# CS-109A Introduction to Data Science

**Final Project - Milestone 4**
**Project : Machine Learning & Analysis for Twitter Bot Detection**
**Contributors: Group #68: Nisrine Elhauz, Huan Liu, Fayzan Talpur, Samasara Tamrakar**
**Harvard University**
**Fall 2018**

---

<a id ='TOC'></a>

**Table of Contents**

1. Introduction
   1.1 Motivation
   1.2 Problem Statement
2. Data
   2.1 Data in Twitter API
   2.2 Collection of Data
   2.3 Data Labelling: Using Botometer
   2.4 Data Labelling: Manually
   2.5 Data Collection: Most Recent 100 Tweets per Each User
   2.6 Description of Raw Data (Tweets)
3. Exploratory Data Analysis
   3.1 Data Wrangling & Cleansing
   3.2 Feature Engineering
   3.3 Advanced Feature Engineering - NLP Features
   3.4 Important Features
   3.5 Relations in Data
   3.6 Standardization and Discussion
4. Models
   4.1 Baseline Model - Simple Linear Regression
   4.2 Linear Regression with Ridge
   4.3 Lasso
   4.4 Lasso and Ridge Coefficients Comparison
   4.5 Logistic Regression
   4.6 Logistic Regression with cross validation
   4.7 Logistic Regression with polynomial degree 3
   4.8 KNN
   4.9 Decision tree
   4.10 Random Forest
   4.11 Boosting - AdaBoost Classifier
   4.12 SVM
   4.13 K-Means Clustering
   4.14 Validate Botometer Results
   4.15 Sentence Embeddings + Clutering + Neural Networks
5. Results and Conclusion
   5.1 Summary of Results
   5.2 Noteworthy Findings
   5.3 Conclusion and Future Work
6. Literature Review and Related Work
   6.1 Bots in the Twittersphere
   6.2 How Twitter Bots Help Fuel Political Feuds
   6.3 The spread of low-credibility content by social bots
   6.4 Twitter Topic Modeling by Tweet Aggregation

Back to TOC
<a id ='Introduction'></a>

## 1- Introduction

The main objective of the project is explore twitter dataset using twitter API and try to create a learning algorithim that can differentiate between bot and human Twitter account.

Back to TOC
<a id ='Motivation'></a>

#### 1.1 - Motivation

With the increase of popularity of social media, our traditional channels of receiving news and information have been greatly shifted from newspaper, television, conversation with neighbors, to social media, such as Facebook, Twitter, Instagram, etc.

There has been a lot of news over the past few years about the impact of bots on these platforms and how they are trying to influence people's opinions and perceptions. It is important to be able to detech these bots so we can recognize their influence on human users, and how common they are on these platforms.

Back to TOC
<a id ='Problem-Statement'></a>

#### 1.2 - Problem Statement

How to detect Twitter Bots using tweets data from Twitter developer API by using machine learning techniques. Our objective is to determine whether the source of tweets are from accounts that are bot users [1] or non-bot users [0]. (we define bot as: no direct human involvement in generating tweets)

1. Start by collection data using Twitter API and encode using Botometer API and manual verifications
2. Perform feature engineering and preprocessing techniques to aggregate tweet features to account level features
3. Use Data visualization to understand the trend and patterns.
4. Solve as a classification problem of classifying an account to be bot or not-bot
5. Explore as an unsupervised problem of clustering twitter accounts into 2 (or several) clusters
6. Explore tweet topics / patterns

In [201]:
```
#@title
# Import Libraries, Global Options and Styles
import requests
from IPython.core.display import HTML
styles = requests.get(
    "https://raw.githubusercontent.com/Harvard-IACS/2018-CS109A/master/c
ontent/styles/cs109.css").text
HTML(styles)
%matplotlib inline

#import libraries
import warnings
warnings.filterwarnings('ignore')

import tweepy
import random
random.seed(112358)

%matplotlib inline
import numpy as np
import scipy as sp
import json as json
import pandas as pd
import jsonpickle
import time


from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.utils import resample
from sklearn.utils import shuffle
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegressionCV
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import r2_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import PolynomialFeatures
from pandas.plotting import scatter_matrix
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import LassoCV
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.preprocessing import LabelEncoder

import scipy.sparse as ss
import os
import tensorflow as tf
import tensorflow_hub as hub

from tensorflow.keras.models import load_model
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense,Dropout
from tensorflow.keras.utils import np_utils

import statsmodels.api as sm
from statsmodels.api import OLS

import matplotlib as mpl
import matplotlib.cm as cm
import matplotlib.pyplot as plt

import pandas as pd
pd.set_option('display.width', 1500)
pd.set_option('display.max_columns', 100)
pd.set_option('display.notebook_repr_html', True)

import seaborn.apionly as sns
sns.set(style="darkgrid")
sns.set_context("poster")
```

Back to TOC
<a id ='Data'></a>

# 2 - Data

We started with three potential approaches to collect data for bot detection:

***Approach 1: Collect Tweets then Label Bot / Real Users:***

*Approach* - Collect Tweets via Twitter API, extract accounts, then use Botometer API to label bot / real-user
*Pros* - flexible in collecting the specific data we are interested
*Cons* - using Botometer to label accounts might result in a fitting of Botometer's algorithms

***Approach 2: Collect Bot and Real Users Separately***

*Approach* - Manually select / verify bots account, use verified twitter profile accounts for the real user dataset, then use Twitter API to collect Tweets from the selected accounts
*Pros* - very accurate response (bot / real user)
*Cons* - time consuming and therefore small data size

***Approach 3: Use Existing Annotated Dataset***

*Approach* - Use existing datasets that have already labelled the tweets as bot / real-user
*Pros* - convenient
*Cons* - less flexible in collecting tweets with specific topic; results highly rely on the labelling accuracy

After evaluating the three approaches, we decided to collect our own tweets and use Botometer to label the bot / real-user. We decided to use the following approach to collect and label our data:

***Step 1: Collection of Data : Collect over 6,000 tweets***

***Step 2: Data Labelling: Use Botometer***

***Step 3: Data Labelling: Chose to Manual Verify 40 accouts (20 bots, 20 actual users)***

Back to TOC
<a id ='Data-in-Twitter-API'></a>

### 2.1 - Data Source: Twitter API with Tweepy

We used Twitter API via Tweepy to collect all our data by searching for tweets that include certain keywords, and by retrieving most recent 200 tweets from speicified users.

Back to TOC
<a id ='Collection-of-Data'></a>

### 2.2 - Collection of Data : Collect over 6,000 Tweets using Keywords

We first collected some tweets that contains one of the following keywords that are likely to lead to controversial topics:

> 1) Immigration
> 2) Brexit
> 3) bitcoin

We used keywords of more controversial topics as those are more likely to have non-obvious bots.

In [9]:
```
# http://www.tweepy.org/
import tweepy

# Replace the API_KEY and API_SECRET with your application's key and sec
ret.
auth = tweepy.AppAuthHandler("apikey", "api secret")

api = tweepy.API(auth, wait_on_rate_limit=True, wait_on_rate_limit_notif
y=True)

if (not api):
    print ("Can't Authenticate")
    sys.exit(-1)
```

In [33]:
```
# The following code was adapted from sample code provided by TFs / Prof
s for this project

def collect_tweets(maxTs, requestCount, filename):
    searchQuery = 'Immgration OR Brexit OR bitcoin'  # this is what we'r
e searching for
    maxTweets = maxTs # some arbitrary large number
    tweetsPerQry = 100  # this is the max the API permits
    fName = filename # we'll store the tweets in a text file.

    # If results from a specific ID onwards are reqd, set since_id to th
at ID.
    # else default to no lower limit, go as far back as API allows
    sinceId = None

    # If results only below a specific ID are, set max_id to that ID.
    # else default to no upper limit, start from the most recent tweet m
atching the search query.
    max_id = -1
    error_count = 0

    request_count = 0

    tweetCount = 0


    print("Downloading max {0} tweets".format(maxTweets))
    with open(fName, 'w') as f:
        while tweetCount < maxTweets and request_count < requestCount:
            try:
                if (max_id <= 0):
                    if (not sinceId):
                        new_tweets = api.search(q=searchQuery, count=twe
etsPerQry)
                    else:
                        new_tweets = api.search(q=searchQuery, count=twe
etsPerQry,
                                                since_id=sinceId)
                else:
                    if (not sinceId):
                        new_tweets = api.search(q=searchQuery, count=twe
etsPerQry,
                                                max_id=str(max_id - 1))
                    else:
                        new_tweets = api.search(q=searchQuery, count=twe
etsPerQry,
                                                max_id=str(max_id - 1),
                                                since_id=sinceId)
                if not new_tweets:
                    print("No more tweets found")
                    break
                for tweet in new_tweets:
                    f.write(jsonpickle.encode(tweet._json, unpicklable=F
alse) +
                            '\n')
                tweetCount += len(new_tweets)
```

```
                print("Downloaded {0} tweets".format(tweetCount))
                max_id = new_tweets[-1].id
                request_count += 1
                time.sleep(2)
            except tweepy.TweepError as e:
                # Just exit if any error
                error_count += 1
                print("some error : " + str(e))
                time.sleep(2)
                if error_count >= 5:
                    print("too many errors ....break.")
                    break

    print ("Downloaded {0} tweets, Saved to {1}".format(tweetCount, fNam
e))
```

In [ ]:

```
# collect samples (which we will use botometer to encode)
collect_tweets(7000, 70, 'immigration_brexit_bitcoin_extended.json')
```

In [54]:

```
# load the file
raw_df = pd.read_json('immigration_brexit_bitcoin_extended.json', lines=
True)
```

In [ ]:

```
# take a look at the separate data
display(raw_df.shape)
```

In [ ]:

```
# take a look at the combined data
display(raw_df.columns.values()
display(raw_df.shape)
```

In [ ]:

```
# delete duplicate accounts
raw_df = raw_df.drop_duplicates(subset='id_str')
raw_df.shape
```

In [66]:

```
# save as csv
raw_df.to_csv('immigration_brexit_bitcoin_full.csv')

# save as json
raw_df.to_json('immigration_brexit_bitcoin_full.json')
```

Back to TOC
<a id ='Labelling-Botometer'></a>

#### 2.3 - Data Labelling: Using Botometer

We labelled each account using botometer score via Botometer API.

In [327]:

```
#load the data
raw_df = pd.read_json('immigration_brexit_bitcoin_full.json')
raw_df.shape
```

Out[327]: (13251, 31)

In [68]:
```
# add account id to dataframe
raw_df['id'] = raw_df['user'].map(lambda d: d['id'])
```

In [72]:
```
# set up botometer
# the code below was adapted from
# https://github.com/IUNetSci/botometer-python

import botometer

mashape_key = "MASHAPE KEY"
twitter_app_auth = {
    'consumer_key': 'consumer key',
    'consumer_secret': 'consumer secret',
    'access_token': 'api key',
    'access_token_secret': 'api secret',
  }

bom = botometer.Botometer(wait_on_ratelimit=True,
                          mashape_key=mashape_key,
                          **twitter_app_auth)
```

In [ ]:
```
# retrieve response objects from Botometer
botometer_results = {}
count = 0
for index, user_id in raw_df['id'][.iteritems():
    try:
        botometer_results[index] = bom.check_account(user_id)
        print(count)
        count +=1
    except tweepy.TweepError as err:
        print("Skipping user {} due to error {}".format(user_id, err))
    except NoTimeLineError as err:
        print("Skipping user {} due to error {}".format(user_id, err))
    time.sleep(2)
```

In [95]:
```
raw_df['botometer_result'].dropna().shape
```

Out[95]: (6032,)

In [84]:
```
# convert to series
botometer_series = pd.Series(botometer_results)
```

In [85]:
```
# add results to a new column
raw_df['botometer_result'] = botometer_series
```

In [144]:
```
# extract universal score (botometer score)
raw_df['boto_univ'] = raw_df['botometer_result'].map(lambda s: s['cap'][
'universal'])
raw_df['boto_univ'].describe()
```

Out[144]:
```
count    6032.000000
mean        0.070146
std         0.160049
min         0.001643
25%         0.004304
50%         0.009037
75%         0.038677
max         0.967026
Name: boto_univ, dtype: float64
```

In [69]:
```
# encode bot / non-bot via score of 0.2 threshold
# we chose 0.2 threshold instead of 0.5 as we quickly verify the botomet
er results, and found many of the accounts with less than 0.5 are still
 bots
threshold = 0.2
raw_df['class_boto'] = np.where(raw_df['boto_univ']>threshold, 1, 0)
```

In [70]:
```
# examine number of 'bots' as identified by Botometer
sum(raw_df['class_boto'])
```

Out[70]: 593

In [71]:
```
# save as csv
raw_df.to_csv('immigration_brexit_bitcoin_full_boto.csv')

# save as json
raw_df.to_json('immigration_brexit_bitcoin_full_boto.json')
```

Back to TOC
<a id ='Labelling-Manual'></a>

#### 2.4 - Data Labelling: Manual Verification for Each Account (Until Reach 20 Bots 20 Real Users)

We verified accounts by manually search the username to check if they are bots or not using our best judgement.

We only verified English tweets in this project.

The following rules are used for manual Twitter account verification:
1) Constant retweets of media (especially only retweets no replies)
2) Strong concentration on a specific topic
3) Significantly large number of tweets
4) Significantly large number of replying - not humanly possible speed

We keep manually identifying bots / non-bots account, and only record the ones we are certain about. We keep identifying until reached 22 bots and 22 non-bots - we extended number slightly larger so we don't have too few accounts when some of them get dropped during the data processing process.

In [68]:
```
# load the data
raw_df = pd.read_json('immigration_brexit_bitcoin_full_boto.json')
raw_df.shape
```

Out[68]: (6032, 34)

In [150]:
```
# to verify each user, we only need "screen_name"
raw_df['screen_name'] = raw_df['user'].map(lambda d: d['screen_name'])
```

In [151]:
```
# form a simple dataframe with only screen_name and Botometer score for
 references (so we can manually verify accounts)
# create 'class_verified for verified score'
raw_df_verify = raw_df.loc[:,['screen_name', 'class_verified']]
```

In [152]:
```
# save as csv (so we can manually verify and input results in excel)
raw_df_verify.to_csv('to_verify.csv')
```

In [76]:
```
# we manually verified 40 accounts by searching screen_name, view the us
er's previous tweets, profiles, etc.
# we recorded in the cvs as 1(bot) and 0(non-bot), and only recorded the
 accounts that we feel certain about
# we kept searching until reach 20 bots and 20 users
verify_df =pd. read_csv('boto_verify.csv')
```

In [77]:
```
users_list = verify_df.loc[verify_df['class_verified']==0]
bots_list = verify_df.loc[verify_df['class_verified']==1]
```

In [ ]:
```
display(users_list.shape)
display(bots_list.shape)
```

Back to TOC
<a id ='Data-Collection-Recent'></a>

#### 2.5 - Data Collection - Get 200 (max) Most Recent Tweets from Verified Bot / User

For each of the 6032 accounts we identified, we requested users' most recent 200 tweets using api.user_timeline via tweepy.

In [74]:
```
# read the verified dataframe
raw_df = pd.read_json('immigration_brexit_bitcoin_full_boto.json')
raw_df.shape
```

Out[74]: (6032, 34)

In [113]:
```
#names = raw_df['screen_name'].tolist()
names = raw_df[raw_df['botometer_result'].notnull()]['user'].map(lambda
u: u['screen_name']).tolist()
```

In [114]:
```
len(names)
```

Out[114]:
```
6032
```

In [120]:
```
def get_tweets(names, fName, t_count, verify_df):
    # INPUT:
    # names: list of screen_name
    # fName: file name, .json
    # t_count: maximum number of tweets for each user
    # verify_df: a dataframe with 1) screen name; 2) class_bot; 3) class
_verified; 4) boto_univ
    # OUTPUT:
    # tweets: pd dataframe of all the tweets

    # get tweets
    error_count = 0

    with open(fName, 'w') as f:
        tweetCount = 0
        for name in names:
            try:
                tweets = api.user_timeline(screen_name=name, count=t_cou
nt, tweet_mode='extended')
                for tweet in tweets:
                    f.write(jsonpickle.encode(tweet._json, unpicklable=F
alse) + '\n')
                print("Downloaded {} tweets.".format(len(tweets)))
                tweetCount += len(tweets)
                time.sleep(2)
            except Exception as e:
                # Just exit if any error
                error_count += 1
                print("some error : " + str(e))
                if error_count >= 100:
                    print("too many errors ....break.")
                    break
    print ("Downloaded {0} tweets, Saved to {1}".format(tweetCount, fNam
e))
```

In [ ]:
```
# get max 200 tweets for each user
get_tweets(names=names, fName='tweets.json', t_count=200, verify_df=raw_
df) #the fName and corresponding data will be updated later
```

In [ ]:
```
# read the data
tweets_df = pd.read_json('tweets.json', lines=True)
```

In [38]:
```
tweets_df.columns.values
```

Out[38]:
```
array(['contributors', 'coordinates', 'created_at', 'display_text_rang
e',
       'entities', 'extended_entities', 'favorite_count', 'favorited',
       'full_text', 'geo', 'id', 'id_str', 'in_reply_to_screen_name',
       'in_reply_to_status_id', 'in_reply_to_status_id_str',
       'in_reply_to_user_id', 'in_reply_to_user_id_str',
       'is_quote_status', 'lang', 'place', 'possibly_sensitive',
       'quoted_status', 'quoted_status_id', 'quoted_status_id_str',
       'quoted_status_permalink', 'retweet_count', 'retweeted',
       'retweeted_status', 'source', 'truncated', 'user',
       'withheld_copyright', 'withheld_in_countries', 'withheld_scop
e'],
      dtype=object)
```

Back to TOC
<a id ='Description-of-Raw-Data'></a>

#### 2.6 - Description of Raw Data (Tweets)

Among the data we collected as json files from the tweepy api.search, the data set contains objects such as ‘user’, which includes comprehensive information of user accounts. Additionally, detailed information about each individual tweet was also collected.

The following describes some of the fields of the raw data collected:

> *followers* : number of user’s followers
> *friends* : information about relationship/interaction with other users
> *following* : users following the specified user/account
> *retweet count* : number of retweets
> *verify_credentials* : verifies whether the user credentials are valid
> *screen_name* : screen name of account user
> *retweets* : number of retweets of a given tweet
> *user_description* : description set by user in profile
> *profile_background_url* : user’s background image for profile
> *profile_image_url* : user’s profile image
> *geo_enabled* : location of tweet if user source has geo-location enabled

Botometer’s response object returned bot-scores in various different categories. This included categories such as the Complete Automation Probability, which determines how likely the account is a bot. The bot-scores, on a scale, determines if a given account is closer to a bot or a real user. Then. from the json data we gathered through the tweepy api.search, we extracted user account id to retrieve their corresponding Botometer scores.

Back to TOC
<a id ='Exploratory-Data-Analysis'></a>

# 3 - Exploratory Data Analysis

In this section, we performed data cleansing, exploratory data analysis, feature engineering (aggregate tweet level features to account level) and standardize our data to prepare for the modelling.

Back to TOC
<a id ='Data-Wrangling'></a>

### 3.1 - Data Wrangling & Cleansing

First, we parsed features, only inlude the features with value, and drop features with mostly null value.

In [2]:
```
# read the dataset
tweets_df = pd.read_json('tweets.json', lines=True)
```

In [81]:
```
# first we want to reduce columns by dropping the features that miss dat
a more than 50% of the time
threshold = len(tweets_df.columns.values)*0.5
tweets_df = tweets_df.dropna(thresh = threshold, axis='columns')
```

In [82]:
```
# take a look at the shape
tweets_df.shape
```

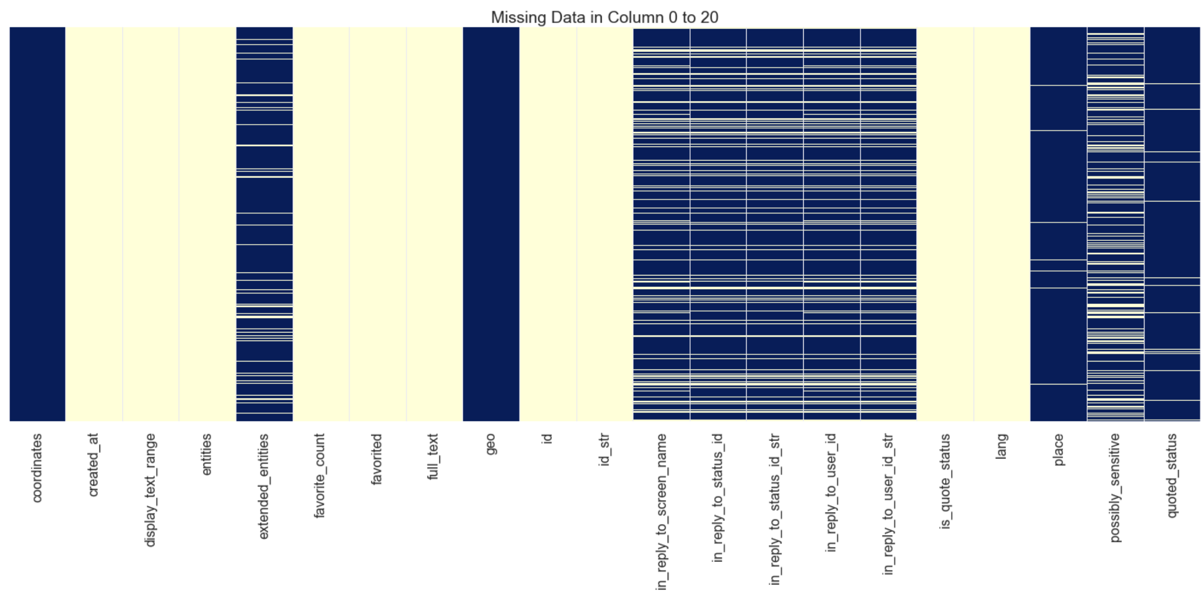
Out[82]:
```
(1172951, 31)
```

In [83]:
```
# explode 'entities', 'user'
# although it would be interesting to see 'retweeted_status', it might b
e a bit too complicated
# especially when the # of reweets of the retweeted post is availabel di
rectly ('retweet_count')
# it might be more efficient just to add a new column showing if a tweet
 contains retweet
def explode(df):
    dicts = ['user', 'entities']
    for d in dicts:
        keys = list(df.iloc[0]['user'].keys())
        for key in keys:
            df[str(d) + '_' + key] = df[d].map(lambda x: x[key] if key i
n x and x[key] else None)
    return df
```

In [ ]:
```
# parse
tweets_df = explode(tweets_df)
```

In [85]:
```
# heatmap to visualize the missing data in different columns
sns.set(style="darkgrid")
sns.set_context("poster")

def get_heatmap(df, imgName='NaN_heatmap.png'):
    #This function gives heatmap of all NaN values or only zero
    plt.figure(figsize=(20,10))
    sns.heatmap(df.isnull(), yticklabels=False, cbar=False,cmap="YlGnBu"
).set_title('Missing Data in Column 0 to 20')

    plt.tight_layout()

    # save image for report, need to run cell
    plt.savefig(imgName)

    return plt.show()
```

In [86]:
```
#plotting first null values
get_heatmap(tweets_df.ix[:,0:21], imgName='NaN_heatmap_col0_20.png')
```




In [93]:
```
# drop empty columns again (after exploding 'user' and 'entities')
threshold = len(tweets_df.columns.values)*0.5
tweets_df = tweets_df.dropna(thresh = threshold, axis='columns')
```

In [94]:
```
# take a look at the columns left
display(len(tweets_df.columns.values))
display(tweets_df.columns.values)
```

```
65

array(['coordinates', 'created_at', 'display_text_range', 'entities',
       'extended_entities', 'favorite_count', 'favorited', 'full_text',
       'geo', 'id', 'id_str', 'in_reply_to_screen_name',
       'in_reply_to_status_id', 'in_reply_to_status_id_str',
       'in_reply_to_user_id', 'in_reply_to_user_id_str',
       'is_quote_status', 'lang', 'place', 'possibly_sensitive',
       'quoted_status', 'quoted_status_id', 'quoted_status_id_str',
       'quoted_status_permalink', 'retweet_count', 'retweeted',
       'retweeted_status', 'source', 'truncated', 'user',
       'withheld_in_countries', 'user_created_at', 'user_default_profil
e',
       'user_default_profile_image', 'user_description', 'user_entitie
s',
       'user_favourites_count', 'user_followers_count',
       'user_friends_count', 'user_geo_enabled',
       'user_has_extended_profile', 'user_id', 'user_id_str',
       'user_is_translation_enabled', 'user_lang', 'user_listed_count',
       'user_location', 'user_name', 'user_profile_background_color',
       'user_profile_background_image_url',
       'user_profile_background_image_url_https',
       'user_profile_background_tile', 'user_profile_banner_url',
       'user_profile_image_url', 'user_profile_image_url_https',
       'user_profile_link_color', 'user_profile_sidebar_border_color',
       'user_profile_sidebar_fill_color', 'user_profile_text_color',
       'user_profile_use_background_image', 'user_screen_name',
       'user_statuses_count', 'user_translator_type', 'user_url',
       'user_verified'], dtype=object)
```

In [99]:
```
# we only interested in english tweets
tweets_df_en = tweets_df.loc[tweets_df['lang']=='en']
tweets_df_en.shape
```

Out[99]:
```
(1042177, 65)
```

In [100]:
```
# duplicated / no longer userful columns
col_duplicate = ['entities','user', 'lang', 'user_lang', 'user_id', 'use
r_id_str', 'id_str']
# we dropped 'lang' as we only use english accounts for our dataset
# 'entities' and 'user' have already been parsed

# columns that we are obviously not interested
col_not_interested = ['user_entities']
# retweeted_status is the tweet object of the retweet - perhaps
```

In [102]:
```
# drop duplicated columns and columns that we are not interested
tweets_df_en = tweets_df_en.drop(columns= (col_duplicate + col_not_inter
ested))
```

In [103]:
```
# take a look at shape
tweets_df_en.shape
```

Out[103]: (1042177, 57)

In [106]:
```
# save as json
tweets_df_en.to_json('tweets_clean.json')
```

Back to TOC
<a id ='Feature-Engineering'></a>

#### 2.2 - Feature Engineering

Next, we want to aggregate tweet features to the accounts.

In [447]:
```
# read previous json file
tweets_df = pd.read_json('tweets_clean.json')
```

Back to TOC
<a id ='FE-Tweet-Features'></a>

##### *2.2.1 - Feature Engineering - Tweet Features*

We want to create the following features to prepare for NLP feature engineering / analysis:
1) text_rt: text of the retweet
2) text_tweet: text of the tweet (when there is no retweet)
3) encode tweet features

In [448]:
```
# although using tweet_mode='extended', we are still not getting the ful
l text
# therefore, we tried to get full_text from retweeted_status
tweets_df['text_rt'] = tweets_df['retweeted_status'].map(lambda x: x['fu
ll_text'] if x and (not isinstance(x, float)) and ('full_text' in x) els
e None)
tweets_df['text_tweet'] = tweets_df['full_text'].where(tweets_df['text_r
t'].map(lambda x: x is None), None)
tweets_df[['text_tweet', 'text_rt']].head(5)
```

Out[448]:

| | text_tweet | text_rt |
|---|---|---|
| **0** | NEW Coinpot Multiplier : How to Win more ( Bes... | None |
| **1** | How To Buy Things With Bitcoin Coinpot Litecoi... | None |
| **10** | Filling the Brita all the way to the top count... | None |
| **100** | You can collect other bitcoin faucets and incr... | None |
| **1000** | None | Michael Gove on 30 November 2018 and the truth... |

In [449]:
```
# take a look at retweets
tweets_df[['text_tweet', 'text_rt']][tweets_df['text_rt'].map(lambda s:
s is not None)].head()
```

Out[449]:

| | text_tweet | text_rt |
|---|---|---|
| **1000** | None | Michael Gove on 30 November 2018 and the truth... |
| **10000** | None | This is me being held by my Dad in 1955. I hav... |
| **100002** | None | Nicola Sturgeon uses the threat of a referendu... |
| **100003** | None | This horse was spotted walking around a Tesco ... |
| **100004** | None | When did worrying about immigration's impact o... |

In [450]:
```
# encode tweet features

# 1 = favorited - True; 0 = favorited - False
tweets_df['favorited'] = tweets_df['favorited'].map(lambda x: 0 if x==Fa
lse else 1)

# 1 = retweeted-true; 0 = retweeted-false
tweets_df['retweeted'] = tweets_df['retweeted'].map(lambda x: 1 if x==Tr
ue else 0)

# 1 = tweet includes retweet; 0 = tweet does not include retweet
tweets_df['retweeted_status'] = tweets_df['retweeted_status'].map(lambda
 x: 0 if x==None else 1)

# 0 = none or information not available
tweets_df['user_listed_count'] = tweets_df['user_listed_count'].map(lamb
da x: x if x>0 else 0)

# replace nan with 0 for the following features (as for these features,
 missing values usually means 0)
for f in ['user_favourites_count', 'user_followers_count', 'user_friends
_count']:
    tweets_df[f] = tweets_df[f].replace(np.nan, 0, regex=True)
```

In [451]:
```
tweets_df.shape
```

Out[451]:
```
(1042177, 59)
```

Back to TOC
<a id ='Data-Cleansing'></a>

### *3.2.2 - Feature Engineering - User Features*

As we suspected bots tweet more frequently / have different tweeting pattern from real users, we want to engineer the following features in order to examine them in the model:
1) length of user description
2) tweet frequencies (the mean, std, min, and max time between tweets for each account)
3) account age (seconds from the account creation time to the latest tweet time)

In [452]:
```
# extract
tweets_df['screen_name'] = tweets_df['user_screen_name']
```

In [453]:
```
# account feature engineering
# create an intermedium df with all account-related data from tweets

users_description_len_df = tweets_df.drop_duplicates(subset=['screen_nam
e'])
users_description_len_df['user_description_len'] = users_description_len
_df['user_description'].map(lambda x: len(x) if x!=None else 0)
```

In [454]:
```
# account feature engineering
# get tweets interval stats (in seconds)

def create_tweet_time_stats(created_at_series):
    times = created_at_series['created_at'].sort_values().diff().dt.tota
l_seconds()[1:]
    cols = ['tweet_time_mean', 'tweet_time_std', 'tweet_time_min', 'twee
t_time_max']
    return pd.Series([times.mean(), times.std(), times.min(), times.max
()], index=cols)

tweet_time_stats_df = tweets_df[['screen_name', 'created_at']].groupby(
'screen_name').apply(create_tweet_time_stats).reset_index()
tweet_time_stats_df.head()
```

Out[454]:

| | screen_name | tweet_time_mean | tweet_time_std | tweet_time_min | tweet_time_max |
|---|---|---|---|---|---|
| 0 | 0604Arb1320 | 1103.088481 | 11493.389030 | 0.0 | 170593.0 |
| 1 | 07_smith | 98062.614973 | 153488.369492 | 6.0 | 975586.0 |
| 2 | 0AngelHeart | 1354.688172 | 5038.485992 | 6.0 | 34269.0 |
| 3 | 0ttaM | 14095.382653 | 29149.325281 | 3.0 | 134152.0 |
| 4 | 100Climbs | 2862.817204 | 6481.823049 | 6.0 | 40810.0 |

In [455]:
```
# account feature engineering
# get account age (in seconds)

reference_date = tweets_df['created_at'].max()
user_account_age_df = tweets_df[['screen_name', 'user_created_at']].grou
pby('screen_name').min().reset_index()
user_account_age_df['account_age'] = user_account_age_df['user_created_a
t'].map(lambda d: (reference_date - pd.to_datetime(d)).total_seconds())
del user_account_age_df['user_created_at']
user_account_age_df.head()
```

Out[455]:

| | screen_name | account_age |
|---|---|---|
| 0 | 0604Arb1320 | 190794240.0 |
| 1 | 07_smith | 222929527.0 |
| 2 | 0AngelHeart | 238521195.0 |
| 3 | 0ttaM | 142325785.0 |
| 4 | 100Climbs | 279804439.0 |

In [456]:
```
# account feature engineering
# create a new dataframe with engineered features that are associated wi
th each user
users_df = pd.DataFrame(tweets_df['screen_name']).drop_duplicates(subset
='screen_name')
users_df = pd.merge(users_df, tweet_time_stats_df, left_on='screen_name'
, right_on='screen_name')
users_df = pd.merge(users_df, users_description_len_df[['screen_name',
'user_description_len']], left_on='screen_name', right_on='screen_name')
users_df = pd.merge(users_df, user_account_age_df, left_on='screen_name'
, right_on='screen_name')
users_df.head(5)
```

Out[456]:

| | screen_name | tweet_time_mean | tweet_time_std | tweet_time_min | tweet_time_max | us |
|---|---|---|---|---|---|---|
| 0 | ICVeo | 1033.132275 | 1.937518e+03 | 0.0 | 8375.0 | 77 |
| 1 | ianw2000uk | 254.486146 | 1.616182e+03 | 0.0 | 26368.0 | 10 |
| 2 | hmsglasgow | 107.966921 | 1.068803e+03 | 0.0 | 16601.0 | 12 |
| 3 | MarkHW1 | 607784.043011 | 2.526945e+06 | 3.0 | 28007804.0 | 0 |
| 4 | RabHutchison | 1122.005291 | 3.159343e+03 | 4.0 | 21281.0 | 11 |

In [457]:
```
# read the dataset with botometer score
boto_df = pd.read_json('immigration_brexit_bitcoin_full_boto.json')
boto_df['screen_name'] = boto_df['user'].map(lambda u: u['screen_name'])
```

In [458]:
```
# add botometer back
boto_class_df = boto_df[['class_boto','screen_name']].drop_duplicates(su
bset='screen_name')
tweets_df = pd.merge(tweets_df, boto_class_df, left_on='screen_name', ri
ght_on='screen_name')
tweets_df.columns.values
```

Out[458]:
```
array(['coordinates', 'created_at', 'display_text_range',
       'extended_entities', 'favorite_count', 'favorited', 'full_text',
       'geo', 'id', 'in_reply_to_screen_name', 'in_reply_to_status_id',
       'in_reply_to_status_id_str', 'in_reply_to_user_id',
       'in_reply_to_user_id_str', 'is_quote_status', 'place',
       'possibly_sensitive', 'quoted_status', 'quoted_status_id',
       'quoted_status_id_str', 'quoted_status_permalink', 'retweet_coun
t',
       'retweeted', 'retweeted_status', 'source', 'truncated',
       'withheld_in_countries', 'user_created_at', 'user_default_profil
e',
       'user_default_profile_image', 'user_description',
       'user_favourites_count', 'user_followers_count',
       'user_friends_count', 'user_geo_enabled',
       'user_has_extended_profile', 'user_is_translation_enabled',
       'user_listed_count', 'user_location', 'user_name',
       'user_profile_background_color',
       'user_profile_background_image_url',
       'user_profile_background_image_url_https',
       'user_profile_background_tile', 'user_profile_banner_url',
       'user_profile_image_url', 'user_profile_image_url_https',
       'user_profile_link_color', 'user_profile_sidebar_border_color',
       'user_profile_sidebar_fill_color', 'user_profile_text_color',
       'user_profile_use_background_image', 'user_screen_name',
       'user_statuses_count', 'user_translator_type', 'user_url',
       'user_verified', 'text_rt', 'text_tweet', 'screen_name',
       'class_boto'], dtype=object)
```

In [459]:
```
# merge the account information back to the dataset
tweets_df = pd.merge(tweets_df, users_df, left_on='screen_name', right_o
n='screen_name')
tweets_df.columns.values
```

Out[459]:
```
array(['coordinates', 'created_at', 'display_text_range',
       'extended_entities', 'favorite_count', 'favorited', 'full_text',
       'geo', 'id', 'in_reply_to_screen_name', 'in_reply_to_status_id',
       'in_reply_to_status_id_str', 'in_reply_to_user_id',
       'in_reply_to_user_id_str', 'is_quote_status', 'place',
       'possibly_sensitive', 'quoted_status', 'quoted_status_id',
       'quoted_status_id_str', 'quoted_status_permalink', 'retweet_coun
t',
       'retweeted', 'retweeted_status', 'source', 'truncated',
       'withheld_in_countries', 'user_created_at', 'user_default_profil
e',
       'user_default_profile_image', 'user_description',
       'user_favourites_count', 'user_followers_count',
       'user_friends_count', 'user_geo_enabled',
       'user_has_extended_profile', 'user_is_translation_enabled',
       'user_listed_count', 'user_location', 'user_name',
       'user_profile_background_color',
       'user_profile_background_image_url',
       'user_profile_background_image_url_https',
       'user_profile_background_tile', 'user_profile_banner_url',
       'user_profile_image_url', 'user_profile_image_url_https',
       'user_profile_link_color', 'user_profile_sidebar_border_color',
       'user_profile_sidebar_fill_color', 'user_profile_text_color',
       'user_profile_use_background_image', 'user_screen_name',
       'user_statuses_count', 'user_translator_type', 'user_url',
       'user_verified', 'text_rt', 'text_tweet', 'screen_name',
       'class_boto', 'tweet_time_mean', 'tweet_time_std',
       'tweet_time_min', 'tweet_time_max', 'user_description_len',
       'account_age'], dtype=object)
```

Back to TOC
<a id ='Data-Cleansing'></a>

### *3.2.3 - Feature Engineering - Finalize and Clean Up Data*

We want to cealnup the data by dropping the columns that are no longer interesting / useful. For isntance, features such as 'created_at' has already been captured in account_age, 'user_orofile_sidebar_fill_color' might be interesting to see if that correlates with user types but we chose not to proceed in this project.

In [460]:
```
# delete columns that no longer useful
col_del = ['display_text_range', 'in_reply_to_status_id_str', 'in_reply_
to_user_id_str','in_reply_to_status_id',
            'in_reply_to_user_id', 'is_quote_status', 'quoted_status', 'q
uoted_status_id', 'quoted_status_id_str',
           'quoted_status_permalink', 'user_url', 'user_translator_type',
 'user_default_profile_image',
           'user_default_profile', 'user_geo_enabled', 'user_has_extended
_profile', 'user_profile_background_tile',
           'user_profile_image_url', 'user_profile_image_url_https', 'ful
l_text', 'created_at',
           'user_created_at', 'user_profile_background_image_url', 'user_
profile_background_image_url_https',
           'user_profile_banner_url', 'user_profile_link_color', 'user_pr
ofile_sidebar_border_color',
            'possibly_sensitive', 'user_profile_sidebar_fill_color', 'use
r_profile_text_color', 'user_screen_name',
           'user_profile_background_color', 'extended_entities', 'in_repl
y_to_screen_name', 'truncated', 'user_location',
           'user_name', 'source', 'geo', 'place', 'withheld_in_countries'
, 'coordinates', 'user_is_translation_enabled',
            'user_profile_use_background_image']

tweets_df = tweets_df.drop(columns=col_del, axis=1)
```

In [461]:
```
tweets_df.dtypes
```

Out[461]:
```
favorite_count              int64
favorited                   int64
id                          int64
retweet_count               int64
retweeted                   int64
retweeted_status            int64
user_description           object
user_favourites_count     float64
user_followers_count      float64
user_friends_count        float64
user_listed_count         float64
user_statuses_count         int64
user_verified             float64
text_rt                    object
text_tweet                 object
screen_name                object
class_boto                  int64
tweet_time_mean           float64
tweet_time_std            float64
tweet_time_min            float64
tweet_time_max            float64
user_description_len        int64
account_age               float64
dtype: object
```

In [462]:
```
# check user_verified
display(tweets_df.shape)
display(tweets_df[tweets_df['user_verified'].isnull()].shape)
```

```
(1042177, 23)

(1030080, 23)
```

In [463]:
```
# as it is mostly None, we decided to delete this column
del tweets_df['user_verified']
```

In [464]:
```
display(tweets_df.columns.values)
display(tweets_df.shape)
```

```
array(['favorite_count', 'favorited', 'id', 'retweet_count', 'retweete
d',
       'retweeted_status', 'user_description', 'user_favourites_count',
       'user_followers_count', 'user_friends_count', 'user_listed_coun
t',
       'user_statuses_count', 'text_rt', 'text_tweet', 'screen_name',
       'class_boto', 'tweet_time_mean', 'tweet_time_std',
       'tweet_time_min', 'tweet_time_max', 'user_description_len',
       'account_age'], dtype=object)

(1042177, 22)
```

In [465]:
```
tweets_df.describe()
```

Out[465]:

| | favorite_count | favorited | id | retweet_count | retweeted | retweeted_sta |
|---|---|---|---|---|---|---|
| **count** | 1.042177e+06 | 1042177.0 | 1.042177e+06 | 1.042177e+06 | 1042177.0 | 1.042177e+06 |
| **mean** | 1.369078e+00 | 0.0 | 1.065372e+18 | 1.161262e+03 | 0.0 | 6.618329e-01 |
| **std** | 4.821426e+01 | 0.0 | 3.293977e+16 | 1.102336e+04 | 0.0 | 4.730860e-01 |
| **min** | 0.000000e+00 | 0.0 | 1.240361e+09 | 0.000000e+00 | 0.0 | 0.000000e+00 |
| **25%** | 0.000000e+00 | 0.0 | 1.069875e+18 | 0.000000e+00 | 0.0 | 0.000000e+00 |
| **50%** | 0.000000e+00 | 0.0 | 1.071769e+18 | 2.100000e+01 | 0.0 | 1.000000e+00 |
| **75%** | 0.000000e+00 | 0.0 | 1.072138e+18 | 3.770000e+02 | 0.0 | 1.000000e+00 |
| **max** | 1.890900e+04 | 0.0 | 1.072320e+18 | 3.560981e+06 | 0.0 | 1.000000e+00 |

In [466]:
```
# create list of columns names for different categories and see if we ha
ve missed anything
col_response = ['class_boto']
col_pred_text = list(tweets_df.select_dtypes(['object']).columns.values)
col_id = ['id']
col_pred_numerical = list(tweets_df.select_dtypes(['float64', 'int64']).
drop(columns=['class_boto', 'id']).columns.values)
```

In [467]:
```
# take a look at numerical features
display(col_pred_numerical)
```

```
['favorite_count',
 'favorited',
 'retweet_count',
 'retweeted',
 'retweeted_status',
 'user_favourites_count',
 'user_followers_count',
 'user_friends_count',
 'user_listed_count',
 'user_statuses_count',
 'tweet_time_mean',
 'tweet_time_std',
 'tweet_time_min',
 'tweet_time_max',
 'user_description_len',
 'account_age']
```

In [468]:
```
# take a look at text features
display(col_pred_text)
```

```
['user_description', 'text_rt', 'text_tweet', 'screen_name']
```

In [469]:
```
# delete numerical columns that have mean or std equals 0 (which implies
 same values for the columns)
col_name_del = []
for col in col_pred_numerical:
    if tweets_df[col].mean() == 0 or tweets_df[col].std() == 0:
        del tweets_df[col]
        col_name_del.append(col)
        col_pred_numerical.remove(col)
display(tweets_df.shape)
print ('{} are deleted as they only have one values across all the row
s.'.format(str(col_name_del)))
```

```
(1042177, 20)

['favorited', 'retweeted'] are deleted as they only have one values acr
oss all the rows.
```

In [470]:
```
# before saving the file, we want to delete any rows with NaN values fro
m the new columns
col_w_nan = tweets_df.columns[tweets_df.isna().any()].tolist()
col_w_nan
```

Out[470]:
```
['user_description',
 'text_rt',
 'text_tweet',
 'tweet_time_mean',
 'tweet_time_std',
 'tweet_time_min',
 'tweet_time_max']
```

In [471]:
```
# while it is okay to have NaN in texts, we want to delete the rows with
 NaN Values in the tweet_time related columns
tweets_df = tweets_df.dropna(axis=0, subset=['tweet_time_mean', 'tweet_t
ime_std', 'tweet_time_min', 'tweet_time_max'])
display(tweets_df.shape)
display(tweets_df.isna().any())
```

```
(1042095, 20)

favorite_count           False
id                       False
retweet_count            False
retweeted_status         False
user_description          True
user_favourites_count    False
user_followers_count     False
user_friends_count       False
user_listed_count        False
user_statuses_count      False
text_rt                   True
text_tweet                True
screen_name              False
class_boto               False
tweet_time_mean          False
tweet_time_std           False
tweet_time_min           False
tweet_time_max           False
user_description_len     False
account_age              False
dtype: bool
```

In [472]:
```
# great! let's save as json
users_df.to_json('users.json')
tweets_df.to_json('tweets_clean_final.json')
```

Back to TOC
<a id ='NLP-Features'></a>

### 3.3 - Advanced Feature Engineering - NLP Features

After cleaning up the file and did some feature engineering, we tried to create some NLP features that might be interesting to our project, such as the length of tweets, the average word length an account use in the tweets.

In [513]:
```
# read the data
tweets_df = pd.read_json('tweets_clean_final.json')
users_df = pd.read_json('users.json')
```

In [514]:
```
col_nlp_text = ['tweet_len_mean', 'tweet_len_std', 'tweet_word_mean', 't
weet_word_std',
                'retweet_len_mean', 'retweet_len_std', 'retweet_word_mea
n', 'retweet_word_std']

with open('col_nlp_text.txt', 'w') as fp:
        ls_str = ",".join(col_nlp_text)
        fp.write(ls_str)
```

In [515]:
```
# function to get tweet length
def get_tweet_lens(tweet_series):
    return tweet_series.dropna().map(lambda s: len(s))
```

In [516]:
```
# function to get length of each word. filtering out hashtags, @, and li
nks
def get_tweet_word_lens(tweet_series):
    tweets = tweet_series.dropna().values.tolist()
    words = [w for s in [t.split() for t in tweets] for w in s]
    filtered_words = filter(lambda w: not (w.startswith('@') or w.starts
with('#') or w.startswith('http')), words)
    word_len = np.array([len(w) for w in filtered_words])
    return word_len
```

In [517]:
```
# function to create feature
def tweet_text_features(df):
    cols = col_nlp_text
    tweet_lens = get_tweet_lens(df['text_tweet'])
    tweet_word_lens = get_tweet_word_lens(df['text_tweet'])
    retweet_lens = get_tweet_lens(df['text_rt'])
    retweet_word_lens = get_tweet_word_lens(df['text_rt'])

    return pd.Series((tweet_lens.mean(), tweet_lens.std(),
                      tweet_word_lens.mean(), tweet_word_lens.std(),
                      retweet_lens.mean(), retweet_lens.std(),
                      retweet_word_lens.mean(), retweet_word_lens.std()),
index=cols)
```

In [518]:
```
# get text features
text_df = tweets_df.groupby("screen_name").apply(tweet_text_features).re
set_index()
```

In [519]:
```
# merge text features with tweets_df
tweets_df = pd.merge(tweets_df, text_df, left_on='screen_name', right_on
='screen_name')
```

In [520]:
```
display(tweets_df.shape)
display(tweets_df.columns.values)
```

```
(1042095, 28)

array(['favorite_count', 'id', 'retweet_count', 'retweeted_status',
       'user_description', 'user_favourites_count',
       'user_followers_count', 'user_friends_count', 'user_listed_coun
t',
       'user_statuses_count', 'text_rt', 'text_tweet', 'screen_name',
       'class_boto', 'tweet_time_mean', 'tweet_time_std',
       'tweet_time_min', 'tweet_time_max', 'user_description_len',
       'account_age', 'tweet_len_mean', 'tweet_len_std',
       'tweet_word_mean', 'tweet_word_std', 'retweet_len_mean',
       'retweet_len_std', 'retweet_word_mean', 'retweet_word_std'],
      dtype=object)
```

In [521]:
```
users_df.shape
```

Out[521]:
```
(4226, 28)
```

In [524]:
```
# merge text features with uers_df
users_df = pd.merge(users_df, text_df, left_on='screen_name', right_on=
'screen_name')
```

In [525]:
```
# clean up users_df a bit and join boto scores
users_df = pd.merge(users_df, tweets_df[['class_boto', 'screen_name']],
left_on='screen_name', right_on='screen_name')
users_df = users_df.drop_duplicates(subset='screen_name')
```

In [528]:
```
# great! let's save as json-again
users_df.to_json('users_final.json')
tweets_df.to_json('tweets_clean_final2.json')
```

Back to TOC
<a id ='Important-Features'></a>

### 2.4 - Important Features

Before we conclude our data processing, we want to explore if there are any tweets features that we haven't captured but might be interesting for our analysis.

We also want to explore the relationship among account-level features we have selected / engineered, and see if any of them are particularly interesting in identifying bots / nonbots.

In [377]:
```
# read the data
tweets_df = pd.read_json('tweets_clean_final2.json')
```

In [529]:
```
# separte bots and non-bots tweets for easy plotting
tweets_0 = tweets_df.loc[tweets_df['class_boto']==0]
tweets_1 = tweets_df.loc[tweets_df['class_boto']==1]
```

In [534]:
```
# read the user dataframe
users_df = pd.read_json('users_final.json')
```

In [535]:
```
# separte bots and non-bots accounts for easy plotting
users_0 = users_df.loc[users_df['class_boto']==0]
users_1 = users_df.loc[users_df['class_boto']==1]
```

Let's examine the data. We removed "screen_name" from showing in the dataframe for privacy.

In [536]:
```
# examine the tweets data
# we drop screen_name for privacy
tweets_df.drop(columns=['screen_name']).head(5)
```

Out[536]:

| | favorite_count | id | retweet_count | retweeted_status | user_descripti |
|---|---|---|---|---|---|
| **0** | 0 | 1072242970626326532 | 0 | 0 | Tweet to LOVE not to HATE! Simply sharing the... |
| **1** | 0 | 1072242968810131457 | 0 | 0 | Tweet to LOVE not to HATE! Simply sharing the... |
| **2** | 0 | 1071853531710324736 | 0 | 0 | Tweet to LOVE not to HATE! Simply sharing the... |
| **3** | 0 | 1071462035878236165 | 0 | 0 | Tweet to LOVE not to HATE! Simply sharing the... |
| **4** | 0 | 1071462034150174727 | 0 | 0 | Tweet to LOVE not to HATE! Simply sharing the... |

In [537]:
```
# examine the users data
# we drop screen_name for privacy
users_df.drop(columns=['screen_name']).head(5)
```

Out[537]:

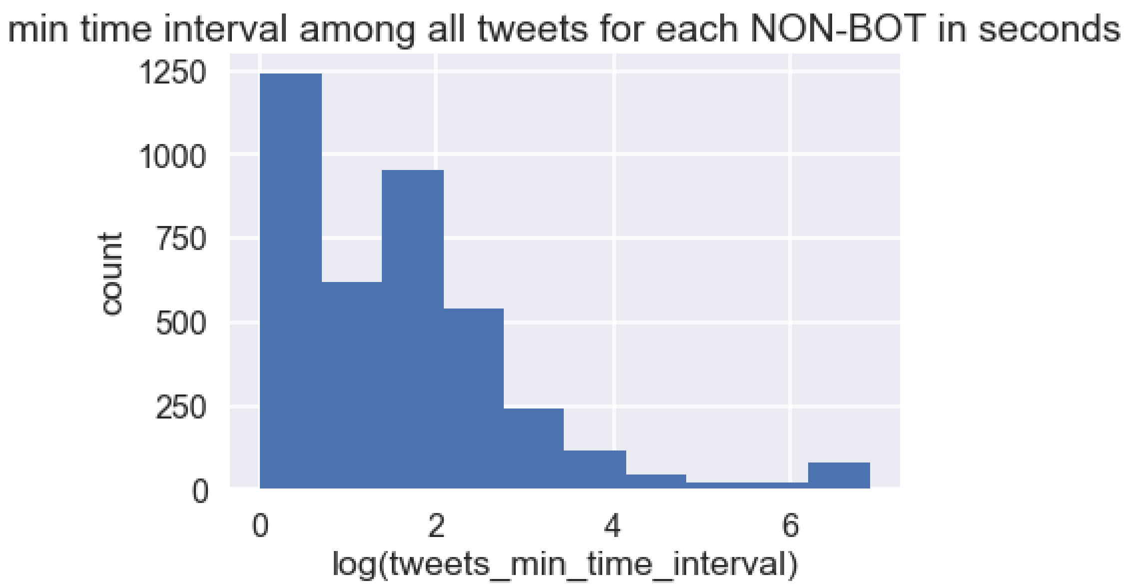
| | favorite_count | id | retweet_count | retweeted_status | user_d |
|---|---|---|---|---|---|
| **0** | 0 | 1072242970626326532 | 0 | 0 | Tweet t not to H Simply the... |
| **1000085** | 0 | 1072250182291648512 | 0 | 0 | Lifelong fan, see good & go... |
| **1000268** | 1 | 1072182528595779585 | 0 | 0 | Britain's selling newspa |
| **1000455** | 0 | 1072250902470422528 | 0 | 0 | Paddlin canoe. |
| **1000603** | 0 | 1071920364790448128 | 97 | 1 | A sover is \nnei depend |

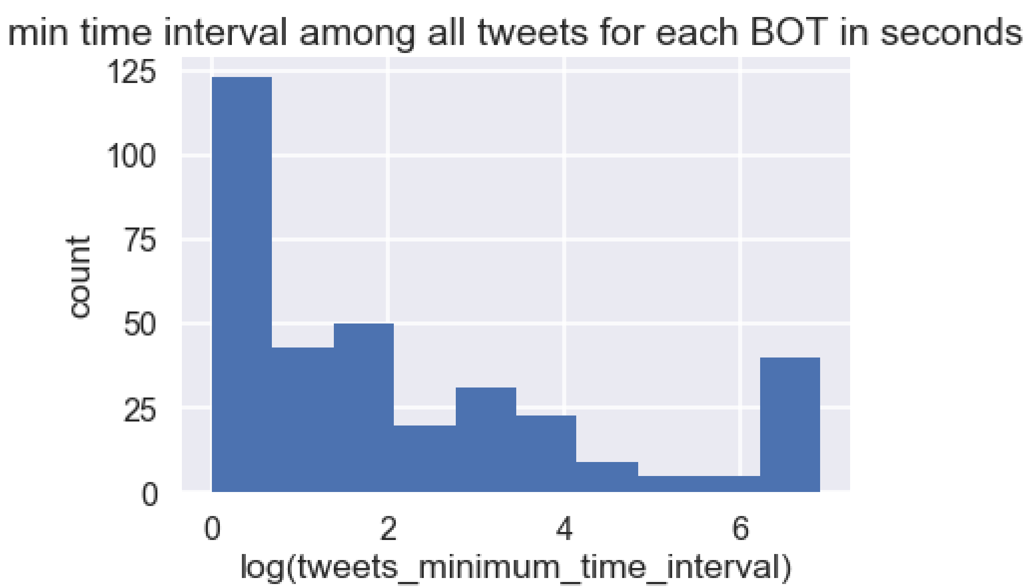
In [488]:
```
# scatter plot
def scatterplot (col_b1, col_b2, col_r1, col_r2, col1, col2):
    plt.scatter(col_b1, col_b2, s=5, color='salmon', label='bot', alpha=
0.75)
    plt.scatter(col_r1, col_r2, s=5, color='royalblue', label='non-bot',
 alpha=0.75)
    plt.xlabel(str(col1))
    plt.ylabel(str(col2))
    #plt.xlim(xlimit)
    #plt.ylim(ylimit)
    plt.legend(loc='best', bbox_to_anchor=(0.85, 0., 0.5, 0.5))
    title = str(col1) + ' vs ' + str(col2)
    plt.title(title)
    plt.savefig(str(title)+'.png')
```

In [489]:
```
# scatter plot2
def scatterplot2 (col_b1, col_b2, col_r1, col_r2, col1, col2):
    plt.scatter(col_b1, col_b2, s=3, color='salmon', label='bot', alpha=
0.0025)
    plt.scatter(col_r1, col_r2, s=3, color='royalblue', label='non-bot',
 alpha=0.0025)
    plt.xlabel(str(col1))
    plt.ylabel(str(col2))
    #plt.xlim(xlimit)
    #plt.ylim(ylimit)
    plt.legend(loc='best', bbox_to_anchor=(0.85, 0., 0.5, 0.5))
    title = str(col1) + ' vs ' + str(col2)
    plt.title(title)
    plt.savefig(str(title)+'.png')
```

In [490]:
```
# histogram
def hist_plot(col, xlabel, ylabel, title):
    #values = col.values[~np.isnan(col.values)]
    plt.hist(col)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    #plt.xlim(xlimit)
    plt.title(title)
    plt.savefig(str(title)+'.png')
    return None
```

In [491]:
```
# quick plots
plt.figure(figsize=(6,4))
hist_plot(np.log(users_0['tweet_time_min'].values.clip(1, 1000)), 'log(t
weets_min_time_interval)','count', 'min time interval among all tweets f
or each NON-BOT in seconds')
plt.figure(figsize=(6,4))
hist_plot(np.log(users_1['tweet_time_min'].values.clip(1, 1000)), 'log(t
weets_minimum_time_interval)','count', 'min time interval among all twee
ts for each BOT in seconds')
```

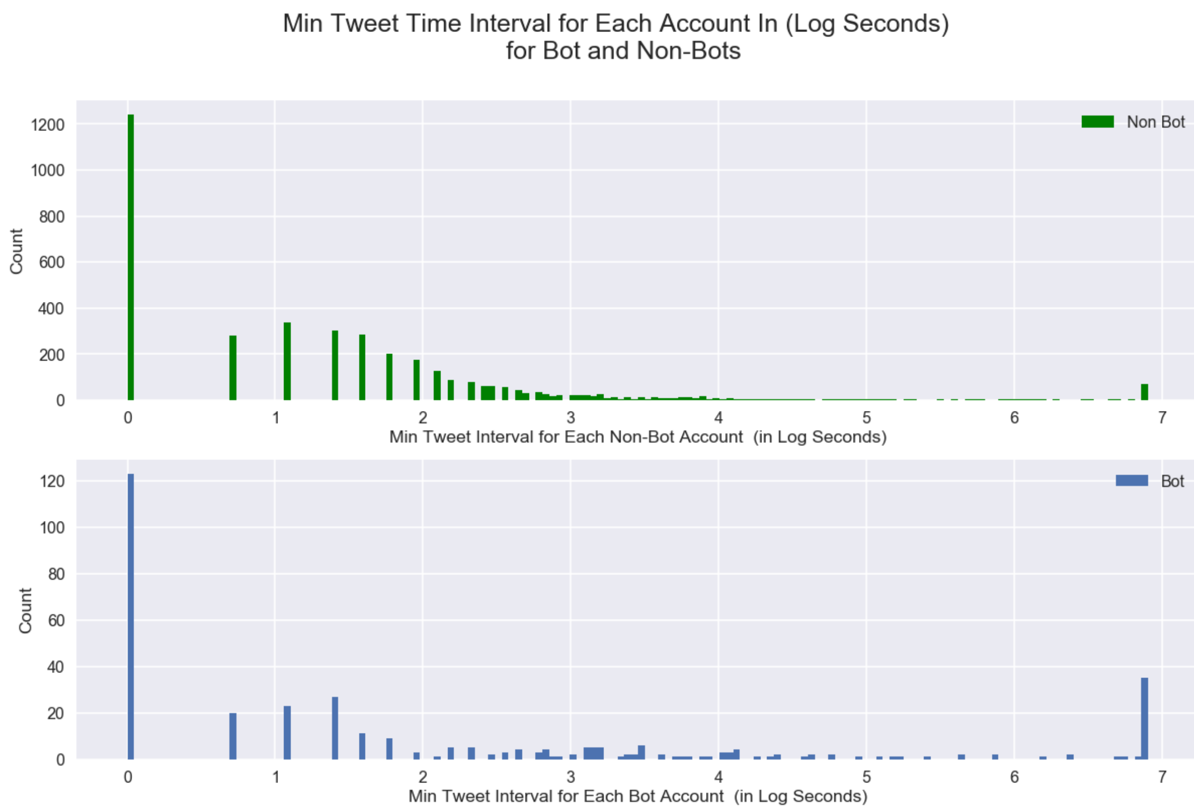



It looks like both botometer-defined non-bot (botometer score < 0.2) and bots (botometer score >=0.2) are heavily screwed towards almost 0 seconds for minimum time interval between tweets of each users. Bots tend to have even more screwed minimum time interval towards 0.

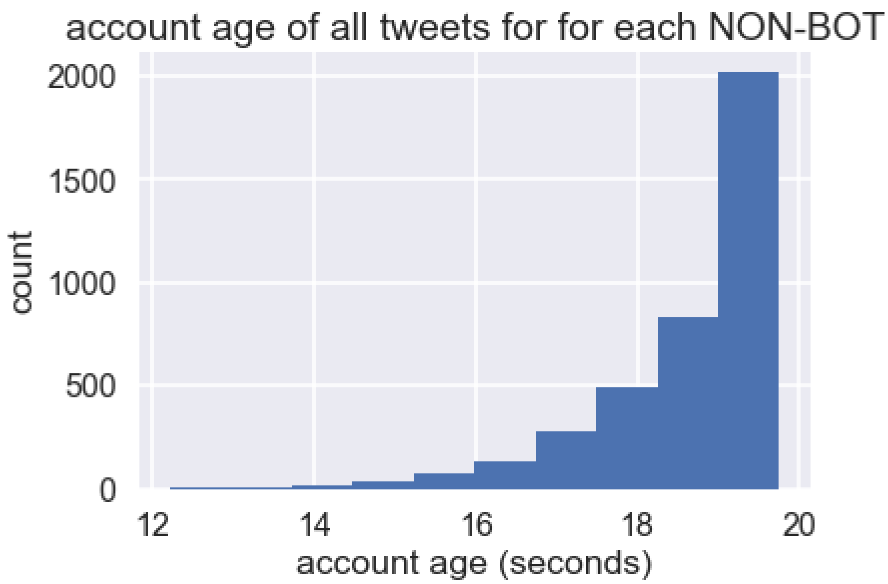
The botometer-identified bots also have heavily screwed minimim tweet time interval but many of them have significantly larger minimum time interval. We think it is reasonable as the bots might be set up to only tweet at a certain interval.

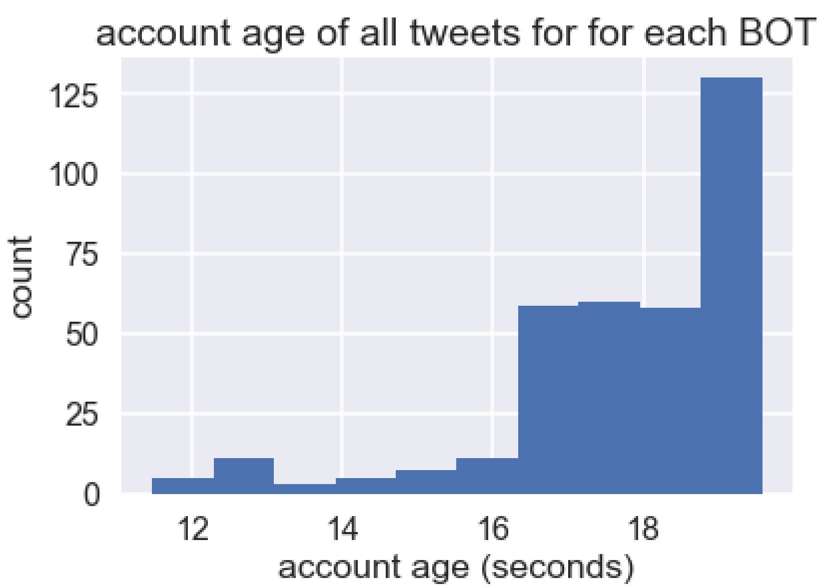
In [492]:
```
fig, ax = plt.subplots(2,1, figsize=(20,12))
fig.suptitle("Min Tweet Time Interval for Each Account In (Log Seconds)
\n for Bot and Non-Bots", fontsize=25)

bot_data = np.log(users_0['tweet_time_min'].values.clip(1, 1000))
non_bot_data = np.log(users_1['tweet_time_min'].values.clip(1, 1000))

ax[0].hist(bot_data, bins=150, label='Non Bot', color="g")
ax[1].hist(non_bot_data, bins=150, label='Bot')

ax[0].set_xlabel('Min Tweet Interval for Each Non-Bot Account  (in Log S
econds)')
ax[0].set_ylabel('Count')
ax[1].set_xlabel('Min Tweet Interval for Each Bot Account  (in Log Secon
ds)')
ax[1].set_ylabel('Count')

ax[0].legend()
ax[1].legend();
```



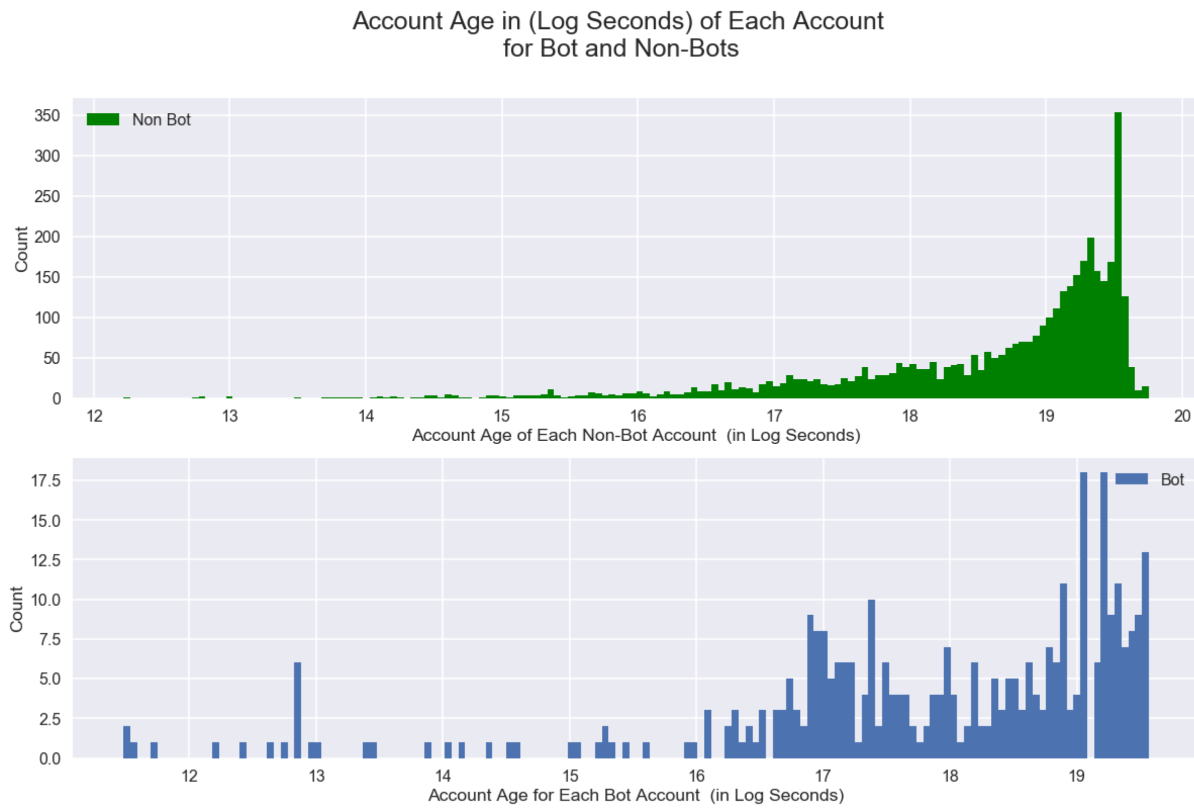

From the plot above, it seems like bots tend to have more large min tweets interval time. One explanation could be that some bots pre-set a time to tweets.

In [493]:
```
# quick plots
plt.figure(figsize=(6,4))
hist_plot(np.log(users_0['account_age'].values.clip(0,1000000000)), 'acc
ount age (seconds)','count', 'account age of all tweets for for each NON
-BOT')
plt.figure(figsize=(6,4))
hist_plot(np.log(users_1['account_age'].values.clip(0,1000000000)), 'acc
ount age (seconds)','count', 'account age of all tweets for for each BO
T')
```



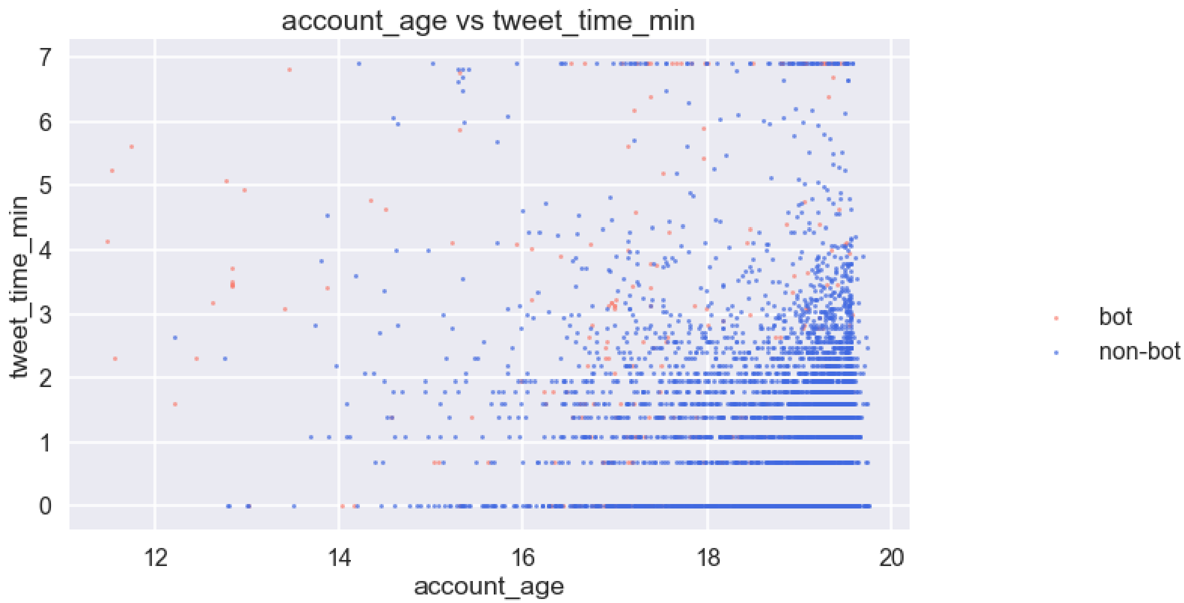



In [494]:
```
fig, ax = plt.subplots(2,1, figsize=(20,12))
fig.suptitle("Account Age in (Log Seconds) of Each Account\n for Bot and
 Non-Bots", fontsize=25)

bot_data = np.log(users_0['account_age'].values.clip(0,1000000000))
non_bot_data =np.log(users_1['account_age'].values.clip(0,1000000000))

ax[0].hist(bot_data, bins=150, label='Non Bot', color="g")
ax[1].hist(non_bot_data, bins=150, label='Bot')

ax[0].set_xlabel('Account Age of Each Non-Bot Account  (in Log Seconds)'
)
ax[0].set_ylabel('Count')
ax[1].set_xlabel('Account Age for Each Bot Account  (in Log Seconds)')
ax[1].set_ylabel('Count')

ax[0].legend()
ax[1].legend();
```



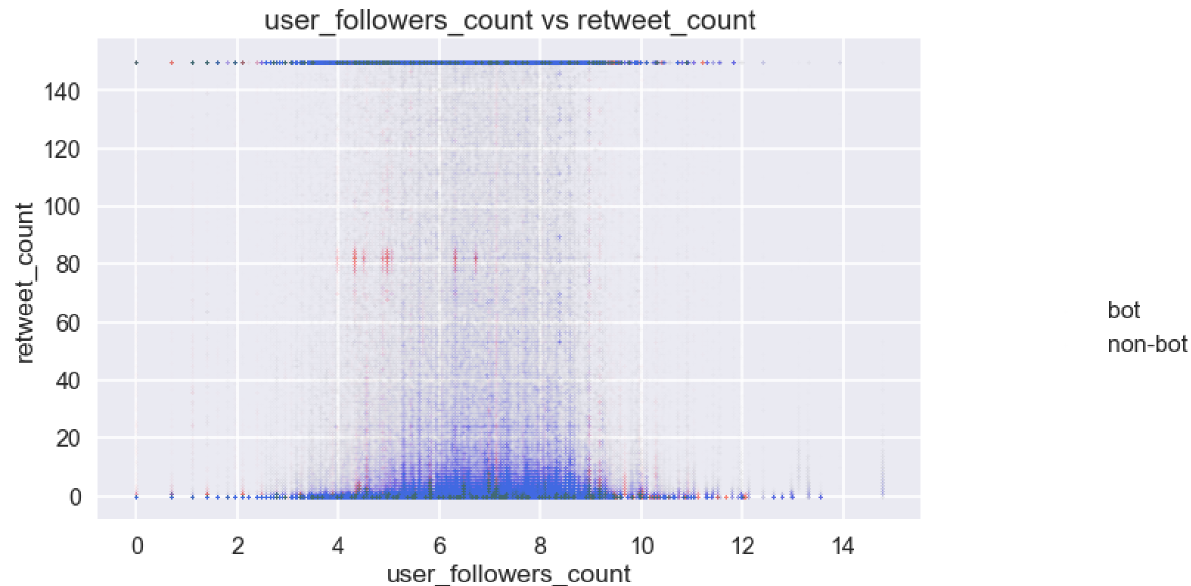

While the account age for non-bot accounts looks continuous, the account_age for bots seemed to be more fragmented - one reason could be waves of hot topics / events.

In [409]:
```
# quick plots
plt.figure(figsize=(10,6))
scatterplot(np.log(users_1['account_age'].values.clip(0,1000000000)), np
.log(users_1['tweet_time_min'].values.clip(1, 1000)),
          np.log(users_0['account_age'].values.clip(0,1000000000)), np.l
og(users_0['tweet_time_min'].values.clip(1, 1000)),
           'account_age', 'tweet_time_min')
```




Given the tweets are the most recent 200 tweets from each account, it looks like the user with older accounts have long minimum time inbetweem two tweets, while bots' minimum time inbetween tweets spread across regardless of account_age.

In [410]:
```
# quick plots
plt.figure(figsize=(10,6))
scatterplot2(np.log(tweets_1['user_followers_count'].values.clip(0,10000
00000)), tweets_1['retweet_count'].values.clip(0,150),
          np.log(tweets_0['user_followers_count'].values.clip(0,10000000
00)), tweets_0['retweet_count'].values.clip(0,150),
           'user_followers_count', 'retweet_count')
```



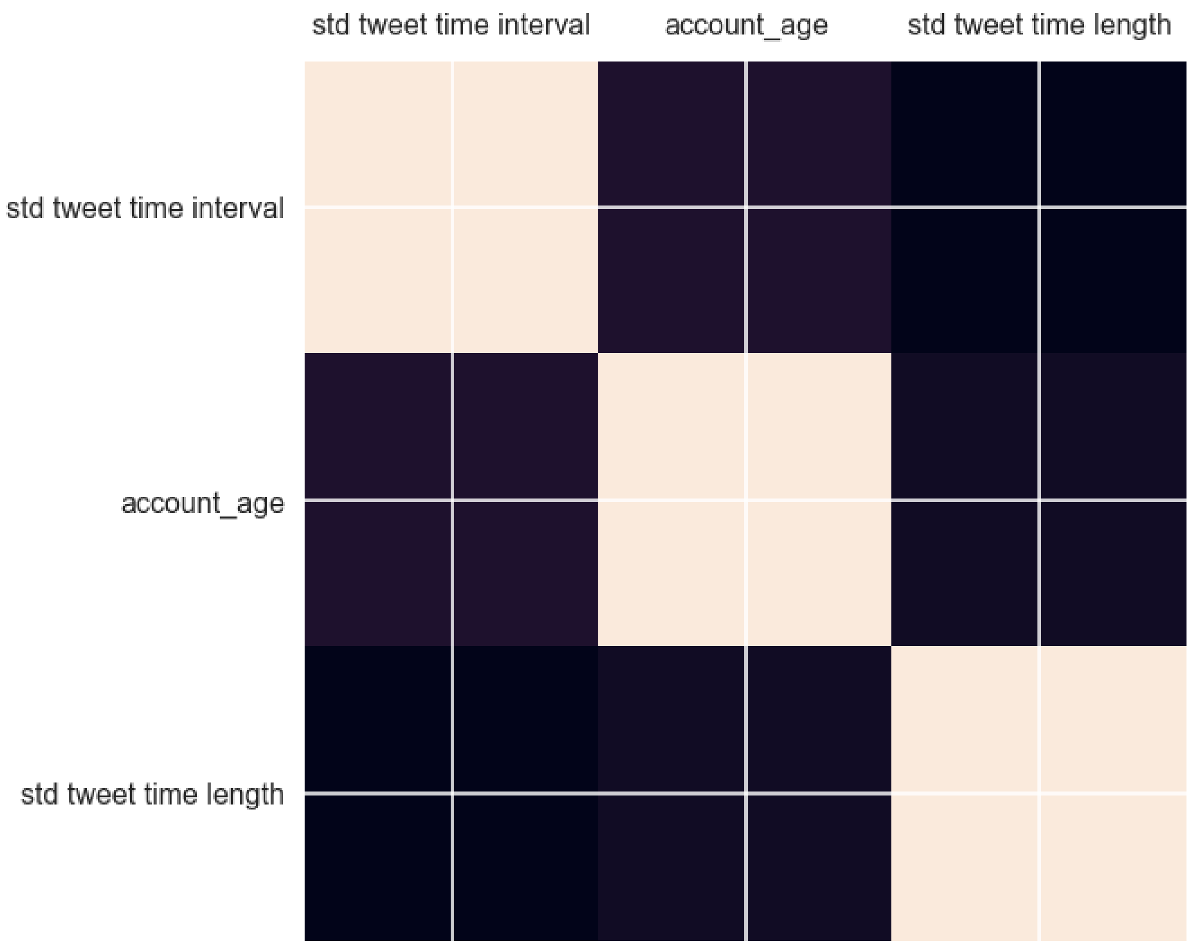

Although the word count of the retweeted post for each tweet has an interesting pattern at around 80 word counts (has most bots), we decided not to include as the rest of the bots are very well blended with non-bots regarding retweet count, and the clusters we've observed above, given the rest of the plot, might be outliers or special events.

Back to TOC
<a id ='Relations-in-Data'></a>

### 2.5 - Relations in Data

Last step before wrapping up the preprocessing, we want to explore the correlation among the different features.

In [495]:
```
# correlation matrix
# quick look at some features we might be interested in

fig, ax = plt.subplots( figsize=(18,9))

col_corr = ['tweet_time_std', 'account_age', 'tweet_len_std']
labels_corr = ['std tweet time interval', 'account_age', 'std tweet time
 length']
ax.matshow(users_df[col_corr].corr())
ax.set_xticklabels([''] + labels_corr)
ax.set_yticklabels([''] + labels_corr);
```




In [496]:
```
# correlation matrix - to be udpated
pd.DataFrame(users_df[col_corr].corr())
```

Out[496]:

| | tweet_time_std | account_age | tweet_len_std |
|---|---|---|---|
| **tweet_time_std** | 1.000000 | 0.036602 | -0.047074 |
| **account_age** | 0.036602 | 1.000000 | -0.001269 |
| **tweet_len_std** | -0.047074 | -0.001269 | 1.000000 |

To our surprise, it seems like there is no strong correlation among these three features, which we thought would be correlated.

We also want to look at the common words / topics of each account among their most recent 200 tweets.

In [553]:
```
def clean_str(string):
    string = re.sub(r"[^A-Za-z0-9(),!?\'\`]", " ", string)
    string = re.sub(r"\'s", " \'s", string)
    string = re.sub(r"\'ve", " \'ve", string)
    string = re.sub(r"n\'t", " n\'t", string)
    string = re.sub(r"\'re", " \'re", string)
    string = re.sub(r"\'d", " \'d", string)
    string = re.sub(r"\'ll", " \'ll", string)
    string = re.sub(r",", " , ", string)
    string = re.sub(r"!", " ! ", string)
    string = re.sub(r"\(", " \( ", string)
    string = re.sub(r"\)", " \) ", string)
    string = re.sub(r"\?", " \? ", string)
    string = re.sub(r"\s{2,}", " ", string)
    string = re.sub(r"\s{2,}", " ", string)
    string = re.sub(r"|", "", string)
    string = re.sub(r"u\s", "", string)
    string = re.sub(r'[^\x00-\x7F]+',' ', string)
    return string.strip().lower()
```

In [554]:
```
def generate_wordCloud(text, title):
    stopwords = set(STOPWORDS)
    extra_stopwords = {'one', 'al','et', 'br', 'Po', 'th', 'sayi', 'fr',
'wi', 'Unknown','co',
                       'https', 'u\'rt', 'nhttp','text', 'rt', "amp", "n
https",'u','n\'t'}
    stopwords = stopwords.union(extra_stopwords)
    wc = WordCloud(stopwords=stopwords,
                  max_font_size=100,
                  max_words=100,
                  random_state=30,
                  background_color='white',mask=None).generate(str(text
))

    plt.figure(figsize=(15,20))
    plt.imshow(wc, interpolation="bilinear")
    plt.axis('off') # dont show the axes
    plt.title(title, fontdict={'size': 35,'color':"red" ,
                                'verticalalignment': 'bottom'})
    plt.show()
```

In [555]:
```
bot_tweets_str = ""
nonbot_tweets_str = ""

df1 = tweets_1[['text_tweet']]
df0 = tweets_0[['text_tweet']]


df1.fillna(value=pd.np.nan, inplace=True) #Make all NA variable as NAN
df0.fillna(value=pd.np.nan, inplace=True) #Make all NA variable as NAN

dataset_1=df1[df1.text_tweet.notnull()] #get not NULL data into dataset_
dataset_0=df0[df0.text_tweet.notnull()]
```

In [556]:
```
# take a look at the first 10 tweets
display(dataset_0[0:10])
```

| | text_tweet |
|---|---|
| 200 | @HistoryMeister @paulgilding Lascelles - there... |
| 203 | Almost more important that Brexit #england #br... |
| 208 | @Prof_Malhotra some local colour amidst this l... |
| 209 | Spot on as always @Prof_Malhotra ! It was alwa... |
| 219 | Dear @BBCNewsnight Please ask @EmilyMaitlisMVP... |
| 220 | #PRCAAwards Well done @VicNayLey New PR firm... |
| 221 | #PRCAAwards Well done @PembrokeAndRye New PR... |
| 224 | @RBKC The residents of Blenheim Crescent, Port... |
| 228 | @PoppyLegion I saw this in Petworth today - in... |
| 229 | They have strong views on these matters at the... |

In [557]:
```
for tweet in  dataset_1.text_tweet[0:10]:
    bot_tweets_str = bot_tweets_str+ dataset_1.text_tweet.replace('NaN',
'')


for tweet in  dataset_0.text_tweet[0:10]:
    nonbot_tweets_str  = nonbot_tweets_str+ dataset_0.text_tweet.replace
('NaN','')
```

In [ ]:
```
# to download : conda install -c conda-forge wordcloud or pip install wo
rdcloud
from wordcloud import WordCloud

generate_wordCloud(bot_tweets_str,'Bot')
generate_wordCloud(nonbot_tweets_str,'Non-Bot')
```

Back to TOC
<a id ='Standardization'></a>

#### 2.6 - Standardization and Discussion

Up to this point, it has become very obvious that the most interesting features in telling bots / non-bots apart are account-level features. Moreover, account-level engineered features are much better at telling bot / non-bots apart.

Moreover, from the plots in the previous sections, it became obvious that many of our data are not normally distributed. Standardization is necessary to make our model legit.

As the last step of the EDA and data preprocessing, we consolidated all our account-level data, remove the columns that no longer be useful in our analysis, the standardize the numerical features.

In [440]:
```
# read the data
users_df = pd.read_json('users_final.json')
```

In [538]:
```
users_df.columns.values
```

Out[538]:
```
array(['favorite_count', 'id', 'retweet_count', 'retweeted_status',
       'user_description', 'user_favourites_count',
       'user_followers_count', 'user_friends_count', 'user_listed_coun
t',
       'user_statuses_count', 'text_rt', 'text_tweet', 'screen_name',
       'class_boto', 'tweet_time_mean', 'tweet_time_std',
       'tweet_time_min', 'tweet_time_max', 'user_description_len',
       'account_age', 'tweet_len_mean', 'tweet_len_std',
       'tweet_word_mean', 'tweet_word_std', 'retweet_len_mean',
       'retweet_len_std', 'retweet_word_mean', 'retweet_word_std'],
      dtype=object)
```

In [539]:
```
# we want to check how many accounts have left after all the cleansing
users_df.shape
```

Out[539]:
```
(4226, 28)
```

In [540]:
```
display(users_df.columns.values)
display(users_df.shape)
```

```
array(['favorite_count', 'id', 'retweet_count', 'retweeted_status',
       'user_description', 'user_favourites_count',
       'user_followers_count', 'user_friends_count', 'user_listed_coun
t',
       'user_statuses_count', 'text_rt', 'text_tweet', 'screen_name',
       'class_boto', 'tweet_time_mean', 'tweet_time_std',
       'tweet_time_min', 'tweet_time_max', 'user_description_len',
       'account_age', 'tweet_len_mean', 'tweet_len_std',
       'tweet_word_mean', 'tweet_word_std', 'retweet_len_mean',
       'retweet_len_std', 'retweet_word_mean', 'retweet_word_std'],
      dtype=object)
```

```
(4226, 28)
```

We still have 28 columns, which include two reference columns ('id' and 'screen_name'), one predictor column ('class_boto').

In [541]:
```
users_df.dtypes
```

Out[541]:
```
favorite_count            int64
id                        int64
retweet_count             int64
retweeted_status          int64
user_description         object
user_favourites_count     int64
user_followers_count      int64
user_friends_count        int64
user_listed_count         int64
user_statuses_count       int64
text_rt                  object
text_tweet               object
screen_name              object
class_boto                int64
tweet_time_mean         float64
tweet_time_std          float64
tweet_time_min          float64
tweet_time_max          float64
user_description_len      int64
account_age               int64
tweet_len_mean          float64
tweet_len_std           float64
tweet_word_mean         float64
tweet_word_std          float64
retweet_len_mean        float64
retweet_len_std         float64
retweet_word_mean       float64
retweet_word_std        float64
dtype: object
```

In [545]:
```
# separate numerical columns and text columns again
col_response = ['class_boto']
col_pred_text = list(users_df.select_dtypes(['object']).drop(columns=['s
creen_name']).columns.values)
col_ref = ['id', 'screen_name']
col_pred_numerical = list(users_df.select_dtypes(['float64', 'int64']).d
rop(columns=['class_boto', 'id']).columns.values)
```

In [546]:
```
# save the column lists
c_list_names = ['col_pred_numerical', 'col_ref', 'col_response', 'col_pr
ed_text']
c_list = [col_pred_numerical, col_ref, col_response, col_pred_text]
for c_name, c in zip(c_list_names, c_list):
    with open(c_name+'.txt', 'w') as fp:
        ls_str = ",".join(col_pred_numerical)
        fp.write(ls_str)
```

In [547]:
```
display(users_df.shape)
display(users_df.isna().any())
```

```
(4226, 28)

favorite_count             False
id                         False
retweet_count              False
retweeted_status           False
user_description            True
user_favourites_count      False
user_followers_count       False
user_friends_count         False
user_listed_count          False
user_statuses_count        False
text_rt                     True
text_tweet                  True
screen_name                False
class_boto                 False
tweet_time_mean            False
tweet_time_std             False
tweet_time_min             False
tweet_time_max             False
user_description_len       False
account_age                False
tweet_len_mean              True
tweet_len_std               True
tweet_word_mean             True
tweet_word_std              True
retweet_len_mean            True
retweet_len_std             True
retweet_word_mean           True
retweet_word_std            True
dtype: bool
```

In [548]:
```
# cleaning up NaN on numerical columns by assigning them 0
users_df[col_pred_numerical] = users_df[col_pred_numerical].fillna(0)
```

In [549]:
```
from sklearn import preprocessing

def standardize(df):
    scaler = preprocessing.StandardScaler()
    df = scaler.fit_transform(df)
    return df
```

In [550]:
```
# create a new copy with numercial columns standardized
users_df[col_pred_numerical] = standardize(users_df[col_pred_numerical])
```

In [551]:
```
# check if the copy
display(users_df.describe())
display(users_df.shape)
```

| | favorite_count | id | retweet_count | retweeted_status | user_favourites_c |
|---|---|---|---|---|---|
| **count** | 4.226000e+03 | 4.226000e+03 | 4.226000e+03 | 4.226000e+03 | 4.226000e+03 |
| **mean** | 9.703943e-17 | 1.072062e+18 | -5.133731e-17 | 1.249986e-16 | -2.878015e-17 |
| **std** | 1.000118e+00 | 3.518030e+15 | 1.000118e+00 | 1.000118e+00 | 1.000118e+00 |
| **min** | -1.017011e-01 | 8.705541e+17 | -1.115030e-01 | -1.265154e+00 | -4.849644e-01 |
| **25%** | -1.017011e-01 | 1.072195e+18 | -1.115030e-01 | -1.265154e+00 | -4.587417e-01 |
| **50%** | -1.017011e-01 | 1.072253e+18 | -1.105211e-01 | 7.904174e-01 | -3.381680e-01 |
| **75%** | -1.017011e-01 | 1.072275e+18 | -7.827123e-02 | 7.904174e-01 | 2.087058e-02 |
| **max** | 3.378112e+01 | 1.072320e+18 | 4.637228e+01 | 7.904174e-01 | 1.793084e+01 |

```
(4226, 28)
```

In [552]:
```
# save to json
users_df.to_json('users_final_std.json')
```

Back to TOC
<a id ='Models'></a>

## 4 - Models

We splited train / test dataset by 0.25 and stratify by class_boto to ensure equal presentation of bots account in both datasets. The baseline accuracy of training dataset was 91.73%, the baseline accuracy for test set was 91.77%. Both of which are quite high.
By testing several model, we were able to achieve an accuracy up to

In [2]:
```
# read the data
users_df = pd.read_json('users_final_std.json')
```

In [3]:
```
# Train/Test split
'''
change as needed, do we want test_size of .25?
'''
train_df, test_df = train_test_split(users_df, test_size=.25,
                                                              stratify=users_df
.class_boto, random_state=99)
```

In [4]:
```
with open('col_pred_numerical.txt', 'r') as fp:
    col_pred_numerical = fp.read().split(',')
with open('col_response.txt', 'r') as fp:
    col_response = fp.read().split(',')
with open('col_pred_text.txt', 'r') as fp:
    col_pred_text = fp.read().split(',')
with open('col_ref.txt', 'r') as fp:
    col_ref = fp.read().split(',')
```

In [5]:
```
# write a function to split the data
def split_data(df):
    # num_pred: standardized numerical predictors - what we will be usin
g for most of the models
    # text_pred: text features that are associated with the tweets - onl
y useful for NLP
    # response: response - manually verified classification. 1=bot; 0=no
n-bot
    # ids: 'id'
    # boto: botometer values
    num_pred, text_pred, response = df[col_pred_numerical], df[col_pred_
text], df['class_boto']
    ids, screen_name = df['id'], df['screen_name']
    return num_pred, text_pred, response, ids, screen_name
```

In [6]:
```
# get the predictors, responses, and other features from train and test
 set
xtrain, xtrain_text, ytrain, train_id, train_sn = split_data(train_df)
xtest, xtest_text, ytest, test_id, test_sn = split_data(test_df)
```

In [7]:
```
# save to json
f_list_names = ['train_df', 'test_df', 'xtrain', 'xtrain_text', 'ytrain'
, 'train_id', 'train_sn', 'xtest', 'xtest_text', 'ytest', 'test_id', 'te
st_sn']
f_list = [train_df, test_df, xtrain, xtrain_text, ytrain, train_id, trai
n_sn, xtest, xtest_text, ytest, test_id, test_sn]
for f_name, f in zip(f_list_names, f_list):
    f.to_json(f_name + '.json')
```

In [8]:
```
# create a dictionary to store all our models
models_list = {}
acc ={}
```

In [9]:
```
# take a quick look at the accuracy if we just choose to classifying eve
rything as users
baseline_train_acc = float(1-sum(ytrain)/len(ytrain))
baseline_test_acc = float(1-sum(ytest)/len(ytest))
print('the baseline accuracy for training set is {:.2f}%, for test set i
s {:.2f}%.'.format(baseline_train_acc*100,

                   baseline_test_acc*100))
```

```
the baseline accuracy for training set is 91.73%, for test set is 91.7
7%.
```

In [10]:
```
# save baseline acc to model list
acc['bl'] = (baseline_train_acc, baseline_test_acc)
```

Back to TOC
<a id ='Baseline-Model'></a>

#### 4.1 - Baseline Model - Simple Linear Regression

Although this is a classification problem that people normally won't use linear regression, we thought we could try with a threshold of 0.5 and use it as a baseline model.

Our Test score is around 91.39% on the test data which is not bad for a Base Model at the first glance; as our possibilies are either Bot or No-Bot. However, it is actually lower than our baseline accuracy on test set, which was 91.77%. Therefore, OLS, even we tried to use threshold, it is not performing, we need to improve the model.

In [11]:
```
# multiple linear regression(no poly)on numerical predictors
X_train = sm.add_constant(xtrain)
X_test = sm.add_constant(xtest)
y_train = ytrain.values.reshape(-1,1)
y_test = ytest.values.reshape(-1,1)
```

In [12]:
```
# Fit and summarize OLS model
model = OLS(y_train, X_train)
results = model.fit()

results.summary()
```

Out[12]:

OLS Regression Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | y | **R-squared:** | 0.217 |
| **Model:** | OLS | **Adj. R-squared:** | 0.212 |
| **Method:** | Least Squares | **F-statistic:** | 45.83 |
| **Date:** | Wed, 12 Dec 2018 | **Prob (F-statistic):** | 3.40e-151 |
| **Time:** | 21:45:50 | **Log-Likelihood:** | -23.162 |
| **No. Observations:** | 3169 | **AIC:** | 86.32 |
| **Df Residuals:** | 3149 | **BIC:** | 207.5 |
| **Df Model:** | 19 | | |
| **Covariance Type:** | nonrobust | | |

| | coef | std err | t | P>\|t\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **const** | 0.0845 | 0.004 | 19.328 | 0.000 | 0.076 | 0.093 |
| **user_favourites_count** | -0.0111 | 0.004 | -2.478 | 0.013 | -0.020 | -0.002 |
| **user_followers_count** | -0.0473 | 0.011 | -4.433 | 0.000 | -0.068 | -0.026 |
| **user_friends_count** | 0.0281 | 0.005 | 5.138 | 0.000 | 0.017 | 0.039 |
| **user_listed_count** | 0.0332 | 0.009 | 3.816 | 0.000 | 0.016 | 0.050 |
| **user_statuses_count** | -0.0052 | 0.004 | -1.218 | 0.223 | -0.014 | 0.003 |
| **tweet_time_mean** | 0.1159 | 0.039 | 2.964 | 0.003 | 0.039 | 0.193 |
| **tweet_time_std** | -0.0043 | 0.029 | -0.148 | 0.883 | -0.061 | 0.052 |
| **tweet_time_min** | -0.0337 | 0.008 | -4.427 | 0.000 | -0.049 | -0.019 |
| **tweet_time_max** | -0.0053 | 0.015 | -0.360 | 0.719 | -0.034 | 0.023 |
| **user_description_len** | -0.0011 | 0.005 | -0.234 | 0.815 | -0.010 | 0.008 |
| **account_age** | -0.0367 | 0.004 | -8.171 | 0.000 | -0.046 | -0.028 |
| **tweet_len_mean** | 0.0171 | 0.007 | 2.621 | 0.009 | 0.004 | 0.030 |
| **tweet_len_std** | -0.0579 | 0.006 | -9.066 | 0.000 | -0.070 | -0.045 |
| **tweet_word_mean** | -0.0577 | 0.008 | -7.453 | 0.000 | -0.073 | -0.043 |
| **tweet_word_std** | 0.0065 | 0.007 | 0.871 | 0.384 | -0.008 | 0.021 |
| **retweet_len_mean** | 0.0167 | 0.008 | 2.003 | 0.045 | 0.000 | 0.033 |
| **retweet_len_std** | 0.0007 | 0.007 | 0.107 | 0.915 | -0.013 | 0.014 |
| **retweet_word_mean** | -0.1547 | 0.016 | -9.530 | 0.000 | -0.187 | -0.123 |
| **retweet_word_std** | 0.0783 | 0.015 | 5.145 | 0.000 | 0.048 | 0.108 |

| | | | |
|---|---|---|---|
| **Omnibus:** | 1418.815 | **Durbin-Watson:** | 1.975 |

| | | | |
|---|---|---|---|
| **Prob(Omnibus):** | 0.000 | **Jarque-Bera (JB):** | 6760.888 |
| **Skew:** | 2.163 | **Prob(JB):** | 0.00 |
| **Kurtosis:** | 8.699 | **Cond. No.** | 21.2 |

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [13]:
```
y_hat_train = results.predict()
y_hat_test = results.predict(exog=X_test)

# get Train & Test R^2
print('Train R^2 = {}'.format(results.rsquared))
print('Test R^2 = {}'.format(r2_score(test_df['class_boto'], y_hat_test
)))
```

```
Train R^2 = 0.21662050202218985
Test R^2 = -0.2992911496733639
```

In [14]:
```
# accuracy score
ols_train_acc = accuracy_score(y_train, results.predict(X_train).round()
.clip(0, 1))
ols_test_acc = accuracy_score(y_test, results.predict(X_test).round().cl
ip(0, 1))
print("Training accuracy is {:.4}%".format(ols_train_acc*100))
print("Test accuracy is {:.4} %".format(ols_test_acc*100))
```

```
Training accuracy is 91.86%
Test accuracy is 91.39 %
```

In [15]:
```
# save model to the list
models_list["ols"] = results
acc['ols'] = (ols_train_acc, ols_test_acc)
```

In [16]:
```
# pickle ols
import pickle

filename = 'ols.sav'
pickle.dump(results, open(filename, 'wb'))
```

In [17]:
```
#loaded_model = pickle.load(open(filename,'rb'))
```

Back to TOC
<a id ='Linear-Regression-with-Ridge'></a>

#### 4.2 - Linear Regression with Ridge

Although in the simple linear model, the test score is comparable to training score and there was no sign of overfitting, we still want to try Ridge to see if we could reduce any potential overfitting.

With ridge selection, we received a test accuracy of 91.96%, which is slightly improved from 91.39% (OLS), which implies that the OLS model does not have overfitting. However, it is still about the same / lower than baseline accuracy.

In [18]:
```
alphas = np.array([.01, .05, .1, .5, 1, 5, 10, 50, 100])
fitted_ridge = RidgeCV(alphas=alphas, cv=5).fit(X_train, y_train)
```

In [19]:
```
# accuracy score
ridge_train_acc = accuracy_score(y_train, fitted_ridge.predict(X_train).
round().clip(0, 1))
ridge_test_acc = accuracy_score(y_test, fitted_ridge.predict(X_test).rou
nd().clip(0, 1))
print("Training accuracy is {:.4}%".format(ridge_train_acc*100))
print("Test accuracy is {:.4} %".format(ridge_test_acc*100))
```

```
Training accuracy is 91.92%
Test accuracy is 91.96 %
```

In [20]:
```
# save model to the list
models_list["ridge"] = fitted_ridge
filename = 'ridge.sav'
pickle.dump(fitted_ridge, open(filename, 'wb'))
acc['ridge'] = (ridge_train_acc, ridge_test_acc)
```

Back to TOC
<a id ='Lasso'></a>

#### 4.3 - Lasso

We also want to try feature reductions with Lasso and see if the model will perform better by dropping less important features. The lasso model received an accuracy of 91.77%, which again improves from 91.14% but not very significant, and it is just slightly higher than baseline accuracy. However, Lasso may not have significant improvement on test accuracy but lead to differnet coefficients. We want to examine that.

In [21]:
```
fitted_lasso = LassoCV(alphas=alphas, max_iter=100000, cv=5).fit(X_train
, y_train)
```

```
/anaconda3/lib/python3.6/site-packages/sklearn/linear_model/coordinate_
descent.py:1094: DataConversionWarning: A column-vector y was passed wh
en a 1d array was expected. Please change the shape of y to (n_samples,
), for example using ravel().
  y = column_or_1d(y, warn=True)
```

In [22]:
```
# accuracy score
lasso_train_acc = accuracy_score(y_train, fitted_lasso.predict(X_train).
round().clip(0, 1))
lasso_test_acc = accuracy_score(y_test, fitted_lasso.predict(X_test).rou
nd().clip(0, 1))
print("Training accuracy is {:.4}%".format(lasso_train_acc*100))
print("Test accuracy is {:.4} %".format(lasso_test_acc*100))
```

```
Training accuracy is 91.8%
Test accuracy is 91.77 %
```

In [23]:
```
# save model to the list
models_list["lasso"] = fitted_lasso
filename = 'lasso.sav'
pickle.dump(fitted_lasso, open(filename, 'wb'))
acc['lasso']=(lasso_train_acc, lasso_test_acc)
```

Back to TOC
<a id ='Lasso-and-Ridge-Coefficients-Comparison'></a>

#### 4.4 - Lasso and Ridge Coefficients Comparison

We want to see how lasso and ridge results in different coefficients. As expected, Lasso greatly reduced the number of non-zero coefficients.

In [24]:
```
for feature, coef in zip(xtrain.columns.values.tolist(), fitted_ridge.co
ef_[0].tolist()):
    print("{}: {}".format(feature, coef))
```

```
user_favourites_count: 0.0
user_followers_count: -0.010984392775297717
user_friends_count: -0.0373860331993809
user_listed_count: 0.024158234955190823
user_statuses_count: 0.026646084813644076
tweet_time_mean: -0.004497720856292946 5
tweet_time_std: 0.044232900606536285
tweet_time_min: 0.029995359711894685
tweet_time_max: -0.02480300277881136
user_description_len: -0.012475212935169604
account_age: -0.0015423611100371431
tweet_len_mean: -0.03616251833633441
tweet_len_std: 0.01451572510127823
tweet_word_mean: -0.054079504222467004
tweet_word_std: -0.0514121927327697
retweet_len_mean: 0.0025870175288910087
retweet_len_std: 0.007751629182111389
retweet_word_mean: -0.003022968737127
retweet_word_std: -0.09823618767855992
```

In [25]:
```
for feature, coef in zip(xtrain.columns.values.tolist(), fitted_lasso.co
ef_.tolist()):
    print("{}: {}".format(feature, coef))
```

```
user_favourites_count: 0.0
user_followers_count: -0.0030684119532310974
user_friends_count: -0.0
user_listed_count: 0.005861006334253806
user_statuses_count: 0.0
tweet_time_mean: -0.0
tweet_time_std: 0.0
tweet_time_min: 0.019705055827586075
tweet_time_max: -0.0
user_description_len: 0.0
account_age: -0.0
tweet_len_mean: -0.02878718176736991
tweet_len_std: 0.0
tweet_word_mean: -0.04152323503512772
tweet_word_std: -0.03718634181572111
retweet_len_mean: -0.0
retweet_len_std: -0.0
retweet_word_mean: -0.0
retweet_word_std: -0.06062048667775922
```

Back to TOC
<a id ='Logistic-Regression'></a>

#### 4.5 - Logistic Regression

The logistic regression presented a small improvement on the accuracy from the base model, we need to try additional techniques to improve the accuracy.

In [26]:
```
X_train = sm.add_constant(xtrain)
X_test = sm.add_constant(xtest)

logistic_model = LogisticRegression().fit(X_train, ytrain)

logistic_model_score = logistic_model.score(X_test, ytest)

print("Train set score: {0:4.4}%".format(logistic_model.score(X_train, y
train)*100))
print("Test set score: {0:4.4}%".format(logistic_model.score(X_test, yte
st)*100))
```

```
Train set score: 92.55%
Test set score: 91.49%
```

In [27]:
```
models_list["simple_logistic"] = logistic_model
filename = 'simple_logistic.sav'
pickle.dump(logistic_model, open(filename, 'wb'))
acc['lm'] = (logistic_model.score(X_train, ytrain), logistic_model_score
)
```

Back to TOC
<a id ='Logistic-Regression-with-cross-validation'></a>

#### 4.6 - Logistic Regression with cross validation

Logistic regression with Cross Validation has improved the accuracy and reached 91.96% on Test data which is an improvement from the Logistic regression, we will continue to see if we can improve further using other techniques.

In [28]:
```
logistic_model_cv = LogisticRegressionCV(Cs=[1,10,100,1000,10000], cv=3,
 penalty='l2',
                                          solver='newton-cg').fit(X_train,y
train)

print("Train set score with Cross Validation: {0:4.4}%".format(logistic_
model_cv.score(X_train, ytrain)*100))
print("Test set score with Cross Validation: {0:4.4}%".format(logistic_m
odel_cv.score(X_test, ytest)*100))
```

```
Train set score with Cross Validation: 92.58%
Test set score with Cross Validation: 91.96%
```

In [29]:
```
models_list["simple_logistic_Cross_Validation"] = logistic_model_cv
filename = 'logistic_model_cv.sav'
pickle.dump(logistic_model_cv, open(filename, 'wb'))
acc['lm_cv3'] = (logistic_model_cv.score(X_train, ytrain), logistic_mode
l_cv.score(X_test, ytest))
```

Back to TOC
<a id ='Logistic-Regression-with-polynomial-degree-3'></a>

#### 4.7 - Logistic Regression with polynomial degree 3

Test score accuracy has increased with Polynomial degree of predictors for Logistic Regression on the test data and reached 93.47%.

In [30]:
```
X_train_poly = PolynomialFeatures(degree=3, include_bias=False).fit_tran
sform(X_train)

logistic_model_poly_cv = LogisticRegressionCV(Cs=[1,10,100,1000,10000],
cv=3, penalty='l2',
                                          solver='newton-cg').fit(X_train_p
oly,ytrain)

X_test_poly = PolynomialFeatures(degree=3, include_bias=False).fit_trans
form(X_test)
print("Train set score with Polynomial Features (degree=3) and with Cros
s Validation: {0:4.4}%".
      format(logistic_model_poly_cv.score(X_train_poly, ytrain)*100))
print("Test set score with Polynomial Features (degree=3) and with Cross
 Validation: {0:4.4}%".
      format(logistic_model_poly_cv.score(X_test_poly, ytest)*100))
```

```
Train set score with Polynomial Features (degree=3) and with Cross Vali
dation: 98.26%
Test set score with Polynomial Features (degree=3) and with Cross Valid
ation: 93.47%
```

In [31]:
```
models_list["poly_logistic_cv"] = logistic_model_poly_cv
filename = 'logistic_model_poly_cv.sav'
pickle.dump(logistic_model_poly_cv, open(filename, 'wb'))
acc['lm_poly3'] = (logistic_model_poly_cv.score(X_train_poly, ytrain), l
ogistic_model_poly_cv.score(X_test_poly, ytest))
```

Back to TOC
<a id ='KNN'></a>

#### 4.8 - KNN

We have tested the k-Nearest Neighbors algorithm as well and we used cross validation to evaluate the best k with the highest accuracy score. We have stored the best k in the variable best_k which has a value equal of 17. The test score is higher than the base model but lower than Logistic Regression with polynomial degree 3.

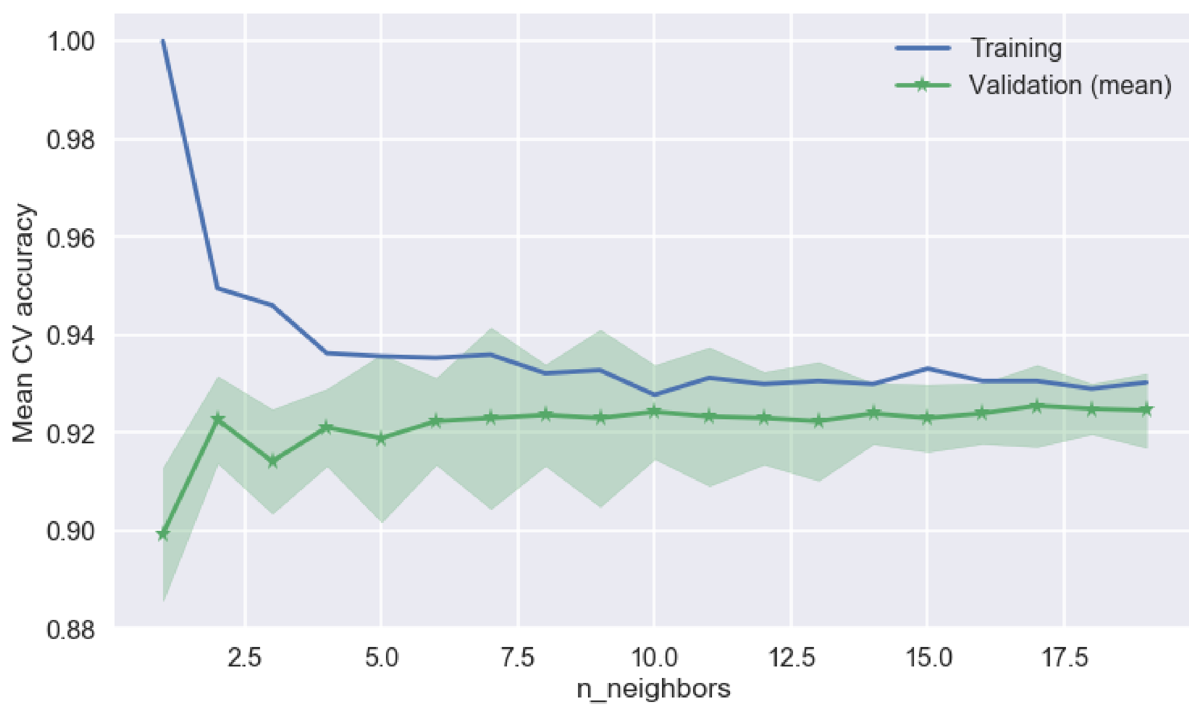
In [32]:
```
# the code below in KNN is adapted from HW2 solution

# define k values
k_values = range(1,20)

# build a dictionary KNN models
KNNModels = {k: KNeighborsClassifier(n_neighbors=k) for k in k_values}
train_scores = [KNeighborsClassifier(n_neighbors=k).fit(xtrain, ytrain).
score(xtrain, ytrain) for k in k_values]
cv_scores = [cross_val_score(KNeighborsClassifier(n_neighbors=k), xtrain
, ytrain, cv=5) for k in k_values]


# fit each KNN model
for k_value in KNNModels:
    KNNModels[k_value].fit(xtrain, ytrain)
```

In [33]:
```
# Generate predictions
knn_predicted_train = {k: KNNModels[k].predict(xtrain) for k in KNNModel
s}
knn_predicted_test = {k: KNNModels[k].predict(xtest) for k in KNNModels}
```

In [34]:
```
# the following code was adapted from HW7 solutions
def plot_cv(ax, hyperparameter, cv_scores):
    cv_means = np.mean(cv_scores, axis=1)
    cv_stds = np.std(cv_scores, axis=1)
    handle, = ax.plot(hyperparameter, cv_means, '-*', label="Validation
 (mean)")
    plt.fill_between(hyperparameter, cv_means - 2.*cv_stds, cv_means +
2.*cv_stds, alpha=.3, color=handle.get_color())
```

In [35]:
```
# the following code was adapted from HW7 solutions
# find the best model
fig, ax = plt.subplots(figsize=(12,7))
ax.plot(k_values, train_scores, '-+', label="Training")
plot_cv(ax, k_values, cv_scores)
plt.xlabel("n_neighbors")
plt.ylabel("Mean CV accuracy");
plt.legend()

best_k = k_values[np.argmax(np.mean(cv_scores, axis=1))]
print("Best k:", best_k)
```

```
Best k: 17
```




In [36]:
```
# evaluate classification accuracy
best_model_KNN_train_score = accuracy_score(ytrain, knn_predicted_train[
best_k].round())
best_model_KNN_test_score = accuracy_score(ytest, knn_predicted_test[bes
t_k].round())
print("Training accuracy is {:.4}%".format(best_model_KNN_train_score*10
0))
print("Test accuracy is {:.4} %".format(best_model_KNN_test_score*100))
```

```
Training accuracy is 93.06%
Test accuracy is 92.72 %
```

In [37]:
```
# save model to the list
best_k = 17
best_k_17 = KNNModels[best_k].fit(xtrain, ytrain)

models_list["knn_17"] = best_k_17
filename = 'knn_17.sav'
pickle.dump(best_k_17, open(filename, 'wb'))
acc['knn_17'] = (best_model_KNN_train_score, best_model_KNN_test_score)
```

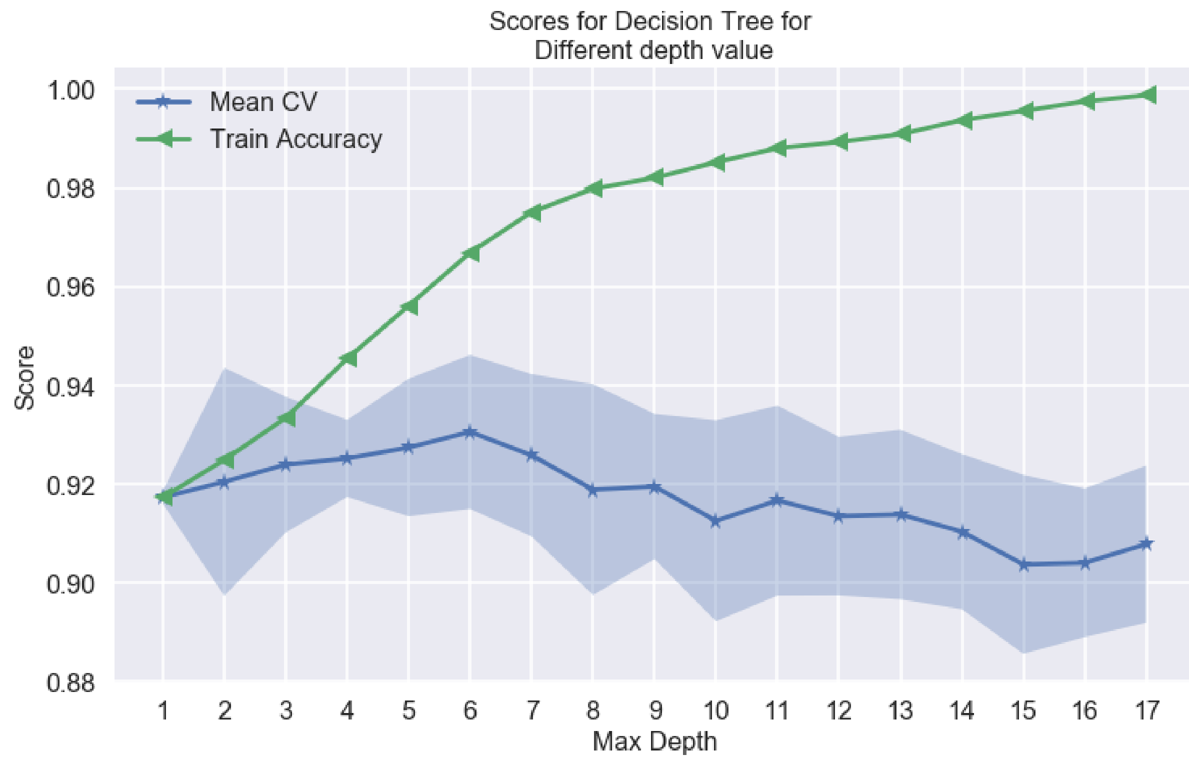
Back to TOC
<a id ='Decision-tree'></a>

#### 4.9 - Decision tree

The decision tree is performing similiar to the logistic regression with polynomial 3.

In [38]:
```
depth_list =list(range(1, 18))

cv_means = []
cv_stds = []
train_scores = []
best_model_mean = 0

for depth in depth_list:
    #Fit a decision tree to the training set
    model_DTC = DecisionTreeClassifier(max_depth=depth).fit(xtrain, ytra
in)
    scores = cross_val_score(model_DTC, xtrain, ytrain, cv=5)

    #training set performance
    train_scores.append(model_DTC.score(xtrain, ytrain))

    #save best model
    if scores.mean() > best_model_mean:
            best_model_mean=scores.mean()
            best_model_DTC=model_DTC
            best_model_std =scores.std()


    #performance for 5-fold cross validation
    cv_means.append(scores.mean())
    cv_stds.append(scores.std())


cv_means = np.array(cv_means)
cv_stds = np.array(cv_stds)
train_scores = np.array(train_scores)
```

In [39]:
```
plt.subplots(1, 1, figsize=(12,7))
plt.plot(depth_list, cv_means, '*-', label="Mean CV")
plt.fill_between(depth_list, cv_means - 2*cv_stds, cv_means + 2*cv_stds,
 alpha=0.3)
ylim = plt.ylim()
plt.plot(depth_list, train_scores, '<-', label="Train Accuracy")
plt.legend()
plt.ylabel("Score", fontsize=16)
plt.xlabel("Max Depth", fontsize=16)
plt.title("Scores for Decision Tree for \nDifferent depth value", fontsi
ze=16)
plt.xticks(depth_list);
```




In [40]:
```
best_model_DTC_train_score = accuracy_score(ytrain, best_model_DTC.predi
ct(xtrain))
best_model_DTC_test_score = accuracy_score(ytest, best_model_DTC.predict
(xtest))
print("Training accuracy is {:.4}%".format(best_model_DTC_train_score*10
0))
print("Test accuracy is {:.4}%".format(best_model_DTC_test_score*100))
```

```
Training accuracy is 96.69%
Test accuracy is 92.53%
```

In [41]:
```
models_list["decision_tree"] = best_model_DTC
filename = 'decision_tree.sav'
pickle.dump(best_model_DTC, open(filename, 'wb'))
acc['dtc'] = (best_model_DTC_train_score, best_model_DTC_test_score )
```

Back to TOC
<a id ='Random-Forest'></a>

#### 4.10 -Random Forest

The Random Forest is giving us the highest accuracy from all the models tested so far on the test data. but we may be able to increase this value with Boosting or Bagging.

In [42]:
```
rf = RandomForestClassifier(max_depth=6)
rf_model = rf.fit(xtrain, ytrain)
rf_train_acc = rf_model.score(xtrain, ytrain)
rf_test_acc = rf_model.score(xtest, ytest)

print("Random Forest Training accuracy is {:.4}%".format(rf_train_acc*10
0))
print("Random Forest Test accuracy is {:.4}%".format(rf_test_acc*100))
```

```
Random Forest Training accuracy is 95.71%
Random Forest Test accuracy is 93.85%
```

In [43]:
```
models_list["random_forest"] = rf_model
filename = 'random_forest.sav'
pickle.dump(rf_model, open(filename, 'wb'))
acc['rf'] = (rf_train_acc, rf_test_acc)
```

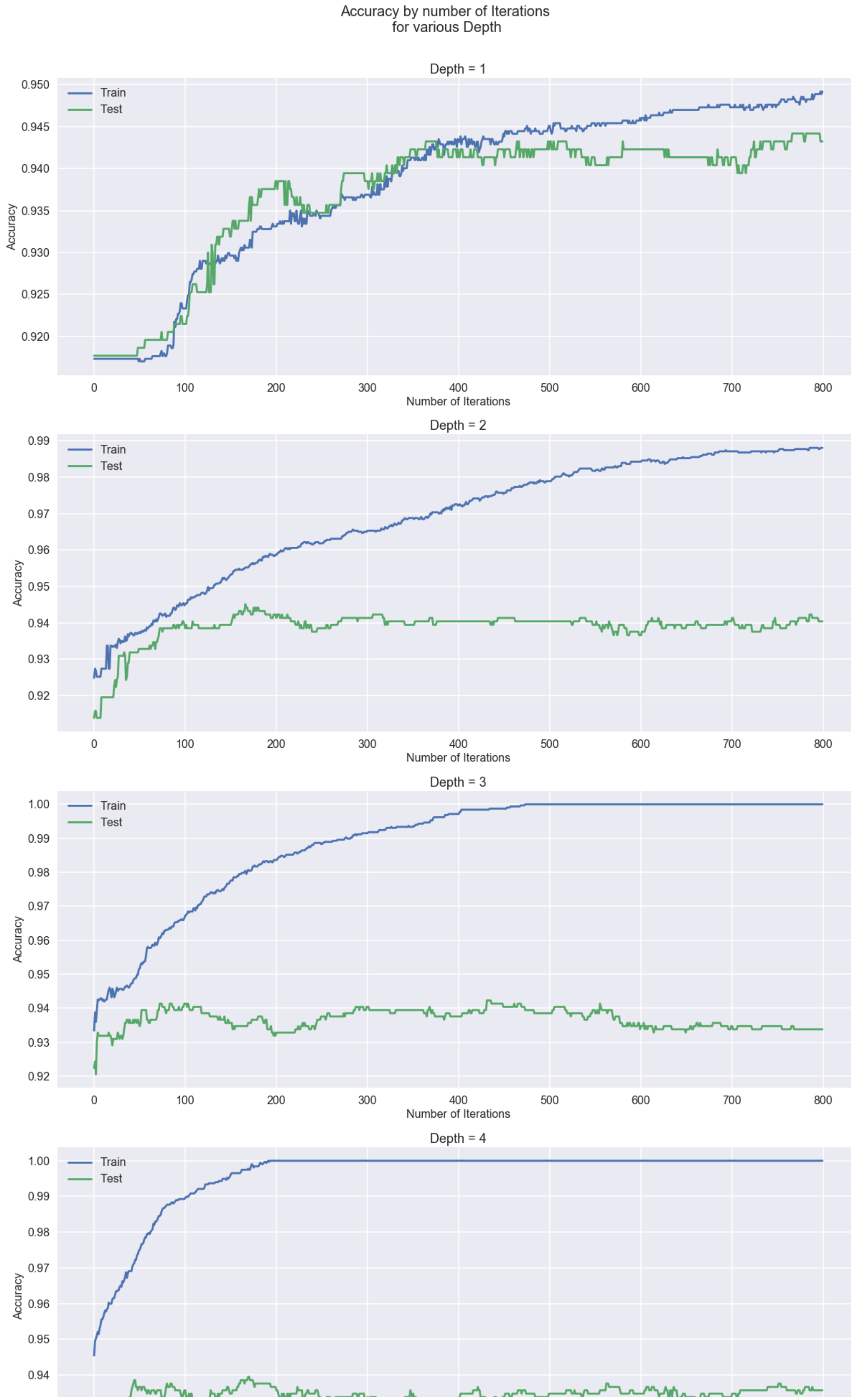
Back to TOC
<a id ='Boosting-AdaBoost-Classifier'></a>

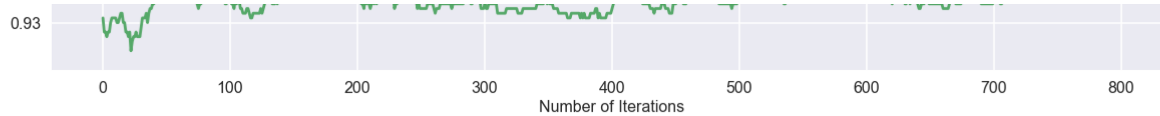
#### 4.11 -Boosting - AdaBoost Classifier

For the model with depth = 1, the accuracy for train and test datasets are close to each other. However, for the models with depth = 2, 3 and 4, there are a big difference in the accuracy for test and train data. I would choose depth =2 and iterations = 180. This model is performing the best so far.

In [44]:
```
AdaBoost_models = {}
AdaBoost_scores_train = {}
AdaBoost_scores_test = {}
for e in range(1, 5):
    AdaBoost = AdaBoostClassifier(DecisionTreeClassifier(max_depth=e), n
_estimators=800, learning_rate=0.05)
    AdaBoost_models[e] = AdaBoost.fit(xtrain, ytrain)
    AdaBoost_scores_train[e] = list(AdaBoost_models[e].staged_score(xtra
in, ytrain))
    AdaBoost_scores_test[e] = list(AdaBoost_models[e].staged_score(xtest
, ytest))
```

In [45]:
```
fig, ax = plt.subplots(4,1, figsize=(20,35))
for e in range(0, 4):
    ax[e].plot(AdaBoost_scores_train[e+1], label='Train')
    ax[e].plot(AdaBoost_scores_test[e+1], label='Test')
    ax[e].set_xlabel('Number of Iterations', fontsize=16)
    ax[e].set_ylabel('Accuracy', fontsize=16)
    ax[e].tick_params(labelsize=16)
    ax[e].legend( fontsize=16)
    ax[e].set_title('Depth = %s'%(e+1), fontsize=18)
fig.suptitle('Accuracy by number of Iterations\n for various Depth',y=0.
92,fontsize=20);
```


Accuracy by number of Iterations
for various Depth
Depth = 1
Train
Test
Accuracy
Number of Iterations
Depth = 2
Train
Test
Accuracy
Number of Iterations
Depth = 3
Train
Test
Accuracy
Number of Iterations
Depth = 4
Train
Test
Accuracy




In [46]:
```
AdaBoost = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2), n_est
imators=800, learning_rate=0.05)
AdaBoost_2 = AdaBoost.fit(xtrain, ytrain)
```

In [47]:
```
models_list["AdaBoost_2"] = AdaBoost_2
filename = 'AdaBoost_2.sav'
pickle.dump(AdaBoost_2, open(filename, 'wb'))
acc['adaboost'] = (AdaBoost_scores_train[2][179], AdaBoost_scores_test[2
][179])
```

Back to TOC
<a id ='SVM'></a>

### 4.12 -SVM

We tried SVM and reached a test accuracy of 93.28%. As it is an expensive model, we ended up using eyeballing to fit a model so we can try the SVM method. However, ideally, we would like to perform a grid search to find te best kernal and c value.

In [48]:
```
# Import the Libraries Needed
from sklearn import svm
from sklearn.model_selection import GridSearchCV

# Load the Data

# Fit a SVM Model by Grid Search
# parameters = {'kernel':('linear','rbf','poly','sigmoid'), 'C':[0.01,0.
1,1,10,100]}
# svc = svm.SVC(random_state=0)
# svm_model = GridSearchCV(svc, parameters, cv=5)
# svm_model.fit(X_train, ytrain)

# Fit a Model by Eyeballing
svm_model = svm.SVC(kernel='poly',C=1,degree=4, random_state=0)
svm_model.fit(xtrain, ytrain)

#models_list = []
models_list["SVM"] =  svm_model
print("Train set score: {0:4.4}%".format(svm_model.score(xtrain, ytrain)
*100))
print("Test set score: {0:4.4}%".format(svm_model.score(xtest, ytest)*10
0))
```

```
Train set score: 94.98%
Test set score: 93.28%
```

In [49]:
```
filename = 'svm.sav'
pickle.dump(svm_model, open(filename, 'wb'))
acc['svm_poly_c1'] = (svm_model.score(xtrain, ytrain), svm_model.score(x
test, ytest))
```

In [50]:
```
# we have finished all our models. we want to save the accuracy score an
d models to json
with open('models.pickle', 'wb') as handle:
    pickle.dump(models_list, handle, protocol=pickle.HIGHEST_PROTOCOL)
acc = pd.DataFrame(acc)
acc.to_json('acc.json')
```

Back to TOC
<a id ='KMeans-Clustering'></a>

#### 4.13 - K-Means Clustering

We want to explore if unsupervised k-means clustering align with bot / non-bot classification.

In [51]:
```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2, init='random', random_state=0).fit(users_d
f[col_pred_numerical].values)
```

In [52]:
```
# add the classification result
k2 = users_df[col_pred_numerical]

k2['k=2'] = kmeans.labels_
```

In [53]:
```
# create df for easy plot
kmean_0 = k2.loc[k2['k=2']==0]
kmean_1 = k2.loc[k2['k=2']==1]
class_0 = users_df.loc[users_df['class_boto']==0]
class_1 = users_df.loc[users_df['class_boto']==1]
```

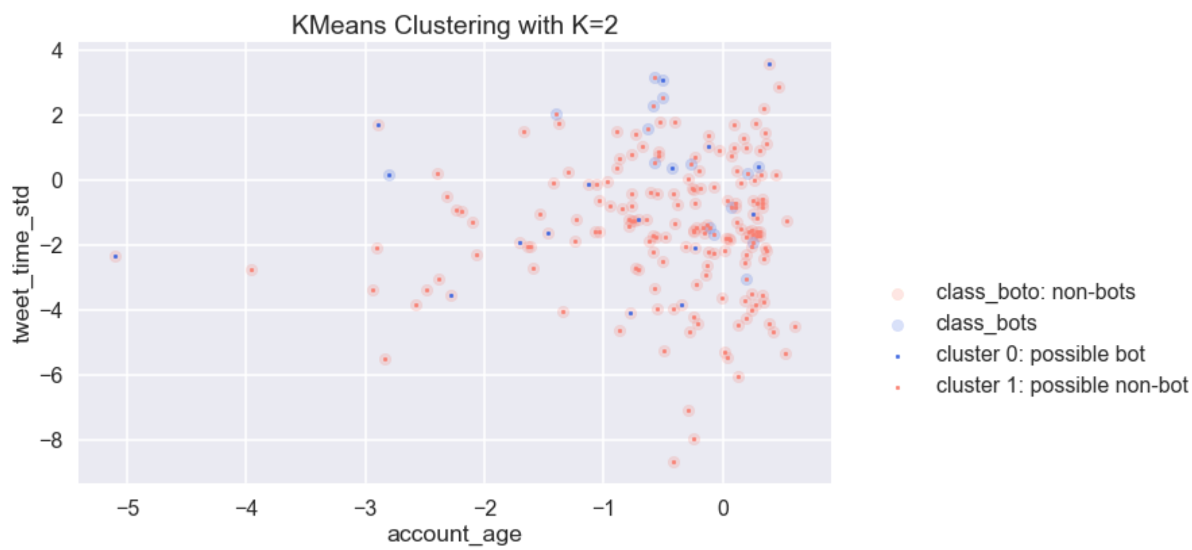
In [54]:
```
# see how many were classified as bots
print ('The size of the two clusters from kmeans clustering are {} and
{}.'.format(len(kmean_0), len(kmean_1)))
```

```
The size of the two clusters from kmeans clustering are 277 and 3949.
```

Given the size of cluster 0, it looks like cluster 0 might be a bot cluster.

We picked two arbitary features to visualize the two clusters from unsupervised KMeans (k=2), and how they align with botometer classification. Visually they align well, and we want to see how many bots are in cluster 0 and non-bots in cluster 1.

In [55]:
```
# quick plot to see if it naturally come into two clusters
plt.figure(figsize=(10,6))
plt.scatter(np.log(class_0['account_age']), np.log(class_0['tweet_time_s
td']), c='salmon', s=70, label = 'class_boto: non-bots', alpha=0.2)
plt.scatter(np.log(class_1['account_age']), np.log(class_1['tweet_time_s
td']), c='royalblue', s=70, label = 'class_bots', alpha=0.2)
plt.scatter(np.log(kmean_0['account_age']), np.log(kmean_0['tweet_time_s
td']), c='royalblue', s=7, label = 'cluster 0: possible bot', alpha=1)
plt.scatter(np.log(kmean_1['account_age']), np.log(kmean_1['tweet_time_s
td']), c='salmon', s=7, label = 'cluster 1: possible non-bot', alpha=1)
plt.xlabel('account_age')
plt.ylabel('tweet_time_std')
plt.title('KMeans Clustering with K=2')
plt.legend(loc='best', bbox_to_anchor=(1, 0., 0.5, 0.5));
```




In [56]:
```
# proportion of cluster 0 users which are bots (precision)
precision_bot_0 = k2[(users_df['class_boto']==1) & (k2['k=2']==0)].shape
[0] / kmean_0.shape[0]
print ('proportion of cluster 0 users which are bots (precision) is {:.2
f}%'.format(precision_bot_0*100))
```

```
proportion of cluster 0 users which are bots (precision) is 36.46%
```

In [57]:
```
# proportion of bots which are in cluster 0 (recall)
recall_bot_0 = k2[(users_df['class_boto']==1) & (k2['k=2']==0)].shape[0]
 / class_1.shape[0]
print ('proportion of bots which are in cluster 0 (recall) is {:.2f}%'.f
ormat(recall_bot_0*100))
```

```
proportion of bots which are in cluster 0 (recall) is 28.94%
```

In [58]:
```
# proportion of cluster 1 users which are bots (precision)
precision_bot_1 = k2[(users_df['class_boto']==1) & (k2['k=2']==1)].shape
[0] / kmean_1.shape[0]
print ('proportion of cluster 1 users which are bots (precision) is {:.2
f}%'.format(precision_bot_1*100))
```

```
proportion of cluster 1 users which are bots (precision) is 6.28%
```

In [59]:
```
# proportion of bots which are in cluster 1 (recall)
recall_bot_1 = k2[(users_df['class_boto']==1) & (k2['k=2']==1)].shape[0]
 / class_1.shape[0]
print ('proportion of bots which are in cluster 0 (recall) is {:.2f}%'.f
ormat(recall_bot_1*100))
```

```
proportion of bots which are in cluster 0 (recall) is 71.06%
```

However, when we look at precision and recall for cluster 0 being bots and cluster 1 being bots, we observed that clusters are not as well aligned with botometer classification as the graph is showing above.

It looks like cluster 0 would a better choice as bot cluster as it has a better precision. Therefore KMeans looks like a promising approach in identifying bots and non-bots with unsupervised learning. KMeans clustering could also be used in supervised learning model as a predictor.

In [60]:
```
filename = 'kmeans.sav'
pickle.dump(kmeans, open(filename, 'wb'))
```

Back to TOC
<a id ='Validate-Botometer-Results'></a>

#### 4.14 - Validate Botometer Results

When comparing botometer scores and manually classified results, we noticed that botometer does not always predict actual bot / non-bot correctly. Therefore we want to compare our verified users with Botometer classifications, and see if we can capture the subspace between botometer results and the manually verified results.

We try to use a random forest to explore the subspace between botometer results and the actual result (manually verified classification). We chose to use non-linear model as we expect the relationship between botometer result and actual result to be non-linear.

We want to train a model with one feature plus botometer score as predictors, and the actual classification as the response. In the principle that the botometer is occasionally accurate, and we want to see under what occasions they are accurate / inaccurate, and therefore to capture the residuals between our predictions (which use botometer score as predictors) and the actual results. (* we chose to features as we want to minimize number of features, given our sample size - manually verified bot account - is only 44)

While the model above improved accuracy from 72.73% to 83.33%, the model is very arbitary especially given that our sample size (44) is very small. However, this is an approach that could potentially be further devloped to improve prediction accuracy, especially to train a model with larger training with imperfect labels, and improve it with a smaller training set with better labels.

In [61]:
```
# load verified bots and nonbots
verify_df = pd.read_csv('boto_verify.csv')[['screen_name', 'class_verifi
ed']]
verify_df = verify_df[~verify_df['class_verified'].isnull()]
```

In [62]:
```
# build a dataframe that has screen_name, class_bot, class_verified, fea
ture 1
# we picked one arbitary features we think will be important
# and see if we can improve botometer's prediction on verified account a
ccuracy using decision tree
feature_1 = 'tweet_time_mean'
verify_df = pd.merge(verify_df, users_df[['class_boto', 'screen_name', f
eature_1]])
```

In [63]:
```
# take a look at data
verify_df.drop(columns=['screen_name']).head(5)
```

Out[63]:

| | class_verified | class_boto | tweet_time_mean |
|---|---|---|---|
| **0** | 1.0 | 1 | -0.074599 |
| **1** | 1.0 | 1 | -0.075464 |
| **2** | 1.0 | 0 | -0.074661 |
| **3** | 1.0 | 0 | -0.075440 |
| **4** | 1.0 | 0 | -0.075345 |

In [64]:
```
# first we want to examine the accuracy of class_boto when cross checkin
g with manually verified classifications
boto_vf_acc = sum(verify_df['class_boto']==verify_df['class_verified'])/
len(verify_df['class_boto'])
print ('The accuracy of Botometer in predicting manually verified classi
fication is {:.2f}%.'.format(boto_vf_acc*100))
```

```
The accuracy of Botometer in predicting manually verified classificatio
n is 71.43%.
```

In [65]:
```
# use features and botometer score to predict the validated score

x_train_vf, x_test_vf, y_train_vf, y_test_vf = train_test_split(verify_d
f[['class_boto', feature_1]],
                                                                  verify_d
f['class_verified'], test_size=0.4, random_state=50)

dtc_vf = DecisionTreeClassifier(max_depth=2).fit(x_train_vf, y_train_vf)
score = dtc_vf.score(x_test_vf, y_test_vf)

print("The accuracy of decision tree model (depth=3) in predicting manua
lly verified classification is {:.2f}%.".format(score*100))
```

```
The accuracy of decision tree model (depth=3) in predicting manually ve
rified classification is 42.86%.
```

Back to TOC
<a id ='Sentence-Embeddings-Clutering-Neural-Networks'></a>

#### 4.15 - Classification of tweets using Sentence Embeddings + Clutering + LDA + Neural Networks

Additionally, we want to explore some models on classifying tweets.

The team wanted to explore for this project how we can read the text tweets to predict whether the tweets are coming from bot or human. First, we found out that the text tweets require data cleansing (by navigating through the tweets). So we took a sample data and performed manual data cleansing by replacing stopwords, special characters, emoji expressions, numbers and we saved the new clean data under cleaned_tweets.txt file. Then we decided to find how the data can be clustered and grouped together, so we have converted textual tweets data into numerical vectors using tensor flow encoder for the conversion and we have used text clustering using K-means (Mini Batch Kmeans). Then we labeled data into two categories (Group A and Group B as Bot and Human), at this stage we didn't manually labeled the data to check which tweets are coming from Bots or human (as this will require checking the records manually), so we just assigned the data to be labeled into two categories randomly as there are only two options a bot or non-bot. Then we build the Classification model using Neural Network on all sample data(Used Keras lib on top of tensor flow) The next step is to test the model on new datasets and checking the tweets content, this model was done to explore new techniques and discuss how we can do NLP on tweets data.

word embedding details https://www.tensorflow.org/tutorials/representation/word2vec (https://www.tensorflow.org/tutorials/representation/word2vec) https://www.tensorflow.org/guide/embedding (https://www.tensorflow.org/guide/embedding) https://www.fer.unizg.hr/_download/repository/TAR-07-WENN.pdf (https://www.fer.unizg.hr/_download/repository/TAR-07-WENN.pdf)

Clustering https://scikit-learn.org/stable/modules/clustering.html#mini-batch-kmeans (https://scikit-learn.org/stable/modules/clustering.html#mini-batch-kmeans) https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html) https://algorithmicthoughts.wordpress.com/2013/07/26/machine-learning-mini-batch-k-means/ (https://algorithmicthoughts.wordpress.com/2013/07/26/machine-learning-mini-batch-k-means/) https://scikit-learn.org/stable/auto_examples/cluster/plot_mini_batch_kmeans.html (https://scikit-learn.org/stable/auto_examples/cluster/plot_mini_batch_kmeans.html)

Classification http://www.zhanjunlang.com/resources/tutorial/Deep%20Learning%20with%20Keras.pdf (http://www.zhanjunlang.com/resources/tutorial/Deep%20Learning%20with%20Keras.pdf) https://machinelearningmastery.com/multi-class-classification-tutorial-keras-deep-learning-library/ (https://machinelearningmastery.com/multi-class-classification-tutorial-keras-deep-learning-library/) https://machinelearningmastery.com/binary-classification-tutorial-with-the-keras-deep-learning-library/ (https://machinelearningmastery.com/binary-classification-tutorial-with-the-keras-deep-learning-library/)

# Sentence Embeddings for Clustering

In [66]:
```
# converting textual data into numerical vectors for clustering; we have
 used tensor flow encoder for the conversion
def build_index(embedding_fun, batch_size, sentences):
    ann = []
    batch_sentences = []
    batch_indexes = []
    last_indexed = 0
    num_batches = 0
    with tf.Session() as sess: #starting tensor session
        sess.run([tf.global_variables_initializer(), tf.tables_initializ
er()])
        with open('cleaned_tweets.txt', 'r') as fr:
            for sindex, sentence in enumerate(fr):
                batch_sentences.append(sentence)
                batch_indexes.append(sindex)

                if len(batch_sentences) == batch_size:
                    context_embed = sess.run(embedding_fun, feed_dict={s
entences: batch_sentences})
                    for index in batch_indexes:
                        ann.append(context_embed[index - last_indexed])
                        batch_sentences = []
                        batch_indexes = []
                    last_indexed += batch_size
                    num_batches += 1
            if batch_sentences:
                context_embed = sess.run(embedding_fun, feed_dict={sente
nces: batch_sentences})
                for index in batch_indexes:
                    ann.append(context_embed[index - last_indexed])
    return ann
```

In [67]:
```
batch_size = 128
embed = hub.Module("https://tfhub.dev/google/universal-sentence-encoder/
2")
sentences = tf.placeholder(dtype=tf.string, shape=[None])
embedding_fun = embed(sentences)
ann = build_index(embedding_fun, batch_size, sentences)
```

```
INFO:tensorflow:Using /var/folders/cd/js4b46vx0rq_2zt5bnm1fblw0000gn/T/
tfhub_modules to cache modules.
INFO:tensorflow:Saver not created because there are no variables in the
graph to restore
```

# Text Clustering using Kmeans

In [68]:
```
#We used Kmeans for clustering the data because data is not labeled
from sklearn.cluster import MiniBatchKMeans
no_clus = 2
```

In [69]:
```
km = MiniBatchKMeans(n_clusters=no_clus, random_state=0, batch_size=1000
)
km = km.fit(ann)
```

In [70]:
```
label_ = km.predict(ann)
```

# Labels Choosen after Cluster Analysis

In [71]:
```
#we can give other labels to tweets after analysing the data but right n
ow our motive is to identify bot & non-bots tweets.
label = ["human","bot"]
```

# Data Preperation for Training Neural Network

In [72]:
```
#preparing the model fior neural netwwork
ds = pd.DataFrame()
for j in range(0,no_clus):
    temp = pd.DataFrame()
    temp = pd.DataFrame(np.array(ann)[np.where(label_ == j)[0]])
    temp['label'] = (label[j])
    ds = pd.concat((ds,temp), ignore_index = True)
ds.head()
```

Out[72]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 0.014465 | -0.041160 | -0.080382 | 0.049232 | -0.072470 | 0.035281 | -0.007432 | -0.023164 |
| **1** | -0.004159 | 0.037407 | 0.010676 | 0.051571 | -0.082445 | 0.042067 | 0.085635 | -0.068073 |
| **2** | 0.018760 | 0.017700 | -0.001465 | 0.026475 | -0.065444 | 0.055917 | 0.083909 | -0.037637 |
| **3** | 0.061679 | 0.001478 | 0.031168 | 0.022693 | -0.015502 | 0.021912 | -0.052999 | 0.000949 |
| **4** | 0.046772 | 0.025765 | 0.018090 | -0.011639 | -0.080991 | -0.020765 | 0.075177 | -0.056908 |

5 rows × 513 columns

In [73]:
```
label_c = len(ds.label.unique())
```

In [74]:
```
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(ds.label)
encoded_Y = encoder.transform(ds.label)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = np_utils.to_categorical(encoded_Y)
X = ds.drop('label',axis=1)

encoder.classes_
```

Out[74]:
```
array(['bot', 'human'], dtype=object)
```

In [75]:
```
import pickle
def save_object(obj, filename):
    with open(filename, 'wb') as output:  # Overwrites any existing fil
e.
        pickle.dump(obj, output, pickle.HIGHEST_PROTOCOL)
save_object(encoder,"encoder.pkl")
```

## NN-Architecture for Multi-Class classification and Training

In [76]:
```
model = Sequential()
model.add(Dense(50, activation='relu', input_dim=512))
model.add(Dense(25, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(label_c, activation='softmax'))

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

model.fit(X,dummy_y, epochs=15, batch_size=64,validation_split=0.15,verb
ose=2,shuffle=True)
```

```
Train on 5241 samples, validate on 926 samples
Epoch 1/15
 - 4s - loss: 0.3245 - acc: 0.9454 - val_loss: 0.0410 - val_acc: 0.9838
Epoch 2/15
 - 0s - loss: 0.0356 - acc: 0.9901 - val_loss: 0.0433 - val_acc: 0.9773
Epoch 3/15
 - 0s - loss: 0.0237 - acc: 0.9920 - val_loss: 0.0218 - val_acc: 0.9924
Epoch 4/15
 - 0s - loss: 0.0200 - acc: 0.9927 - val_loss: 0.0251 - val_acc: 0.9903
Epoch 5/15
 - 0s - loss: 0.0141 - acc: 0.9960 - val_loss: 0.0128 - val_acc: 0.9946
Epoch 6/15
 - 0s - loss: 0.0119 - acc: 0.9966 - val_loss: 0.0274 - val_acc: 0.9881
Epoch 7/15
 - 0s - loss: 0.0107 - acc: 0.9968 - val_loss: 0.0146 - val_acc: 0.9946
Epoch 8/15
 - 0s - loss: 0.0078 - acc: 0.9983 - val_loss: 0.0190 - val_acc: 0.9935
Epoch 9/15
 - 0s - loss: 0.0069 - acc: 0.9983 - val_loss: 0.0251 - val_acc: 0.9914
Epoch 10/15
 - 0s - loss: 0.0046 - acc: 0.9992 - val_loss: 0.0246 - val_acc: 0.9935
Epoch 11/15
 - 0s - loss: 0.0047 - acc: 0.9990 - val_loss: 0.0325 - val_acc: 0.9892
Epoch 12/15
 - 0s - loss: 0.0040 - acc: 0.9996 - val_loss: 0.0232 - val_acc: 0.9946
Epoch 13/15
 - 0s - loss: 0.0032 - acc: 0.9992 - val_loss: 0.0226 - val_acc: 0.9935
Epoch 14/15
 - 0s - loss: 0.0024 - acc: 0.9998 - val_loss: 0.0141 - val_acc: 0.9957
Epoch 15/15
 - 0s - loss: 0.0015 - acc: 1.0000 - val_loss: 0.0210 - val_acc: 0.9957
```

Out[76]:
```
<tensorflow.python.keras.callbacks.History at 0x1c3f5f0908>
```

## Saving Model

In [77]:
```
model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
```

Back to TOC
<a id ='Results-and-Conclusion'></a>

# 5. Results and Conclusion

We did not have a lot of data to train our models, good labelling mechanics to label our data, and most of the users were not bots. However, we were able to train models that performs better than random guessing using the base rate for bots vs non-bots in our samples.

Our best model reached an accuracy of `94%`.

Back to TOC
<a id ='Summary-of-Results'></a>

## 5.1 Summary of Results

#### Model Comparison and Analysis

We chose to use linear regression as our base model although this is a classification problem, we thought we could try with a threshold of 0.5. Our test score was 91.39% which is fair for a base model as first step. To improve the model we have run several models on user account data, and all the models were performing between 91% and 94% on the test data with Adaboost having the higher accuracy but these were done on a small sample datasets as we had manually to verify if accounts are Bots or non-Bots.

Alternatively, We wanted to train a model with one feature plus botometer score as predictors, and the actual classification as the response. While the model above improved accuracy from 72.73% to 83.33%, the model is very arbitary especially given that our sample size (44) is very small. However, this is an approach that could potentially be further devloped to improve prediction accuracy, especially to train a model with larger training with imperfect labels, and improve it with a smaller training set with better labels.

Finally, The team wanted to explore how we can read the text tweets to predict whether the tweets are coming from bot or human. This model was done to explore new techniques and discuss how we can use NLP on tweets data to identify bots and non-bots users.

In [2]:
```
acc = pd.read_json('acc.json')
display(acc)
```

| | bl | ols | ridge | lasso | lm | lm_cv3 | lm_poly3 | knn_17 | c |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.917324 | 0.918586 | 0.919217 | 0.917955 | 0.925529 | 0.925844 | 0.982644 | 0.930577 | 0.9668 |
| **1** | 0.917692 | 0.913907 | 0.919584 | 0.917692 | 0.914853 | 0.919584 | 0.934721 | 0.927152 | 0.9252 |

Back to TOC
<a id ='Noteworthy-Findings'></a>

### 5.2 - Noteworthy Findings

**Botometer Label Accuracy**
We noticed that botometer scores were not always accurate. We were able to improve the botometer score prediction for actual bot / non-bot detection using a simple extended model. As we only have a small number of manually verified samples, the results we got was not perfect. However, there is an improvement could be achieved using this technique with a larger manually verified user dataset.

A generalization of this technique / approch is that it allow us to train a model using a large dataset with imperfect labels, use those predictions to train a model on a smaller dataset with better labels. This ensembled model could achieve an improvement on prediction than using the large dataset or the small datset alone.

We were able to get some promising initial results from an unsupervised KMeans model, which we could investigate further to see if we could avoid the need for using botometer labels. Similar to botometer score, KMeans clustering could also be used to a smaller dataset with manually verified labels to create an ensemble model.

**Class and Imbalance**
Among all the users, vastly majority of them were labeled as real users by botometer, which casued class imbalance in our data and potentially could result in very high accuracy (even if the model may not be that good). We tried to resolve this issue by stratify our data by botometer results, so similar proportion of bots were presented in trian and test set.

One thing we could have done, however, is use sampling to reach 50/50 balance.

**Weights**
Another technique we could have done is to change loss functions to weight errors on bots higher. Similar to fraud detection in practive, we would want to make sure we do not miss any fraud (bots, in our case) as we can always verify fraud/non-fraud (bots/non-bots) with actual legit users (non-bot actual users), and we will get feedback. However, it would be very difficult to do the other way around.

Back to TOC
<a id ='Noteworthy-Findings'></a>

#### 5.3 - Challenges

We have learned a lot during the project, especially on how to get data and performing feature engineering, which took up most of our time and much longer than we anticipated. This is mostly due to the following challenges that we have encountered during the process:

**Memory**
the 6000 * 200 tweets ended up to be a file of almost 7 GB. While we have access to computer with 48 Gb memory, it is still fail to load data some time. Not to mention that it becomes very challneging to run on regular PCs. This could potentially be resolved by only reading the json features that we are interested in. In that case, only part of the files will be read and it is easier for computers to handle. The memory issue is also the result of panda dataframe inefficiency and bad coding habbit (e.g. keep copying files without deleting them).

**Downloading Data with Error**
One common thing we have encoutered quite often during the project is not except errors (tweepy errors, user_timeline errors, etc.), especially when running api. This often leads to a break with only one error and made the data collecting process longer than we expected.

**API Rate Limits**
Collecting many tweets / botometer scores have been time consuming due to API rate limits (both from twitter and botometer). However, we also found that some API pricing could be quite affordable. Regarding the time a paid API will save on a project, we would think of this option next time.

**Data Cleaning**
Data cleaning has been very challenging for this project - especially given the number of features embeded in each tweet, and the large number of missing data, errors, etc. Although sometimes we tried to first test on a small dataset, new errors would often occur when we tried to load a larger set of data.

**Lack of Labelled Data**
As we were not provided with labelled data for this project, we need to find labels by ourselves (using botometer, manual verification, etc.) in order to train and/or evaluate our success. Moreover, while self-claimed bots accounts are easy to identify, often times the bots with malicious intentions would try to pretend to be a normal user, which is franky quite difficult to tell sometimes even by going through all the tweets history and reading user profiles.

**Open Ended Challenge**
Unlike other assignments in the course, which we were provided with identified problems where approaches are clear and straightforward, for this challenge we were given an open ended challenge. Identifying the problem and design the approach have been very interesting but also challenging.

**Feature Engineering**
Feature engineering generated most of the predictors in the dataset we used to train models. We tried to aggregate tweet level data to account level to provide more insights for each user (e.g. more uniformed tweeting time might imply a bot). However, similar to idenfitying the problem, what features to look for, how to extract them, how to execute our plan, have been challenging and time consuming.

**DEBUGGING!**
From code not running, to graphs do not make senses, debugging has always be one of the most challenging part of this proejct. One thing we have found helpful is to debug systematically by breaking down the chunk of code one execution at a time.

Back to TOC
<a id ='Conclusion-and-Future-Work'></a>

**5.3 -Conclusion and Future Work**

We think we need to use both account user data and tweets text data with NLP to best identify bots and non-bots users in the future. We need to identy a bigger dataset manually as bots and non-bots or alternavely use clustering to identify objects in the same group that are more similiar then the other group which is in this case bot and non-bot.

# CS-109A Introduction to Data Science

**Final Project - Milestone 4**
**Project : Machine Learning & Analysis for Twitter Bot Detection**
**Contributors: Group #68: Nisrine Elhauz, Huan Liu, Fayzan Talpur, Samasara Tamrakar**
**Harvard University**
**Fall 2018**

---

<a id ='TOC'></a>

**Table of Contents**

1. Literature Review and Related Work
   6.1 Bots in the Twittersphere
   6.2 How Twitter Bots Help Fuel Political Feuds
   6.3 The spread of low-credibility content by social bots
   6.4 Twitter Topic Modeling by Tweet Aggregation
   6.5 Additional Sources

---

<a id ='Literature-Review-and-Related-Work'></a>

## 6 - Literature Review and Related Work

Before starting modeling our data and exploring different techniques for identidying Twitter bots using tweets data from the Twitter developer API. We have reviewed several literature in this perspective. In general, many different classification models have been already developed adapted to this field, and below are a few highlights.

Back to TOC
<a id ='Bots-in-the-Twittersphere'></a>

#### 6.1 -Bots in the Twittersphere

Stefan Wojcik, "Bots in the Twittersphere" (http://www.pewinternet.org/2018/04/09/bots-in-the-twittersphere/)

In this work, Botometer which is a tool that uses machine learning algorithm were used to identify the tweets account. The botometer gives a score between 0 and 1 for each tweeter account by analysing more than 1000 information about the tweeter account. Then by manually identifying around 300 tweets accounts, it was possible to assign the treshold that classify the tweet accounts as bot or non-bot.

Back to TOC
<a id ='How-Twitter-Bots-Help-Fuel-Political-Feuds'></a>

#### 6.2 - How Twitter Bots Help Fuel Political Feuds

Chris Baraniuk,"How Twitter Bots Help Fuel Political Feuds" (https://www.scientificamerican.com/article/how-twitter-bots-help-fuel-political-feuds/)

In this study, we can see how bots can influence people in the political decision by having bots retweeting messages and trying to give a wrong percepective of the political situation. It seems these bots had influence multiple critical political decision such as U.K.'s “Brexit” referendum and Donald Trump's 2016 campaign. The twitter CEO is working on stopping the bots abuse on Twitter, another test will be seen with the coming US elections.

Back to TOC
<a id ='The-spread-of-low-credibility-content-by-social-bots'></a>

#### 6.3 - The spread of low-credibility content by social bots

Chengcheng Shao et al., "The spread of low-credibility content by social bots" (https://arxiv.org/pdf/1707.07592.pdf)

This paper studies how bots influence the the spreading of misinformation produced by sources that have low-credibility. The study found that bots utilitized specific strategies that proved to be effective in "viral" spreading of non credible content. Such strategies were to mention the names of individuals that are highly influential in tweets that contained or linked to misinforming content. The spreading of the information by influential people creates the false illusion that the content is credible.

Back to TOC
<a id ='Twitter-Topic-Modeling-by-Tweet-Aggregation'></a>

#### 6.4 -Twitter Topic Modeling by Tweet Aggregation

Asbjan Ottesen Steinskogetal et al., "Twitter Topic Modeling by Tweet Aggregation" (http://www.aclweb.org/anthology/W17-0210)

This paper explores the utilization of topic modeling to gain insight into trending topics on twitter. Due to the limitedness of tweet texts, various differerent methods of tweet aggregation have been studied for more effective topic modeling. More specifically, Hashtag aggregation and author aggregation seems to make topic modeling more effective and results in better interpretability than standard topic models.

In [0]:
```
---
```

#### Additional Resources

Back to TOC
<a id ='Additional-Sources'></a>

- #### The tweepy Python library http://www.tweepy.org (http://www.tweepy.org)
- #### Twitter's Developer resources https://developer.twitter.com (https://developer.twitter.com)
  Twitter's Tweet object (https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object)
  Twitter's User object (https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/user-object)
- #### Botometer API API Documentation: https://github.com/IUNetSci/botometer-python (https://github.com/IUNetSci/botometer-python)
  Botometer API Overview: https://market.mashape.com/OSoMe/botometer (https://market.mashape.com/OSoMe/botometer)