Fake News Detection using Natural Language Processing

Mohamed Ayman†
Computer Engineering
The American University in Cairo
mohamedayman15069@aucegypt.edu

Abdelrahman Shaaban
Computer Engineering
The American University in Cairo
abdelrahmanfawzy@aucegypt.edu

1. Introduction

Internet and social media have made the access to news information much easier and comfortable. When people need to follow their event of interest, they just use their phones online. Unfortunately, some challenges have become apparent in disseminating information. More clearly, the prevalence of information on the Internet and social media has made it difficult for users to distinguish between fake and real news. As a result, mass media which has a great influence on society may manipulate information in different ways. It means that people can easily produce mixed news of true and false information or false information completely. This has given motivation that there are several websites generating false information exclusively. Our idea comes here to create a model detecting fake news. Throughout the semester, we could train over different types of machine learning algorithms and found neural network algorithms to be the best in terms of accuracy. We have tried different types of hyperparameters of the network to achieve an acceptable result. We could reach 69% accuracy with our implementation, and 72% accuracy using the Sklearn MLP classifier. To be able to train words in a neural network, we used Skip gram Word2Vec to have numerical representation for each word. We believe that it is a good start to be able to improve the model further in the future. We use web applications as a bonus to clarify our idea of fake news. We have done a naive Frontend for a limited time.

2. Implementation of our Model

2.1 Word To Vector Embedding Layer architecture explanation

Word2vec embedding is the first layer of the model that was implemented to be used to learn the words' embeddings of the titles' corpus. We implemented the Skip-Gram model, which is mainly used to predict the context words for a given target word. It is less sensitive to the words' positions in the sentence, compared to the CBOW model, which is what we need in our model. The implementation of the Skip-Gram is basically a single-hidden-layer neural network that receives an input of one-hot-encoded word and outputs a vector of its most nearby related words.

The first step is to collect the training data that we need to train our neural network. That was implemented by identifying first a window size (w=3), which identifies the number of words to be considered within each word's context. **As shown in figure 1 below**, the training samples are all the words as an input, with each of its context words as an output. By collecting the training samples of 3125 unique words within around 500,000 titles, we got around 4,600,000 training samples for the skip-gram model.

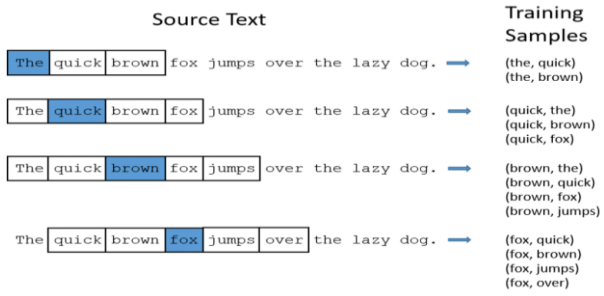

Fig. 1. Explaining the idea of our implemented Embedding Layer.

For the neural network and as shown below in figure 2, we have an input layer of 3125 inputs, a one-hot-encoded representation for the input word, a single hidden layer of 100 neurons with a linear activation function, and an output layer of 3125 outputs with a softmax activation function. The model is trained to learn to map each word to its context words, by getting higher probabilities by the softmax, while the non-context words get very low probability. These probabilities can show well the representation of the corpus words and the relationships between them. Our end goal of training this model is to

WOODSTOCK'18, June, 2018, El Paso, Texas USA

https://doi.org/10.1145/1234567890

learn the 100-hidden-layer weights, to be saved and used by the main neural network and the web application to directly get the embeddings of the input titles and use them for training and testing our classifier.

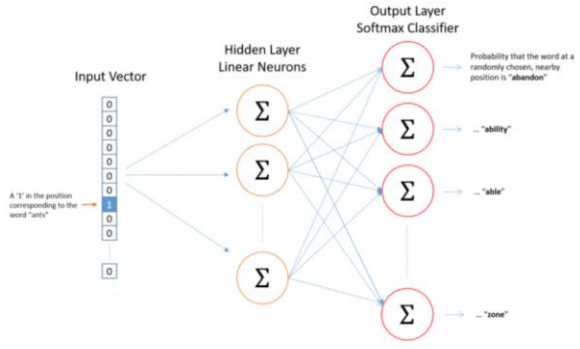

Fig. 2. The architecture of our skip-gram Word2Vec.

This implementation we have, with the small number of words to be trained on, couldn't achieve any good results as shown in figure 3. The embedding couldn't converge well and because of using it to train our classifier, we couldn't expect anything from it. Therefore, we tended to train the genism pre-trained model with the words we have, to obtain good results.

```
In [244]: y_hat

Out[244]: array([[0.90626832],
                 [0.88644152],
                 [0.89461215],
                 ...,
                 [0.88890697],
                 [0.83496279],
                 [0.90836429]])

In [245]: y_predict=np.array(y_predict)
          sum(y_predict)

Out[245]: 133554

In [246]: print(classification_report(y_test,y_predict))

                  precision    recall  f1-score   support

               0       0.00      0.00      0.00     70538
               1       0.47      1.00      0.64     63016

        accuracy                           0.47    133554
       macro avg       0.24      0.50      0.32    133554
    weighted avg       0.22      0.47      0.30    133554
```

Fig. 3. Results of the Manual Embeddings

2.1 Neural Network Classifier architecture explanation

In this final phase, we worked on the implementation of the neural network classifier from scratch. This network basically has an input layer of 100 inputs, as it is fed with each title's embedding vector, and a binary output layer of two outputs, which represent the probability of being fake news versus being real news. In this output layer, we can use sigmoid as an activation function, but we used softmax instead as it was much easier in implementation. Not only this, but the other reason is that according to our training, we have found that softmax behaves much better than sigmoid. From our thorough research, we have found that softmax is derived from sigmoid, so we believe that there is no problem using softmax as our output activation function with 2 neurons instead of 1 neuron with the sigmoid, which is supposedly the same thing. Regarding the hidden layers and the model's complexity, we first implemented a one-hidden layer network, which resulted in underfitting. By experimenting with different random architectures, we decided to consider the following architecture to be ours in implementing this network, as we found that it is the best one that can handle the large text dataset we have and learn insights and patterns from it that helps better the classification results.

Our neural network consists of three hidden layers, with the first layer including 50 neurons, the second one including 25 neurons, and the third one including 4 neurons. We found Leaky-Relu activation function is the best to be used as the embedding vectors have negative numbers, and we understand that normal relu discards any negative number to zero which will negatively affect our data. In this sense, it is important to use an activation function that doesn't discard any negative numbers and doesn't have any zero-slope parts, such as leaky-relu. Regarding the loss function, and as the output layer uses the softmax activation function, we use the softmax log-likelihood function. The learning rate is 0.25 with no optimizers used although Adam is implemented, we didn't obtain good results by using it. The model made 200 epochs going through the training and the validation data and showing no overfitting problem. By implementing and training this model, we could get an accuracy of 69% compared to 72% obtained by the Sk-learn MLP classifier. All the results are shown below.

```
                precision    recall  f1-score   support

             1       0.65      0.75      0.70     63133
             2       0.74      0.63      0.68     70421

      accuracy                           0.69    133554
     macro avg       0.69      0.69      0.69    133554
  weighted avg       0.70      0.69      0.69    133554
```

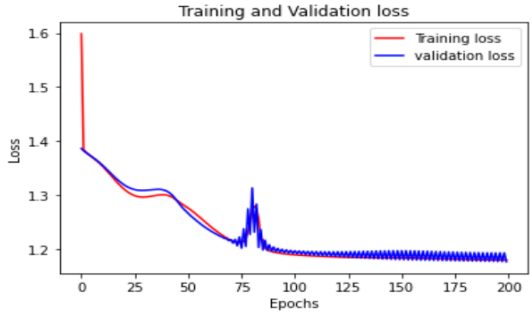

Fig 4. Results of our Model

3. The Web application

The web application has two parts: the backend and the Frontend. As detailed in the last phase, we are using a neural network to detect the authenticity of news while making use of the Word2Vec as a way for converting the words to numerical representations while considering the semantics between the words.

3.1 The Backend

The backend is simply accountable to preprocess and predict the data. In other words, it takes the input string from the frontend, preprocesses it, predicts it using the forwarding propagation to the pre-trained neural network, and returns it back to the frontend. This occurs in the view part in the app called Neural-Network.

3.1.1 The Preprocessing Step

First, we need to clean the string from any sort of commas, unnecessary characters, stopping words, and non-standard language words. Then, we needed to tokenize the string using nltk.word_tokenize(). Afterward, we need to stem the words to get their roots. Finally, we need to do Word2Vec embedding on the stemmed words. To be able to do embedding on the input words, we need the saved Word2Vec model from our implemented words only. In this sense, we expect that there are unknown words we may get from the user, so we ignore them.

```
def tokenization(self, x):
    return x.split()

def stemming(self, x):
    stems = []
    porter = PorterStemmer()
    for t in x:
        lemma=str(porter.stem(t))
        if lemma == '-PRON-' or lemma == 'be':
            lemma=t
        stems.append(lemma)
    return stems
def make_to_base(self, x):
    x_list=[]
    nlp= spacy.load('en_core_web_sm')
    doc=nlp(" ".join(x))
    for token in doc:
        lemma= str(token.lemma_)
        if lemma == '-PRON-' or lemma == 'be':
            lemma=token.text
        x_list.append(lemma)
    return x_list

def preprocessing(self, W2V_model):
    self.title= self.cont_to_exp(self.title)
    self.title= self.title.lower()
    self.title= " ".join([t for t in self.title.split() if t not in STOP_WORDS])
    print(self.title)
    self.title= self.tokenization( self.title)
    self.title= self.stemming(self.title)
    self.title= self.make_to_base(self.title)
```

Fig 5. Preprocessing code in Backend of the Web Application

3.1.2 The Prediction Step

After loading the saved weights, we need to get the resulting numerical representations for prediction. We simply use forward propagation to predict whether the user input is fake or not.

```
def Forward_Prob(Xt,Wts,NN):
    # we need to store W,b,inputs,outputs,activations for each layer for each node
    parameters=[]
    Ai=Xt #current layer output
    #print("SHAPE", Xt.shape)
    Ap=Ai #previous layer output (current layer input)
    for i in range(1,(NN)-1):
        W,b=Wts['W'+str(i)],Wts['b'+str(i)]
        Ap=Ai
        Z=np.dot(W,Ap.T)+b
       # print("W",W.shape)
       # print("AP", Ap.T.shape)
       # print("Z: ", Z.shape)
        actv=Relu(Z)
        Ai=actv.T
       # print(Ai.shape,Ap.shape,actv.shape)
        #print(Wts['b'+str(i)])
        parameters.append((W,b,Ap,actv,Z))
    W,b=Wts['W'+str((NN)-1)],Wts['b'+str((NN)-1)]
    #print("W",Ai.shape)

    Ap=Ai
    Z=np.dot(W,Ap.T)+b
    actv=Sigmoid(Z) #np.exp(Z) / np.sum(np.exp(Z), axis=0)#
    Ai=actv.T
   # print(Ai.shape,Ap.shape,actv.shape)
    #print(Wts['b'+str(i)])
    parameters.append((W,b,Ap,actv,Z))
    Ao=Ai #output of last layer=last Ai
    #print(len(parameters))
    return Ao
```

Fig 6. Prediction code in Backend of the Web Application

3.2 The Frontend

We implemented our web interface using Django Framework. The hierarchy of the project is simply a server and several applications as user interfaces (The detail is in the architecture of Django). Fortunately, Django can create the server itself. This server contains the database, as Django is suitable for backend and frontend, some sub-URLs called endpoints to be able to go from a specific URL to another one, and so on.

3.3 The APIs

We use Django in the backend as well since it is suitable for the frontend and backend with suitable APIs.This simply takes the input from the client using a **POST** request. The preprocessing and predicting data occurs in the backend. Afterward, the results are returned to the frontend using a **GET** request. We did not use any sophisticated APIs due to the limited time.

3.4 The Data Flow in the web application

The Fakeddit dataset needs to be preprocessed somehow to be able to train the weights of the Neural network. In the second phase, we preprocessed the dataset as follows:

1. We dropped the unnecessary columns with discernible justification as mentioned in other phases.
2. We removed all special characters and punctuations.
3. We needed to stem all words to their originality for the sake of decreasing the number of words.
4. We removed all stopping words

5. Finally, we made a threshold to remove all rare and frequent words.

From figure 7, we can see that we put the preprocessed dataset in terms of arrays in Word2Vec Embedding, coming out with a complete numerical representation for each unique word. Now, the words in terms of numerical representation are ready to be considered as input to the model to be trained. After training and validating, the pre-trained weights are saved in the database for testing later.

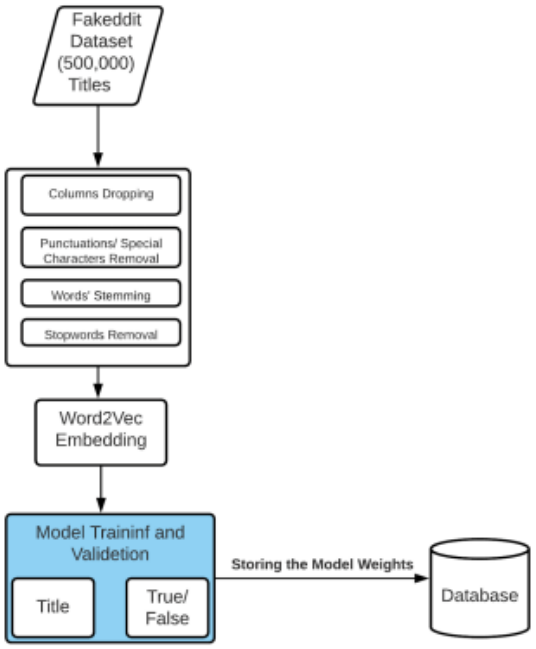

Fig 7. The Data flow of our model in the web application

3.5 The Bonus Part of Retraining Method

Unfortunately, we could not fully implement the whole bonus, but we have done most of it. First, the backend should work. We simply need to validate from the user whether it is a valid generated output from the model or not. If it is not a valid output, we need to backpropagate the network to be able to update the weights. Then, we will save the weights in the disk. Here is the thing, we could implement the frontend and the backend, but the problem is in the communication between the frontend and backend. We could have solved it, but the limited time led us not to be able to solve it.