Detecting Twitter Bots

Anantha Natarajan Selvaganapathy
MSCS NYU Tandon School of Engineering
Brooklyn, USA
http://ananth.co.in/
ans599@nyu.edu

Abstract— **The objective of this project is to use machine learning techniques to detect weather a given Twitter account is a bot or not. We will be using various machine learning algorithms and compare and analyze their predictions. We will also explore the use of deep learning techniques and compare their results with regression and classification algorithms.**

Keywords—Machine Learning

I. Introduction

Twitter is an online social network where users can upload text, images and video clips as "tweets" – which are restricted to 140 characters. Users can follow other users they are interested in, and can send direct messages to users. Over the years, Twitter has quickly become one of the most popular social media platforms. As of late 2016, Twitter has about 300 million active monthly users. Twitter is used by a lot of celebrities, world leaders and organizations to share news and updates to the world.

This this project, we aim to come up with efficient ways to use machine learning algorithms to detect Twitter bots. We also discuss the various methods use to collect and filter data, algorithms and assumptions, compare and try to analyze the performance of various models and the results of the experiments.

II. Motivation

It's estimated that about 16% of the US adults are active on Twitter, and about 8% of them use Twitter as a primary source of news [2]. About 50% of the users have shared news articles or participated in conversations. With the rising influence of this platform, it's important that identify bots that can affect the reach and influence of articles on Twitter.

Bots are automated accounts that can post, share, retweet, follow and even reply to tweets on Twitter. Bots can be used to useful and productive tasks: to automatically tweet information like weather, stock prices and trending news articles. Bots can also be used to automate repetitive tasks like sharing regular updates of sports scores. On the other hand, bots can be used to disrupt the importance and reach of certain articles. Bots can be used to aggressively retweet and share, favorite or reply to certain articles thus creating a 'fake' trend, and thereby increasing the reach of that post. These articles could be spam, advertisements, political propaganda, or as we noticed in the recent elections, even fake news. To avoid such content pollution, and to prevent and contain spam, it is important that we understand that the accounts that are contributing to articles are humans or bots.

This project aims to do just that. By using data provide by Twitter API, we try to analyze the data, train models and eventually classify accounts as bots, or humans.

III. Related Work

Researchers have proposed several methods, for example, Center for Complex Networks and Systems Research, Indiana University, Bloomington, IN (USA), BotOrNot: A System to Evaluate Social Bots [3] have come up with a classification system to identify if a given Twitter handle is a bot or not by estimating a confidence based on various parameters like the number of followers, tweets, friends, the content of the tweets.

Others have taken a different approach - Using Sentiment to Detect Bots on Twitter: Are Humans More Opinionated than Bots? John P. Dickerson Carnegie Mellon University, Vadim Kagan Sentimetrix, Inc. and V.S. Subrahmanian University of Maryland College Park have used collection of network-, linguistic-, and application oriented variables that could be used as possible features, and identify specific features that distinguish well between humans and bots.

IV. Data

The data we use to train the models is obtained by using Twitter API. Since we will be looking at supervised approaches, we need a labeled dataset to train our algorithms on. The data we use for each account are: id, Id_str, Screen_name, Location, Description, Url, Followers_count Friends_count, Listed_count, Created_at, Favourites_count Verified, Statuses_count, Lang, Status, Default_profile, Default_profile_image, Has_extended_profile ,name, Bot. I used a couple of approaches to collect and label our data:

A. BotOrNot API

In the first approach, I collected user handles with the follower's section of popular pages that have known to have a lot of bot followers: for example, CNN, Barack Obama and Donald Trump. Then I scraped user handles from the followers' sections of these accounts using javascript and some DOM manipulation. I used the API provided by BotOrNot [3] to get the confidence scores of the scraped handles. The API returns us a score ranging from 0 to 1, where the handle given is a bot when the probability is 1. I used a threshold of >0.6 to label the corresponding account as bot and as human otherwise. This higher threshold is to increase the quality of the resultant dataset.

Then, we use the Twitter API to extract the required data points for that handle.

B. *Honeypot*

In the next approach, I created a honeypot account – a fake account for the purpose of attracting bot followers. I then subscribed to an online service that grants followers to registers accounts. After registering to the service [4], I noticed about 50 instant followers to the bot account. These accounts are most certainly bots, part of a botnet [5]. These handles, and their account information was then scraped using the Twitter API.

Using these methods, I collected data for 269 accounts, out of which 75 are labeled as bots.

V. Algorithms Used

I plan on using supervised learning algorithms to solve this problem: Random Forrest and SVM.

Apart from these algorithm, I plan to train a neural network using Theano or TesorFlow.

In the following sections, I will discuss the implementation details, and the comparison of the results obtained by these models.

VI. RESULTS

The ultimate aim of the project is to identify a given account as a bot, not a non bot account. This is clearly a classification problem. To be precise, this is a binary classification problem. This presents us with a lot of opportunities in choosing the approach in model selection.

Random Forrest Classifier, Support Vector Machines, Naïve Bayes, Convoluted Neural Networks are some of the popular methods used in classification problem.

We will be using python as the primary language to tabulate the results and observations. We will be using the popular python ML packages like: numpy, matplotlib, pandas and scikitlearn.

In the first part of our experiments, we will use ensemble methods like the Random Forrest Classifier to classify our datasets.

Let's first take a look at the dataset in table 1.1:

Bot data shape	(1056, 20)
Non bot data shape	(1176, 20)

Table 1.1: Dataset shape

The dataset contains roughly an equal split in the classes. This is important, as the split is a major factor in the entropy of the dataset and eventually the information gain of the attributes.

Now let's take a look at the attributes and their data types in table 1.2. As we notice, a lot of attributes are of object type. This is usually a string. In all machine learning models, we have to represent the dataset as an array of floats. We will come back to this problem in the next part.

Attribute	Data Type
id	float64
id_str	object
screen_name	object
location	object
description	object
url	object
followers_count	int64
friends_count	int64
listedcount	int64
created_at	object
favourites_count	int64
verified	bool
statuses_count	int64
lang	object
status	object
default_profile	bool
default_profile_image	bool
has_extended_profile	object
name	object
bot	int64

Table 1.2 Table describing the data type of the attributes in the dataset. Note that in python, the object data type will be treated as a string.

The field manes in the table above are quite self descriptive. The screen name refers to the Twitter handle without the '@' symbol. The url refers to the profile url. The status field contains a stringified json object: a json containing information about the user's tweets such as the tweet text, number of favorites and retweets, date and time posted and so on. The final field bot contains an binary int: 0 if non bot account, 1 for bot accounts.

The dataset is also fairly clean. We will still have to clean the dataset to convert missing fields and handle non float inputs. By loading it into a pandas data frame, we are able to analyze the dataset's contents such as the number of fields, the min, max, average values.

A. Feature Extraction

Now that we have successfully loaded the dataset and cleaned it, we can decide to extract the features we are going to use to train our model.

Looking at the given fields, we can intuitively choose some of the fields and justify our selection though statistical visualization.

Take a look at the following set of features:

['screen_name','followers_count', 'friends_count', 'listedcount', 'favourites_count', 'verified', 'statuses_count', 'status', 'default_profile', 'default_profile_image']

Intuitively, we can understand that the number of friends, favorites, followers, profile image, status count and verification will all be contributing factors. Let us use these features initially, observe the results, and then fine tune our feature selection.

To give a sense of the contribution of the features, let the see how the data varies for 'favorites_count' for bot and non bot accounts from figure 1.1.

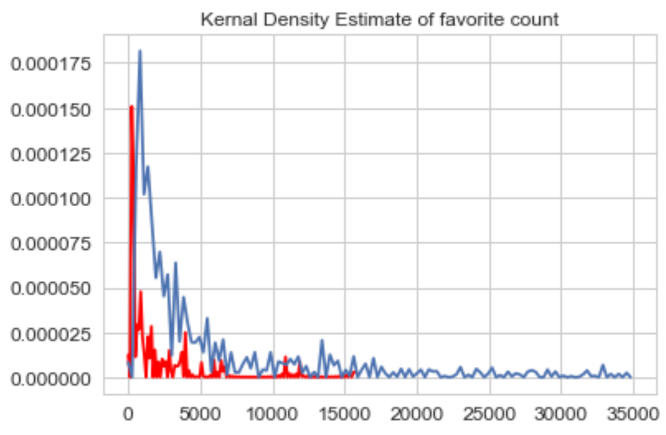


Figure 1.1: A KDE estimate of favorite count

We use a Kernal Density Estimate to visualize the distribution of 'favorites_count' in the dataset. The red line corresponds to bot accounts and the blue line to non bot accounts. As we see, non bot accounts tend to have higher number of favorites, and a lot of accounts have more than 15,000 favorites – when almost no bot account has those many favorites. This is a very good sign of a good feature. This can further be verified by looking at the average number of favorites by each category in figure 1.2.

From the table 1.2 we see that the chosen attributes for our first model are already in float type or bool type. We therefore do not need to transform the fields.

One might also suggest that the 'status' field would be a useful feature in our problem. The 'status' field in face contains some interesting data like the date, time and text of tweets. Unfortunately, since the given data is highly inconsistent due to the API response's encoding differing from each student's contributed dataset, we will not be able to parse the string into a json directly. Instead, we will look at different ways to utilize the date in the string, and discuss its effectiveness.

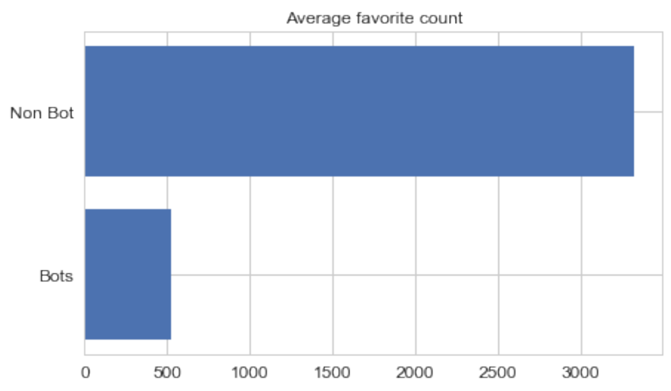


Figure 1.2: Average favorite counts

A look at the kernel density estimate of the number of followers as shown in Fig 1.3 clearly shows that bots generally have a fewer number of followers-usually other bots- and that a high number of followers is a strong signal that the account is non bot.

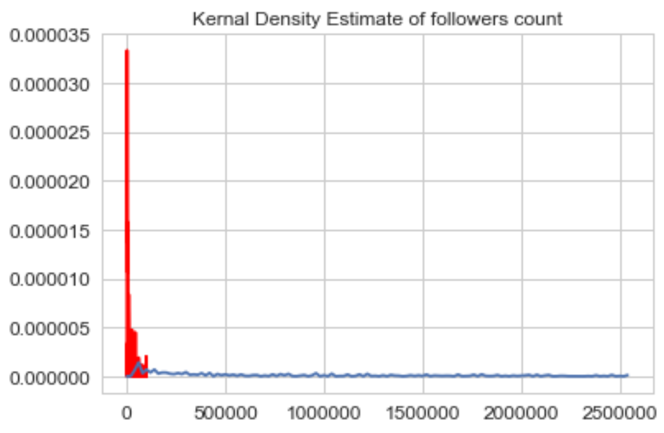


Figure 1.3: KDE of follower count. The blue line indicates non bot accounts. As we notice, the bots generally have far fewer followers.

A similar interesting but obvious observation can be made by looking at Fig 1.4 and Fig 1.5, which display the accounts that contain the word `bot` in their name. The bot dataset has a lot of accounts contain the word `bot` in their name. A few non bot account seem to contain the word too, but it could be names like Abbot, Longbottom, Botha, etc.

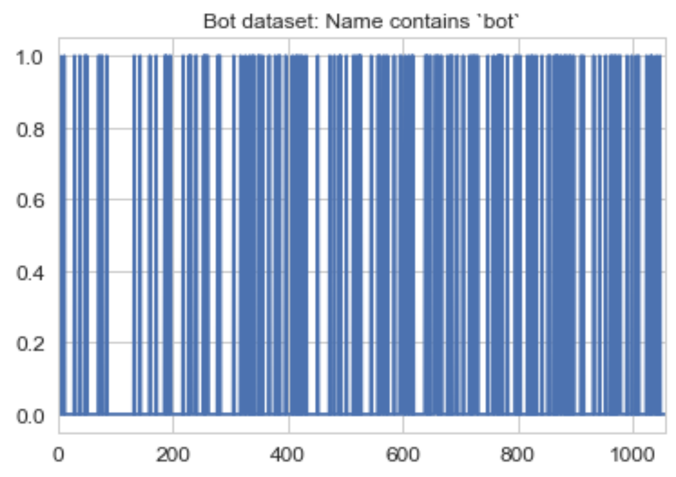


Figure 1.2: The blue lines indicate that the account name contains the word `bot`. The bot dataset shows that most of the accounts contain the word `bot` in them.

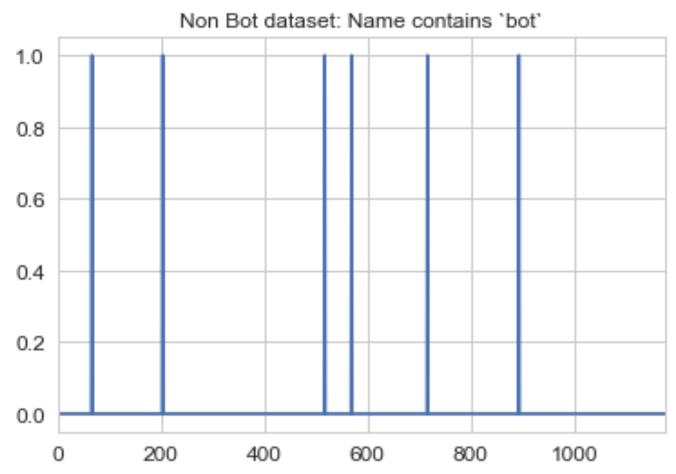


Figure 1.2: The blue lines indicate that the account name contains the word `bot`. The non not dataset shows that most of the accounts do not contain the word `bot` in them, except a few outliers

B. *Random Forrest Regressor Classification*

Random Forest is one of the most popular and most powerful machine learning algorithms. It is a type of ensemble machine learning algorithm called Bootstrap Aggregation or bagging. Random Forests grows many classification trees. To classify a new object from an input vector, put the input vector down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification having the most votes (over all the trees in the forest).

We first chose to try out Random Forrest for the following reasons [6]:

- It is unexcelled in accuracy among current algorithms.
- It runs efficiently on large data bases.
- It can handle thousands of input variables without variable deletion.
- It gives estimates of what variables are important in the classification.
- It generates an internal unbiased estimate of the generalization error as the forest building progresses.

It also has methods for balancing error in class population unbalanced data sets. Generated forests can be saved for future use on other data. Prototypes are computed that give information about the relation between the variables and the classification. It computes proximities between pairs of cases that can be used in clustering, locating outliers, or (by scaling) give interesting views of the data. The capabilities of the above can be extended to unlabeled data, leading to unsupervised clustering, data views and outlier detection. It offers an experimental method for detecting variable interactions.

The random forests algorithm (for both classifications and regression) is as follows [7]:

1. Draw ntree bootstrap samples from the original data.

2. For each of the bootstrap samples, grow an unpruned classification or regression tree, with the following modification: at each node, rather than choosing the best split among all predictors, randomly sample entry of the predictors and choose the best split from among those variables. (Bagging can be thought of as the special case of random forests obtained when mtry = p, the number of predictors.)

3. Predict new data by aggregating the predictions of the n-tree trees (i.e., majority votes for classification, average for regression).

An estimate of the error rate can be obtained, based on the training data, by the following:

1. At each bootstrap iteration, predict the data not in the bootstrap sample (what Breiman calls “out-of-bag”, or OOB, data) using the tree grown with the bootstrap sample.

2. Aggregate the OOB predictions. (On the average, each data point would be out-of-bag around 36% of the times, so aggregate these predictions.) Calculate the error rate, and call it the OOB estimate of error rate.

Let’s use the features we engineered in IV.A in a Random Forrest Classifier.

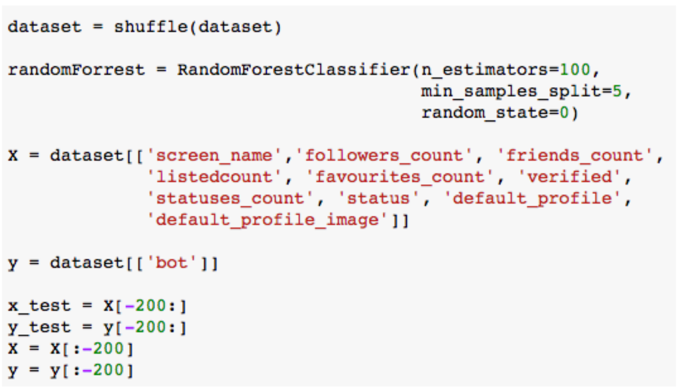
['screen_name','followers_count', 'friends_count', 'listedcount', 'favourites_count', 'verified', 'statuses_count', 'status', 'default_profile', 'default_profile_image']

We load the dataset into a pandas data frame, clean the data and normalize it.

The code snippet in Fig 1.3 explains the initialization of the classifier. After merging the bot and non bot data sets, we shuffle the data set using the shuffle method provided by sklearn. Shuffling the data set is necessary in order to get an unbiased split in the training and test dataset.

Then, we initialize the classifier. We create the classifier by calling the constructor with the required parameters. The `n_estimators` is the number of trees in the forest. Based on trail and error, we estimate that the optimal number of trees after with the accuracy stagnates is 100. The `min_sample_split` is the minimum number of samples required to split an internal node.

Then we assign the values of `X` and `y`- the input feature vector and the corresponding output vector. These are the inputs to our model on which we will train on. We select the required fields from the dataset for `X` and the output value bot for `y`.



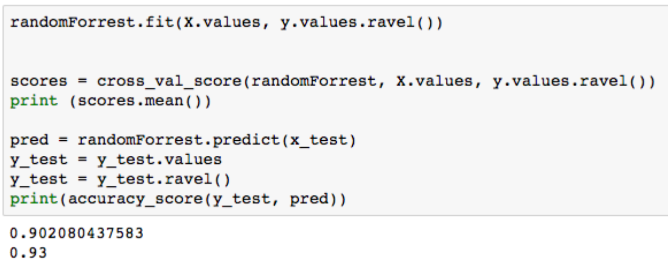
```
dataset = shuffle(dataset)

randomForrest = RandomForestClassifier(n_estimators=100,
                                       min_samples_split=5,
                                       random_state=0)

X = dataset[['screen_name','followers_count', 'friends_count',
             'listedcount', 'favourites_count', 'verified',
             'statuses_count', 'status', 'default_profile',
             'default_profile_image']]

y = dataset[['bot']]

x_test = X[-200:]
y_test = y[-200:]
X = X[:-200]
y = y[:-200]
```

Figure 1.3: Code snippet that initialized a randomForrest classifier and splits training and test sets.

Next, we have to separate the test data from the training data. We do this so that when estimating the accuracy of our model, we do so on a data set that our model had not seen before, i.e., data that it has not been trained on.

As the total size of the dataset is about 2000 rows, we set apart 10% of it, roughly 200 rows for training. Then, we remove those fields from the training data. This is why we needed to shuffle the dataset – so that the train-test split has a reasonable amount of both the classes.

Now that we are done with splitting our test and train data, we can fit the classifier and start predicting. Fig. 1.4 shows the fitting and the validation of our model.



```
randomForrest.fit(X.values, y.values.ravel())


scores = cross_val_score(randomForrest, X.values, y.values.ravel())
print (scores.mean())

pred = randomForrest.predict(x_test)
y_test = y_test.values
y_test = y_test.ravel()
print(accuracy_score(y_test, pred))

0.902080437583
0.93
```

Figure 1.4: The RF Classifier in action. The first run gave an accuracy of 93% on unseen test data.

To estimate the performance of our model, we use `accuracy_score` and the `cross_val_score` methods provided by scikitlearn. Based on the initial set of features, we get a cross validation score of 0.9 and accuracy of 93% on the test data.

C. Support Vector Machine Classifier

Support Vector Machines are perhaps one of the most popular and talked about machine learning algorithms. In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis [8].

They were extremely popular around the time they were developed in the 1990s and continue to be the go-to method for a high-performing algorithm with little tuning. The SVM algorithm is implemented in practice using a kernel.

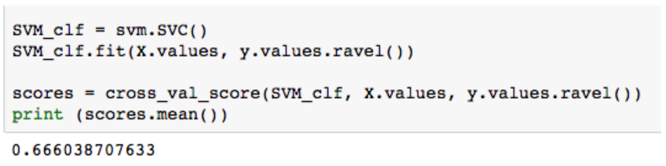
The learning of the hyperplane in linear SVM is done by transforming the problem using some linear algebra, which is out of the scope of this introduction to SVM. For our purposes, we will be using a radial kernel:

K(x,xi) = exp(-gamma * sum((x – xi^2))

Where gamma is a parameter that must be specified to the learning algorithm. A good default value for gamma is 0.1, where gamma is often $0 < \text{gamma} < 1$. The radial kernel is very local and can create complex regions within the feature space, like closed polygons in two-dimensional space.

Just like in Random Forrest, we use a similar feature set:

['screen_name', 'followers_count', 'friends_count', 'favourites_count', 'verified', 'default_profile_image']



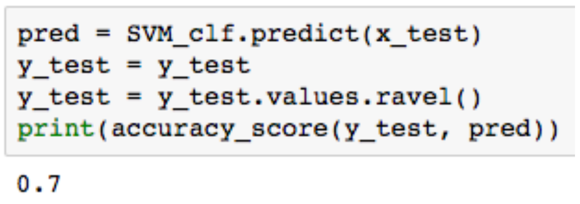
```
SVM_clf = svm.SVC()
SVM_clf.fit(X.values, y.values.ravel())

scores = cross_val_score(SVM_clf, X.values, y.values.ravel())
print (scores.mean())
```

```
0.666038707633
```

Figure: 1.5 SVM Classifier in action

After shuffling the dataset again and splitting the test and train data, we initialize the classifier. We use scikitlearn's svm.SVC method to create the classifier. After fitting the classifier with the input dataset, we measure the accuracy and cross validation score as shown in Fig. 1.5 and Fig.1.6.

The SVM classifier produces a cross validation score of 0.66 and 70% accuracy on unseen test data.



```
pred = SVM_clf.predict(x_test)
y_test = y_test
y_test = y_test.values.ravel()
print(accuracy_score(y_test, pred))
```

```
0.7
```

Figure: 1.5 SVM Classifier using the above features produces an accuracy of 70%.

D. *Improving prediction accuracy*

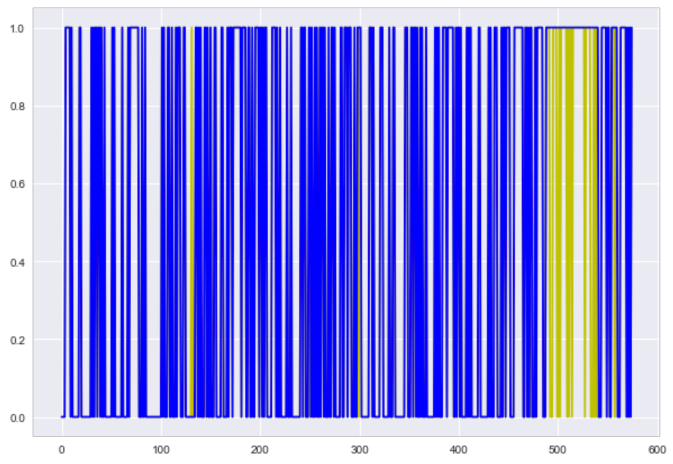
Now that we have 3 different models, we can try to improve the accuracy of our predictions.

Tuning the parameters of RF trees helps in boosting the accuracy of the model but not by a significant amount. Tuning the parameters gives us a consistent cross validation accuracy of 91%.

E. *Expert's Advice: Randomized Weighted Majority Algorithm*

The randomized weighted majority algorithm is an algorithm in machine learning theory. It improves the mistake bound of the weighted majority algorithm. [9]

When we visualize our predictions on the final test dataset, we can notice some variations in the predictions of various models. The figures below represent predictions by 4 different models:



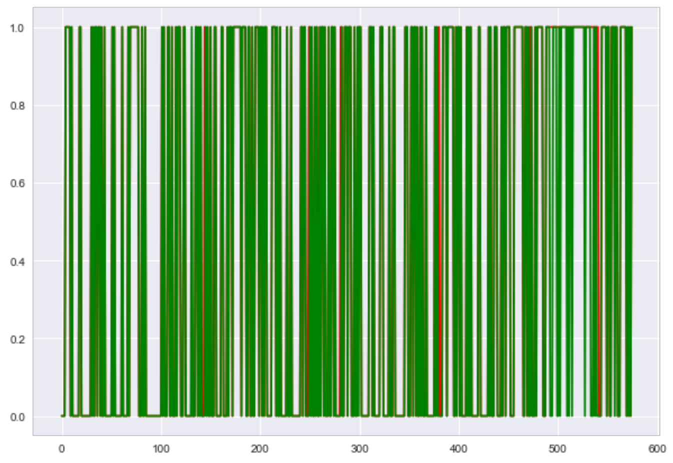


Figure: 1.6

In this approach, we combine the predictions of different models, and use the Expert's Advice algorithm to come to a final prediction.

We also take into account the confusion matrix of each of the models we created.

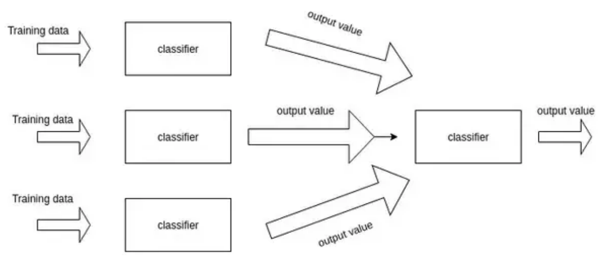


Figure: 1.6 The architecture of Randomized Weighted Majority Algorithm

By using the predictions from every single model, we can counter balance the bias each individual model might have because of particular features. This approach was loosely modeled on the meta learning approaches to machine learning.

Meta learning is a subfield of Machine learning where automatic learning algorithms are applied on meta-data about machine learning experiments. Although different researchers hold different views as to what the term exactly means (see below), the main goal is to use such meta-data to understand how automatic learning can become flexible in solving different kinds of learning problems, hence to improve the performance of existing learning algorithms.

Flexibility is very important because each learning algorithm is based on a set of assumptions about the data, its inductive bias. This means that it will only learn well if the bias matches the data in the learning problem. A learning algorithm may perform very well on one learning problem, but very badly on the next. From a non-expert point of view, this poses strong restrictions on the use of machine learning or data mining techniques, since the relationship between the learning problem (often some kind of database) and the effectiveness of different learning algorithms is not yet understood. [10]

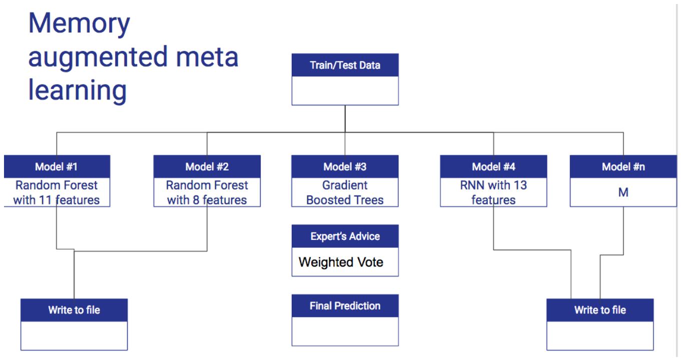


Figure: 1.8 The accuracy on the 50% of the test set results in 99.6% accuracy using the Expert's Advice algorithm.

VII. Code

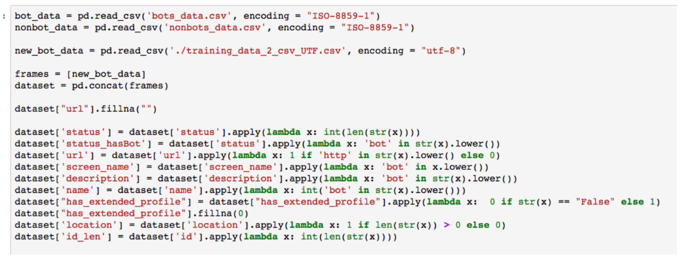
The ipython notebooks containing all the code and output files can also be viewed at:

https://github.com/ananthjan/TwitterBot

The repo contains 3 notebooks: I created over 7 different models, and hence separated the code in 3 notebooks.

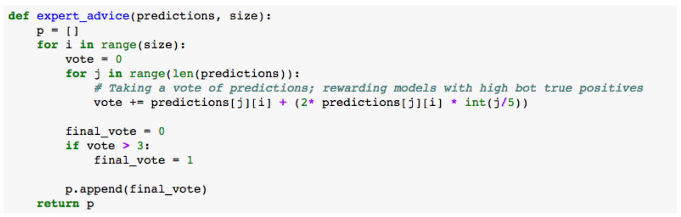
The notebook named "Twitter Bot or Not - Final Pipeline using Expert's Advice.ipynb" contains the final part of the pipeline that generated the final predictions.

The code in the notebook is well documented, and contains graphs and tables to analyze the results.



```
bot_data = pd.read_csv('bots_data.csv', encoding = "ISO-8859-1")
nonbot_data = pd.read_csv('nonbots_data.csv', encoding = "ISO-8859-1")

new_bot_data = pd.read_csv('./training_data_2_csv_UTF.csv', encoding = "utf-8")

frames = [new_bot_data]
dataset = pd.concat(frames)

dataset["url"].fillna("")

dataset['status'] = dataset['status'].apply(lambda x: int(len(str(x))))
dataset['status_hasBot'] = dataset['status'].apply(lambda x: 'bot' in str(x).lower())
dataset['url'] = dataset['url'].apply(lambda x: 1 if 'http' in str(x).lower() else 0)
dataset['screen_name'] = dataset['screen_name'].apply(lambda x: 'bot' in x.lower())
dataset['description'] = dataset['description'].apply(lambda x: 'bot' in str(x).lower())
dataset['name'] = dataset['name'].apply(lambda x: int('bot' in str(x).lower()))
dataset["has_extended_profile"] = dataset["has_extended_profile"].apply(lambda x:  0 if str(x) == "False" else 1)
dataset["has_extended_profile"].fillna(0)
dataset['location'] = dataset['location'].apply(lambda x: 1 if len(str(x)) > 0 else 0)
dataset['id_len'] = dataset['id'].apply(lambda x: int(len(str(x))))
```

```
bot_data = pd.read_csv('bots_data.csv', encoding = "ISO-8859-1")
nonbot_data = pd.read_csv('nonbots_data.csv', encoding = "ISO-8859-1")

new_bot_data = pd.read_csv('./training_data_2_csv_UTF.csv', encoding = "utf-8")

frames = [new_bot_data]
dataset = pd.concat(frames)

dataset["url"].fillna("")

dataset['status'] = dataset['status'].apply(lambda x: int(len(str(x))))
dataset['status_hasBot'] = dataset['status'].apply(lambda x: 'bot' in str(x).lower())
dataset['url'] = dataset['url'].apply(lambda x: 1 if 'http' in str(x).lower() else 0)
dataset['screen_name'] = dataset['screen_name'].apply(lambda x: 'bot' in x.lower())
dataset['description'] = dataset['description'].apply(lambda x: 'bot' in str(x).lower())
dataset['name'] = dataset['name'].apply(lambda x: int('bot' in str(x).lower()))
dataset["has_extended_profile"] = dataset["has_extended_profile"].apply(lambda x:  0 if str(x) == "False" else 1)
dataset["has_extended_profile"].fillna(0)
dataset['location'] = dataset['location'].apply(lambda x: 1 if len(str(x)) > 0 else 0)
dataset['id_len'] = dataset['id'].apply(lambda x: int(len(str(x))))
```



```
def expert_advice(predictions, size):
    p = []
    for i in range(size):
        vote = 0
        for j in range(len(predictions)):
            # Taking a vote of predictions; rewarding models with high bot true positives
            vote += predictions[j][i] + (2* predictions[j][i] * int(j/5))

        final_vote = 0
        if vote > 3:
            final_vote = 1

        p.append(final_vote)
    return p
```

Figure: 1.9 Few code samples. Visit the git repo for the detailed, fully documented source code

VIII. Video Link

A video explaining the approaches used can be viewed at: https://youtu.be/gsnxBxtJH-0

IX. Conclusion

Out of the individual models used, a finely tuned Random Forrest Classifier gives a 97% accuracy on the test dataset. Generally, different combinations of features in a random forest gives us a 92%+ accuracy consistently. SVM classifier gives a 64% accuracy and a neural network produces about 80-85% accuracy depending on the layers.

However, by combining the results of multiple classifier, and taking a majority vote along with weighted votes, we can achieve up to ***99.6% accuracy on 50% of the test dataset***.

This accuracy is very high for the given data set, and can be attributed to the fact that some of the test data points occur in the training dataset. It would be interesting to see how the model performs in a real life scenario.

The model can further be polished to include features like the sentiment of tweet, the tweet content, time analysis of posts and the frequency of tweets.

REFERENCES

[1] http://www.adweek.com/digital/heres-how-many-people-are-on-facebook-instagram-twitter-other-big-social-networks/

[2] http://www.pewresearch.org/fact-tank/2014/09/24/how-social-media-is-reshaping-news/

[3] BotOrNot: A System to Evaluate Social Bots https://arxiv.org/pdf/1602.00975.pdf

[4] Bot followers. http://mefollowers.com/index.php#

[5] Honeypot to obtain bot followers: http://www.erinshellman.com/bot-or-not/

[6] Random Forrest Classifiter: Leo Breiman and Adele Cutler https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[7] Classification and Regression with Random Forrest : http://www.bios.unc.edu/~dzeng/BIOS740/randomforest.pdf

[8] Support Vector Machine for classification: http://machinelearningmastery.com/support-vector-machines-for-machine-learning/

[9] https://en.wikipedia.org/wiki/Randomized_weighted_majority_algorithm

[10] https://en.wikipedia.org/wiki/Meta_learning_(computer_science)