Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 189 (2021) 374–381

Procedia Computer Science

www.elsevier.com/locate/procedia

5th International Conference on AI in Computational Linguistics

# QDetect: An Intelligent Tool for Detecting Quranic Verses in any Text

Samhaa R. El-Beltagy[a] and Ahmed Rafea[b]

[a]Newgiza University, Newgiza, km 22 Cairo-Alex Desert Rd, Cairo, Egypt
[b]The American University in Cairo, AUC Avenue Road, New Cairo, Egypt

**Abstract**

The past decade has witnessed an explosion in the generation of Arabic textual data, written in both modern standard Arabic (MSA) and in dialectal formats. Observing existing data reveals that verses from the Holy Quran are often quoted whether someone is giving a speech, conveying condolences, going through a hardship, and when expressing hope or joy among other emotions. Many applications can benefit from the automatic detection of those verses yet detecting Quranic Verses in text is not as easy as it may initially seem. This paper lists the challenges posed by this task, and proceeds to present an intelligent matching algorithm that addresses those. The paper also briefly describes a tool that has implemented the proposed algorithm, and which is currently publicly available as an open-source tool.



*Keywords:* Type your keywords here, separated by semicolons ;

## 1. Introduction

Quran is the holy text of Muslims, who in 2015, were estimated to be the second largest religious group in the world[1]. Quran is divided into 114 chapters or surahs, and each chapter contains a varying number of verses. Verses or fragments of verses are often used in everyday life by Arabic speaking individuals to express all sorts of emotions ranging from sympathy to joy and anger. With the rapid increase in the generation of Arabic textual data, many NLP tasks can benefit from the detection and identification of Quranic verses in text. These tasks include, but are not limited

[1] https://www.pewforum.org/2017/04/05/the-changing-global-religious-landscape/#global-population-projections-2015-to-2060.


10.1016/j.procs.2021.05.107

to: sentiment analysis, emotion detection, and automatic speech to text transcription, among others. In sentiment analysis for example, Quranic verses can be marked up or replaced with a used verse's name and number to make sure that a sentiment analysis model is not confused by words occurring in the verses and to create an association with frequently used verses instead of the words used in those verses. The same can be done with respect to emotion detection systems which will be quite useful as specific verses are often used to convey specific emotions. When verses of Quran are used in formal text, more often than not, they have to be properly referenced, meaning that the Surah and the number of each used verse, or group of verses, has be written next to it. An automatic detection system can aid in this task which can be very useful in Arabic speech to text systems.

Detection of Quranic verses in text can also be used to examine how Quran is used in social media as presented in [1] or any other medium and can be used to analyze the style of a certain person in terms of their most frequently used verses, the change in their pattern of verse usage, etc.

The rest of this paper is divided as follows: section 2 details what is involved in trying to address the verse detection task, section 3 presents the proposed framework for approaching the problem, section 4 overviews related work, section 5, presents the evaluation of the implemented tool and finally section 6, concludes this paper.

## 2. Problem Statement

The holy Quran has a total of 6236 verses. When quoting Quran, people do not necessarily quote entire verses, but often quote parts of a verse that can be found in the middle of a long verse. To detect Quran in any text, not only does a system need to detect whole verses, but also verse fragments. So, the goal of this work is to try to identify any verse fragment that is equal to or greater than 3 words, in any piece of text, and match it to its originating chapter and verse number and to do so as efficiently as possible. Matching with a sequence that is less than 3 words, is likely to generate false matches. In this work, extracted verse fragments are not allowed to end with a stop word such as: من, لا, يا, etc. Whole verses that are 1 word or 2 words long, should also be detected, but only if preceded or followed by one or more verses/verse fragments from the same chapter (surah).

A single piece of text can contain multiple verse fragments, or verses that are either consecutive or isolated and the designed algorithm must be able to match all of those. Other factors to keep in mind include the following:

- When a person is quoting a verse from memory, they may forget a word, or replace it with another word that is not a correct one. It is important for an intelligent verse detection tool to identify such errors and correct them.
- When writing Quran, spelling mistakes are not uncommon. A very common mistake for example is to write تك as تكن or المتعال as المتعالي. Again, it is important to detect and correct errors like these.
- A single verse from Quran, can occur in multiple chapters. For example the text "اللَّهُ لَا إِلَٰهَ إِلَّا هُوَ الْحَيُّ الْقَيُّومُ" is a whole verse in (آل عمران:2) and a partial verse in (البقرة:255). A system should be able to disambiguate between those if such verses are followed by other verses or verse fragments.

While the developed tool is capable of addressing the points listed above, focus in this paper has been placed on the way in which to identify any Quranic verse fragment occurring in any piece of text. To carry out this task, we must first understand the pattern space we are trying to match. For example, a verse consisting of 4 words, can match with a piece of text in one of 3 ways: the first is fully, the second is if its first three of words occurs in the text, and the last is if the last 3 of its words occur in a piece of text, so the number of patterns generated by a four-word verse is 3. Longer verses generate a larger number of patterns. An examination of the number of words per verse in Quran reveals that these vary considerably. Figure 1 shows how the length of verses (in terms of words) varies within the holy book. As can be seen in the figure, verse length varies between 1 word (28 verses) to 129 words (1 verse) with approximately half of the verses in Quran having 10 words or less and the other half having more than 10 words. A script to generate all possible patterns from all verses, revealed that there are 626,830 possible unique patterns to match with.

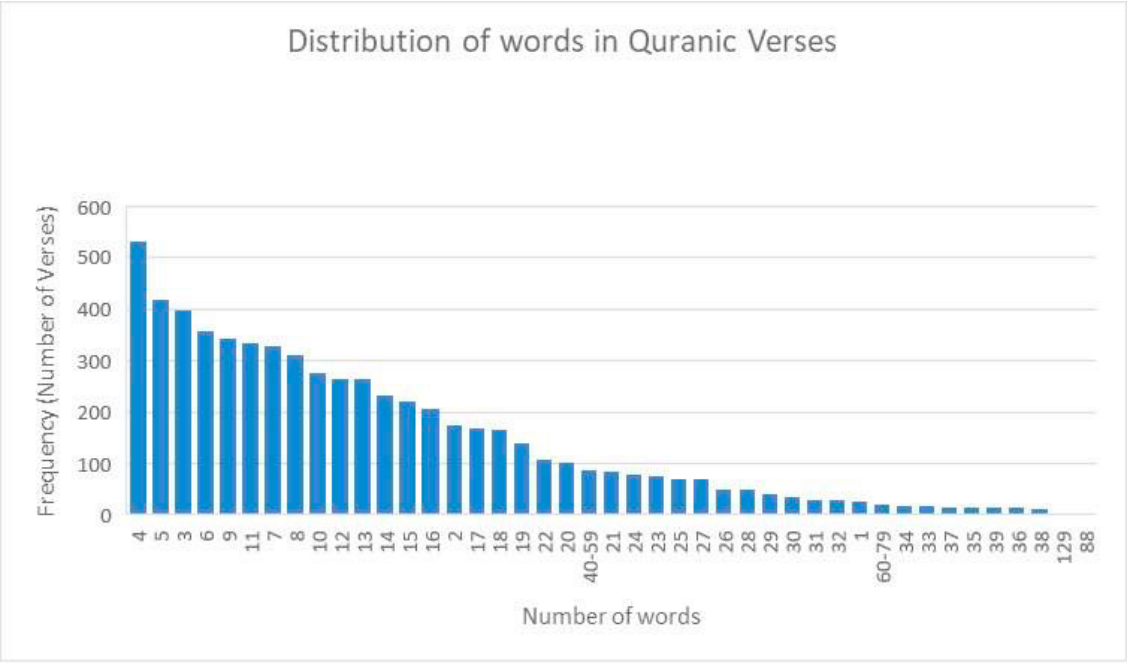


Fig. 1. Number of words per verse in Quran correlated with the number of verses in which they appear.

## 3. The proposed Model

The problem presented in the previous section can be re-iterated as follows: given some input text **T** of length ***n*** words, and the huge pattern space $\mathbf{P= \{ P_1, .., P_k\}}$ where ***k*** is approximately equal to 626,830, find all patterns $\mathbf{P_i}$, that occur in **T**, in **O(n)** time. The focus of the model is on exact matching.
The matching process happens in two steps: first all verses are stored in a compact data structure to facilitate the matching process. This is called the preparation step. The second step is where the actual detection takes place in any piece of text. This is called the matching step. Both of those steps are described in detail in the following two subsections. Error correction and the implementation of the model are presented afterwards.

### *3.1. The preparation step*

In this step, verses of Quran are read[1], normalized, and stored in a data structure that is composed of linked hash tables where each entry in the top level hash table, is a key to a node that has a hash table that stores its children which are also nodes, and which are also linked to other hash tables. The data structure is very similar to a tree except that it has no root, and each of its top level nodes can be thought of as a tree in its own right. The depth of this structure is equal to the max number of words that appear in any verse, which in this case is 129. Figure 2 illustrates the idea of the adopted data structure by displaying a very small subset of top level keys and the keys of their children. In real life, the number of entries in the top level hash table is 12,412 nodes. All dotted lines in the figure, indicate an incomplete representation and only one node per level is expanded for clarification. Building this data structure is done only once, and then this structure can be used repeatedly for matching purposes as well be detailed.

1 A soft copy of the holy Quran was downloaded from: https://tanzil.net/

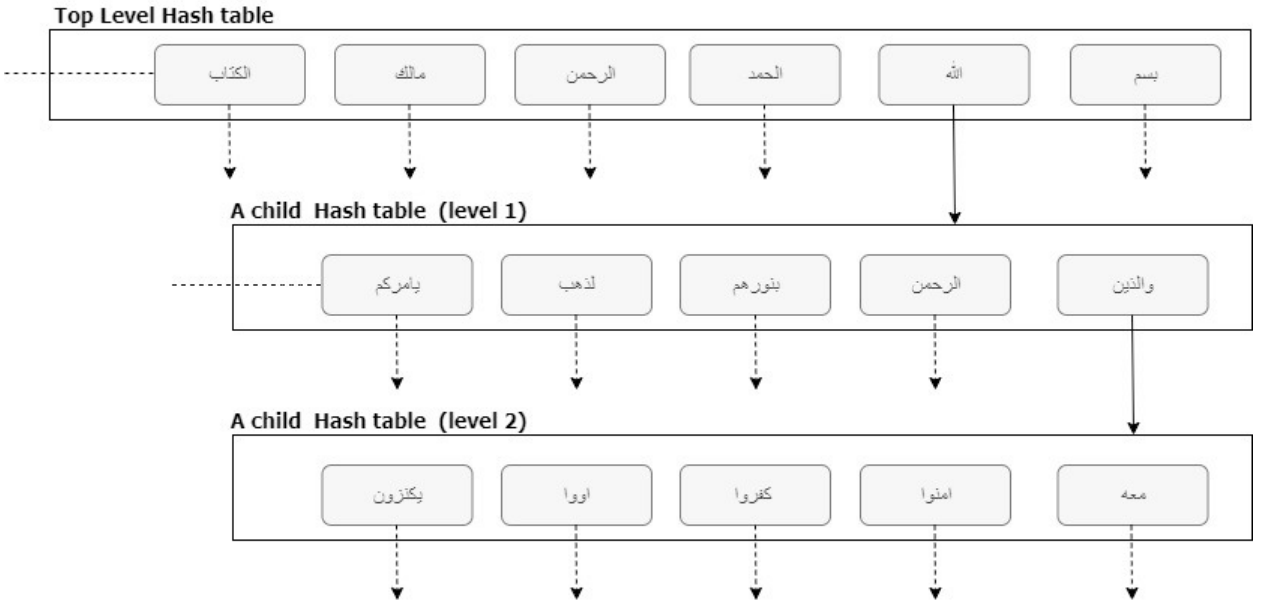


Fig. 2. A small glimpse of the constructed data structure

A node in this data structure has the following information:

- A terminal flag to indicate whether the word in this node can be a valid terminator to a verse fragment or not
- An absolute terminal flag that indicates whether the word in this node is the last term in a verse
- A set of verse records in which this branch (this word, and all its parent terms) appears where a verse record consists of the verse number and chapter/surah name in which the branch has appeared.
- A hash table which contains all the terms that appear directly after this term (initially empty). This table represents the children of this node.

From figure 2, it can be seen that in Quran, when the word الله appears as a first word, it can be followed by: والذين or الرحمن or بنورهم etc. All of this is shown in the expanded node's linked hash table (only partially shown) in which all of those appear as keys. Similarly, the figure shows that the branch 'الله والذين', can be followed by معه or امنوا or كفروا, etc.

Normalization of the verses is done by removing all diacritics, replacing letters "إ" , "أ" and "آ" with "ا", and replacing letter "ة" with "ه", and the letter "ى" with "ي". Generation of patterns is done on the fly as verses are being read and processed. The pseudo code for building this data structure is shown below.

**Pseudo code for building a data structure of linked hash tables populated with verses**

```
function buildTable()
       top_Level_table =  empty hash table
       minLen = 3                                  //minimum length of a pattern
       for each verse in the Quran
              verse = normalize(verse)
              addVerse(verse,verse_name, verse_number,  top_Level_table)
       return top_Level_table

function addVerse(verse, verse_name, verse_number, table)
       curr_table = table
       w_counter = 0
```

```
for each word in verse
      increment w_counter
      if word is in curr_table
            word_node = curr_table.get(word)
       else
            word_node = new node
            curr_table[word] = word_node
       curr_table = word_node.table

       //Check if we have reached the end of the verse
       if w_counter == verse.length
            word_node.absTerminal = True
            word_node.verses.add (verse_name, verse_number)

       //Check the current term marks the end of a valid pattern
        else if w_counter >=minLen and not word in stops_words
              word_node.terminal =True
              word_node.verses.add(verse_name, verse_number)

    //Check if more patterns can be generated from the verse
    if (verse.length-minLen) > 0
        segment_to_add = verse - first_word
        addVerse(segment_to_add, verse_name, verse_number, table )
```

The execution of this step populates the data structure that can be used for verse detection.

### *3.2. The matching step*

In this step, we have some input text input text **T** of length **n** words from which we want to detect all occurrences of Quranic verses or verse fragments. Normalization of the text takes place in exactly the same way Quranic verses are normalized as presented in the previous section. Words in the text are scanned in order, and a lookup for each encountered word in the top level hash table takes place until a match is found. When a match is found, the node of the matching key is retrieved and so is the hash table in that node which includes all valid successors of the matching term. The next term in the text is then looked up in that table, and this process continues until an encountered word in the text is not in the currently active hash table. When this happens, either a valid verse fragment of length greater than or equal to three is found, or no match is found. Since a hashing function is used, the cost of each lookup operation is O(1). A simplified version of the matching algorithm is shown below.

**Pseudo code for Matching Some Input Text to Verses or Verse fragments**

```
function match(input_Str, active_table)
      matches  = empty dictionary
      currLine = normalize(input_Str)
      i = 1
      while i <= number of words in input_Str
          word = the ith word in input_Str
          if word in active_table
              final_result_string, final_matching_verses = get_longest_match(currLine,
                                                           active_table)
              //If a match has been found
              if length of final_matching_verses >0
                  matches[final_result_string] = final_matching_verses //store the result
```

```
                i =  i + number of words in final_result_string
            else
                i = i+1
        else
            i = i+1
            currLine = input_Str - the first (i-1) words
    return matches

function get_longest_match(input_Str, active_table)
    result_string = final_result_string = empty string
    final_matching_verses = empty set

    for each word in input_str
        if word in active_table
            append word to result_string
                    activeNode = active_table.getNodeFor(word)
            matching_verses = activeNode.verses
            if (activeNode.terminal) or (activeNode.absTerminal)
                final_result_string = result_string
                final_matching_verses = matching_verses
            active_table = activeNode.table
        else
            return final_result_string, final_matching_verses
    return final_result_string, final_matching_verses
```

What is not shown in the above pseudo code, is the positional information is also stored, i.e the boundaries (start position and end position) of each matching verse fragment are also returned. The steps for spelling correction and missing word detection are also not shown.

### *3.3. Error correction*

Error correction is an optional step, that can be turned on or off using a flag. While it also should have been included in the pseudo code for matching, it was omitted in order not to over complicate the code. As mentioned before, two types of errors are handled by the proposed model: the first is the detection of mis-spelled words and the second is the detection of a missing word or wrong. Both of those steps are activated in order when there is an active branch of one or more words and then a word that is not in the active hash table is encountered. To make sure that this word is not simply mis-spelled, the Levenshtein distance [2] between this word and all words in the active hash table is calculated. If this distance is smaller than a threshold $\boldsymbol{\alpha}$, it is assumed that it is mis-spelled, and the search continues. If this step fails, a search for the non-matching word in the hash tables of all nodes in the current active table takes place. If a match is found, the node in which it was found is assumed to be missing.

### *3.4. Implementation*

The above-described model was implemented using Python. In the actual implementation, a few heuristic rules were added to reduce the possibility of detecting a verse that is not actually a verse. There is a short list of verses (exactly 2 but can be expanded by a developer) that should not be detected if they appear on their own: "بسم الله الرحمن الرحيم" and "الله ونعم الوكيل". Other rules were also included to address allowable error percentages and allowable stopword percentage in short verse fragments. The code along with examples on how to use it can be found at: https://github.com/SElBeltagy/Quran_Detector. Below are a few examples of the **annotation** feature. The input text for these examples was taken from actual tweets.

**Input Text**: " صفات #اليهود قول الله تعالى عنهم :-{تَحْسَبُهُمْ جَمِيعًا وَقُلُوبُهُمْ شَتَّى}نحن ننظر إلى اليهود فنعتقد أنهم على قلب #رجل واحد وهم في الواقع يتصارعون صراعاً مريراً فيما بينهم كما قال الله عنهم :-{بَأْسُهُمْ بَيْنَهُمْ شَدِيدٌ}"
**Output Text:** ' صفات #اليهود قول الله تعالى عنهم :-{"...ۚتَحْسَبُهُمْ جَمِيعًا وَقُلُوبُهُمْ شَتَّىٰ..."(**الحشر:14**)} نحن ننظر إلى اليهود فنعتقد أنهم على قلب #رجل واحد وهم في الواقع يتصارعون صراعاً مريراً فيما بينهم كما قال الله عنهم :-{"...ۚبَأْسُهُمْ بَيْنَهُمْ شَدِيدٌ..."(**الحشر:14**)}'

**Input Text**: RT @user: {بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ قل هو الله أحد ، اللهُ الصمد ، لم يلد ولم يولد ، ولم يكن لهُ كفواً أحد }
**Output Text:** RT @user: {بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ قُلْ هُوَ اللَّهُ أَحَدٌ، اللَّهُ الصَّمَدُ، لَمْ يَلِدْ وَلَمْ يُولَدْ، وَلَمْ يَكُنْ لَهُ كُفُوًا أَحَدٌ (**الإخلاص:1-4**)}

**Input Text**: إِنَّ الَّذِينَ يَتْلُونَ كِتَابَ اللَّهِ وَأَقَامُوا الصَّلَاةَ وَأَنْفَقُوا رَزَقْنَاهُمْ سِرًّا وَعَلَانِيَةً يَر
**Output Text:** إِنَّ الَّذِينَ يَتْلُونَ كِتَابَ اللَّهِ وَأَقَامُوا الصَّلَاةَ وَأَنْفَقُوا **مِمَّا** رَزَقْنَاهُمْ سِرًّا وَعَلَانِيَةً**...** **(فاطر:29)** يَر

The last example shows how the annotator was able to detect a missing word. The 3 dots at the end indicate that this verse is not a complete one, and still has some trailing words. Similarly, when 3 dots appear at the beginning, it means that the verse has some missing preceding terms.

## 4. Analysis

The best case performance for the presented matching algorithm is O(***n***) where ***n*** is the number of words in the input text in which we are attempting to detect Quranic verses. The worst case occurs when all words in the input string can be found in the top-level hash table, and when each of those words is followed by a word that is also in the next linked table, but the third word is not in the active table. While this case is highly unlikely, it can have a cost of up to O(**3n**). On average however, it is expected that the actual cost for applying the presented algorithm will be in the range of O(**n**) + ***C***, where ***C*** is a small enough number for the algorithm to still be in the order of ***n***. This of course just relates to exact matching without attempting to do any error detection or correction. Error detection, whether in terms of spelling or detection of missing words is expensive and carries with it a significant increase in the complexity of the algorithm.

## 5. Related Work

Very little work has been carried out for the detection and identification of Qur'anic verses from text. The work presented in [1] proposed a simple algorithm for extracting Qur'anic verses from tweets. The algorithm breaks down a tweet into sentences, and then checks if each sentence is in a list of Qur'anic verses (either fully or partially). It is not clear how the algorithm detects the boundaries of verses, nor is it clear how partial detection is achieved. There is no mention of error detection and correction. Like this work, the work presented in [3] specifically addresses the automatic detection of Quran inside of text. To do so, the authors assign a unique positional number to each word in the Quran. When a word from Quran is detected in text, all positions in which it has occurred, are retrieved. This is also done for the following words until a non Quranic word is encountered. At this point, another algorithm is applied to find the longest pattern of consecutive words by examining multiple arrays containing positional information (after sorting those arrays). While the authors claim that their algorithm is faster and more precise than traditional string-matching algorithms, they offer no numbers or analysis for comparison. However, it is obvious that if we consider ***n*** to be the number of words in the text in which they want to locate Quranic verses, the time requirement for doing so will be much larger than ***n*** as they need to carry out: searching, sorting and then matching.

The problem being addressed is essentially a string matching problem. String matching[4][5] is one of the earliest addressed problems in text processing. When there is a limited number of patterns to match with a large piece of text, the goal of these algorithms is to find a match in a time that is proportional to pattern length. When the opposite is true, i.e we have a relatively short piece of text, and a huge amount of patterns, the goal is to find all pattern matches in a time that is proportional to length of the text. Conceptually, the closest approach to the one presented is the Aho-Corasick one [6], but the presented work has replaced the finite state machine, with a linked hash structure. The proposed model is an efficient one customized for the detection of Quranic verses.

## 6. Evaluation

A dataset consisting of 473 tweets was collected[2] using the following terms for retrieval: الصلاة, استعينوا, الله تعالي, الله, and قل. The results were aggregated and then filtered to eliminate any tweets that have the same id or the exact same text. The dataset was then annotated by a person to indicate whether it contains a verse or verse fragment from Quran or not. QDetect was then applied to the tweets to see whether it matches with the human annotation or not. The tool was used to output two files, one with the number of verses/verse fragments per tweet for direct comparison with the human annotation and another with the actual detected verses (an example of an entry in this file is shown in figure 3). Furthermore, detected verses were verified by a human to make sure that the verse names and numbers returned by the tool are correct. There was a total of 95 verses in the 473 tweets. The tool was able to detect 93 verses of which 90 were correct (true positives). So the precision of the tool was 96.8%. The tool missed the detection of 5 verses (false negatives) so the recall was 94.8% and the F-score was 95.8%. One of the 3 false positive results was encountered in the following tweet (highlighted in bold). The reason is that the stop word list was missing **بها.**

user_@لا أحد يفكر أو يعتقد أن هذا ومن هم علي شاكلته ممن يخرجون عليكم يشككونكم ويضللونكم في دينكم ويحاربونكم هم مسلمين .منذ فتره بعيده كان هناك رجل يخطب في أهل المغرب ويؤمهم في الصلاه ويستشيرونه ويفتونه في أمورهم ثم مات هذا الرجل وفوجئ المغاربه بتركه **وصيه** **يوصى بها** بدفنه بمقابر اليهود.

| ID | Tweet | Matched text | Matched verses(starts) |
|---|---|---|---|
| 1389295578106980000 | RT @vU4YKgkXXOZhvJt: @umyoussef2_2_2 كنا نسميه في ابواب الشريعة (فقه المناح) انطلاقا من قوله تعالى ( لا يكلف الله نفسا الا ما آتاها) ٧٠اي ا... | ['لا يكلف الله نفسا الا ما اتاها'] | ['الطلاق7'] |

Fig. 3. An example of the output of the tool.

An example of a verse fragment that has not been detected is highlighted in the below tweet:
RT @user: 21 | رمضان 🌙 ﴿ **وَلَيَالٍ عَشْرٍ** ﴾ ربي اجعل أجورنا بها لا تُعد وبلغّنا ليلة القدر ، اللهم في العشر الأواخر قدراً جميلاً...

The reason for this is simple; the fragment does not meet the minimum length requirements.

## 5. Conclusion

This paper presented a model for full or partial identification and detection of Quranic verses in any piece of text in a time that is proportional to the length of the text when used without the error detection features. Error detection is more expensive in terms of time but can be useful for many applications. The model is available as open-source code and can be used in any application that requires the identification or annotation of Quranic verses. The result of evaluating the model, shows that it is consistently capable of identifying Quranic verses correctly, even when minor errors are encountered. Future work will aim to experiment with a larger dataset as well as target speeding up its error detection features.

## References

[1] Abokhodair N, Elmadany A, Magdy W. Holy Tweets. Proc ACM Human-Computer Interact 2020;4:1–32. https://doi.org/10.1145/3415230.

[2] Levenshtein VI. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Sov Phys Dokl 1966;10:707.

[3] Shahmohammadi M, Alizadeh T, Bijani MH, Minaei B. A framework for detecting Holy Quran inside Arabic and Persian texts. Work. Organ., n.d., p. 71.

[4] Hakak S, Kamsin A, Shivakumara P, Gilkar GA, Khan WZ, Imran M. Exact String Matching Algorithms: Survey, Issues, and Future Research Directions. IEEE Access 2019;7:69614–37. https://doi.org/10.1109/ACCESS.2019.2914071.

[5] Faro S, Lecroq T. The Exact String Matching Problem: a Comprehensive Experimental Evaluation 2010.

[6] Aho A V, Corasick MJ. Efficient string matching: an aid to bibliographic search. Commun ACM 1975;18:333–40.

[2] The dataset is available for download from: https://github.com/SElBeltagy/Quran_Detector