SENTIMENT ANALYSIS OF TWEETS WITH PYTHON, NLTK, WORD2VEC & SCIKIT-LEARN

25 MAY 2017 • MARRRCIN • MACHINE-LEARNING , PYTHON , SENTIMENT-ANALYSIS , TEXT-MINING , SCIKIT-LEARN

Z .NET

HOME ABOUT ME CONTACT

This post describes full machine learning pipeline used for sentiment analysis of twitter posts divided by 3 categories: positive, negative and neutral. For this task I used **python with: scikit-learn, nltk, pandas, word2vec and xgboost** packages.

TL;DR

Detailed description & report of tweets sentiment analysis using machine learning techniques in Python. With details, but this is not a tutorial.

Jupyter Notebook of this post

This post is compiled version of Jupyter Notebook, which you can download here:
https://github.com/marrrcin/ml-twitter-sentiment-analysis

Abstract

The goal of this project was to predict sentiment for the given Twitter post using Python. Sentiment analysis can predict many different emotions attached to the text, but in this report only 3 major were considered: positive,

negative and neutral. The training dataset was small (just over 5900 examples) and the data within it was highly skewed, which greatly impacted on the difficulty of building good classifier. After creating a lot of custom features, utilizing both bag-of-words and word2vec representations and applying the Extreme Gradient Boosting algorithm, the classification accuracy at level of 58% was achieved.

Used Python Libraries

Data was pre-processed using *pandas*, *gensim* and *numpy* libraries and the learning/validating process was built with *scikit-learn*. Plots were created using *plotly*.

```
from collections import Counter
import nltk
import pandas as pd
from emoticons import EmoticonDetector
import re as regex
import numpy as np
import plotly
from plotly import graph_objs
from sklearn.metrics import f1_score, precision_score, recall_score, accuracy_score
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV, Randomize
from time import time
import gensim

# plotly configuration
plotly.offline.init_notebook_mode()
```

Notebook code convention

This report was first prepared as a classical Python project using object oriented programming with maintainability in mind. In order to show this project as a Jupyter Notebook, the classes had to be splitted into multiple code-cells. In order to do so, the classes are suffixed with *_PurposeOfThisSnippet* name and they inherit one from another. The final class will be then run and the results will be shown.

Data source

The input data consisted two CSV files: train.csv (5971 tweets) and test.csv (4000 tweets) - one for training and one for testing. Format of the data was the following (test data didn't contain Category column):

Id	Category	Tweet

Id	Category	Tweet
635930169241374720	neutral	IOS 9 App Transport Security. Mm need to check if my 3rd party network pod supports it

All tweets are in english, so it simplifies the processing and analysis.

Data preprocessing

Loading the data

```
class TwitterData_Initialize():
    data = []
    processed_data = []
    wordlist = []

    data_model = None
    data_labels = None
    is_testing = False

    def initialize(self, csv_file, is_testing_set=False, from_cached=None):
        if from_cached is not None:
            self.data_model = pd.read_csv(from_cached)
            return

        self.is_testing = is_testing_set

        if not is_testing_set:
            self.data = pd.read_csv(csv_file, header=0, names=["id", "emotion", "text"])
            self.data = self.data[self.data["emotion"].isin(["positive", "negative", "neutra
        else:
            self.data = pd.read_csv(csv_file, header=0, names=["id", "text"],dtype={"id":"int
            not_null_text = 1 ^ pd.isnull(self.data["text"])
            not_null_id = 1 ^ pd.isnull(self.data["id"])
            self.data = self.data.loc[not_null_id & not_null_text, :]

        self.processed_data = self.data
```

```
        self.wordlist = []
        self.data_model = None
        self.data_labels = None
```

The code snippet above is prepared, to load the data form the given file for further processing, or just read already preprocessed file from the cache. There's also a distinction between processing testing and training data. As the test.csv file was full of empty entries, they were removed. Additional class properties such as data_model, wordlist etc. will be used further.

```
data = TwitterData_Initialize()
data.initialize("data\\train.csv")
data.processed_data.head(5)
```

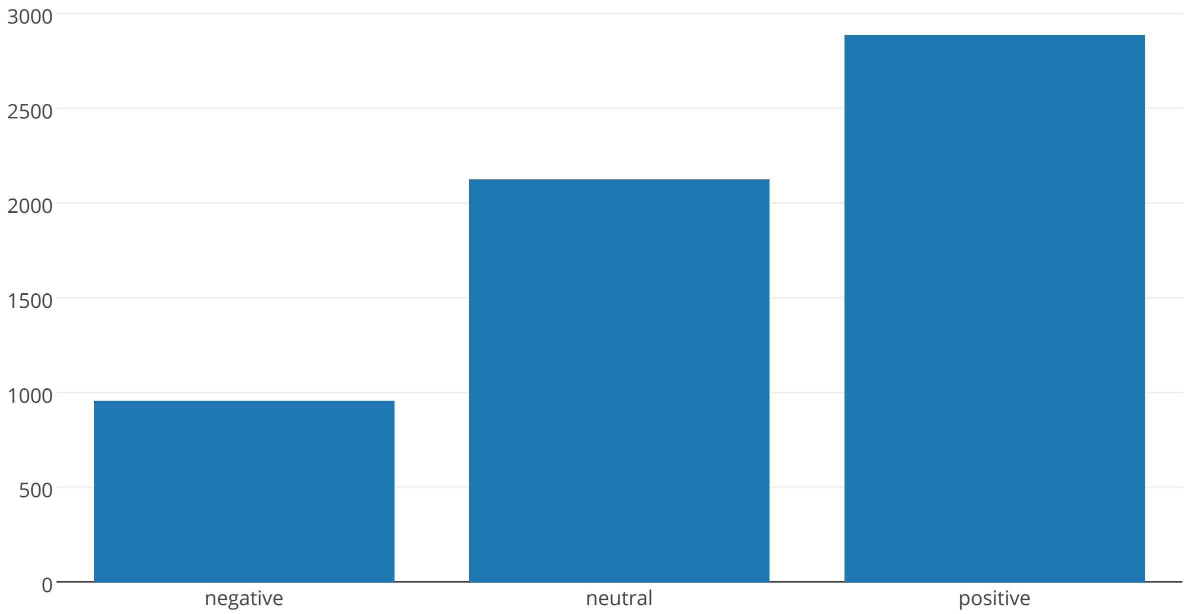
	id	emotion	text
0	635769805279248384	negative	Not Available
1	635930169241374720	neutral	IOS 9 App Transport Security. Mm need to check...
2	635950258682523648	neutral	Mar if you have an iOS device, you should down...
3	636030803433009153	negative	@jimmie_vanagon my phone does not run on lates...
4	636100906224848896	positive	Not sure how to start your publication on iOS?...

Data distribution

First thing that can be done as soon as the data is loaded is to see the data distribution. The training set had the following distribution:

```
df = data.processed_data
neg = len(df[df["emotion"] == "negative"])
pos = len(df[df["emotion"] == "positive"])
neu = len(df[df["emotion"] == "neutral"])
dist = [
    graph_objs.Bar(
        x=["negative","neutral","positive"],
        y=[neg, neu, pos],
)]
plotly.offline.iplot({"data":dist, "layout":graph_objs.Layout(title="Sentiment type distribu
```

Sentiment type distribution in training set



Export to plot.ly »

Preprocessing steps

The targed of the following preprocessing is to create a **Bag-of-Words** representation of the data. The steps will execute as follows:

1. Cleansing
    1. Remove URLs
    2. Remove usernames (mentions)
    3. Remove tweets with *Not Available* text
    4. Remove special characters
    5. Remove numbers
2. Text processing
    1. Tokenize
    2. Transform to lowercase
    3. Stem
3. Build word list for Bag-of-Words

Cleansing

For the purpose of cleansing, the TwitterCleanup class was created. It consists methods allowing to execute all of the tasks show in the list above. Most of those is done using regular expressions. The class exposes it's interface through iterate() method - it yields every cleanup method in proper order.

```
class TwitterCleanuper:
    def iterate(self):
        for cleanup_method in [self.remove_urls,
                               self.remove_usernames,
                               self.remove_na,
                               self.remove_special_chars,
                               self.remove_numbers]:
            yield cleanup_method

    @staticmethod
    def remove_by_regex(tweets, regexp):
        tweets.loc[:, "text"].replace(regexp, "", inplace=True)
        return tweets

    def remove_urls(self, tweets):
        return TwitterCleanuper.remove_by_regex(tweets, regex.compile(r"http.?://[^\s]+[\s]?"

    def remove_na(self, tweets):
        return tweets[tweets["text"] != "Not Available"]

    def remove_special_chars(self, tweets):  # it unrolls the hashtags to normal words
        for remove in map(lambda r: regex.compile(regex.escape(r)), [",", ":", "\"", "=", "&
                                                                     "@", "%", "^", "*", "(",
                                                                     "[", "]", "|", "/", "\\",
                                                                     "!", "?", ".", "'",
                                                                     "--", "---", "#"]):
            tweets.loc[:, "text"].replace(remove, "", inplace=True)
        return tweets

    def remove_usernames(self, tweets):
        return TwitterCleanuper.remove_by_regex(tweets, regex.compile(r"@[^\s]+[\s]?"))

    def remove_numbers(self, tweets):
        return TwitterCleanuper.remove_by_regex(tweets, regex.compile(r"\s?[0-9]+\.?[0-9]*"))
```

The loaded tweets can be now cleaned.

```
class TwitterData_Cleansing(TwitterData_Initialize):
    def __init__(self, previous):
        self.processed_data = previous.processed_data
```

```
    def cleanup(self, cleanuper):
        t = self.processed_data
        for cleanup_method in cleanuper.iterate():
            if not self.is_testing:
                t = cleanup_method(t)
            else:
                if cleanup_method.__name__ != "remove_na":
                    t = cleanup_method(t)

        self.processed_data = t


data = TwitterData_Cleansing(data)
data.cleanup(TwitterCleanuper())
data.processed_data.head(5)
```

	id	emotion	text
1	635930169241374720	neutral	IOS App Transport Security Mm need to check if...
2	635950258682523648	neutral	Mar if you have an iOS device you should downl...
3	636030803433009153	negative	my phone does not run on latest IOS which may ...
4	636100906224848896	positive	Not sure how to start your publication on iOS ...
5	636176272947744772	neutral	Two Dollar Tuesday is here with Forklift Quick...

Tokenization & stemming

For the text processing, nltk library is used. First, the tweets are tokenized using nlkt.word_tokenize and then, stemming is done using **PorterStemmer** as the tweets are 100% in english.

```
class TwitterData_TokenStem(TwitterData_Cleansing):
    def __init__(self, previous):
        self.processed_data = previous.processed_data

    def stem(self, stemmer=nltk.PorterStemmer()):
        def stem_and_join(row):
            row["text"] = list(map(lambda str: stemmer.stem(str.lower()), row["text"]))
            return row

        self.processed_data = self.processed_data.apply(stem_and_join, axis=1)

    def tokenize(self, tokenizer=nltk.word_tokenize):
        def tokenize_row(row):
```

```
            row["text"] = tokenizer(row["text"])
            row["tokenized_text"] = [] + row["text"]
            return row

        self.processed_data = self.processed_data.apply(tokenize_row, axis=1)
```

```
data = TwitterData_TokenStem(data)
data.tokenize()
data.stem()
data.processed_data.head(5)
```

	id	emotion	text	tokenized_text
1	635930169241374720	neutral	[io, app, transport, secur, mm, need, to, chec...	[IOS, App, Transport, Security, Mm, need, to, ...
2	635950258682523648	neutral	[mar, if, you, have, an, io, devic, you, shoul...	[Mar, if, you, have, an, iOS, device, you, sho...
3	636030803433009153	negative	[my, phone, doe, not, run, on, latest, io, whi...	[my, phone, does, not, run, on, latest, IOS, w...
4	636100906224848896	positive	[not, sure, how, to, start, your, public, on, ...	[Not, sure, how, to, start, your, publication,...
5	636176272947744772	neutral	[two, dollar, tuesday, is, here, with, forklif...	[Two, Dollar, Tuesday, is, here, with, Forklif...

Building the wordlist

The wordlist (dictionary) is build by simple count of occurences of every unique word across all of the training dataset.

Before building the final wordlist for the model, let's take a look at the non-filtered version:

```
words = Counter()
for idx in data.processed_data.index:
    words.update(data.processed_data.loc[idx, "text"])

words.most_common(5)

# output:
# [('the', 3744), ('to', 2477), ('i', 1667), ('a', 1620), ('on', 1557)]
```

The most common words (as expected) are the typical english stopwords. We will filter them out, however, as purpose of this analysis is to determine sentiment, words like "not" and "n't" can influence it greatly. Having this in mind, this word will be whitelisted.

```
stopwords=nltk.corpus.stopwords.words("english")
whitelist = ["n't", "not"]
for idx, stop_word in enumerate(stopwords):
    if stop_word not in whitelist:
        del words[stop_word]
words.most_common(5)

# output
# [('may', 1027), ('tomorrow', 764), ('day', 526), ('go', 499), ('thi', 495)]
```

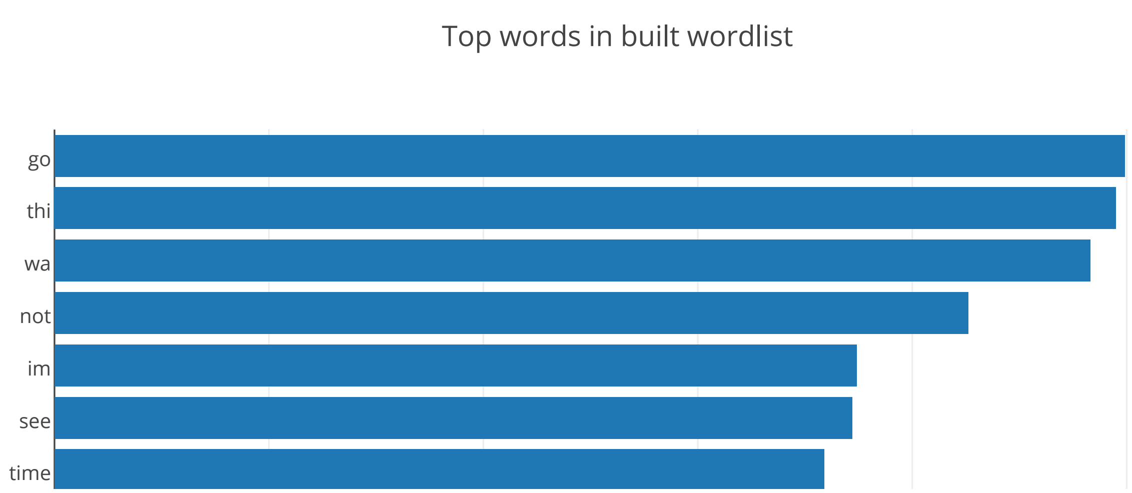
Still, there are some words that seem too be occuring to many times, let's filter them. After some analysis, the lower bound was set to 3.

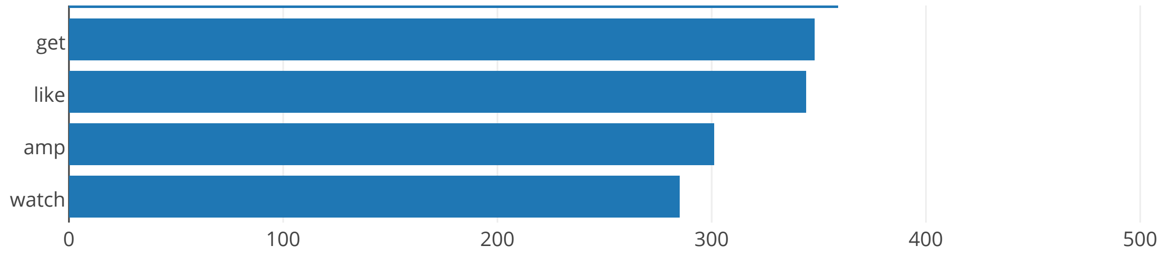
The wordlist is also saved to the csv file, so the same words can be used for the testing set.

```
class TwitterData_Wordlist(TwitterData_TokenStem):
    def __init__(self, previous):
        self.processed_data = previous.processed_data

    whitelist = ["n't","not"]
    wordlist = []

    def build_wordlist(self, min_occurrences=3, max_occurences=500, stopwords=nltk.corpus.sto
                       whitelist=None):
        self.wordlist = []
        whitelist = self.whitelist if whitelist is None else whitelist
        import os
        if os.path.isfile("data\\wordlist.csv"):
            word_df = pd.read_csv("data\\wordlist.csv")
            word_df = word_df[word_df["occurrences"] > min_occurrences]
            self.wordlist = list(word_df.loc[:, "word"])
            return

        words = Counter()
        for idx in self.processed_data.index:
            words.update(self.processed_data.loc[idx, "text"])
```

```
        for idx, stop_word in enumerate(stopwords):
            if stop_word not in whitelist:
                del words[stop_word]

        word_df = pd.DataFrame(data={"word": [k for k, v in words.most_common() if min_occurre
                                     "occurrences": [v for k, v in words.most_common() if min_
                                columns=["word", "occurrences"])

        word_df.to_csv("data\\wordlist.csv", index_label="idx")
        self.wordlist = [k for k, v in words.most_common() if min_occurrences < v < max_occure
```

```
data = TwitterData_Wordlist(data)
data.build_wordlist()


words = pd.read_csv("data\\wordlist.csv")
x_words = list(words.loc[0:10,"word"])
x_words.reverse()
y_occ = list(words.loc[0:10,"occurrences"])
y_occ.reverse()

dist = [
    graph_objs.Bar(
        x=y_occ,
        y=x_words,
        orientation="h"
)]
plotly.offline.iplot({"data":dist, "layout":graph_objs.Layout(title="Top words in built word
```





Export to plot.ly »

Bag-of-words

The data is ready to transform it to bag-of-words representation.

```
class TwitterData_BagOfWords(TwitterData_Wordlist):
    def __init__(self, previous):
        self.processed_data = previous.processed_data
        self.wordlist = previous.wordlist

    def build_data_model(self):
        label_column = []
        if not self.is_testing:
            label_column = ["label"]

        columns = label_column + list(
            map(lambda w: w + "_bow",self.wordlist))
        labels = []
        rows = []
        for idx in self.processed_data.index:
            current_row = []

            if not self.is_testing:
                # add label
                current_label = self.processed_data.loc[idx, "emotion"]
                labels.append(current_label)
                current_row.append(current_label)

            # add bag-of-words
            tokens = set(self.processed_data.loc[idx, "text"])
            for _, word in enumerate(self.wordlist):
                current_row.append(1 if word in tokens else 0)

            rows.append(current_row)
```

```
        self.data_model = pd.DataFrame(rows, columns=columns)
        self.data_labels = pd.Series(labels)
        return self.data_model, self.data_labels
```

Let's take a look at the data and see, which words are the most common for particular sentiments.

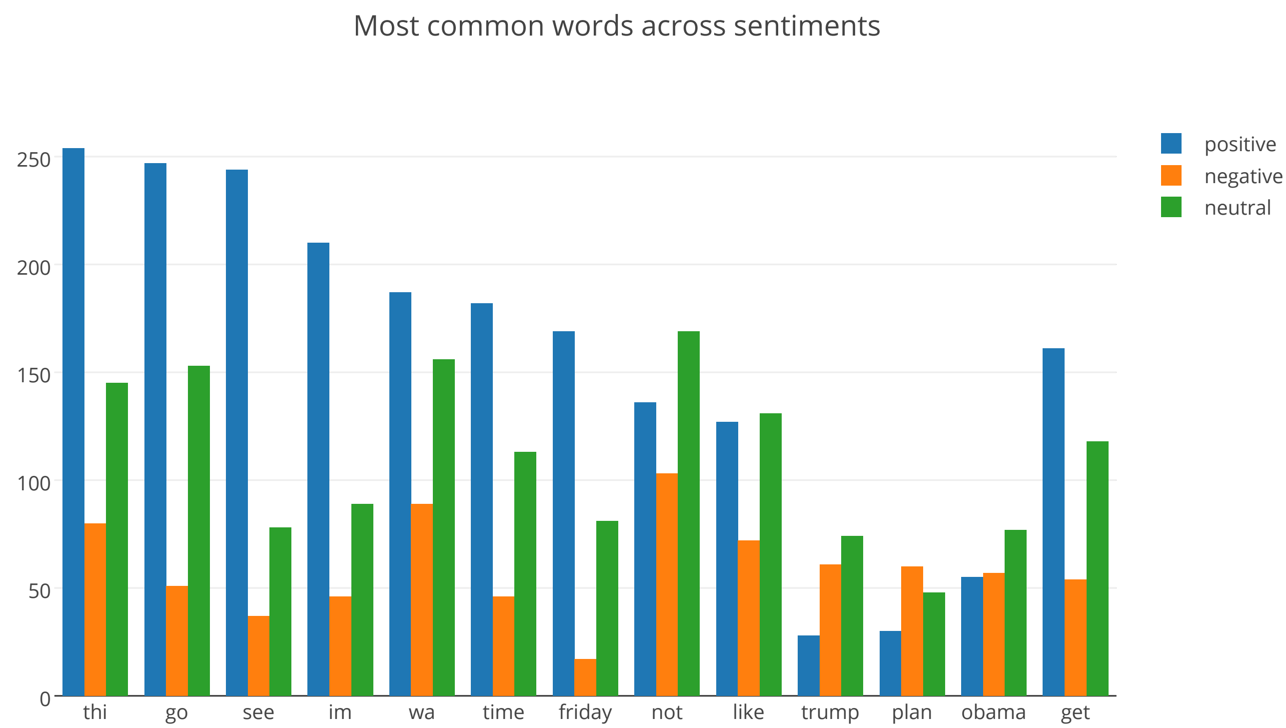
```
data = TwitterData_BagOfWords(data)
bow, labels = data.build_data_model()
bow.head(5)
```

	label	go_bow	thi_bow	wa_bow	not_bow	im_bow	...	sole_bow	rafe_bow	nc_bow
0	neutral	0	0	0	0	0	...	0	0	0
1	neutral	0	0	0	0	0	...	0	0	0
2	negative	0	0	1	1	0	...	0	0	0
3	positive	0	0	0	1	0	...	0	0	0
4	neutral	0	0	0	0	0	...	0	0	0

5 rows × 2185 columns

```
grouped = bow.groupby(["label"]).sum()
words_to_visualize = []
sentiments = ["positive","negative","neutral"]
#get the most 7 common words for every sentiment
for sentiment in sentiments:
    words = grouped.loc[sentiment,:]
    words.sort_values(inplace=True,ascending=False)
    for w in words.index[:7]:
        if w not in words_to_visualize:
            words_to_visualize.append(w)


#visualize it
plot_data = []
for sentiment in sentiments:
    plot_data.append(graph_objs.Bar(
            x = [w.split("_")[0] for w in words_to_visualize],
            y = [grouped.loc[sentiment,w] for w in words_to_visualize],
            name = sentiment
    ))
```

```
plotly.offline.iplot({
        "data":plot_data,
        "layout":graph_objs.Layout(title="Most common words across sentiments")
    })
```



Export to plot.ly »

Some of the most common words show high distinction between classes like *go* and *see* and other are occuring in similiar amount for every class (*plan*, *obama*).

None of the most common words is unique to the negative class. At this point, it's clear that skewed data distribution will be a problem in distinguishing negative tweets from the others.

Classification

First of all, lets establish seed for random numbers generators.

```
import random
seed = 666
random.seed(seed)
```

The following utility function will train the classifier and show the F1, precision, recall and accuracy scores.

```
def test_classifier(X_train, y_train, X_test, y_test, classifier):
    log("")
    log("===============================================")
    classifier_name = str(type(classifier).__name__)
    log("Testing " + classifier_name)
    now = time()
    list_of_labels = sorted(list(set(y_train)))
    model = classifier.fit(X_train, y_train)
    log("Learing time {0}s".format(time() - now))
    now = time()
    predictions = model.predict(X_test)
    log("Predicting time {0}s".format(time() - now))

    precision = precision_score(y_test, predictions, average=None, pos_label=None, labels=list_
    recall = recall_score(y_test, predictions, average=None, pos_label=None, labels=list_of_lal
    accuracy = accuracy_score(y_test, predictions)
    f1 = f1_score(y_test, predictions, average=None, pos_label=None, labels=list_of_labels)
    log("=================== Results ===================")
    log("            Negative     Neutral     Positive")
    log("F1       " + str(f1))
    log("Precision" + str(precision))
    log("Recall   " + str(recall))
    log("Accuracy " + str(accuracy))
    log("===============================================")

    return precision, recall, accuracy, f1

def log(x):
    #can be used to write to log file
    print(x)
```

Experiment 1: BOW + Naive Bayes

It is nice to see what kind of results we might get from such simple model. The bag-of-words representation is binary, so Naive Bayes Classifier seems like a nice algorithm to start the experiments.

The experiment will be based on 7:3 train:test stratified split.

```
from sklearn.naive_bayes import BernoulliNB
X_train, X_test, y_train, y_test = train_test_split(bow.iloc[:, 1:], bow.iloc[:, 0],
                                                    train_size=0.7, stratify=bow.iloc[:, 0],
                                                    random_state=seed)
precision, recall, accuracy, f1 = test_classifier(X_train, y_train, X_test, y_test, BernoulliNI
```

```
==============================================
Testing BernoulliNB
Learing time 0.451092004776001s
Predicting time 0.1190040111541748s
=================== Results ===================
            Negative     Neutral     Positive
F1       [ 0.38949672  0.45072993  0.71369782]
Precision[ 0.45408163  0.48431373  0.65906623]
Recall   [ 0.34099617  0.42150171  0.77820513]
Accuracy 0.579594345421
==============================================
```

Result with accuracy at level of 58% seems to be quite nice result for such basic algorithm like Naive Bayes (having in mind that random classifier would yield result of around 33% accuracy). This performance may not hold for the final testing set. In order to see how the NaiveBayes performs in more general cases, 8-fold crossvalidation is used. The 8 fold is used, to optimize speed of testing on my 8-core machine.

```
def cv(classifier, X_train, y_train):
    log("===============================================")
    classifier_name = str(type(classifier).__name__)
    now = time()
    log("Crossvalidating " + classifier_name + "...")
    accuracy = [cross_val_score(classifier, X_train, y_train, cv=8, n_jobs=-1)]
    log("Crosvalidation completed in {0}s".format(time() - now))
    log("Accuracy: " + str(accuracy[0]))
    log("Average accuracy: " + str(np.array(accuracy[0]).mean()))
    log("===============================================")
    return accuracy


nb_acc = cv(BernoulliNB(), bow.iloc[:,1:], bow.iloc[:,0])
```

```
==============================================
Crossvalidating BernoulliNB...
Crosvalidation completed in 4.4375975131988525s
Accuracy: [ 0.54639175  0.48820059  0.28023599  0.31415929  0.32743363  0.50073855
  0.47119645  0.53106509]
```

```
Average accuracy: 0.432427668406
===============================================
```

This result no longer looks optimistic. For some of the splits, Naive Bayes classifier showed performance below the performance of random classifier.

Additional features

In order to **not** push any other aglorithm to the limit on the current data model, let's try to add some features that might help to classify tweets.

A common sense suggest that special characters like exclamation marks and the casing might be important in the task of determining the sentiment. The following features will be added to the data model:

Feature name	Explanation
Number of uppercase	people tend to express with either positive or negative emotions by using A LOT OF UPPERCASE WORDS
Number of !	exclamation marks are likely to increase the strength of opinion
Number of ?	might distinguish neutral tweets - seeking for information
Number of positive emoticons	positive emoji will most likely not occur in the negative tweets
Number of negative emoticons	inverse to the one above
Number of ...	commonly used in commenting something
Number of quotations	same as above
Number of mentions	sometimes people put a lot of mentions on positive tweets, to share something good
Number of hashtags	just for the experiment
Number of urls	similiar to the number of mentions

Extraction of those features must be done before any preprocessing happens.
For the purpose of emoticons, the EmoticonDetector class is created. The file emoticons.txt contains list of positive and negative emoticons, which are used.

```
class EmoticonDetector:
    emoticons = {}

    def __init__(self, emoticon_file="data\\emoticons.txt"):
        from pathlib import Path
        content = Path(emoticon_file).read_text()
        positive = True
        for line in content.split("\n"):
            if "positive" in line.lower():
```

```
                positive = True
                continue
            elif "negative" in line.lower():
                positive = False
                continue

            self.emoticons[line] = positive

    def is_positive(self, emoticon):
        if emoticon in self.emoticons:
            return self.emoticons[emoticon]
        return False

    def is_emoticon(self, to_check):
        return to_check in self.emoticons


class TwitterData_ExtraFeatures(TwitterData_Wordlist):
    def __init__(self):
        pass

    def build_data_model(self):
        extra_columns = [col for col in self.processed_data.columns if col.startswith("number_
        label_column = []
        if not self.is_testing:
            label_column = ["label"]

        columns = label_column + extra_columns + list(
            map(lambda w: w + "_bow",self.wordlist))

        labels = []
        rows = []
        for idx in self.processed_data.index:
            current_row = []

            if not self.is_testing:
                # add label
                current_label = self.processed_data.loc[idx, "emotion"]
                labels.append(current_label)
                current_row.append(current_label)

            for _, col in enumerate(extra_columns):
                current_row.append(self.processed_data.loc[idx, col])
```

```
            # add bag-of-words
            tokens = set(self.processed_data.loc[idx, "text"])
            for _, word in enumerate(self.wordlist):
                current_row.append(1 if word in tokens else 0)

            rows.append(current_row)

        self.data_model = pd.DataFrame(rows, columns=columns)
        self.data_labels = pd.Series(labels)
        return self.data_model, self.data_labels

    def build_features(self):
        def count_by_lambda(expression, word_array):
            return len(list(filter(expression, word_array)))

        def count_occurences(character, word_array):
            counter = 0
            for j, word in enumerate(word_array):
                for char in word:
                    if char == character:
                        counter += 1

            return counter

        def count_by_regex(regex, plain_text):
            return len(regex.findall(plain_text))

        self.add_column("splitted_text", map(lambda txt: txt.split(" "), self.processed_data[

        # number of uppercase words
        uppercase = list(map(lambda txt: count_by_lambda(lambda word: word == word.upper(), t
                             self.processed_data["splitted_text"]))
        self.add_column("number_of_uppercase", uppercase)

        # number of !
        exclamations = list(map(lambda txt: count_occurences("!", txt),
                                self.processed_data["splitted_text"]))

        self.add_column("number_of_exclamation", exclamations)

        # number of ?
        questions = list(map(lambda txt: count_occurences("?", txt),
```

```
                              self.processed_data["splitted_text"]))

        self.add_column("number_of_question", questions)

        # number of ...
        ellipsis = list(map(lambda txt: count_by_regex(regex.compile(r"\.\s?\.\s?\."), txt),
                            self.processed_data["text"]))

        self.add_column("number_of_ellipsis", ellipsis)

        # number of hashtags
        hashtags = list(map(lambda txt: count_occurences("#", txt),
                            self.processed_data["splitted_text"]))

        self.add_column("number_of_hashtags", hashtags)

        # number of mentions
        mentions = list(map(lambda txt: count_occurences("@", txt),
                            self.processed_data["splitted_text"]))

        self.add_column("number_of_mentions", mentions)

        # number of quotes
        quotes = list(map(lambda plain_text: int(count_occurences("'", [plain_text.strip("'")
                                                 count_occurences('"', [plain_text.strip('"').
                          self.processed_data["text"]))

        self.add_column("number_of_quotes", quotes)

        # number of urls
        urls = list(map(lambda txt: count_by_regex(regex.compile(r"http.?://[^\s]+[\s]?"), tx
                        self.processed_data["text"]))

        self.add_column("number_of_urls", urls)

        # number of positive emoticons
        ed = EmoticonDetector()
        positive_emo = list(
            map(lambda txt: count_by_lambda(lambda word: ed.is_emoticon(word) and ed.is_posit
                self.processed_data["splitted_text"]))

        self.add_column("number_of_positive_emo", positive_emo)
```

```
        # number of negative emoticons
        negative_emo = list(map(
            lambda txt: count_by_lambda(lambda word: ed.is_emoticon(word) and not ed.is_posit
            self.processed_data["splitted_text"]))

        self.add_column("number_of_negative_emo", negative_emo)

    def add_column(self, column_name, column_content):
        self.processed_data.loc[:, column_name] = pd.Series(column_content, index=self.process
```

```
data = TwitterData_ExtraFeatures()
data.initialize("data\\train.csv")
data.build_features()
data.cleanup(TwitterCleanuper())
data.tokenize()
data.stem()
data.build_wordlist()
data_model, labels = data.build_data_model()
```

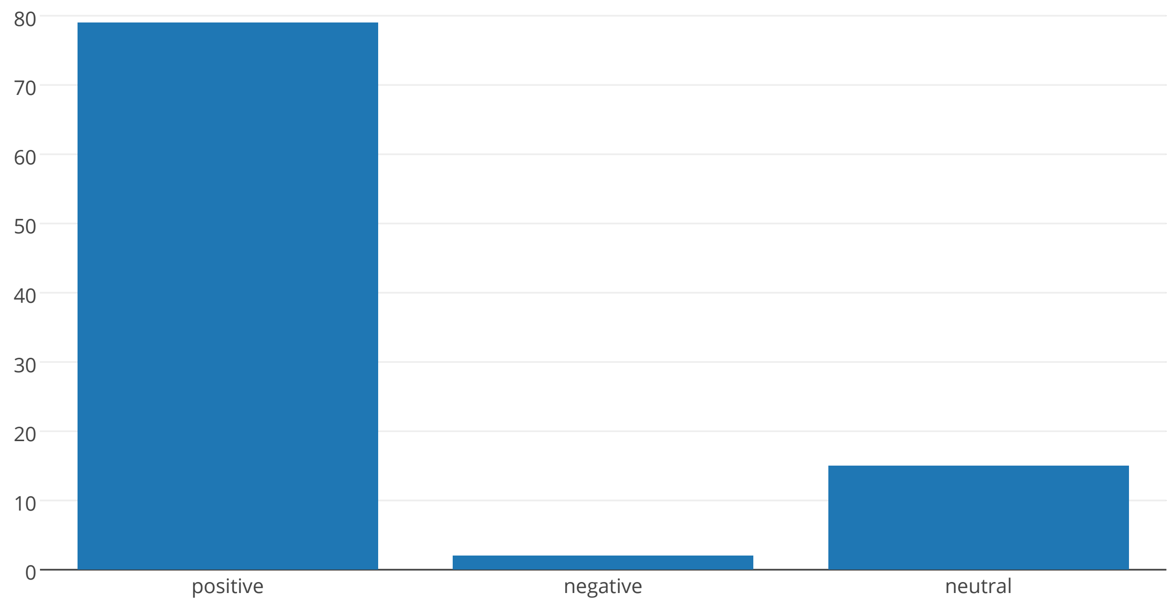
Logic behind extra features

Let's see how (some) of the extra features separate the data set. Some of them, i.e number exclamation marks, number of pos/neg emoticons do this really well. Despite of the good separation, those features sometimes occur only on small subset of the training dataset.

```
sentiments = ["positive","negative","neutral"]
plots_data_ef = []
for what in map(lambda o: "number_of_"+o,["positive_emo","negative_emo","exclamation","has
    ef_grouped = data_model[data_model[what]>=1].groupby(["label"]).count()
    plots_data_ef.append({"data":[graph_objs.Bar(
            x = sentiments,
            y = [ef_grouped.loc[s,:][0] for s in sentiments],
    )], "title":"How feature \""+what+"\" separates the tweets"})


for plot_data_ef in plots_data_ef:
    plotly.offline.iplot({
            "data":plot_data_ef["data"],
            "layout":graph_objs.Layout(title=plot_data_ef["title"])
```

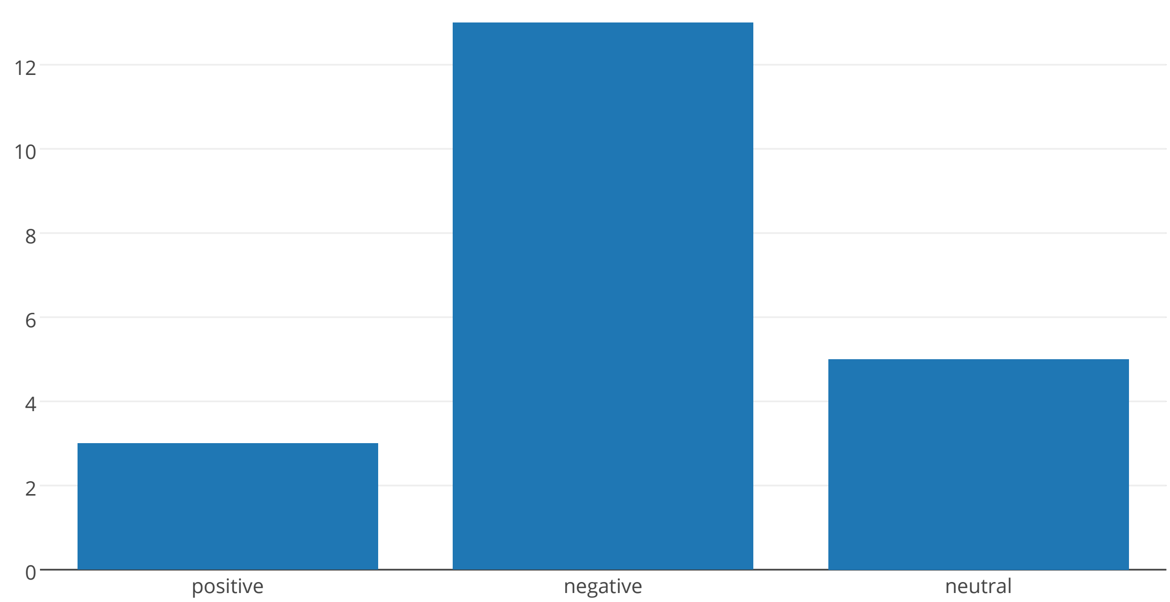
```
})
```

How feature "number_of_positive_emo" separates the tweets



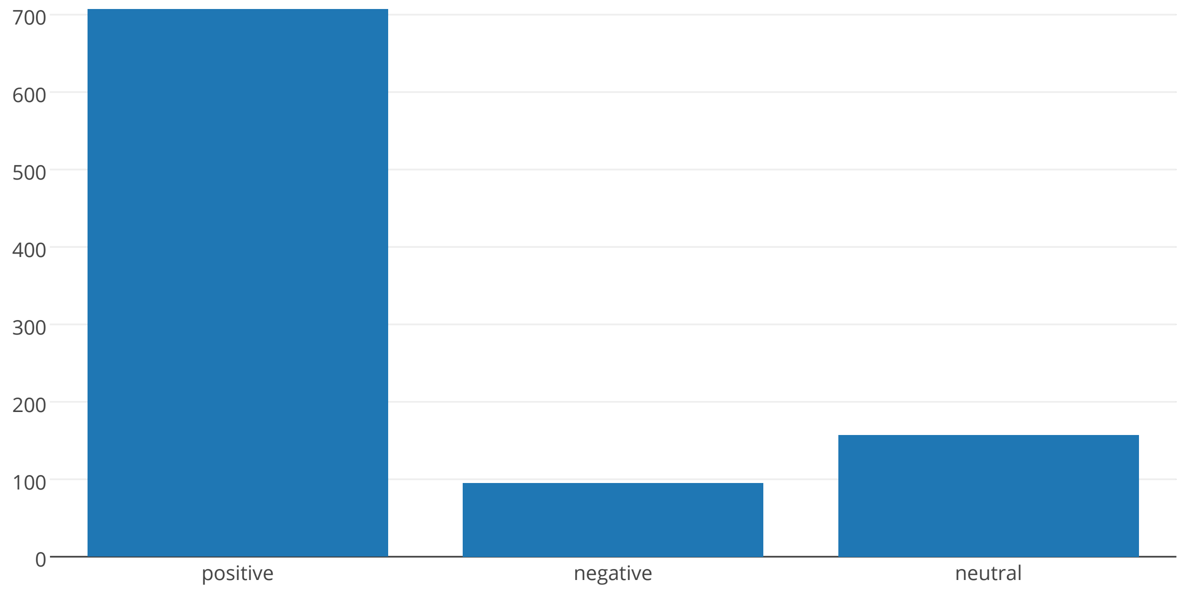
Export to plot.ly »

How feature "number_of_negative_emo" separates the tweets



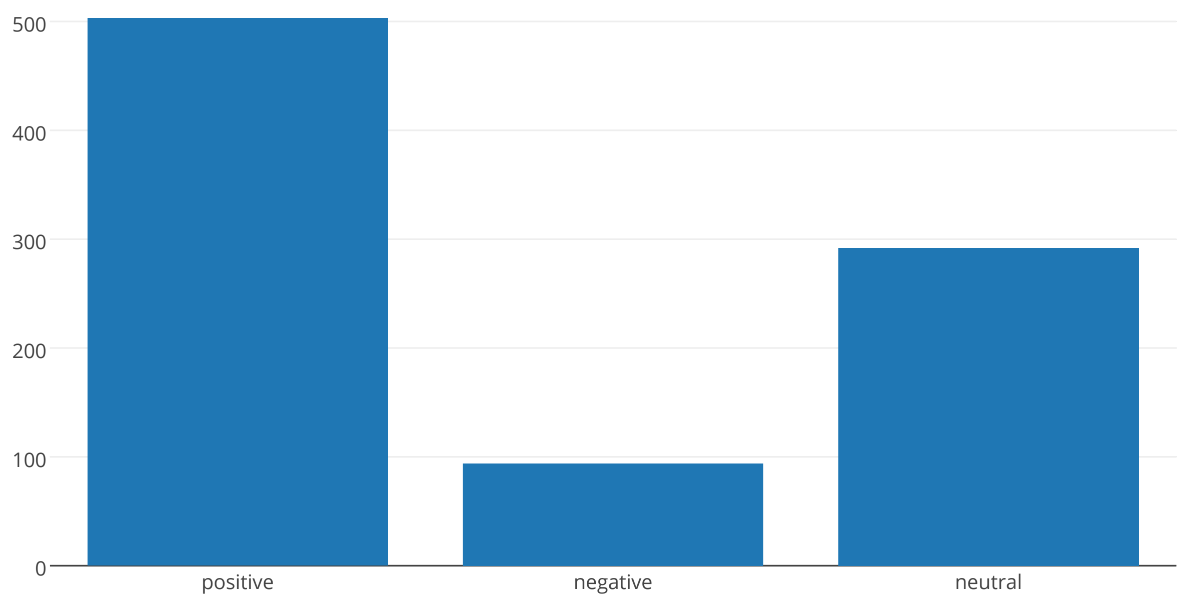
Export to plot.ly »

How feature "number_of_exclamation" separates the tweets



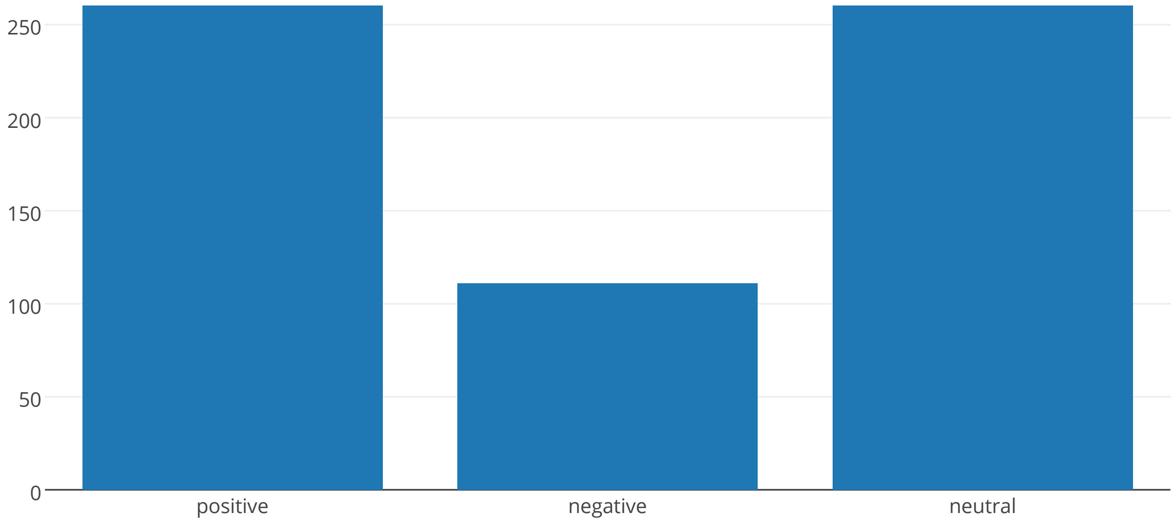
Export to plot.ly »

How feature "number_of_hashtags" separates the tweets



Export to plot.ly »

How feature "number_of_question" separates the tweets



Export to plot.ly »

Experiment 2: extended features + Random Forest

As a second attempt on the classification the **Random Forest** will be used.

```
from sklearn.ensemble import RandomForestClassifier
X_train, X_test, y_train, y_test = train_test_split(data_model.iloc[:, 1:], data_model.iloc[:,
                                                    train_size=0.7, stratify=data_model.iloc[:,
                                                    random_state=seed)
precision, recall, accuracy, f1 = test_classifier(X_train, y_train, X_test, y_test, RandomFores
```

```
===============================================
Testing RandomForestClassifier
Learing time 6.9144287109375s
Predicting time 0.21802711486816406s
=================== Results ===================
            Negative     Neutral     Positive
F1       [ 0.24501425  0.47944007  0.70340909]
Precision[ 0.47777778  0.49192101  0.63163265]
Recall   [ 0.16475096  0.46757679  0.79358974]
Accuracy 0.575291948371
===============================================
```

The accuracy for the initial split was lower than the one for the Naive Bayes, but let's see what happens during crossvalidation:

```
rf_acc = cv(RandomForestClassifier(n_estimators=403,n_jobs=-1, random_state=seed),data_model.i
```

```
===============================================
Crossvalidating RandomForestClassifier...
Crosvalidation completed in 70.09595036506653s
Accuracy: [ 0.54344624  0.50442478  0.37905605  0.27876106  0.37905605  0.52141802
  0.5155096   0.55621302]
Average accuracy: 0.459735602399
===============================================
```

It looks better, however it's still not much above accuracy of the random classifier and barely better than Naive Bayes classifier.

We can observe a low recall level of the RandomForest classifier for the negative class, which may be caused by the data skewness.

More features - word2vec

The overall performance of the previous classifiers could be enhanced by performing time-consuming parameters adjustments, however there's not guarantee on how big the gain will be.

If the out-of-the-shelf methods did not performed well, it seems that there's not much in the data itself. The next idea to add more into data model is to use word2vec representation of a tweet to perform classification.

The word2vec allows to transform words into vectors of numbers. Those vectors represent abstract features, that describe the word similarities and relationships (i.e co-occurence).

What is the best in the word2vec is that operations on the vectors approximately keep the characteristics of the words, so that joining (averaging) vectors from the words from sentence procude vector that is likely to represent the general topic of the sentence.

A lot of pre-trained word2vec models exists, and some of them were trained on huge volumes of data. For the purpose of this analysis, the one trained on over **2 billion of tweets** with 200 dimensions (one vector consists of 200 numbers) is used. The pre-trained model can be downloaded here: https://github.com/3Top/word2vec-api

From GloVe to word2vec

In order to use GloVe-trained model in gensim library, it needs to be converted to word2vec format. The only difference between those formats is that word2vec text files starts with two numbers: *number of lines in file* and *number of dimensions*. The file glove.twitter.27B.200d.txt does not contain those lines.

Unfortunaltely, this text file size is over 1.9GB and text editors cannot be used to open and modify it in reasonable amount of time, this **C#** snippet adds this required line (sorry that it's not Python, but I was having memory problems with encoding of the file in Python. It's required to use x64 target):

```
using (var fileStream = new FileStream("glove.twitter.27B.200d.txt", FileMode.Open,FileAcce
{
    var lines = new LinkedList<string>();
    using (var streamReader = new StreamReader(fileStream))
    {
        while (!streamReader.EndOfStream)
        {
            lines.AddLast(streamReader.ReadLine());
        }
    }
    lines.AddFirst("1193514 200");
    File.WriteAllLines("word2vec.twitter.27B.200d.txt.txt", lines);
}
```

Already modified GloVe for file The file that has the first line appended can be downloaded from here (622MB 7-zip file, ultra compression): https://marcin.egnyte.com/dl/eIbKWE9N2Y

Using Word2Vec

The following class exposes a easy to use interface over the word2vec API from gensim library:

```
class Word2VecProvider(object):
    word2vec = None
    dimensions = 0

    def load(self, path_to_word2vec):
        self.word2vec = gensim.models.Word2Vec.load_word2vec_format(path_to_word2vec, binary=Fa
        self.word2vec.init_sims(replace=True)
        self.dimensions = self.word2vec.vector_size

    def get_vector(self, word):
        if word not in self.word2vec.vocab:
            return None

        return self.word2vec.syn0norm[self.word2vec.vocab[word].index]

    def get_similarity(self, word1, word2):
```

```
        if word1 not in self.word2vec.vocab or word2 not in self.word2vec.vocab:
            return None

        return self.word2vec.similarity(word1, word2)
```

```
word2vec = Word2VecProvider()

# REPLACE PATH TO THE FILE
word2vec.load("C:\\__\\machinelearning\\glove.twitter.27B.200d.txt")
```

Extra features from word2vec

Besides the 200 additional features from the word2vec representation, I had an idea of 3 more features. If word2vec allows to find similarity between words, that means it can find similarity to the specific emotion-representing words. The first idea was to compute similarity of the whole tweet with words from labels: *positive, negative, neutral*. Since the purpose was to find the sentiment, I thought that it will be better to find similarity with more expressive words such as: **good** and **bad**. For the neutral sentiment, I've used word **information**, since most of the tweets with neutral sentiment were giving the information.

The features were builded by computing **mean similarity of the whole tweet to the given word**. Then, those mean values were normalized to [0;1] in order to deal with different word count across tweets.

Final data model

The final data model will contain:

- extra text features (number of: !, ?, :-) etc)
- word2vec similarity to "good", "bad" and "information" words
- word2vec 200 dimension averaged representation of a tweet
- bag-of-word representation of a tweet

```
class TwitterData(TwitterData_ExtraFeatures):

    def build_final_model(self, word2vec_provider, stopwords=nltk.corpus.stopwords.words("eng
        whitelist = self.whitelist
        stopwords = list(filter(lambda sw: sw not in whitelist, stopwords))
        extra_columns = [col for col in self.processed_data.columns if col.startswith("number_
        similarity_columns = ["bad_similarity", "good_similarity", "information_similarity"
        label_column = []
        if not self.is_testing:
```

```
    label_column = ["label"]

columns = label_column + ["original_id"] + extra_columns + similarity_columns + list(
    map(lambda i: "word2vec_{0}".format(i), range(0, word2vec_provider.dimensions)))
    map(lambda w: w + "_bow",self.wordlist))
labels = []
rows = []
for idx in self.processed_data.index:
    current_row = []

    if not self.is_testing:
        # add label
        current_label = self.processed_data.loc[idx, "emotion"]
        labels.append(current_label)
        current_row.append(current_label)

    current_row.append(self.processed_data.loc[idx, "id"])

    for _, col in enumerate(extra_columns):
        current_row.append(self.processed_data.loc[idx, col])

    # average similarities with words
    tokens = self.processed_data.loc[idx, "tokenized_text"]
    for main_word in map(lambda w: w.split("_")[0], similarity_columns):
        current_similarities = [abs(sim) for sim in
                                map(lambda word: word2vec_provider.get_similarity(main_
                                sim is not None]
        if len(current_similarities) <= 1:
            current_row.append(0 if len(current_similarities) == 0 else current_simila
            continue
        max_sim = max(current_similarities)
        min_sim = min(current_similarities)
        current_similarities = [((sim - min_sim) / (max_sim - min_sim)) for sim in
                                current_similarities]  # normalize to <0;1>
        current_row.append(np.array(current_similarities).mean())

    # add word2vec vector
    tokens = self.processed_data.loc[idx, "tokenized_text"]
    current_word2vec = []
    for _, word in enumerate(tokens):
        vec = word2vec_provider.get_vector(word.lower())
```

```
            if vec is not None:
                current_word2vec.append(vec)

        averaged_word2vec = list(np.array(current_word2vec).mean(axis=0))
        current_row += averaged_word2vec

        # add bag-of-words
        tokens = set(self.processed_data.loc[idx, "text"])
        for _, word in enumerate(self.wordlist):
            current_row.append(1 if word in tokens else 0)

        rows.append(current_row)

    self.data_model = pd.DataFrame(rows, columns=columns)
    self.data_labels = pd.Series(labels)
    return self.data_model, self.data_labels
```

```
td = TwitterData()
td.initialize("data\\train.csv")
td.build_features()
td.cleanup(TwitterCleanuper())
td.tokenize()
td.stem()
td.build_wordlist()
td.build_final_model(word2vec)

td.data_model.head(5)
```

	label	original_id	number_of_uppercase	number_of_exclamation	number_of_question	number
0	neutral	635930169241374720	2	0	0	0
1	neutral	635950258682523648	0	0	0	0
2	negative	636030803433009153	2	0	0	0
3	positive	636100906224848896	0	0	1	0
4	neutral	636176272947744772	4	0	0	0

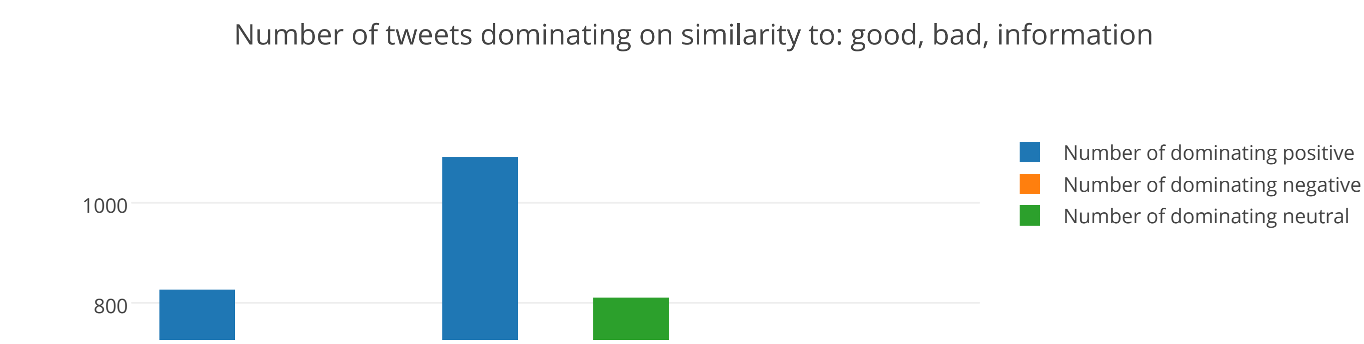
5 rows × 2399 columns

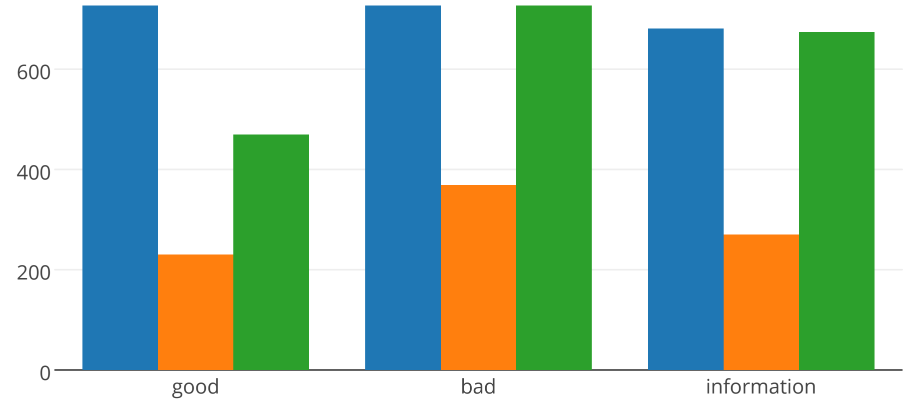
```
data_model = td.data_model
data_model.drop("original_id",axis=1,inplace=True)
```

Logic behind word2vec extra features

In order to show, why the 3 extra features built using word2vec might help in sentiment analysis. The chart below shows how many tweets from given category were dominating (had highest value) on similarity to those words. Although the words doesn't seem to separate the sentiments themselves, the differences between them in addition to other parameters, may help the classification process - i.e when tweet has highest value on *good_similarity* it's more likely for it to be classified to have positive sentiment.

```
columns_to_plot = ["bad_similarity", "good_similarity", "information_similarity"]
bad, good, info = columns_to_plot
sentiments = ["positive","negative","neutral"]
only_positive = data_model[data_model[good]>=data_model[bad]]
only_positive = only_positive[only_positive[good]>=only_positive[info]].groupby(["label"]).cou

only_negative = data_model[data_model[bad] >= data_model[good]]
only_negative = only_negative[only_negative[bad] >= only_negative[info]].groupby(["label"]).co

only_info = data_model[data_model[info]>=data_model[good]]
only_info = only_info[only_info[info]>=only_info[bad]].groupby(["label"]).count()

plot_data_w2v = []
for sentiment in sentiments:
    plot_data_w2v.append(graph_objs.Bar(
            x = ["good","bad", "information"],
            y = [only_positive.loc[sentiment,:][0], only_negative.loc[sentiment,:][0], only_ir
            name = "Number of dominating " + sentiment
    ))

plotly.offline.iplot({
        "data":plot_data_w2v,
        "layout":graph_objs.Layout(title="Number of tweets dominating on similarity to: goo
    })
```





Export to plot.ly »

Experiment 3: full model + Random Forest

The model is now complete. With a lot of new features, the learning algorithms should perform totally differently on the new data set.

```
X_train, X_test, y_train, y_test = train_test_split(data_model.iloc[:, 1:], data_model.iloc[:,
                                                    train_size=0.7, stratify=data_model.iloc[:,
                                                    random_state=seed)
precision, recall, accuracy, f1 = test_classifier(X_train, y_train, X_test, y_test, RandomFore
rf_acc = cv(RandomForestClassifier(n_estimators=403,n_jobs=-1,random_state=seed),data_model.il
```

```
===============================================
Testing RandomForestClassifier
Learing time 3.579137086868286s
Predicting time 0.2350320816040039s
=================== Results ===================
            Negative     Neutral     Positive
F1       [ 0.08695652  0.48138056  0.72029835]
Precision[ 0.8         0.51456311  0.61622607]
Recall   [ 0.04597701  0.45221843  0.86666667]
Accuracy 0.585740626921
===============================================
===============================================
Crossvalidating RandomForestClassifier...
Crosvalidation completed in 41.57810711860657s
Accuracy: [ 0.58468336  0.52654867  0.54129794  0.54867257  0.51622419  0.5465288
  0.58493353  0.56508876]
Average accuracy: 0.551747226492
===============================================
```

As we can see, the average accuracy from crossvalidation is almost 58%, and the results from the crossvalidation runs are more stable - they never drop below 51%. It might seem that the algorithm will perform pretty well on the testing set. There's a one problem with that - recall for the 7:3 split shows, that only about 3% of the negative tweets from the whole set were recognized properly - it seems like the algorithm is good with recognition of postive vs. neutral cases, but it does really poor job when it comes to recognize negative sentiment. Let's try something else.

Experiment 4: full model + XGBoost

XGBoost is relatively new machine learning algorithm based on decision trees and boosting. It is highly scalable and provides results, which are often higher than those obtained using popular algorithms such as Random Forest or SVM.

Important: XGBoost exposes scikit-learn interface, but it needs to be installed as an additional python package. See this page to see more: https://xgboost.readthedocs.io/en/latest/build.html

```
from xgboost import XGBClassifier as XGBoostClassifier
```

```
X_train, X_test, y_train, y_test = train_test_split(data_model.iloc[:, 1:], data_model.iloc[:,
                                                    train_size=0.7, stratify=data_model.iloc[:,
                                                    random_state=seed)
precision, recall, accuracy, f1 = test_classifier(X_train, y_train, X_test, y_test, XGBoostCla
```

```
===============================================
Testing XGBClassifier
Learing time 94.12349128723145s
Predicting time 0.2379899024963379s
=================== Results ===================
            Negative     Neutral     Positive
F1       [ 0.33595801  0.50823938  0.72674419]
Precision[ 0.53333333  0.51675485  0.66489362]
Recall   [ 0.24521073  0.5         0.80128205]
Accuracy 0.60356484327
===============================================
```

60% accuracy is the highest result yet. Still, it's only a result on the splitted training set, let's see what happens when we crossvalidate this algorithm (with default parameters).

```
xgb_acc = cv(XGBoostClassifier(seed=seed),data_model.iloc[:, 1:], data_model.iloc[:, 0])
```

```
===============================================
Crossvalidating XGBClassifier...
Crosvalidation completed in 335.4184989929199s
```

```
Accuracy: [ 0.62592047  0.53834808  0.50737463  0.47492625  0.42182891  0.53471196
  0.56277696  0.54585799]
Average accuracy: 0.526468157158
==============================================
```

Averaged accuracy seems to be lower and less stable than for the Random Forest, but given over 8 times better recall for the negative cases, the XGBoost seems to be a good starting point for the final classifier.

Finding best parameters for XGBoost

Warning the code bellow executes for a really long time. On computer with i7@3.4GHz processor it took over 90 minutes to perform 5 iterations of RandomizedSearchCV, so depending on your resources, you can find even better parameters.

The following utility functions are used to find the best parameters for the XGBoost using RandomizedSearchCV funciton.

```
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] == i)
        for candidate in candidates:
            log("Model with rank: {0}".format(i))
            log("Mean validation score: {0:.3f} (std: {1:.3f})".format(
                results['mean_test_score'][candidate],
                results['std_test_score'][candidate]))
            log("Parameters: {0}".format(results['params'][candidate]))
            log("")

def best_fit(X_train, y_train, n_iter=5):

    parameters = {
        "n_estimators":[103,201, 403],
        "max_depth":[3,10,15, 30],
        "objective":["multi:softmax","binary:logistic"],
        "learning_rate":[0.05, 0.1, 0.15, 0.3]
    }

    rand_search = RandomizedSearchCV(XGBoostClassifier(seed=seed),param_distributions=parameter
                                     n_iter=n_iter,scoring="accuracy",
                                     n_jobs=-1,cv=8)

    import time as ttt
```

```
    now = time()
    log(ttt.ctime())
    rand_search.fit(X_train, y_train)
    report(rand_search.cv_results_, 10)
    log(ttt.ctime())
    log("Search took: " + str(time() - now))
```

```
best_fit(data_model.iloc[:, 1:], data_model.iloc[:, 0], n_iter=10)
```

For 10 runs and 12400s running time, the following parameters were discovered:

```
Model with rank: 1
Mean validation score: 0.550 (std: 0.034)
Parameters: {'n_estimators': 403, 'max_depth': 10, 'objective': 'binary:logistic', 'learning_ra

Model with rank: 2
Mean validation score: 0.546 (std: 0.044)
Parameters: {'learning_rate': 0.05, 'n_estimators': 201, 'objective': 'multi:softmax', 'max_dep

Model with rank: 2
Mean validation score: 0.546 (std: 0.035)
Parameters: {'n_estimators': 403, 'max_depth': 15, 'objective': 'binary:logistic', 'learning_ra

Model with rank: 4
Mean validation score: 0.545 (std: 0.033)
Parameters: {'n_estimators': 403, 'max_depth': 30, 'objective': 'binary:logistic', 'learning_ra

Model with rank: 5
Mean validation score: 0.543 (std: 0.033)
Parameters: {'max_depth': 30, 'n_estimators': 103, 'objective': 'multi:softmax', 'learning_rate

Model with rank: 6
Mean validation score: 0.542 (std: 0.041)
Parameters: {'n_estimators': 103, 'max_depth': 30, 'objective': 'binary:logistic', 'learning_ra

Model with rank: 7
Mean validation score: 0.542 (std: 0.046)
Parameters: {'n_estimators': 103, 'max_depth': 10, 'objective': 'binary:logistic', 'learning_ra

Model with rank: 8
Mean validation score: 0.541 (std: 0.033)
Parameters: {'learning_rate': 0.1, 'n_estimators': 103, 'objective': 'binary:logistic', 'max_de

Model with rank: 9
Mean validation score: 0.509 (std: 0.068)
Parameters: {'n_estimators': 403, 'max_depth': 3, 'objective': 'multi:softmax', 'learning_rate'
```

```
Model with rank: 10
Mean validation score: 0.504 (std: 0.068)
Parameters: {'max_depth': 3, 'n_estimators': 403, 'objective': 'multi:softmax', 'learning_rate'
```

Test data classification

After finding best cross-validated parameter for the XGBoost, it's time to load the test data and predict sentiment for them. Final classifier will be trained on the whole training set. End score will be revealed when the *Angry Tweets* competition will end (https://inclass.kaggle.com/c/angry-tweets).

The data will be exported to CSV file in format containing two columns: Id, Category. There are 4000 test samples with unknown distribution of the sentiment labels.

```
test_data = TwitterData()
test_data.initialize("data\\test.csv", is_testing_set=True)
test_data.build_features()
test_data.cleanup(TwitterCleanuper())
test_data.tokenize()
test_data.stem()
test_data.build_wordlist()
test_data.build_final_model(word2vec)

test_data.data_model.head(5)
```

	original_id	number_of_uppercase	number_of_exclamation	number_of_question	number_of_ellip
0	628949369883000832	0	0	1	0
1	628976607420645377	1	1	0	0
2	629023169169518592	0	0	0	0
3	629179223232479232	0	0	0	0
4	629186282179153920	1	0	1	0

5 rows × 2398 columns

```
test_model = test_data.data_model

data_model = td.data_model

xgboost = XGBoostClassifier(seed=seed,n_estimators=403,max_depth=10,objective="binary:logisti
xgboost.fit(data_model.iloc[:,1:],data_model.iloc[:,0])
predictions = xgboost.predict(test_model.iloc[:,1:])
```

```
test_model.head(5)
```

	original_id	number_of_uppercase	number_of_exclamation	number_of_question	number_of_ellip
0	628949369883000832	0	0	1	0
1	628976607420645377	1	1	0	0
2	629023169169518592	0	0	0	0
3	629179223232479232	0	0	0	0
4	629186282179153920	1	0	1	0

5 rows × 2398 columns

```
results = pd.DataFrame([],columns=["Id","Category"])
results["Id"] = test_model["original_id"].astype("int64")
results["Category"] = predictions
results.to_csv("results_xgb.csv",index=False)
```

Feature importance

Let's take a look at the final model feature importance.

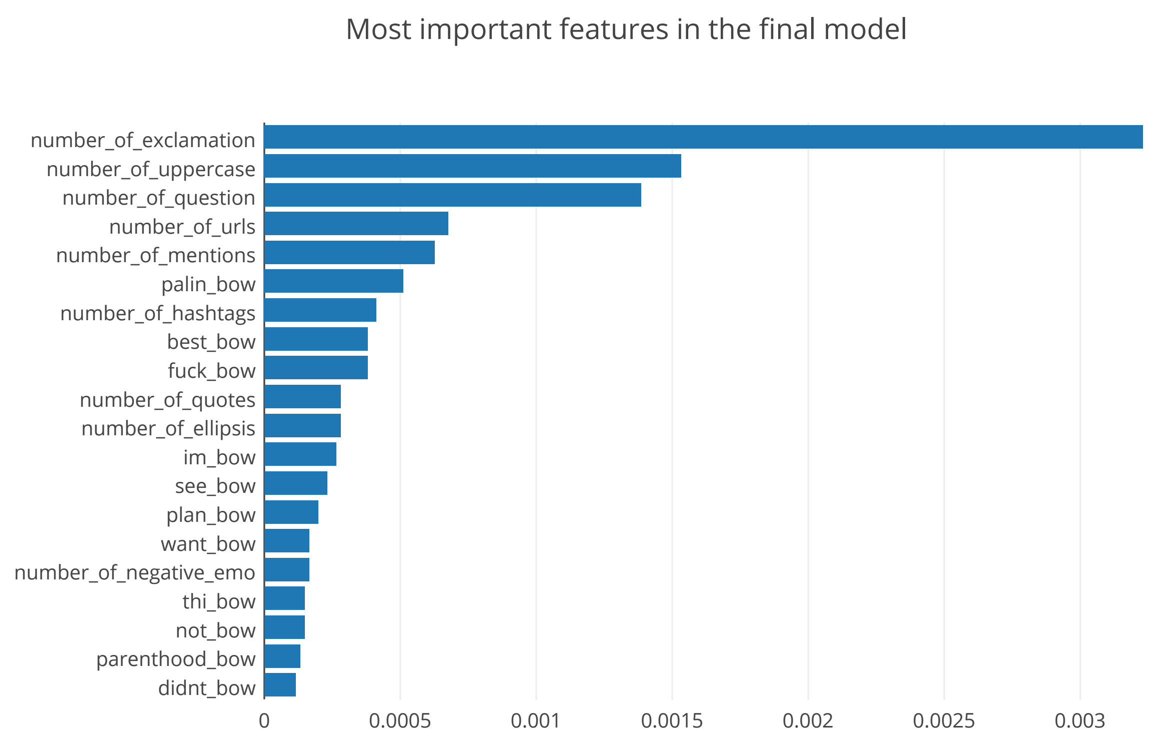
Output list is really long, so the print line is commented in the code block

```
features = {}
for idx, fi in enumerate(xgboost.feature_importances_):
    features[test_model.columns[1+idx]] = fi

important = []
for f in sorted(features,key=features.get,reverse=True):
    important.append((f,features[f]))
    # print(f + " " + str(features[f]))
```

It's not surprise that the word2vec and related good/bad/information similarity features were the most important, because the classification performance was greatly improved after switching to this representation.

What is interesting is that a lot of custom-crafted features (number_of_*) were also highly important, beating a lot of features which came from bag-of-words representation. Here's the chart for the easiness of reading:

```
to_show = list(filter(lambda f: not f[0].startswith("word2vec") and not f[0].endswith("_simi
to_show.reverse()
features_importance = [
    graph_objs.Bar(
        x=[f[1] for f in to_show],
        y=[f[0] for f in to_show],
        orientation="h"
)]
plotly.offline.iplot({"data":features_importance, "layout":graph_objs.Layout(title="Most impc
      margin=graph_objs.Margin(
           l=200,
        pad=3
    ),)})
```



Export to plot.ly »

XGBoost considered custom features as important, so it also confirmed that some non-word features of a text can also be used to predict the sentiment. Most of them were even more important than the actual presence of some emotion-expressing words in the text.

Summary

Experiment showed that prediction of text sentiment is a non-trivial task for machine learning. A lot of preprocessing is required just to be able to run any algorithm and see - usually not great - results. Main problem for sentiment analysis is to craft the machine representation of the text. Simple bag-of-words was definitely not enough to obtain satisfying results, thus a lot of additional features were created basing on common sense (number of emoticons, exclamation marks etc.). Word2vec representation significantly raised the predictions quality. I think that a slight improvement in classification accuracy for the given training dataset could be developed, but since it contained highly skewed data (small number of negative cases), the difference will be probably in the order of a few percent. The thing that could possibly improve classification results will be to add a lot of additional examples (increase training dataset), because given 5971 examples obviously do not contain every combination of words usage, moreover - a lot of emotion-expressing words surely are missing.

Tags

machine-learning python sentiment-analysis text-mining scikit-learn

Related

- Schedule Python Apache Beam DataFlow pipeline using App Engine CRON service (30 June 2017)
- Stroke Width Transform algorithm for Python (11 August 2017)

Comments

Rachal Park • 7 days ago

Thanks for great posting!

Reply • Share ›

raza khan • 12 days ago

is this code is available on github etc?

Reply • Share ›

Marcin Zablocki Mod raza khan • 12 days ago

The link is at the beginning of the post in TL;DR section... https://github.com/marrrcin...

Reply • Share ›

raza khan ➦ Marcin Zablocki • 11 days ago

THANKX

^ | ˅ • Reply • Share ›

Mohammed Hasanuzzaman • 3 months ago

Thank for sharing the code. I am new to python. I am getting the following error while executing the code.

averaged_word2vec = list(np.array(current_word2vec).mean(axis=0))
TypeError: 'numpy.float64' object is not iterable

Pls help

^ | ˅ • Reply • Share ›

Nikolay Frick ➦ Mohammed Hasanuzzaman • 24 days ago

I had the same issue. You probably want to check if length of current_word2vec is greater than zero before calculating averaged_word2vec. If zero then just set averaged_word2vec to empty list [].

^ | ˅ • Reply • Share ›

Proudly developing

.NET Developer at