# **Arabic Sentiment Analysis using NLP & DNNs**

**Team number: 95**

**Highest Kaggle Accuracy: 0.84939 using Embedding and LSTMs**

## **Project Description**

The dataset contains approximately 30,000 reviews written mostly in Arabic, the reviews are labeled -1, 0 and 1 denoting negative, neutral and positive sentiments respectively. The task is to build an effective DNN that can classify and analyze the sentiment of an unlabeled review as accurately as possible.

## **Preprocessing**

Preprocessing is the first step in the project, in this step the text is cleaned and prepared for feature extraction.

Prior to preprocessing, the data is split into a training set and a validation set that is only 1000 samples to keep tack of the DNNs generalization capabilities.

The dataset contains mostly Arabic entries, but there also exists a minority of English reviews, as a solution to this, language detection is applied, and a different NLP pipeline is used to preprocess each language independently.

**NLP Preprocessing pipeline:**

**For English Text:**

1. Punctuation is removed.
2. Text is transformed to lower case.
3. Text is tokenized using NLTK’s word\_tokenize().
4. Stopwords are removed using the set of stopwords provided by NLTK.
5. Individual tokens are lemmatized using WordNet lemmatizer to get the origin of words.

**For Arabic Text:**

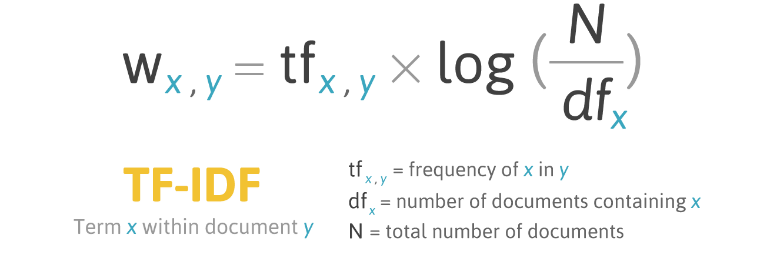
1. Extra whitespace is removed.
2. Elongation (when one letter is repeated more than twice) is removed.
3. Letters are normalized and repetitions are removed (for example أ ->ا )
4. Tashkeel and tatweel are removed.
5. Punctuation is removed.
6. Replace rare words and expressions with common synonyms (for example ما شاء الله-> ممتاز) (experimental)
7. Replacing emojis with relevant words (for example 😍 -> ممتاز)
8. Removing Stopwords except negation words like (لا, لم) because they change the meaning of the sentence. (Stopwords removal was aborted because it is seen that the model performs better with the stopwords existing and not removed).
9. Text is tokenized by splitting on white space, then finally it is stemmed using one of three stemmers, Snowball stemmer, ISRIS stemmer and finally the stemmer in Stemmer library.

## **Feature Extraction**

Feature extraction involves transforming the text data into a numerical data that is suitable to train models on.

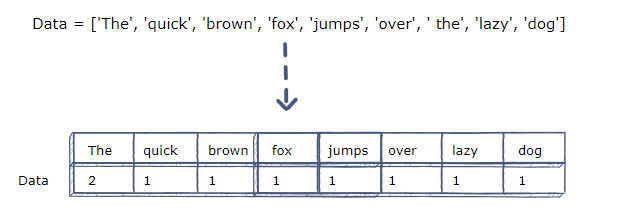
**Feature Extraction Techniques:**

**TF-IDF Vectorizer**



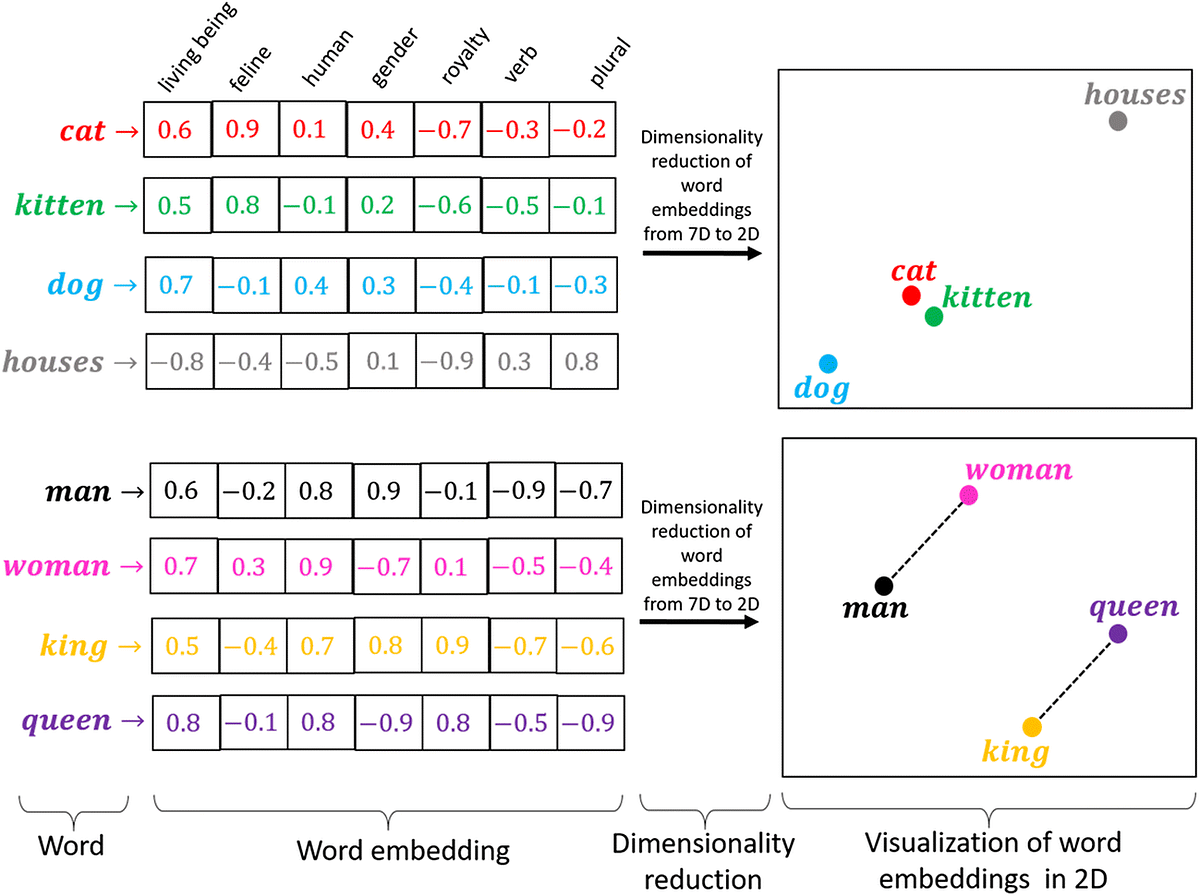
TF-IDF stands for Term Frequency-Inverse Document Frequency. It's a numerical statistic used to reflect how important a word is to a document in a collection or corpus. The TF part measures the frequency of a word in a document, while the IDF part measures the inverse of the word's frequency across the whole corpus, which helps to adjust for the fact that some words appear more frequently in general. TF-IDF increases with the number of times a word appears in a document but is offset by the frequency of the word in the corpus, which helps to differentiate common words from those that are more specific to the document.

**Count Vectorizer**

****

Count Vectorizer is a method used in text analysis to convert a collection of text documents into a matrix of token counts. It simply tallies the occurrence of each word within a document, treating every term as a feature and the count as its value. This approach is straightforward and efficient for converting text into a numerical format for machine learning algorithms, but it doesn't account for the relative importance of words in the documents or across the corpus, unlike methods such as TF-IDF.

**Word Embedding**



Word embedding is a type of word representation that allows words with similar meaning to have a similar representation. It is a distributed representation for text that is perhaps one of the key breakthroughs for the impressive performance of deep learning methods on challenging natural language processing problems. Word embeddings are learned from data and are based on the context in which words appear. Words that occur in similar contexts are embedded closely together in the vector space, capturing semantic and syntactic meanings. This allows for capturing subtle nuances and relationships between words, making it a powerful feature for many natural language processing tasks.

Embedding was done in this project through the following steps:

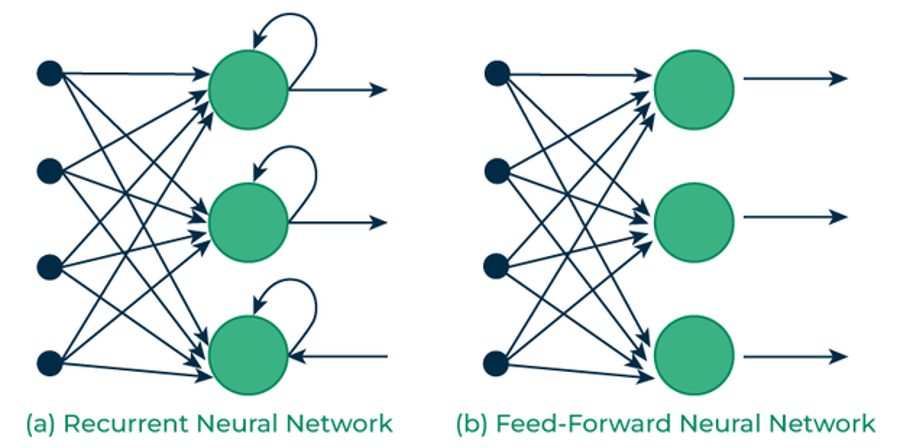
1. Text is tokenized into individual tokens.
2. The stream of tokens is transformed into a sequence of numbers that are padded to a certain length (max length).
3. The sequence is fed into a Keras embedding layer for the model to learn the word embedding from scratch.

## **Models Training and Evaluation Trials**

In this phase, different models and architectures were experimented with to find out which one performs the best.

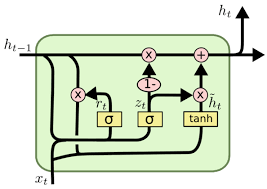
**About the models:**

**Simple RNNs**

****

A simple Recurrent Neural Network (RNN) is a type of neural network that is designed to recognize patterns in sequences of data, such as text, genomes, handwriting, or numerical time series data. Unlike feedforward neural networks, RNNs have "memory" in that they use information from previous inputs to influence the current output. In a simple RNN, each neuron takes input not only from the previous layer but also from itself from the previous time step, allowing it to maintain information across inputs. However, simple RNNs can struggle with long-term dependencies due to issues like vanishing gradients, where the influence of information decreases exponentially over time.

**GRUs (Gated Recurrent Unit)**



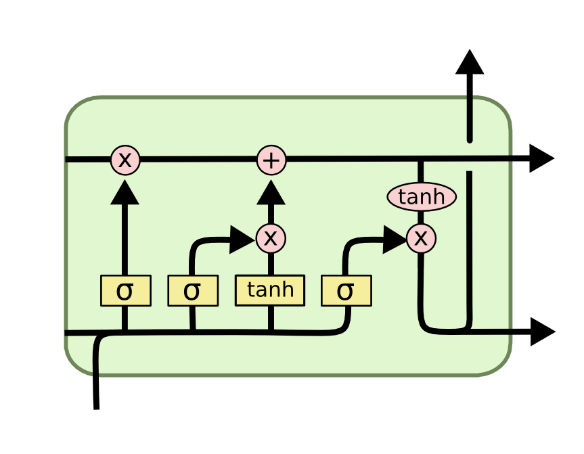
Gated Recurrent Units (GRUs) are an advanced type of RNN that aim to solve the vanishing gradient problem encountered by simple RNNs. They achieve this by using gating mechanisms to control the flow of information. Each GRU cell has two gates:

**Update Gate:** This gate decides how much of the past information needs to be passed along to the future. It helps the model to determine the level of importance of the information from the past and how much of this information should be carried to the next state.

**Reset Gate:** This gate decides how much of the past information to forget. It allows the model to decide whether the previous state is irrelevant for the current prediction and effectively allows the model to drop any irrelevant information.

GRUs are particularly effective in tasks where you need to capture dependencies for sequences of data, such as language translation, speech recognition, and time-series analysis. They are simpler than their counterpart LSTM (Long Short-Term Memory) cells as they use fewer parameters, but in many cases, they can perform equally well.

**LSTMs (Long Short-Term Memory)**



Long Short-Term Memory networks (LSTMs) are a special kind of RNN capable of learning long-term dependencies. They were introduced to overcome the vanishing gradient problem that affects standard RNNs. LSTMs maintain a separate cell state along with a hidden state, and they use several gates to regulate the flow of information:

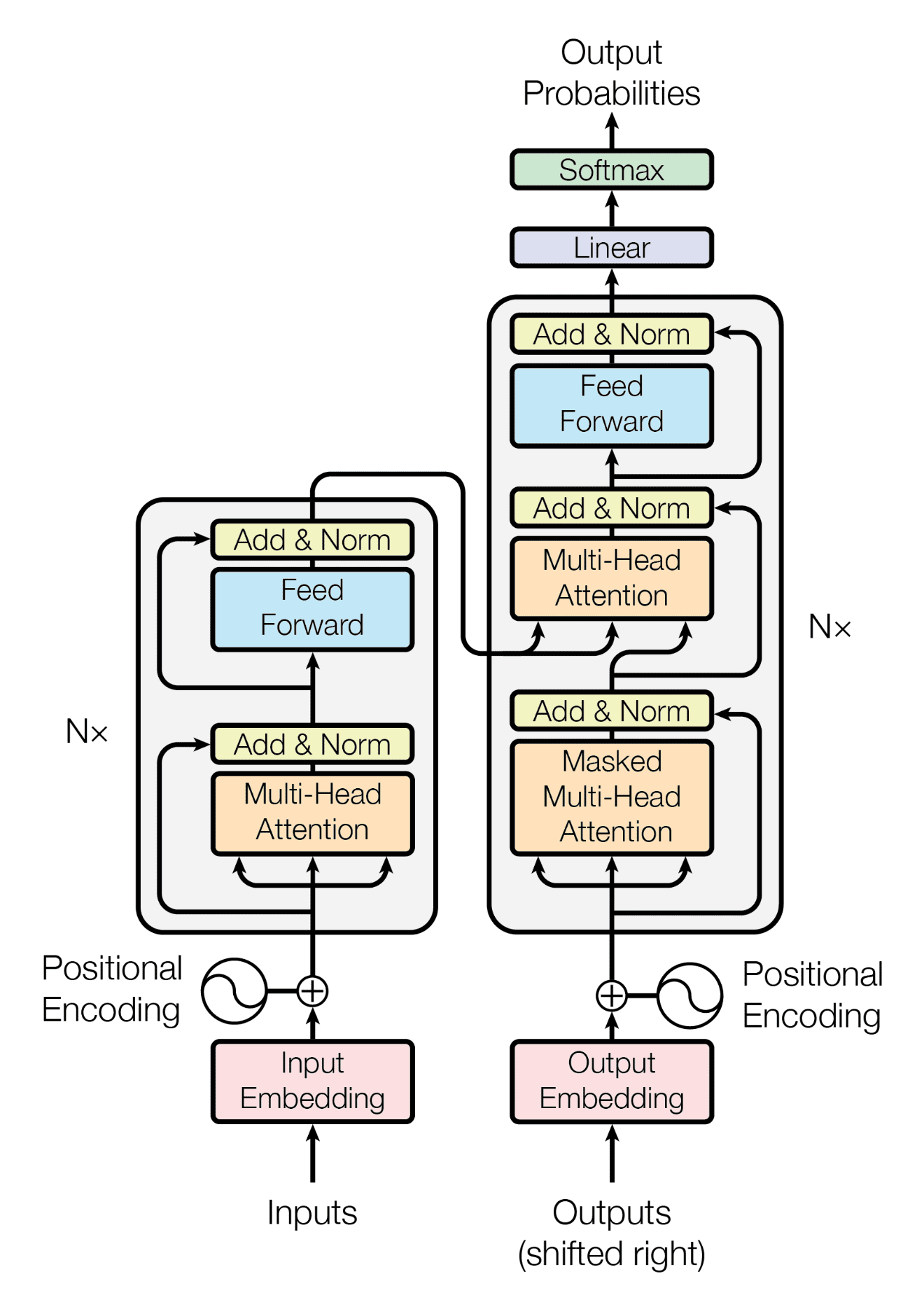
**Forget Gate:** Determines what information is discarded from the cell state.

**Input Gate:** Decides which values are updated in the cell state.

**Output Gate:** Controls what part of the cell state makes it to the output.

These gates allow LSTMs to selectively remember and forget things over long periods, making them highly effective for a range of sequential tasks such as speech recognition, language modeling, and time series prediction. The architecture of LSTMs allows them to capture complex temporal patterns and dependencies, making them a robust solution for problems in sequential data processing.

**Transformers**



Transformers are an advanced neural network architecture that revolutionized natural language processing (NLP) tasks. Introduced in the "Attention Is All You Need" paper in 2017, they are distinguished by their use of the attention mechanism, specifically the multi-head attention mechanism, allowing them to process sequences in parallel and capture complex dependencies.

Key aspects of transformers include:

Multi-Head Attention: This mechanism allows the model to jointly attend to information from different representation subspaces at different positions. With multi-head attention, the attention process is run several times in parallel, with each "head" focusing on different parts of the input sequence, thus enabling the model to capture a richer range of information.

Query, Key, Value Vectors: The attention mechanism in transformers operates using three main components: queries, keys, and values. These are vectors representing different aspects of the input:

Query: Represents the part of the input that is being focused on.

Key: Corresponds to all parts of the input sequence that the query interacts with.

Value: Contains the actual content of the input part that is being attended to.

**Trials:**

Global settings for all trials (found by experimenting to determine the best options)

**General options for all models**

Learning Rate: 0.0001

Batch Size: 32

Epochs: 20

Early Stopping: Enabled with a patience of 3 and monitoring the validation loss to avoid overfitting.

Dropouts: Added dropout layers with probabilities of 0.1 ~ 0.3 to prevent overfitting.

Checkpoints: Checkpoints were added to save the model with the lowest validation loss throughout the training across the epochs.

Hidden layers and Neurons count: In most of the models, 1~2 layers were used with around 32 neurons in each layer, these settings deliver a good combination of low training times and good generalization capabilities.

**TF-IDF related options**

No special settings were used, the default TF-IDF vectorizer options provided the best results.

**Embedding Related options**

Embedding Dimensions: 25

Max Sequence Length: 120

**Transformer Related options**

Number of heads: 5

Number of transformer blocks: 1

Number of feed forward networks: 1

**Results Table**

|  |  |  |
| --- | --- | --- |
| Algorithm | Validation Accuracy | Test Accuracy on Kaggle |
| TF-IDF + LSTM | 82~83% | ~82% |
| TF-IDF + LSTM (Bidirectional) | 82~84% | ~83% |
| TF-IDF + Simple RNN | 82~83% | 81-82% |
| TF-IDF + GRU | 82~83% | 81-82% |
| Embedding + LSTM | 82~85% | 84-85% |
| Embedding + Transformer | 82~85% | 84-85% |

## **Conclusion**

* Arabic preprocessing is a challenging task, with many dialects across the reviews, and lots of miss- spellings and rare synonyms, it is hard to come up with normalization techniques that are well suited to preprocess Arabic data in the best way possible.
* Word embedding provides better results than TF-IDF, likely because it expresses the words numerically in a more nuanced way.
* LSTMs and Transformers deliver the best performance, which is not surprising considering they are more complex and sophisticated than the other models.