Cairo University
Faculty of Engineering

Department of Computer Engineering

NLP Project

Arabic Text Diacritization

Team 10

Name	ID
Eslam Ashraf	9202256
Beshoy Morad Atya	9202405
Abdelrahman Hamdy	9202833
Abdelrahman Noaman	9202851

Date: 12/30/2023

Presented to: Eng. Omar Samir

Project Pipeline

1.Preprocessing

We have tried various techniques to efficiently preprocess the data.

- **Data Cleaning:** First step is always to clean the sentences we read from the corpus by defining the basic Arabic letters with all different formats of them they encountered 36 unique letter, the main diacritics we have in the language and they encountered 15 unique diacritic, all punctuations and white spaces. Anything other than the mentioned characters gets filtered out. Therefore we're sure that the data is free from any undesired characters which are irrelevant for the next step like any English characters or numbers as specified.
- Then we'll extract the valid Arabic text including the diacritics but without the punctuations from the cleaned sentences.
- **Tokenization:** The way we found yielding the best result is to divide the corpus into sentences of fixed size (A window we set with length of 1000) which means that if a sentence exceeds the window size we will go backward until the first space we will face then this will be the cutting edge of the first sentence, and the splitted word will be the first one in the next sentence and keep going like this. If the sentence length is less than the window size then we pad the rest of the empty size to ensure they're all almost equal.
- Now we have a list of sentences where the max length of each is the window size we have specified earlier. Our goal now is to **separate letters and diacritics**. We do this by looping over each sentence and checking for the character placements if any belongs to a diacritic list we define as well (contains all possible diacritics basically). We also take into account if a character has a double diacritic above it, therefore we concatenate both diacritics into one and append them to the list. In the end we'll end up with 2 lists. One contains all characters of each sentence and the other contains the corresponding diacritics for these characters.
- The last step is to **encode each character and diacritic** to a specific index which is defined in our character to index and diacritic to index dictionaries. Basically transforming letters and diacritics into a numerical form to be input to our Neural Network later on.
- **Failed Trails**: We Tried not to give it a full sentence but a small sliding window and this sliding window in flexible in size as we can determine the size of previous words we want to get and the size of the next words, so we will apply the previous operations using that sliding window that iterates over each word and get the previous K words and also the next N words so that the size of the

sliding window will be K+N+1(word itself) so we will be able to determine the context the word has come in, but finally we found that it doesn't increase the accuracy so we decided to neglect it.

- Also we tried to do the same but by splitting on the punctuations to get each single sentence we have as we won't split here by the size but by the context of sentence itself

2.Feature Extraction

Features we trained:

- Our first attempt was to search about feature extraction techniques and python libraries that trains data for feature descriptors and we first tried to implement the **Bag Of Words** by using the CountVectorizer method which is trained on the whole corpus. The vectorizer gets the feature names after fitting the data and then we save them to a csv file which represents the bag of words model. We can index any word and get its corresponding vector which describes its count in the corpus and no info about the position of this word.
- Another attempt was **word2vec** and **Fasttext** word embeddings.
    - From the Fasttext model provided by gensim, we were able to extract word embeddings from each word in the corpus.
    - We first specify the embedding size we want from the model, the window it considers around the word which is set by the window size, the down sampling factor, and any other parameters such as the number of epochs when training.
    - Next, we pass the data loaded to build the vocab and the model knows the data it will train on. The last step is the training itself which will give us the fasttext model to be saved and loaded later on for retrieving embeddings of a certain word.
    - From this model, it's very easy to get these embeddings which is done by just indexing them. We can also calculate similarities between words or get the most similar words to a given one.
- We used the **TF-IDF** as another representation for each word by using a similar process to the bag of words.
    - The TfIdfVectorizer initializes our model and we choose to turn off lowercasing the words. After transforming and fitting the model on the input data, we extract the feature names out and this will be out words set that we'll place them in column headings of each column in the output csv file.
    - Then, get the vectors from the model for all unique words we have in the dataset and write them in the csv file cells where the row represents the

sentence index so that we know this particular word in the column has a TF-IDF value for each corresponding sentence.
    - This is written using the pandas data frame and used later to encode each word with the number from the csv file.
- One last feature we implemented but didn't use is **ngrams**. We basically choose a window (for example 3) and having every sentence, we place a window of size 3 on each index and the output is a list of tuples where each tuple has the 3 letters in this window and the diacritic of the last letter. We thought this would be useful to experiment with how each character depends on the surrounding letters in terms of the diacritic placed on it but didn't find it useful because there was no way to represent it numerically and account the tuple for weights in our network.

- The above approaches weren't possible on pure Arabic letters because these libraries tokenize on English statements. They expect the data to be joined sentences in English form so we had to find a way to deal with this issue. After a bit of research, we found a method that basically maps Arabic letters and diacritics to English letters and symbols by using **Buckwalter** transliterate and untransiliterate functions, we were able to switch the language for the feature extraction by ourselves part.

- **Trainable Embeddings:** Here we use the Embedding layer provided by torch.nn which gives us trainable embeddings on the **character level**.
    - ✓ This layer in a neural network is responsible for transforming discrete input elements, in our case character indices, into continuous vector representations where each unique input element is associated with a learnable vector and these vectors capture semantic relationships between the elements.
    - ✓ We give it our vocabulary size and the desired embedding dimension we want. During training, the parameters of this embedding layer (embedding vectors) will be learned and updated based on the optimization process.
    - ✓ The model will adjust the embeddings to capture useful patterns or representations for the task at hand. The output of this layer would be the learned dense representations (embeddings) for each character in our vocabulary.

✓ **Pretrained embeddings:**

- <u>AraVec CBOW Word Embeddings</u>
  - AraVec is an open-source project that offers pre-trained distributed word representations, specifically designed for the Arabic natural language processing (NLP) research community.
  - In its initial release, AraVec presented six distinct word embedding models, created from three different Arabic content domains: Tweets and Wikipedia.
  - This paper outlines the resources employed, data cleaning methodologies, preprocessing steps, and the techniques used for generating word embeddings in the first version.
  - In the third iteration of AraVec, the project extended its offerings to include 16 diverse word embedding models, now spanning two Arabic content domains: Twitter tweets and Wikipedia Arabic articles. A notable enhancement in this version is the introduction of two model types—unigrams and n-grams models.
  - The generation of n-grams involved the application of various statistical techniques to capture commonly used n-grams within each data domain.
  - The combined dataset comprises more than 1,169,075,128 tokens.
- <u>AraBERT Contextual Embeddings</u>
  - Pre-trained transformer for Arabic Language Understanding

3. Model Training

All Approaches we tried

- *Tensorflow:*

➢ **RNN**

- Model is defined as Sequential()
- Pre-trained Fasttext Word Embeddings layer
- LSTM layer with 100 units
- Dense layer with softmax activation function

- *We used PyTorch to initialize and train our main models.*

➢ **1 Layer Bidirectional LSTM**

- Embedding Layer (Trainable) [300]

- Bidirectional LSTM layer [300, 512]
- Fully Connected Linear layer [2 * 512, classes=15]

➢ **CNN**
- Embedding Layer [300]
- Convolutional layer [300, 256]
- 2 Bidirectional LSTM layers [256, 512]
- Dropout layer with probability 0.5
- Linear layer [2 * 512, classes=15]

➢ **LSTM + CRF**
- Embedding Layer [300]
- 2 Bidirectional LSTM layers [300, 512]
- Fully Connected linear layer [2 * 512, 15]
- Dropout layer with probability 0.5
- CRF layer

➢ **3-Layer BiLSTM**
- Embedding Layer (Trainable) [300]
- 3 Bidirectional LSTM layers [300, 512]
- Fully Connected Linear layer [2 * 512, classes=15]
- Dropout=0.4

➢ **5-Layer BiLSTM**
- Embedding Layer (Trainable) [200]
- 5 Bidirectional LSTM layers [200, 512]
- Fully Connected Linear layer [2 * 512, classes=15]
- Dropout=0.4

➢ **CBHG Model**
- Embedding Layer [512]
- Pre-net Layer [512, 256]
- CBHG:
  - BatchNorm 1D Convolutional Banks
  - 1D Max pooling layer
  - ReLU Activation layer
  - Batch Normalized 1D Convolutional layer projections

        - Pre-highway (Linear) Layer
        - Module list of Highways, they basically consist of
            1. 2 Linear layers
            2. ReLU
            3. Sigmoid Layer
        - Bidirectional GRU (Gated Recurrent Unit) Layer
    - A Module list of bidirectional LSTM layers
    - Linear and Batch Normalization layers in the end

For the CNN and CRF models, we have also tried to use pre-trained embeddings instead of the trainable character level embeddings layer in the beginning.

The preprocessing differs here as we no longer have each element in X_train as a complete sentence of window size as explained above. The input to our model now is on the word level but each character is encoded by an index. Each char is grouped together representing every word by itself (Not sentence)

We have attempted:

- **CBOW word embeddings from Aravec with CRF**
    - ✓ https://github.com/bakrianoo/aravec
- **Contextual Embeddings from AraBERT with CNN**
    - ✓ https://github.com/aub-mind/arabert

These models were trained and DERs were computed.
Overall, we have tried way too many features and ended up using the Trainable embeddings which basically yielded the highest accuracy for all models.

Evaluation

CRF with AraVec = 0.085
AraBERT with CRF = 0.045
2-LSTM: DER = 0.035
CNN with 2-LSTM: DER = 0.024
3-layer BiLSTM: DER = 0.021
CRF with BiLSTM: DER = 0.025
CBHG: DER = 0.024
5-layer BiLSTM: DER = 0.013

Final Model Used

5-Layer Bidirectional LSTM