**Analyzing Twitter Using Sentiment Analysis and Entity Recognition**

A powerful means of communication that has grown popular in the last decade or so has been microblogging, which is the art of broadcasting short, crisp messages through the internet to a large audience. The leader of the microblogging movement has been Twitter, with millions of tweets (short messages with 140 character text messages with pictures and videos) being sent through its platform by over 300 million monthly active users. Twitter is used across the world by individuals, companies, political parties, celebrities, media, authors, and almost everyone else. The type of messages that are sent varies from mindless chatter to conversational topics to promotional content and news. Unlike traditional media, the popularity of Twitter is due to its brief and instantaneous nature, which allows people to consult or broadcast important pieces of information at a split second at the very moment. Countless breaking and sensational news stories of natural disasters like earthquakes have reached millions of people at the very instant due to the existence of Twitter. A very popular use of Twitter by people is to broadcast their opinion "live" on events they are at or watching such as sports, concerts, television series, or movies. Therefore, Twitter is a great means for those behind these events to gage public perception rapidly and due to the large volume of opinions, exhaustively. To be able to do so, they must apply robust techniques that allow them to gather and store these tweets in large scale and analyze them using relevant techniques.

In this chapter, we have chosen the topic of the football English Premier League, which is one of the most popular football events in the world, to demonstrate the analysis of tweets. We will apply two important semantic algorithms--**Sentiment Analysis** and **Entity Recognition** to understand what the tweets are about (matches, people, players, locations, clubs) and how are the opinions (positive, negative, neutral) being emanated. These are two powerful techniques that can help reveal important information from large amounts of textual data. We will use popular open source modules to show the application of sentiment analysis and entity recognition. Sentiment analysis is an important classification problem in machine learning, so we'll also show you how to implement a sentiment analysis algorithm yourself.

In this chapter, we will cover the following topics:

* Applications of Twitter data
* Sentiment analysis
* Customized sentiment analysis

**Scope and process**

The project and analysis in the chapter will cover the data gathered from the Twitter feeds through the Twitter API. Working with the API, the user has a selection of different endpoints (functionalities). We will focus on two of the most popular: the streaming and the search endpoints (REST API). The first one gives access to real-time data, showing tweets as they are published (in fact the access is to the sample, not all tweets). The latter allows to query historical tweets (up to about a week), based on several criteria, which is more suitable for a static analysis. The following are the steps to gather the data from the Twitter feeds:

* Getting the data
* Data pull
* Data cleaning

Let us take a look at each one in detail.

**Getting the data**

The first step to get the data is to set up the Twitter access from its developer platform with two main methods:

* Getting Twitter API keys
* Connecting to the Twitter API:
  + Streaming API
  + REST API (Search endpoint)

**Getting Twitter API keys**

Firstly, you will need to have a Twitter account and obtain credentials (consumer key, consumer secret, access token, and access secret) on the Twitter developer platform to access the Twitter API, following these steps:

1. Create a Twitter user account.
2. Log in with your Twitter user account at [https://apps.Twitter.com/](https://apps.twitter.com/).
3. Click Create New App.
4. Fill out the form, agree to the terms, and click on Create your Twitter application.
5. Go to the next page, click on the Keys and Access Tokens tab, and copy your API key and API secret. Scroll down and click on Create my access token, and copy your Access token, and Access token secret.

Once, we're ready with the Twitter credentials we can move to the next stages of the analysis:

**Data extraction**

Having all required authorization keys, we can prepare the toolset for data retrieval. The Twitter API gives several ways to extract the data, but we will focus on two main methods:

* Keyword search query to obtain recent, historical tweets
* Streaming facility, to obtain tweets as they are posted

**REST API Search endpoint**

The search query requires the authorized connection to the search endpoint: [https://api.](https://api.twitter.com/1.1/search/tweets.json) [Twitter](https://api.twitter.com/1.1/search/tweets.json) [.com/1.1/search/tweets.json](https://api.twitter.com/1.1/search/tweets.json)

There are many data extraction functionalities of the endpoint, depending on the query format and additional parameters. For our case, we will build the following query:

premier league -filter:retweets AND -filter:replies

Here:

* premier league is a combination of two keywords,
* -filter:retweets lets us get rid of all retweets in the extraction results
* -filter:replies filters all replies to the tweet often unnecessary for the analysis

It is worth noticing that the query construction contains logical operator AND between two -filter statements.

Each endpoint has a set of additional parameters, which can help the user to fine tune the results data. We will use:

* count: 100: To indicate the number of tweets to be retrieved per one call
* lang: 'en': To filter the language of the tweets for English language only
* result\_type: 'recent': To obtain the most recent tweets

All these parameters are optional, but using them we will obtain a much more structured format of data. It saves a lot of work in the phase of data cleaning.

Thus, the full script to call Twitter API and request the data is as follows:

import requests

from requests\_oauthlib import OAuth1

q = 'premier league -filter:retweets AND -filter:replies'

url = 'https://api.Twitter.com/1.1/search/tweets.json' ### url according to Twitter API

pms = {'q' : q, 'count' : 100, 'lang' : 'en', 'result\_type': 'recent'} ### parameters according to Twitter API

auth = OAuth1(consumer\_key, consumer\_secret, access\_token, access\_token\_secret)

res = requests.get(url, params = pms, auth=auth)

If the connection status is OK, we should retrieve a set of documents, which we can convert to json format:

tweets = res.json()

The documents have similar structure with few main blocks and additional information:

Information about tweet:

"\_id" : ObjectId("58ee5317b1ffda423e612be2"), \*\*\*\*\* MongoDB id field

"text" : "Arsene Wenger woes continue as fixture reschedule hands   
Arsenal five matches in 14 days: Arsenal now face five..   
https://t.co/QGyXTwzlhj",

"created\_at" : "Wed Apr 12 15:41:21 +0000 2017",

"favorite\_count" : NumberInt(0),

"favorited" : false,

"id\_str" : "852184875792822272",

"id" : NumberLong(852184875792822272),

"possibly\_sensitive" : false,

"in\_reply\_to\_status\_id" : null,

"lang" : "en",

"in\_reply\_to\_screen\_name" : null,

"in\_reply\_to\_user\_id\_str" : null,

"in\_reply\_to\_status\_id\_str" : null,

"retweeted" : false,

"coordinates" : null,

"geo" : null,

"place" : null,

"truncated" : false,

"is\_quote\_status" : false,

"in\_reply\_to\_user\_id" : null,

"source" : "<a href=\"https://www.socialoomph.com\" rel=\"nofollow\">SocialOomph</a>"

Entities attached:

"entities" : {

"symbols" : [

],

"hashtags" : [

],

"user\_mentions" : [

],

"urls" : [

{

"expanded\_url" : "http://dld.bz/fAZzn",

"url" : "https://t.co/QGyXTwzlhj",

"indices" : [

NumberInt(113),

NumberInt(136)

],

"display\_url" : "dld.bz/fAZzn"

}

]

}

Metadata and retweet information:

"metadata" : {

"iso\_language\_code" : "en",

"result\_type" : "recent"

},

"retweet\_count" : NumberInt(0)

And finally, information about the tweet's author:

"user" : {

"profile\_background\_tile" : false,

"protected" : false,

"entities" : {

"description" : {

"urls" : [

]

}

},

"is\_translator" : false,

"created\_at" : "Tue Jan 10 16:09:32 +0000 2017",

"profile\_banner\_url" :   
 "https://pbs.twimg.com/profile\_banners/818852281953173506/1484064703",

"name" : "Football Mania",

"profile\_use\_background\_image" : false,

"profile\_sidebar\_border\_color" : "000000",

"id\_str" : "818852281953173506",

"id" : NumberLong(818852281953173506),

"followers\_count" : NumberInt(2084),

"notifications" : false,

"utc\_offset" : null,

"favourites\_count" : NumberInt(0),

"profile\_link\_color" : "ABB8C2",

"following" : false,

"follow\_request\_sent" : false,

"profile\_image\_url\_https" :   
 "https://pbs.twimg.com/profile\_images/818852991478484995  
 /d4NtBzxG\_normal.jpg",

"profile\_text\_color" : "000000",

"default\_profile\_image" : false,

"time\_zone" : null,

"profile\_image\_url" :   
 "http://pbs.twimg.com/profile\_images/818852991478484995  
 /d4NtBzxG\_normal.jpg",

"location" : "United Kingdom",

"has\_extended\_profile" : false,

"default\_profile" : false,

"profile\_background\_image\_url\_https" :   
 "https://abs.twimg.com/images/themes/theme1/bg.png",

"profile\_background\_image\_url" :   
 "http://abs.twimg.com/images/themes/theme1/bg.png",

"friends\_count" : NumberInt(4009),

"profile\_sidebar\_fill\_color" : "000000",

"translator\_type" : "none",

"lang" : "en",

"statuses\_count" : NumberInt(31078),

"profile\_background\_color" : "000000",

"screen\_name" : "footymania247",

"description" : "Latest football news from around the   
 world,transfers,fixtures,results and much more 18+ only",

"contributors\_enabled" : false,

"geo\_enabled" : false,

"url" : null,

"listed\_count" : NumberInt(3),

"is\_translation\_enabled" : false,

"verified" : false

}

Loaded JSON becomes a dictionary in Python. Each field can be simply accessed by the following key:

tweets['statuses']

Nested fields can be accessed using multiple keys and lists by numerical index:

tweets['statuses'][0]['text']

# Rate Limits

After successful connection to the API, we have to prepare our scripts for data retrieval. As the API limits data access (Rate Limits), it is necessary to build an efficient workflow.

Twitter allows you to get up to 100 tweets per one call (thus, we set the parameter count to 100). If we want to retrieve more and we need to remember already downloaded tweets' IDs not to extract the same tweets during next calls. This procedure is commonly called **paging**.

The paging procedure is implemented in the script as follows:

pages\_counter = 0

number\_of\_pages = 100  
  
 while pages\_counter < number\_of\_pages:  
 pages\_counter += 1  
 res = requests.get(url, params = pms, auth=auth)  
 print("Connection status: %s" % res.reason)  
 tweets = res.json()  
 ids = [i['id'] for i in tweets['statuses']]   
 # collect ids of all tweets to select min(val)  
 pms['max\_id'] = min(ids) - 1   
 # because it would include and then duplicate  
 collection.insert\_many(tweets['statuses'])

The number\_of\_pages variable indicates how many pages we want to retrieve. In combination with the count parameter, we can calculate the theoretical number of tweets count multiply by number\_of\_pages, assuming that for each call the system will find 100 tweets (which is often not the case).

Extracted tweets must be stored, so we should initialize the connection to MongoDB as follows:

from pymongo import MongoClient

client = MongoClient('mongodb://localhost:27017/')

db = client[database\_name]

collection = db[collection\_name]

In the previous script, we use the insert\_many method in order to store all retrieved tweets in the database.

**Streaming API**

Another method of obtaining information from Twitter is the streaming API. It gives access to Twitter's global stream of data. There are several basic streaming endpoints, each customized to certain use cases. Based on the Twitter documentation:

* **Public streams**: Streams of the public data flowing through Twitter. It is suitable for following specific users or topics, and data mining.
* **User streams**: These are single-user streams, containing roughly all of the data corresponding with a single user's view of Twitter.

In order to connect to the API, we need to use the following endpoint:

url = 'https://stream.Twitter.com/1.1/statuses/filter.json'

The authorization requires the oauth library as follows:

from requests\_oauthlib import OAuth1

auth = OAuth1(consumer\_key, consumer\_secret, access\_token, access\_token\_secret)

Finally, we set up the parameters—tracking keywords and the language:

pms = {'track' : 'premier league -filter:retweets AND -filter:replies', 'lang': 'en'}

The API call should be executed as a POST request with parameters:

res = requests.post(url, auth=auth, params = pms, stream = True)

for line in res.iter\_lines():

if line:

tweet = json.loads(line.decode('utf-8'))

try:

mongo.insert(tweet)

except:

pass

The response comes in a binary format, so we convert it to JSON and save it to the database. This method of getting data may take some time, as tracked tweets are not being posted often. Such workflows enable us to archive a number of queried tweets and prepare a comfortable environment for furthers analysis. However, we have to be aware of the time factor, especially with the Stream API.

# Data pull

Having finished with the data retrieval, we can start the preparation of the data structures for further processing. We will load our data into the dataframe model.

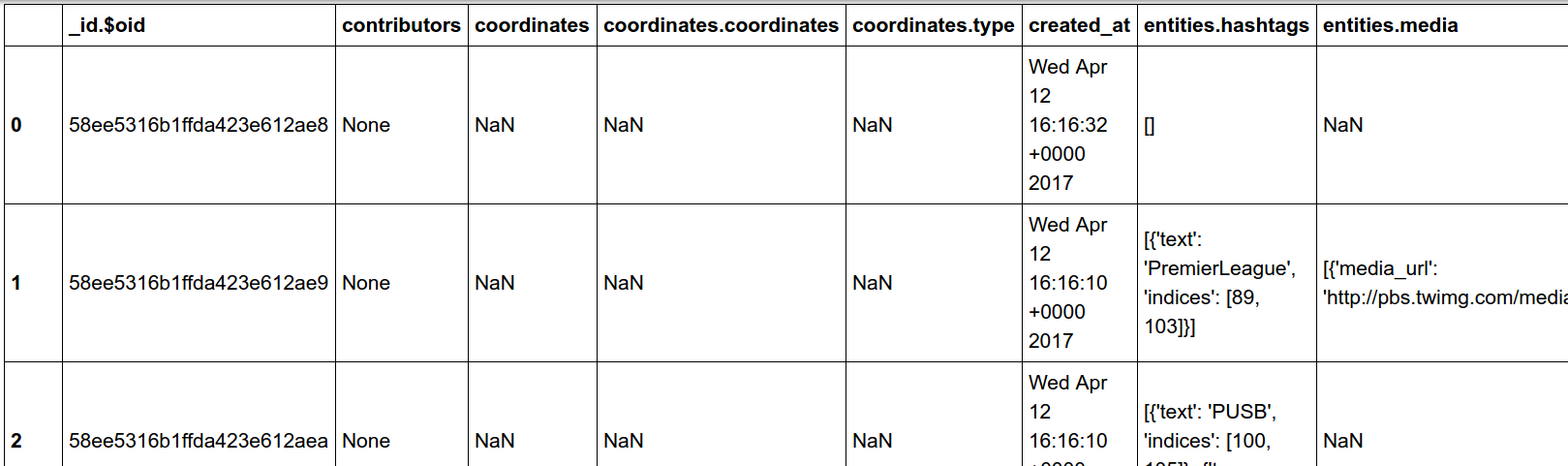
Mongo connection object:

client = MongoClient('mongodb://localhost:27017/')  
db = client['db']  
collection = db['collection']  
  
documents = []  
for doc in collection.find():  
 documents.append(doc)

Now, we can create a dataframe object using our documents list:

df = pd.DataFrame(documents)

So, our data structure is in tabular form, where columns indicate the names of the document nodes and the rows represent the data:



**Data cleaning**

Our goal is to prepare the data in the format which is most appropriate for analysis. It means we need to clean up unnecessary fields and focus only on the relevant parts.

We would like to analyze the sentiments of the tweets that are written by people from their different devices. Therefore, we should get rid of the tweets composed by bots, web pages, automated posting services, and so on. We cannot identify those tweets 100%, but quite a good assumption would be to select tweets posted from physical devices, meaning iPhones, Android phones, desktops, and laptops.

Twitter documents have an interesting feature, because they keep the information about the source of the tweet creation. Whenever someone uses a device to compose a tweet, the information about it is maintained. It can be illustrated by querying our dataframe:

df.source

As a result, we will get a list of the posting methods:

0 <a href="https://ifttt.com" rel="nofollow">IFT...  
1 <a href="https://dlvrit.com/" rel="nofollow">d...  
2 <a href="http://draldevelopment.uk" rel="nofol...  
3 <a href="https://dlvrit.com/" rel="nofollow">d...  
4 <a href="https://dlvrit.com/" rel="nofollow">d...  
5 <a href="http://rightrelevance.com" rel="nofol...  
6 <a href="https://dlvrit.com/" rel="nofollow">d...  
7 <a href="http://bufferapp.com" rel="nofollow">...  
8 <a href="https://dlvrit.com/" rel="nofollow">d...  
9 <a href="http://Twitter.com/download/android" ...  
10 <a href="https://about.Twitter.com/products/tw...  
11 <a href="https://dlvrit.com/" rel="nofollow">d...  
12 <a href="http://www.hootsuite.com" rel="nofoll...  
13 <a href="http://www.facebook.com/Twitter" rel=...  
14 <a href="https://dlvrit.com/" rel="nofollow">d...  
15 <a href="http://www.socialflow.com" rel="nofol...  
16 <a href="https://dlvrit.com/" rel="nofollow">d...  
17 <a href="https://about.Twitter.com/products/tw...  
18 <a href="https://ifttt.com" rel="nofollow">IFT...  
19 <a href="https://ifttt.com" rel="nofollow">IFT...

We see two issues: first—the information is given in HTML format, second—there are many more posting methods, containing not only devices, but also bots and automated services.

Let's address the issues. In order to clean up the HTML code we will use a library called BeautifulSoup, which is used for web crawling for information. We'll use it here to remove the HTML code from the required information:

from bs4 import BeautifulSoup

df['tweet\_source'] = df['source'].apply(lambda x: BeautifulSoup(x).get\_text())

We have created a new column, tweet\_source, where human readable tweet sources are stored:

0 IFTTT  
1 dlvr.it  
2 Coventry City News Twitter  
3 dlvr.it  
4 dlvr.it  
5 RightRelevanceTweetApp  
6 dlvr.it  
7 Buffer  
8 dlvr.it  
9 Twitter for Android  
10 TweetDeck  
11 dlvr.it  
12 Hootsuite  
13 Facebook  
14 dlvr.it  
15 SocialFlow  
16 dlvr.it  
17 TweetDeck  
18 IFTTT  
19 IFTTT

If we scroll down the data, we see the variety of tweet sources. However, we are only interested in devices. The information about devices is very clear—we will find tweet sources like Twitter for iPhone, Twitter for Android, Twitter Web Client, Twitter for BlackBerry, Twitter for Mac, Twitter for Windows, and so on. The names of the devices start with the word "Twitter". We will use this property in addressing our second issue.

We need to create the list of all devices, where the name starts with Twitter:

devices = list(set(df[df['tweet\_source'].str.startswith('Twitter')]['tweet\_source']))

Our devices list looks as follows:

**['Twitter Lite',  
'Twitter Web Client',  
'Twitter Ads',  
'Twitter for BlackBerry',  
'Twitter for Windows Phone',  
'Twitter for Android Tablets',  
'Twitter for Mac',  
'Twitter for iPad',  
'Twitter for Windows',  
'Twitter for BlackBerry®',  
'Twitter for iPhone',  
'Twitter for Android']**

Among the names indicating the devices, we have Twitter Advertising Service, which is not needed in our approach. Thus, we must remove it from our list:

devices.remove('Twitter Ads')

Now, we need to slice our data set by the list:

df = df[df['tweet\_source'].isin(devices)]

Our current dataframe is much smaller, but it contains the tweets composed only on the devices.

When we take a look at our data, mainly the df.text column, it is seen that it contains the tweets not only concerning the English Premier League, but also Premier Leagues from other countries. As we want to concentrate on the English Premier League, we must clean the data from other countries. This issue can be addressed by adding "English" in our initial search query, but that could have caused us to lose the relevant tweets that don't have the word "English" attached. We generally prefer to get the maximum data and then clean it rather than specifying too much at the beginning, in order to avoid losing volumes of data.

After identifying the keywords for cleaning, we implement it to our data frame:

df = df[~df['text'].str.contains("Ghana|ghana|jamaica|Jamaica|Ladbrokes|India|Pakistan|Ghana Premier League|Vijay|Predictions|Egyptian Premier League|cricket|Kings|Caribbean Premier League|@cricbuzz|Cricinfo")]

The statement means that we want to keep all rows in the df['text'] column NOT (~) containing the list of keywords. It is worth noticing that the list of keywords is divided by the logical operator OR (indicated by |).

After the cleaning process, we have a data frame structure with all tweets referring to the English Premier League, composed using the devices (not bots or automated services).

In order to correctly prepare the textual data, some more extra cleaning is required. Literally, we will remove the stopwords and special characters. The process consists of three steps:

1. Text tokenization.
2. Stopwords removal.
3. Special characters removal.

Tokenization:

from nltk.tokenize import TweetTokenizer

df['tokens'] = df['text'].apply(TweetTokenizer().tokenize)

Stopwords:

from nltk.corpus import stopwords

stopwords\_vocabulary = stopwords.words('english')

df['stopwords'] = df['tokens'].apply(lambda x: [i for i in x if i.lower() not in stopwords\_vocabulary])

Special characters and stopwords removal:

import string

punctuations = list(string.punctuation)

df['punctuation'] = df['stopwords'].apply(lambda x: [i for i in x if i not in punctuations])

df['digits'] = df['punctuation'].apply(lambda x: [i for i in x if i[0] not in list(string.digits)])

df['final'] = df['digits'].apply(lambda x: [i for i in x if len(i) > 1])

**Sentiment analysis**

Sentiment analysis involves classifying comments or opinions in text into categories such as "positive" or "negative" often with an implicit category of "neutral". A classic sentiment application would be tracking what people think about different topics. Sentiment analysis in data science and machine learning is also called "opinion mining" or in marketing terminology "voice of the customer". It can be a very useful tool to check the affinity to brands, products, or domains. Sentiment analysis is extremely useful in social media monitoring as it allows us to get an overview of the wider public opinion behind specific topics.

Sentiment analysis also has its limitations and is not to be used as a 100% accurate marker.

As natural language can be very ambiguous with multiple connotations, it's hard if not impossible for machines and algorithms to detect them all. Sentiment analysis basically analyses patterns of words in phrases that are more likely to be positive, negative, or neutral, finally, giving a score on each. This approach is quite effective, but not always accurate on informal language.

For example, in the context of coffee, "hot" or "cold" is neutral, but in the context of people "hot" or "cold" can be positive or negative. Another classic example is the word "sick", in the case of people it's a negative concept, but people use it as a positive expression such as for occasions *(it was a sick party and there were tons of cool people!*), and things like skateboard *(it is a sick board and worth every penny of your hard earned cash*). It gets even harder for algorithms to pick up sentiments that contain humor or sarcasm.

Sentiment analysis is essentially a classification problem in machine learning, which is a mathematical model to classify the tweets into certain categories and it relies on labeled datasets. In our case, the categories will be defined as "positive", "negative", and "neutral". A labeled dataset, also called the training dataset, is a sample of data that is manually tagged with the sentiment from human understanding. Based on these labels of the training dataset, the classifier learns the patterns for positive, negative, and neutral content that it applies then to the actual or test dataset. The accuracy of the algorithm depends heavily on the size of the labeled dataset. The more examples, the better the pattern recognition, hence, the better the algorithm. Very often the classifiers return the continuous measure instead of discrete ones. It means we get the probability that a tweet belongs to a category:

tweet text: { positive: 0.7, neutral: 0.2, negative: 0.1 }

For our analysis, we will use a pre-trained model from the NLTK library.

For our sentiment analysis, we chose a sentiment analyzer called **VADER** (**Valence Aware Dictionary for Sentiment Reasoning**), which is available with Python's NLTK library as follows:

from nltk.sentiment.vader

import SentimentIntensityAnalyzer

VADER was designed for analyzing live streams of social media content. The VADER algorithm outputs sentiment scores to four classes of sentiments:

* neg: Negative
* neu: Neutral
* pos: Positive
* compound: Compound (that is, aggregated score)

from nltk.sentiment.vader import SentimentIntensityAnalyzer

sentiment = SentimentIntensityAnalyzer()

df['sentiment'] = df.text.apply(lambda x: sentiment.polarity\_scores(x)['compound'])

After importing the libraries, we need to instantiate the SentimentIntensityAnalyzer object. The object has an interesting method, polarity\_scores, which takes a text as an argument, and returns a sentiment score.

If we call the method with a text argument:

sentiment.polarity\_scores(text)

It returns the dictionary object:

{'compound': 0.7003, 'neg': 0.0, 'neu': 0.691, 'pos': 0.309}

These are simply the sentiment measures for our text argument. We can focus on specific ones ('pos', 'neg', 'neu') or refer to the compound value, which is a normalized score of all three categories with a value range between -1 and 1. The normalization function uses alpha (15 by default), which approximates maximum expected value:

http://my.safaribooksonline.com/getfile?item=NzRhczI3OHNhODE3ci9kdDUxL3AvMWdpc204OWNlZjB0MTQ3MS03LTItc2ZlYTQ0cy80NnNzYWU4ZHQvZTkucGNuYi1nZmY3MzM2MzM0ZDE0

http://my.safaribooksonline.com/getfile?item=NzRhczI3OHNhODE3ci9kdDUxL3AvMWdpc204OWNlNDd0MTg3YS05LWUtc2RlYTQzcy82MnNzYWVmZnQvM2IucGFuOC1nZWJjMGQ3MTU2MDU5

In our approach, we have created a new column ['sentiment'] where we calculate the compound measure to all ['text'] rows.

Now, we can plot our results. First, we must count the occurrences of the sentiment categories:

pos = len(df[df.sentiment > 0])

neg = len(df[df.sentiment < 0])

neu = len(df[df.sentiment == 0])

In the variables we will store the amount of tweets belonging to the categories:

y = [pos, neu, neg] # vector of y-values

plt.title("Sentiment Analysis")

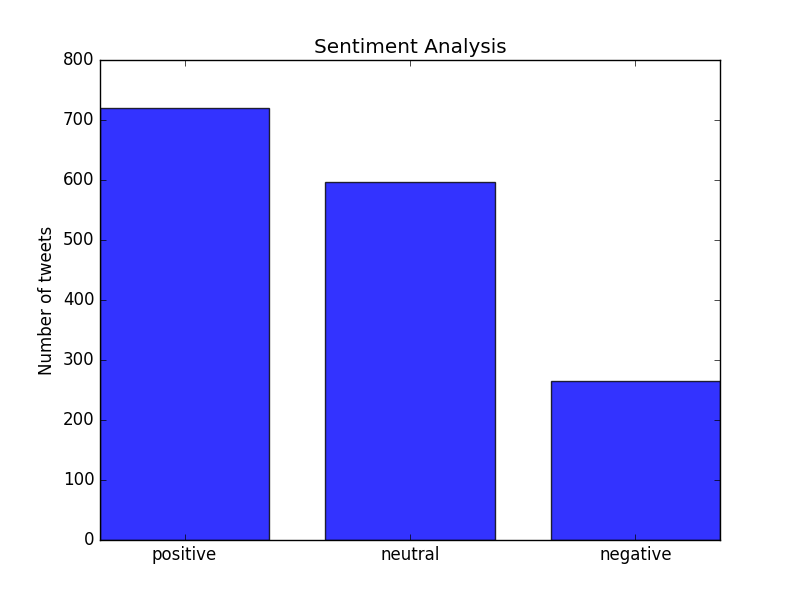
plt.ylabel('Number of tweets')

plt.xticks(range(len(y)), ['positive', 'neutral', 'negative'])

plt.bar(range(len(y)), height=y, width = 0.75, align = 'center', alpha = 0.8)

plt.show()

This gives the following result:



A bar chart like the one in the previous figure is most often a very effective tool for results visualization and interpretation. In general, it shows overall positive affirmation in regard to our topic of the English Premier League. There are more than twice positive tweets than negative and there is a substantial amount of neutral tweets as well.

**Customized sentiment analysis**

As mentioned earlier, sentiment analysis is the process of identifying and extracting sentiment information related to a specified topic, domain, or entity, from a set of documents. The sentiment is identified using trained sentiment classifiers. Thus, the quality and the type of the training data have a big impact on the classifier's performance. Most pre-trained classifiers (like VADER) are trained on general texts because they are designed to be versatile for use on different topics. Unfortunately, when we need to extract sentiment from a specific textual data (for example, very domain specific) such as a general classifier might not perform very well. That is why, it makes great sense to train our own classifier that will fit specific needs, or alternately, just train a general classifier, but based on customized, verified, and known datasets. In short, the magnitude of adaptation to the domain is what makes the difference between a good sentiment analysis and a better one.

There are many sources of datasets available on the internet free of charge, but in extreme cases, we can also prepare our own.

The preparation of a custom classifier requires two data sets:

* **Training data set**: The data on which the classifier algorithm learns the model parameters
* **Test data set**: This is used to determine the accuracy of the algorithm

There is no rule of thumb for selecting training and testing data set sizes, but there is a broad agreement among practitioners that 60-80% of the total data should be training data, and 40-20% should be testing data, respectively.

As it is a supervised learning task, the data (tweets) must be tagged by output categories. In our case, we want to categorize our texts into three classes: positive, neutral, and negative. Thus, each sentence (tweet) should be assigned to one of the classes:

('Kasami vs Palace is the best premier league goal ever by the way', 'pos')

In order to create custom-made classifiers for sentiment analysis, we will use the Python scikit-learn library. The library features multiple machine learning algorithms, among which, we will find regression, classification, or clustering implementations.

Our first classifier will be a simple sentiment analyzer trained on a small dataset of tweets.

To begin, we will import a few elements from the library:

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.naive\_bayes import MultinomialNB

TfidfVectorizer is needed for transforming our data into numerical features usable for the model. It means the text will be represented as numerical data. Next, we select the type of the model for our classifier. Naive Bayes is a simple, yet powerful technique, thus very popular in many prototyping cases. Based on Bayes Theorem, it assumes that every feature contributes independently to the probability of each class (positive, neutral, and negative in our case). This machine learning technique is often used for simple classification tasks such as spam or document classification. It is also a very suitable algorithm for our "bag of words" approach to sentiment classification.

In the next step, we import all necessary libraries:

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

from sklearn.model\_selection import cross\_val\_predict

We also need to import several modules for model evaluation. Evaluating the performance of a model is one of the key stages in the model building process. It indicates how successful the predictions of a dataset have been by a trained model.

# Labeling the data

As explained in the previous section, we have to manually label a training dataset. We assume that the sentiment is specific to the topic of analysis and that a person who labels the data is able to do it correctly. In order to store the labels, we create a column label, which associates a class with a tweet.

The general rule is that the more labeled data the better, but labeling is a costly operation. In order to be efficient we label only observations that are clearly associated with a class. It will help us to get only the best examples of positive, neutral, and negative sentiment. As a result, the algorithm should have a good performance in predicting the most insightful tweets in terms of sentiment, which is the objective. Great...

We create 96 balanced labels, which means that we have around 33% of rows related to each class:

classes = ['pos', 'neu', 'neg']

train\_data = dataset['final'][0:80]

train\_labels = dataset['label'][0:80]

test\_data = dataset['final'][80:96]

test\_labels = dataset['label'][80:96]

train\_data = list(train\_data.apply(' '.join))

test\_data = list(test\_data.apply(' '.join))

We merge all the tokens of each tweet and create a list that will be used for vectorization.

**Creating the model**

Now, we can use the dataset with labeled observations to create a Naive Bayes model:

### Create feature vectors

vectorizer = TfidfVectorizer(min\_df=5,

max\_df = 0.8,

sublinear\_tf=True,

use\_idf=True)

**Parameters**:

* min\_df = 5 discards words appearing in less than five documents
* max\_df = 0.8 discards words appearing in more than 80% of the documents
* sublinear\_tf = True uses sublinear weighting (scale the term frequency in logarithmic scale)
* use\_idf = True enables the inverse document frequency

As a result, we obtain, train, and test vectors that can be directly used to train and validate models:

train\_vectors = vectorizer.fit\_transform(train\_data)  
test\_vectors = vectorizer.transform(test\_data)   
### Perform a logistic regression model, and fit with X and y  
   
nb = MultinomialNB()   
nb.fit(train\_vectors, train\_labels).score(test\_vectors, test\_labels)

**Model performance evaluation and cross-validation**

Once we finished the training phase, we have to validate the performance and check if the predictions are good enough to keep the model. For this purpose, we will use two techniques:

* Confusion matrix
* K-fold cross-validation

**Confusion matrix**

A confusion matrix is a technique for summarizing the performance of a classification algorithm. It provides information of what the classification model is getting right and what types of errors it is making. Predictions of the results on a classification problem are usually visualized by the following matrix:

http://my.safaribooksonline.com/getfile?item=NzRhczI3OHNhODE3ci9kdDUxL3AvMWdpc204OWNlMmV0NmYxOS05LTItczFlYTQzcy82NnNzN2U5YnQvZjcucDFuYS1nMzRjNjlkMGE0ZGFh

For illustration we are using a two-class problem, and we have to select specific outcome from observations and define it as a base case (for example, it rains versus alternative (rejected) no rain). It becomes a reference point for evaluating our model with the test data.

* **True Positive** for correctly predicted values (correct prediction)
* **False Positive** for incorrectly predicted values (incorrect prediction)
* **True Negative** for correctly rejected values (correct prediction)
* **False Negative** for incorrectly rejected values (incorrect prediction)

There are multiple ways to measure the performance of the model. The Key classification accuracy indicator is called precision.

**Precision** is defined as the proportion of positive predictions to the number of observations that are actually positive. We can express it as follows:

http://my.safaribooksonline.com/getfile?item=NzRhczI3OHNhODE3ci9kdDUxL3AvMWdpc204OWNlYjl0YzhkOC1jLTUtczJlYTQwcy85MXNzNGVlMHQvMWMucGJuYS1nOWY0Nzc5YmIwNGU1

**Recall** tells us what is the proportion of actually positive observations predicted as positive:

http://my.safaribooksonline.com/getfile?item=NzRhczI3OHNhODE3ci9kdDUxL3AvMWdpc204OWNlZmR0ZDNlMi00LTUtc2FlYTQzcy8xNHNzNmVjMnQvYTcucGVuYi1nZDQ0ZDZiMTdiMzBm

**F1-score** combines precision and recall to measure the test accuracy. It can be interpreted as weighted average of precision and recall with its best value at 1 and the worst at 0:

http://my.safaribooksonline.com/getfile?item=NzRhczI3OHNhODE3ci9kdDUxL3AvMWdpc204OWNlNTB0YWRhMS1mLWYtc2RlYTQ0cy82MnNzNGUyNXQvYjgucDluOC1nZTU4NDc1YmNiMWM5

**Support** is the number of observations that are predicted in a particular class.

**K-fold cross-validation**

The input data is split into K parts where one is reserved for testing, and the other K-1 for training. This process is repeated K times and the evaluation metrics are averaged. This helps in determining how well a model would generalize to new datasets.

In our example, we have labeled 96 observations in three classes (positive, negative, and neutral). We used 80 as a training set and 16 observations (17%) as a test set. Many tweets are ambiguous for sentiment classification even for human beings. Therefore, we would expect the performance in terms of precision of around 80%.

We have split our tests into three parts:

* Training set 83% - Test set 17%
* Cross validation
* Qualitative verbatim evaluation

print("Naive Bayes")

print(classification\_report(test\_labels, nb.predict(test\_vectors)))

print(confusion\_matrix(test\_labels, nb.predict(test\_vectors)))

predicted = cross\_val\_predict(nb, train\_vectors, train\_labels, cv=10)

print("Cross validation %s" % accuracy\_score(train\_labels, predicted))

The first test showed a precision of 75%, which is acceptable for a dataset with few labels:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Naive Bayes** | **precision** | **recall** | **f1-score** | **support** |
| negative | 0.80 | 0.50 | 0.62 | 8 |
| neutral | 1.00 | 0.20 | 0.33 | 5 |
| positive | 0.20 | 0.67 | 0.31 | 3 |
| avg / total | 0.75 | 0.44 | 0.47 | 16 |

In terms of k-fold cross-validation, we obtained the results of around 73% of precision:

* **Cross validation = 0.7375**: Thus, the human check of the sentiment of the tweets looks very promising. We have extracted some random verbatims to illustrate the results.
* **Positive**: The success of the Premier League, with its record-breaking takings is impacting in Europe <https://t.co/JDulKICszb>
* **Neutral**: Arsenal and Manchester United home fixtures moved <https://t.co/kyg7H1H6BN>, #saintsfc
* **Negative**:Wenger's future at Arsenal plunged into further uncertainty as Palace profit <https://t.co/gvbbdLH9gi>

As you can see, certain verbatims are too ambiguous for even humans to correctly interpret the sentiment, so a perfect sentiment analysis algorithm is unrealistic. In the cases when we analyze content on a specific topic, such as football in this chapter, creating a custom sentiment analysisalgorithm is a good idea. In the case of mixed content, where the topic is not evident, one can use a readily available open source module. However, when building a custom algorithm, it's critical to use validation techniques to be sure of a minimum accuracy.

# Named entity recognition

Now, we arrive at another important concept called the **named entity recognition**, which aims to sort textual content into default categories such as the names of persons, organizations, locations, expressions of time, quantities, monetary values, and so on. The process is also known as entity identification, entity chunking, or entity extraction. This is a very powerful technique to understand large chunks of textual content in an automated manner.

Here, we will use an open source module to demonstrate the concept called Stanford **NER** (**named entity recognizer**), which is a widely used and one of the most popular named entity recognition tools. As Stanford NER is implemented in Java, we'll use the NLTK library, which provides an interface of Stanford NER to be used using Python.

The download is a zipped file (mainly consisting of classifiers). After unpacking, we have all needed files for running under Windows or Unix/Linux/macOS, a simple GUI, and the ability to run as a server. It is worth mentioning that Stanford NER requires Java.

**Installing NER**

Stanford NER requires Java v1.8+ installed on the system:

1. Download Stanford Named Entity Recognizer from the page: <https://nlp.stanford.edu/software/CRF-NER.shtml#Download>
2. Unpack the zipped file. The classifier can be found in the /classifiers/ folder.

The use of Stanford NER Tagger is very straightforward. We put the download classifier into an arbitrary directory and indicate this directory in the StanfordNERTagger object. Then we invoke a tag() method on each tweet to get the results:

from nltk.tag import StanfordNERTagger   
st = StanfordNERTagger('path\_to\_your\_folder/english.all.3class.distsim.crf.ser.gz')   
st.tag(sentence.split())

We will use three class classifiers english.all.3class.distsim that will find three classes of named entities:

* Location
* Person
* Organization

All the words that are not recognized will be labeled as 'O'. We can see an example of tagging in the following list of tuples:

[('Is', 'O'),  
('this', 'O'),  
('like', 'O'),  
('when', 'O'),  
('Premier', 'ORGANIZATION'),  
('League', 'ORGANIZATION'),  
('managers', 'O'),  
('have', 'O'),  
('the', 'O'),  
('full', 'O'),  
('confidence', 'O'),  
('of', 'O'),  
('the', 'O'),  
('board?', 'O'),  
('https:t.coXLcOeLUvpb', 'O')]

The classifier found two words that are related to ORGANIZATION: Premier and League. All the other words were tagged as O.

Then, we can iterate through all tweets to extract named entities and store only entities related to three classes:

for r in tweets:   
 lst\_tags = st.tag(r.split())   
  
for tup in lst\_tags:   
 if(tup[1] != 'O'):   
 entities.append(tup)

Then, we can create a dataframe with all the results:

df\_entities = pd.DataFrame(entities)   
df\_entities.columns = ["word","ner"]

And compute the number of occurrences using the Counter module:

organizations = df\_entities[df\_entities['ner'].str.contains("ORGANIZATION")]   
cnt = Counter(organizations['word'])   
cnt.most\_common(10)

|  |  |
| --- | --- |
| **Word** | **Count** |
| League | 316 |
| Premier | 312 |
| Liverpool | 39 |
| Chelsea | 36 |
| United | 29 |
| Arsenal | 25 |
| Manchester | 21 |
| Palace | 18 |
| Crystal | 17 |
| Everton | 11 |

We can notice that the most frequent ORGANIZATION is Premier League expressed by two words. The next places in our ranking correspond to football teams: Liverpool, Chelsea, United, and Arsenal.

We perform the same extraction for PERSON:

|  |  |
| --- | --- |
| **Word** | **Count** |
| Wenger | 27 |
| Dybala | 21 |
| Chelsea | 19 |
| Arsene | 15 |
| Mourinho | 14 |
| Jorge | 11 |
| Sampaoli | 11 |
| Alexis | 9 |
| Antonio | 9 |
| Carragher | 9 |

As we can see, NER classified Chelsea as both ORGANIZATION and PERSON. Some ambiguous words may appear in all classes.

We perform the same extraction for locations:

|  |  |
| --- | --- |
| **Word** | **Count** |
| City | 20 |
| Leicester | 18 |
| West | 14 |
| Ham | 10 |
| Europe | 6 |
| England | 6 |
| Madrid | 5 |
| Manchester | 5 |
| Spain | 4 |
| Brom | 4 |

The words for locations are quite expected, but there is Madrid (which could also be the club Real Madrid) and Spain along with Leicester and Manchester among the locations mentioned. There's always a scope for ambiguity in named entity recognition where words have various meaning such as Manchester being a location is also part of the club named "Manchester United". The essence is to get all the entities tagged and then with a bit of manual cleaning we can get good insight into the content.

**Combining NER and sentiment analysis**

In order to get insightful information we'll calculate the sentiment for the most frequent entities related to football clubs. We take the three most mentioned clubs and check the mean sentiment for each of them using the np.mean() function from numpy as follows:

subset = dataset[dataset['tweet'].str.contains('Liverpool')]   
avg\_sentiment = np.mean(subset['sentiment'])

We obtain the following results illustrated by some random verbatim:

* **Liverpool 0.1166:** Milner focused on Liverpool results #SSFootball via @SuperSportTV <https://t.co/CIthkFY5Qs>. Juninho says he is delighted Liverpool forward Philippe Coutinho replaced him as the top-scoring Brazilian in the Premier League. African striker on his love for Liverpool. <https://t.co/Mfk6wXWwhf>

Similarly, applying the other two keywords we get the following results:

* **Chelsea 0.2121**: Melo melo@ChelseaFansUSA: Zouma: One of the best memories I have from my time at Chelsea so far was my first goal in the Premier League.... Would be great to see one of Dybala/ Griezzmann in the premier league next season. Hopefully at Chelsea. "Chelsea coach reveals the advantage Antonio Conte's side have in Premier League title race <https://t.co/pYBV7ZrbNp>.
* **Arsenal 0.0135**: Arsenal face exhausting end to Premier League season with five games in 14 days <https://t.co/SPSxrFD5pW> AW will use as excuse for losses. Arsenal are a damaged club decaying by the day but the solution is obvious <https://t.co/88PYKcKI5Z>. Jamie Carragher absolutely destroys Arsenal "cowards" in extraordinary rant <https://t.co/NHuWc0gBPu>.

This example shows that among the most frequent clubs extracted through the Entity Recognition, we conclude that Liverpool and Chelsea are mentioned in an overall positive context, whereas Arsenal in a negative one. You can always extract the original verbatims to understand precisely the reason.

# Summary

Sentiment analysis and entity recognition are two powerful social media analytics techniques to get context around user content. Sports being a sentiment and emotion inciting subject among audiences, for this chapter the dataset we used were tweets using the Twitter API on the English Football Premier League. We used the Twitter REST and Streaming API to collect the data and also applied basic cleaning explained in [Chapter 2](http://my.safaribooksonline.com/9781787121485/0dd97794_03c1_42df_9235_c9eba7daa309_xhtml), Harnessing Social Data - Connecting, Capturing, and Cleaning) and new cleaning methods such as device detection from Twitter API metadata. SentimentAnalysis allows us to categorize text into positive, negative, and neutral categories. We also learnt that there are limitations to sentiment analysis with accuracy, especially in ambiguous expressions. We used the **VADER** (**Valence Aware Dictionary for Sentiment Reasoning**) module from NLTK for sentiment analysis. We also saw that we can build our own sentiment analysis algorithm through machine learning on test and train set datasets. Accuracy of custom sentiment analysis depends heavily on the quality and size of the example or training set. Building and applying our own sentiment analyzer using the Python Scikit Learn library we got an accuracy of around 73%. We applied the cross-validation, confusion matrix, K-Fold, and precision/recall techniques to evaluate the performance of our algorithm.

Entity recognition allows us to categorize textual data into categories such as name, place, organization, and others. This is an efficient method to get a broad understanding on large amounts of social media conversations. We used a Java-based popular entity recognition module, Stanford NER. Using the library on our football dataset allowed us to extract the most frequent clubs, locations, and names being mentioned. We combined Sentiment Analysis and Entity recognition on the chosen dataset by computing sentiments on the entity club detected. Chelsea, Arsenal, and Liverpool being among the most frequent clubs as entities, the application of sentiment analysis on them gave us some insights.

In the next chapter, we will explore data from YouTube to analyze campaigns.