لا غالب إلا الله

Helwan University

Learning meters of Arabic and English poems with recurrent neural networks

Members, alphabetically:
Abdaullah Ramzy
Ali Abdemoniem
Ali Osama
Taha Magdy
Umar Mohamed

Supervisor:
Prof. Waleed A.Yousuf

June 12, 2018

Abstract

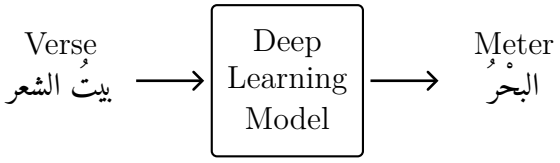
People can easily determine whether a piece of writing is a poem or prose, but only specialists can determine which meter a poem is belonged to. In the present paper, we build a model that can classify poems according to their meters; a forward step towards machine understanding of Arabic language.

A number of different *deep learning* models are proposed for poem meter classification. As poems are sequence data, then *recurrent neural networks* are suitable for the task. We have trained three variants of them, *LSTM*, *GRU* and *Bi-LSTM* with different architectures. Because meters are nothing but sequences of characters, then we have encoded the input text at the *character-level*, so that we preserve the information provided by the letters succession directly fed to thep models. In addition, we introduce a comparative study on the difference between binary and one-hot encoding in terms of their effect on the learning curve. We also introduce a new encoding technique called *Two-Hot* which merges the advantages of both *Binary* and *One-Hot* techniques.

I Introduction and Problem Statement

Detecting the meter of poems is not an easy task for ordinary people, but how computers will perform? Our task is to train a model so that it can detect the meter of the input verse/text. We have worked on Arabic and English in parallel, everything thing is applied to Arabic is applied also in English, as possible as we can.

To be clearer, the model's input is a verse/text بيت شعر and the output is a class which is the verse's meter البحر, as shown in the figure below.



The output variable is a class/categorical, then our problem can be described as *supervised learning classification*. We have trained some deep learning models such as LSTM, Bi-LSTM and GRU. Those models are chosen because of the nature of our problem. We were trying to detect the verse's meter, which is a sequence of characters and *recurrent neural network* are suitable to learn that pattern, thanks to its cell's share-memory and its recursive structure.

II The Project Road Map

This is are four-stage project.

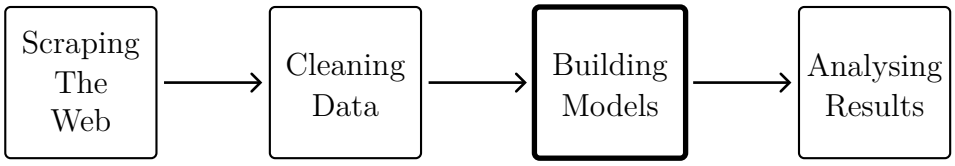
1. The first one is where we get the raw data and put it in a feature-response form.

2. The second stage is cleaning the data, by which we mean that any non-letter character and unnecessary white-spaces are removed, also, handling any diacritics issue, as it will be demonstrated in the next sections, this stage include encoding the data to the form that is suitable to be fed to our neural networks.

3. The third stage is where the models are built and are tuned then we have automated the experiments to run all the iterations one after another, much details will be presented.

4. Finally, we gather the results and analyse them, also we then conduct some additional experiments to see the language and encoding effect over the learning curve.

The first two steps were much difficult than building the models. The following figure shows the road map.



Gathering Data

III Tools

Python is pseudo-code like programming language, it is so easy and high-level that we can describe complex structures in a few lines of code, the main second reason is that python recently has been so papular in the Artificial Intelligence community. Its library is so rich with packages for Machine Learning, Deep Learning, data manipulation, even for web-scraping; we don't need to parse HTML by your hands.

We have used: Two columns:

- *Python* 3.6.5
- *Tensorflow* x.x as back-end of Keras.
- *Keras* x.x for deep learning.
- *BeautifulSoup* for web scraping.

IV Scraping The Web *–gathering the data*

To train our models we need dataset of poems. For Arabic, this kind of dataset are not so popular so you can hardly find a dataset for poetry. So, we had to create our own. During our search we have found two big poetry web sites الديوان[1], الموسوعة الشعرية[2], that contain tons of metrical-classified poems. And similar is happened for English. So we have scrapped those web sites. *Scraping* is to write a script to browse the target web site, and copy the specific content and dump it to our *csv* file. This was the most difficult step in the hole project.

Also, we present a very large Arabic dataset that can be used for further research purposes, next sub-section contains much details about the Arabic dataset. For the English dataset, the situation was worse than scraping big complex web site, because the there were not big web site that contain a large amount of metrical-classified poems, so the English dataset is to small if it is compared to its Arabic counterpart, but this what we have found after an extensive search. cite(Farren) had been generously granted his data from the Stanford English literature department, we have tried to contact them through their email but none has reposed.

The scraping scripts are under the directory `repository_root/scraping the web`, accompanied by a thorough self-contained `README.md` file, which contains everything you may need to use or re-use the code, even the code is written to be re-used with a lot of *in-line* documentation.

V Introduction

V.I Arabic Language

Arabic is the fifth most widely spoken language[3]. It is written from right to left. Its alphabet consists of 28 primary letters, and there are 8 more derived letters from the basic ones, so the total count of Arabic characters is

[1] *aldiwan.net*

[2] *poetry.tcaabudhabi.ae*

[3] *according to the 20th edition of Ethnologue, 2017*

Diacritics	*without*	*fat-ha*	*kas-ra*	*dam-ma*	*sukun*
Shape	د	دَ	دِ	دُ	دْ

Table 1: *Diacritics on the letter* د

36 characters. The writing system is cursive; hence, most letters join to the letter that comes after them, a few letters remain disjoint.

Each Arabic letter represents a consonant, which means that short vowels are not represented by the 36 characters, for this reason, the need of *diacritics* rises. *Diacritics* are symbols that comes after a letter to state the short vowel accompanied by that letter. There are four diacritics ◌َ ◌ُ ◌ِ ◌ْ which represent the following short vowels /*a*/, /*u*/, /*i*/ and *no-vowel* respectively, their names are *fat-ha, dam-ma, kas-ra and sukun* respectively. The first three symbols are called *harakat*. Table 1 shows the 4 diacritics on a letter. There are two more sub-diacritics made up of the basic four to represent two cases: the first case is when a letter is doubled where the first letter is accompanied by *sukun* and the second letter is accompanied by *haraka*, instead of doubling each letter accompanied by their *diacritics* explicitly, both of them are written once accompanied by *shadda* ◌ّ, *shadda* states that its letter is doubled, the first letter is always accompanied by *sukun*, the second letter's *haraka* is followed by *shadda*; for example, دْدَ is written دَّ. The second case, Arabs pronounce the sound /*n*/ accompanied *sukun* at the end the indefinite words, that sound corresponds to this letter نْ, it is called *noon-sakinah*, however, it is just a phone, it is not a part of the indefinite word, if a word comes as a definite word, no additional sound is added. Since it is not an essential sound, it is not written as a letter, but it is written as *tanween* ◌ً ◌ٍ ◌ٌ. *Tanween* states the sound *noon-sakinah*, but as you have noticed, there are 3 *tanween* symbols, this because *tanween* is added as a diacritic over the last letter of the indefinite word, last letter is accompanied by one of the 3 *harakat*, the last letter's *harakah* needs to be stated in addition to the sound *noon-sakinah*, so *tanween* is doubling the last letter's *haraka*, this way the last letter's *haraka* is preserved in addition to stating the sound *noon-sakinah*; for example, رَجُلُ + نْ is written رَجُلٌ and رَجُلِ + نْ is written رَجُلٍ. Those two cases will help us to reduce the dimension of the letter's feature vector as we will see in *preparing data* section.

Diacritics are just to make short vowels clearer, but they are not necessary. Moreover, a phrase without full diacritics or with just some on some letters is right linguistically, so it is allowed to drop them from text.

In Unicode, Arabic diacritics are standalone symbols, each of them has its own unicode. This is in contrast to the Latin diacritics; e.g., in the set *{ê, é, è, ë, ē, ĕ, ě}*, each combination of the letter *e* and a diacritic is represented by one unicode.

V.II Arabic Poetry الشعر العربى

Arabic poetry is the earliest form of Arabic literature. It dates back to the sixth century. Poets have written poems without knowing exactly what rules which make a collection of words a poem. People recognize poetry by nature, but only talented ones who could write poems. This was the case until *Al-Farahidi* (718 – 786 CE) has analyzed the Arabic poetry, then he came up with that the succession of consonants and vowels produce patterns or *meters*, which make the music of poetry. He has counted them fifteen meters. After that, a student of *Al-Farahidi* has added one more meter to make them sixteen. Arabs call meters بحور which means "*seas*".

A poem is a collection of verses, a verse looks like the following:

أَلِا قاتل الله الحمامة غدوةً على الأَيكِ ماذا هيّجتْ حين غنّتِ
تغنّت بصوت أعجمى فهيجت من الوجدَ ما كانت ضلوعى أجنّتِ

A verse, known as *bayt* in Arabic, consists of two halves; a half is called a *shatr*[4]. *Al-Farahidi* has introduced *al-'arud* العروض[5]; it is the study of poetic meters, in which he has laid down rigorous rules and measures, with them we can determine whether a meter of a poem is sound or broken. A meter is an ordered sequence of *feet*. Feet are the basic units of meters, there are eight of them. A Foot consists of a sequence of consonant and vowels. Traditionally, feet are represented by mnemonic words called *tafa'il* (تفاعيل). According to *al-Farahidi* and his student, there are sixteen combinations of *tafa'il*. A meter appears in a *verse* twice; each *shatr* carries the same complete meter.

For example, the following *shatr* وَيُسأَلُ في الحَوادِثِ ذو صَوابٍ is equivalent to the *meter* مفاعلتن مفاعلتن فعول, which means it belongs to *Al-Wafeer* meter. We can get the pattern of the *sukun* and *harakat* by replacing each feet by the corresponding code in table 2, which produces the following pattern that should be read from right to left:

0/0// 0///0// 0///0//

[4]it is a singular in arabic, but for simplicity we will use it for both singular and plural.
[5]it is often called the *Knowledge of Poetry*.

Feet	Scansion
فَعُولُنْ	0/0//
فَاعِلُنْ	0//0/
مُسْتَفْعِلُنْ	0//0/0/
مَفَاعِيلُنْ	0/0/0//
مَفْعُولَاتُ	0//0///
فَاعِلَاتُنْ	0/0//0/
مُفَاعَلَتُنْ	0///0//
مُتَفَاعِلُنْ	0//0///

Table 2: The eight feet. Every digit represents the corresponding diacritic over each latter in the feet. / If a letter has got *harakat* (◌َ ◌ُ ◌ِ), 0 if a letter has got *sukun* (◌ْ). Any *mad* (و, ا, ى) is equivalent to *sukun*.

This is a very brief introduction to *Arud*, many details are reduced.

Meter Name	**Meter** *feet combination*
al-Wafeer	مُفَاعَلَتُن مُفَاعَلَتُن فَعُولُن
al-Taweel	فَعُوْلُنْ مَفَاْعِيْلُنْ فَعُوْلُنْ مَفَاْعُلُنْ
al-Kamel	مُتَفَاْعِلُنْ مُتَفَاْعِلُنْ مُتَفَاْعِلُنْ
al-Baseet	مُسْتَفْعِلُنْ فَاْعِلُنْ مُسْتَفْعِلُنْ فَاْعِلُنْ
al-Khafeef	فَاْعِلَاتُنْ مُسْتَفْعِلُنْ فَاْعِلَاتُنْ
al-Rigz	مُسْتَفْعِلُنْ مُسْتَفْعِلُنْ مُسْتَفْعِلُنْ
al-Raml	فَاْعِلَاتُنْ فَاْعِلَاتُنْ فَاْعِلَاتُنْ
al-Motakarib	فَعُوْلُنْ فَعُوْلُنْ فَعُوْلُنْ فَعُوْلُنْ
al-Sar'e	مُسْتَفْعِلُنْ مُسْتَفْعِلُنْ مَفْعُوْلَاتُ
al-Monsafeh	مُسْتَفْعِلُنْ مَفْعُوْلَاتُ مُسْتَفْعِلُنْ
al-Mogtath	مُسْتَفْعِلُنْ فَاْعِلَاتُنْ فَاْعِلَاتُنْ
al-Madeed	فَاْعِلَاتُنْ فَاْعِلُنْ فَاْعِلَاتُنْ
al-Hazg	مَفَاْعِيْلُنْ مَفَاْعِيْلُنْ
al-Motadarik	فَاْعِلُنْ فَاْعِلُنْ فَاْعِلُنْ فَاْعِلُنْ
al-Moktadib	مَفْعُوْلَاتُ مُسْتَفْعِلُنْ مُسْتَفْعِلُنْ
al-Modar'e	مَفَاْعِيْلُنْ فَاْعِلَاتُنْ مَفَاْعِيْلُنْ

Table 3: *The sixteen Arabic poem meters*

V.III English poetry Introduction

English poetry dates back to the seventh century. At that time, poems were written in *Anglo-Saxon*, also known as *The Old English*. Many political changes have influenced the language until it becomes as it is nowadays. English prosody was not formalized rigorously as a stand-alone knowledge, but many tools of the *Greek* prosody were borrowed to describe the English prosody, tools like the Greek meters types which pre-dates the English language by a long time.

A *syllable* is the unit of pronunciation having one vowel sound, with or without surrounding consonants. English words consist of one or more syllables. For example the word "Water" /ˈwɔːtə/ consists of two phonetic syllables: /ˈwɔː/ and /tə(r)/. As you notice, each syllable have only one vowel sound. Syllables can be either stressed or unstressed which are referred to by / and

Feet	Stresses Combination
Iamb	×/
Trochee	/×
Dactyl	/××
Anapest	××/
Pyrrhic	××
Amphibrach	×/×
Spondee	//

Table 4: Every foot is a combination of stressed and unstressed syllables, where stressed syllable is denoted by / and unstressed syllable is denoted by ×.

×, respectively. In previous "Water" example, the first syllable is stressed, stresses are shown using the primary stress symbol ˈ in phonetics, the second syllable is unstressed, so the word "Water" is a stressed-unstressed word, which can be denoted by /×, abstracting the word as stressed and unstressed syllables. There are seven different combinations of stressed and unstressed syllables form make the seven poetic *feet*s. They are shown in table 4. Meters are described as a sequence of feet. English meters are *qualitative* meters; which are stressed syllables coming at regular intervals. A meter is repeating one of the previous seven feet one to eight times, for every verse, then a verse's meter is determined by the repeated foot. If the foot is repeated once, then verse is *monometer*, if it is repeated twice then it is *dimeter* verse, until *octameter* which means a foot is repeated eight times. Here is an example, (stressed syllables are bold).

That **time** of **year** thou **mayst** in **me** be**hold**

The first verse belongs to *Iambic* foot and it is repeated five times; so it is *Iambic pentameter*.

VI Literature review

Classifying and detecting poems problem has been addressed and formalized differently across the literature. Moreover, the history is so rich of poetry analysis studies, hundreds of years ago, even before computer appears. However, the topic is still unexplored, computationally.

Abuata and Al-Omari [1] present the most related work to our topic. They classify Arabic poems according to their *meters*. But they have not addressed it as a *learning problem*, they have designed a deterministic five-step *algorithm*

for analysing and detecting meters. The first step and the most important is having the input text carrying full diacritics, this means that every single letter must carry a diacritic, explicitly. The next step is converting input text into *Arud writing*[6] using if-else like rules. Then metrical *scansion* rules are applied to the *Arud writing*, which leaves the input text as a series of zeros and ones. After that each group of zeros and ones are defined as a *tafa'il* 2, so now we have a sequence of *tafa'il*. And finally the input text is classified to the closest meter to the *tafa'il* sequence 3. 82.2% is the classification accuracy on a relatively small sample, only 417 verse.

Alnagdawi et al. [4] has taken a similar approach to the previous work, but they formalized the *scansion*, *Arud* and some lingual[7] rules as *context-free grammar* and *regular expression* templates, the result is 75% correctly classified from 128 verses.

Kurt and Kara [6] have worked on detection and analysis of *arud* meter in Ottoman Language. They have depended on Ottoman *aurd* rules to construct an algorithm that analyses Ottoman poems. First Ottoman text must be transliterated to Latin transcription alphabet (LTA) after that text is fed to the algorithm which uses a database containing all Ottoman meters to compare the detected meter extracted from LTA to the closest meter found in the database.

Both Abuata and Al-Omari [1] and Alnagdawi et al. [4] have common problems. The first problem is that the test size cannot give an accurate performance for the algorithms they have constructed, because it is very small. And a 75% total accuracy of 128 verses is even worse. The second problem is that the operation of converting verses into zeros and ones patterns is probabilistic; it also depends on the meaning, which is a source of randomness. Then treating such a problem as a deterministic problem is not going to be satisfying. Moreover, it results in numerous limitations like obligating verses to have full diacritics on every single letter, before conducting the classification.

Here is a different approach to the previous, the algorithmic ones, Almuhareb et al. [3] has used machine learning to recognize modern Arabic poems inside documents. He has built *Naive Bayes* and *Decision Tree* classifiers which detect poems based on the visual features, like line length average, line length standard deviation, average number of block[8], standard deviation of block number, word repetition rate, diacritic rate, punctuation rate. Those features have been extracted from 2067 documents which are divided into 513

[6] It is a pronounced version of writing; where only pronounced sounds are written.

[7] like pronounced and silent rules, which is directly related to *harakat*

[8] *a block: is a group of lines separated by an empty character or more.*

modern poems and 1554 prose. Then classifiers have been evaluated using *10-fold cross-validation*. The best accuracy 99.81% has been achieved by the *decision tree* classifier which is trained on all features together.

Tizhoosh and Dara [9] has presented similar work to Almuhareb et al. [3], have trained *Naive Bayes* and *Decision Tree* using visual features, they reached accuracy above 90%.

There is a point here, visual features may work when detecting poems inside documents due to poems are written in specific structure which distinguishes them from other text inside documents. In theses approaches models have no clue about the real patterns that create poems, of course the way how words are structured inside text does not produce a poems, at all.

Tanasescu et al. [8] has worked on binary classifying English poems where *metric* and *free-verse* are the categories, he faced an interesting problem with their dataset, it was imbalanced, (871 metrical poems, 4115 free-verse), for this reason they have used *bootstrap aggregating* (also known as *bagging*), which is a meta-algorithm that can greatly improve decision tree accuracy. With *J48* and *bootstrap aggregating*, he was able to achieve a 94.39% correctly classifying poetry as metrical or not.

Encoding the categorical variables and its impact on neural network performance

Encoding features has an impact on the neural network performance. Potdar et al. [7] has done a comparative study on six encoding techniques. We are interested in the comparison of *one-hot* and *binary*. They have used Artificial Neural Network for evaluating cars based on seven ordered qualitative features. The accuracy of the model was the same in both encodings—*one-hot* and *binary*.

VII Motivation

Our work in poems classification problem proposed with a lot of new methods and techniques. In this paper we are target to acheive the human experts performance to classify the poems which is not presented in such way or performance before.

- We introduced Poem classifications as a deep learning problem, not an algorithmic problem nor rule-based to utilize the feature in a poem that it is a sequence of character and how can we apply RNN with LSTM to classify it.

1. The size of the dataset for each encoding.
2. The learning curve and the time is taken to learn the problem.
3. Classification performance results: The results we got into this paper is not achieved into any of the previous work in this field before with the variety of options or in machine learning techniques.
4. Runtime performance.

- We used different hyperparameters for each data encoding and provide comparative study based on our experiments.
- We worked on big datasets which are not represented by such amount of data before to show the effect and provide accurate and reliable results for our models.
- Our data was not balanced so, we introduce how we solve this problem using custom weighted loss function and also, compare it with the normal one. Moreover, we tried to remove the small classes to check the effect on the total accuracy for the RNN model. All the above experiments provided with comparative study and performance recommendation results.
- We applied the same techniques to the Arabic and English and provided the experiments results for both and show how we can apply it to any other language and text in general.

VIII Datasets

VIII.I Arabic dataset

We have scrapped the Arabic dataset from two big poetry websites: الديوان[9], الموسوعة الشعرية[10]. Both are merged into one large dataset. It is important to note that the verses' diacritic states are not consistent, this means that a verse can carry full, semi diacritics or it can carry nothing. The total number of verses is 1,862,046 poetic verses; each verse is labeled by its meter, the poet who wrote it, and the age which it was written in. There are 22 meters, 3701 poets and 11 ages; and they are Pre-Islamic, Islamic, Umayyad, Mamluk, Abbasid, Ayyubid, Ottoman, Andalusian, era between Umayyad and Abbasid,

[9] *aldiwan.net*
[10] *poetry.tcaabudhabi.ae*

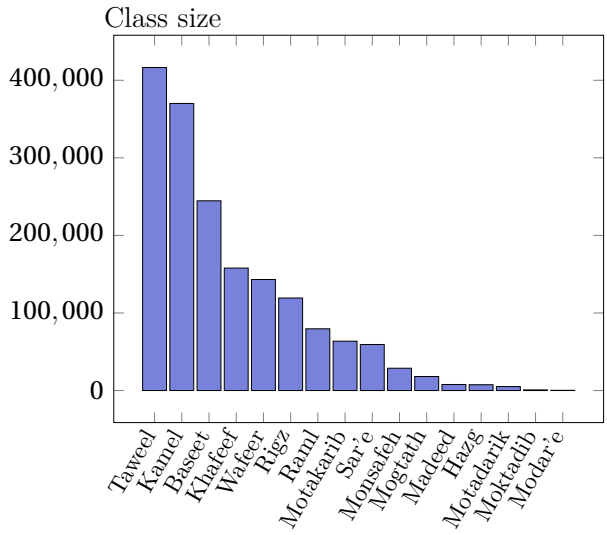


Figure 1: Meter names are on the x-axis, size is on the y-axis.

Fatimid and modern. We are only interested in the 16 classic meters which are attributed to *Al-Farahidi*, and they are the majority of the dataset with a total number of 1,722,321 verses[11].

VIII.II English dataset

The English dataset is scraped from many different web resources[12]. It consists of 199,002 verses, each of them is labeled with one of these four meters: *Iambic*, *Trochee*, *Dactyl* and *Anapaestic*. The *Iambic* class dominates the dataset; there are 186,809 *Iambic* verses, 5418 *Trochee* verses, 5378 *Anapaestic* verses, 1397 *Dactyl* verses. We have downsampled the *Iambic* class to 5550 verses.

IX Preparing Data

IX.I Data Cleaning

For both Arabic and English data, they were not clean enough, there were some non-alphabetical characters and many unnecessary white spaces inside the text, they have been removed. For Arabic, there were diacritics mistakes,

[11]https://wwww.github.com/tahamagdy
[12]http://www.eighteenthcenturypoetry.org

like the existence of two consecutive *harakat*, we have only kept one and have removed the other, or a *haraka* comes after a white space, it has been removed.

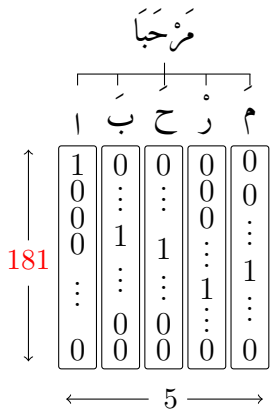
As a pre-encoding step, we have factored both of *shadda* and *tanween* to two letters, as it previously presented in Arabic language introduction, by this step we shorten the encoding vector, and save more memory.

IX.II Data Encoding

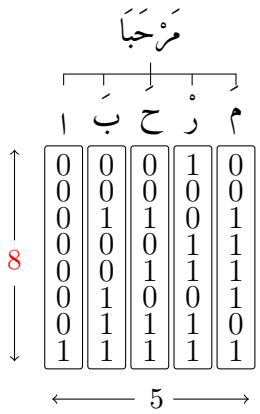
Agirrezabal et al. [2] shows that representations of data learned from character-based neural models are more informative than the ones from hand-crafted features. In our case, we want to detect meters inside verses, meters are nothing but patterns of the letters successions as previously presented, so it is convenient to encode verses at *character level* so that our model learns from sequences of characters.

Generally, a character will be represented as an n vector. Consequently, a verse would be an $n \times p$ matrix, where n is the character representation length and p is the verse's length, n varies from one encoding to another, we have used 3 different techniques to encode the characters. There is an issue; we will take the *one-hot* encoding as an example to demonstrate it. A character without diacritics is represented as a 37×1 vector, where 37 is the 36 letters in addition to a white-space character, so a phrase like مرحبا having 5 letters is encoded as a 37×5 matrix if it came with diacritics it would be represented as 41×1 vector, where 41 is 37 characters + 4 diacritics, therefore a phrase like مَرْحَبًا which has 5 letters and 4 diacritics, would be encoded as a 41×9 matrix. The issue comes from considering diacritics as stand-alone characters while encoding, a phrase with diacritics has more characters than if it was without diacritics, this leads to different number of time steps in the *RNN* cell and different input matrix dimensions which in turn leads to changing the model architecture, we do not want models architectures to differ whether diacritics exist of not in a verse so that we be able to conduct fair performance comparisons between models. As a solution we have encoded both a letter with its diacritic as one character, there are 36 letters and 36×4 combinations between each letter and each diacritic, then we have 181 characters including the white-space. From now forward, we have a 181-character alphabet, according to it, we will encode verses.

So, we used 180×1 vector to represent a character in order to avoid the previous two problems, see the upcoming figure and figure 2a.

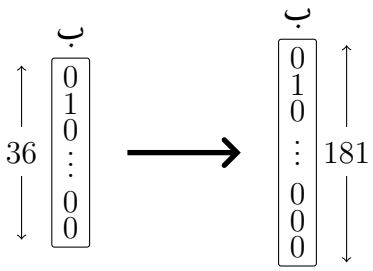


(a) *One-Hot* encoding



(b) *Binary* encoding

Figure 2: 2a Encoding the word مَرْحَبَا using n-letter alphabet *one-hot* encoding; the *one-hot* is applied to encode Arabic and English verses; the only difference is the vector length is 181 in Arabic and is 28 is English (26 letter, a white-space character and an apostrophe). In figure 2b the same word is encoded using *binary* encoding, where the vector length $n = \lceil \log_2 l \rceil$, $l \in \{181, 28\}$ is the alphabet length (Arabic/English).



One-Hot Vector: from 36×1 to 181×1

One-Hot encoding

Here, a character is represented by an $n \times 1$ *one-hot* vector, where n is the alphabet length, in *Arabic* it is the 181-character alphabet, and in *English* is the 28-letter alphabet.

Binary Encoding

The idea is to represent a character with an $n \times 1$ vector which contains a unique combination of ones and zeros. $n = \lceil \log_2 l \rceil$ where l is the alphabet length, and n is the sufficient number of digits to represent l unique binary combinations. For example a phrase like this مَرْحَبا, it has 5 characters, figure 2b shows how it is encoded as a 8×5 matrix, which saves more memory than the *one-hot* and reduces the model capacity, significantly. But on the other hand, the input characters share some features between them due to the binary representation as it is shown in figure IX.II. We will make a comparison between both encoding techniques *binary* and *one-hot*, at the end.

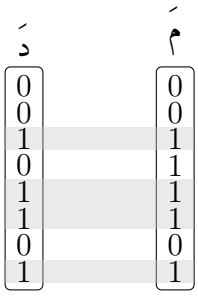


Figure 3: Common feature problem; letters may share some representation feature.

It means when characters are fed them to NN, it will compute linear combination, $Z = XW + B$, and we will multiply the features X by its weights W and apply activation function then we will compute the cross-entropy loss function and after computing the gradients using BPTT, the optimizer will update the weights to reduce the loss so if the optimizer updates weights of this two common features to reduce error of one character of them it will effect the rest characters that have the same feature.

Two-Hot encoding

This is an intermediate technique which takes the advantages of the previous two encoding techniques. In which we encode the letter and its diacritic separately using *one-hot* encoding, this way the letter is encoded as 37×1 *one-hot* vector and the diacritic is encoded as 4×1 *one-hot* vector, then both vectors are stacked to form one 41×1 vector. By this way, we reduces the vector dimension from 180 to 41 and also minimizes the number of shared features between vectors to maximum 2 ones at each vector.

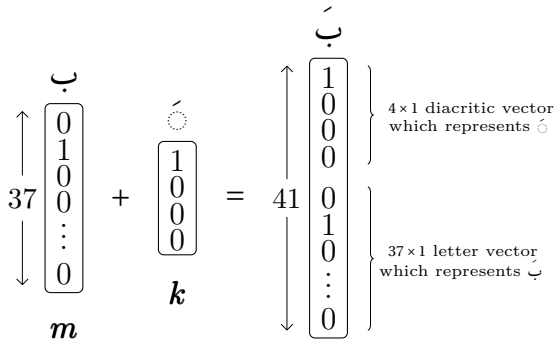


Figure 4: By stacking $\boldsymbol{k}_{4\times1}$ on the top of $\boldsymbol{m}_{37\times1}$ we get the $Two-Hot_{41\times1}\left[\begin{smallmatrix}k\\m\end{smallmatrix}\right]$, which repersents a letter and its diacritic, simultaneously.

X Model

Our experiments depend on *LSTM* introduced by Hochreiter and Urgen Schmidhuber [5]; and *Bi-LSTM*, which is two *LSTM*s stacked on top of each other. LSTM is designed to solve the *long-term dependency* problem. In theory *RNN*s are capable of handling long-term dependencies, but in practice they don not, due to *exploding gradient* problem. Where weights are updated by the gradient of the loss function with respect to the current weights in each epoch in training. In some cases the gradient maybe small, vanishingly! this prevent the weights from changing and may stop the neural network from further learning. *LSTM*s overcome that problem.

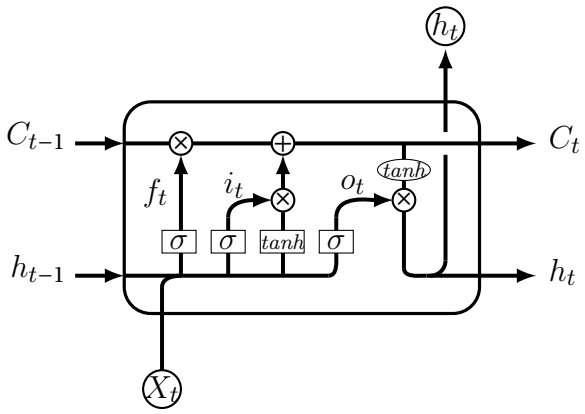


Figure 5: LSTM Cell

Figure 5 [13] shows an LSTM unit. f_t is the forgetting gate, i_t is the input

[13] figure is inspired by http://colah.github.io/posts/2015-08-Understanding-LSTMs

gate, o_t is the output gate, C_t is the memory across cells. W_j,U_j, b_j are the weight matrices and bias vector, $j \in \{f, i, o\}$. The cell's hidden representation h_t of x_t is computed as the following:

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_t) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
C_t &= f_t \circ c_{t-1} + i_t \circ tanh(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t \circ tanh(c_t)
\end{aligned}
$$

We have conducted many experiments using LSTM and BiLSTM with different architectures. As it is shown in the table 1, the dataset is imbalanced, there is a huge gap between the big and the small classes. Our training set-up is as the following. We built a model then we fed the data. There are operations performed on the data before it is fed to the model. Each operation is a set of options. The first operation is the encoding, we have 3 encoding techniques {One-Hot, Binary, Two-Hot}. The second operation is whether we drop the diacritics or we keep it {With diacritics, Without diacritics}. The third operation is whether we drop the last 5 classes or we keep them {Eliminate data, Full data}, by Eliminated data we mean dropping the last 5 tiny classes so that we have 11 classes, and by Full data we mean keeping the 16 classes. The reason of eliminating the last 5 classes is the gap between them and the rest, in addition this helps us in studying imbalanced data training. This leads to the last operation which is weighting the loss function by

$$
w_c = \frac{1/n_c}{\sum_c 1/n_c},
$$

where n_c is the sample size of class c, $c = 1, 2, ...C$, where C is the number of classes. Weighting the loss this way keeps the following:

1. the density is constant (for normalization).
2. the smaller the n_c the larger the w_c.
3. $\sum_c w_c = 1$

The total number of the experiments is the Cartesian product of all the previous options sets. The same approch is taken in for the English poems, the operations does not include Eliminated/Full and Weighted/Not-Weighted.

XI Results

XI.I Arabic Results

#	data size	encoding	diacritic	archit.	f1
1	full data	two-hot	Yes	7L, 50U, 1	95.79%
2	full data	two-hot	No	7L, 50U, 0	95.43%
3	full data	binary	Yes	7L, 81U, 0	95.51%
4	full data	binary	No	10L, 81U, 0	93.2%
5	full data	one-hot	Yes	7L, 50U, 1	95.32%
6	full data	one-hot	No	7L, 82U, 0	93.94%
7	eliminated	two-hot	Yes	7L, 81U, 1	95.31%
8	eliminated	two-hot	No	4L, 50U, 1	96.29%
9	eliminated	binary	Yes	7L, 81U, 1	94.87%
10	eliminated	binary	No	4L, 82U, 0	96.38%
11	eliminated	one-hot	Yes	7L, 75U, 0	95.65%
12	eliminated	one-hot	No	7L, 50U, 0	94.35%

Table 5: The best models' results, the architecture column's form is xL, yU, 0 or 1; x is the number of layers, y is the number of units, and the last flag is 1 if the our wieghted loss function is used, 0 otherwise; x is the number of layers, y is the number of units, and the last flag is 1 if the loss function is wieghted, 0 otherwise.

After analysing all models, we have noticed that Bi-LSTM models always outperforms LSTM models significantly.

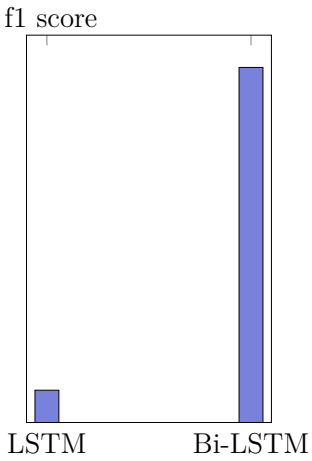


Figure 6: This is the average f1 score of 100 different models.

All results in the table 5 is from Bi-LSTM models. The best full data models, with diacritics is model 3 and without diacritics model 2.

Class	Model 2	Model 3
Wafeer	95.89%	96.19%
Monsareh	89.48%	90.32%
Madeed	81.28%	82.79%
Mogtath	84.52%	87.77%
Motakarib	96 %	95.1 %
Kamel	96.74%	96.37%
Taweel	97.81%	98.29%
Sar'e	90.17%	91.1 %
Raml	92.98%	92.79%
Rigz	86.11%	85.8 %
Khafeef	96.59%	96.63%
Baseet	98.03%	97.97%
Moktadib	68.37%	62.24%
Hazg	77.79%	78.31%
Modar'e	20.83%	33.33%
Motadarik	78.86%	77.52%

Table 6: The accuracy per class for Model 2 and 3.

XI.II Encoding effect

We can compare the three encoding techniques according to two comparison aspects; the first is their effect on the learning curve and the second is their memory consumption which what we will start with. The following figure shows how much memory an encoded verse consumes. It is clear that the One-Hot encoding needs 4.2 times and 22.4 times more than the Two-Hot and the Binary, respectively. And the Two-Hot encoding needs memory 5 times more than the Binary, but difference is not as huge as in the One-Hot, which in turns will have an effect on the computational power and the training time, so they need be chosen very carefully.

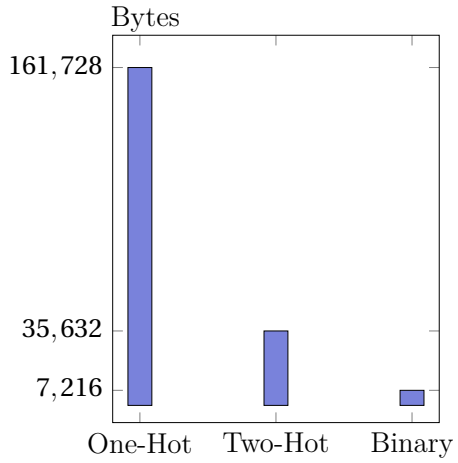


Figure 7: The memory space needed to encode on Arabic verse using the three encoding techniques.

This was the first aspect, the second aspect is the encoding effect on the learning curve. On average, the three encoding methods almost achieve the same results but with different architectures. From figure 8, the best models could learn faster when the data is represented by One-Hot and Two-Hot encoding; which makes a great sense, because both encoding techniques share a property, they do not have common features in representing the letters, or at least Two-Hot has at most only one common features; letters may share the same diacritic state, so the data representation was so clear for the models. This is not the case in the Binary encoding, the best models' learning was slower than the previous two because of the common feature problem IX.II so models 3, 4, 9 and 10 from the table 5 need to be deeper to distinguish between the letters first.

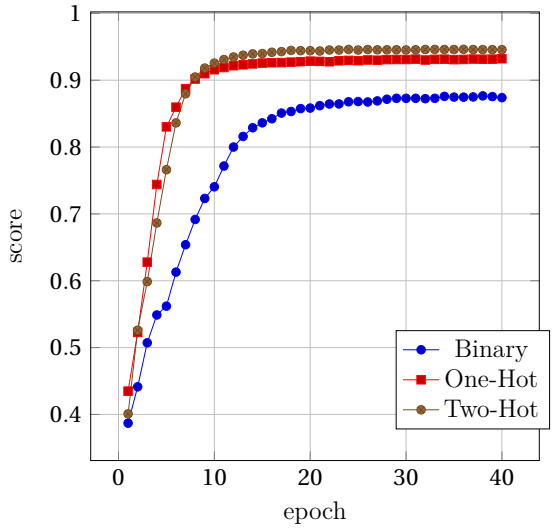


Figure 8: Each point in this chart is the average of the validation accuracy on the best models at epoch i; we have performed the same process three times by changing one factor, the encoding.

Diacritics

Diacritics give more information about the pattern. On average, diacritic models outperforms the no-diacritic models, but from the result table 5, model 1 and 2 which are Two-Hot diacritic and no-diacritic, respectively, reached the same results. But in other encoding methods, diacritic models outperforms the no-diacritic models.

XI.III Imbalanced Dataset effect

As char 1 shows, the dataset is imbalanced, big classes have enough data to train the model well, while small classes do not; small classes' size is less than 18,000 data points. To handle this situation we have presented two methods: weighting the loss function and eliminating the small classes. For weighting the loss function, it does not effect the results as we previously thought. The following table contains 4 models.

Class	Model 1	Model 2	Model 3	Model 4
Wafeer	94.64%	95.98%	96.19%	95.76%
Monsareh	86.79%	89.48%	90.32%	88.7%
Madeed	78.01%	81.28%	82.79%	81.78%
Mogtath	80.09%	84.53%	87.77%	83.76%
Motakarib	94.18%	96 %	95.1 %	94%
Kamel	95.74%	96.74%	96.37%	95.84%
Taweel	97.61%	97.81%	98.29%	97.27%
Sar'e	86.73%	90.18%	91.1 %	90.33%
Raml	92.54%	92.98%	92.79%	91.13%
Rigz	85.23%	86.11%	85.8 %	83.22%
Khafeef	95.58%	96.59%	96.63%	96.24%
Baseet	97.64%	98.03%	97.97%	97.63%
Moktadib	52.04%	68.36%	62.24%	59.81%
Hazg	78.57%	77.79%	78.31%	72.73%
Modar'e	0	20.83%	33.33%	16.67%
Motadarik	70.67%	78.86%	77.52%	78.48%

Table 7: Model 1 and 2 share the same architecture (7 layer, 50 units) but the model 1's loss function is weighted and model 2's is not. Also, model 3 and 2's architecture is (7 layers, 81 units), model 3's loss function is wieghted and model 4's is not.

The second method, is to eliminate the small classes, the eliminating purpose is not balancing the dataset; but our purpose is to study the effect of existing such the small classes on the performance. We have noticed that eliminated method needs at least 82 units so that it increases the accuracy.

XI.IV English Results

id	**encoding**	**cell type**	**f1 test**
1	one-hot	GRU	81.35%
2	one-hot	LSTM	80.34%
3	binary	LSTM	75.43%
4	binary	GRU	75.04%

Table 8: Best performance models

Generally, GRU models outperform LSTM and Bi-LSTM models. The model which shows the highest performance is in the experiment 1 in the

previous table, f1 = 0.8135. The model consists of 7 hidden layers, 40 units per each layer, 0.1 dropout and 128 batch size.

On average, the highest results are achieved by the one-hot representation not by the binary; this is because in one-hot encoding each character is represented by a one-hot vector which have no common features between letters, this is not the case in the binary encoding, in which each letter is represented by a unique binary combination, so each letter may share some common features in its representation with the others, which in turn confuses the model, it needs more complex and deeper network to achieve results similar to what one-hot achieves due to the common feature problem IX.II.

	Iambic	Anapaestic	Trochaic	Dactyle
Iambic	0.8998	0.045	0.0529	0.00189
Anapaestic	0.0919	0.884	0.01838	0.00551
Trochaic	0.34	0.0375	0.6196	0
Dactyle	0	0.014	0	0.9859

Table 9: The best model's confusion matrix

The above confusion matrix is of the best model, experiment 1 in the table 8, which we have presented in the beginning of this section. Row are the real classes and columns are the predictions. There are some notes we can derive from it like; *Dactyl* class is hardly misclassified whether false-positive or false-negative; this is because *Dactyl* verses are hexameter which is relatively longer than the three remaining classes, they are pentameter or less; so usually any long verse is classified as *Dactyl*. To study the that case, we have dropped the relatively long-verse *Dactyl* and trained the best model (experiment 1), we have got f1=78.8%, which is less than the experiment 1 f1, this is because of the lack of data. Our English dataset is so small, as it discussed in the dataset section.

XII Future Work

In this paper, we introduced several ways and methods to classify Arabic and English Poems. We provide some methods and techniques to work with such problem based on the character level and analyze the text concerning musical way. Also, we have published our datasets to be open source for the community to encourage the community of research into Artifical intelligence to continue from our original works here. We believe the problem and our datasets can be used in the below future works.

- Enhance the classification results to be same as the human expert.
- Use the current datasets to classify the poem meaning as this paper did not work for this idea.
- Analyze the historical impact based on the Poem and the Poetry for example for a specific period Is the Poem affected by this period, or there are patterns of writing between the Poetry or not.
- Can we generate from each class some poem similar to the poetry poems?

Bibliography

[1] Belal Abuata and A Al-Omari. A Rule-Based Algorithm for the Detection of Arud Meter in Classical Arabic Poetry. *researchgate.net*. URL https://www.researchgate.net/profile/Belal_Abuata/publication/298199192_A_Rule-Based_Algorithm_for_the_Detection_of_Arud_Meter_in_Classical_Arabic_Poetry/links/5790817108ae64311c0ff2fd.pdf.

[2] Manex Agirrezabal, Iñaki Alegria, and Mans Hulden. A Comparison of Feature-Based and Neural Scansion of Poetry. *Ranlp*, 2017. URL https://arxiv.org/pdf/1711.00938.pdf.

[3] Abdulrahman Almuhareb, Waleed A Almutairi, Haya Altuwaijri, Abdulelah Almubarak, and Marwa Khan. Recognition of Modern Arabic Poems. 10(4):454–464, 2015. ISSN 1796217X. doi: 10.17706/jsw.10.4.454-464.

[4] Mohammad a Alnagdawi, Hasan Rashideh, and Ala Fahed. Finding Arabic Poem Meter using Context Free Grammar. *J. of Commun. & Comput. Eng.*, 3(1):52–59, 2013.

[5] Sepp Hochreiter and J Urgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://www7.informatik.tu-muenchen.de/{~}hochreit{%}5Cnhttp://www.idsia.ch/{~}juergen.

[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(6):948–963, 2012. ISSN 13000632. doi: 10.3906/elk-1010-899.

[7] Kedar Potdar, Taher S., and Chinmay D. A Comparative Study of Categorical Variable Encoding Techniques for Neural Network Classifiers. *International Journal of Computer Applications*, 175(4):7–9, 2017. ISSN

09758887. doi: 10.5120/ijca2017915495. URL http://www.ijcaonline.org/archives/volume175/number4/potdar-2017-ijca-915495.pdf.

[8] Margento Chris Tanasescu, Bryan Paget, and Diana Inkpen. Automatic Classification of Poetry by Meter and Rhyme. *Florida Artificial Intelligence Research Society Conference*, 5, 2016. URL https://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS16/paper/view/12923.

[9] H. R. Tizhoosh and R. A. Dara. On poem recognition. *Pattern Analysis and Applications*, 9(4):325–338, 2006. ISSN 14337541. doi: 10.1007/s10044-006-0044-8.