**Machine Learning Final Project Result**

**Twitter Bot Detect**

Ye Yuan Duan Wang

yy2068@nyu.edu dw2201@nyu.edu

**Abstract**

Twitter robots are created for many purposes. Too many robots will do a harm to the experience of human users. In this mid-way report, we will do some simple filter of the data set and apply decision tree algorithm and evaluate current result and plane a further improve and schedule other algorithm.

**Keyword -** *machine-learning, bot, Twitter*

I. INTRODUCTION

In this report, we will do some simple filter of the data set and apply decision tree algorithm and evaluate current result and plane a further improve and schedule other algorithm. To apply a decision-tree algorithm, we process the data into a more simple way: feature attribute 'id' by length which can imply id created time; apply logarithm to attribute 'followers\_count' value which can decrease the size of tree and so on. Then we apply classification algorithm and evaluate result. We use several method and give the result in Part VI and we will discuss it in detail.

II. MOTIVATION

Twitter robots are created for many purposes, like spreading rumors, winning a lottery,  or advertising products, or just following some specific users. Too many robots will do a harm to the experience of human users. The users may loss competition with someone got many bots followers. The companies need twits by users to make decision, and the robots user may mislead their judgment. So it is important to classify human users with robots.

III. RELATED WORK

*-https://www.r-bloggers.com/programming-a-twitter-bot-and-the-rescue-from-procrastination/*

*-John P. Dickerson,Vadim Kagan,V.S. Subrahmanian “Using Sentiment to Detect Bots on Twitter: Are Humans more Opinionated than Bots?”*

*-David Mandell Freeman “Using Naive Bayes to Detect Spammy Names in Social Networks”*

*-https://www.technologyreview.com/s/529461/how-to-spot-a-social-bot-on-twitter/  -July 28, 2014*

*- Fame for sale: efficient detection of fake Twitter followers, Stefano Crescia,b, Roberto Di Pietrob , Marinella Petrocchia , Angelo Spognardia,1,∗ , Maurizio Tesconia*

*Detecting Automation of Twitter Accounts: Are You a Human, Bot, or Cyborg? -Zi Chu, Steven Gianvecchio, Haining Wang, Senior Member, IEEE, and Sushil Jajodia, Senior Member, IEEE*

IV. DATA

The data set came from twitter API, where we can obtain eighteen attributes of a normal account: *id; id\_str; screen\_name; location; description; url; followers\_count; friends\_count; listedcount; created\_at; favourites\_count; verified; statuses\_count; lang; status; default\_profile; default\_profile\_image; has\_extended\_profile; name.*

Then, each group of the class manually detect 50 robot data and 50 non-robot data. Combine all groups works, we get 2232 twitter account data finally.

 In mid-way case, since we will apply decision-tree algorithm to build model, we deal some tricks on the data to decrease tree complexity and filter some meaningless data by feature data value into a more meaningful form:

- since attribute 'id\_str' is a type convert from 'id', we delete 'id\_str' from work data set.

- 'id' is a primary key of data set. Consider it in entropy calculation is meaningless. Instead we apply log10 as addition created time info. Yet in some algorithm, log10 is poorer than the original count.

- 'screen\_name' contain some key words which can bring some important information. We build a simple key words dictionary ["bot"] and detect whether user's screen name contain one or more of the key words. If has, mark it as 1, if not, mark 0. To improve our score of the algorithm, we may increase size of our key word dictionary in a future version.

- Since we don't have an efficient method to check value in 'location' is a real location on the earth or not, we fail to use above method matching the string. However, some users leave the location info. as blank which give us some tiny signal. Thus, we check if 'location' attribute is nan or not and return 0 or 1. Also we found some seems like wrong location like ’botkitch’, which lead us applying checking special words on this attribute.

- 'description' may contain some key words which straightforward reveal a robot account. But lots of account in the data set remain this field as blank, it is also necessary to check if this field is nan or not. We initiate with a simple words dictionary of ‘bot’ and ‘robot’ and during the work we try different dictionary. Only check if a bot like word in the sentence or not is not accurate enough, apply a bayes method here seems more reasonable. We will discuss in the Addition Method Part below.

- 'url' field allow users put their own personal page on it. The 'url' in data set is automatically produced by Twitter with some encryption. We can always ping the server " /t.co" successfully. Thus, we can only check whether user has their page upload in this field. As for check if this url is accessible or check the content of the sites like bot or not, we will discuss in Addition Method Part below.

- 'followers\_count', 'friends\_count' and ‘status\_count’are important attributes in data set. However, their value have a large range from 0 to more than 10 thousand. We here apply logarithm with base 2 on it and decrease possible child number of these two nodes within 10. However, according to our result of test, we learn that these attribute actually not that important. The result correct us that what we thought straightforward may mislead us. Some result reveal a totally different truth with we expected. The final result is only a high level of amount of followers as well as friends can provide some clusion of classification. Yet in some algorithms, log2 is poorer than the original number.

- 'created\_at' presented in a date form whose value span from 2013 to 2017. Considering reduce complexity of the tree, we category 'created\_at' in one half year interval form year 2013-2017, which return 13 possible children (0 as before 2013). This attribute play an important role. Because bot account my reference to some big event such as the vote, war or new production publish. The bot accounts associate with these event will boost in a certain period. As well as the effect by apply the logarithm method on `id` attribute, create time can tell us something.

- Since our computer can't correctly display some value in the field 'name' (like a French name or a Japanese name), we drop this attribute. We can neither put this almost unique value into consideration nor to figure out special information hide behind. We try method to figure out a real name's form and check the value if it's a normal name but with a trivial effect on the final result. We will discuss this latter.

- 'status' has a mess of information include other attribute value. The length of the value can somehow measure the account activity rate. We count the number of attribute included in 'status' field as the feature number. We also try use the same method mentioned in `description` attribute and the method based on bayes. However, the result is not good as we expected. The status value contain lots of neutral words, calculate them is meaningless. But check if the status value has some sensitive words provide some help. We will discuss this latter

- 'lang', ' default profile'' 'default profile image' and 'has extended profile' conform with our aim to improve data set, we leave it unchanged.

With all featured new value ordered and some attributes dropped, we get a new data set ready for next step.

Addition method for data clean:

-Since the final training data set has a mess of noise (including mistake data type expected, nan value and mistake corresponding data with column title). An improvement of data clean method is necessary. A simple way to do this is just check the error data and drop them.

-While testing, we try some new method according to what we learn about the attribute pattern trying to improve our predict score.

We try to check if user’s name value is a true normal human name by checking the form. The final result of this method is just average, nothing bad and nothing good. Though this method, we found that a bot account can easily create a real form name and a healthy account may have a wrong name just for fun!

We try to access url the user given and check if the url is accessible or if their sites has some sensitive words mention ‘bot’ (we found that a lot of bot account given an accessible url leading to a site introducing their bot products). But this method cost a lot and need an internet environment. What’s worse is that some url can not be access by python ‘urllib.request’ prevented by some firewall or third party program which cause a block of the method.

We use the method in assignment 3 using bernoulli or multi bayes method apply on attribute like `description` and `statues` which contain a set of words may contain some clusion. (The final result show that this help a little. Because it’s actually a redundancy algorithm applied. We do the same work in the latter algorithm, thus, this method won’t help). Finally we just check if it is none as a feature.

We also try methods to combine and compress multi-attribute into one or two by extract their relationship between each other. These method aimed on reduce cost of the algorithm and extract the pattern between two attribute into consideration. The most classical case of this goal is between `followers\_count` and `friends\_counts` which is a strong check clusion we used in midway to check if an account is a bot or not. We think that accounts has a large rate of `followers\_count` by `friends\_counts` are more likely bots. However, the result suffer a lot because of noise data and also a redundancy work with our classification algorithm we used. It is oblivious that combine multi-attribute into one will cause a loss of information and since our test and train data size have a small size, the reduce of cost of the algorithm is meaningless.

Finally, for each attributes:

‘id’: Take the original number for AdaBoost, Gradient Boosting and Combination\_2, the rest take log10.

‘id\_str’: Don’t take it as a feature.

‘screen\_name’, ‘descrpiton’, ‘name’: check if it contains ‘bot’ and ‘Bot’

‘location’, ‘status’, ‘url’: check if it is none or not

‘followers\_count’,‘friends\_count’,‘listed\_count’, ‘favorites\_count’, ‘status\_count: Take the original number for AdaBoost, Gradient Boosting and Combination\_2, the rest take log2.

‘created\_at’: map the create time to 13 values, 1 for none.

‘verified’, ‘default\_profile’, ‘default\_profile\_image’, ‘has\_extended\_profile’: take the boolean value.

‘lang’: map the lang to 5 values.

V. ALGORITHM USED

Basic methods:

**Decision Tree**

A decision tree is a hierarchical data structure implementing the divide-and-conquer strategy. It is an efficient nonparametric method, which can be used for classification[1]. And compared with other methods, decision tree is easy to implement. Also it may become the base of other methods, like bagging, boosting or random forests. So we apply it into practice first.

As mentioned in the data part, we do modification on the origin data, like mapping the string attribute to an integer or a bool. This helps to make the tree simpler. Then it’s a common decision tree part: We apply the recursion on make tree. On each node, we first check if it is a leaf. If all the datas here are same in result, which means all bots or all non-bots, we mark this node to be a leaf and return. Otherwise, first, we record the percent of the ‘bot’ in the data in this node. Then, we catch the entropy of each attribute, take the attribute with the least attribute, or the highest information gain, divide the data list into sublists, each sublist are in the attribute, and perform make tree function recursively on each subtree.

One question is that, since we modify but not use the original data, it is possible that: several datas are same in other attributes, but are different in ‘bot’ attribute. For this situation, we modify the make tree part. In each node, it will get not only the datalist, but also a list of attribute, when choose one attribute to divide data, it will remove the attribute from attribute list for building its children. When there is not attribute that can use be to divide, we also mark the node leaf. For this kind of node, we check the percent of ‘bot’ here. If there are more than or equal to half data are bots, we return bot for this leaf, otherwise we return non-bot.

After training the tree, we begun the classification. On each node, if it is not a leaf, by the attribute in the node and the corresponding value of the attribute in the input, we choose a child from the node and perform search on the child node. There is also one situation that: the value of the attribute from the input cannot find the corresponding child from the node. In this situation, we check the percent of ‘bot’ in this node. If it is more than 0.5(which means more than half of the data here are ‘bot’ when training the tree), we treat the input data as ‘bot’ and return bot, otherwise we return non-bot.

By the build tree and classification method, we got the decision tree.

**Naive bayes:**

Naive Bayes Classifiers are a family of simple probabilistic classifiers based on applying Bayes’ theorem with strong (naive) independence. It’s a also a basic parametric method and easy to implement. Here we also wrote the code on ourselves. It is a binary (2-class) classification based on the same attributes we use for the decision tree.

In our own naive bayes code. First, based on the data, we calculate the likelihood P(x|C) of all the values on each attribute for the two classes: bot and non-bot. By this we build our model. Also we will calculate the prior P(C) of the two classes ‘bot’ and ‘non-bot’. Second, for each data to predict, we calculate the posterior P(C|x)\*P(x) for ‘bot’ and ‘non-bot’, which is P(x|C) \* P(C), based on Bayes Theorem, and choose the class with high posterior probability to be the predict result. To avoid the situation that one data to predict get 0 as likelihood in some attributes, we apply k-smoothing here, in which we set k to be 0.01.

**Logistic regression:**

Logistic regression is a regression model. Compared with linear regression, which gave continuous value as predict result, logistic regression gives categorical values. So it can be also used as a classifier, especially for binary classification. Compared with naive bayes and decision tree, it can handle the situation that the attributes take value from a continuous and infinite domain but not only on finite categorical values.

We didn’t write logistic regression on our own. We use the function in scikit-learn, sklearn.linear\_model.LogisticRegression(). For the parameters, we set C, the inverse of regularization strength, to be 1000, which we found can get the best accuracy, and the other parameters as default.

**Boosting Methods:**

In basic methods, we use a single learner to make a model. Besides improve our models or improve our algorithm, we could also use several weak learners together to make a strong learner. Boosting is one method to combine weak learners. In boosting, we make several learners. Each learner wll compensat the shortcommings of existing weak learners.

**AdaBoost**

AdaBoost, short for adaptive boosting, is one of the boosting method. It use the mistake predicts made by the previous classifier as the shortcomings to compensate. In each iteration, it will build a training set by randomly choose data from the origin training set on some weights. After building the training set and train the learner, it predicts the whole origin training set. The data which is predicted wrongly will get a higher weight, which means it will be more likely to choose in the next iteration. After finishing the iteration, we get several weak learners, and it is a voting method. Given an instance,  all learners will make the decision and the decisions will be combined by a weighted learner, where the weight of each learner based on the accuracy of the simple learner.

At first we wrote an adaboost based on our own decision tree, but it gets a very poor result, the second weak learner’s accuracy will be less than 50%, so we choose to use the adaboost in the scikit-learn, sklearn.ensemble.AdaBoostClassifier(), with arguments n\_estimators=100, the maximum number of estimators at which boosting is terminated and learning\_rate=1.2, an attribute to shrinks the contribution of each classifier. It is based on sklearn.tree.DecisionTreeClassifier(), which has a better performance on our own decision tree.

**Gradient Boosting**

Gradient Boosting is another boosting method. It takes the gradient as the shortcomings to compensate. For the model F to predict values in form of y’=F(x), with the true value of y, 1the boosting try to minimizing the mean squared error (y’-y)^2 to get closer to the true value y. At each iteration 1<=m<=M of gradient boosting, from the model Fm(x), we get a function h(x) = y-Fm(x). Gradient boosting tries to fit h(x), and use the fitted h’(x)+Fm(x) as Fm+1(x), and continuous to make a better model.

We didn’t write the gradient boosting on ourselves. We use the function in scikit-learn, sklearn.ensemble.GradientBoostingClassifier(), use decision tree as base learner. In we use the following arguments: n\_estimators=100, learning\_rate=0.2 (there two arguments are the same in adaboost) and max\_depth=4, the maximum depth of the individual decision tree.

Our own method

**Combination\_1:**

We combine Naive bayes, decistion tree (written by ourselves) and logistic regression together. That is, we make for model, make 3 predictions for each data, and take the average as the result. It gets a better performance than the three base classifier.

**Combination\_2:**

We combine the AdaBoost, gradient boosting and logistic regression together, and it works like the combination\_1 do.

VI. RESULT

**Decision Tree:**

First we randomize the order of the data(since it is all ‘bot’ row continued with all ‘non-bot’ row), then we use 10-Fold Cross-Validation to get the performance of the data. We calculate accuracy, precision, recall and f1score here.

Here is one result:

accuracy: 0.82846732863549

precision: 0.8273923317148277

recall: 0.8036031017719789

f1score: 0.8145687083986358

Because we randomized the order of data in the beginning, it returns different answer when we calculate it. So we perform 5 randomize in total, each time perform a 10-Fold Cross-Validation, and return the average.

accuracy: 0.8347425528507364

precision: 0.8308946302345429

recall: 0.8178888758569414

f1score: 0.8236313847394834

VII. CODE

Brief code:

Modification data:

*import pandas as pd*

*import numpy as np*

*import math*

*import dateutil.parser as dparser*

*import mybayes*

*import random*

*import decision*

*import urllib.request*

*#data pre process*

*def checkNone(data\_source, data\_class, attrstr):*

*#return None:1 not-None:*

*data\_class[attrstr]=0*

*idx = 0*

*for value in data\_source[attrstr]:*

*if value == "" or value == "None" or value == "\"\"" or value == "null" or pd.isnull(value):*

*data\_class.loc[idx,attrstr] = 1*

*idx += 1*

*def checkTF(data\_source, data\_class, attrstr):*

*#return True:0 False:1*

*data\_class[attrstr]=0*

*idx = 0*

*count=0;*

*for value in data\_source[attrstr]:*

*v = value*

*if (type(value)==str):*

*if value == "TRUE" or value == 'true':*

*v=True*

*else:*

*v=False*

*if v == False:*

*data\_class.loc[idx,attrstr] = 1*

*else:*

*count+=1*

*idx += 1*

*def checkNumber(data\_source, data\_class, attrstr, effi):*

*#return n=log with base-effi 0:non-int type*

*data\_class[attrstr]=0*

*idx = 0*

*for value in data\_source[attrstr]:*

*v = value;*

*try:*

*v = np.int64(v)*

*except ValueError:*

*v = 0;*

*if (idx==0):*

*print(type(v))*

*if v<0:*

*v=0;*

*if type(v) is np.int64:*

*if effi == 10:*

*data\_class.loc[idx,attrstr]= int(math.log10(v+1))*

*elif effi == 2:*

*data\_class.loc[idx,attrstr] = int(math.log2(v+1))*

*else:*

*data\_class.loc[idx,attrstr] = int(math.log(v+1, effi))*

*else:*

*data\_class.loc[idx,attrstr]=0*

*idx += 1*

*def checkEnglish(data\_source, data\_class, attrstr):*

*#return isEnglish:1 other lang:2-3 errortype:0*

*data\_class[attrstr]=0*

*idx = 0*

*for value in data\_source[attrstr]:*

*if value == "en":*

*data\_class.loc[idx,attrstr] = 1*

*elif value == "fr":*

*data\_class.loc[idx,attrstr] = 2*

*elif value == "ja":*

*data\_class.loc[idx,attrstr] = 3*

*elif value == "ko":*

*data\_class.loc[idx,attrstr] = 4*

*idx += 1*

*def checkDate(data\_source, data\_class, attrstr):*

*#date index:*

*# year<=2012: 0:3*

*# year==2013: 4:7*

*# year==2014: 8:11*

*# year==2015: 12:15*

*# year==2016: 16:19*

*# year==2017: 20:23*

*data\_class[attrstr]=0*

*idx = 0*

*for value in data\_source[attrstr]:*

*times =0;*

*try:*

*y = dparser.parse(value,fuzzy=True).year*

*m = dparser.parse(value,fuzzy=True).month*

*if y ==2013:*

*times = 4*

*elif y == 2014:*

*times = 8*

*elif y == 2015:*

*times = 12*

*elif y == 2016:*

*times = 16*

*elif y == 2017:*

*times = 20*

*#month*

*if m>6:*

*times+= 1*

*except TypeError:*

*times=24*

*data\_class.loc[idx,attrstr]=times*

*idx += 1*

*botDict = ["bot",*

*"Bot"]*

*def checkStringBot(data\_source,data\_class, attrstr):*

*#return has sub string in botDict:1 else:0*

*data\_class[attrstr]=0*

*idx = 0*

*for value in data\_source[attrstr]:*

*if type(value) is str:*

*for dic in botDict:*

*if value.find(dic) != -1:*

*data\_class.loc[idx,attrstr] = 1*

*else:*

*data\_class.loc[idx,attrstr] = 2*

*idx += 1*

*def checkName(data\_source, data\_class, attrstr):*

*#return invalid name:1 or not:1*

*data\_class[attrstr]=0*

*idx = 0*

*for value in data\_source[attrstr]:*

*if type(value) is str:*

*if value[0]>='A' and value[0]<='Z':*

*data\_class.loc[idx,attrstr] = 1*

*else:*

*data\_class.loc[idx,attrstr] = 0*

*else:*

*data\_class.loc[idx,attrstr] = 0*

*idx += 1*

*def divideCol(data\_source, data\_class, attrstr, divideAttrstr, effi):*

*#return divide result in float32*

*data\_class[attrstr]=0*

*idx = 0*

*for value in data\_source[attrstr]:*

*if attrstr == 'id':*

*continue;*

*v = value;*

*try:*

*v = np.int64(v)*

*except ValueError:*

*v = 0;*

*if (idx==0):*

*print(type(v))*

*if v<0:*

*v=0;*

*if type(v) is np.int64:*

*v\_d = data\_source.loc[idx, divideAttrstr]*

*v\_d = np.int64(v)*

*if v\_d<= 0:*

*v\_d = 1*

*v\_r = math.fabs(np.float32(v)/np.float32(v\_d))*

*data\_class.loc[idx,attrstr] = v\_r*

*else:*

*data\_class.loc[idx,attrstr]=0*

*idx += 1*

*def checkUrl(data\_source, data\_class, attrstr):*

*#return has sub string in botDict:1     else:0    none: 3    typeError:2   404:4*

*data\_class[attrstr]=0*

*idx = 0*

*for value in data\_source[attrstr]:*

*if value == "" or value == "None" or value == "\"\"" or value == "null" or pd.isnull(value):*

*data\_class.loc[idx,attrstr] = 0*

*elif type(value) is str:*

*#print(value)*

*if value.startswith('"') and value.endswith('"'):*

*value = value[1:-1]*

*try:*

*url = urllib.request.urlopen(value)*

*data\_class.loc[idx,attrstr] = 1*

*s = url.read()*

*for dic in botDict:*

*if dic in str(s):*

*data\_class.loc[idx,attrstr] = 1*

*else:*

*data\_class.loc[idx,attrstr] = 0*

*except:*

*data\_class.loc[idx,attrstr] = 0*

*else:*

*data\_class.loc[idx,attrstr] = 0*

*idx += 1*

*clf=["",""]*

*def checkBayesWords(data\_source, data\_class, attrstr, isBernoulli, indexOfC)*

*#return result from bayes method*

*global clf*

*#return two class: 1/0/2 as bot or not bot or nan*

*#Train a model and return predict result for both train and test set*

*data\_class[attrstr]=0*

*doc = []*

*target = []*

*countertrain = 0*

*countertest = 0*

*for idx in range(len(sData\_train)):*

*value = sData\_train[attrstr][idx]*

*if value == "" or value == "None" or value == "\"\"" or value == "null" or pd.isnull(value):*

*continue*

*else:*

*doc.append(str(value))*

*target.append(sData\_train[labelStr][idx])*

*countertrain = countertrain+1*

*for idx in range(len(sData\_test)):*

*value = sData\_test[attrstr][idx]*

*if value == "" or value == "None" or value == "\"\"" or value == "null" or pd.isnull(value):*

*continue*

*else:*

*doc.append(str(value))*

*target.append(sData\_test[labelStr][idx])*

*countertest = countertest+1*

*#train*

*count\_vect = CountVectorizer()*

*X\_counts = count\_vect.fit\_transform(doc)*

*X\_train\_counts = X\_counts[0:countertrain];*

*X\_test\_counts = X\_counts[countertrain:countertrain+countertest];*

*train\_target = target[0:countertrain];*

*test\_target = target[countertrain:countertrain+countertest];*

*if (clf[indexOfC] == ""):*

*tfidf\_transformer = TfidfTransformer()*

*X\_train\_tfidf = tfidf\_transformer.fit\_transform(X\_train\_counts)*

*if(isBernoulli == True):*

*clf[indexOfC] = BernoulliNB().fit(X\_train\_tfidf, train\_target)*

*else:*

*clf[indexOfC] = MultinomialNB().fit(X\_train\_tfidf, train\_target)*

*predictss = clf[indexOfC].predict(X\_train\_tfidf)*

*#predict*

*cc = 0;*

*for idx in range(len(data\_source)):*

*value = data\_source[attrstr][idx]*

*if value == "" or value == "None" or value == "\"\"" or value == "null" or pd.isnull(value):*

*#data\_Test[attrstr][counter] = "NaN NaN"*

*data\_class.loc[idx,attrstr] = 2*

*else:*

*data\_class.loc[idx,attrstr] = predictss[cc]*

*cc+=1;*

*else:*

*tfidf\_transformer = TfidfTransformer()*

*X\_test\_tfidf = tfidf\_transformer.fit\_transform(X\_test\_counts)*

*predictss = clf[indexOfC].predict(X\_test\_tfidf)*

*print(len(predictss))*

*print(len(data\_source))*

*cc = 0;*

*for idx in range(len(data\_source)):*

*value = data\_source[attrstr][idx]*

*if value == "" or value == "None" or value == "\"\"" or value == "null" or pd.isnull(value):*

*#data\_Test[attrstr][counter] = "NaN NaN"*

*data\_class.loc[idx,attrstr] = 2*

*else:*

*#data\_Test[attrstr][counter] = value*

*#counter = counter+1*

*data\_class.loc[idx,attrstr] = predictss[cc]*

*cc+=1;*

*def DataFrameFilter(dataSource):*

*data\_class = dataSource[['bot']].copy(deep=True)*

*attrList = dataSource.columns.values*

*for attrStr in attrList:*

*if attrStr == "id":*

*checkNumber(dataSource, data\_class, attrStr, 10)*

*print(attrStr)*

*elif attrStr == "id\_str":*

*print(attrStr)*

*elif attrStr == "screen\_name":*

*checkStringBot(dataSource, data\_class, attrStr)*

*print(attrStr)*

*elif attrStr == "location":*

*checkStringBot(dataSource, data\_class, attrStr)*

*print(attrStr)*

*elif attrStr == "description":*

*checkStringBot(dataSource, data\_class, attrStr)*

*elif attrStr == "url":*

*checkNone(dataSource, data\_class, attrStr)*

*print(attrStr)*

*elif attrStr == "followers\_count":*

*checkNumber(dataSource, data\_class, attrStr, 2)*

*print(attrStr)*

*elif attrStr == "friends\_count":*

*checkNumber(dataSource, data\_class, attrStr, 2)*

*print(attrStr)*

*elif (attrStr == "listedcount" or attrStr == "listed\_count"):*

*checkNumber(dataSource, data\_class, attrStr, 2)*

*print(attrStr)*

*elif attrStr == "created\_at":*

*checkDate(dataSource, data\_class, attrStr)*

*print(attrStr)*

*elif (attrStr == "favorites\_count" or attrStr=="favourites\_count"):*

*checkNumber(dataSource, data\_class, attrStr, 2)*

*print(attrStr)*

*elif attrStr == "verified":*

*checkTF(dataSource, data\_class, attrStr)*

*print(attrStr)*

*elif attrStr == "statuses\_count":*

*checkNumber(dataSource, data\_class, attrStr, 2)*

*print(attrStr)*

*elif attrStr == "lang":*

*checkEnglish(dataSource, data\_class, attrStr)*

*print(attrStr)*

*elif attrStr == "status":*

*checkNone(dataSource, data\_class, attrStr)*

*print(attrStr)*

*elif attrStr == "default\_profile":*

*checkTF(dataSource, data\_class, attrStr)*

*print(attrStr)*

*elif attrStr == "default\_profile\_image":*

*checkTF(dataSource, data\_class, attrStr)*

*print(attrStr)*

*elif attrStr == "has\_extended\_profile":*

*checkTF(dataSource, data\_class, attrStr)*

*print(attrStr)*

*elif attrStr == "name":*

*checkStringBot(dataSource, data\_class, attrStr)*

*print(attrStr)*

*return data\_class*

#Make Decision Tree

*def entroy(df,name,resultname,indexlist):*

*#calculate entropy of one attribute*

*def makesubtree(df,namelist,resultname,indexlist):*

*#calculate the percent of bots on this node*

*….*

*mynode.percent = pos / (pos+neg)*

*if (pos\*neg==0):*

*mynode.leaf = True*

*mynode.judge = t*

*return mynode;*

*min=32767*

*#calculate the entropy and choose attribute to use*

*for name in namelist:*

*val = entroy(df,name,resultname,indexlist)*

*if val<min:*

*min = val*

*ans = name*

*#mark it leaf if there is no attribtue to choose*

*if (min==32767):*

*mynode.leaf = True*

*….*

*return mynode*

*#add children and make subtrees*

*….*

*for key in ma.keys():*

*mynode.addchild(key,makesubtree(df,namelist,resultname,ma[key]))*

*….*

*return mynode;*

*def judge(node,ma):*

*if node.leaf:*

*return node.judge;*

*else:*

*try:*

*return judge(node.list[ma[node.attribute]],ma);*

*except KeyError:*

*# we can’t find the value in the node, so we just return the majority*

*return node.percent > 0.5;*

Poject branch:

*https://github.com/Dwan9/TwitterBot\_CS6923.git*

VIII. EVALUATION

IX. CONCLUTION

**reference**

[1] Introduction to Machine Learning 2e -Ethem Alpaydin

[2]John P. Dickerson,Vadim Kagan,V.S. Subrahmanian “Using Sentiment to Detect Bots on Twitter: Are Humans more Opinionated than Bots?”

[3]David Mandell Freeman “Using Naive Bayes to Detect Spammy Names in Social Networks”

[4]Fame for sale: efficient detection of fake Twitter followers, Stefano Crescia,b, Roberto Di Pietrob , Marinella Petrocchia , Angelo Spognardia,1,∗ , Maurizio Tesconia

[5]Detecting Automation of Twitter Accounts: Are You a Human, Bot, or Cyborg? -Zi Chu, Steven Gianvecchio, Haining Wang, Senior Member, IEEE, and Sushil Jajodia, Senior Member, IEEE