጠ	FOUR_7
ሞለገገ	FOUR_8
ቆለመመ	FOUR_8
ሆመጠጠ	FOUR_8
ቀረቀበ	FOUR_9
ቸረቸመ	FOUR_9
ገረገደ	FOUR_9
ከረከመ	FOUR_9
ከረከሰ	FOUR_9
ዶለዶመ	FOUR_10
ጎረጎጨ	FOUR_10
ጎረጎደ	FOUR_10
ኮረኮመ	FOUR_10
ፈለሰፈ	FOUR_11
ሸበረቀ	FOUR_11
ደነቆለ	FOUR_12
ሸረሞጠ	FOUR_12
ሸረቆጠ	FOUR_12

NOUNS

ኢኮኖሚስት
ወስከምባይ
ቸብቸቦ
ታጥእ
ሻኛ
ፈለክ
መልማይ
ብር
አሸሙረኛ
ልደታ
ብርሃን
ዘማዊ
ልምላሜ
ፀፀት
ጀምበር
ጥበብ
ኣደባባይ
ህዋእ
ኮሚሽነር
እንትና
ባለታሪክ
መጫወቻ
ጨዋታ
ውታፍ
ዶቃ
ሠራተኛ
ማህፀን
ቦረንትቻ
ኣኮፋዳ
ሸንኮሬ
ልጓጥ
ትጋት
ሪፐብሊክ
ጊንጥ
አሰት
ኣንቂ
ቢጃማ
ጆሮ
ፍርሃት
መድረሻ
ጥበቅና

ADJECTIVES

ደጋሚ
አሲዳም
ድቃቂ
ኣሻሚ
ትግስተኛ
ጡንቻማ
ኣንዳች
ፈቃጅ
ባለጌ
ዋዘኛ
አራዊት
ችግረኛ
አንጣሪ
ኣሰልቺ
ተብታባ
ኩሩ
አምልኮኛ
ኮምታራ
እንከፍ
ጨብራራ
ኣሜተኛ
ሰይጣን
ፈረስ
ኣቅኚ
ጀብራራ
ሸንጋይ
ኮትኳች
ጀማሪ
ኃለኛው
ቃራሚ
ቂመኛ
ጥርሰ
ፍንጭት
ልስልስ
ልክፍታም
ብላታ
ሰራፂ
ጠንሳሽ
እብድ
ልከኛ
ሰፍሳፋ
ፉንጋ

ADVERBS

ወዲያው
ስንኳ
ላመል
እንጃባቱ
እንግዲህ
እንቢዮ
አንድጋ
በገሀድ
በኣጋጣሚ
እንቢ
ደግነቱ
በተለምዶ
ጊዜ
እንደኔ
እንግዲያስ
በጸጥታ
ከዚህ
እንዲሁም
እጅግ

Appendix 3 – Test Case 1 Sample Words

Words	Classification	Derivation Category
ተሰርቶለታል	correct	ተደራጊ
ተሰርቶለት	correct	ተደራጊ
ተሰርቶላቸው	correct	ተደራጊ
ተሰርቶበታል	correct	ተደራጊ
ተሰርቶባቸው	correct	ተደራጊ
ተሰርቷል	correct	ተደራጊ
ተሰትጥቷል	incorrect	ተደራጊ
ተሰቶአቸው	incorrect	ተደራጊ
አስረዝመው	correct	አስደራጊ
አስረዝማ	correct	አስደራጊ
አስረዝሞ	correct	አስደራጊ
አስረዝናል	incorrect	አስደራጊ
አስረደተዋል	incorrect	አስደራጊ
አስረደተው	incorrect	አስደራጊ
አስረዱ	correct	አስደራጊ
አስረዱት	correct	አስደራጊ
አስረዱን	correct	አስደራጊ
ይሸለማሉ	correct	ይሁንታዊ
ይሸነፋሉ	correct	ይሁንታዊ
ይሸለማሉ	incorrect	ይሁንታዊ
ይፈልጋሉ	correct	ይሁንታዊ
ይፈራጃሉ	incorrect	ይሁንታዊ
ይፈርማሉ	correct	ይሁንታዊ
ይፈርሳሉ	correct	ይሁንታዊ
ይመደብላቸዋል	correct	ይሁንታዊ
ይተረክላቸዋል	correct	ይሁንታዊ
ይነገርላቸዋል	correct	ይሁንታዊ
ይከበርላቸዋል	correct	ይሁንታዊ
ይጠበቅልን	correct	ይሁንታዊ
ይጨመርልን	correct	ይሁንታዊ
አሰባሰበ	correct	አደራራጊ
አሰባሰቡ	correct	አደራራጊ
አሰባሰቡን	correct	አደራራጊ
አሰባሰባቸው	correct	አደራራጊ
አሰባሰባችሁ	correct	አደራራጊ
አሰባሰብሁ	incorrect	አደራራጊ
አሰባሰብን	correct	አደራራጊ
አሰባሰብንና	correct	አደራራጊ
አሰባሰብኩ	correct	አደራራጊ
አሰልጥነዋል	correct	አድራጊ
አሰልጥነው	correct	አድራጊ
አሰልጥኑልን	correct	አድራጊ
አሰልጥናለች	correct	አድራጊ
አሰልጥናል	incorrect	አድራጊ
አሰልፈን	correct	አድራጊ
አሰልፈዋል	correct	አድራጊ
አሰልፈዋታል	correct	አድራጊ
አሰልፈው	correct	አድራጊ
አሰልፈውበናል	correct	አድራጊ

Appendix 4 – Test Case 2 Sample Unique Words

No	Classification	Words
0	correct	ፍሬን
1	correct	ፈጠረ
2	correct	ጨለማውንም
3	correct	ጨለማን
4	correct	ጨለማም
5	correct	ጥዋትም
6	correct	ጠፈርን
7	correct	ጠፈር
8	correct	ጠራው
9	correct	ይገለጥ
10	correct	ይከፈል
11	correct	ይሰብሰብ
12	correct	ይሁን
13	correct	ያሉትንም
14	correct	ያለው
15	correct	ያለበትን
16	correct	ያ
17	correct	የውኃ
18	correct	የእግዚአብሔርም
19	correct	የብሱን
20	correct	የብሱም
21	correct	የሚያፈራ
22	correct	የሚሰጥ
23	correct	ዛፍን
24	correct	ዘርን
25	correct	ዘሩም
26	correct	ዘሩ
27	correct	ውኖች
28	correct	ውኃ
29	correct	ወገኑ
..	... ...	...
59	correct	በውኃና
60	correct	በውኃ
61	correct	በአንድ
62	correct	በታችና
63	correct	በታች
64	correct	በምድርም
65	correct	በመጀመሪያ
66	correct	በላይ
67	correct	ቀን
68	correct	ስፍራ
69	correct	ሰፍፎ
70	correct	ሰማይንና
71	correct	ሰማይ
72	correct	ሦስተኛ
73	correct	ሣርንና
74	correct	ምድርን
75	correct	ምድርም
76	correct	ምድር
77	correct	ማታም
78	correct	መካከልም
79	correct	መካከል
80	correct	መከማቻውንም

Appendix 5 – Amharic Characters Set

ha	le	hmha	me	sse	re	se	she	qe	be	te	che	hha	ne	gne	aa	ke	hhe	we	oa	ze	zze	ye	de	je	ge	tte	cche	ppe	tse	otse	fe	pe
ሀ	ለ	ሐ	መ	ሠ	ረ	ሰ	ሸ	ቀ	በ	ተ	ቸ	ኀ	ነ	ኘ	አ	ከ	ኸ	ወ	ዐ	ዘ	ዠ	የ	ደ	ጀ	ገ	ጠ	ጨ	ጰ	ጸ	ፀ	ፈ	ፐ
ሁ	ሉ	ሑ	ሙ	ሡ	ሩ	ሱ	ሹ	ቁ	ቡ	ቱ	ቹ	ኁ	ኑ	ኙ	ኡ	ኩ	ኹ	ዉ	ዑ	ዙ	ዡ	ዩ	ዱ	ጁ	ጉ	ጡ	ጩ	ጱ	ጹ	ፁ	ፉ	ፑ
ሂ	ሊ	ሒ	ሚ	ሢ	ሪ	ሲ	ሺ	ቂ	ቢ	ቲ	ቺ	ኂ	ኒ	ኚ	ኢ	ኪ	ኺ	ዊ	ዒ	ዚ	ዢ	ዪ	ዲ	ጂ	ጊ	ጢ	ጪ	ጲ	ጺ	ፂ	ፊ	ፒ
ሃ	ላ	ሓ	ማ	ሣ	ራ	ሳ	ሻ	ቃ	ባ	ታ	ቻ	ኃ	ና	ኛ	አ	ካ	ኻ	ዋ	ዓ	ዛ	ዣ	ያ	ዳ	ጃ	ጋ	ጣ	ጫ	ጳ	ጻ	ፃ	ፋ	ፓ
ሄ	ሌ	ሔ	ሜ	ሤ	ሬ	ሴ	ሼ	ቄ	ቤ	ቴ	ቼ	ኄ	ኔ	ኜ	ኤ	ኬ	ኼ	ዌ	ዔ	ዜ	ዤ	ዬ	ዴ	ጄ	ጌ	ጤ	ጬ	ጴ	ጼ	ፄ	ፌ	ፔ
ህ	ል	ሕ	ም	ሥ	ር	ስ	ሽ	ቅ	ብ	ት	ች	ኅ	ን	ኝ	እ	ክ	ኽ	ው	ዕ	ዝ	ዥ	ይ	ድ	ጅ	ግ	ጥ	ጭ	ጵ	ጽ	ፅ	ፍ	ፕ
ሆ	ሎ	ሖ	ሞ	ሦ	ሮ	ሶ	ሾ	ቆ	ቦ	ቶ	ቾ	ኆ	ኖ	ኞ	ኦ	ኮ	ኾ	ዎ	ዖ	ዞ	ዦ	ዮ	ዶ	ጆ	ጎ	ጦ	ጮ	ጶ	ጾ	ፆ	ፎ	ፖ
ኋ	ኋ	ኋ	ሟ	ሧ	ሯ	ሷ	ሿ	ቋ	ቧ	ቷ	ቿ	ኋ	ኗ	ኟ	0	ኳ	ኋ	0	0	ዟ	ዧ	0	ዷ	ጇ	ዣ	ጧ	ቋ	0	0	0	ፏ	ፗ

Appendix 6 – Sample derivations of the root verb ል-ም-ን (l-m-n)

ልለመናቸው	ለመነኝ	ልትለምነን
እንለመናቸው	ለመነችኝ	እለምንሃለሁ
ይለመናቸው	ለመኑኝ	እንለምንሃለን
ትለመናቸው	እንለምናለን	ይለምንሃል
ይለመኗቸው	ትለምናለህ	ትለምንሃለች
ተለመነኝ	ትለምኛለሽ	ይለምኑሃል
ተለመኚኝ	ትለምናላችሁ	እለምንሻለሁ
ተለመኑኝ	ይለምናል	እናስለምናታለን
ተለመነን	ትለምናለች	ታስለምናታለህ
ተለመኚን	ይለምናሉ	ታስለምኚያታለሽ
ተለመኑን	ለምኜ	ታስለምኗታላችሁ
ተለመነው	ለምነን	ያስለምናታል
ተለመኚው	ለምነህ	ታስለምናታለች
ተለመኑት	ለምነሽ	ያስለምኗታል
መለመን	ለምናችሁ	አስለምናቸዋለሁ
ለመንኩ	ለምኖ	እናስለምናቸዋለን
ለመንን	ለምና	ታስለምናቸዋለህ
ለመንክ	ለምነው	ታስለምኚያቸዋለሽ
ለመንሽ	ልትለምኑኝ	ታስለምኗቸዋላችሁ
ለመናችሁ	ሊለምነኝ	ያስለምናቸዋል
ለመነ	ልትለምነኝ	ታስለምናቸዋለች
ለመነች	ሊለምኑኝ	ያስለምኗቸዋል
ለመኑ	ልትለምነን	
ለመንከኝ	ልትለምኚን	
ለመንሽኝ	ልትለምኑን	
ለመናችሁኝ	ሊለምነን	

Appendix 7 – Misspelled words identified in common error patterns analysis in Amharic

Misspelled word	Type of error	Family of error	Location of error	Single/ Multiple Error	Classification of error
በቀርቡ	Substitution	Same family	Middle	Single	Non-word
ለማውጣ	Deletion		Last	Single	Non-word
እጀ ማራለሁ	Substitution, Split word	Same family	Middle	Single	Non-word
መ ጽሐፋቸው	split word			Single	Real-word
ወደ ተፈለገው	split word			Single	Real-word
በኢትዮጵያተዋህዶ	Run-on			Single	Real-word
ያውቋትልና	Substitution	Same family	Middle	Single	Non-word
ለማነኛውም	Substitution	Same family	Middle	Single	Non-word
ሕንጻወች	Substitution	Same family	Middle	Single	Non-word
ተሸከርክራ	Substitution	Same family	Middle	Single	Non-word
ትዝታወች	Substitution	Same family	Middle	Single	Non-word
የሚታወሱባቸ	Deletion		Last	Single	Non-word
በአላት(ዝክረ	Run-on			Single	Real-word
የመቴዎስ	Substitution	Same family	Middle	Single	Non-word
ማፍቅ	Deletion		Middle	Single	Non-word
እንደመነጽር	Run-on			Single	Real-word
ባአይነ	Substitution	Same family	First	Single	Non-word
ላያ	Substitution	Same family	Last	Single	Non-word
ጠቅሸ	Substitution	Same family	Last	Single	Non-word
ያገናኝናል	Substitution	Same family	Middle	Single	Non-word
አጋጣሚወች	Substitution	Same family	Middle	Single	Non-word
ስሰማይ	Insertion		Last	Single	Non-word
ክርስቲያና	Deletion		Last	Single	Non-word
የከንሳስ የከንሳስ	Context			Single	Real-word
የማይውጥላቸው	Substitution	Same family	Middle	Single	Non-word
በላየ	Substitution	Same family	Last	Single	Non-word
መመሰክር	Substitution	Same family	Middle	Single	Non-word
እንጅ	Substitution	Same family	Last	Single	Non-word
እንደርሰዎ	Substitution	Same family	Middle	Single	Non-word
ቆሞው	Substitution	Same family	Middle	Single	Non-word
እጸዋት	Substitution	Same family	Middle	Single	Non-word
ህይዋናን	Substitution, Insertion	Same family	Middle, last	Multiple	Non-word
በእጸዋትና	Substitution	Same family	Middle	Single	Non-word
ግለ ሰብ	split word			Single	Real-word
የሚተየው	Substitution	Same family	Middle	Single	Non-word
እንደይነቀል	Substitution	Same family	Middle	Single	Non-word
መጻሕፉቻቸንን	Deletion		Middle	Single	Non-word
የተማረነውን	Substitution	Same family	Middle	Single	Non-word
እንዲት	Substitution	Same family	First	Single	Non-word

የሌሽነት	Substitution	Same family	Middle	Single	Non-word
ከዶ ክተር	split word			Single	Real-word
አረበኛ	Substitution	Same family	Middle	Single	Non-word
ከዲስ	Substitution	Same family	First	Single	Non-word
ሙያወች	Substitution	Same family	Middle	Single	Non-word
ለመቆጣጠ	Deletion		Last	Single	Non-word
እየደረስ	Substitution	Same family	Last	Single	Non-word
መደምደሚየ	Substitution	Same family	Last	Single	Non-word
እሸገራለሁ	Substitution	Same family	Middle	Single	Non-word
ቦታወችና	Substitution	Same family	Middle	Single	Non-word
እንዲንቀጠቀትበት	Substitution	Different family	Middle	Single	Non-word
ቦታወችና	Substitution	Same family	Middle	Single	Non-word
ሚፈጽመውን	Deletion		First	Single	Non-word
እንደለው	Substitution	Same family	Middle	Single	Non-word
የተኛቸው	Substitution	Same family	Middle	Single	Non-word
በበርሀ	Substitution	Same family	Middle	Single	Non-word
ተንቀጥቅጦባትል	Substitution	Same family	Middle	Single	Non-word
ጠቷ	Substitution	Same family	First	Single	Non-word
የሚያሰድጉበት	Substitution	Same family	Middle	Single	Non-word
በወግን	Substitution	Same family	Middle	Single	Non-word
እያነሳስ	Substitution	Same family	Last	Single	Non-word
እርስስ	Insertion	Same char	Last	Single	Non-word
ጥላችን	context			Single	Real-word
ኢየየሱስ	Insertion	Same char	Middle	Single	Non-word
ሰውል	Insertion	Different family	Middle	Single	Non-word
ለማውጣ	Deletion		Last	Single	Non-word
በማይታዘ	Deletion		Last	Single	Non-word
እንዴትይልቅ	Run-on			Single	Real-word
የሚፈጸመ	Deletion		Last	Single	Non-word
ሚሸሸበት	Deletion		First	Single	Non-word
ሊፈበጥስ	Substitution, Insertion		different, Same family	Single	Non-word
እነደነ	Substitution	Same family	Middle	Single	Non-word
ለማውጣ	Deletion		Last	Single	Non-word
ሰፈሮ	Substitution	Same family	Middle	Single	Non-word
በፊካሪው	Substitution	Same family	midde	Multiple	Non-word
እስክንገናገኝኝ	Insertion	Same char	Middle	Multiple	Non-word

Appendix 8 – Sample code used by the Rule Filter component

```
def findcommonrules(word):
    spelling = ""
    conjrules = ""
    suffixrules = ""
    preprules = ""
    prefixrules = ""
    conjrules1 = []
    suffixrules1 = []
    preprules1 = []
    prefixrules1 = []
    conjrules2 = []
    suffixrules2 = []
    preprules2 = []
    prefixrules2 = []
    conjrules3 = []
    suffixrules3 = []
    preprules3 = []
    prefixrules3 = []

    allrules = []
    preceed=""
    errorpositions = []
    nextngram = "false"
    ruleweight = 0
    conjweight = []
    suffixweight = []
    prepweight = []
    sufweight = []
    prefixweight = []

    if (searchindictionary(word, "RootDict1.csv")=="true"):
        spelling = "correct"
    else:
        w = word
        n=1
        ngramslist1 = divideintongrams(word, n)
        print "Unigrams"
        print "========="
        print ngramslist1
        conjrules1 = searchConj(ngramslist1[len(ngramslist1)-1], "Rules1.csv")
        suffixrules1 = searchSuffix(ngramslist1[len(ngramslist1)-1], "Rules1.csv")
        preprules1 = searchPrep(ngramslist1[0], "Rules1.csv")
        prefixrules1 = searchPrefix(ngramslist1[0], "Rules1.csv")

        for cr in conjrules1:
            allrules.append(cr)
            allrules.append(ngramslist1[len(ngramslist1)-1])
            allrules.append(0)
            allrules.append('C')

        for pr in preprules1:
            allrules.append(pr)
            allrules.append(ngramslist1[0])
            allrules.append(0)
            allrules.append('P')

        for k in range(len(suffixrules1)/2):
            allrules.append(suffixrules1[(k*2)])
            allrules.append(ngramslist1[len(ngramslist1)-1])
            allrules.append(int(suffixrules1[(k*2)+1]) + 7)
            allrules.append('S')

        for m in range(len(prefixrules1)/2):
```

```
            allrules.append(prefixrules1[(m*2)])
            allrules.append(ngramslist1[0])
            allrules.append(int(prefixrules1[(m*2)+1]))
            allrules.append('R')


    print "conjrules1"
    print conjrules1
    print "suffixrules1"
    print suffixrules1
    print "preprules1"
    print preprules1
    print "prefixrules1"
    print prefixrules1

    n=2
    ngramslist2 = divideintongrams(word, n)
    print "Bigrams"
    print "=========="
    print ngramslist2
    conjrules2 = searchConj(ngramslist2[len(ngramslist2)-1], "Rules1.csv")
    suffixrules2 = searchSuffix(ngramslist2[len(ngramslist2)-1], "Rules1.csv")
    preprules2 = searchPrep(ngramslist2[0], "Rules1.csv")
    prefixrules2 = searchPrefix(ngramslist2[0], "Rules1.csv")

    for cr in conjrules2:
        allrules.append(cr)
        allrules.append(ngramslist2[len(ngramslist2)-1])
        allrules.append(0)
        allrules.append('C')

    for pr in preprules2:
        allrules.append(pr)
        allrules.append(ngramslist2[0])
        allrules.append(0)
        allrules.append('P')

        for k in range(len(suffixrules2)/2):
            allrules.append(suffixrules2[(k*2)])
            allrules.append(ngramslist2[len(ngramslist2)-1])
            allrules.append(int(suffixrules2[(k*2)+1]) + 7)
            allrules.append('S')

        for m in range(len(prefixrules2)/2):
            allrules.append(prefixrules2[(m*2)])
            allrules.append(ngramslist2[0])
            allrules.append(int(prefixrules2[(m*2)+1]))
            allrules.append('R')

        print "conjrules2"
        print conjrules2
        print "suffixrules2"
        print suffixrules2
        print "preprules2"
        print preprules2
        print "prefixrules2"
        print prefixrules2

        if len(word.decode('utf-8')) >= 3:
            n=3
            ngramslist3 = divideintongrams(word, n)
            print "Trigrams"
            print "=========="
            print ngramslist3

```

```
129.                conjrules3 = searchConj(ngramslist3[len(ngramslist3)-
    1], "Rules1.csv")
130.                suffixrules3 = searchSuffix(ngramslist3[len(ngramslist3)-
    1], "Rules1.csv")
131.                preprules3 = searchPrep(ngramslist3[0], "Rules1.csv")
132.                prefixrules3 = searchPrefix(ngramslist3[0], "Rules1.csv")
133.
134.                for cr in conjrules3:
135.                    allrules.append(cr)
136.                    allrules.append(ngramslist3[len(ngramslist3)-1])
137.                    allrules.append(0)
138.                    allrules.append('C')
139.
140.                for pr in preprules3:
141.                    allrules.append(pr)
142.                    allrules.append(ngramslist3[0])
143.                    allrules.append(0)
144.                    allrules.append('P')
145.
146.                for k in range(len(suffixrules3)/2):
147.                    allrules.append(suffixrules3[(k*2)])
148.                    allrules.append(ngramslist3[len(ngramslist3)-1])
149.                    allrules.append(int(suffixrules3[(k*2)+1]) + 7)
150.                    allrules.append('S')
151.
152.                for m in range(len(prefixrules3)/2):
153.                    allrules.append(prefixrules3[(m*2)])
154.                    allrules.append(ngramslist3[0])
155.                    allrules.append(int(prefixrules3[(m*2)+1]))
156.                    allrules.append('R')
157.
158.
159.                print "conjrules3"
160.                print conjrules3
161.                print "suffixrules3"
162.                print suffixrules3
163.                print "preprules3"
164.                print preprules3
165.                print "prefixrules3"
166.                print prefixrules3
167.
168.            print "All rules"
169.            print allrules
170.
171.            commonrules = checkIfDuplicates(allrules)
172.
173.            conjrules1, conjrules2, conjrules3, preprules1, preprules2, preprules3, suf
    fixrules1, suffixrules2, suffixrules3, prefixrules1, prefixrules2, prefixrules3 = e
    xcludeCommonRules(commonrules, conjrules1, preprules1, suffixrules1, prefixrules1,
    conjrules2, preprules2, suffixrules2, prefixrules2, conjrules3, preprules3, suffixr
    ules3, prefixrules3)
174.            return commonrules, conjrules1, conjrules2, conjrules3, preprules1, preprul
    es2, preprules3, suffixrules1, suffixrules2, suffixrules3, prefixrules1, prefixrule
    s2, prefixrules3
```

Appendix 9 – Sample code to reverse derivation pattern to its stem form

```
def reverse_derivation(derivation, ruleid, filechars, filerules):

    cols=['ha', 'le', 'hmha', 'me', 'sse','re','se','she','qe','be','te','che','hha',
'ne','gne','aa','ke','hhe','we','oa','ze','zze','ye','de','je','ge','tte','cche','p
pe','tse','otse','fe', 'pe']
    wordpattern=""
    word=""
    wpatternlist = []
    dpatternlist = []
    charRepr = ['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P']
    numbers = [1, 2, 3, 4, 5, 6, 7, 8]
    col = 0
    combinedword = ""

    import csv
    with open(filerules, 'r') as csvfile:
        csvreader = csv.reader(csvfile)
        for r in csvreader:
            Rules = r[0].rsplit(',')
            WordPatterns = r[6].rsplit(',')
            DerPatterns = r[7].rsplit(',')
            for rule in Rules:
                for wpat in WordPatterns:
                    if (rule == ruleid):
                        wpattern = wpat
                for dpat in DerPatterns:
                    if (rule == ruleid):
                        dpattern = dpat


    wpatterncount = 0
    for j in range(len(wpattern)):
        if (wpattern[j] in charRepr):
            wpatterncount += 1
            wpatternlist.append(wpattern[j])
        else:
            wpatterncount += 1
            wpatternlist.append(wpattern[j])

    dpatterncount = 0
    for j in range(len(dpattern)):
        if (dpattern[j] in charRepr):
            dpatterncount += 1
            dpatternlist.append(dpattern[j])
        else:
            dpatterncount += 1
            dpatternlist.append(dpattern[j])

    import pandas
    df=pandas.read_csv(filechars)

    if(wpatternlist[0] != '@' and wpatternlist[len(wpatternlist)-2] != 'Z'):

        for i in range(len(derivation)):

            generated = dpatternlist[i]
            der = derivation
            chr = der[i:i+1]

            col = 0
            for k in cols:
                order = 1
                col+=1
```

```
                    for row in df[k]:
                        rw = row.decode('utf-8')
                        if (rw == chr):
                            ord = order
                            generated = generated + str(ord)
                            generated = dpatternlist[2*i] + dpatternlist[(2*i)+1]

                            k = "none"
                            row = "~"

                            order +=1

                            for n in range(len(wpattern)):
                                wcol = 0
                                if (wpattern[n] == generated[0:1]):

                                    worder = wpattern[n+1]
                                    wcol = col
                                    pos = n
                                    n = len(wpattern) + 1

                                    for m in range(len(cols)):
                                        orr = 0
                                        for q in df[cols[m]]:

                                            if (((orr+1) == int(worder)) and ((m+1) == in
    t(wcol))):
                                                wpatternlist[pos] = q
                                            orr+=1

        for w in wpatternlist:
            if (w not in str(numbers)):
                word = word + w

    elif wpatternlist[0] == '@' and wpatternlist[len(wpatternlist)-2] != 'Z':

        generated = dpatternlist[0]
        der = derivation
        chr = der[0:1]

        col = 0
        for k in cols:
            order = 1
            col+=1
            for row in df[k]:
                rw = row.decode('utf-8')
                if (rw == chr):
                    ord = order
                    generated = generated + str(ord)
                    generated = dpatternlist[0] + dpatternlist[1]

                    k = "none"
                    row = "~"

                    order +=1

                    for n in range(len(wpattern)):
                        wcol = 0
                        if (wpattern[n] == generated[0:1]):

                            worder = wpattern[n+1]
                            wcol = col
                            pos = n
                            n = len(wpattern) + 1

                            for m in range(len(cols)):
```

```
                                    orr = 0
                                    for q in df[cols[m]]:

                                        if (((orr+1) == int(worder)) and ((m+1) == int(wc
    ol))):
                                            wpatternlist[pos] = q
                                        orr+=1
        word = wpatternlist[0]
        if type(derivation) != unicode:
            derivation=derivation.decode('utf-8')
        word = word.decode('utf-8') + derivation[1:len(derivation)]

    elif wpatternlist[len(wpatternlist)-2] == 'Z' and wpatternlist[0] != '@':
        word = word[1:len(word)-1]

        generated = dpatternlist[len(dpatternlist)-1]
        der = derivation
        chr = der[len(der)-1:len(der)]

        col = 0
        for k in cols:
            order = 1
            col+=1
            for row in df[k]:
                rw = row.decode('utf-8')
                if (rw == chr):

                    ord = order
                    generated = generated + str(ord)
                    generated = dpatternlist[len(dpatternlist)-
    2] + dpatternlist[len(dpatternlist)-1]

                    k = "none"
                    row = "~"

                    order +=1

                    for n in range(len(wpattern)):
                        wcol = 0
                        if (wpattern[n] == generated[0:1]):

                            worder = wpattern[n+1]
                            wcol = col
                            pos = n
                            n = len(wpattern) + 1

                            for m in range(len(cols)):
                                orr = 0
                                for q in df[cols[m]]:

                                    if (((orr+1) == int(worder)) and ((m+1) == int(wc
    ol))):
                                        wpatternlist[pos] = q
                                    orr+=1

        if type(derivation) != unicode:
            derivation=derivation.decode('utf-8')
        word = derivation[0:len(derivation)-1]
        if (wpatternlist[len(wpatternlist)-2] != unicode):
            word = word + wpatternlist[len(wpatternlist)-2].decode('utf-8')
        else:
            word = word + wpatternlist[len(wpatternlist)-2]

    elif(wpatternlist[0] == '@' and wpatternlist[len(wpatternlist)-2] == 'Z'):

        word = word[1:len(word)-1]
```

```
190.
191.         generated = dpatternlist[0]
192.         der = derivation
193.         chr = der[0:1]
194.
195.         col = 0
196.         for k in cols:
197.             order = 1
198.             col+=1
199.             for row in df[k]:
200.                 rw = row.decode('utf-8')
201.                 if (rw == chr):
202.                     ord = order
203.                     generated = generated + str(ord)
204.                     generated = dpatternlist[0] + dpatternlist[1]
205.
206.                     k = "none"
207.                     row = "~"
208.
209.                     order +=1
210.
211.                     for n in range(len(wpattern)):
212.                         wcol = 0
213.                         if (wpattern[n] == generated[0:1]):
214.
215.                             worder = wpattern[n+1]
216.                             wcol = col
217.                             pos = n
218.                             n = len(wpattern) + 1
219.
220.                             for m in range(len(cols)):
221.                                 orr = 0
222.                                 for q in df[cols[m]]:
223.
224.                                     if (((orr+1) == int(worder)) and ((m+1) == int(wc
    ol))):
225.                                         wpatternlist[pos] = q
226.                                     orr+=1
227.
228.         word = wpatternlist[0]
229.         if type(derivation) != unicode:
230.             derivation=derivation.decode('utf-8')
231.         word = word.decode('utf-8') + derivation[1:len(derivation)]
232.
233.         combinedword = word
234.         word = word[1:len(word)-1]
235.
236.         generated = dpatternlist[len(dpatternlist)-1]
237.         der = derivation
238.         chr = der[len(der)-1:len(der)]
239.
240.         col = 0
241.         for k in cols:
242.             order = 1
243.             col+=1
244.             for row in df[k]:
245.                 rw = row.decode('utf-8')
246.                 if (rw == chr):
247.
248.                     ord = order
249.                     generated = generated + str(ord)
250.                     generated = dpatternlist[len(dpatternlist)-
    2] + dpatternlist[len(dpatternlist)-1]
251.
252.                     k = "none"
253.                     row = "~"
```

```
254.
255.                     order +=1
256.
257.                     for n in range(len(wpattern)):
258.                         wcol = 0
259.                         if (wpattern[n] == generated[0:1]):
260.
261.                             worder = wpattern[n+1]
262.                             wcol = col
263.                             pos = n
264.                             n = len(wpattern) + 1
265.
266.                             for m in range(len(cols)):
267.                                 orr = 0
268.                                 for q in df[cols[m]]:
269.
270.                                     if (((orr+1) == int(worder)) and ((m+1) == int(wc
      ol))):
271.                                         wpatternlist[pos] = q
272.                                     orr+=1
273.         if type(derivation) != unicode:
274.             derivation=derivation.decode('utf-8')
275.         word = derivation[0:len(derivation)-1]
276.         word = word + wpatternlist[len(wpatternlist)-2].decode('utf-8')
277.
278.         combinedword = combinedword[0:len(combinedword)-1] + word[len(word)-
      1:len(word)]
279.         word = combinedword
280.     return word
```