Machine Learning Engineer Nanodegree - Capstone Project
George Traskas
16 June 2020

Fake News Detector

An end to end solution in AWS

I. Definition

Project Overview

The term 'fake news' regularly hits the headlines nowadays. Whether it is to do with political events or information on the various social platforms, it seems like it is getting harder and harder to know who to trust to be a reliable source of information. This project is about understanding of what is actually fake news and propose an efficient way to detect them using the latest machine learning and natural language process techniques.

In more detail, in this project I developed a machine learning program to identify when a news source may be producing fake news. I used a corpus of labeled real and fake new articles to build a classifier that can make decisions about information based on the content from the corpus. The model focus on identifying fake news, based on multiple articles originating from a source. Eventually, the model can predict with high confidence that any future articles from that source will also be fake news. Focusing on sources widens our article misclassification tolerance, because we will have multiple data points coming from each source.

The intended application of the project is for use in applying visibility weights in social media. Using weights produced by this model, social networks can make stories, which are highly likely to be fake news less visible.

The project's desired results to build are:

- a model in which text can be provided as input and
- predict if it is fake or true.

Specifically, in my study, I investigated:

- The performance of a fast traditional machine learning model, such as Naive Bayes and used this as a Proof of Concept to move on a second analysis in AWS.
- The performance of the AWS BlazingText machine learning algorithm, which then I deployed and put it on work building a simple web page application, where a user can input some text and get the prediction if this news text is fake or true.

Datasets

A dataset from Kaggle was used as a training, validation, and test input for the algorithms that were used. The dataset features a list of articles, together with the subject of the article and its title categorised as 'Fake' or 'True'.

Acknowledgements

Ahmed H, Traore I, Saad S. "Detecting opinion spams and fake news using text classification", Journal of Security and Privacy, Volume 1, Issue 1, Wiley, January/February 2018.

Ahmed H, Traore I, Saad S. (2017) "Detection of Online Fake News Using N-Gram Analysis and Machine Learning Techniques. In: Traore I., Woungang I., Awad A. (eds) Intelligent, Secure, and Dependable Systems in Distributed and Cloud Environments. ISDDC 2017. Lecture Notes in Computer Science, vol 10618. Springer, Cham (pp. 127-138).

Problem Statement

The problem can be stated in the following way: given a text of an article, the algorithm will be able to predict whether it refers to True or Fake news. The text data will be prepared appropriately using Natural Language Processing (NLP) techniques, will be vectorised, and then will be fed into machine learning algorithms.

When we use machine learning algorithms, it is necessary to create the so called Bag of Words model to represent the texts, either as word (or bigrams, trigrams, n-grams) counts or one hot encoding of term frequency- inverse document frequency that can be used as features to train the model.

I decided that it is more wise to first create a Proof of Concept (PoC) locally in my machine (even with a fraction of data) in order to demonstrate that the proposed solution has

a practical potential. Then, I proceeded to an end to end solution deploying a ML model in AWS. For the implementation, I followed the next steps:

- Explored data to derive useful insights and get a better feeling of the data (EDA). I made several plots to check balance of data and word clouds to get the feeling of the most frequent words and bigrams.
- Cleaned, simplified, and prepared the dataset in a proper format:
    - Converted the entire document's text into lower case, so that capitalisation is ignored (e.g., IndIcaTE is treated the same as Indicate).
    - Normalised numbers, i.e. replaced all numbers with the text "number".
    - Removed non-words and punctuation, as well as trimmed all white spaces (tabs, newlines, spaces) to a single space character.
    - Tokenised, i.e. break up sequence of strings into pieces, such as words, keywords, phrases, symbols and other elements called tokens. Tokens can be individual words, phrases or even whole sentences. In the process of tokenisation, some characters like punctuation marks were discarded.
    - Stemmed words, i.e. reduced words to their stemmed form. For instance, "discount", "discounts", "discounted" and "discounting" will be all replaced with "discount".
    - Applied other normalising techniques (e.g. html, link normalisation, etc.) and tested models for the best combinations.
- Implemented a bag-of-words model. Since we cannot work with text directly when using machine learning algorithms, I needed to convert the text to numbers. Algorithms take vectors of numbers as input, therefore I needed to convert documents to fixed-length vectors of numbers. A simple and effective model for thinking about text documents in machine learning is called the bag-of-words model, or BoW. The model is simple in that it throws away all of the order information in the words and focuses on the occurrence of words in a document. This can be done by assigning each word a unique number. Then, any document we see can be encoded as a fixed-length vector with the length of the vocabulary of known words. The value in each position in the vector could be filled with a count or frequency of each word in the encoded document. This is the bag of words model, where we are only concerned with encoding schemes that represent what words are present or the degree to which they are present in encoded documents without any

information about order. The Scikit-learn library provides 3 different schemes that we could use: CountVectorizer, TfidfVectorizer, HashingVectorizer.

- Split the dataset to train, validation, and test sets.
- Trained a simple and fast model (Naive Bayes offered by Sklearn module) to get some first results from our data. This was done locally as a PoC before proceeding to a paid service for deployment, such as AWS.
- Evaluated performance with various metrics, i.e. accuracy score, confusion matrix and report (precision, recall, f1 scores), confusion plot.
- Prepared data for AWS SageMaker BlazingText algorithm.
- Trained the model and saved its artefacts in S3.
- Deployed the model in another instance.
- Tested with the unseen test data located in S3. Evaluated with accuracy score.
- Created a Lambda function that prepares data/text input from a REST API.
- Tested everything deploying a simple Web App html. The endpoint can be called from a simple HTML webpage. In this page, the user is able to post a text and check whether is true or fake news.

Metrics

The dataset properly split to train, validation, and test sets before doing any data transformations and induce data leakage. The validation test can be used for the model tweaking in order to optimise its hyper-parameters. The dataset was clearly balanced so accuracy score only can be sufficient as an evaluation metric. However, I also used a classification report, confusion matrix and plot provided by the Sklearn framework. These offers more metrics, such as precision, recall, and f1 scores, which can depict all the type of errors of our models (Type I and Type II errors).

II. Analysis

Data Exploration and Visualizations

The analysis of the EDA can read:

- either downloading the `fake-news-detector/data-exploration.html` (best experience to see the Plotly charts),
- or having a look at the `fake-news-detector/data-exploration.ipynb`

The first step in working with any dataset is loading the data in and noting what information is included in the dataset. This is an important step in eventually working with the data, and knowing what kinds of features we have to work with as we transform and group the data. So, the first step is all about exploring the data and finding patterns about the features we are given and the distribution of data. The dataset is made of multiple text strings and other characteristics summarised in csv files named Fake.csv and True.csv, which we can read in using pandas.

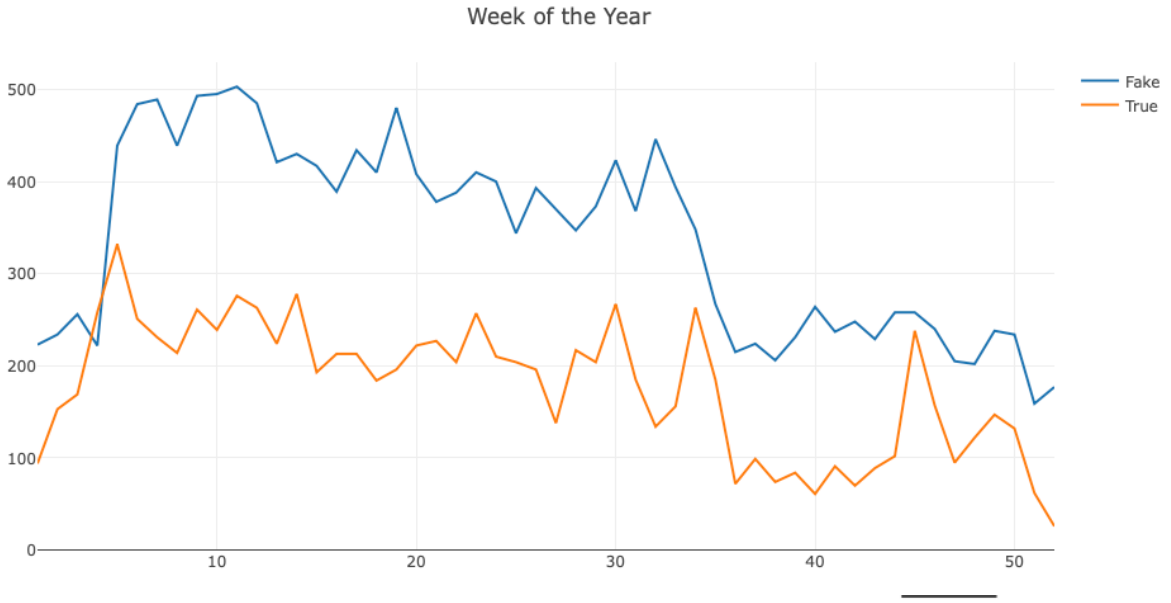
	title	text	subject	date
0	As U.S. budget fight looms, Republicans flip t...	WASHINGTON (Reuters) - The head of a conservat...	politicsNews	December 31, 2017
1	U.S. military to accept transgender recruits o...	WASHINGTON (Reuters) - Transgender people will...	politicsNews	December 29, 2017
2	Senior U.S. Republican senator: 'Let Mr. Muell...	WASHINGTON (Reuters) - The special counsel inv...	politicsNews	December 31, 2017
3	FBI Russia probe helped by Australian diplomat...	WASHINGTON (Reuters) - Trump campaign adviser ...	politicsNews	December 30, 2017
4	Trump wants Postal Service to charge 'much mor...	SEATTLE/WASHINGTON (Reuters) - President Donal...	politicsNews	December 29, 2017

	title	text	subject	date
0	Donald Trump Sends Out Embarrassing New Year'...	Donald Trump just couldn t wish all Americans ...	News	December 31, 2017
1	Drunk Bragging Trump Staffer Started Russian ...	House Intelligence Committee Chairman Devin Nu...	News	December 31, 2017
2	Sheriff David Clarke Becomes An Internet Joke...	On Friday, it was revealed that former Milwauk...	News	December 30, 2017
3	Trump Is So Obsessed He Even Has Obama's Name...	On Christmas day, Donald Trump announced that ...	News	December 29, 2017
4	Pope Francis Just Called Out Donald Trump Dur...	Pope Francis used his annual Christmas Day mes...	News	December 25, 2017

```
There are 21417 real and 23481 fake news
```

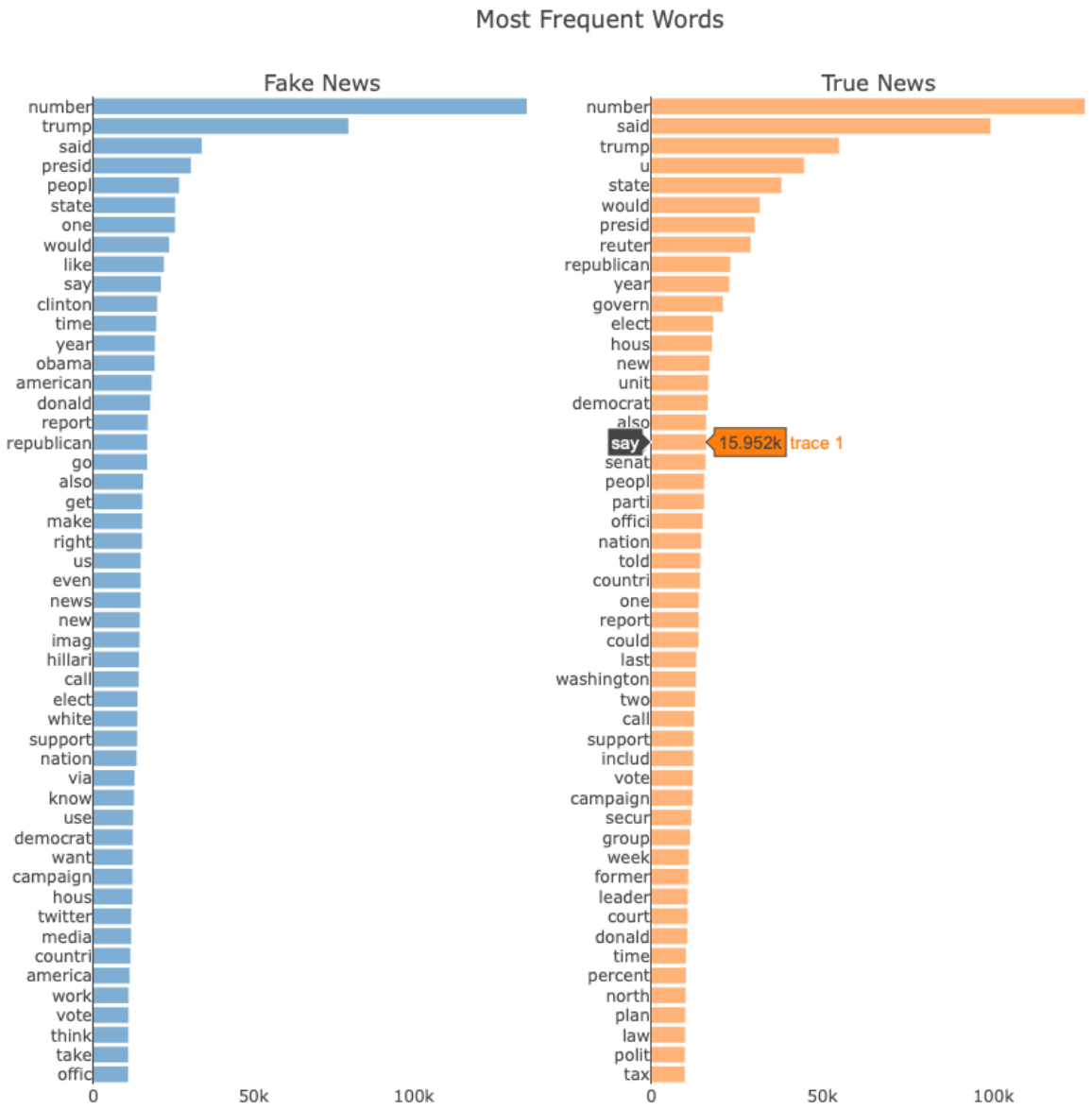
Then I plotted the distributions of the news. I noticed that although True data subjects' are almost evenly divided into 'politics' and 'world' news, in the Fake data article subjects are spread across several different classes. For this reason, I decided to remove the subject field as a feature since it is not helpful.

I also investigated if there are any patterns in the dates between fake and true news, without finding anything. I checked various date features as day of week, week of the year, monthly, etc.



Finally, I checked the word frequency of the fake and true news separately either with word clouds

or computing the frequencies and plotting the most frequent words.



I noticed that true news contains more dates like on Tuesday, on Friday, etc.

Algorithms and Techniques

The algorithms I used for this project are two. First I used locally a fast and simple algorithm from Sklearn, the Naive Bayes. With this approach I wanted to create a Proof of Concept and demonstrate that my solution can give good results before proceeding to a production viable solution using a paid service such as AWS. Then, I used the AWS

BlazingText model which is a fast algorithm that used specifically for text machine learning tasks. The models are discussed and presented thoroughly in the following files:

- `fake-news-detector/fake-news-detector.html` or
- `fake-news-detector/fake-news-detector.ipynb`

The Amazon SageMaker BlazingText algorithm provides highly optimised implementations of the Word2vec and text classification algorithms. The Word2vec algorithm is useful for many downstream natural language processing (NLP) tasks, such as sentiment analysis, named entity recognition, machine translation, etc. Text classification is an important task for applications that perform web searches, information retrieval, ranking, and document classification.

The Word2vec algorithm maps words to high-quality distributed vectors. The resulting vector representation of a word is called a word embedding. Words that are semantically similar correspond to vectors that are close together. That way, word embeddings capture the semantic relationships between words.

Many natural language processing (NLP) applications learn word embeddings by training on large collections of documents. These pretrained vector representations provide information about semantics and word distributions that typically improves the generalizability of other models that are later trained on a more limited amount of data. Most implementations of the Word2vec algorithm are not optimized for multi-core CPU architectures. This makes it difficult to scale to large datasets.

With the BlazingText algorithm, you can scale to large datasets easily. Similar to Word2vec, it provides the Skip-gram and continuous bag-of-words (CBOW) training architectures. BlazingText's implementation of the supervised multi-class, multi-label text classification algorithm extends the fastText text classifier to use GPU acceleration with custom CUDA kernels. You can train a model on more than a billion words in a couple of minutes using a multi-core CPU or a GPU. And, you achieve performance on par with the state-of-the-art deep learning text classification algorithms.

Benchmark

As a benchmark, I chose the accuracy score obtained by the authors of the articles. In particular, they report an accuracy score of 90% by using a Linear Support Vector Machine.

III. Methodology

Data Preprocessing

Data processing was made during the Exploratory Data Analysis so as to create word clouds and other plots presenting the words frequency. In the `fake-news-detector.ipynb` I used various helper functions from the `helper.py` in order to prepare and processed data with Natural Language Techniques, which were discussed earlier. I took several steps depending on the model I was working with. However, most of the crucial steps were mentioned previously:

- Combine the two datasets
- Remove useless columns
- Prepare the class labels, i.e. encode Fake with 1 and True with 0

```
# Create a function to read and prepare data.
def read_data(path, random_state):
    """This is a function that reads data, creates a column 'label' to indicate if news is fake or true, concatenate
the two datasets, shuffle data, and return the df.

    Args:
        path (str): The directory of the datasets.
        random_state (int): A number that sets a seed to the random generator, so that shuffles are always determinis
tic.

    Returns:
        pandas dataframe: A pandas dataframe with prepared data.
    """
    # Read data in pandas.
    true = pd.read_csv(path + "True.csv")
    fake = pd.read_csv(path + "Fake.csv")

    # Create the 'label' column.
    true['label'] = 0
    fake['label'] = 1

    # Concatenate the 2 dfs.
    df = pd.concat([true, fake])

    # To save a bit of memory set fake and true to None.
    fake = true = None

    #  Shuffle data.
    df = df.sample(frac=1, random_state=random_state).reset_index(drop=True)

    # Return the df.
    return df
```

- Clean and normalise text
- Tokenise
- Split to train, test, and validation sets.

```
# Create a function to process text.
def process_text(text):
    """Remove any punctuation, numbers, newlines, and stopwords. Convert to lower case. Split the text string into in
dividual words, stem each word, and append the stemmed word to words. Make sure there's a single space between each s
temmed word.

    Args:
        text (str): A text.

    Returns:
        str: Cleaned, normalized, and stemmed text.
    """
    # Remove HTML tags.
    text = BeautifulSoup(text, "html.parser").get_text()

    # Normalize links replacing them with the str 'link'.
    text = re.sub('http\S+', 'link', text)

    # Normalize numbers replacing them with the str 'number'.
    text = re.sub('\d+', 'number', text)

    # Normalize emails replacing them with the str 'email'.
    text = re.sub('\S+@\S+', 'email', text, flags=re.MULTILINE)

    # Remove punctuation.
    text = text.translate(str.maketrans('', '', string.punctuation))

    # Remove whitespaces.
    text = text.strip()

    # Convert all letters to lower case.
    text = text.lower()

    # Create the stemmer.
    stemmer = SnowballStemmer('english')

    # Split text into words.
    words = text.split()

    # Remove stopwords.
    words = [w for w in words if w not in stopwords.words('english')]

    # Stem words.
    words = [stemmer.stem(w) for w in words]

    return ' '.join(words)
```

Eventually, if the input text is for instance:

"Germany extends passport controls on Austrian border, flights from Greece BERLIN (Reuters) - Germany has extended temporary passport controls on its border with Austria and for flights coming from Greece for six more months amid continuing concerns about attacks and illegal immigration,…"

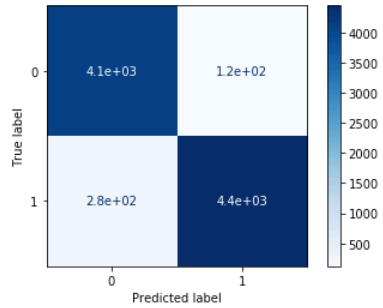
The output after the text processing will be like this:

"germani extend passport control austrian border flight greec berlin reuter germani extend temporari passport control border austria flight come greec six month amid continu concern attack illeg immigr"

i.e. all lower casing, stemmed, no punctuation, and normalised. Then, I tokenised the text using CountVectorizer, split the data, and train locally the Naive Bayes model. The model gave very high accuracy, recall, precision, and f1 scores.

```
# Make and save the predictions.
predictions = model.predict(test_X)

print(confusion_matrix(test_y, predictions))
print(classification_report(test_y, predictions))
plot_confusion_matrix(model, test_X, test_y, cmap='Blues')
```

```
[[4131  120]
 [ 280 4449]]
              precision    recall  f1-score   support

           0       0.94      0.97      0.95      4251
           1       0.97      0.94      0.96      4729

    accuracy                           0.96      8980
   macro avg       0.96      0.96      0.96      8980
weighted avg       0.96      0.96      0.96      8980

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7fcf18fcd240>
```



Because of the very good results of the PoC with the Naive Bayes model, I decided to move the project in AWS using the BlazingText algorithm. Specifically, the training/ validation data for the BlazingText algorithm must be processed appropriately so as to contain a training sentence per line along with the labels. Labels are words that are prefixed by the string __label__. So, I needed again to relabel correctly my data. Here is an example of a training/validation file:

- __label__1 linux ready for prime time , intel says , despite all the linux hype , the open-source movement has yet to make a huge splash in the desktop market . that may be about to change , thanks to chipmaking giant intel corp .
- __label__0 bowled by the slower one again , kolkata , november 14 the past caught up with sourav ganguly as the indian skippers return to international cricket was short lived .

After this I uploaded the datasets to AWS S3, setup the model, and trained it in a new instance. The code can be seen in the following pictures.

```
# Store the current SageMaker session.
session = sagemaker.Session()

# Store the bucket.
bucket = session.default_bucket()

# S3 prefix (which folder will we use).
prefix = 'fake-news-bt'

# Upload the processed test, train and validation files,
# which are contained in data directory to S3 using session.upload_data().
test_location = session.upload_data(os.path.join(DIR, 'news.test'), key_prefix=prefix)
val_location = session.upload_data(os.path.join(DIR, 'news.valid'), key_prefix=prefix)
train_location = session.upload_data(os.path.join(DIR, 'news.train'), key_prefix=prefix)
```

```
# Our current execution role is required when creating the model as the training
# and inference code will need to access the model artifacts.
role = get_execution_role()
```

```
# We need to retrieve the location of the container, which is provided by Amazon for using XGBoost.
# As a matter of convenience, the training and inference code both use the same container.
container = get_image_uri(session.boto_region_name, 'blazingtext', 'latest')
```

```
'get_image_uri' method will be deprecated in favor of 'ImageURIProvider' class in SageMaker Python SDK v2.
```

```
# First we create a SageMaker estimator object for our model.
bt_model = sagemaker.estimator.Estimator(container, # The location of the container we wish to use
                                         role, # What is our current IAM Role
                                         train_instance_count=1, # How many compute instances
                                         train_instance_type='ml.c4.4xlarge', # What kind of compute instances
                                         train_volume_size = 30,
                                         train_max_run = 360000,
                                         input_mode= 'File',
                                         output_path='s3://{}/{}/output'.format(bucket, prefix),
                                         sagemaker_session=session)

# And then set the algorithm specific parameters.
bt_model.set_hyperparameters(mode="supervised",
                             epochs=10,
                             min_count=2,
                             learning_rate=0.05,
                             vector_dim=10,
                             early_stopping=True,
                             patience=4,
                             min_epochs=5,
                             word_ngrams=3)
```

```
WARNING:root:Parameter image_name will be renamed to image_uri in SageMaker Python SDK v2.
```

```
s3_input_train = sagemaker.s3_input(s3_data=train_location, distribution='FullyReplicated',
                                    content_type='text/plain')
s3_input_validation = sagemaker.s3_input(s3_data=val_location, distribution='FullyReplicated',
                                         content_type='text/plain')
```

```
WARNING:sagemaker:'s3_input' class will be renamed to 'TrainingInput' in SageMaker Python SDK v2.
WARNING:sagemaker:'s3_input' class will be renamed to 'TrainingInput' in SageMaker Python SDK v2.
```

```
bt_model.fit({'train': s3_input_train, 'validation': s3_input_validation})
```

The job was completed successfully resulting a remarkable score on the validation set over 98%. The last training epoch and the score can be seen in the following picture.

```
Using 16 threads for prediction!
Validation accuracy: 0.986359
Validation accuracy improved! Storing best weights...
##### Alpha: 0.0004  Progress: 99.19%  Million Words/sec: 20.21 #####
-------------- End of epoch: 10
Using 16 threads for prediction!
Validation accuracy: 0.986498
Validation accuracy improved! Storing best weights...
##### Alpha: 0.0000  Progress: 100.00%  Million Words/sec: 19.28 #####
Training finished.
Average throughput in Million words/sec: 19.28
Total training time in seconds: 3.65

#train_accuracy: 0.9915
Number of train examples: 28734

#validation_accuracy: 0.9865
Number of validation examples: 7184

2020-06-15 07:35:05 Completed - Training job completed
Training seconds: 60
Billable seconds: 60
```

Refinement

A refinement step I took was to increase the number of n-grams in the BoW model. I started using only unigrams and eventually I used unigrams, bigrams, and trigrams all together and got the best results.

IV. Results

Model Evaluation and Validation

As aforementioned, I just used the accuracy score on the test unseen dataset since the dataset is well balanced. Once I constructed and fit my model, SageMaker stored the resulting model artefacts, which can be used to deploy an endpoint. Deploying an endpoint is a lot like training the model with a few important differences. The first is that a deployed model doesn't change the model artefacts, so as you send it various testing instances the model won't change. Another difference is that since we aren't performing a fixed computation, as we were in the training step or while performing a batch transform, the compute instance that gets started stays running until we tell it to stop. This is important to note as if we forget and leave it running we will be charged the entire time. To get the predictions on the test set, I used this deployed model, but first I had to create batches of the test dataset as can be seen in the following picture.

```
# Create a function to define the batches.
# From: https://stackoverflow.com/questions/8290397/how-to-split-an-iterable-in-constant-size-chunks
def batch(iterable, n=1):
    l = len(iterable)
    for ndx in range(0, l, n):
        yield iterable[ndx:min(ndx + n, l)]
```

```
# Create batches of 512 inputs for prediction and add them to a list.
predictions = []
for x in batch(test_X.iloc[:,0].str[12:-1].tolist(), 512):
    payload = {"instances" : x}
    prediction_batch = predictor.predict(json.dumps(payload))
    prediction_batch = [int(prediction.get("label")[0][9:]) for prediction in json.loads(prediction_batch)]
    predictions.append(prediction_batch)
```

```
# Flatten list.
predictions = sum(predictions, [])
```

Lastly, we check again to see what the accuracy of our model is.

```
accuracy_score(test_y, predictions)
```

```
0.9874164810690423
```

The accuracy score even on the test dataset was over 98% showing that our model was generalised very well on the training data.

Justification

Compared to the benchmark score (90%), my model presented more than 8 units greater accuracy. The BlazingText model I used generalises great to unseen data, is fast to train, and can be used to as a backend to a web application. However, it must be noted that this model is limited to news from one source and if we want more functionality, we should train with more data from various sources.

V. Conclusion

Reflection

It seems that the model can classify with very high accuracy fake and true news from a specific news source. This could be extended with more data from various sources and eventually create a viable solution for the end user as a helpful guide to detect fake news.

AWS can offer us all the sources needed to create and scale-up such a system. The dataset I used here was about 44,000 examples with about 110 MB of size. This dataset was trained in about few seconds in a simple AWS machine learning instance, so even if we use a much larger dataset with millions of data, we will get fast results. BlazingText is a very fast AWS algorithm and SageMaker provides all the functionality to build a working production solution with the minimal cost.

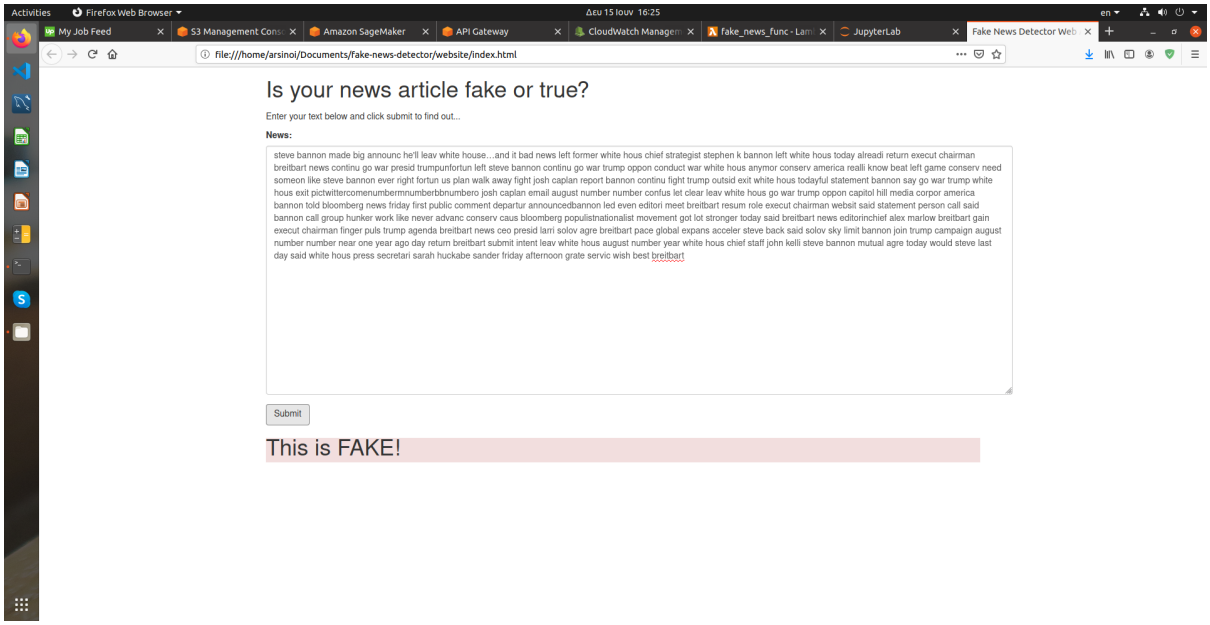
As a final task of this study was to deploy the model and put it to work accessing it by using a very simple web app. For the deployment, I used the following services:

- SageMaker
- IAM
- S3
- ClouWatch
- Lambda

- API Gateway

The Lambda function I used as well as an example of fakes news detected through the website can be seen in the next pictures.

```
# We need to use the low-level library to interact with SageMaker since the SageMaker API
# is not available natively through Lambda.
import boto3

# And we need the following libraries to do some of the data processing.
import re
import string
import json

# Create a function to process text.
def process_text(text):
    # Normalize links replacing them with the str 'link'.
    text = re.sub('http\S+', 'link', text)

    # Normalize numbers replacing them with the str 'number'.
    text = re.sub('\d+', 'number', text)

    # Normalize emails replacing them with the str 'email'.
    text = re.sub('\S+@\S+', 'email', text, flags=re.MULTILINE)

    # Remove punctuation.
    text = text.translate(str.maketrans('', '', string.punctuation))

    # Remove whitespaces.
    text = text.strip()

    # Convert all letters to lower case.
    text = text.lower()

    # Split text into words.
    words = text.split()

    return ' '.join(words)

runtime= boto3.client('runtime.sagemaker')

def lambda_handler(event, context):
    data = event['body']
    sentence = process_text(data)

    try:
        payload = {"instances" : sentence}

        response = runtime.invoke_endpoint(EndpointName='blazingtext-2020-06-15-07-32-18-142',
                                           ContentType='application/json',
                                           Body=json.dumps(payload))

        result = json.loads(response['Body'].read().decode())
        # prob = []
        labels = []
        for label in result[0]['label']:
            labels.append(label[9:])
        print("DATA", data)
        print("SENTENCE", sentence)
        return {
            'statusCode' : 200,
            'headers' : { 'Content-Type' : 'text/plain', 'Access-Control-Allow-Origin' : '*' },
            'body' : str(labels[0])
    }
        # return {'statusCode': 200, 'body': str(labels[0])}
    except Exception as e:
        print(e)
        return {'statusCode': 400,
                'body': json.dumps({'error_message': 'Unable to generate tag.'})}
```



Improvement

As it was aforementioned, the model is limited to specific news source, so this could be improved by:

- using more data from various news sources,
- optimising the hyper-parameters to get even higher scores,
- implementing a higher quality web page for the end user to insert the news text,
- exporting also as a result (except the Fake or True label) the probability of a news text to be fake or true. For example, it is more informative for the user to know that this news text was detected 65% as a True. This means that there are 35% probability to be Fake.