NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

Semester Project Report

Fake News Detector - Summer 2018/19

Maroulis Nikolaos - Μαρούλης Νικόλαος
AM: M1605
Msc. At Department of informatics (Data Science)

Course:	**Big Data (M111)**
Professor:	**Alexandros Doulas**

Athens 2019

1.Introduction

What is the purpose of this project?

The main purpose of this project is to implement a web-site that lets the user find out if an article he/she read online, is legitimate or fake. There are two different ways this can be achieved. The first way is by copy-pasting the article's main body into the input box, as shown in figure 1. The second way is by using various metadata of the Article like the Title, Author or/and Domain Url, as shown in figure 2.

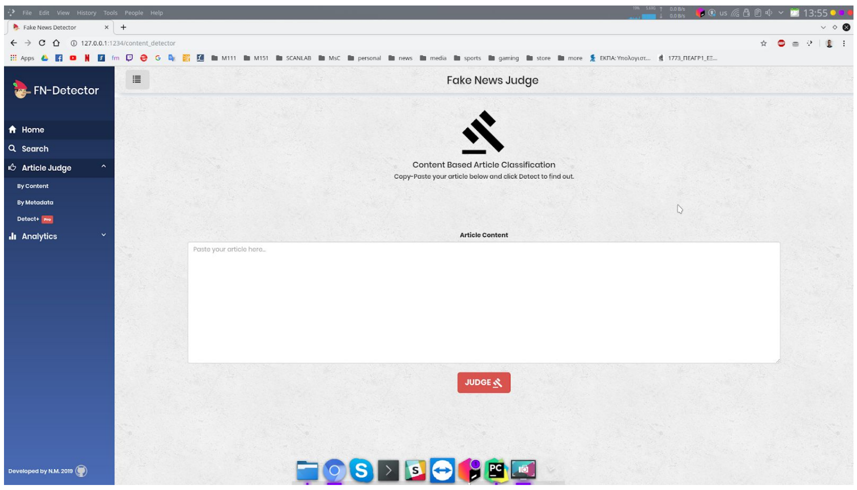


Figure 1. Classification by Content

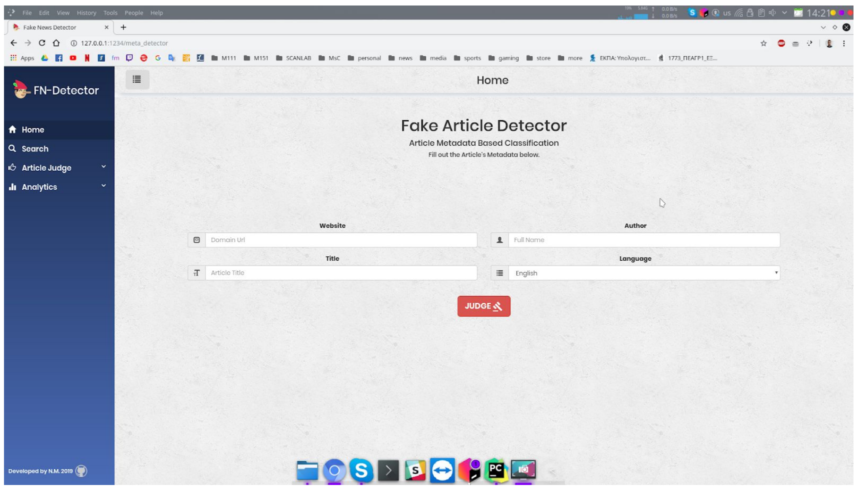


Figure 2. Classification by Metadata

Main Functionalities

Classification by content

This method uses only the article's main body for the classification input. The first step here is to find a Dataset that contains a large corpus of articles. For the content based classification, I used two datasets from Kaggle [1][2] with a total of 33 thousand articles. Then the articles were vectorized using the BoW vectorizer and the Tf-idf transformer. Finally for the classification process, the first step is to calculate the cosine similarity with the existing corpus, thus if a 'copy' of the target article is already in the stored dataset, there is no need for further investigation and the article is classified as fake. If no similarity is found using the cosine similarity method, the ML classification model comes to the rescue. The classification model is a Feed-Forward Deep Neural Network, with 3 layers, that was developed using the Keras API with a Tensorflow backend.

Classification by metadata

This method uses metadata from the article in order to classify it. In more detail the user has 3 options available, providing the Article's title, author or domain url that the article was found. The Datasets used here are the first Dataset from Kaggle as before, as well as, a dataset containing articles from the Onion Subreddit, a forum where users post only fake articles. As in the previous method, the inputs are initially compared with the training corpus using the cosine similarity method. If no similar metadata exist in the dataset, the main ML model is used. This is a Support Vector Machine (SVM) model, using a linear Kernel.

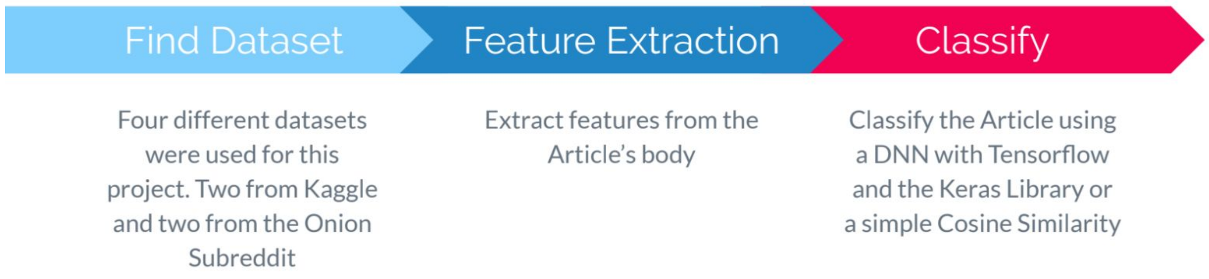

Find Dataset
Feature Extraction
Classify
Four different datasets were used for this project. Two from Kaggle and two from the Onion Subreddit
Extract features from the Article's body
Classify the Article using a DNN with Tensorflow and the Keras Library or a simple Cosine Similarity

Figure 3. Article Classification Steps

2. Document Corpus Vectorization

For the implementation of this section I used the TfidfVectorizer Library from sci-kit learn.

First let’s define what Tf-idf Vectorizer does.

What is **Tf-idf** ?

Computers are good with numbers, but not that much with textual data. One of the most widely used techniques to process textual data is TF-IDF. From our intuition, we think that the words which appear more often should have a greater weight in textual data analysis, but that’s not always the case. Words such as “the”, “will”, and “you”, called **stopwords**, appear the most in a corpus of text, but are of very little significance. Instead, the words which are rare are the ones that actually help in distinguishing between the data, and carry more weight.

TF-IDF stands for “Term Frequency — Inverse Data Frequency”. First, we will learn what this term means mathematically.

Term Frequency (tf): gives us the frequency of the word in each document in the corpus. It is the ratio of number of times the word appears in a document compared to the total number of words in that document. It increases as the number of occurrences of that word within the document increases. Each document has its own tf.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

Inverse Data Frequency (idf): used to calculate the weight of rare words across all documents in the corpus. The words that occur rarely in the corpus have a high IDF score. It is given by the equation below.

$$idf(w) = log(\frac{N}{df_t})$$

Combining these two we come up with the TF-IDF score (w) for a word in a document in the corpus. It is the product of tf and idf:

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

So in simple terms if a word comes up a lot in all the Documents it is considered less important than others. Because the term "the" is so common, term frequency will tend to incorrectly emphasize documents which happen to use the word "the" more frequently, without giving enough weight to the more meaningful terms "brown" and "cow". The term "the" is not a good keyword to distinguish relevant and non-relevant documents and terms, unlike the less-common words "brown" and "cow". Hence an inverse document frequency factor is incorporated which diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely.

What is a Text Vectorizer?

To represent documents in vector space, we first have to create mappings from terms to term IDS. We call them terms instead of words because they can be arbitrary n-grams not just single words. We represent a set of documents as a sparse matrix, where each row corresponds to a document and each column corresponds to a term. This can be done in 2 ways: using the vocabulary itself or by feature hashing.

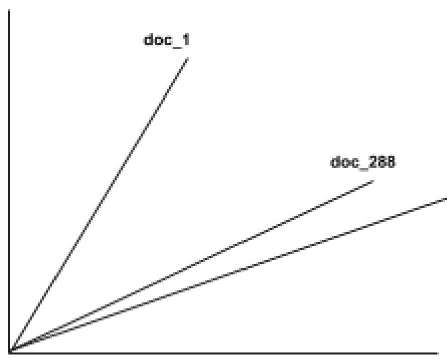
Initial Document Corpus

Document ID	Content
doc_1	Sed ut perspiciatis unde omnis...
doc_2	Ut enim ad minima veniam, quis nostrum..
…..	…..
doc_n	Quis autem vel eum iure reprehenderit….

So we Start with a Corpus of documents where each row represents a document with (let’s say) 2 columns, one the document id and the other the text content. We want to represent that in the vector space so we create a Matrix where each row is the document, but there we have a column for each word (term) in the entire Corpus. So a simple solution is to have 1 in the column that the term appears in the corresponding document. The tf-idf vectorizer, instead of just filling 1, or the number of occurrences in the corresponding field, additionally, adds a weight to the term based on its frequency (low weight if it appears a lot).

Term Matrix

	t1	t2	t3	t4	t5	t6	…..	t_m
doc_1	0	0	0	1	1	0		1
doc_2	1	1	0	0	0	0		0
…..								
doc_n	0	0	0	0	0	1		0



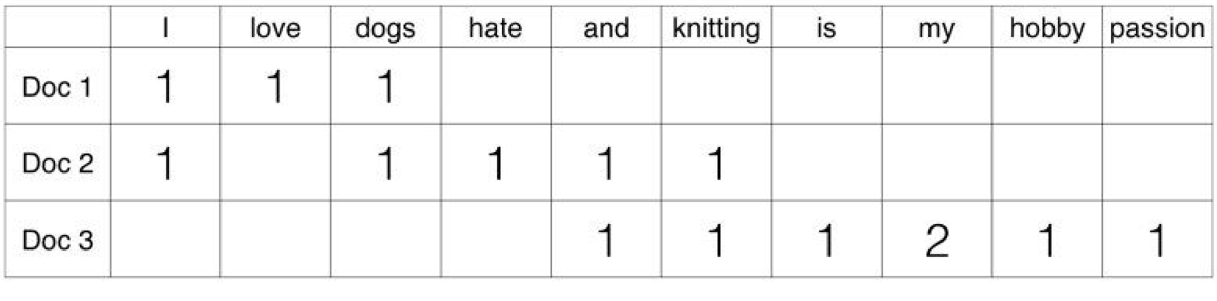
Document Representation in the Vector Space

Text Transformation Techniques

Bag of Words (BoW)

In practice, the Bag-of-words model is mainly used as a tool of feature generation. After transforming the text into a "bag of words", we can calculate various measures to characterize the text. The most common type of characteristics, or features calculated from the Bag-of-words model is term frequency, namely, the number of times a term appears in the text. This I vector representation does not preserve the order of the words in the original sentences. This is just the main feature of the Bag-of-words model. This kind of representation has several successful applications. However, term frequencies are not necessarily the best representation for the text. Common words like "the", "a", "to" are almost always the terms with highest frequency in the text. Thus, having a high raw count does not necessarily mean that the corresponding word is more important. To address this problem, one of the most popular ways to "normalize" the term frequencies is to weight a term by the inverse of document frequency, or tf–idf.

For the Project Implementation I used **CountVectorizer** (from the sklearn library), which creates a BoW representation of the document in the vector space.
> CountVectorizer(analyzer ='word', lowercase=True).



	I	love	dogs	hate	and	knitting	is	my	hobby	passion
Doc 1	1	1	1							
Doc 2	1		1	1	1	1				
Doc 3					1	1	1	2	1	1

A BoW matrix example

Word to Vec (W2V)

Word2vec is a group of related models that are used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space.

Word2vec can utilize either of two model architectures to produce a distributed representation of words: continuous bag-of-words (CBOW) or continuous skip-gram. In the continuous bag-of-words architecture, the model predicts the current word from a window of surrounding context words.

For the Project's purposes I used the pretrained Word2Vec model from Google[3] . This implementation was tested for the Metadata-based classification of the title.

My main issue was that W2V model creates a **vector for each word**, but I needed **a vector for each document**. So i created the functions ***MeanEmbeddingVectorizer*** and ***TfidfEmbeddingVectorizer***, which iterate each documents' words and takes each word's vector and take the mean. So each document's vector is the mean of all its' words' vectors. The TfidfEmbeddingVectorizer also uses Tf-Idf. I also experimented a little with the model and which can give the neighbors of words etc. An example is the code below:

`> w2v_model.most_similar('youtube',  topn=10)`, which finds the top 10 neighbor words for the word youtube (cosine similarity), the result:

```
('videos', 0.7882595062255859),
('channels', 0.7623605728149414),
('spotify', 0.7340512871742249),
('facebook', 0.7319403886795044),
('video', 0.7314403057098389),
('minecraft', 0.7295858263969421),
('content', 0.7102934718132019),
('ads', 0.7091004848480225),
('twitch', 0.7027051448822021),
('adverts', 0.6964372992515564)
```

The Tfidf Vectorizer from the Scikit-Learn Library

For the implementation of the vectorizer, I used the Tfidfvectorizer class from the Scikit-Learn Library. This is a combination of two other classes from the same library, the countVectorizer which is a BoW vectorizer and the TfidfTransformer which transforms the term vector using the Tf-idf technique. The code below shows the vectorizer:

"vectorizer = TfidfVectorizer(max_features=8192, stop_words='english', lowercase=True, ngram_range=(1, 3),smooth_idf=False)\n",

```
vectorizer = TfidfVectorizer( max_features=8192, stop_words='english',
lowercase=True, ngram_range=(1, 3), smooth_idf=False)
```

The attribute **max_features** indicates the column size of the term vector, thus leaving out the less relevant terms and keeping 8192. I chose the previous value as the size of the feature vector in order to have a tolerable input in the neural network. If the **stop_words** a=and **lowercase** attributes are chosen, stop words are removed and all letters become lowercase from the input corpus prior to the vectorization.

The **ngram_range** is an important parameter, as it specifies the n-grams of words to be stored in the term matrix. An N-gram is a sequence of N words: a 2-gram (or bigram) is a two-word sequence of words like “New York” and a 3-gram (or trigram) is a three-word sequence of words etc. the ngram range I used was from 1-gram (single word) up to 3-gram, so the column of the term vector contains all the single words, all the combinations of two consecutive words, as well as a sequence of three words. This is used in order not to lose any valuable information a sequence of words have.

3.Document Classification using Cosine Similarity

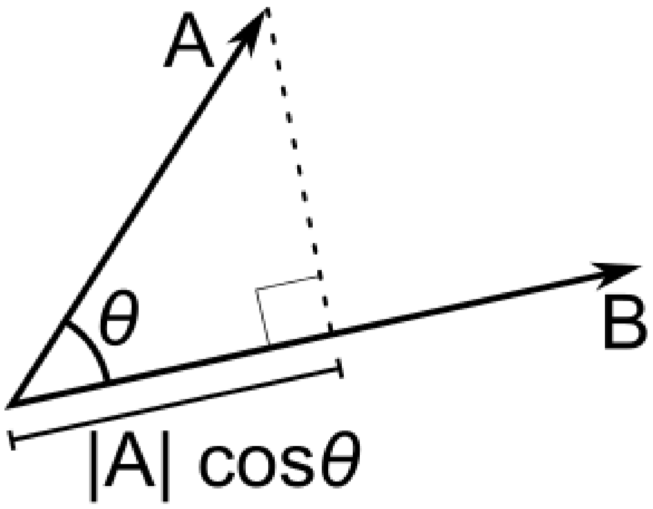
What is the Cosine Similarity?

The cosine similarity between two vectors (or two documents on the Vector Space) is a measure that calculates the cosine of the angle between them. This metric is a measurement of orientation and not magnitude, it can be seen as a comparison between documents on a normalized space because we’re not taking into the consideration only the magnitude of each word count (tf-idf) of each document, but the angle between the documents. What we have to do to build the cosine similarity equation is to solve the equation of the dot product for the $\cos\theta$:

$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \|\vec{b}\| \cos\theta$$

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|}$$

A graphical representation:



The projection of the vector A into the vector B. By Wikipedia.

So now that we have the matrix, we take each term vector for each document and calculate the cosine similarity. We set a threshold **θ** and if it is greater than 0.9 we assume that the two documents are similar.

```
similarity_df = train_df['Content'].head(np.size(train_df,0)).values

tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(similarity_df)
cos_similarity_matrix = cosine_similarity(tfidf_matrix, tfidf_matrix)
```

What the code above does, is to create the term matrix and the calculate the cosine similarity for all the documents with each other and create a matrix with these similarities. I then iterate the matrix to find similarities greater than 0.9 (as shown below)

```
for i in range(tfidf_matrix.shape[0]):
    for c in range(len(cos_similarity_matrix[i])):
        if cos_similarity_matrix[i][c] > sim_threshold  and c != i:
            if([c,i,cos_similarity_matrix[i][c]] not in similarity_vector): # prevent duplicate entry
                return i,c # similar article found
```

Then, if any similar article is found (>90% similarity), there is no need to use the classifier because the training corpus already contains a ‘copy’ of the target article, thus it is **fake**. The document ID of the similar document and the similarity indicator is stored in order to notify the user about where the fake article was first found.

4.Document Classification using Machine Learning Models

What is Classification in Machine Learning?

Classification is the process of predicting the class of given data points. Classes are sometimes called as targets/ labels or categories. Classification predictive modeling is the task of approximating a mapping function (f) from input variables (X) to discrete output variables (y). Classification belongs to the category of supervised learning where the targets also provided with the input data.

In this project the Task is to classify text documents into one of the following Categories **Fake**, **Legitimate***.* Given a training set where each entry contains a Document ID, its Text Content (Body, Title, Author, Url Domain) and the given Label (Class), we train a model and test it to take its accuracy.

Required Steps to Classify a document corpus:

1) Split dataset to Training and Testing Data.
2) Vectorize / Transform: Convert a collection of text documents to a matrix of token counts.
3) Train a model given the training data and test it with the Test data. (10 fold cross validation)
4) Aggregate Results (Accuracy, Recall, Precision, F1-Score) and compare each model’s performance.

What is K-fold Cross Validation?

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample. The procedure has a single parameter called k that refers to the number of groups that a given data sample is to be split into. As such, the procedure is often called k-fold cross-validation. When a specific value for k is chosen, it may be used in place of k in the reference to the model, such as k=10 becoming 10-fold cross-validation. Cross-validation is primarily used in applied machine learning to estimate the skill of a machine learning model on unseen data. That is, to use a limited sample in order to estimate how the model is expected to perform in general when used to make predictions on data not used during the training of the model.
The general procedure is as follows:

1. Shuffle the dataset randomly.
2. Split the dataset into k groups
3. For each unique group:
    1. Take the group as a hold out or test data set
    2. Take the remaining groups as a training data set
    3. Fit a model on the training set and evaluate it on the test set
    4. Retain the evaluation score and discard the model
4. Summarize the skill of the model using the sample of model evaluation scores

For the purposes of the project I implemented my own 10-fold cross validation algorithm, in order to be more flexible with the inner operations.

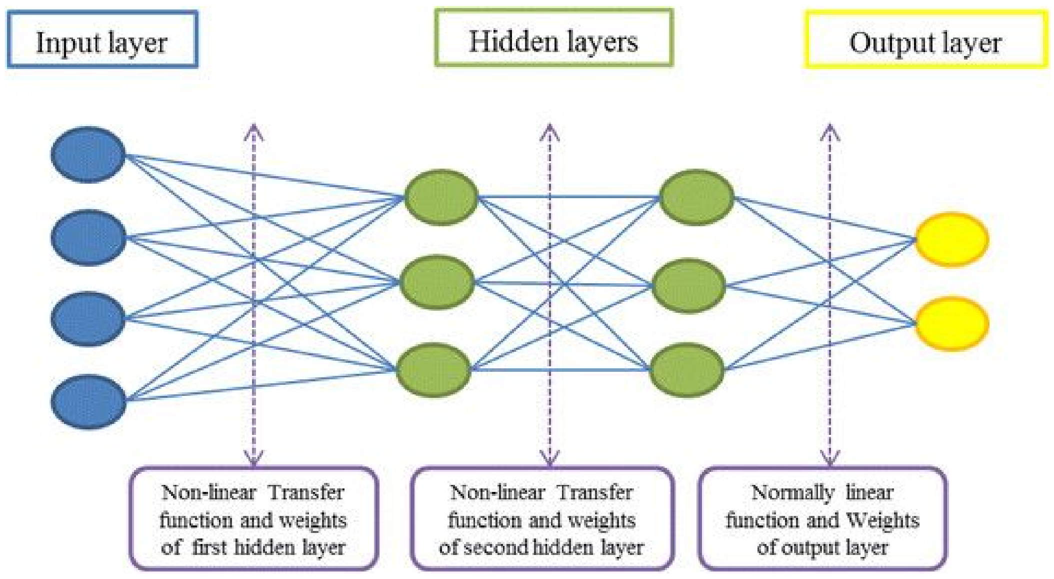
The models that will be demonstrated below were developed in **Jupyter Notebooks**. Jupyter Notebook allows the user for more dynamic writing, by having blocks of code that the user can run independently. In order to retrieve the models that the website uses, I run the code in the notebooks and after the training they are stored using the **Joblib** and **Pickle** libraries. The python code in the website loads the models and uses them after a query from the user.

Machine Learning Models

Feed Forward Deep Neural Network Classifier using Tensorflow

An DNN is a deep, artificial neural network. It is composed of more than one layers. They are composed of an input layer to receive the signal, an output layer that makes a decision or prediction about the input, and in between those two, an arbitrary number of hidden layers that are the true computational engine of the DNN.

Neural Networks with one hidden layer are capable of approximating any continuous function. (shown below)


Input layer
Hidden layers
Output layer
Non-linear Transfer function and weights of first hidden layer
Non-linear Transfer function and weights of second hidden layer
Normally linear function and Weights of output layer

The **activation function** takes the input of the neuron and applies a function on it. I chose **RELU** which keeps only the positive inputs of the neuron.

I used solver= ‘adam’ which is the Adam Optimizer optimization algorithm. The **Adam optimization** algorithm is an extension to stochastic gradient descent hat maintains a per-parameter learning rate that improves performance on problems with sparse gradients. ***Gradient Descent** is a procedure where at the Back Propagation phase, the weights of the neural network are updated.

Hidden Layer Size indicates the nodes in each hidden layer, I set hidden 3 layers of 512, 256 and 256 nodes accordingly. The input layer has a size of 8192, which is the size of the feature vector of each Document.

One **epoch** is when an entire dataset is passed both forward and backward through the neural network only once, so I decided to finish the training procedure after 60 epochs.

The **batch size** is a number of samples processed before the model is updated. I chose to pass 64 samples each time, before every update.

```
vectorizer = TfidfVectorizer(max_features=8192, stop_words='english', lowercase=True,
ngram_range=(1, 3),smooth_idf=False)

model = keras.models.Sequential()
model.add(keras.layers.Dense(512, input_dim=np.size(X,1), activation='relu'))
model.add(keras.layers.Dense(256, activation='relu'))
model.add(keras.layers.Dense(256, activation='relu'))
model.add(keras.layers.Dense(1, activation='sigmoid'))
model.summary()
kmodel.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer='adam')
model.fit(X,Y, epochs=60, batch_size=64, verbose=1, shuffle=True, class_weight=None,
sample_weight=None) # train

model.save('../../models/keras_content_classifier.h5') # MODEL SAVE
```

Support Vector Machine (SVM)

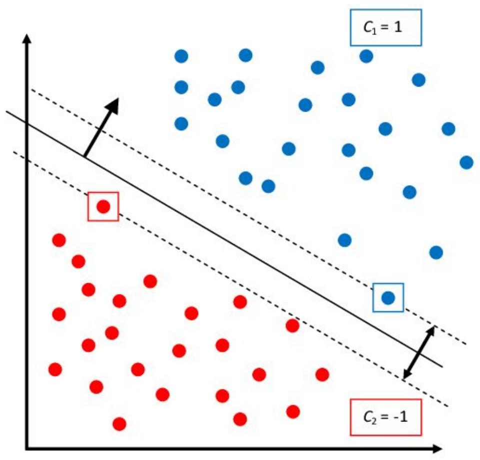
In machine learning, support-vector machines (SVMs) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier (although methods such as Platt scaling exist to use SVM in a probabilistic classification setting). An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

So in order to train an **SVM model for text classification,** I need to train it with data it can work with. That's why I used the Vectorizer to turn the docs into term vectors, because SVM works with vectors, separating them ideally.

I used the Sklearn Library for SVM

> svm.SVC(kernel='linear', C=0.92, decision_function_shape='ovr'),

- kernel: Defines the **kernel function** to be used. A kernel function (or kernel trick), adds an additional dimension to the features in order to make them linearly separable by a hyperplane.
- C: Penalty Parameter for mismatched data (outliers)
- decision_function_shape: Because we have a multi-class dataset, we have to choose what the hyperplane separates. ovr means ‘One vs Rest’, so it looks at one class and all the other also as one.



A 2D Representation of an SVM Model. The feature vectors in squares are the support vectors and the dotted lines the margins.

Random Forests

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set. Decision trees are a popular method for various machine learning tasks. Tree learning come closest to meeting the requirements for serving as an off-the-shelf procedure for data mining, because it is invariant under scaling and various other transformations of feature values, is robust to inclusion of irrelevant features, and produces inspectable models.

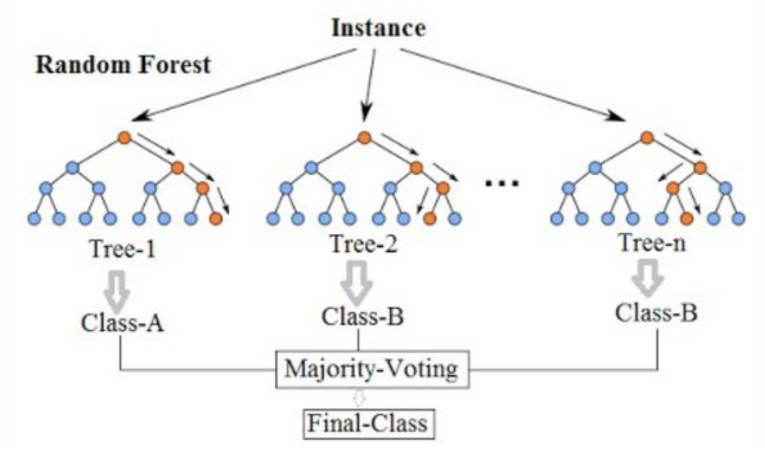
However, they are seldom accurate. In particular, trees that are grown very deep tend to learn highly irregular patterns: they overfit their training sets, i.e. have low bias, but very high variance. Random forests are a way of averaging multiple deep decision trees, trained on different parts of the same training set, with the goal of reducing the variance. This comes at the expense of a small increase in the bias and some loss of interpretability, but generally greatly boosts the performance in the final model.

I used the Sklearn Library for Random Forests

> RandomForestClassifier(n_estimators=240, max_depth=30, random_state=0)

- n_estimators: number of decision trees
- max_depth: The max depth of each decision tree.

These settings for the random forests were tested by me, having already tried various combination and I think they are close to the optimal ones. (but not the optimal)



Random Forest Decision Trees illustration

K-nearest Neighbors (KNN) Classifier with BoW and Tf-Idf

KNN is a lazy learning model, which means it doesn’t exactly create a model. The way it works is simple, it just takes the square root sum of all the Euclidean Distances between all the attributes of the test set feature vector with each feature vector in the Train Dataset. The feature vector with the smallest Euclidean distance from the test feature vector is considered as the closest neighbor, so the class that the test sample belongs, is its’ neighbor’s class. The K in KNN is the implementation where we choose the K (Integer > 0) closest neighbors, and pick the class where the majority of those neighbors have. For the implementation of KNN in this Project the optimal number of neighbors I found was **13.** The sklearn KNeighbors Classifier API was used.

Model Performance Evaluation

Performance Indicators

1. **Accuracy**

   Accuracy measures how many of the predicted classes of the test dataset were correctly classified. So it’s the fraction

   Accuracy = Correctly_Predicted_Classes / Total_Number_of_Test_Data

   e.g. if the test set has 100 feature vectors and we classify correctly the 92 of them, the total accuracy is 92%.

2. **Precision**

   Precision talks about how precise/accurate the model is out of those predicted positive, how many of them are actual positive. Precision is a good measure to determine, when the costs of False Positive is high.

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

3. Recall

Recall calculates how many of the Actual Positives our model capture through labeling it as Positive (True Positive). Applying the same understanding, we know that Recall shall be the model metric we use to select our best model when there is a high cost associated with False Negative.

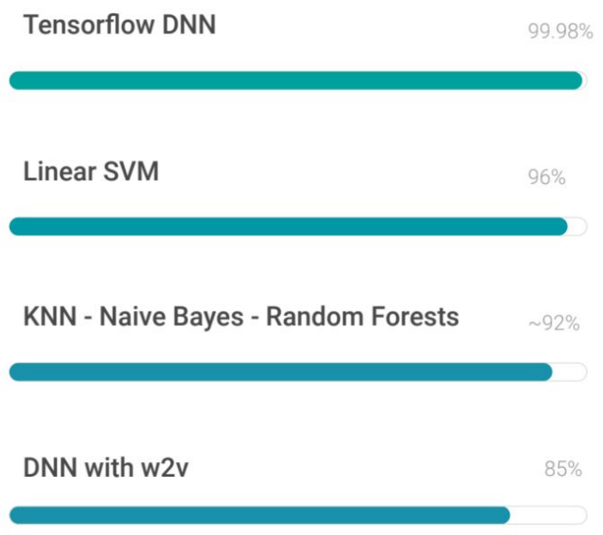
$$\text{Recall} = \frac{\textit{True Positive}}{\textit{True Positive} + \textit{False Negative}}$$

4. F1-Score

F1 Score might be a better measure to use if you need to seek a balance between Precision and Recall and there is an uneven class distribution. It’s just a simple fraction of precision and recall.

$$\text{F1} = 2 \times \frac{\textit{Precision} * \textit{Recall}}{\textit{Precision} + \textit{Recall}}$$

The accuracy scores of some tested ML models for the content-based classification, over the training set using 10-cross fold validation are shown below:



5.Analytics

WordCloud Creation

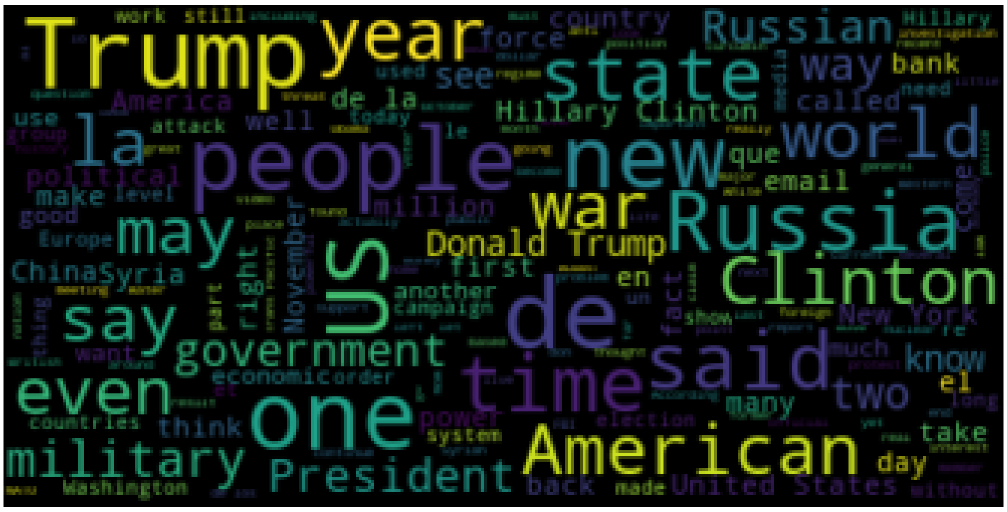
What are Word Clouds?

Word clouds (also known as text clouds or tag clouds) work in a simple way: the more a specific word appears in a source of textual data (such as a speech, blog post, or database), the bigger and bolder it appears in the word cloud.

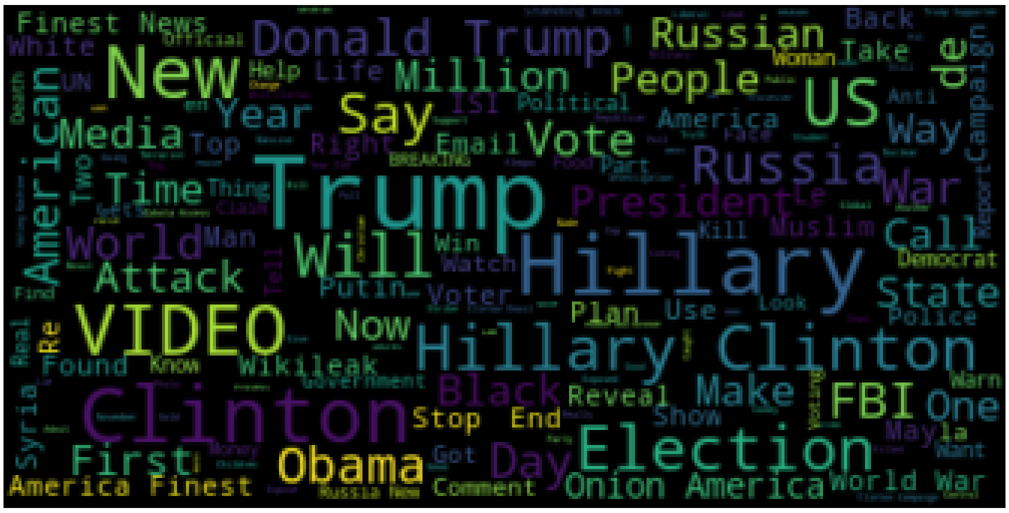
For my implementation I used the WordCloud Library (can be found here https://github.com/amueller/word_cloud).

A moderate cleaning takes places, which doesn't consider stop words as words, because they are really frequent, so they would appear a lot. Some words that don't have a very specific meaning are shown, but in order to tackle that issue, I would have to exclude each specific word, which is rather inconvenient.
The Word Clouds are presented below.

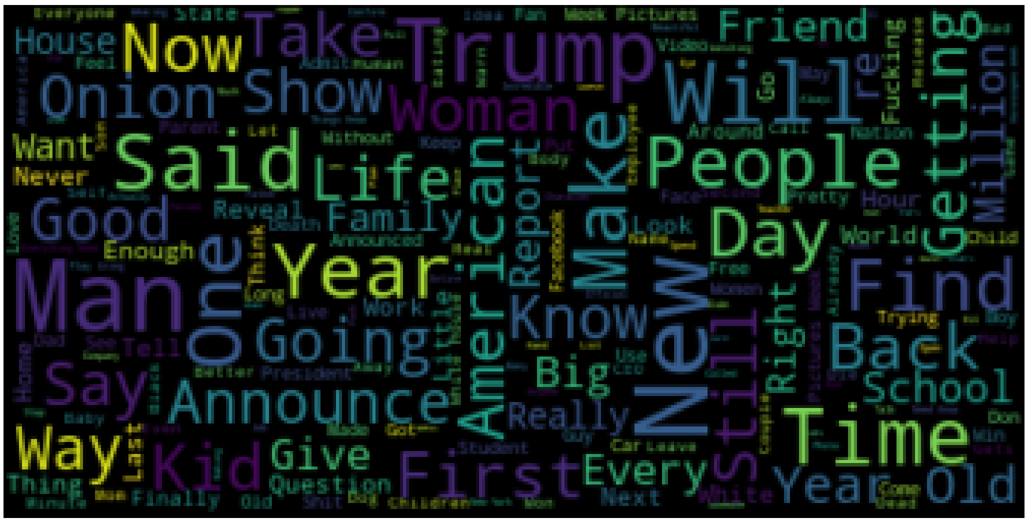
Kaggle Dataset #1



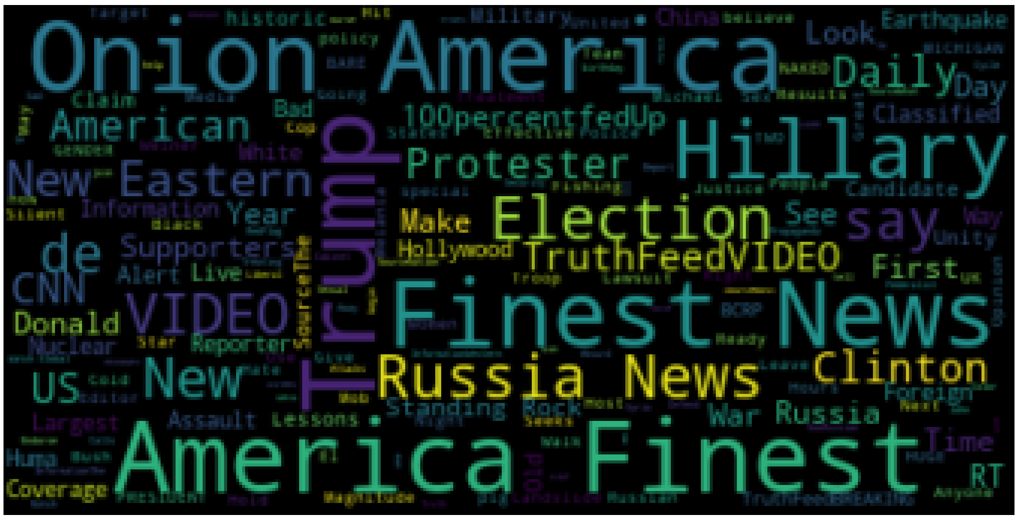
Kaggle Dataset #2



theOnion Dataset #1



theOnion Dataset #2



6.The website

Website Framework and Tools

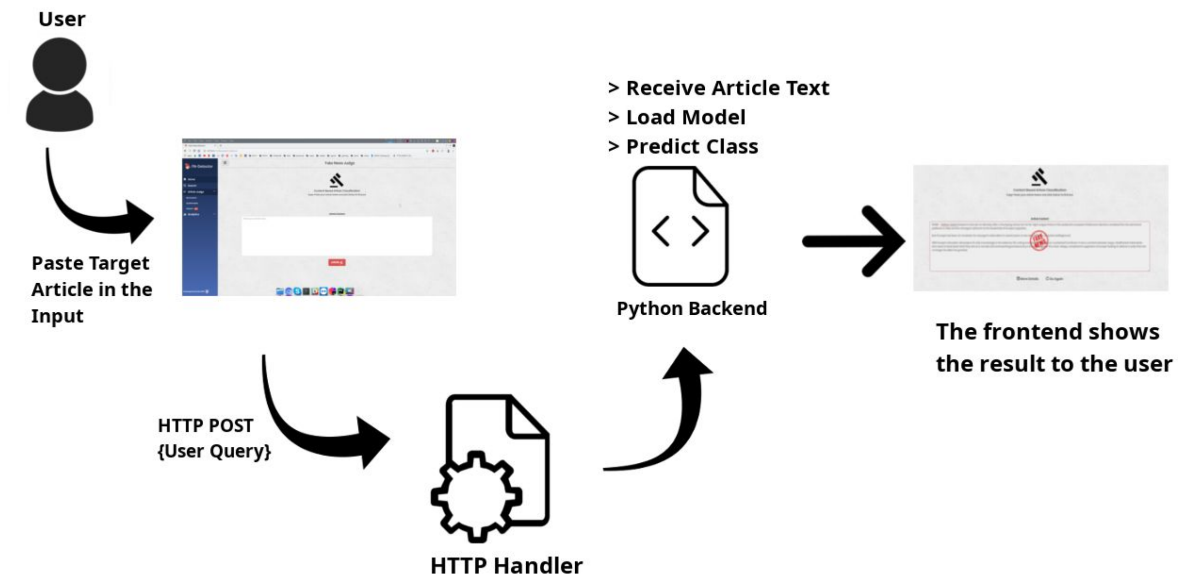
The Python Tornado web framework was used for the implementation of the website. This choice was made due to previous experience with the framework, as well as, the fact that the backend is in python, so it is easier to handle the ML models and the input data. Moreover, for the frontend, I used HTML, CSS, Javascript, JQuery and the Bootstrap Framework.

Project FIle Map

The files in the deliverable are:

- Datasets Directory: The directory that contains all four of the datasets used.
- Models Directory: Contains all the models and term vectors saved from the jupyter notebook code. (Most important: Tensorflow DNN Model)
- db_deploy Directory: Contains the handler of the Database (not used)
- static Directory: Contains all the CSS, Javascript, Image and font files of the frontend
- templates Directory: Contains all the HTML files of the frontend
- src Directory: Contains the main code
  - Notebooks: Contains the Jupyter Notebooks with the main code
    - meta_classification.ipynb: the code of the metadata-based classification (SVM Model etc.)
    - news_classification: Various test implementations for the content-based classification (SVM,KNN,Bayes etc.)
    - tensorflow_classifier: Contains the final DNN Model of the content-based classifier
    - untrusted_test: contains the code that created the list of untrusted websites
    - w2v_classifier: the w2v - svm implementation
    - wordcloud: the wordcloud creation
  - rest_services/handlers.py: the HTTP GET/POST Handlers that return the requested webpages
  - article_classification.py: When it receives a query from the user, from the content-based input form, it loads the keras model and classifies it accordingly
  - meta_classification.py: the same as the previous for the metadata query
  - untrusted_sites: loads the list with the untrusted sites
- init.py: Starts the Server and its services
- settings.py: various variable initiations

Usage Flowchart



7.Conclusions

I appreciate that I had the chance to experiment with Text Classification and NLP techniques. The thing that I didn't expect the most, was that really simple models like KNN and Naive Bayes, perform so well compared to really sophisticated models, like Deep Neural Networks with Gradient Descent, SVM etc.

The task of detecting if an article is fake or not is really tricky and needs further research in order to be functional in the real world. There is an imperative need to keep the training data of the ML models as fresh as possible, meaning that the latest fake news that pop up on the internet must be contained in the training dataset.

Finally as future work, I would like to experiment with Recurrent Neural Networks and LSTM, as they seem pretty promising for text classification.

8.References

[1] First Kaggle Dataset, https://www.kaggle.com/c/fake-news/data
[2] Second Kaggle Dataset, https://www.kaggle.com/mrisdal/fake-news#fake.csv
[3] Google’s pre-trained w2v model, https://code.google.com/archive/p/word2vec/