FINAL YEAR PROJECT

Detecting Twitter Bots Using Machine Learning

Author:
Marcell BATTA

Supervisor:
Dr. Lahcen OUARBYA

A thesis submitted in fulfilment of the requirements for BSc Computer Science Degree

May 17, 2019

Declaration of Authorship

I, Marcell BATTA, declare that this thesis titled, "Detecting Twitter Bots Using Machine Learning" and the work presented in it are my own. I confirm that:

- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

UNIVERSITY OF LONDON

Abstract

Computing Department

BSc Computer Science Degree

Detecting Twitter Bots Using Machine Learning

by Marcell BATTA

This project aims to classify Twitter accounts as either bots or human-made using machine learning along with a variety of parameters. This is achieved using core functionality of Python as well as popular libraries such as Numpy, Pandas and SKLearn. Some portion of Deep Learning was also explored briefly.

Acknowledgements

I'd like to thank my supervisor Lahcen for helping me out as much as he did. Calling me, answering me and everything else even during holidays. Until the very last day he was still helping me out. I am grateful that he helped out as much as he did regardless of how busy he was.

Contents

Chapter 1

Introduction

Social media has become an integral part of our lives in the past years as we spend more time online than ever before. Roughly 30% of our online time is spent on social media interactions, Twitter being one of them [1]. Twitter is arguably the largest source of news on the internet. This is due to the nature of information spreading on the platform through tweets and retweets. At this point, because of the scale, it is impossible to monitor it all to make sure everything is accurate and that there is no false information being spread. Due to it being impossible to be fully managed through human monitoring, a network of detection systems would be required.

1.1 Aim

The aim of this project is to create a program with an underlying algorithm that will attempt to classify a Twitter account as being under the control of a human or purely being run by some form of a script that someone wrote. There might be a misconception that all bot accounts are bad, which is untrue. There are plenty of examples of public bot accounts for things such as weather or news. Instead, the issues come with the ones that claim to be real individuals when they in fact are not. With the use of a program such as this, it is be possible to identify these bots by comparing their features like number of followers and number of accounts being followed and see if they match patterns of other real people or not.

Ideally the program shouldn't misjudge too often and should be a reliable way to identify false accounts from real ones. This would be done through a supervised machine learning algorithm which would be fed with as much data of previously labelled accounts as possible.

It's also important that the system uses features that give the best possible accuracy. Due to this there will be a lot of trial and error involved with trying to find the best possible combination of features and things such as number of trees to use or maximum depth of a given tree.

1.2 Motivation

The idea for the project came after I read an article on the effects of social media on politics and how it can be influenced. I found it very intriguing and this was shortly after I had begun to learn about machine learning and deep learning in my third year of university. That was when I knew that this would make for an interesting project.

1.3 Results

By the end of the project the system achieved results that were expected. All classifiers were within 80% to 95% accuracy overall.

1.4 Overview of report

The rest of this report is structured as follows;
Chapter 2 is about background research. It displays some of the areas in which issues arise along with some of the solutions or systems that others have come up with.
Chapter 3 contains the planning for the project. It describes the processes taken before beginning development to ensure everything would go as planned.
Chapter 4 contains the system design. This details the method, system requirements, data and languages used in the project.
Chapter 5 contains the implementation of the system with some main functions used in the programs. Each subsection in this chapter details a separate part of the implementation. All the code in there is briefly described. Testing is also found in this chapter.
Chapter 6 details the results from the systems and how these can be interpreted.
Chapter 7 discusses the overall finding in this report. It also shows the possibilities of future work added on and where this project could go.

Chapter 2

Background Research

This chapter contains the information found before beginning development of the program, along with some systems that are already available and a summary on them.

2.1 Politics

Politics is probably the biggest concern when it comes to these bot accounts. They are the reason why false information spreads so fast. This is because of the way Twitter works with its trending hashtags. These bots will tweet and retweet about important and most likely incorrect matters. They also make use of popular hashtags that basically define the topic of a tweet. This then leads to these malicious tags to become trending for everyone to see.

2.1.1 2016 US Elections

The 2016 elections in America was one of the, if not the biggest outburst of Twitter bots we have yet to see. It was found that by extrapolating some findings, roughly 19% of 20 million election related tweets originated from bots between September and October of 2016 [2]. According to the same study it was also found that around 15% of all accounts that were involved in election related tweets were bots. Now even though that is a lot of attention for these tweets containing false information, they will mostly only be seen by people who are already on the same side and agree. However, this doesn't rule out the affects. A study by the NBER(National Bureau of Economic Research) came to the conclusion that these bots were the cause of up to 3.23% of the votes that went towards Donald Trump [3]. This tells us that even if it's just marginal, it does still affect the outcomes.

The interesting part of all this is that the bots immediately went silent and disappeared as the election ended. The accounts though didn't get deleted but they simply went into hibernation waiting for their next bit of propaganda that needed to be spread. In 2017, 2000 of these bots reemerged to take part in the French and German elections as well, meaning they were run by the same people. They were discovered to make up for 1 in 5 election related tweets [4].

2.1.2 2018 US Mid-Term Election

Following the 2016 elections, the 2018 Mid-Terms were another prime target for Twitter bots. Before the voting took place, there were automated accounts trying to discourage people from voting. Of these, 10,000 were banned by Twitter. Even legislations were signed in an attempt to control the situation [5]. Interestingly, nearly

two weeks after the election day, there was still activity amongst these bots which accounted for a fifth of the #ivoted tweets [6]. The numbers recorded during this recent election compared to the one in 2016 was believed to be much lower. This could either be due to the reduced number of accounts being used or it could even be that the bots are now much more sophisticated and can remain undetected as they might be able to recreate human interactions and behaviour at a higher standard.

2.2 Twitter junk

Political issues aren't the only thing being caused by these bots. A study done at the University of Iowa has shown that through the third-party applications that Twitter allows its users to utilise can be, and is often abused in malicious ways such as phishing or even just spam. Twitter themselves have a way of dealing with these toxic accounts and do most of the time eventually ban them, however this study has found that 40% of the accounts that their algorithm detected as in some way benign were located about a month before Twitter took any action towards them [7]. As this does show that there are still improvements to be made even at Twitters end in terms of detection speed, we mustn't forget that there are many other parameters we must watch out for, some unknown outside of Twitter. A Twitter spokesperson wrote:

> "Research based solely on publicly available information about accounts and tweets on Twitter often cannot paint an accurate or complete picture of the steps we take to enforce our developer policies" [8]

2.3 Random Forest

Random forest is a supervised learning algorithm. This means the data must be labelled, otherwise the algorithm won't know what to do with it. It's a useful algorithm, as it can be utilized for both classification and regression problems. In order to understand and implement the random forest algorithm, we first need to know about its building blocks, the decision tree.
A decision tree is made up of an ensemble of branches and leaves. The branches contain the decisions, whilst the leaves determine the label that the tree believes the data belongs to. These decisions are made based on the features that best help us determine what label the data belongs to. A major downside to decision trees is that they suffer from overfitting when they become too deep with many branches.
This is where random forest comes in. As the name implies, it combines many decision trees into a forest like object where each trees outcome is thrown together and averaged out to get one answer. However, the clever thing it manages to do is the random feature selection. Each tree takes a limited number of random features from the original total and creates its own decision tree based on that. This helps it give a much more accurate result and mostly remove the overfitting aspect of the algorithm. The other form of randomness comes from the random subset of data selected with replacement for each individual tree, also known as bagging.
Once the subsets are decided upon and the trees are split up we must go through the nodes and decide on how to set these rules and which features to base them on. This is done using the gini impurity equation. It's the probability of any given node that a randomly selected sample would be incorrectly labelled if it was labelled by the distribution of samples in that node [9]. The gini impurity is worked out using

the following equation:

$$I_G(n) = 1 - \sum_{i=1}^{J}(p_i)$$

The gini impurity of a node n is 1 minus the sum over all the classes J of the fraction of examples in each class p_i squared [9]. At nodes beyond the root the gini impurity is additionally weighted by the fraction of points from its parent node. As it is the probability of incorrect labelling, we are looking for values as small as possible here. This is repeated throughout the algorithm recursively, finding the best possible values for the best features to pick until a given depth or if there is only one class' samples remaining.

To further help understand the algorithm it's crucial that we look at the pseudocode as that's much easier to translate into code.

Algorithm 1 Random Forest

Precondition: A training set $S := (x_1, y_1), \ldots, (x_n, y_n)$, features F, and number of trees in forest B.

```
function RandomForest(S, F)
    H ← ∅
    for i ∈ 1, ..., B do
        S^(i) ← A bootstrap sample from S
        h_i ← RandomizedTreeLearn(S^(i), F)
        H ← H ∪ {h_i}
    end for
    return H
end function
function RandomizedTreeLearn(S, F)
    At each node:
        f ← very small subset of F
        Split on best feature in f
    return The learned tree
end function
```

FIGURE 2.1: Random forest pseudocode

As previously mentioned, the algorithm works by creating a forest of decision trees. This means that for B number of trees we take a bootstrap or bagging sample of features from the original data and create these decision trees by splitting the nodes on the best features. Once this has been complete all the way down the trees recursively, we return the finished trees and decide based on majority vote which outcome is the most likely.

To give a visual representation of a decision tree it would look something like the following:

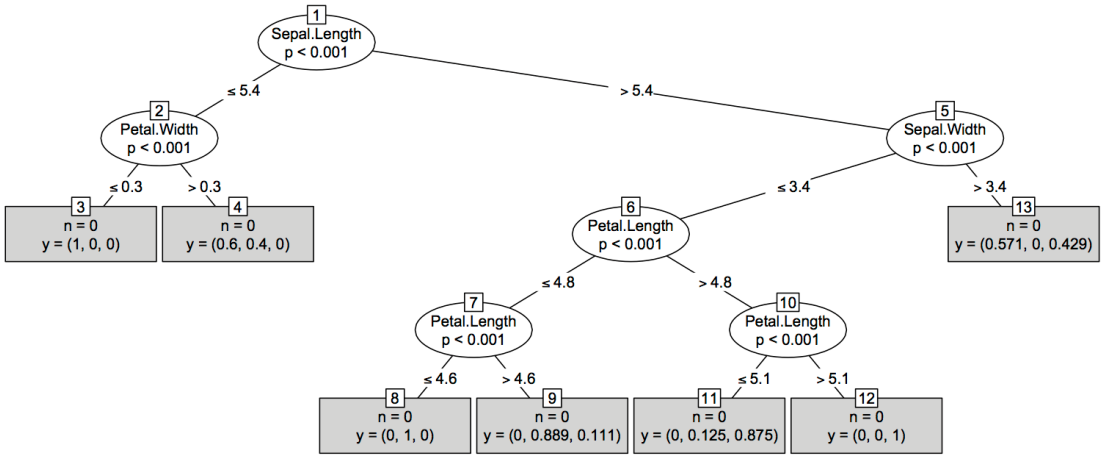

FIGURE 2.2: An example decision tree using the iris dataset

Here you can see that at each node there is a condition being made. This is the 'best' feature that is decided based on the earlier mentioned gini impurity being minimised. Then based on the value of these features for a given datapoint, the algorithm will traverse down a branch of the tree all the way to a root node. This then gives us our prediction for that one decision tree within the forest.

2.4 Deep Learning(Neural Networks)

Deep learning or more specifically artificial neural networks are a relatively new form of machine learning that is more of a framework than an actual algorithm. ANN's aren't really a specific algorithm, more of a general decision making model. They do not need any sort of rules fed into them or being told what to do. Just some data and it begins training immediately. To understand the basics of ANN's we have to look at what they are made of. An ANN is made up of a multitude of layers. In these layers there are a whole bunch of perceptrons.

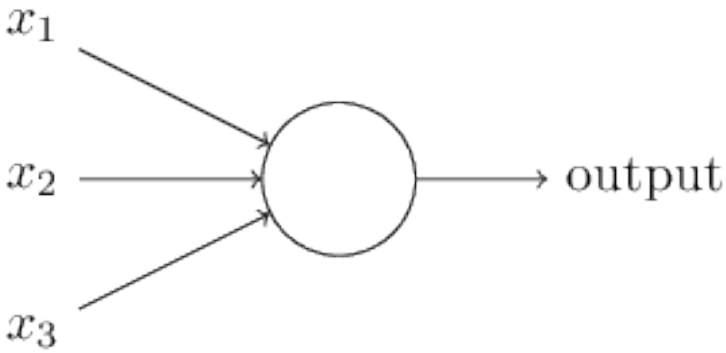

FIGURE 2.3: A single perceptron

The inputs are summed up with some weights as their coefficients and then compared to a threshold. If greater than said threshold then the output is 1 else it's 0. This rule can be written mathematically such as:

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Once you start training your model, the weights w_j start to change to match the labels on the training data. As the layers of perceptrons goes on, the more complicated these values are in theory. They aren't just simple decisions any more, but instead represent the decisions made based on all the previous layers too.

2.5 Existing Systems

There are a handful of algorithms or programs that have been designed to detect these bots. Most of them tend to use a machine learning algorithm as a way to classify accounts and tell them apart from each other, while others have attempted to use deep neural network architectures such as long short-term memory(LSTM).

2.5.1 Tweetbotornot

Tweetbotornot is a package built in R that uses machine learning to classify Twitter accounts. It has two 'levels'. One for users where it uses information related to an account such as location or number of followers. The other is a tweet-level which checks for details like hashtags, mentions or capital letters out of the user's more recent 100 tweets. This could prove useful when testing my program to compare results as the accuracy of this library is 93.8 percent. As this is just a package created, it doesn't have any user-interface program built around it or anything like that, therefore it is unusable by anyone not knowledgable in R.

2.5.2 DeBot

DeBot is a fully functioning python API for bot detection on Twitter. It has the ability to obtain a list of bots already detected by DeBot, or just simply checking individual accounts. You can also get a list of bots which appear in the archives more than a given number of times. They can even be requested based on topics. It does however have a somewhat working built in search mechanism on its website.

Chapter 3

Planning

This section contains all the things related to planning from all the way at the beginning until the very end.

3.1 Gantt chart

Gantt charts are a nice way to keep track of your time available over a whole project. It really helps understand the scope of things and to give a better estimate on how to split up your time into smaller chunks.

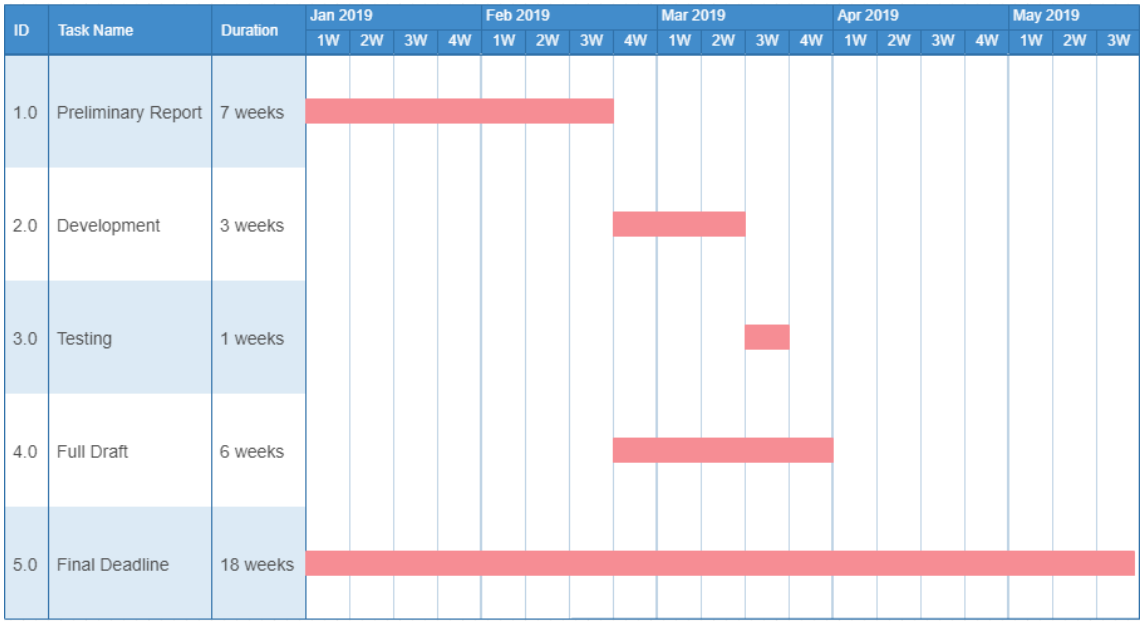

FIGURE 3.1: Gantt chart

Here you can see the gantt chart created for this project. It has a rough sketch of the bigger tasks, nothing too precise, as it is helpful enough this way to keep track of the weeks overall.

3.2 Progress Logs

The weekly progress logs act as a weekly sprint. In them I write down what I did, what I wanted to do and what I will do by next week. This helps keep track of the more short term week-by-week goals. I also find the recaps at the end of the week to be quite helpful to remind myself of what I did and what I still need to do.

3.3 Version Control

Version control is an essential part of any project. I'm using GitHub, but there are many others out there that do the same job. It helps mainly because of the ability to go back to previous versions whenever needed, for example when something goes horribly wrong.

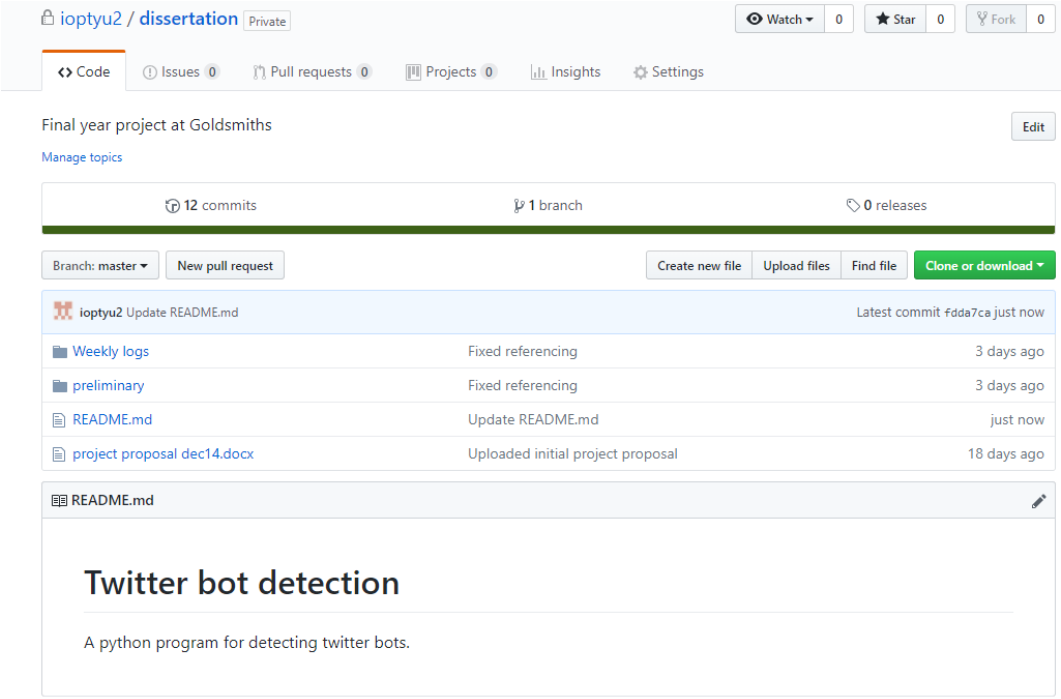

FIGURE 3.2: Github Repository

Chapter 4

System Design

This chapter is about the methodology behind the program and some of the features and requirements.

4.1 Method

The core functionality of the system is the classification and prediction of said classes. Different machine learning algorithms are used to predict the bots. This task can be treated as either a regression or binary classification problem. In the end the system is a binary predictor therefore the latter makes more sense.

4.2 System Requirements

In order to achieve the aims of the system, there are a few things the program will need to do.

1. The system must be able to determine whether an account is a bot or not.
2. Display the likelihood that an account is a bot or not.
3. The system should evaluate the results.
4. The system should compare results with the other classifiers.

4.3 Algorithm

There are several algorithms in the system running together. These classifiers can be used to compare results with each other to better evaluate them. The different algorithms are; random forest implemented from the ground up and using the sklearn library, as well as an artificial neural network. It's important to keep in mind the accuracy versus the performance of the system. Achieving good results will come due to finding a balance between the two.

4.4 Data

The data will have to be kept the same throughout for a fair comparison. As it would take an extended period of time to obtain all this data using the system, it's much easier if it uses data already gathered from the internet.

To start off with, the system will use only numerical data from Twitter accounts, such as number of followers and number of tweets. Using text like the actual content

of tweets would be very complicated and most likely require a more complex system then the one designed. If the predictions aren't great then using more or different features can be considered.

4.5 Language

The language being used to implement the system in it's entirety will be Python. The main reason for selecting this as the language is because of the libraries that can be utilised by it.

4.5.1 Sklearn

Sklearn is a python library designed specifically for machine learning. It has a huge range of supervised and unsupervised learning algorithms.

4.5.2 Keras

Keras is a high-level neural networks API. It runs on top of tensorflow. It is the basis of the neural network designed for this system. It provides easy to create models that also perform very well and fast.

4.5.3 Numpy

Numpy is a multi-dimensional array package. It's what the system uses for most of its arrays and data manipulation. It allows the system to combine and split multi-dimensional arrays with ease.

4.5.4 Matplotlib

Matplotlib is only used within the neural network for evaluation of the system. It makes it very simple by drawing plotted graphs of validation accuracy and loss values.

Chapter 5

Implementation

This chapter is about the implementation of the system. This will include functions of the code along with an explanation on what they do. For the results from all the implementations, please refer to *Chapter 6*.

5.1 Preprocessing

The first thing that needed to be done was to get the data and clean it up so it's ready to be fed to the algorithm. The *import_data* function is used to process all the data from various types of text files and convert them into separate variables to then be parsed on for further cleaning.

Code 5.1: Import data function

```
1 def import_data():
2
3     varol_users = np.loadtxt("Z:\ioptyu2\Desktop\gitDissertation\local\
      Data\\varol_2017.dat")
4
5
6     col_name = ["UserID","CreatedAt","CollectedAt","Followings","Followers
      ","Tweets","NameLength","BioLength"]
7     bot_users = pd.read_csv("Z:/ioptyu2/Desktop/gitDissertation/local/Data
      /social_honeypot_icwsm_2011/content_polluters.txt",
8                                     sep="\t",
9                                     names = col_name)
10
11
12    legit_users = pd.read_csv("Z:/ioptyu2/Desktop/gitDissertation/local/
      Data/social_honeypot_icwsm_2011/legitimate_users.txt",
13                                      sep="\t",
14                                      names = col_name)
15
16    return [varol_users,bot_users,legit_users,col_name]
```

Once the function is called in the appropriate place, the data is then sliced from the original size to fit the requirements of the algorithm. In the example below we only want the first 1000 entries. We also don't need the first 4 columns, so those are also being sliced.

Code 5.2: Cleaning/adding labels

```
1 #cleaning up data(picking out useful rows)
2 df = preprocessing.import_data()
3 bots = df[1].values[:1000,3:].astype(int)
4 legit = df[2].values[:1000,3:].astype(int)
5
```

```
6
7 #adding label, bots=1 legit=0
8 bots = np.hstack((bots,np.ones((bots.shape[0],1))))
9 legit = np.hstack((legit,np.zeros((legit.shape[0],1))))
10
11 dataset = np.vstack((bots,legit))
```

5.2 Random Forest implementation

This is the main machine learning algorithm implemented for the classification. It's divided into several main functions.

The *random_forest* function, as shown in Code 5.1, is main predictor that provides the predictions for the algorithm using all the other functions. In here the data is split up into samples as well as building the trees and using them for predictions.

Code 5.3: Random Forest main function

```
1 def random_forest(train, test, max_depth, min_size, sample_size, n_trees,
      n_features):
2     trees = list()
3     for i in range(n_trees):
4         print(i)
5         sample = subsample(train, sample_size)
6         tree = build_tree(sample, max_depth, min_size, n_features)
7         trees.append(tree)
8     predictions = [bagging_predict(trees, row) for row in test]
9     return predictions
```

The *subsample* function splits the dataset into a specific amount of smaller sets based on the ratio given as the parameter. These smaller sets are appended to a list which is then returned out of the function.

Code 5.4: Function for splitting the dataset

```
1 def subsample(dataset, ratio):
2     sample = list()
3     n_sample = round(len(dataset) * ratio)
4     while len(sample) < n_sample:
5         index = randrange(len(dataset))
6         sample.append(dataset[index])
7     return sample
```

The *build_tree* function finds the best split for a given dataset and then splits the data into a binary tree then returns the root node.

Code 5.5: Function for building a decision tree

```
1 def build_tree(train, max_depth, min_size, n_features):
2     root = best_split(train, n_features)
3     split(root, max_depth, min_size, n_features, 1)
4     return root
```

The *split* function is responsible for the splitting of the nodes to create the trees. As long as there are child nodes of the current node remaining and the maximum depth hasn't been reached yet, we recursively keep going through the function.

Code 5.6: Split a given node into its children

```
def split(node, max_depth, min_size, n_features, depth):
    left, right = node['groups']
    del(node['groups'])
    #check for no splits
    if not left or not right:
        node['left'] = node['right'] = terminal(left + right)
        return
    #check if max depth
    if depth >= max_depth:
        node['left'], node['right'] = terminal(left), terminal(right)
        return
    #go down left child
    if len(left) <= min_size:
        node['left'] = terminal(left)
    else:
        node['left'] = best_split(left, n_features)
        split(node['left'], max_depth, min_size, n_features, depth + 1)
    #go down right child
    if len(right) <= min_size:
        node['right'] = terminal(right)
    else:
        node['right'] = best_split(right, n_features)
        split(node['right'], max_depth, min_size, n_features, depth + 1)
```

The *terminal* function is a small section used in the *split* function. It is used to create a terminal or final node.

Code 5.7: Create a terminal node

```
def terminal(group):
    outcomes = [row[-1] for row in group]
    return max(set(outcomes), key=outcomes.count)
```

The *best_split* function helps find the best value to split the data at, by calculating the various values of the gini index. It finds the value at every split possible and then keeps the one with the lowest final value and saves that as the index and value.

Code 5.8: Find the best split using gini index

```
def best_split(dataset, n_features):
    class_values = list(set(row[-1] for row in dataset))
    b_index, b_value, b_score, b_groups = 999,999,999,None
    features = list()
    while len(features) < n_features:
        index = randrange(len(dataset[0])-1)
        if index not in features:
            features.append(index)
    for index in features:
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = get_gini(groups, class_values)
            if gini < b_score:
                b_index, b_value, b_score, b_groups = index, row[index],
    gini, groups
    return {'index':b_index, 'value':b_value, 'groups':b_groups}
```

The *test_split* function is where the final splitting happens. The parameters tell the function which feature it should be focusing on and what value to split the dataset at. It splits the data into the two groups and returns them.

Code 5.9: Split the data by a value

```
def test_split(index, value, dataset):
    left, right = list(), list()
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left,right
```

The *get_gini* function returns the gini index of the given group. This is done using the equation given by the gini index which can also be seen in *Chapter 2.3*.

Code 5.10: Calculate the gini index for a split

```
def get_gini(groups, classes):
    instances = float(sum([len(group) for group in groups]))
    gini= 0.0
    for group in groups:
        size = float(len(group))
        if size == 0:
            continue
        score = 0.0
        for class_val in classes:
            p = [row[-1] for row in group].count(class_val) / size
            score += p*p
        gini += (1.0 - score) * (size / instances)
    return gini
```

The *bagging_predict* function is what provides the predicting for the algorithm. It uses the *predict* function to predict the class of a given entry for each tree that we have. It then returns the class that there is more of outputted.

Code 5.11: Make a list of predictions

```
def bagging_predict(trees, row):
    predictions = [predict(tree, row) for tree in trees]
    return max(set(predictions), key=predictions.count)
```

The *evaluate* function is used to determine how well the algorithm is doing. It uses cross validation to essentially test it out. This is done by splitting the dataset up into training and testing sets. The training sets are used to optimise the algorithm, while the testing set is used to see how well it's doing. It returns how much of the test set it correctly predicts and returns that as the score.

Code 5.12: Evaluate algorithm

```
def evaluate(dataset, algorithm, folds, *args):
    folds = cv(dataset,folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_calc(actual, predicted)
        scores.append(accuracy)
    return scores
```

The *cv* function is what completes the *evaluate* function through the use of cross validation. This is done by splitting the dataset into folds and then returning it ready to be used for training and testing.

Code 5.13:Function for cross validation

```
def cv(dataset,folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset)/folds)
    for i in range(folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split
```

The *accuracy_calc* function calculates the accuracy of the algorithm by comparing the actual class with the predicted ones and seeing how many of them match. It returns the accuracy as a percentage.

Code 5.14: Calculating accuracy

```
def accuracy_calc(actual, predicted):
    acc = 0.0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            acc += 1.0
    return acc / len(actual) * 100.0
```

The *predict* function is used to return the actual prediction for a given row. The node is the current tree being used to predict and the row is the entry that we are trying to get a prediction for. It recursively makes its way down the tree and arrives at one of the end nodes where it returns the value in that node. To read more about how the predictions as well as the trees are formed, refer to *Chapter 2.3*

Code 5.15: Make a prediction

```
def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']
```

5.3 Sklearn implementation

Here is the implementation of random forest using the Sklearn library. It's comprised of 3 main functions.

The *random_forest* function uses Sklearn to create and fit a random forest algorithm to the data provided. Once the data is fitted, the predictions using the *predict* function are stored and returned.

Code 5.16: Main random forest function

```
def random_forest(n_trees, max_depth, n_features):
    rf = RandomForestRegressor(n_estimators = n_trees, max_depth =
    max_depth, max_features = n_features, bootstrap = True)
    rf.fit(X_train, Y_train)
    predictions = rf.predict(X_test)
    save_tree(rf)
    return predictions
```

The *save_tree* function is used to save a given decision tree as a png with all the values and features included. Examples of this can be seen in the *results* section of the report.

Code 5.17: Saving graph

```
def save_tree(rf):
    tree = rf.estimators_[1]
    export_graphviz(tree, out_file = 'tree.dot', feature_names =
    feature_list, rounded = True, precision = 1)
    (graph, ) = pydot.graph_from_dot_file('tree.dot')
    graph.write_png('tree.png')
    return
```

The *accuracy* function is used to show the accuracy of the predictions. Once calculated it prints the final percentage.

Code 5.18: Getting prediction accuracy

```
def accuracy(predictions):
    for i in range(len(predictions)):
        if predictions[i] < 0.5:
            predictions[i] = 0.0
        else:
            predictions[i] = 1.0
    accuracy = 0.0
    for i in range(len(predictions)):
        if predictions[i] == Y_test[i]:
            accuracy += 1.0
    accuracy = accuracy / len(predictions) * 100
    print("Accuracy: ",accuracy, "%")
    return
```

5.4 Artificial Neural Network(ANN)

The following consists of the main parts of the ANN that was built for binary classification.

This code snippet is responsible for creating the model and its layers. They all have an output dimension given along with what activation functio is being used. The final layer is what gives us our predicted labels using a sigmoid function. The loss function used to optimise the model is binary cross entropy.

Code 5.19: Create/compile model

```
model = models.Sequential()
model.add(layers.Dense(64, activation = "relu"))
model.add(layers.Dense(32, activation = 'relu'))
model.add(layers.Dense(2, activation = 'sigmoid'))


model.compile(optimizer = 'adam',
              loss = 'binary_crossentropy',
              metrics = ['accuracy'])
```

Once the model was created, it was time to fit the data to it. In *5.20* you can see the model being fit to the training data along with the hyper-parameters being provided in the form of epochs and batch size. These two can be changed to try and get different results. The model is then validated using the validation data and the results are stored in the variable *history*.

Code 5.20: Fit data to model

```
epochs = 1000
batch_size = 16
history = model.fit(x_train,
                    y_train,
                    epochs = epochs,
                    batch_size = batch_size,
                    validation_data = (x_val, y_val))
```

5.5 Testing

As this isn't a system with a user interface or such, testing goes a little differently. For the base random forest implementation, cross validation was used for testing. In the sklearn algorithm and the neural network, a validation set was used to keep track of how the training of the model went.

5.5.1 Cross Validation

Cross validation works by splitting up the training data we have into n parts, based on the selected number of folds. The data is then split into that many equal pieces. One part is stored as validation and the rest is used for training. Once the training is complete, the validation dataset can be used to check how well the model predicts the classes. This is repeated several times until each separate chunk of data has been used as validation. Then looking at the several accuracies from the iterations of training and predicting, the system averages it out and that is the total accuracy of the model.

This is especially useful when there isn't much data available, since it is essentially turning a dataset into many smaller ones that have merely one portion of it missing. For the implementation of cross validation refer to *figure 5.13*

5.5.2 Validation set

For the other two models(sklearn & NN) all that was used to test it was a single validation set. This basically does the same thing as one iteration of cross validation. It stores one part of the dataset to test the model one it is done training.

Chapter 6

Results

This chapter contains the results from all the implementations provided with some assessment of them along with any interesting findings in the process. There will also be comparisons where appropriate as well as trying to figure out why exactly the results are the way they are.

It's important that before looking at the results there is an acknowledged method of deciding how *good* the results are. The first obvious unit of measurement would be accuracy. To see how accurately the designed model predicts the classes of the data. This is the primary value needed to be as high as possible to avoid mistakes. The secondary feature would be time taken to run. This isn't as much of an issue in this experimental setting, however if this sort of system were to be attempted to integrate into other live systems, it is definitely a variable to keep in mind.

There are a handful of tunable parameters for all the systems designed. To illustrate exactly which systems contain which parameters refer to the following table:

Parameter	Description	Used in
Number of features	The maximum number of features to be considered when building a tree	Random forest & Sklearn
Maximum depth	The maximum depth a tree can have along any path	forest & Sklearn
Minimum size	The minimum number of nodes required to be present on both sides after a split for it to occur	Random forest
Sample size	The ratio of the subsample that we pick for the bagging process of building a tree	Random forest
Number of trees	The number of trees built in the forest	forest & Sklearn
Number of folds	The number of folds used in cross validation	Random forest
Epochs	The number of learning cycles done for training	Neural network
Batch size	The number of training examples used in one iteration	Neural network

Table 6.1: Table of tunable parameters

6.1 Random Forest without Sklearn

The following results are a varying range of accuracies based on some of the parameters being changed around. Most of the parameters have been altered at times to

see what difference they make and seeing what combination of parameters give the best results.

Results for random forest implementation without sklearn:

Trees	Maximum depth	Minimum size	Features	Folds	Sample size	Accuracy
2	5	1	$\sqrt{features}$	2	1.0	90.7%
5	5	1	$\sqrt{features}$	2	1.0	92.0%
10	5	1	$\sqrt{features}$	2	1.0	92.4%
2	10	5	$\sqrt{features}$	2	1.0	92.8%
5	10	5	$\sqrt{features}$	2	1.0	96.0%
10	10	5	$\sqrt{features}$	2	1.0	96.1%
2	10	1	$\sqrt{features}$	2	1.0	93.7%
5	10	1	$\sqrt{features}$	2	1.0	96.0%
10	10	1	$\sqrt{features}$	2	1.0	97.3%
2	10	1	$\sqrt{features}$	2	0.5	91.5%
5	10	1	$\sqrt{features}$	2	0.5	95.2%
10	10	1	$\sqrt{features}$	2	0.5	95.4%
2	5	1	$\sqrt{features}$	5	1.0	90.5%
5	5	1	$\sqrt{features}$	5	1.0	92.5%
10	5	1	$\sqrt{features}$	5	1.0	92.3%

Table 6.2: Table of random forest implementation results

6.1.1 Interpretation of results

Overall, the results have a good, somewhat expected range based on the other systems researched on the internet. Due to the heavy performance of the system, it was only possible to test the number of trees up to 10. It was clear out of the results that the more trees the better, however this is most likely only the case early on. At some point adding more trees won't really change the overall predictive power of the forest. The tree parameter was one of the two values that had the largest effect on the speed and runtime of the algorithm, with the other being number of folds.

For the maximum depth parameter, out of the two options, the larger one seemed to do a much better job. It's important to keep in mind that adding too much depth to the trees would cause overfitting due to the tree splitting into many paths with very little objects per node.

When it came to minimum size, changing it from 1 to 5 showed a slight drop in performance. The only beneficial part of increasing this parameter would be in combination with maximum depth. This is due to the minimum size limiting the thin spread out trees from having too many branches with just a few objects going through them.

The number of features to be considered by the trees was left unchanged because of how small it already was, it would only make the predictions worse.

The folds' results were quite surprising as increasing the folds actually made the accuracy worse. It's the second parameter that had a huge effect on the speed, due to more folds causing the algorithm to loop that many more times.

The sample size caused the accuracy to decrease when lowered closer to 0, which is expected as all it does is essentially reduce the number of training data used when building the decision trees.

Overall with all of the results giving at least 90% accuracy was quite impressive.

6.2 Random Forest with Sklearn

The following section displays the results from the system built using the sklearn library. The tests were concluded in similar ways to the other implementation. The results are as follows:

Trees	Maximum depth	Features	Accuracy
2	10	$\sqrt{features}$	87.3%
5	10	$\sqrt{features}$	88.3%
10	10	$\sqrt{features}$	88.9%
50	10	$\sqrt{features}$	88.8%
100	10	$\sqrt{features}$	89.0%
500	10	$\sqrt{features}$	89.1%
1000	10	$\sqrt{features}$	89.2%
2	20	$\sqrt{features}$	84.9%
5	20	$\sqrt{features}$	87.4%
10	20	$\sqrt{features}$	88.3%
50	20	$\sqrt{features}$	88.4%
100	20	$\sqrt{features}$	88.6%
500	20	$\sqrt{features}$	89.0%
1000	20	$\sqrt{features}$	88.7%

Table 6.3: Table of sklearn implementation results

6.2.1 Interpretation of results

The first thing to note about this table is that the range of trees are much higher compared to the previous. This is due to the library running much more efficiently, allowing to create more dense forests with a relatively small amount of time required. There are less tunable parameters here compared to the implementation without sklearn. The accuracies are much closer and there is a slower increase. The overall range is somewhat lower, which is surprising.

The number of trees has a clear effect on the accuracy just not that much in the later amounts. This is probably due to the fact that there aren't that many features to select from, so most of the later trees end up being duplicates and are predicting the same thing as the other ones. It seems that both systems support the idea that the number of trees make a difference when the overall tree count is quite low.

Once the system is done training and predicting, it then saves one of the decision trees from the forest as an image so we can see what's happening exactly. An example of that can be seen below:

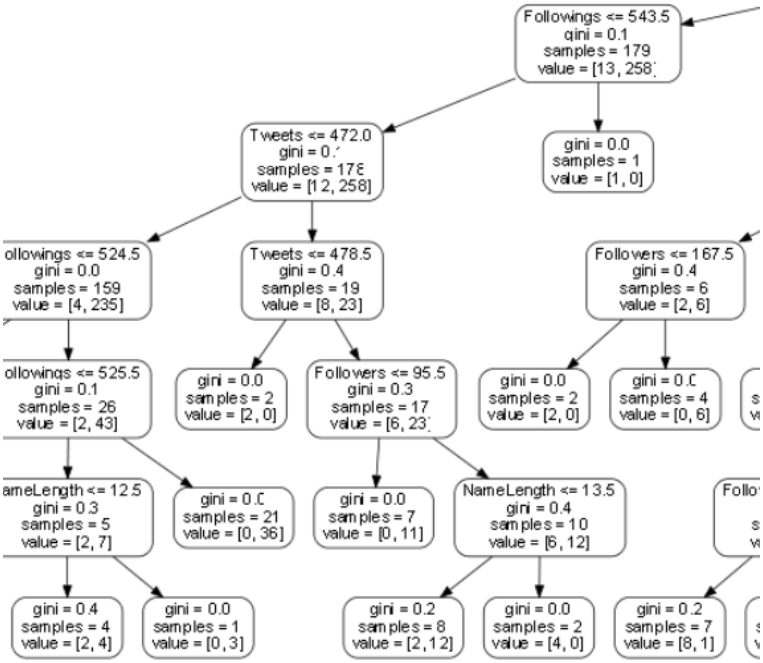

FIGURE 6.1: A section of a decision tree

6.3 Neural Network(NN) model

The NN model works completely different compared to the random forest algorithms looked at until now. There are only two hyper-parameters, epochs and batch size. NN's tend to overfit after many epochs so it was important to find this point. The strategy was to first run the model for a large amount of epochs and see where the overfitting begins to happen, then only train the model for that specific length to get the best possible prediction.

Here are the results of the accuracy and loss for both training and validation sets over 1000 epochs:

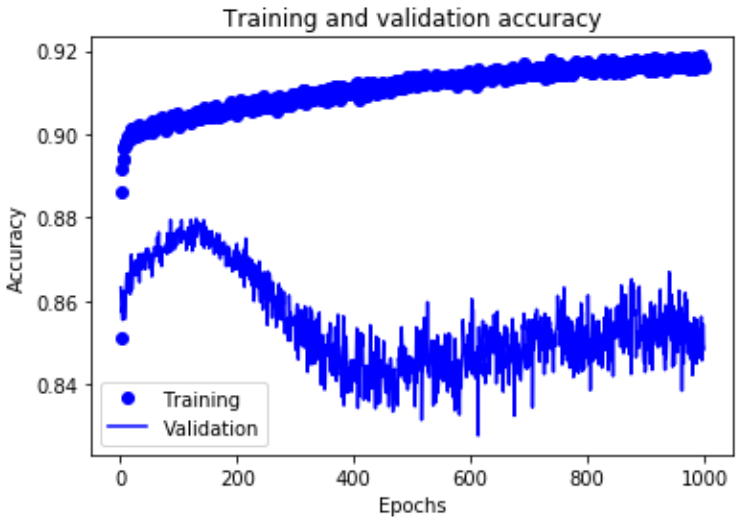

FIGURE 6.2: Accuracy over 1000 epochs

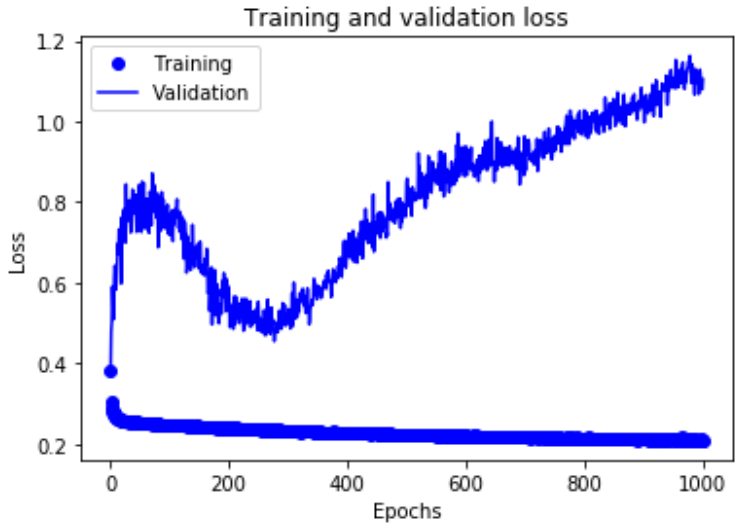

FIGURE 6.3: Loss over 1000 epochs

In *figure 6.1* the accuracy can be seen over the many epochs. The main focus is on the validation plots as that's the part that tells us how good the model is doing predicting new data. Initially the accuracy increases and around 150 epochs it cuts off and starts to decrease. That is roughly the point where overfitting starts to occur. We can confirm this by looking at the loss graph.

When it comes to the other graph, *figure 6.2* we see the loss function, which we are trying to minimise to get rid of the errors. This figure is almost an inverted version of the accuracy one. The validation loss initially shoots up and then begins to decrease as the model is improving itself and adjusting to the data. At around 250 epochs in the validation loss begins to increase indefinitely. This is another major sign of overfitting and is telling us to stop.

Using these two graphs we need to find a decent stopping point for when we retrain the model with a limited number of epochs then predict the labels of the test data that we put aside from the beginning.

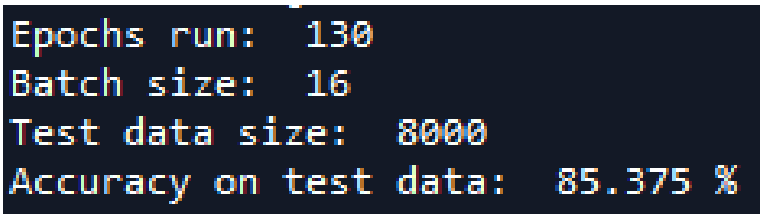

FIGURE 6.4: Accuracy on test data after 130 epochs

Figure 6.3 shows the result of retraining the model for 130 epochs, then predicting the classes for the test data and comparing it to the actual labels given, which gives us 85.4% accuracy. This is close to sklearn implementation but quite far off the other one, almost 10% difference. Overall it's not a bad model as it outperforms randomly assigning the data labels which would result in a basic 50% accuracy. The good thing about NN is it trains relatively fast and with a lot of data. This means potential future updates would be easy to complete compared to the random forest systems which took much longer.

Chapter 7

Conclusion

This chapter concludes the report with a conclusion of everything done, what went well and what went wrong. It also includes any possiblities of future work and how this project could be expanded.

7.1 System successes

Overall the system, based on the results, seems to be a success. It has reached most if not all initial requirements and achieved good accuracies on predicting labels. Along with the good accuracies, having decently fast training times is also a success. Another particularly good thing is being able to save an image of a decision tree. This allows us to view it anytime and really get a good idea of what's actually happening within the system.

7.2 System failures

An issue was the neural network model slightly underperforming compared to the other algorithms. This is somewhat made up for by the boost in performance compared to the others.

The data the system relies on for training and testing is a little aged now which could lead to it not predicting as accurately in a live scenario.

7.3 Future work

There are many things that could be done to further develop this system, here are just a few.

7.3.1 Algorithms

There is always the option to add more machine learning algorithms into the system for comparison. It will provide more options and could potentially get better results.

7.3.2 Combining algorithms

It's possible to build a system that would perhaps combine some mixture of algorithms or layer them, so they work in unison and sort of help each other reach better results. This would be quite a large expansion and would require in-depth knowledge of the algorithms and a lot of research.

7.3.3 Expanding features

Adding more features to the current list would help increase the likelihood that these algorithms find patterns within the data that help them predict the labels correctly. This would require more data with different features.

7.3.4 Tweets

This system could be further developed to not only classify Twitter accounts but also individual tweets. This could lead to finding interesting patterns within tweets, such as tweets made by a bot are more likely to be retweeted by bots then people.

7.3.5 Twitter integration

The ultimate goal would be something like getting this sort of system integrated into Twitter directly. This could lead to bots being removed much more accurately and faster than currently through reports and such.

Bibliography

[1] Katie Young. *Social Media Captures Over 30 percent of Online Time*. URL: https://blog.globalwebindex.com/chart-of-the-day/social-media-captures-30-of-online-time/.

[2] Alessandro Bessi and Emilio Ferrara. "Social bots distort the 2016 U.S. Presidential election online discussion". In: *First Monday* 21.11 (2016). ISSN: 13960466. DOI: 10.5210/fm.v21i11.7090. URL: https://firstmonday.org/ojs/index.php/fm/article/view/7090.

[3] Yuriy Gorodnichenko, Tho Pham, and Oleksandr Talavera. *Social Media, Sentiment and Public Opinions: Evidence from #Brexit and #USElection*. Working Paper 24631. National Bureau of Economic Research, 2018. DOI: 10.3386/w24631. URL: http://www.nber.org/papers/w24631.

[4] Denise Clifton. *Twitter Bots Distorted the 2016 Election—Including Many Likely From Russia*. URL: https://www.motherjones.com/politics/2017/10/twitter-bots-distorted-the-2016-election-including-many-controlled-by-russia/.

[5] BBC. *US mid-terms: Twitter deletes anti-voting bots*. URL: https://www.bbc.co.uk/news/technology-46080157.

[6] The Economic Times. *Thousands of Twitter bots active during 2018 US mid-term elections*. URL: https://economictimes.indiatimes.com/news/international/world-news/thousands-of-twitter-bots-active-during-2018-us-mid-term-elections/articleshow/67850597.cms.

[7] Shehroze Farooqi and Zubair Shafiq. *Measurement and Early Detection of Third-Party Application Abuse on Twitter*. URL: http://homepage.divms.uiowa.edu/~sfarooqi/Files/Farooqi-AbusiveTwitterApplications.pdf.

[8] Andy Greenberg. *TWITTER STILL CAN'T KEEP UP WITH ITS FLOOD OF JUNK ACCOUNTS, STUDY FINDS*. URL: https://www.wired.com/story/twitter-abusive-apps-machine-learning/?mbid=social_twitter&utm_brand=wired&utm_campaign=wired&utm_medium=social&utm_social-type=owned&utm_source=twitter.

[9] Will Koehrsen. *An Implementation and Explanation of the Random Forest in Python*. URL: https://towardsdatascience.com/an-implementation-and-explanation-of-the-random-forest-in-python-77bf308a9b76.

[10] Jason Brownlee. *Machine Learning Mastery*. URL: https://machinelearningmastery.com/implement-random-forest-scratch-python/.

[11] *What are Random Forests*. URL: https://www.python-course.eu/Random_Forests.php.

[12] Matthew wb. *Random Forests*. URL: http://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/ensembles/RandomForests.pdf.

[13] Brian David Eoff Lee Kyumin and James Caverlee. *Seven Months with the Devils: A Long-Term Study of Content Polluters on Twitter.* URL: https://botometer.iuni.iu.edu/bot-repository/datasets.html.

[14] Jason Brownlee. *A Gentle Introduction to Scikit-Learn: A Python Machine Learning Library*. URL: https://machinelearningmastery.com/a-gentle-introduction-to-scikit-learn-a-python-machine-learning-library/.

[15] Michael Nielsen. *Using neural nets to recognize handwritten digits*. URL: http://neuralnetworksanddeeplearning.com/chap1.html.

Appendix A

Code

A.1 preprocessing.py

```
# -*- coding: utf-8 -*-

import numpy as np
import pandas as pd

np.set_printoptions(suppress=True)


def import_data():

    varol_users = np.loadtxt("Z:\ioptyu2\Desktop\gitDissertation\local\
    Data\\varol_2017.dat")


    col_name = ["UserID","CreatedAt","CollectedAt","Followings","Followers
    ","Tweets","NameLength","BioLength"]
    bot_users = pd.read_csv("Z:/ioptyu2/Desktop/gitDissertation/local/Data
    /social_honeypot_icwsm_2011/content_polluters.txt",
                                    sep="\t",
                                    names = col_name)


    legit_users = pd.read_csv("Z:/ioptyu2/Desktop/gitDissertation/local/
    Data/social_honeypot_icwsm_2011/legitimate_users.txt",
                                     sep="\t",
                                     names = col_name)

    return [varol_users,bot_users,legit_users,col_name]

#col_name = ["UserID","TweetID","Tweet","CreatedAt"]
#bot_tweets = pd.read_csv("Z:/ioptyu2/Desktop/gitDissertation/local/Data/
    social_honeypot_icwsm_2011/content_polluters_tweets.txt",
#                                sep="\t",
#                                names = col_name)
```

A.2 random_forest.py

```
# -*- coding: utf-8 -*-

from sklearn import datasets
from random import seed
from random import randrange
from math import sqrt
import numpy as np
import preprocessing

```

```
10
11 iris = datasets.load_iris()
12
13 X = iris.data
14 Y = iris.target
15 dataset = []
16 #for i in range(len(X)):
17 #    dataset.append(np.hstack((X[i],Y[i])))
18
19 #cleaning up data(picking out useful rows)
20 df = preprocessing.import_data()
21 bots = df[1].values[:2000,3:].astype(int)
22 legit = df[2].values[:2000,3:].astype(int)
23
24
25 #adding label, bots=1 legit=0
26 bots = np.hstack((bots,np.ones((bots.shape[0],1))))
27 legit = np.hstack((legit,np.zeros((legit.shape[0],1))))
28
29 dataset = np.vstack((bots,legit))
30
31 test_bots = df[1].values[2000:3000,3:].astype(int)
32 test_legit = df[2].values[2000:3000,3:].astype(int)
33
34 test_data = np.vstack((test_bots, test_legit))
35 test_label = np.vstack((np.ones((test_bots.shape[0],1)), np.zeros((
       test_legit.shape[0],1))))
36
37 #References:
38 #https://www.python-course.eu/Random_Forests.php
39 #https://machinelearningmastery.com/implement-random-forest-scratch-python
       /
40
41 #split a dataset into k folds
42 def cv(dataset,folds):
43     dataset_split = list()
44     dataset_copy = list(dataset)
45     fold_size = len(dataset)/folds
46     for i in range(folds):
47         fold = list()
48         while len(fold) < fold_size:
49             index = randrange(len(dataset_copy))
50             fold.append(dataset_copy.pop(index))
51         dataset_split.append(fold)
52     return dataset_split
53
54 #calculate accuracy
55 def accuracy_calc(actual, predicted):
56     acc = 0.0
57     for i in range(len(actual)):
58         if actual[i] == predicted[i]:
59             acc += 1.0
60     return acc / len(actual) * 100.0
61
62 #evaluate the algorithm with cv
63 def evaluate(dataset, algorithm, folds, *args):
64     folds = cv(dataset,folds)
65     scores = list()
66     for fold in folds:
67         train_set = list(folds)
68         train_set = sum(train_set, [])
69         test_set = list()
70         for row in fold:
```

```
71              row_copy = list(row)
72              test_set.append(row_copy)
73              row_copy[-1] = None
74          predicted = algorithm(train_set, test_set, *args)
75          actual = [row[-1] for row in fold]
76          accuracy = accuracy_calc(actual, predicted)
77          scores.append(accuracy)
78      return scores
79
80
81  #split the data into two based on a threshold(value)
82  def test_split(index, value, dataset):
83      left, right = list(), list()
84      for row in dataset:
85          if row[index] < value:
86              left.append(row)
87          else:
88              right.append(row)
89      return left,right
90
91  #calculate the gini index
92  def get_gini(groups, classes):
93      instances = float(sum([len(group) for group in groups]))
94      gini= 0.0
95      for group in groups:
96          size = float(len(group))
97          if size == 0:
98              continue
99          score = 0.0
100         for class_val in classes:
101             p = [row[-1] for row in group].count(class_val) / size
102             score += p*p
103         gini += (1.0 - score) * (size / instances)
104     return gini
105
106 #pick the best split
107 def best_split(dataset, n_features):
108     class_values = list(set(row[-1] for row in dataset))
109     b_index, b_value, b_score, b_groups = 999,999,999,None
110     features = list()
111     while len(features) < n_features:
112         index = randrange(len(dataset[0])-1)
113         if index not in features:
114             features.append(index)
115     for index in features:
116         for row in dataset:
117             groups = test_split(index, row[index], dataset)
118             gini = get_gini(groups, class_values)
119             if gini < b_score:
120                 b_index, b_value, b_score, b_groups = index, row[index],
    gini, groups
121     return {'index':b_index, 'value':b_value, 'groups':b_groups}
122
123 #create a terminal node value
124 def terminal(group):
125     outcomes = [row[-1] for row in group]
126     return max(set(outcomes), key=outcomes.count)
127
128 #create child splits for a node or make a terminal
129 def split(node, max_depth, min_size, n_features, depth):
130     left, right = node['groups']
131     del(node['groups'])
132     #check for no splits
```

```
133     if not left or not right:
134         node['left'] = node['right'] = terminal(left + right)
135         return
136     #check if max depth
137     if depth >= max_depth:
138         node['left'], node['right'] = terminal(left), terminal(right)
139         return
140     #go down left child
141     if len(left) <= min_size:
142         node['left'] = terminal(left)
143     else:
144         node['left'] = best_split(left, n_features)
145         split(node['left'], max_depth, min_size, n_features, depth + 1)
146     #go down right child
147     if len(right) <= min_size:
148         node['right'] = terminal(right)
149     else:
150         node['right'] = best_split(right, n_features)
151         split(node['right'], max_depth, min_size, n_features, depth + 1)
152
153
154 #build a tree
155 def build_tree(train, max_depth, min_size, n_features):
156     root = best_split(train, n_features)
157     split(root, max_depth, min_size, n_features, 1)
158     return root
159
160
161 #make a prediction using the tree and nodes
162 def predict(node, row):
163     if row[node['index']] < node['value']:
164         if isinstance(node['left'], dict):
165             return predict(node['left'], row)
166         else:
167             return node['left']
168     else:
169         if isinstance(node['right'], dict):
170             return predict(node['right'], row)
171         else:
172             return node['right']
173
174
175 #create a subsample for the data (the bagging/bootstrapping process)
176 def subsample(dataset, ratio):
177     sample = list()
178     n_sample = round(len(dataset) * ratio)
179     while len(sample) < n_sample:
180         index = randrange(len(dataset))
181         sample.append(dataset[index])
182     return sample
183
184 #make a prediction with a list of bagged trees
185 def bagging_predict(trees, row):
186     predictions = [predict(tree, row) for tree in trees]
187     return max(set(predictions), key=predictions.count)
188
189 #random forest algorithm
190 def random_forest(train, test, max_depth, min_size, sample_size, n_trees,
        n_features):
191     trees = list()
192     for i in range(n_trees):
193         #print(i)
194         sample = subsample(train, sample_size)
```

```
195         tree = build_tree(sample, max_depth, min_size, n_features)
196         trees.append(tree)
197     predictions = [bagging_predict(trees, row) for row in test]
198     testing_predictions = [bagging_predict(trees,row) for row in test_data
        ]
199     print('Backup testing:', accuracy_calc(test_label, testing_predictions
        ))
200     return predictions
201
202 #seed
203 seed(1314)
204
205 dataset = np.random.permutation(dataset)
206
207
208 #print results and stuff
209 n_folds = 2
210 max_depth  = 10
211 min_size = 1
212 sample_size = 1
213 n_features = int(sqrt(len(dataset[0])-1))
214 for n_trees in [2,5,10]:
215     scores = evaluate(dataset, random_forest, n_folds, max_depth, min_size
        , sample_size, n_trees, n_features)
216     print('Trees: %d' % n_trees)
217     print('Scores: %s' % scores)
218     print('Mean accuracy: %.3f%%' %(sum(scores)/float(len(scores))))
```

A.3 sklearnRF.py

```
1 # -*- coding: utf-8 -*-
2 from sklearn.ensemble import RandomForestClassifier
3 from sklearn.tree import export_graphviz
4 import pydot
5 import numpy as np
6 import preprocessing
7 from math import sqrt
8
9
10 #1 = bots, 0 = legit
11 #training data
12 df = preprocessing.import_data()
13 train_bots = df[1].values[:12000,3:].astype(int)
14 train_legit = df[2].values[:12000,3:].astype(int)
15
16 feature_list = df[3][3:]
17
18 X_train = np.vstack((train_bots,train_legit))
19
20 #testing data
21 test_bots = df[1].values[15000:16000,3:].astype(int)
22 test_legit = df[2].values[15000:16000,3:].astype(int)
23
24 X_test = np.vstack((test_bots,test_legit))
25
26 #training labels
27 train_bots_label = np.ones((train_bots.shape[0],1))
28 train_legit_label = np.zeros((train_legit.shape[0],1))
29
30 Y_train = np.vstack((train_bots_label,train_legit_label))
31 Y_train = Y_train.ravel(order='C')
32
```

```
33 #testing labels
34 test_bots_label = np.ones((test_bots.shape[0],1))
35 test_legit_label = np.zeros((test_legit.shape[0],1))
36
37 Y_test = np.vstack((test_bots_label,test_legit_label))
38 Y_test = Y_test.ravel(order='C')
39 a = list()
40 #random forest algorithm using sklearn
41 def random_forest(n_trees, max_depth, n_features):
42     rf = RandomForestClassifier(n_estimators = n_trees, max_depth =
       max_depth, max_features = n_features, bootstrap = True)
43     rf.fit(X_train, Y_train)
44     predictions = rf.predict(X_test)
45     save_tree(rf)
46     return predictions
47
48 #calculate accuracy for predictions
49 def accuracy(predictions):
50 #     for i in range(len(predictions)):
51 #         if predictions[i] < 0.5:
52 #             predictions[i] = 0.0
53 #         else:
54 #             predictions[i] = 1.0
55     accuracy = 0.0
56     for i in range(len(predictions)):
57         if predictions[i] == Y_test[i]:
58             accuracy += 1.0
59     accuracy = accuracy / len(predictions) * 100
60     print("Accuracy: ",accuracy, "%")
61     return
62 #reference for visualisation
63 #https://towardsdatascience.com/random-forest-in-python-24d0893d51c0
64
65 #Visualise a tree
66 def save_tree(rf):
67     tree = rf.estimators_[1]
68     export_graphviz(tree, out_file = 'tree.dot', feature_names =
       feature_list, rounded = True, precision = 1)
69     (graph, ) = pydot.graph_from_dot_file('tree.dot')
70     graph.write_png('tree.png')
71     return
72
73
74 max_depth = 10
75 n_features = int(sqrt(len(X_test[0])))
76
77 for n_trees in [2,5,10,50,100,500,1000]:
78     print("Trees: ",n_trees)
79     predictions = random_forest(n_trees, max_depth, n_features)
80     accuracy(predictions)
```

A.4 ANN.py

```
1 # -*- coding: utf-8 -*-
2
3 import numpy as np
4 import preprocessing
5 from sklearn.preprocessing import StandardScaler
6 from keras import layers, models
7 from keras.utils import to_categorical
8 import matplotlib.pyplot as plt
9 from sklearn import datasets
```

```
10
11
12
13 #iris = datasets.load_iris()
14 #
15 #X = iris.data
16 #Y = iris.target
17 #dataset = []
18 #for i in range(len(X)):
19 #    dataset.append(np.hstack((X[i],Y[i])))
20 #dataset = np.random.permutation(dataset)
21 #x_train = dataset[:100,:4]
22 #y_train = dataset[:100,-1]
23 #
24 #x_test = dataset[100:120,:4]
25 #y_test = dataset[100:120,:4]
26 #
27 #x_val = dataset[120:,:4]
28 #y_val = dataset[120:,-1]
29 #
30 #y_true = dataset[:,-1]
31
32
33 #############PREPROCESSING##############
34 df = preprocessing.import_data()
35 train_bots = df[1].values[:10000,3:].astype(int)
36 train_legit = df[2].values[:10000,3:].astype(int)
37
38 x_train = np.vstack((train_bots,train_legit))
39
40
41 test_bots = df[1].values[12000:16000,3:].astype(int)
42 test_legit = df[2].values[12000:16000,3:].astype(int)
43
44 x_test = np.vstack((test_bots,test_legit))
45
46
47 train_bots_label = np.ones((train_bots.shape[0],1))
48 train_legit_label = np.zeros((train_legit.shape[0],1))
49
50 y_train = np.vstack((train_bots_label,train_legit_label))
51
52
53 test_bots_label = np.ones((test_bots.shape[0],1))
54 test_legit_label = np.zeros((test_legit.shape[0],1))
55
56 y_test = np.vstack((test_bots_label,test_legit_label))
57 y_true = y_test
58
59 val_bots = df[1].values[10000:12000,3:].astype(int)
60 val_legit = df[2].values[10000:12000,3:].astype(int)
61
62 x_val = np.vstack((val_bots, val_legit))
63
64 val_bots_label = np.ones((val_bots.shape[0],1))
65 val_legit_label = np.zeros((val_legit.shape[0],1))
66
67 y_val = np.vstack((val_bots_label,val_legit_label))
68
69
70
71 y_train = to_categorical(y_train)
72 y_test = to_categorical(y_test)
```

```
y_val = to_categorical(y_val)

sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.fit_transform(x_test)
x_val = sc.fit_transform(x_val)

######create model######
model = models.Sequential()
model.add(layers.Dense(64, activation = "relu"))
model.add(layers.Dense(32, activation = 'relu'))
model.add(layers.Dense(2, activation = 'sigmoid'))


model.compile(optimizer = 'adam',
              loss = 'binary_crossentropy',
              metrics = ['accuracy'])

epochs = 100
batch_size = 16

history = model.fit(x_train,
                    y_train,
                    epochs = epochs,
                    batch_size = batch_size,
                    validation_data = (x_val, y_val))


###predictions###
y_pred = model.predict(x_test)

for i in range(len(y_pred)):
    if(y_pred[i][0] < y_pred[i][1]):
        y_pred[i] = 1
    else:
        y_pred[i] = 0
y_pred = y_pred[:,0]

acc = 0.0
for i in range(len(y_pred)):
    if y_pred[i] == y_true[i]:
        acc += 1.0
acc = acc / len(y_pred) * 100
print('Epochs run: ', epochs)
print('Batch size: ', batch_size)
print('Test data size: ', len(y_pred))
print('Accuracy on test data: ', acc, '%')

def comparison_plot(x,
                    y_A, style_A, label_A,
                    y_B, style_B, label_B,
                    title, x_label, y_label):

    plt.clf()
    plt.plot(x, y_A, style_A, label = label_A)
    plt.plot(x, y_B, style_B, label = label_B)
    plt.title(title)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.legend()
    plt.show()

loss = history.history['loss']
```

```
136 val_loss = history.history['val_loss']
137 comparison_plot(range(1, len(loss) + 1),
138                 loss, 'bo', 'Training',
139                 val_loss, 'b', 'Validation',
140                 'Training and validation loss',
141                 'Epochs',
142                 'Loss')
143
144 acc = history.history['acc']
145 val_acc = history.history['val_acc']
146 comparison_plot(range(1, len(loss) + 1),
147                 acc, 'bo', 'Training',
148                 val_acc, 'b', 'Validation',
149                 'Training and validation accuracy',
150                 'Epochs',
151                 'Accuracy')
152
153 print("Highest validation accuracy:", val_acc[np.argmax(val_acc)], "Epochs
    :", np.argmax(val_acc))
154 print("Lowest validation loss:", val_loss[np.argmin(val_loss)], "Epochs:",
     np.argmin(val_loss))
```

Appendix B

Weekly logs

B.1 Week1

Weekly Log (25/01/19)

This was the last week of exams, so I didn't get much done. Did some more research on what sort of algorithms I should be using and outlining how it is going to work exactly. In the seminar we went over the basics of LaTeX, which is what I will be using to write my reports as it allows a much more structured and customisable approach at writing. I have gone over the template provided to us and have tried to make myself familiar with it. It will take a little getting used to, but it should make everything much more organised. I looked into a bit more about machine learning to see exactly how it will be possible to create this program. I had a meeting with my supervisor, where we setup a weekly meeting for Tuesday every week. Since there wasn't much that I have done this past week, there wasn't anything to talk about. I will be making sure I have the template filled out with all my personal information and make sure it is altered for myself. I also need to make sure to setup a GitHub project as the version control provided will help out a lot.

B.2 Week2

Weekly Log (01/02/19)

This week I was looking to get started on my preliminary report which will contain most of the introductory sections of the report along with plans of how the project will come together. The git repository is now setup and will now be used to store all relevant files on there for easy version control just in case something goes wrong. I have also found this interesting package in R for detecting Twitter bots much like what I plan to create. It is quite interesting to see something like this available online and could be useful in validating results of my own. During the meeting with Lahcen, we talked about the preliminary report and what it should contain, as well as completing the compliance statement which was required before getting access to the report page. In the seminar we looked at a processing framework called Hadoop. It seems very useful for processing big data and interacting with it in multiple ways. It's an option that is there in case I need it.

B.3 Week3

Weekly Log (10/02/19)

This week I spent all my time working on the preliminary report. I worked mainly on the introduction, background research and system design. In the introduction I wrote about the basic idea of the project along with the aims or it. In background research it was about the things I read about before beginning the project and what sort of existing systems there are. As for system design, I wrote about the way the system will work. The planning part is still left which I will write this week as well as finish off some sections from the previous ones. Lahcen said that the work so far on the report is good and I should finish it soon, probably by next week, in order to start looking for data for the actual program.

B.4 Week4

Weekly Log (17/02/19)

This week was spent adding more stuff to the preliminary report. The sections that needed more in them were planning, system design and background research. For planning I needed to create a gantt chart which helps keep track of where I should be. Also, after talking with Lahcen, he helped me sort out my referencing which was something that I was lacking. Reading week is coming up along with the deadline for the report. Things to do include:
Finish off background research and system design in report
Create gantt chart and planning in report
Find databases to use for system

B.5 Week5

Weekly Log (03/03/19)

I have found some data for the training part of the model. This will all need to be pre-processed as they are mostly in csv format with some unnecessary parts. This was the main task for this week. The other thing Lahcen wanted me to do was to do some more reading about the algorithm (random forest) that I plan to use and to write about it in detail. I have also attempted to start with the actual algorithm, although it is quite difficult to begin. Next week I will probably need to get a good start on the implementation of random forest.

B.6 Week6

Weekly Log (10/03/19)

This week I didn't have much time available to work on the project. Lahcen suggested that I take a break from working on the report and do more things for the program. I have started working on the implementation of the random forest algorithm. I have found several resources that helped get started along with some pseudocode I found online. About half of the program is done and I think by the end of next week it should be finished to a baseline where I can go back to working on the report for things such as testing.

B.7 Week7

Weekly Log (17/03/19)

This week I did a mixture of report and programming work. For the report I added parts about the random forest algorithm and used more diagrams to describe the processes such as trees and pseudocode just like Lahcen suggested. The algorithm implementation is mostly complete. I'm just waiting for Twitter to still reply to me and send me my API key so I can get started on retrieving data. Most of that should be sorted by next week and then I can move on to adding some more finalised sections to the final document and getting some stuff ready to submit for marking over the Easter holiday.

B.8 Week8

Weekly Log (24/03/19)

This week I carried on with the pre-processing that I ended up leaving half finished to work on the report a few weeks ago. The annoying part was to deal with all the hashes in the data of the tweets as this caused issues with python in which hashes are used for comments. I needed to find a way to escape all the characters when importing the dataset. I also ended up spending some time creating diagrams. Mainly a UML class diagram and a simple overview of the system to provide aid in describing the system. Lahcen mentioned that I needed to swap some sections around. The part about random forest that I wrote needed to be moved to the background research section. I need to start thinking about wrapping up the programming part and get started on properly writing towards the final report.

Appendix C

PPR

FINAL YEAR PROJECT

Preliminary Report

Author:
Marcell BATTA

Supervisor:
Dr. Lahcen OUARBYA

A thesis submitted in fulfilment of the requirements for BSc Computer Science Degree

February 22, 2019

Contents

Chapter 1

Introduction

Social media has become an integral part of our lives in the past years as we spend more time online than ever before. Roughly 30% of our online time is spent on social media interactions, Twitter being one of them [1]. Twitter is arguably the largest source of news on the internet. This is due to the nature of information spreading on the platform through tweets and retweets. At this point, because of the scale, it is impossible to monitor it all to make sure everything is accurate and that there is no false information being spread.

1.1 Aim

The aim of this project is to create a program with an underlying algorithm that will attempt to figure out whether a Twitter account is under the control of a human or is purely being controlled by a script that someone wrote. The issue doesn't come from there being accounts not directly used by people or 'bots'. There are plenty of examples of public bot accounts for things such as weather or news. Instead the issues come with the ones that claim to be real individuals when they in fact are not. With the use of a program such as this, it can be possible to identify these bots by comparing their parameters like tweet content and see if they match patterns of other real people or not.
Ideally the program shouldn't misjudge too often and should be a reliable way to identify false accounts from real ones. This would be done through a supervised machine learning algorithm which would be fed with as much data of previously labelled accounts as possible.

Chapter 2

Background Research

This chapter contains the information found before beginning development of the program, along with some systems that are already available and a summary on them.

2.1 Politics

Politics is probably the biggest concern when it comes to these bot accounts. They are the reason why false information spreads so fast. This is because of the way Twitter works with its trending hashtags. These bots will tweet and retweet about important and most likely incorrect matters. They also make use of popular hashtags that basically define the topic of a tweet. This then leads to these malicious tags to become trending for everyone to see.

2.1.1 2016 US Elections

The 2016 elections in America was one of the, if not the biggest outburst of Twitter bots we have yet to see. It was found that by extrapolating some findings, roughly 19% of 20 million election related tweets originated from bots between September and October of 2016 [2]. According to the same study it was also found that around 15% of all accounts that were involved in election related tweets were bots. Now even though that is a lot of attention for these tweets containing false information, they will mostly only be seen by people who are already on the same side and agree. However, this doesn't rule out the affects. A study by the NBER(National Bureau of Economic Research) came to the conclusion that these bots were the cause of up to 3.23% of the votes that went towards Donald Trump [3]. This tells us that even if it's just marginal, it does still affect the outcomes.

The interesting part of all this is that the bots immediately went silent and disappeared as the election ended. The accounts though didn't get deleted but they simply went into hibernation waiting for their next bit of propaganda that needed to be spread. In 2017, 2000 of these bots reemerged to take part in the French and German elections as well, meaning they were run by the same people. They were discovered to make up for 1 in 5 election related tweets [4].

2.1.2 2018 US Mid-Term Election

Following the 2016 elections, the 2018 Mid-Terms were another prime target for Twitter bots. Before the voting took place, there were automated accounts trying to discourage people from voting. Of these, 10,000 were banned by Twitter. Even legislations were signed in an attempt to control the situation [5]. Interestingly, nearly

two weeks after the election day, there was still activity amongst these bots which accounted for a fifth of the #ivoted tweets [6]. The numbers recorded during this recent election compared to the one in 2016 was believed to be much lower. This could either be due to the reduced number of accounts being used or it could even be that the bots are now much more sophisticated and can remain undetected as they might be able to recreate human interactions and behaviour at a higher standard.

2.2 Twitter junk

Political issues aren't the only thing being caused by these bots. A study done at the University of Iowa has shown that through the third-party applications that Twitter allows its users to utilise can be, and is often abused in malicious ways such as phishing or even just spam. Twitter themselves have a way of dealing with these toxic accounts and do most of the time eventually ban them, however this study has found that 40% of the accounts that their algorithm detected as in some way benign were located about a month before Twitter took any action towards them [7]. As this does show that there are still improvements to be made even at Twitters end in terms of detection speed, we mustn't forget that there are many other parameters we must watch out for, some unknown outside of Twitter. A Twitter spokesperson wrote:

> "Research based solely on publicly available information about accounts and tweets on Twitter often cannot paint an accurate or complete picture of the steps we take to enforce our developer policies" [8]

2.3 Existing Systems

There are a handful of algorithms or programs that have been designed to detect these bots. Most of them tend to use a machine learning algorithm as a way to classify accounts and tell them apart from each other, while others have attempted to use deep neural network architectures such as long short-term memory(LSTM).

2.3.1 Tweetbotornot

Tweetbotornot is a package built in R that uses machine learning to classify Twitter accounts. It has two 'levels'. One for users where it uses information related to an account such as location or number of followers. The other is a tweet-level which checks for details like hashtags, mentions or capital letters out of the user's more recent 100 tweets. This could prove useful when testing my program to compare results as the accuracy of this library is 93.8 percent. As this is just a package created, it doesn't have any user-interface program built around it or anything like that, therefore it is unusable by anyone not knowledgable in R.

2.3.2 DeBot

DeBot is a fully functioning python API for bot detection on Twitter. It has the ability to obtain a list of bots already detected by DeBot, or just simply checking individual accounts. You can also get a list of bots which appear in the archives more than a given number of times. They can even be requested based on topics. It does however have a somewhat working built in search mechanism on its website.

Chapter 3

System Design

This chapter is about the methodology behind the program and some of the features and requirements.

3.1 Method

The core part of the system will be deciding whether the account it is checking is a bot or not. It will use a form of machine learning to classify the account as either 'bot' or 'not bot'. Initially it would seem as if this was a binary classification issue, however it makes more sense to treat it as a regression problem. This makes it much easier to interpret the results for the user as a probability is easily understood and is a much more honest answer compared to just giving the user a 'yes' or 'no' since we can never be too sure either way.

3.2 System Requirements

In order to achieve the aims of the system, there are a few things the program will need to do.

1. Allow users to input a Twitter account.
2. A connection using the Twitter API must be established in order to retrieve users' data.
3. The system must be able to determine whether an account is a bot or not.
4. Display the likelihood that an account is a bot or not.

3.3 Algorithm

The algorithm for my system will consist of a supervised machine learning algorithm called random forest. This works by creating a multitude of decision trees and outputting the mean prediction of these trees. However, depending on how things go during development, it might make more sense to use a deep learning neural network instead for the regression.

3.3.1 Data

Regardless of which algorithm I end up finalising with, I will need data for training. This is important since everything will be based on it. This data needs to also be correctly labelled and will most likely need to be pre-processed somewhat before being fed to my algorithm. It will then be used to train the system.

3.3.2 Twitter API

As the user will be able to enter a Twitter account name themselves, the program will need to have Twitter REST API functionality. This is to access the relevant accounts information such as tweets and account details. There is a limited number of requests allowed within a 15 minute window therefore I need to make sure to keep the API requests to a bare minimum.

Chapter 4

Planning

In this section I will talk about some of the things I have done to make sure everything goes as planned and my time is used efficiently.

4.1 Gantt chart

Gantt charts are a nice way to keep track of your time available over a whole project. It really helps understand the scope of things and to give a better estimate on how to split up your time into smaller chunks.

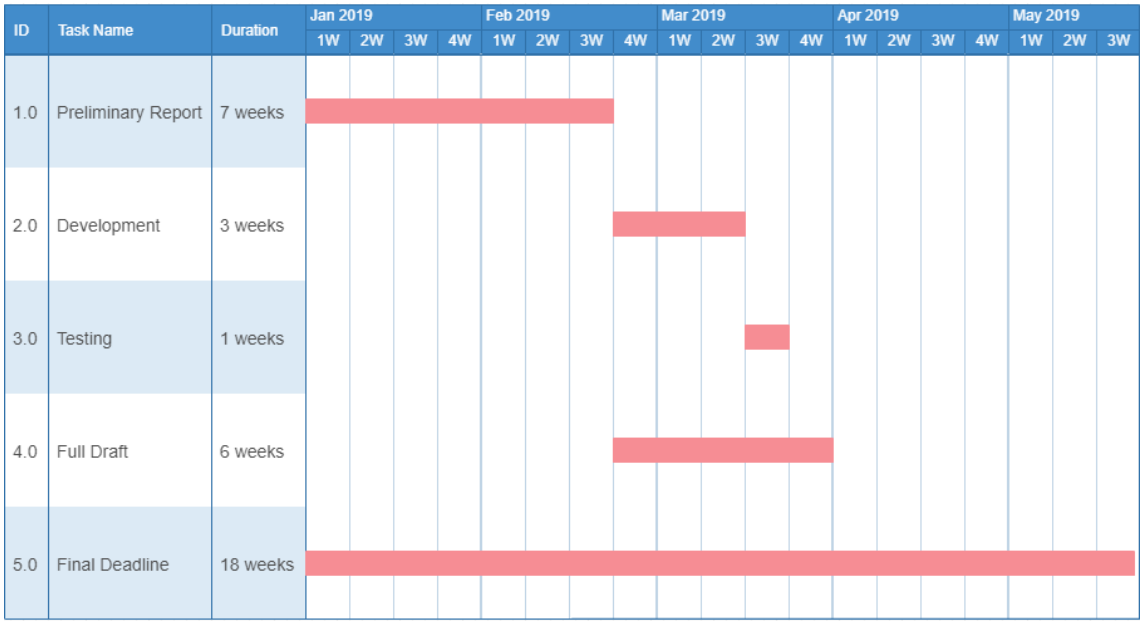

FIGURE 4.1: Gantt chart

Here you can see the gantt chart created for this project. It has a rough sketch of the bigger tasks, nothing too precise, as it is helpful enough this way to keep track of the weeks overall.

4.2 Progress Logs

The weekly progress logs act as a weekly sprint. In them I write down what I did, what I wanted to do and what I will do by next week. This helps keep track of the more short term week-by-week goals. I also find the recaps at the end of the week to be quite helpful to remind myself of what I did and what I still need to do.

4.3 Version Control

Version control is an essential part of any project. I'm using GitHub, but there are many others out there that do the same job. It helps mainly because of the ability to go back to previous versions whenever needed, for example when something goes horribly wrong.

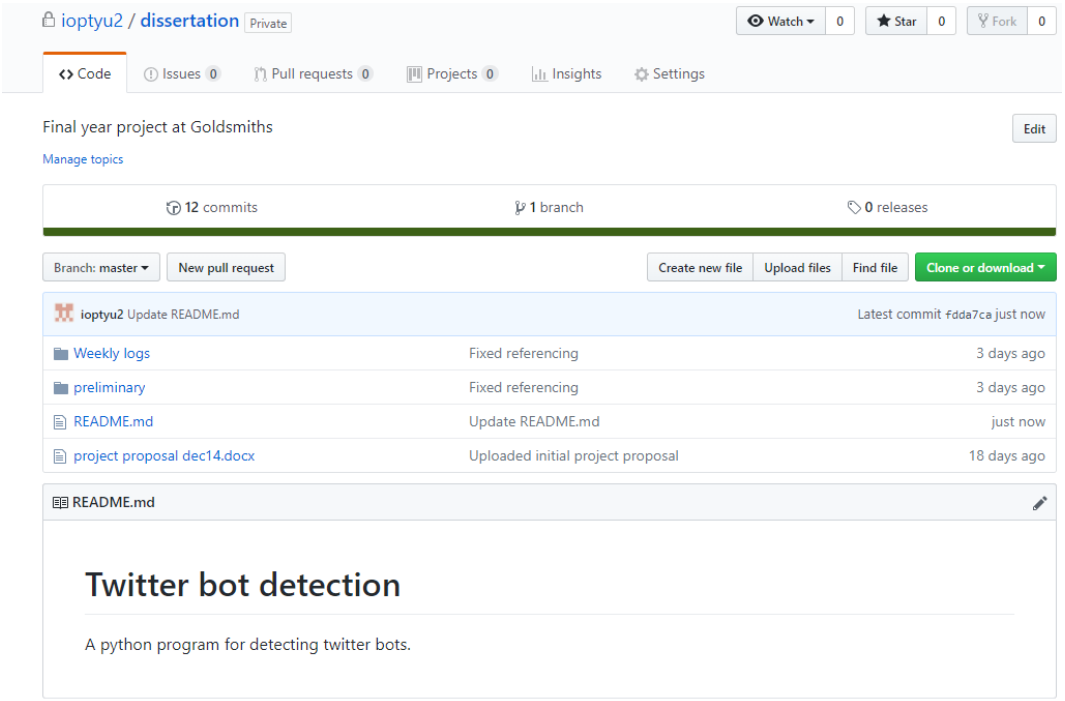

FIGURE 4.2: Github Repository

Bibliography

[1] Katie Young. *Social Media Captures Over 30 percent of Online Time*. URL: https://blog.globalwebindex.com/chart-of-the-day/social-media-captures-30-of-online-time/.

[2] Alessandro Bessi and Emilio Ferrara. "Social bots distort the 2016 U.S. Presidential election online discussion". In: *First Monday* 21.11 (2016). ISSN: 13960466. DOI: 10.5210/fm.v21i11.7090. URL: https://firstmonday.org/ojs/index.php/fm/article/view/7090.

[3] Yuriy Gorodnichenko, Tho Pham, and Oleksandr Talavera. *Social Media, Sentiment and Public Opinions: Evidence from #Brexit and #USElection*. Working Paper 24631. National Bureau of Economic Research, 2018. DOI: 10.3386/w24631. URL: http://www.nber.org/papers/w24631.

[4] Denise Clifton. *Twitter Bots Distorted the 2016 Election—Including Many Likely From Russia*. URL: https://www.motherjones.com/politics/2017/10/twitter-bots-distorted-the-2016-election-including-many-controlled-by-russia/.

[5] BBC. *US mid-terms: Twitter deletes anti-voting bots*. URL: https://www.bbc.co.uk/news/technology-46080157.

[6] The Economic Times. *Thousands of Twitter bots active during 2018 US mid-term elections*. URL: https://economictimes.indiatimes.com/news/international/world-news/thousands-of-twitter-bots-active-during-2018-us-mid-term-elections/articleshow/67850597.cms.

[7] Shehroze Farooqi and Zubair Shafiq. *Measurement and Early Detection of Third-Party Application Abuse on Twitter*. URL: http://homepage.divms.uiowa.edu/~sfarooqi/Files/Farooqi-AbusiveTwitterApplications.pdf.

[8] Andy Greenberg. *TWITTER STILL CAN'T KEEP UP WITH ITS FLOOD OF JUNK ACCOUNTS, STUDY FINDS*. URL: https://www.wired.com/story/twitter-abusive-apps-machine-learning/?mbid=social_twitter&utm_brand=wired&utm_campaign=wired&utm_medium=social&utm_social-type=owned&utm_source=twitter.