Fyp Report

by Muhammad Rehan Qureshi

Submission date: 04-Jun-2021 03:53PM (UTC+0500)
Submission ID: 1599292721
File name: 93054_Muhammad_Rehan_Qureshi_Fyp_Report_715833_1444904479.pdf (709.29K)
Word count: 4468
Character count: 25426

FYP Report
Rahbar-e-Sukhan

The Digital Poetic Mentor

Advisor and Co-Advisor

Dr. Seemab Latif

Dr. Muhammad Ali Tahir

Prepared By

Muhammad Rehan Qureshi

BEE-9B

CMS: 213701

National University of Sciences and Technology

Contents

1 Introduction

1.1 Background

Urdu computing has unfortunately been a largely neglected area of research and development. The reasons for this attitude are aplenty. Some of it can be accustomed to a lack of interest and autority of most students and researchers on the Urdu language. Their understanding of the Urdu language in general tends to be worse than their understanding of the English language–which they are not experts at either but I digress.

There is an obvious need for software that is geared towards the development of the Urdu language. This is one such effort. I being a poet and linguist wanted to develop modern, aesthetically pleasing software that helps familiarise people with the traditional metrical system of Urdu poetry. It is an ambitious endeavour that echoes Iqbal's inspiring couplet.

اٹھ کہ خورشید کا سامانِ سفر تازہ کریں
نفسِ سوختہ شام و سحر تازہ کریں

This project aims at developing an automaton which could automate the scansion of Urdu couplets and reserve its judgement on the couplets as to whether their rythm is correct and consistent throughout the poem or not, and give apt suggestions for correction.

This would enable beginner poets to focus more on their content without having to worry about the difficulties of rythm and metre.

1.2 Literary Background

For more information on the study of rythm and metre, the reader may consult two of the most revered books in this tradition namely Aaina-e-Balaghat [1] and Baharul Fastaht [2].

1.3 Intended Audience

This project targets learners of Urdu prosody, aspiring Urdu poets and poetry aficionados.

2 Literature Review

2.1 Introduction

This section will provide an overview of the literature that concerns this project, especially its self-contained parts that help realise such a sytem.

2.2 Urdu Metre Representations

Perhaps the most important publication on the Urdu metrical system in the West is FW. Pritchett's Urdu Meter: A Practical Handbook [3].

Pritchett has introduced a syllable base simplification of Urdu metre which has become the de-facto standard in this context. Pritchett disintegrates words in short and long syllables. My approach on the other hand breaks them down on the basis of the presence of حرکات and سکنات.

Representation	تقطیع	تحلیل	لفظ
LL	فِعلُن	مِٹ ٹی	مٹی
1010	فِعلُن	مِٹ ٹی	مٹی

Table 2.1: A comparison of Pritchett's and my encoding. L = long syllable, S = short syllable

2.3 Metre Detection in Arabic Classical Poetry

As I have already explained, Urdu's metrical system has been derived from Persian and Persian's metrical system is a modification of Arabic Arud (عروض).

There are quite a few differences between these systems but they have the same basis. This section will review work done on metre detection in classical Arabic poetry.

A recently published paper in the IEEE Jorunal [4] has introduced AIMS (Arabic Meter Detection System). They claim that the accuracy of already existing solutions is 75% by syllable segmentation and 96.38% with recurrent neural networks. Another paper that describes its solution as numerical prosody [5] claims an accuracy of 98.6%.

2.3.1 Algorithm

The authors describe their method in the paper. A summary is given below.

Algorithm 1: Algorithm for Metre Detection in Classical Arabic Poetry

Input: Poem
Output: Metre

1. Collect pattern combinations
2. Add Combinations for each verse
3. Prune database to mitigate metre conflicts
4. Test metre detection for all metre variants

2.3.2 Results

The authors report an accuracy of 99.3% in the aforementioned paper.

2.3.3 Difference in Approach

This paper also uses the same encoding as I but eventually changes "10" to L and each remaining "1" to S. This actually converts the encoding I use to Pritchett's proposed encoding.

The primary difference here is the fact that this system works only for correct poetry from the persepctive of Arud and can not analyze problems in the rhythm so it is basically a much more primitive version of the system I have developed and has a much smaller scope.

There is also no available application that I have come across that makes use of these algorithms.

2.4 Algorithm for Analysis and Detection of Metres in Turkish

Turkish is another language that has traditionally adopted Persian Arooz for its poetry. Saeb Tabrezi who is the most prolific Persian poet actually had Turkish as his first language.

A 2012 paper [6] discusses the possibility of detecting metre in Turkish poetry and outputting the flaws in the input. This is actually very close to what I have achieved in my system.

The algorithm they have devised can be seen in Algorithm 2.

Algorithm 2: Alogrithm defined in the Kurt paper for Metre Detection

Input: Poem

Output: The meter of the poem, and the scansion, the flaws, and the spoken form

1. Clean and convert Text
2. Extract syllables
3. Find the longest verse
4. Retrieve metres with length close to the longest verse
5. Compute mismatching syllables
6. Output the flaws

2.4.1 Shortcomings

Even though this paper is actually performing a very similar task to what I am doing. There are numerous problems with this approach.

Firstly it is reliant only on the length of the verses for measuring the closeness, I on the other hand have used a much more robust string distance measure in Levenshtein distance.

Secondly simply notifying the user of all mismatched syllables is very inefficient. In my application I use the Levenshtein distance to obtain the best global alignment for the metre and the verse and then notify the users of the mismatches. This approach minimises the number of mismatches and provides the most optimal correction.

2.5 Metre Classification in Persian Poetries

This paper published at the IEEE International Symposium on Signal Processing and Information Technology [7] uses support vector machines to classify metres in Persian poetry.

A summary of their algorithm has been provided in Algorithm 3.

The best metre alignment is used in this case to classify the speech into the closest metre.

Algorithm 3: Algorithm for Metre Detection in Classical Arabic Poetry

Input: Speech
Output: Classified Metre

1. Sampling and Quantisation
2. Preprocessing
3. Syllabificaiton
4. Feature Extraction
5. Syllable Classification
6. Meter Alignment

2.5.1 Differences

This is actually a very different problem from what I have been trying to solve. Metre obviously originates from the musicality of the inflections in speech. The classifier at hand uses Supprot Vector Machines (SVM) to classify the musical utterance of speech.

This approach obviously does not involve text processing and can not point out if a specific pattern of speech is correct or not in the context of the metre.

2.6 Conclusion

This was an overview of published literature that overlaps with the work I have done. It is quite evident that these papers have not exactly been implemented in production quality applications yet while I have developed applications with user interfaces that automate the process of Scansion and correction of Urdu poetry.

3 Structural Development

3.1 Need Assessment

People on different literary forums had expressed the desire for these applications a number of times. There is an obvious vacuum and lack of good software written for the needs of our people.

A lot of people in the poetry community have expressed the need for such solutions on all platforms. This project tries to fill that vacuum.

3.2 Technical Requirements

Applications were to be made for the following platforms to maximise accesibility.

- Web
- Android
- Linux
- MacOS
- Windows

Multiple approaches are available for this including cross-platform solutions that work across the board. The pros and cons had to be considered for native and cross-platform solution.

3.3 A brief overview of Arooz

Arooz is a quantitative metrical system which was originally introduced as a standardisation of the rhythms ubiquitous in Arabic poetry by Khalil bin Ahmed Frahidi. Subsequently it made its way into Persian and Urdu poetry. This system works by quanitfying the stresses of words into smaller units and using them to build common rhythmic structures.

The scansion is performed manually and only someone with a good command on the ins and outs of Arooz can do it well. One example has been given below, its scansion has been provided in Table 2.1.

مٹی تغاروں میں پڑی سوکھتی رہے

حروف	تقطیع
مِٹ ٹی تَ	مفعول
غارِ چو(ں) مِ	فاعلات
پڑی سو ک(ھ)	مفاعیل
تی رہے	فاعلن

Table 3.1: Scansion Table

3.4 Simplified Approach

To make the difficulties of عروض understandable to computers I have introduced a simplified encoding that makes use of binary digits to represent word pronunciations.

لفظ	تحلیل	تقطیع	Representation
مٹی	مِٹ ٹِی	فِعلُن	1010

Table 3.2: Encoding

The encoding works by assigning short vowels (حرکات), which are represented by optional diacritics in Urdu, a value of "1". Each consonant ending (تسکین) is assigned a value of zero.

3.5 Metre Lists

A list of metres commonly used in Urdu poetry also needed for this program to work.

3.6 Keyboard Layout

Most people do not have Urdu keyboard layouts on their devices. I have added a custom layout on my web application which would allow them to write Urdu with an English keyboard layout.

3.7 Dictionary

The first and foremost requirement for the FYP was a dictionary with pronunciations. Fortunately scraped data from Urdu Lughat Board's compiled dictionary (لغتِ کبیر) was available online.

A custom script was used to convert the words with diacritics to their respective binary encoding. This dictionary includes around 122,000 words with their respective encodings. This data is customary to the applications and is stored in very efficient data structures that guarantee constant time retrieval.

3.8 Programming Languages

We know that different programming languages are suitable for different tasks. I wanted this FYP to be a learning opportunity so I have implemented the system in three different programming languages which are most suitable for the different target operating systems. They have been listed below with reasons for opting for them.

- Javascript (Web Application Frontend)
- Golang (Web Application Backend)
- C++ (Desktop Application)
- Qml (Desktop Application UI)
- Kotlin (Android Application)

3.8.1 Javascript

The only real choice when it comes to web frontends. Not exactly a great language to work with but other choices have to be compile to JS anyway.

3.8.2 Golang

A C like imperative language with garbage collection. The backend development tooling is very mature. Compilation times are the best I have come across and is generally convenient to work with. A really solid choice this one.

3.8.3 C++

Not many cross platform UI toolkits exist for C++ so it was the only choice. Modern C++ features are nice to work with and the programmes are generally exceptionally efficient.

3.8.4 Kotlin

A programming language developed by JetBrains that runs in the JVM. One of the best if not the best programming lanuages out there. Really nice to work with.

3.9 Programming Paradigms

- Imperative - Web Application
- Object Oriented - Desktop Application
- Mostly Functional with some OOP - Android Application

3.10 Other Design Choices

- The dictionary is read from text files in the Web and Desktop Applications.
- A memory mapped file is used in the Android application with the help of the MMKV library.
- The Web Applications uses AlpineJS and GoFiber as its frontend and backend frameworks.
- Qt Quick is the GUI toolkit for the desktop application.
- The Desktop application makes use of the ubiquitous CMake build system.
- The Android applciation uses the Gradle build system.
- The Desktop application uses multithreaded code to speed things even up.

3.11 UI Design

The UI has been meticulously designed for all the applications. A lot of time had to be put into it. I have improvised a way of displaying the output in a visually pleasing and easily readable way using cards.

I have tried to make the UI as interactive as possible and have made sure that the UI thread itself never gets stuck.

4 Functional Description

4.1 Product Perspective

This software is a self-contained product and is supposed to be open source under the GNU General Public License.

4.2 Product Functions

The product performs the following major functions.

- Provide a modern and responsive user interface.
- Let the user enter a series of couplets.
- Show a waiting animation while the processing occurs.
- Display the results in well-ordered card like structures for each couplet.
- Allow the users to make edits to the output on the go.
- Regenerate the results for the edited parts.

A simplified data flow diagram of the operations is given in Fig. 4.1.

4.3 User Classes and Characteristics

Users of the software would include prosody experts and researchers, learners of prosody and aspiring poets. The users may not be aware of the development processes and the limitations of such a system.

4.4 Operating Environment

The software has to be developed for three major hardware platforms that are listed below.

- Desktop
- Android
- The Web

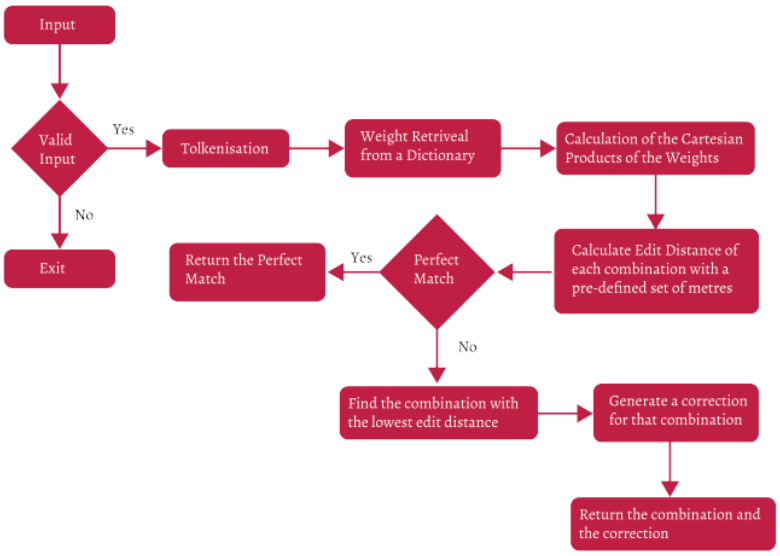

Figure 4.1: Data Flow Diagram

The server-side component of the web application has to operate within a Debian based Linux operating system. Both the client and the server-side must operate within the Transport Layer Security (TLS) cryptographic protocol. The browsers that need to be supported are listed below:

- Mozilla Firefox 60+
- Google Chrome 65+
- Apple Safari 7+

4.5 User Documentation

The Github repository for the project contains all the documentation needed.

4.6 Assumptions and Dependencies

No specific assumptions are considered at this time. The desktop application has the following dependencies:

- Qt Quick
- Intel Tbb Library for Multithreaded Code

These would need to be installed on the systems as they have been dynamically linked.

5 Core Implementation Details

5.1 Overview

This chapter consists of a detailed description of the core implementation of the system. The components of the system will be discussed in a sequential manner. Simplified code snippets will be shown to demonstrate key functionality, the actual system is much more complex.

5.2 Embedding the Dictionary in the Programme

As we know, a dictionary with the proposed binary encodings is necessary for the system to function. The words and encodings have a one-to-many mapping so a string to a list of strings map is required. Multiple approaches are available for this task, some of them I will enlist below with their merits and demerits.

Figure 5.1: Data in a CSV File

5.2.1 Reading Sequentially from a CSV file

This approach has the advantage of great efficiency. A CSV parser can be used but is not necessary since a string split operation is more than enough in most cases.

Each line is parsed one at a time and the encoding is stored in a hashmap where the word is the key and its encoding is the value. This is the approach that has been used in the web and desktop applications.

Listing 5.1: Go code that loads the dictionary into a map in the web application

```
func fetchDictFromFile() map[string][]string {

    path, _ := filepath.Abs("Files/Lughat.csv")
    file, _ := os.Open(path)
    reader := csv.NewReader(file)
    record, err := reader.ReadAll()
    dict := map[string][]string{}
    if err == nil {

        for i := range record {
            dict[record[i][0]] =
            append(dict[record[i][0]], record[i][1])
        }

    }
    return dict
}
```

5.2.2 Using a Serialisation Format

Serialisation formats have the advantage of having much nicer APIs. The performance suffers a bit since the data has to be parsed all at once as is the case with many a serialisation format including JSON, Protocol Buffers and MessagePack. Nevertheless this approach was tried in the Android application but added unnecessary start times eventually had to be removed.

Listing 5.2: Kotlin code that loads the dictionary from a msgpack serialised file into a map

```
fun fetchDict(context: Context): Map<String, Set<String>> {
    val byte =
        context.assets.open("files/dict.msgpack").readBytes()
    val ob = ObjectMapper(MessagePackFactory())
    val typeRef = object : TypeReference<Map<String,Set<String>>>() {}
    return ob.readValue(byte,typeRef)
}
```

5.2.3 Using a Memory Mapped Database

Sql based databases are an option too but not needed since complex querying is not required in general. A memory mapped key-value storage format is ideal as it allows for fast retrieval and eliminates startup times. This is why I have opted for the MMKV library by Tencent in the Android application.

During install the database is copied over to the designated directory from where it is accessible to the application with ease.

Listing 5.3: Initialising MMKV a memory mapped kev-value storage framework

```
fun loadMMKV(context: Context): MMKV? {
    val path = context.applicationInfo.dataDir
    MMKV.initialize(context)
    val dir = File("${path}/files/mmkv")
    val file = File("${path}/files/mmkv/lughat")
    val file2 = File("${path}/files/mmkv/lughat.crc")
    dir.mkdirs()

    if (file.length() < 4000000) {
        file.createNewFile()
       file2.createNewFile()
      context.assets.open("files/lughat").toFile(file)
        context.assets.open("files/lughat.crc")
            .toFile(file2)
    }
    return MMKV.mmkvWithID("lughat")
}
```

5.3 Text Preprocessing and Splitting

Text preprocessing in my case involves removal of non-Urdu characters and unnecessary whitespaces so that the splitting of the input for the system is smooth.

Listing 5.4: Text Preprocessing for the Web Application

```
func splitInput(input string) [][]string {
    input =
        regexp.MustCompile(`[\t\r\n]+`).ReplaceAllString(strings.TrimSpace(input),
        "\n")
    lines := strings.Split(input, "\n")
    words := [][]string{}

    for i := range lines {
        lines[i] = strings.Join(strings.Fields(
            strings.TrimSpace(lines[i])), " ")
        words = append(words, strings.Split(lines[i], " "))
    }
```

```
    return words
}
```

5.4 Searching for Root Words

It is not possible to have each and every word in the system. So we have to implement something that would allow derived words to be identified correctly.

Listing 5.5: Taking care of words with plural endings

```
func plurals(word string, dict map[string][]string, lastWord bool) []string
    {
    temp := word[:len(word)-2]
    weights := []string{}
    if len(dict[temp]) > 0 {
        for i := range weights {
            weights = append(weights, strings.TrimSuffix(weights[i], "0"))
        }
    }
    return weights
}
```

5.5 Handling words with droppable endings

Words ending with certain characters including {الف، و، ی،ه} have droppable endings. Meaning that the last consonant ending could be remmoved. This has to be taken into account.

Listing 5.6: Handling words with droppable endings

```
private fun updateIsqat(weights: Set<String>): Set<String> {
    if (weights.isEmpty()) return emptySet()
    val additionalWeights = weights.filter { it.endsWith('0') }
        .map { it.dropLast(1) }
    return weights.union(additionalWeights)
}
private fun canDrop(word: String): Boolean {
    val droppable = word.takeLast(2) in droppableSuffixes
            || word.takeLast(1) in droppableSuffixes

    val notPersianWord: Boolean = word.takeLast(2) !in farsiSuffix ||
            !wordPresent("${word.dropLast(1) {")
    return droppable && notPersianWord
}
```

5.6 Generating Combinations

Combinations for each line have to be generated. This is done by taking a cartesian product of a 2d list of weights corresponding to each line.

Listing 5.7: Generating combinations for each line

```
fun <T> Collection<Iterable<T>>.getCartesianProduct(): Set<List<T>>? =
    if (isEmpty()) null
    else drop(1).fold(first().map(::listOf)) { acc, iterable ->
       acc.flatMap { list -> iterable.map(list::plus) }
    }.toSet()
```

5.7 Obtaining Closest Matching Metre for a set of Combinations

This is done using a string distance metric. The Web Application uses the SIFT-3 algorithm while the android and desktop applications use Levenshtein distance to gauge string similarity.

Listing 5.8: Finding a closest match in C++

```
combination Meter::functional_set(std::vector<combination> combinations,
    std::string meter){
    combination result;

        result = *std::min_element(combinations.begin(),
            combinations.end(),[&meter](combination& a, combination& b){
            auto len_diff = a.len_diff(meter);
            return (len_diff < 5) && (a.levenshtein_distance(meter) <
                b.levenshtein_distance(meter));
        });
    return result;
}
```

5.8 Obtaining the Most Matched Metre

Each closest match has to kept into memory for this. A complete match has to be preferred over incomplete matches. This is a rather complex operation and functional programming features have been used to make the code more readable. It may however be harder to grasp for those unacquainted with the codebase.

Listing 5.9: Obtain Most Matched Metre

```
std::string Meter::find_universal_meter(std::vector<combination>
    combinations) {
    auto closest_meter =
        std::min_element(std::execution::par_unseq,METERLIST.begin(),METERLIST.end(),
        [&combinations,this](std::string a, std::string b) {
              auto closest1 = functional_set(combinations,a);
              auto closest2 = functional_set(combinations,b);

              if (closest1.combination_mozun(closest_scansion, a)){
                 closest = closest1;
                 return true;
              }
              if (closest1.combination_mozun(closest_scansion, b)){
                 closest = closest2;
                 return false;
              }
              bool condition = closest1.levenshtein_distance(a) <
                  closest2.levenshtein_distance(b);
              closest = condition ? closest1 : closest2;

              return condition;
    });
    return *closest_meter;
}
```

5.9 Obtain Closest Combinations for each Line

After the most matched metre has been obtained, another pass is needed to get the closest combination to the most matched metre for each line. The following code makes use of the "FuzzyWuzzy" library to make things a bit easier on Android.

Listing 5.10: Obtaining closest combinations

```
    fun obtainClosestCombination(
    universalSet: Set<String>, combs: List<Combination>, alienWords: List<Word>)
        : Pair<Combination, String> {

    val closestCombinations =
        combs.map { Pair(it, FuzzySearch.extractOne(it.combinationString,
            universalSet)) }
    val closestCombination = closestCombinations.maxByOrNull { it.second.score
        }!!

    return Pair(closestCombination.first, closestCombination.second.string)
}
```

5.10 Generating Correction

For generaing corrections on Android I make use of the Needleman-Wunsch sequence alignment algorithm. Functional programming helps make this fiendishly hard task pretty compact.

Listing 5.11: Generating Correction on Android

```
fun generateCorrection(weightList: List<String>, metreSting: String) :
    List<String> {

    if (weightList.joinToString("").removeSuffix("1") == metreSting)
        return weightList

    val alignment = levenshteinAlign(weightList.joinToString(" "),
        metreSting)
    val extrasRemoved = alignment[1].filterIndexed { index, _ ->
                    !(alignment[0][index] != ' ' && alignment[1][index] ==
                        '-') }
    val a = extrasRemoved.foldIndexed("") {index, acc, c ->
        if (alignment[0][index] == ' ' && alignment[1][index] != '-')
            "$acc$c-"
        else "$acc$c" }
    return a.split('-')
```

5.11 Conclusion

To make this report easily understandable, I have avoided most implementation details in this section and have provided a very simplified overview. I will not discuss the details concerning the UI and other control-flow structures that make the system work effectively and make it bug resistant. I have also not touched the dependencies that are needed and how the build systems work in this section.

The source Code for the Web and the Mobile application is available at my Github Account, for anyone willing to delve more into detail.

6 Results

The results are in general very impressive. Most of the functional requirements have been achieved and significant experience has been gained with the technologies involved. Some figures are shown below even though it's hard to show all the functionality in text.

6.1 Web Application

Figure 6.1: The Web Application in Action

6.2 Android application

Figure 6.2: The Android Application in Action

6.3 Desktop Application

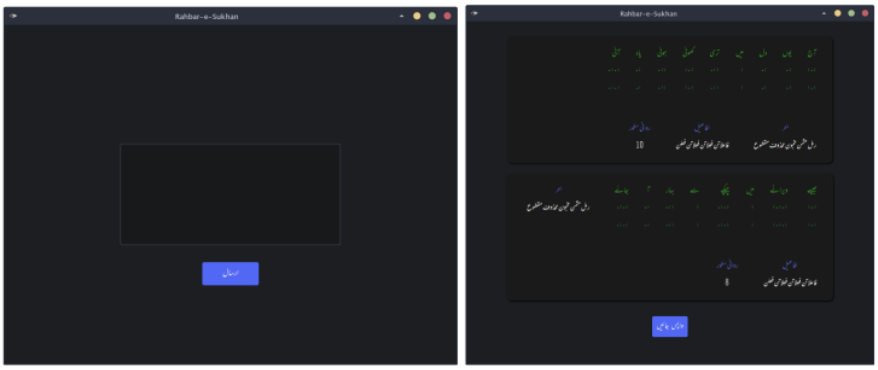

Figure 6.3: The Desktop application in Action

7 Conclusions

The project has gone mostly according to plan. Some additional features are still desired and some bug removals will also be needed. But there is a clear direction for future work in my mind and since it is a passion project I plan to support it for a while and also monetise it on some platforms in the future too.

7.1 Furture Work

This project I believe has a lot of potential and I would like to enlist a few ideas that future work could be based on.

- Further improve the UI/UX of the applications.
- Add Light and Dark modes across the board.
- Embed a word-meaning dictionary in the applications.
- Use Machine Learning to disambiguate homographs.
- Add support for Urdu written in the Roman script.

7.1.1 Urdu in the Roman Script

Most people to this date cannot unfortunately type Urdu in the Arabic scrpt. Support for Roman Urdu would make the app accessible to them too. Urdu in the Roman script could be standardised this way.

Another advantage with the Roman script is that short vowels are explicitly specified unlike the Arabic script this means that keeping a dictionary is not needed and homographs don't come up.

Bibliography

[1] Mohammad Askari. *Aaina-e-Balaghat.* Maulana Azad Library, A.m.u, Lucknow, 1936.

[2] Mohd. Najmul Ghani. *Baharul Fasahat.* State Central Library, Hyderabad, 1917.

[3] Frances W Pritchett and Ahmad Khaliq Khaliq. *Urdu Meter: A Practical Handbook.* FW Pritchett and KA Khaliq, 1987.

[4] Abdelmalek Berkani, Adrian Holzer, and Kilian Stoffel. Pattern matching in meter detection of arabic classical poetry. In *2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–8, 2020.

[5] MAYTHAM ALABBAS, ZAINAB A. KHALAF, and KHASHAN M. KHASHAN. Basrah: an automatic system to identify the meter of arabic poetry. *Natural Language Engineering*, 20(1):131–149, 2014.

[6] Atakan Kurt and Mehmet Kara. An algorithm for the detection and analysis of arud meter in diwan poetry. *Turkish Journal of Electrical Engineering and Computer Sciences*, 20:948–963, 01 2012.

[7] Saeid Hamidi, Farbod Razzazi, and Masoumeh P. Ghaemmaghami. Automatic meter classification in persian poetries using support vector machines. In *2009 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 563–567, 2009.

Fyp Report

ORIGINALITY REPORT

3%	1%	1%	1%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	KURT, Atakan and KARA, Mehmet. "An algorithm for the detection and analysis of arud meter in Diwan poetry", TÜBİTAK, 2012. Publication	1%
2	Submitted to University of Strathclyde Student Paper	<1%
3	umpir.ump.edu.my Internet Source	<1%
4	ro.ecu.edu.au Internet Source	<1%
5	aakash-tech-support-documentation.readthedocs.io Internet Source	<1%
6	Submitted to Rajarambapu Institute of Technology Student Paper	<1%
7	kar.kent.ac.uk Internet Source	<1%
8	Abdelmalek Berkani, Adrian Holzer, Kilian Stoffel. "Pattern Matching in Meter Detection	<1%

of Arabic Classical Poetry", 2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA), 2020
Publication

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	On		