ERG3020 Course Project - Twitter
CUHK-SZ, Apr 22, 2018

Course Project

Student Name: Piao Chuxin, Ye Xiaoxing

Student ID: 115010058, 115010270

The project combines two parts, one part for the sailor2017 (Code for SAILORS 2017 NLP project), and another for the bonus part which analyses the tweets posted on 2016 U.S. Election Day.

All the code included in this report can be found in these two repos:

- `https://github.com/Yexiaoxing/sailors2017` for Part 1
- `https://github.com/Yexiaoxing/tweet-sentiment-analysis` for Part 2

Contents

Chapter 1

Part I: Sailor2017

In this part, we implemented the Naive Bayes Classifier using tweets data, using the code from sailor2017 (https://github.com/abisee/sailors2017). Copies of exported PDFs are attached to the report, and you can also view the files online. Due to some technique problems, the PDF is not wrapping lines and some elements will be missing.

1.1 What is this project?

The original project is a workshop in sailors2017, focus on the introduction to Natural Language Processing. The task was to automatically classify real tweets from Sandy using a Naive Bayes model.

The project is structured in six parts, and each part deals with a small task. In the first part, the project introduces some of the basic usage of how to define simple functions. Through the process of dealing with functions that can successfully classify the grade levels according to score, the project then develops to classify tweets with specific words in sentences at the end of part one.

In the second part, the project introduces `list` which is one of the useful data structure in this project, because 'list' is an excellent way to tied item with labels. Besides the basic usage of list, some new concepts of model evaluation are also listed in this part, such as precision, recall and F1 score.

In the third part, the detailed process of naive bayes algorithm is developed, but in this part, the program using the classifier only in identifying which box different balls come from. Begin with the function of 'Counter', the naive bayes algorithm can correctly count the number of a specific item in the whole dataset, which will provide an approach to calculate the conditional probability directly in the next step. Using for loop, the naive bayes program can calculate all the conditional probability of different item by dividing the number of whole dataset to the result of 'counter' function for each item. After getting the conditional probability, the program then calculates the most probable class of the new items. Setting the equal prior probability, the program multiplies the conditional probability to get the final probability of the item in one class. Finally, find out the class that has the largest posterior probability and that class is the result of the naive bayes prediction.

In the fourth part, the program first constructs the 'counter' unigram_probs, which maps each word to its probability. The program then developed into bigram_probs and trigram_probs, which maps pairs and triples of words to their probability. This part of the program is a foundation of developing classifier with different structure of sentences.

In the fifth part, the program uses the same train of thought in the third part to the tweeter dataset. Similar as before, we first calculate the prior probability of tweeters in separate class, such as food, medical and so on. After calculating the prior probability, the program then focusses on the posterior probability which equals to the prior probability multiply the conditional probability. After getting the posterior probability, the program finds the max posterior probability of one class and use this class as the result of bayes classifier. Finally, to evaluate the model, the program use methods, such as precision and recall in the second part. The overall precision and recall, are all over 90% which indicate the naive bayes classifier have a good prediction.

In the sixth part, originated from the fourth and fifth part, the program incorporating bigrams with the naive bayes classifier, which may improve the average F1 score of the test set. The results show that two fifth of the F1 score have been significantly improved. To visualize the specific way the prediction improves, confusion matrix is a useful tool. A confusion matrix shows you for each true category c how many of the tweets in c were classified into the five distinct categories. To go further, visualizing individual tweets can help the programmer understand why a classifier made a correct or wrong prediction.

For the error analysis, we want to figure out why the classifier made mistakes. For one possible reason, the way of recognition of the specific words may sometimes make mistakes. For example, we define the word 'milk' as the sign of food category, but actually, some sentence contains 'milk' may not indicate the category of food, such as 'milky way' have nothing to do with food, and this may be one of the possible reason for the classifier making mistakes.

1.2 What we have also done?

Besides implementing the codes, we also adopted the original codes to Python 3. The original one is in Python 2 and it is going to be deprecated.

For the bonus part, we have finished a tweet sentiment analysis project, which is attached as well. The project collects 4695447 tweets on the Election Day from 2435717 users. A sentiment analysis is conducted on all tweets, and their political leanings are also analyzed. After that, we will figure out whether the social network can predict the election results.

1.3 Jupyter Notebook Viewers

- lesson1_rulebased.ipynb: https://github.com/Yexiaoxing/sailors2017/blob/Miley/lesson1_rulebased.ipynb
- lesson2_evaluation.ipynb: https://github.com/Yexiaoxing/sailors2017/blob/Miley/lesson2_evaluation.ipynb
- lesson3_naivebayes_exercises.ipynb: https://github.com/Yexiaoxing/sailors2017/blob/Miley/lesson3_naivebayes_exercises.ipynb
- lesson4_languagemodel.ipynb: https://github.com/Yexiaoxing/sailors2017/blob/Miley/lesson4_languagemodel.ipynb
- lesson5_naivebayes.ipynb: https://github.com/Yexiaoxing/sailors2017/blob/Miley/lesson5_naivebayes.ipynb
- lesson6_visualization.ipynb: https://github.com/Yexiaoxing/sailors2017/blob/Miley/lesson6_visualization.ipynb

Chapter 2

Part 2 (Bonus): Twitter Election Data Analysis

2.1 Abstract

Twitter is a great firehouse of real-time information. Using the data from twitter, people can gain information such as political leaning and other tendencies. Through the classification process of the data before the election day, the more competitive candidate with a higher approval rating can be obtained.

2.2 Introduction

American President Election is one of the catchiest affairs all over the world. To predict the candidate with higher approval rate beforehand, social networks can be effective in obtaining the political leaning of citizens in the United States.

Twitter is one of the influential social networks that can be used in predicting the political leaning of people who use twitter. Having downloaded the data of one day from the twitter website, the emotion scores have been computed, represented by a number from -1 to 1, in which 1 represents the positive emotion while -1 represents the negative emotions. After obtaining the score of emotion, the keywords which could represent different candidate have been searched. For example, Trump or Donald could represent Donald Trump, and Hillary or Clinton could represent Hillary Clinton. Compared the twitter score which contains different key words to the average emotional score of all the twitter score, the rough political leaning of different candidates can be obtained.

To tell the political leaning of people, we use sentiment analysis. Sentiment Analysis, sometimes known as opinion mining, is a field of study combining the use of natural language processing, text analysis, and linguistics, to identify and study affective and subjective information. Tipically, it analyses peoples opinions towards entities, such as products, companies, and parties. Thanks to the rapid-developing social media, this area has now been a hot topic with a constant source of textual data.

2.3 Model

2.3.1 Sentiment Analysis

To evaluate the tweets' sentiment, we first trained a series of models. They are...

1. Naive Bayes
   (a) Multinomial Naive Bayes
   (b) Bernoulli Naive Bayes
2. Linear Model
   (a) Logistic Regression
   (b) Stochastic Gradient Descent
3. Support Vector Machine
   (a) Linear Support Vector Classification

In this part, we would like to introduce the models one by one, and then our implementation.

Naive Bayes

Naive Bayes classifier is a simple probabilistic classifier based on Bayes' Theorem, while assuming the independence between features. This technique has been studied for over 60 years, and it is now hot since it is simple but effective. It is still a popular and the baseline method for text classification. With proper pre-processing, ti is evn competitive with advanced methods like SVM.

Naive Bayes is a simple technique for constructing classifiers. Class labels are assigned to instances, where the labels is a finite set. It is a family of algorithems based the principle that all assume the independence of feature, given the class variable. Despite the oversimplifiatied assumptions, the Navie Bayes classifiers works well in many real-world cases. And, one of the advantage is that, only a small training set is required, to guess the essential parameters. [Wik18]

Naive Bayes is a conditional probability model. Given an instance to be classified, its features are represented by a vector $\mathbf{x} = (x_1, \ldots, x_n)$, and the classification is assigned to the probabilities, $p(C_k \mid x_1, \ldots, x_n)$, for each of K possible classes C_k. Using Bayes' Theorem, the conditional probability can be transformed to

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k)\ p(\mathbf{x} \mid C_k)}{p(\mathbf{x})}$$

which, in plain English, can be written as

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{evidence}}$$

If we break down the equation [Let15],

- $p(C_k)$ (Prior) = frequency of class C_k / total number of samples
- $p(\mathbf{x}|C_k)$ (Likelihood) = (frequency of $\mathbf{x}$ / number of samples) where class = C_k
- $p(\mathbf{x})$ (Evidence) = frequency of $\mathbf{x}$ / total number of samples

We only care about the numerator of that fraction, since the denominator does not depend on C, and it is effectively constant since the values x_i are already given. By the joint probability model, the numerator is equivalent to $p(C_k, x_1, \dots, x_n)$, and by the chain rule for repeated applications,

$$\begin{aligned}
p(C_k, x_1, \dots, x_n) &= p(x_1, \dots, x_n, C_k) \\
&= p(x_1 \mid x_2, \dots, x_n, C_k) p(x_2, \dots, x_n, C_k) \\
&= p(x_1 \mid x_2, \dots, x_n, C_k) p(x_2 \mid x_3, \dots, x_n, C_k) p(x_3, \dots, x_n, C_k) \\
&= \dots \\
&= p(x_1 \mid x_2, \dots, x_n, C_k) p(x_2 \mid x_3, \dots, x_n, C_k) \dots p(x_{n-1} \mid x_n, C_k) p(x_n \mid C_k) p(C_k)
\end{aligned}$$

Then the naive conditional independence assumptions (assuming that each feature is conditionally independent with each other, given the category C_k) come, $p(x_i \mid x_{i+1}, \dots, x_n, C_k) = p(x_i \mid C_k)$. Combine all equations,

$$\begin{aligned}
p(C_k \mid x_1, \dots, x_n) \propto p(C_k, x_1, \dots, x_n) &= \\
&= p(C_k)\ p(x_1 \mid C_k)\ p(x_2 \mid C_k)\ p(x_3 \mid C_k)\ \cdots \\
&= p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k)\,.
\end{aligned}$$

Then, the conditional distribution over the C becomes,

$$p(C_k \mid x_1, \dots, x_n) = \frac{1}{Z} p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k)$$

where the evidence $Z = p(\mathbf{x}) = \sum_k p(C_k)\ p(\mathbf{x} \mid C_k)$ is a scaling factor only depending on $x_1, \dots, x_n$, and now we have the naive Bayes probability model.

The naive Bayes classifier combines the naive Bayes probability model with a decision rule. One common rule is MAP decision rule, which is the equation of

$$\hat{y} = \underset{k \in \{1,\ldots,K\}}{argmax}\ p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k).$$

Different models of Naive Bayes classifier The prior of class shall be calculated by assuming all classes have equal probability, or by calculating an estimate from the training set. Before we can estimate the parameters, we shall assume a distribution from the training set. This assumption is often called the event model of the Naive Bayes classifier. Since we are classifying document data, which have discrete features, we use multinomial and Bernoulli distributions.

Multinomial distribution This multinomial navie bayes models the word counts With such a multinomial event model, the feature vectors represent the frequencies, where certain events have been generated by a multinomial distribution $(p_1, \ldots, p_n)$ where p_i is the probability that event **i** occurs. If we take x_i as the numvber of times event **i** in a particular instance, then the feature vector $\mathbf{x} = (x_1, \ldots, x_n)$ is now a histogram. Then, the likelihood function is given by

$$p(\mathbf{x} \mid C_k) = \frac{(\sum_i x_i)!}{\prod_i x_i!} \prod_i {p_{ki}}^{x_i}$$

This model is suitable for text classification. The features are the words, and the values is the frequencies. If we break the previous equation, it becomes

$$p(x_i \mid C_k) = \frac{N_{C_k x_i} + \alpha}{N_{C_k} + \alpha n}$$

,

where $N_{C_k x_i}$ is the occurances of x_i in Class C_k; N_{C_k} is the total occurances of features in Class C_K; n is the number of features and α is the smoothing parameters ranged in $[0, 1]$, and in most cases it is 1. The reason why we need a smoothing parameter is that if a given feature does not exist in the training data, the estimate will become zero, and then all probabilities under the Naive Bayes classifier will be zero. Adding the smoothing parameters is a way to regularizing naive Bayes, which is called Lidstone smoothing or Laplace smoothing (when $\alpha = 1$).

Bernoulli distribution In the Bernoulli event model, we use binary values for features. In document classification, the binary value determines the existence of features, which is, different from the multinomial model which uses the frequencies. The likelihood can be given by

$$p(\mathbf{x} \mid C_k) = \prod_{i=1}^{n} p_{ki}^{x_i}(1 - p_{ki})^{(1-x_i)}$$

It also explicitly models the absent terms, and the absence is now also a feature.

Linear Model

Linear model, or linear regression, is a linear modeling approach to find the relationship between a scalar dependent variable y and explanatory variables X. It is a technique for predicting a real value using a straight line in 2-D model or plane or hyperplane in higher dimensions.

Each input variable $\mathbf{x}$ is weighted using a coefficient ($\mathbf{b}$). The goal of the algorithems is to discover the best-fit set of coefficients.

$$y = b_0 + b_1 \times x_1 + ...$$

Logistic Regression Logistic regression is a regression model where the dependent variable is categorical. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function [lr-].

The cost function for the L2 logistic regression is:

$$\underset{w,c}{min} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1).$$

Stochastic Gradient Descent Stochastic gradient descent is a simple but efficient approach to fit linear models. coefficients are evaluated and updated every iteration, to minimize the error of the model. In this way, the model makes a prediction for an instance, and updated for the next prediction in order to reduce the error. This requires two parameters,

- Learning Rate: Limit the amount each update;
- Epochs: Iterations to run through the training data.

Then, the coefficients ($\mathbf{b}$) are updated using the equation:

$$\mathbf{b} = \mathbf{b} - \text{learning\_rate} \times \text{error} \times x$$

where error is the prediction error attributed to the weight.

Support Vector Machine

Support Vector Machine is a set of supervised machine larning algorithems, which can be used for classification and regression. In this algorithem, we first plot each data as a point in space, with the value being the value of particular coordinate. Classification is then performed by finding the hyperplane which differentiate the two classes well.

Support Vector Classification is one application of the SVM.

Given training vectors $x_i \in \mathbb{R}^p, i = 1, \ldots, n$, in two classes, and a vector $y \in \{1, -1\}^n$, SVC solves the following primal problem:

$$\min_{w,b,\zeta} \left\{ \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i \right\}$$

$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$

$$\zeta_i \geq 0, i = 1, ..., n$$

Its dual is

$$\min_{\alpha} \left\{ \frac{1}{2} \alpha^T Q \alpha - \mathbf{1}^T \alpha \right\}$$

$$\text{subject to } y^T \alpha = 0$$

$$0 \leq \alpha_i \leq C, i = 1, ..., n$$

where $\mathbf{1}$ is the vector of all ones, $C > 0$ is the upper bound, Q is an n by n positive semidefinite matrix, $Q_{ij} \equiv y_i y_j K(x_i, x_j)$, where $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ.

The decision function is $\text{sgn}(\sum_{i=1}^{n} y_i \alpha_i K(x_i, x) + \rho)$ [PVG+11].

2.3.2 Implementation

All fils of the code we mention here are available online: https://github.com/Yexiaoxing/tweet-sentiment-analysis . To implement the project, we first need tweets data. The tweets ID data from GitHub Repo (chrisalbon/election_day_2016_twitter) was then borrowed. The original dataset contains 6,546,824 tweets posted on election day, which includes one of the following keywords,

1. hillary
2. hillary
3. trump
4. #yourefired
5. election

6. #election2016
7. #electionday
8. #uselections2016
9. gop
10. democrat
11. #ivoted vote voted
12. #senate
13. #uselection
14. #house
15. congress
16. #madampresident

Due to the Twitter's Term of Service, the dataset only contains the IDs [twi18]. So, we need to hydrate them. Hydrate means to get the details from a collection of Tweet IDs. The tweets˙fetch.py file is designed to do so. It accepts various parameters, and two of them are input of a csv files containing IDs, and output of the details.

```python
"""This module fetches full tweets (also called hydrate tweets) from given
↪   csv file."""
import pandas as pd
import tweepy
import csv
from tqdm import trange
from optparse import OptionParser
from typing import List

# Insert your Twitter API key here
consumer_key = ''
consumer_secret = ''

access_token = ''
access_secret = ''


def retrieve_tweets(tweet_ids: List[str],
```

```
                output_file_name: str,
                proxy: str = ""):
    """
    Retrieve tweets from list of tweet ids.

    @param tweet_ids: List of tweet id.
    @param output_file_name: The file to write.
    """
    print("Output:", output_file_name)

    # Authorization with Twitter
    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_token, access_secret)
    api = tweepy.API(
        auth,
        wait_on_rate_limit=True,
        wait_on_rate_limit_notify=True,
        retry_count=3,
        retry_delay=5,
        retry_errors=set([401, 404, 500, 503]),
        proxy=proxy)

    # Create output file
    csvFile = open(output_file_name, 'w', encoding='utf-8')
    csvWriter = csv.writer(csvFile)
    csvWriter.writerow([
        "text", "created_at", "geo", "lang", "place", "coordinates",
        "user.favourites_count", "user.statuses_count", "user.description",
        "user.location", "user.id", "user.created_at", "user.verified",
        "user.following", "user.url", "user.listed_count",
        "user.followers_count", "user.default_profile_image",
        "user.utc_offset", "user.friends_count", "user.default_profile",
        "user.name", "user.lang", "user.screen_name", "user.geo_enabled",
        "user.profile_background_color", "user.profile_image_url",
        "user.time_zone", "id", "favorite_count", "retweeted", "source",
        "favorited", "retweet_count"
    ])

    print("Total IDs", len(tweet_ids))
    # Append tweets to output file

```

```python
    # Twitter allows a batch of 100 tweets
    for tweetid_batch in trange(len(tweet_ids) // 100):
        try:
            status_es = api.statuses_lookup(
                tweet_ids[tweetid_batch * 100:tweetid_batch * 100 + 100])
            for status in status_es:
                csvWriter.writerow([
                    status.text, status.created_at, status.geo, status.lang,
                    status.place, status.coordinates,
                    status.user.favourites_count, status.user.statuses_count,
                    status.user.description, status.user.location,
                    status.user.id, status.user.created_at,
                    status.user.verified, status.user.following,
                    status.user.url, status.user.listed_count,
                    status.user.followers_count,
                    status.user.default_profile_image, status.user.utc_offset,
                    status.user.friends_count, status.user.default_profile,
                    status.user.name, status.user.lang,
                    status.user.screen_name, status.user.geo_enabled,
                    status.user.profile_background_color,
                    status.user.profile_image_url, status.user.time_zone,
                    status.id, status.favorite_count, status.retweeted,
                    status.source, status.favorited, status.retweet_count
                ])
        except Exception as e:
            print(str(e))


def main(options, args):
    """Initialize the board, solver object and call the solve() function."""
    df = pd.read_csv(options.infile)
    retrieve_tweets(df.iloc[:, 0], options.out, proxy=options.proxy)


if __name__ == '__main__':
    parser = OptionParser(usage="Usage: %prog -i input_file" +
                          " -o output_file -p proxy_address")
    parser.add_option("-p", "--proxy", dest="proxy",
                      metavar="str", help="Proxy address", default="")
    parser.add_option("-i", "--in", dest="infile",
                      help="Input CSV file", metavar="FILE")
```

```
    parser.add_option("-o", "--out", dest="out",
                      help="Output CSV file", metavar="FILE")
    (options, args) = parser.parse_args()
    if not options.infile:
        parser.error('Input CSV filename not given')

    if not options.outfile:
        parser.error('Output CSV filename not given')
    main(*parser.parse_args())
```

It is required to obtain twitter API key from https://apps.twitter.com/. Since there is a rate limit for the API endpoints, the whole hydrating will take days if multi-process is used. The fetched details include

> "text", "created_at", "geo", "lang", "place", "coordinates", "user.favourites_count", "user.statuses_count", "user.description", "user.location", "user.id", "user.created_at", "user.verified", "user.following", "user.url", "user.listed_count", "user.followers_count", "user.defaul`profile_image", "user.utc_offset", "user.friends_count", "user.default_profile", "user.name", "user.lang", "user.screen_name", "user.geo_enabled", "user.profile`background`color", "user.profile_image_url", "user.time_zone", "id", "favorite_count", "retweeted", "source", "favorited", "retweet_count"

almost all information of a tweet and a user. After hydaration, we have 4,695,447 tweets and 2,435,717 unique users in our dataset. The number is smaller than the ID set, since some tweets may get deleted or the users may choose to lock their account.

Before we can analyze the sentiment, we need a model. Here we implement a voting algorithem, which queries from all models we have mentioned before, and gives the confidence (score) of the sentiment. The voting algorithem is implemented in vote_classifier.py file.

```
from nltk.classify import ClassifierI
from statistics import mode


class VoteClassifier(ClassifierI):
    def __init__(self, *classifiers):
        """Voting Classifier

        Arguments:
            classifiers {ClassifierI} -- List of classifiers
        """
```

```python

        self._classifiers = classifiers

    def sentiment(self, features: list):
        """Classify the sentiment and get the confidence.

        Arguments:
            features {list} -- List of features

        Returns:
            str -- positive or negative
            int -- confidence
        """

        votes = [c.classify(features) for c in self._classifiers]
        choice_votes = votes.count(mode(votes))
        conf = choice_votes / len(votes)
        return mode(votes), conf

    def classify(self, features: list):
        """Classify the sentiment

        Arguments:
            features {list} -- List of features

        Returns:
            str -- positive or negative
        """

        votes = [c.classify(features) for c in self._classifiers]
        try:
            return mode(votes)
        except:
            return "pos"

    def confidence(self, features: list):
        """Get the confidence by talling the votes for and against the
        ↪   winning vote

        Arguments:
            features {list} -- List of features
```

```

        Returns:
            int -- confidence
        """
        votes = [c.classify(features) for c in self._classifiers]
        choice_votes = votes.count(mode(votes))
        conf = choice_votes / len(votes)
        return conf
```

Then, we train all the models we mentioned before. The positive and negative training sets are from GitHub Repo (aalind0/NLP-Sentiment-Analysis-Twitter). In the naive Bayes part, we utilized both NLTK and sklearn's classifiers. When training, the dataset is separated into two parts, the first 10,000 features as training set and the remaining as the testing set.

```
#!/usr/bin/env python3

# Training the classifiers and then pickling.
# Executing it sucks time. :P

import pickle
import random
from datetime import datetime

import nltk
from nltk.classify.scikitlearn import SklearnClassifier
from nltk.tokenize import word_tokenize
from sklearn.linear_model import LogisticRegression, SGDClassifier
from sklearn.naive_bayes import BernoulliNB, MultinomialNB
from sklearn.svm import SVC, LinearSVC

from utils import info, pickling
from vote_classifier import VoteClassifier

allowed_word_types = ["J", "R", "V"]


def read_corporas(positive: str="data/corporas/positive.txt", negative: str="data/corporas/negative.txt"):
    """Read corporas

```

```
    Keyword Arguments:
        positive {str} -- Path to positive text (default: {"data/corporas/positive.txt"})
        negative {str} -- Path to negative text (default: {"data/corporas/negative.txt"})

    Returns:
        list -- documents
        list -- all words
    """

    # Defining and Accessing the corporas.
    # In total, approx 10,000 feeds to be trained and tested on.
    all_words: list = []
    documents: list = []

    info("Accessing the corporas...")

    for p in open(positive, "r"):
        p = p.strip()
        documents.append((p, "pos"))
        words = word_tokenize(p)
        pos = nltk.pos_tag(words)
        for w in pos:
            if w[1][0] in allowed_word_types:
                all_words.append(w[0].lower())

    for p in open(negative, "r"):
        documents.append((p, "neg"))
        words = word_tokenize(p)
        pos = nltk.pos_tag(words)
        for w in pos:
            if w[1][0] in allowed_word_types:
                all_words.append(w[0].lower())

    return documents, all_words


def get_features(all_words: list, length: int=5000):
    """Calculate the most frequent words as features

```

```
    Arguments:
        all_words {list} -- All words of the string.

    Keyword Arguments:
        length {int} -- Length of features (default: {5000})

    Returns:
        list -- features
    """

    return list(nltk.FreqDist(all_words).keys())[:length]


def find_features(document: str, features: list):
    """The feature finding function, using tokenizing by word in the
    ↪ document.

    Arguments:
        document {str} -- Document
        features {list} -- List of features

    Returns:
        [type] -- [description]
    """

    words = word_tokenize(document)
    _features = {w: (w in words) for w in features}
    return _features


if __name__ == '__main__':
    info("Training classifiers. This may take few minutes to finish.")
    documents, all_words = read_corporas()

    info("Getting top 5000 words as features...")
    word_features = get_features(all_words)
    pickling("data/pickles/word_features5k.pickle", word_features)

    info("Tokenizing and finding features for training...")
    featuresets = [(find_features(rev, word_features), category)
                   for (rev, category) in documents]
```

```python

# Shuffling
random.shuffle(featuresets)
info("Length of the feature sets: " + str(len(featuresets)))

# Partitioning the training and the testing sets.
testing_set = featuresets[10000:]
training_set = featuresets[:10000]

print()
info("Training and successive pickling of the classifiers...")
info("This will take much time. Be patient.")

print()
info("Current Algorithm: " + "NLTK Original Naive Bayes")
nb_classifier = nltk.NaiveBayesClassifier.train(training_set)
info("Accuracy Percent:", str((nltk.classify.accuracy(
    nb_classifier, testing_set)) * 100))
pickling("data/pickles/original_naive_bayes.pickle", nb_classifier)

print()
info("Current Algorithm: " + "Sklearn Multinomial Naive Bayes")
mnb_classifier = SklearnClassifier(MultinomialNB())
mnb_classifier.train(training_set)
info("Accuracy Percent:", str(
    (nltk.classify.accuracy(mnb_classifier, testing_set)) * 100))
pickling("data/pickles/multinomial_naive_bayes.pickle", mnb_classifier)

print()
info("Current Algorithm: " + "Sklearn Bernoulli Naive Bayes")
bnb_classifier = SklearnClassifier(BernoulliNB())
bnb_classifier.train(training_set)
info("Accuracy Percent:", str(
    (nltk.classify.accuracy(bnb_classifier, testing_set)) * 100))
pickling("data/pickles/bernoulli_naive_bayes.pickle", bnb_classifier)

print()
info("Current Algorithm: " + "Sklearn Logistic Regression")
lr_classifier = SklearnClassifier(LogisticRegression())
lr_classifier.train(training_set)
info("Accuracy Percent:", str(
```

```
        (nltk.classify.accuracy(lr_classifier, testing_set)) * 100))
    pickling("data/pickles/logistic_regression.pickle", lr_classifier)

    print()
    info("Current Algorithm: " + "Sklearn SGD classifier")
    SGD_classifier = SklearnClassifier(SGDClassifier())
    SGD_classifier.train(training_set)
    info("Accuracy Percent:", str(
        (nltk.classify.accuracy(SGD_classifier, testing_set)) * 100))
    pickling("data/pickles/sgd.pickle", SGD_classifier)

    print()
    info("Current Algorithm: " + "Sklearn Linear SVC")
    linearSVC_classifier = SklearnClassifier(LinearSVC())
    linearSVC_classifier.train(training_set)
    info("Accuracy Percent:", str(
        (nltk.classify.accuracy(linearSVC_classifier, testing_set)) * 100))
    pickling("data/pickles/linear_svc.pickle", linearSVC_classifier)

    print()
    info("Current Algorithm: " + "Sklearn SVC")
    SVC_classifier = SklearnClassifier(SVC())
    SVC_classifier.train(training_set)
    info("Accuracy Percent:", str(
        (nltk.classify.accuracy(SVC_classifier, testing_set)) * 100))
    pickling("data/pickles/svc.pickle", SVC_classifier)

    print()
    # Voting classifier.
    info("All classifiers are trained. Evaluating the voted classifier...")
    voted_classifier = VoteClassifier(
        nb_classifier, mnb_classifier, bnb_classifier, lr_classifier,
         ↪  linearSVC_classifier, SGD_classifier, SVC_classifier)

    info("Accuracy percent:",
          str((nltk.classify.accuracy(voted_classifier, testing_set)) * 100))
```

The training output is like this,

```
# xiaoxing @ bogon in ~/Projects/tweet-sentiment-analysis on git:master x [15:49:34]
$ python train_classifiers.py
[INFO] (02:37:16) Training classifiers. This may take few minutes to finish.
[INFO] (02:37:16) Accessing the corporas...
[INFO] (02:37:31) Getting top 5000 words as features...
[INFO] (02:37:31) Tokenizing and finding features for training...
[INFO] (02:37:54) Length of the feature sets: 10662

[INFO] (02:37:54) Training and successive pickling of the classifiers...
[INFO] (02:37:54) This will take much time. Be patient.

[INFO] (02:37:54) Current Algorithm: NLTK Original Naive Bayes
[INFO] (02:39:00) Accuracy Percent: 71.6012084592145

[INFO] (02:39:00) Current Algorithm: Sklearn Multinomial Naive Bayes
[INFO] (02:39:39) Accuracy Percent: 71.45015105740181

[INFO] (02:39:39) Current Algorithm: Sklearn Bernoulli Naive Bayes
[INFO] (02:40:14) Accuracy Percent: 73.26283987915407

[INFO] (02:40:14) Current Algorithm: Sklearn Logistic Regression
[INFO] (02:40:47) Accuracy Percent: 74.77341389728097

[INFO] (02:40:47) Current Algorithm: Sklearn SGD classifier
/Users/xiaoxing/anaconda3/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:128: FutureWarni
: max_iter and tol parameters have been added in <class 'sklearn.linear_model.stochastic_gradient.SGDClassifier'>
 0.19. If both are left unset, they default to max_iter=5 and tol=None. If tol is not None, max_iter defaults to m
_iter=1000. From 0.21, default max_iter will be 1000, and default tol will be 1e-3.
  "and default tol will be 1e-3." % type(self), FutureWarning)
[INFO] (02:41:26) Accuracy Percent: 72.35649546827794

[INFO] (02:41:26) Current Algorithm: Sklearn Linear SVC
[INFO] (02:42:03) Accuracy Percent: 71.45015105740181

[INFO] (02:42:03) Current Algorithm: Sklearn SVC
[INFO] (02:43:28) Accuracy Percent: 48.338368580060425

[INFO] (02:43:28) All classifiers are trained. Evaluating the voted classifier...
[INFO] (02:44:34) Accuracy percent: 72.9607250755287
```

After we have the trained model, we shall apply the analysis to all tweets we have. Before that, a sentiment analysis function is implemented.

```python
import pickle
import random
from statistics import mode

import nltk
from nltk.classify import ClassifierI
from nltk.classify.scikitlearn import SklearnClassifier
from nltk.tokenize import word_tokenize
from sklearn.linear_model import LogisticRegression, SGDClassifier
from sklearn.naive_bayes import BernoulliNB, MultinomialNB
from sklearn.svm import SVC, LinearSVC

from vote_classifier import VoteClassifier


classifiers = {
    "NLTK Naive Bayes": "data/pickles/original_naive_bayes.pickle",
```

```python
    "Multinomial Naive Bayes": "data/pickles/multinomial_naive_bayes.pickle",
    "Bernoulli Naive Bayes": "data/pickles/bernoulli_naive_bayes.pickle",
    "Logistic Regression": "data/pickles/logistic_regression.pickle",
    "LinearSVC": "data/pickles/linear_svc.pickle",
    "SVC": "data/pickles/svc.pickle",
    "SGDClassifier": "data/pickles/sgd.pickle"
}

trained_classifiers = []
for classifier in classifiers.values():
    with open(classifier, "rb") as fh:
        trained_classifiers.append(pickle.load(fh))

voted_classifier = VoteClassifier(*trained_classifiers)

word_features5k_f = open("pickles/word_features5k.pickle", "rb")
word_features = pickle.load(word_features5k_f)
word_features5k_f.close()


def find_features(document: str, features: list):
    """The feature finding function, using tokenizing by word in the
    ↪ document.

    Arguments:
        document {str} -- Document
        features {list} -- List of features

    Returns:
        [type] -- [description]
    """

    words = word_tokenize(document)
    _features = {w: (w in words) for w in features}
    return _features


def sentiment(text):
    """Sentiment function.

    Arguments:
```

```
        text {str} -- Tweet string.

    Returns:
        str -- sentiment mode (pos or neg)
        int -- confidence
    """

    feats = find_features(text, word_features)
    return voted_classifier.sentiment(feats)
```

Then, use a multi-processing technique sentiment_calculation_multithread.py to apply to all rows.

```
print("Loading datasets... It may take a longer time.")
from sentiment import sentiment
import pickle
import pandas as pd
import numpy as np
from multiprocessing import cpu_count, Pool


print("The following result should be neg and 1.0")
print(sentiment("He is an incapable person. His projects are totally
↪   senseless."))

cores = cpu_count()  # Number of CPU cores on your system
partitions = cores // 4 or 1  # Define as many partitions as you want


def parallelize(data, func):
    data_split = np.array_split(data, partitions)
    pool = Pool(cores)
    data = pd.concat(pool.map(func, data_split))
    pool.close()
    pool.join()
    return data


def par_func(cs):
    print("Processing batch of", len(cs.index))
```

```python
    cs["sentiment"] = cs["text"].apply(sentiment)
    return cs


def main(i):
    print("Reading tweet file", i)
    cs = pd.read_csv("full_tweets_" + str(i) + ".csv")
    print("Total tweets", len(cs.index))
    print("Detecting sentiment in parallel...")
    cs = parallelize(cs, par_func)
    cs.to_csv("full_tweets_with_sentiment_" + str(i) + ".csv")
    print("---------")


if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description='Calculate tweets sentiment')
    parser.add_argument("index")
    args = parser.parse_args()

    main(args.index)
```

2.3.3 Accuracy

The accuracy varies because we randomly our training sets. But it should be stable at around $[65, 75]$. This is a demo run:

- NLTK Multinomial Naive Bayes: 72.9607250755287
- Sklearn Multinomial Naive Bayes: 70.2416918429003
- Sklearn Bernoulli Naive Bayes: 72.35649546827794
- Sklearn Logistic Regression: 70.69486404833837
- Sklearn Linear SVC: 67.97583081570997
- Sklearn SGD classifier: 67.06948640483384
- Voted Classifier: 71.75226586102718

2.3.4 Analysis

After we have the dataset and the sentiment data, we can now do analysis. Since we have limited memory, we need to optimize the usage. Here we mainly convert categorical data into binary numbers, and choose the correct data types (csv_optimize_to_pickle.py).

```
# coding: utf-8

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from tqdm import tqdm, tqdm_notebook
tqdm.pandas(tqdm_notebook)

df = pd.DataFrame()

# If failed to import, run
# `sed -e 's/\r/ /g' full_tweets_with_sentiment_7.csv > 7.csv` on all
for i in range(1, 8):
    print("Reading index", i)
    d = pd.read_csv(open("data/" + str(i) + ".csv", 'r'), encoding='utf-8',
                    engine='c', low_memory=False)
    df = pd.concat([d, df])
    print("File rows", len(d.index), "Total rows", len(df.index))


df['user.lang'] = df['user.lang'].astype('category')
df['user.time_zone'] = df['user.time_zone'].astype('category')
df['lang'] = df['lang'].astype('category')
df['source'] = df['source'].astype('category')
df['user.profile_background_color'] = df['user.profile_background_color'].astype(
    'category')

df['user.created_at'] = pd.to_datetime(df['user.created_at'])
df['created_at'] = pd.to_datetime(df['created_at'])

df['favorite_count'] = df['favorite_count'].apply(
    pd.to_numeric, downcast='unsigned')
df['user.listed_count'] = df['user.listed_count'].apply(
```

```
    pd.to_numeric, downcast='unsigned')

df.select_dtypes(include=['int64']).describe()  # Pick columns for improvement

df.to_pickle('data/all.pickle')  # Save the optimized object
```

And, run the score_calculation.py. This will do two parts, one convert the sentiment data into a score ranged from -1 to 1. If the score is between -1 and 0, it means negative, otherwise, positive. Second is to predict what the tweet is talking about, Trump or Hillary. Here we use a naive method – we have two preset keywords, one for Trump and another for Hillary. If the whole tweet is talking about Trump (no Hillary keyword appear), then the score will be multiplied by -1, while about Hillary by 1. We assume that negative tweets about Trump indicate the positive mind on Hillary, and vice versa. If a tweet is talking about two things, we will then split it into sentenses, and find the most object. After this transformation, the final score will still be ranged from -1 to 1, but now the $[-1, 0)$ part is about Trump, lower then more supportive, and the $(0, 1]$ part is about Hillary, higher then more supportive.

```
# coding: utf-8

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

print("Loading all models for sentiment calculation...")
from sentiment import sentiment as s

from tqdm import tqdm, tqdm_notebook
tqdm.pandas(tqdm_notebook)

df = pd.read_pickle("data/all.pickle")

import ast


def score(senti):
    # Since the previous sentiment calulation will save the score and category into a string, we need to eval it.
    senti = ast.literal_eval(senti)
    if senti[0] == 'neg':
```

```python
        return -1 * float(senti[1])
    return float(senti[1])


# Make the score range from [-1, 1]
df['sentiment_score'] = df['sentiment'].progress_apply(score)


def category(text):
    if any(map(lambda i: i in text, trump_keywords)) and any(map(lambda i: i
    ↪ in text, hillary_keywords)):
        return 0
    if any(map(lambda i: i in text, trump_keywords)):
        return -1
    elif any(map(lambda i: i in text, hillary_keywords)):
        return 1
    else:
        return 0


def sentiment(text):
    sen = s(text)
    if sen[0] == 'neg':
        return -1 * float(sen[1])
    return float(sen[1])


def category_and_score(entry):  # -1 trump, 1 hillary
    '''
    Find the category of tweet string.
    '''
    text = entry['text']
    if any(map(lambda i: i in text, trump_keywords)) and any(map(lambda i: i
    ↪ in text, hillary_keywords)):
        if "." in text:
            split_sentenses = text.split('.')
            s = {s: category(s) * sentiment(s) for s in split_sentenses}
            score = sum(s.values())

            if not score:
                return 0
```

```
            return score / len(s)
        else:
            return 0
    else:
        return float(category(text)) * float(entry['sentiment_score'])


trump_keywords = ['trump', 'youref ired', 'republi', 'gop']
hillary_keywords = ['hillary', 'madampresident', 'democrat']


df['score'] = df.progress_apply(category_and_score, axis=1)

df['score'].describe()

trump_scores = df[df['score'] < 0.0]['score']  # trump
hillary_scores = df[df['score'] > 0.0]['score']  # hillsry
non_scores = df[df['score'] == 0]['score']  # undeciable or no preference


trump_scores.describe()
hillary_scores.describe()
non_scores.describe()


results, edges = np.histogram(trump_scores, normed=True)
binWidth = edges[1] - edges[0]
plt.bar(edges[:-1], results * binWidth, binWidth)

results, edges = np.histogram(hillary_scores, normed=True)
binWidth = edges[1] - edges[0]
plt.bar(edges[:-1], results * binWidth, binWidth)
plt.show()
```

```
In [8]: df['score'] = df.progress_apply(category_and_score, axis=1)

100%|██████████| 4695447/4695447 [36:54<00:00, 2120.42it/s]
```

Here is some descriptive information on the score distribution.

- For Trump:

count : 106121.000000

mean : -0.915455

std : 0.151850

min : -1.000000

25% : -1.000000

50% : -1.000000

75% : -0.800000

max : -0.033333

- For Hillary:

count : 196243.000000

mean : 0.959576

std : 0.111571

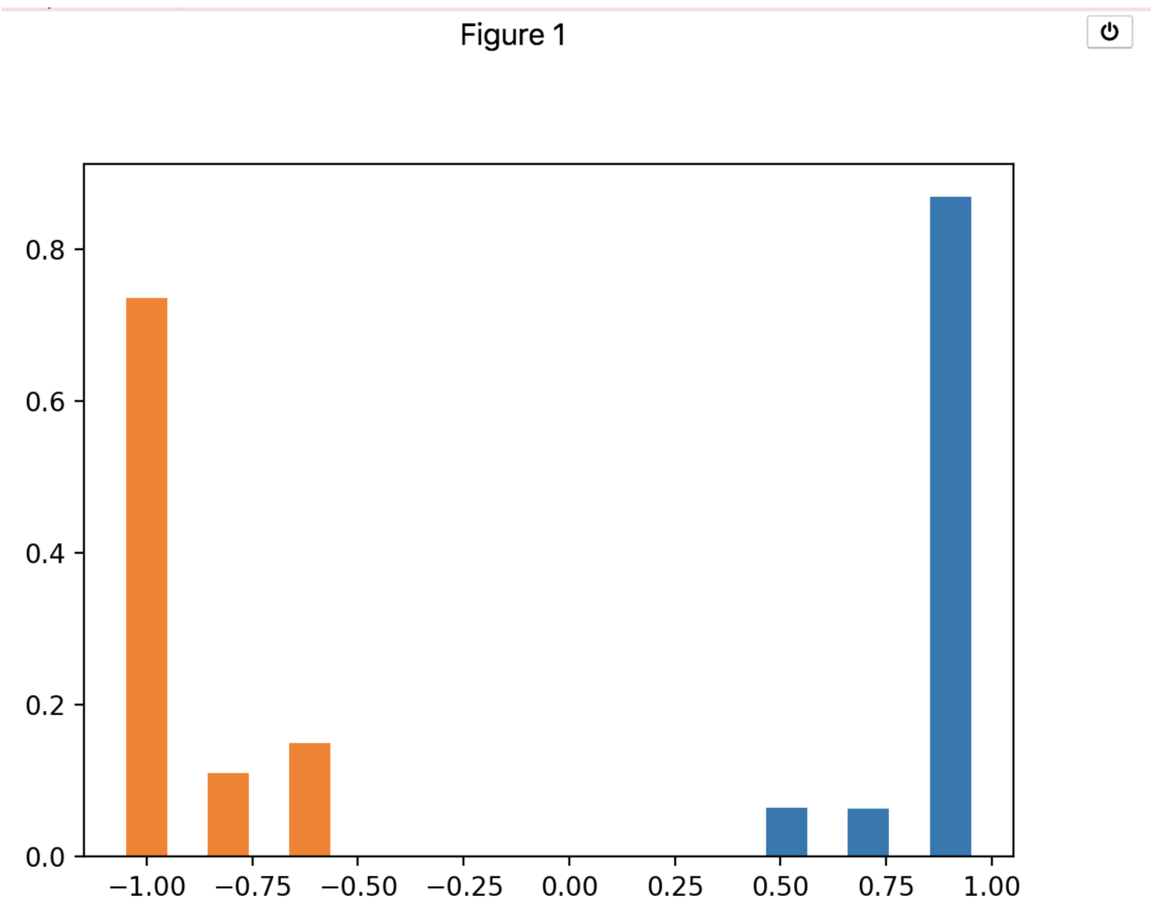
min : 0.028571

25% : 1.000000

50% : 1.000000

75% : 1.000000

max : 1.000000



2.4 Findings and Conclusion

As the analysis part (3.4) mentioned, for Trump, the closer the score is to -1, the more a citizen supporting Trump. Also, the supporting rule is the same for Hillary with the score closer to 1. From the result table given in the analysis part (3.4), tweeters for Hillary have a score averaged at 0.959 and Trump have a score averaged at -0.915. Compared Hillarys score to 1, and Trumps score to -1, one can draw the conclusion that from the sentiment data of the tweeter in the day of election, tweeters about Hillary are slightly more positive than tweeters about Trump. Figure 1 shows the clearer score distribution of Trump and Hillary, we can see that the score distribution of Hillary are much closer to 1 compared to the score distribution of Trump to -1, which indicate the same results of comparing the average score of the two candidates.

Although the result of tweeter sentiment in this report is different from the real election result, this can be explained by the frequent users of the tweeter. Most people who frequently use tweeter live in the big cities, but in the election, all the states in the United States vote for the president, and not all the people utilize tweeter. Therefore, the prediction by tweeter may contain some bias. However, the close disparity in tweet sentiment is similar to the actual election result, which indicates a connection.

To further improved the result, having more data from different days may go a step further from the results and improve the accuracy of the prediction. Further study can also be done on the geographical difference of the score.

Chapter 3

Part 3: Sailor2017 Jupyter Notebook Codes

3.1 Lesson 1: Rule Based

lesson1_rulebased

April 29, 2018

```
In [1]: # Run this every time you open the spreadsheet
        % load_ext autoreload
        % autoreload 2
        from collections import Counter
        import lib
```

1 Load and inspect the data

```
In [2]: # Load the data.
        tweets, test_tweets = lib.read_data()
```

```
In [3]: # The variable "tweets" is a list of tweets.
        # Mini-exercise 1: Print the number of tweets in the list

        print("Number of tweets:", len(tweets))

        # Mini-exercise 2: Assign the 10th tweet to the variable "tweet" and print it

        tweet = tweets[9]
        print(tweet)

        # To view the category of a tweet, we access the attribute tweet.category
        # Mini-exercise 3: Print the category of this tweet.

        category = tweet.category
        print("Category of the tweet:", category)
```

```
Number of tweets: 1120
food prep yoga classes supply runs organization
Category of the tweet: Food
```

```
In [4]: # This function prints out a table containing all the tweets, along with their category
        labels
        lib.show_tweets(tweets)
```

```
<IPython.core.display.HTML object>
```

2 Python refresher

First, let's do some exercises to refresh our memory of a few Python concepts.

2.0.1 Functions

A Python function is written like this:

```
def add_one(x):
    return x+1
```

The name of the function is `add_one`, `x` is the input variable, and the `return` keyword tells us what to give as output.

```
In [5]: # Exercise 1. Define a function called "square_minus_1" that takes one variable (x),
        # squares it, subtracts 1, and returns the result.

        #### YOUR CODE STARTS HERE ####

        def square_minus_1(x):
            return x**2 - 1;

        #### YOUR CODE ENDS HERE ####

        print("Testing:")
        for x in [3,-4,6.5,0]:
            print(str(x) + " -> " + str(square_minus_1(x)))
            print("CORRECT" if square_minus_1(x)==(x**2-1) else "INCORRECT")
```

```
Testing:
3 -> 8
CORRECT
-4 -> 15
CORRECT
6.5 -> 41.25
CORRECT
0 -> -1
CORRECT
```

2.0.2 If-else statements

An if/else statement looks like this:

```
if electoral_votes >= 270:
    print "You win the election"
else:
    print "You lose the election"
```

The if-statement is evaluated (`electoral_votes >= 270`); if it's true then the code under the `if` is executed, if it's false then the code under the `else` is executed.

```
In [6]: # Exercise 2. Define a function called "contains_ss" that takes one variable (word)
        # and returns True if the word contains a double-s and False if it doesn't.
        # Hint: to test whether a string e.g. "ss" is inside another string variable e.g. word,
        write
        #    if "ss" in word:

        #### YOUR CODE STARTS HERE ####

        def contains_ss(word):
            return "ss" in word
```

```
    #### YOUR CODE ENDS HERE ####

    print("Testing:")
    for word in ["computer", "science", "lesson"]:
        print("%s ->" % word, contains_ss(word))
        print("CORRECT" if contains_ss(word) == ("ss" in word) else "INCORRECT")
```

```
Testing:
computer -> False
CORRECT
science -> False
CORRECT
lesson -> True
CORRECT
```

2.1 More complex if-else statements

Maybe you want to check *several* conditions? You can use an if/elif/else statement.

```
if teamA_score > teamB_score:
    print "Team A wins"
elif teamA_score < teamB_score:
    print "Team B wins"
else:
    print "It's a tie!"
```

`elif` stands for "else if". In fact, the above code is just a neater way of writing this:

```
if teamA_score > teamB_score:
    print "Team A wins"
else:
    if teamA_score < teamB_score:
        print "Team B wins"
    else:
        print "It's a tie!"
```

You can have as many `elif` statments as you like. These are useful for when you want several options.

```
In [7]: # Exercise 3. Define a function called "grade" that takes one input (score).
        # If score >= 90, return the string "A"
        # Otherwise, if score >= 80, return the string "B"
        # Otherwise, if score >= 70, return the string "C"
        # Otherwise, if score >= 60, return the string "D"
        # Otherwise, if score >= 50, return the string "E"
        # Otherwise, return the string "F"

        #### YOUR CODE STARTS HERE ####

        def grade(score):
            if score >= 90:
                return "A"
            elif score >= 80:
                return "B"
            elif score >= 70:
                return "C"
```

```
    elif score >= 60:
        return "D"
    elif score >= 50:
        return "E"
    else:
        return "F"

#### YOUR CODE ENDS HERE ####


print("Testing:")
for (score, g) in [(77, "C"), (80, "B"), (32, "F"), (100, "A"), (69, "D")]:
    print("%i -> %s" % (score, grade(score)))
    print("CORRECT" if grade(score) == g else "INCORRECT")
```

```
Testing:
77 -> C
CORRECT
80 -> B
CORRECT
32 -> F
CORRECT
100 -> A
CORRECT
69 -> D
CORRECT
```

3 Write a rule-based tweet classifier

Time to write our rule-based classifier! The function outline below uses a `if/elif/else` statement to return the predicted category of a tweet.

Fill in the missing `if` and `elif` statements with something sensible (there is no one right answer)!

Start with something simple; we'll build it into something more complicated later.

```
In [8]: def classify_rb(tweet):
            tweet = str(tweet).lower()  # this makes the tweet lower-case, so we don't have to
        worry about matching case

            if "medicine" in tweet:
                return "Medical"
            elif "power" in tweet:
                return "Energy"
            elif "water" in tweet:
                return "Water"
            elif "milk" in tweet:
                return "Food"
            else:
                return "None"
```

4 Test your rule-based classifier on some examples

Run the cell below to see the results of your rule-based classifier. You should see a table showing each tweet, along with its true category and the category predicted by your system.

Which types of tweets does your system get right? Which types of tweets does your system get wrong and why?

How would you measure the accuracy of your system?

```
In [9]: predictions = [(tweet, classify_rb(tweet)) for tweet in test_tweets]  # a list of
        (tweet, prediction) pairs

        lib.show_predictions(predictions, True)
```

```
<IPython.core.display.HTML object>
```

5 Break your rule-based classifier!

It's time to FOOL THE RULES!

You'll be deliberately trying to break each others' rule-based classifiers by writing tricky tweets that fool your neighbor's rule-based classifier. Once your own classifier has been fooled by a tricky tweet, it's your job to amend the rules in your classifier to account for the new case.

```
In [10]: def classify_rb_game(tweet):
             tweet = str(tweet).lower()  # this makes the tweet lower-case, so we don't have to
         worry about matching case

             if "medicine" in tweet:
                 return "Medical"
             elif "power" in tweet:
                 return "Energy"
             elif "water" in tweet:
                 return "Water"
             elif "milk" in tweet:
                 return "Food"
             else:
                 return "None"
             pass
```

5.0.1 Write a tweet about Food that will be misclassified

Below, write a disaster-scenario tweet about Food that the classification function above will get wrong (i.e. fail to recognize it's about food).

Hint: think of less-obvious food-related keywords that aren't included in the rule-based system above.

Then run the cell - make sure the tweet is classified as something other than Food!

```
In [11]: food_tweet = "I love watermelon."
         print("This tweet is classified as: %s\n" % classify_rb_game(food_tweet))
```

```
This tweet is classified as: Water
```

5.0.2 Write a tweet about Energy that will be misclassified

```
In [12]: energy_tweet = "What's the medicine to energy crisis?"
         print("This tweet is classified as: %s\n" % classify_rb_game(energy_tweet))
```

```
This tweet is classified as: Medical
```

5.0.3 Write a tweet about Water that will be misclassified

```
In [13]: water_tweet = "What powers our life is water."
         print("This tweet is classified as: %s\n" % classify_rb_game(water_tweet))
```

```
This tweet is classified as: Energy
```

5.0.4 Write a tweet about Medical that will be misclassified

```
In [14]: medical_tweet = "Aspirin is not water, we should be careful."
         print("This tweet is classified as: %s\n" % classify_rb_game(medical_tweet))
```

```
This tweet is classified as: Water
```

5.0.5 Write a tweet NOT about Food, that will be falsely classified as Food

Below, write a disaster-scenario tweet that is NOT about Food, but that the classifier above will classify as Food.

Hint: you want to trick the classifier into thinking you're talking about food when you're not. Look at the keywords the rule-based system associates with food. Can you find a way to use them while actually talking about not-food?

- For example, if the system looks for the word "food" you could write *"Waiting out #Sandy by reading Plato. Food for thought."*
- If the system looks for the word "cook", you could write *"I hear the power's out in Cook County."*
- More simply, you could mention food incidentally but the real subject of the tweet is something else e.g. *"Was out food shopping when I heard about the power outage on the news. Hope everyone's OK."*

Then run the cell - make sure the tweet is classified as Food!

```
In [15]: not_food_tweet = "Milky way is beautiful"
         print("This tweet is classified as: %s\n" % classify_rb_game(not_food_tweet))
```

```
This tweet is classified as: Food
```

5.0.6 Write a tweet NOT about Energy, that will be falsely classified as Energy

```
In [16]: not_energy_tweet = "I am out of power"
         print("This tweet is classified as: %s\n" % classify_rb_game(not_energy_tweet))
```

```
This tweet is classified as: Energy
```

5.0.7 Write a tweet NOT about Water, that will be falsely classified as Water

```
In [17]: not_water_tweet = "I love watermelon"
         print("This tweet is classified as: %s\n" % classify_rb_game(not_water_tweet))
```

```
This tweet is classified as: Water
```

5.0.8 Write a tweet NOT about Medical, that will be falsely classified as Medical

```
In [18]: not_medical_tweet = "skdfjmedicinesjwjer"
         print("This tweet is classified as: %s\n" % classify_rb_game(not_medical_tweet))
```

```
This tweet is classified as: Medical
```

lesson2_evaluation

April 29, 2018

```
In [1]: # Run this every time you open the spreadsheet
        % load_ext autoreload
        % autoreload 2
        import lib
```

1 Load the data and our rule-based classifier

```
In [2]: # Load the data.
        # This function returns "tweets" and "test_tweets", both lists of tweets
        import nltk
        nltk.download('punkt')
        tweets, test_tweets = lib.read_data()
```

```
[nltk_data] Downloading package punkt to /Users/xiaoxing/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

```
In [3]: def classify_rb(tweet):
            tweet = str(tweet).lower()  # this makes the tweet lower-case, so we don't have to
        worry about matching case

            if "medicine" in tweet or "first aid" in tweet:
                return "Medical"
            elif "power" in tweet or "battery" in tweet:
                return "Energy"
            elif "water" in tweet or "bottled" in tweet:
                return "Water"
            elif "food" in tweet or "perishable" in tweet or "canned" in tweet:
                return "Food"
            else:
                return "None"
```

2 Python refresher

Let's review some Python concepts before we write our evaluation code.

2.0.1 Lists

In Python, a *list* is an ordered collection of items. The items can be strings, numbers, booleans, or any other kind of Python object.

You can create lists like this:

```
integer_list = [5, 6, 7, 8]
string_list = ['hello', 'world']
bool_list = [False, True, False, False, True]
```

If you want a list of the numbers up to (but not including) 10, you can use the `range` function.

```
upto10_list = range(10)
```

This gives you [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].

```
In [4]: # Exercise 1(a).
        # Create a list called "my_numbers" that contains the numbers from 0 to 6 (inclusive),
        and then print it
        my_numbers = [0, 1, 2, 3, 4, 5, 6]
        print(my_numbers)
```

```
[0, 1, 2, 3, 4, 5, 6]
```

```
In [15]: # Exercise 1(b).
         # Now use the range() function to create "my_numbers", and print the result.
         # It should match the previous cell.
         # Hint: look carefully at the range(10) example above.
         my_numbers = list(range(7))
         print(my_numbers)
```

```
[0, 1, 2, 3, 4, 5, 6]
```

2.0.2 For loops

In Python, a *for loop* allows you to iterate over a list.

```
shopping_list = ['bread', 'bananas', 'milk']

for item in shopping_list:
    print item
```

For example, the code above prints out the following output:

```
bread
bananas
milk
```

```
In [6]: # Exercise 2.
        # Write a for-loop that iterates through my_numbers, and prints the square of each
        number
        # You should see the following numbers print out, one per line: 0, 1, 4, 9, 16, 25, 36
        for item in my_numbers:
            print(item**2)
```

```
0
1
4
9
16
25
36
```

```
In [7]: # Exercise 3.
        # Use a for-loop to calculate the sum of the squares of my_numbers.
        # Save the result in a variable called "sum_squares".
        # Hint: start by setting sum_squares to 0 before starting the for-loop.

        #### YOUR CODE STARTS HERE ####
        sum_squares = 0
        for item in my_numbers:
            sum_squares += item**2

        #### YOUR CODE ENDS HERE ####

        print("Testing: sum_squares = %i" % sum_squares)
        print("CORRECT" if sum_squares == 91 else "INCORRECT")
```

```
Testing: sum_squares = 91
CORRECT
```

2.0.3 Incrementing

If you have an integer variable e.g. x=3 and you want to increase x by 1 (which is called *incrementing*), then you can write

```
x = x+1
```

or, in shorthand:

```
x += 1
```

This can be useful when you're using x to count something. For example:

```
ages = [7, 14, 23, 3, 10, 19]

num_adults = 0
for age in ages:
    if age >= 18:
        num_adults += 1

print num_adults
```

What should this code print out?

```
In [8]: # Exercise 4.
        # Count the number of Weasleys in the list of characters, and save the result to the
        variable "num_weasleys".
        # Use incrementation with the "x += 1" notation.

        characters = ['Harry Potter', 'Ron Weasley', 'Albus Dumbledore', 'Ginny Weasley', 'Percy
        Weasley', 'Hermione Granger',
                      'Fred Weasley', 'George Weasley']

        #### YOUR CODE STARTS HERE ####
        num_weasleys = 0
        for item in characters:
            if "Weasley" in item:
                num_weasleys += 1

        #### YOUR CODE ENDS HERE ####

        print("Testing: num_weasleys = %i" % num_weasleys)
        print("CORRECT" if num_weasleys == 5 else "INCORRECT")
```

```
Testing: num_weasleys = 5
CORRECT
```

2.0.4 Testing for equality and inequality

Sometimes you want to check if two values are equal, perhaps using an `if` statement. To check for equality you need to use a *double* equals sign `==`.

```
x = 5
y = 8
if x == y:
    print "x and y are equal"
```

To check for *inequality*, i.e. if two things aren't equal, use `!=`.

```
x = 5
y = 8
if x != y:
    print "x and y are NOT equal"
```

In [9]:
```
# Exercise 5.
# Use a for-loop, incrementation and equality testing to count the number of cats in my
list of pets.
# Assign the result to the variable "num_cats"

my_pets = ['cat', 'lizard', 'cat', 'dog', 'cat', 'snake', 'dog', 'cat', 'dog', 'parrot']

#### YOUR CODE STARTS HERE ####
num_cats = 0
for item in my_pets:
    if "cat" == item:
        num_cats += 1


#### YOUR CODE ENDS HERE ####

print("Testing: num_cats = %i" % num_cats)
print("CORRECT" if num_cats == 4 else "INCORRECT")
```

```
Testing: num_cats = 4
CORRECT
```

In [10]:
```
# Exercise 6.
# Use a for-loop, incrementation and inequality testing to count the number of pets that
are neither cats nor dogs.
# Assign the result to the variable "num_unusual".

#### YOUR CODE STARTS HERE ####
num_unusual = 0
for item in my_pets:
    if "cat" != item and "dog" != item:
        num_unusual += 1

#### YOUR CODE ENDS HERE ####

print("Testing: num_unusual = %i" % num_unusual)
print("CORRECT" if num_unusual == 3 else "INCORRECT")
```

```
Testing: num_unusual = 3
CORRECT
```

3 Measure the accuracy of your rule-based classifier

Complete the function below to calculate the Precision, Recall and F1 for a given category (e.g. Food)

```
In [11]: def evaluate(predictions, c):
             """This function calculate the precision, recall and F1 for a single category c
         (e.g. Food)
             Inputs:
                 predictions: a list of (tweet, predicted_category) pairs
                 c: a category
             Returns:
                 The F1 score.
             """

             # Initialize variables to count the number of true positives, false positives and
         false negatives
             true_positives = 0.0
             false_positives = 0.0
             false_negatives = 0.0

             # Iterate through the tweets, counting the number of true positives, false positives
         and false negatives
             for (tweet, predicted_category) in predictions:
                 true_category = tweet.category

                 # Hint: true positives for category c are tweets that have
                 # true category c and predicted category c
                 if c == predicted_category and tweet.category == c:
                     true_positives += 1

                 # Finish the statement: false negatives for category c are tweets that have
                 # true category not c and predicted category not c
                 elif c != predicted_category and tweet.category !=c:
                     false_negatives += 1

                 # Finish the statement: false positives for category c are tweets that have
                 # true category c and predicted category not c
                 elif c != predicted_category and tweet.category ==c:
                     false_positives += 1

             # Before we calculate Precision, Recall and F1 we need to check whether
         true_positives = 0. Why?
             if true_positives == 0:
                 precision = 0.0
                 recall = 0.0
                 f1 = 0.0
             else:
                 # Calculate Precision, Recall and F1
                 # Consult the formulae on the slides
                 precision = true_positives / (false_positives + true_positives)
                 recall = true_positives / (false_negatives + true_positives)
                 f1 = 2 * precision * recall / (precision + recall)

             # Print the category name, Precision, Recall and F1
             print(c)
             print("Precision: ", precision)
             print("Recall: ", recall)
             print("F1: ", f1)
             print("")

             # Return the F1 score
             return f1


         predictions = [(tweet, classify_rb(tweet)) for tweet in test_tweets]  # Make a list of
         (tweet, predicted_category) pairs
```

```
# Get the F1 scores for each category
food_f1 = evaluate(predictions, "Food")
water_f1 = evaluate(predictions, "Water")
energy_f1 = evaluate(predictions, "Energy")
medical_f1 = evaluate(predictions, "Medical")
none_f1 = evaluate(predictions, "None")
```

```
Food
Precision:  0.8217054263565892
Recall:  0.4608695652173913
F1:  0.5905292479108635

Water
Precision:  0.95
Recall:  0.07063197026022305
F1:  0.1314878892733564

Energy
Precision:  0.4
Recall:  0.06477732793522267
F1:  0.11149825783972125

Medical
Precision:  0.5384615384615384
Recall:  0.025454545454545455
F1:  0.048611111111111105

None
Precision:  0.5569620253164557
Recall:  0.21359223300970873
F1:  0.30877192982456136
```

Complete the cell below to calculate the average F1 score, which should be the average of the F1 scores for each category.

```
In [12]: average_f1 = (food_f1 + water_f1 + energy_f1 + medical_f1 + none_f1)/5
         print("Average F1: ", average_f1)
```

```
Average F1:  0.2381796871919227
```

3.1 Look at the confusion matrix

- *Rows* represent the *true category* of the tweet
- *Columns* represent the *predicted category* from your classifier
- So numbers on the diagonal represent correct classifications, and off-diagonal numbers represent misclassification

```
In [13]: lib.show_confusion_matrix(predictions)
```

```
<IPython.core.display.HTML object>
```

3.2 Look at the predictions

```
In [14]: lib.show_predictions(predictions)
```

```
<IPython.core.display.HTML object>
```

3.3 Lesson 3: Navie Bayes Exerciese

lesson3_naivebayes_exercises

April 29, 2018

```
In [1]: # Run this every time you open the spreadsheet
        %load_ext autoreload
        %autoreload 2

        from collections import Counter
        import lib
```

1 Load the content of the boxes

```
In [2]: # Load the data.
        # This function returns box_a and box_b, both lists of colors of the balls in each box
        box_a, box_b = lib.get_box_contents()

        print("Number of balls in box A: %d" % len(box_a))
        print("Number of balls in box B: %d" % len(box_b))
```

```
Number of balls in box A: 100
Number of balls in box B: 100
```

2 Python concepts

Let's review and look at some new Python concepts before we implement the box and ball examples.

2.0.1 Dictionaries

In Python, a *dict* is a collection of items in which each element can be accessed by a *key*. The *key* is typically a string and the items can be of any data type, e.g., booleans, integers, strings. Each key can be used for only one item.

You can create dictionaries like this:

```
west_coast_state_capitals = {"California": "Sacramento", "Oregon": "Salem", "Washington": "O
prices = {"spaghetti": 2.50, "milk": 2.00, "peanut butter": 2.75, "avocado": 0.85, "bread":
```

To access a value in a dictionary, use the name of the dictionary and put they *key* in squared brackets:

```
west_coast_state_capitals["California"] # returns "Sacramento"
prices["milk"] # returns 2.00
```

```
In [3]: # Exercise 1.
        # Create a dictionary called "authors" that maps the following book titles to their
        authors.
        #
        # Harry Potter - J.K. Rowling
        # The Casual Vacancy - J.K. Rowling
        # The Hunger Games - Suzanne Collins
        # Never Let Me Go - Kazuo Ishiguro
        # The Catcher in the Rye - J.D. Salinger
        #
        # Then print the author of "The Catcher in the Rye" and "Harry Potter" using your
        dictionary.

        #### YOUR CODE STARTS HERE ####
        authors={"Harry Potter":"J.K. Rowling","The Casual Vacancy":"J.K. Rowling","The Hunger
        Games":"Suzanne Collins"
                ,"Never Let Me Go":"Kazuo Ishiguro","The Catcher in the Rye":"J.D. Salinger"}

        #### YOUR CODE ENDS HERE ####

        print("CORRECT" if authors["Harry Potter"] == "J.K. Rowling" else "INCORRECT")
        print("CORRECT" if authors["The Casual Vacancy"] == "J.K. Rowling" else "INCORRECT")
        print("CORRECT" if authors["Never Let Me Go"] == "Kazuo Ishiguro" else "INCORRECT")
```

```
CORRECT
CORRECT
CORRECT
```

Adding, updating, and deleting items from dictionaries You can also add, change and delete items after you created an dictionary.

For example, the following code creates an empty dictionary *prices* and then adds two items to it.

```
prices = {}
prices["milk"] = 2.00
prices["avocado"] = 0.85

print prices # outputs {'avocado': 0.85, 'milk': 2.0}
```

To update an item in a dictionary, simply assign a new value to it:

```
prices = {}
prices["milk"] = 2.00
prices["avocado"] = 0.85
print prices # outputs {'avocado': 0.85, 'milk': 2.0}


prices["milk"] = 2.25
print prices # outputs {'avocado': 0.85, 'milk': 2.25}
```

Tto delete an item from a dictionary, use the *del* keyword as in the following snippet:

```
prices = {}
prices["milk"] = 2.00
prices["avocado"] = 0.85
```

```
print prices # outputs {'avocado': 0.85, 'milk': 2.0}

del prices["milk"]
print prices # outputs {'avocado': 0.85}
```

Iterating through dictionaries You can also iterate through all items in a dictionary with a for-loop. When you loop through a dictionary, the key is assigned to the loop variable during each iteration.

```
prices = {"spaghetti": 2.50, "milk": 2.00, "peanut butter": 2.75}

for product in prices:
   print product, prices[product]
   # Sidenote: You can print multiple variables in the same line
   # by separating them with a comma.
```

This program outputs something like: (order may vary)

```
spaghetti 2.5
milk 2.0
peanut butter 2.75
```

```
In [4]: # Imagine you are running a store that sells spaghetti, milk, peanut butter, avocados,
        and bread, and you
        # store the prices for these products in the following dictionary.

        prices = {"spaghetti": 2.50, "milk": 2.00, "peanut butter": 2.75, "avocado": 0.85,
        "bread": 3.25}

        # Exercise 2(a).

        # Your distributor increased prices on all products by 25 cents, so you'll have to
        increase your prices
        # by 25 cents as well. Increase every value in the prices dictionary by 25 cents.

        #### YOUR CODE STARTS HERE ####

        for product in prices:
            prices[product]+=0.25
            pass

        #### YOUR CODE ENDS HERE ####

        print(prices)

        print("CORRECT" if prices["spaghetti"] == 2.75 else "INCORRECT")
        print("CORRECT" if prices["milk"] == 2.25 else "INCORRECT")
        print("CORRECT" if prices["bread"] == 3.50 else "INCORRECT")
```

```
{'spaghetti': 2.75, 'milk': 2.25, 'peanut butter': 3.0, 'avocado': 1.1, 'bread': 3.5}
CORRECT
CORRECT
CORRECT
```

```
In [5]: # Exercise 2(b).

        # You added bananas to your inventory and you sell them for 95 cents. Add a new entry
        # for bananas to the prices dictionary.

        #### YOUR CODE STARTS HERE ####
        prices["bananas"]=0.95

        #### YOUR CODE ENDS HERE ####

        print(prices)
        print("CORRECT" if prices["bananas"] == 0.95 else "INCORRECT")
```

```
{'spaghetti': 2.75, 'milk': 2.25, 'peanut butter': 3.0, 'avocado': 1.1, 'bread': 3.5,
'bananas': 0.95}
CORRECT
```

```
In [6]: # Exercise 2(c).

        # You are no longer selling peanut butter. Remove the entry for peanut butter from
        prices.

        #### YOUR CODE STARTS HERE ####
        del prices["peanut butter"]

        #### YOUR CODE ENDS HERE ####

        print(prices)
```

```
{'spaghetti': 2.75, 'milk': 2.25, 'avocado': 1.1, 'bread': 3.5, 'bananas': 0.95}
```

2.0.2 Counter

Python comes with a special dictionary type, the `Counter` type, which makes it easier to work with counts.

A Counter works just like a dictionary but instead of giving an error when you use a key for which no entry exists, it will return 0.

To use Counters, you first have to run the following import statement.

```
from collections import Counter
```

Let's create a Counter to keep track of how many birds I've seen. Below, I create a new empty Counter called `bird_counter`. Note that if I ask the counter how many sparrows I've seen, it returns 0 even though `"sparrow"` is not in the keys.

```
bird_counter = Counter()
print bird_counter["sparrow"] # outputs 0
```

I just spotted a finch! Let's increment the counter.

```
bird_counter["finch"] += 1
print bird_counter["finch"] # outputs 1
```

```
In [7]: # Exercise 3.
        # Write code that counts how many of each type of fruit there are in the fruit basket.
        # Use a Counter to store the counts and print the final Counter object.
```

```
# Hint: Use a for-loop to iterate through all the items in fruit_basket.

from collections import Counter

fruit_basket = ["apple", "banana", "plum", "apple", "apricot", "plum", "apple", "apple",
"apricot", "apricot"]

#### YOUR CODE STARTS HERE ####
fruit_counter = Counter()
for fruit in fruit_basket:
    fruit_counter[fruit] += 1

print(fruit_counter)


#### YOUR CODE ENDS HERE ####
```

```
Counter({'apple': 4, 'apricot': 3, 'plum': 2, 'banana': 1})
```

2.0.3 Turning lists into Counters

Counters come with several other useful features. One of them is that you can automatically turn a list into a counter. For example, the following snippet counts how many of each letter there are in the list my_letters.

```
my_letters = ["a", "b", "b", "c", "c", "c", "d", "d", "d", "d"]
letter_counter = Counter(my_letters)

print letter_counter # outputs Counter({'d': 4, 'c': 3, 'b': 2, 'a': 1})
```

```
In [8]: # Exercise 4.
        # Re-implement the program from Exercise 3 without using a for-loop.

        from collections import Counter

        fruit_basket = ["apple", "banana", "plum", "apple", "apricot", "plum", "apple", "apple",
        "apricot", "apricot"]

        #### YOUR CODE STARTS HERE ####
        fruit_counter = Counter(fruit_basket)
        print(fruit_counter)

        #### YOUR CODE ENDS HERE ####
```

```
Counter({'apple': 4, 'apricot': 3, 'plum': 2, 'banana': 1})
```

2.0.4 Iterating through counters

You can iterate through a Counter just like a dictionary. If you use a for-loop with a Counter, it will loop through all keys.

```
prices = Counter({"spaghetti": 2.50, "milk": 2.00, "peanut butter": 2.75})

for product in prices:
  print product
```

This program will print something like: (the order may vary)

```
spaghetti
milk
peanut butter
```

You can also get a list of all values stored in a Counter using the *.values()* method.

```
prices = Counter({"spaghetti": 2.50, "milk": 2.00, "peanut butter": 2.75})

vals = prices.values()

print vals # outputs [2.5, 2.0, 2.75] (the order may vary)
```

```
In [9]: # Exercise 6.

        # The following counter stores how many rooms of each type a hotel has.

        hotel_rooms = Counter({"1 queen-sized bed": 25, "1 king-sized bed": 14,
                               "2 queen-sized beds": 12, "Honeymoon suite": 1,
                               "Presidential suite": 1})

        # Write some code that prints each room type and how many rooms of each type there are.
        e.g., "Presidential suite 1"


        #### YOUR CODE STARTS HERE ####

        for room, count in hotel_rooms.items():
            print(room, count)

        #### YOUR CODE ENDS HERE ####
```

```
1 queen-sized bed 25
1 king-sized bed 14
2 queen-sized beds 12
Honeymoon suite 1
Presidential suite 1
```

2.0.5 Computing the sum of a list of numbers

Sometimes it can also be really useful to compute the sum of a list of numbers. For example, assume that the following list stores the weights of products in a package and you want to compute the total weight of the package.

```
weights = [3, 4, 5, 1, 2, 9, 12, 11]
```

Python comes with a function `sum` that allows you to quickly sum over a list of numbers.

```
total_weight = sum(weights)
print total_weight # outputs 47
```

You can also use this function together with the values of a Counter. For example, the following code computes how much it would cost if one bought every item in the prices Counter.

```
prices = Counter({"spaghetti": 2.50, "milk": 2.00, "peanut butter": 2.75})

vals = prices.values()

total = sum(vals)

print total # outputs 7.25
```

```
In [10]: # Exercise 6(a).
         # Compute the sum of all numbers from 1 to 10 and assign to the variable "total"

         #### YOUR CODE STARTS HERE ####

         numbers = list(range(1, 11))
         total = sum(numbers)

         #### YOUR CODE ENDS HERE ####

         print(total)
         print("CORRECT" if total == 55 else "INCORRECT")
```

```
55
CORRECT
```

```
In [11]: # Exercise 6(b).
         # Store the counts of each type of pet in a Counter and use that counter to compute the
         total number of pets.
         # Save the result in the variable "total"

         my_pets = ['cat', 'lizard', 'cat', 'dog', 'cat', 'snake', 'dog', 'cat', 'dog', 'parrot']

         #### YOUR CODE STARTS HERE ####

         pet_counter = Counter(my_pets)
         vals = pet_counter.values()
         total = sum(vals)

         #### YOUR CODE ENDS HERE ####

         print(total)
         print("CORRECT" if total == 10 else "INCORRECT")
```

```
10
CORRECT
```

2.0.6 Dividing integers in Python

One of the pecularities of Python (and some other programming languages) is that if you divide two integers, it will always return the results rounded down to the next integer and never a decimal number.

For example, if you compute 1/2, it will return 0.

This can be particularly problematic when we are dealing with fractions or percentatages, as we often do when we compute probabilities. The easiest way to get around this is by turning one of the two numbers into a decimal number with the function *float*. This will change the representation of the number from an integer to a decimal number and when you then run the division, it will return a decimal. For example, consider the following two divisions:

```
res1 = 1/10

res2 = float(1)/10

print res1
print res2
```

This program will produce the following output:

```
0
0.1
```

```
In [12]: # Exercise 7.
         # Divide each number in the following list by 2 and print it.

         numbers = [3, 5, 6, 7, 9]

         #### YOUR CODE STARTS HERE ####

         for number in numbers:
             print(number / 2)

         #### YOUR CODE ENDS HERE ####

         # The output should be 1.5, 2.5, 3.0, 3.5, 4.5
```

```
1.5
2.5
3.0
3.5
4.5
```

```
In [13]: # Exercise 8.
         # Compute the fraction of each type of animal (e.g., the fraction of lizards = 1/10 =
         0.1)
         # and store them in the counter "fractions".

         my_pets = ['cat', 'lizard', 'cat', 'dog', 'cat', 'snake', 'dog', 'cat', 'dog', 'parrot']

         #### YOUR CODE STARTS HERE ####

         pet_counter=Counter(my_pets)
         fractions={}
         for pet in pet_counter:
             fraction = float(pet_counter[pet]) /10
             fractions[pet] = fraction


         #### YOUR CODE ENDS HERE ####

         print("Testing: fraction of cats = %.1f" % fractions["cat"])
         print("CORRECT" if fractions["cat"] == .4 else "INCORRECT")
```

```
Testing: fraction of cats = 0.4
CORRECT
```

2.1 Applying Bayes rule: Which box did a ball come from?

In this exercise, we are interested in figuring out from which of the two boxes a ball of a certain color most likely came from. We are using Bayes rule to compute the probbaility of box A and box B given a ball of certain color and then we compare which one of these two probabilities is bigger.

Step 1: Inspect the data. How many different colors are there? How many balls of each color are in box A and in box B?

Hint: Turn the two lists `box_a` and `box_b` into Counters and print them.

```
In [14]: #### YOUR CODE STARTS HERE ####

         a_counter = Counter(box_a)
         b_counter = Counter(box_b)
         print(a_counter)
         print(b_counter)

         #### YOUR CODE ENDS HERE ####
```

```
Counter({'blue': 39, 'green': 27, 'orange': 23, 'red': 10, 'yellow': 1})
Counter({'red': 53, 'yellow': 25, 'green': 9, 'orange': 8, 'blue': 5})
```

Step 2: Compute the probability of each color in box A and each color in box B, i.e., compute $P(color \mid \text{box A})$ and $P(color \mid \text{box B})$ for each of the five colors. Store them in the counters `p_box_a` and `p_box_b`.

For example, you can compute the probability of picking a red ball from box 1 as:

$$P(red \mid \text{box A}) = \frac{\text{number of red balls in box A}}{\text{total number of balls in box A}}$$

```
In [15]: #### YOUR CODE STARTS HERE ####

         # Hint: This is very similar to Exercise 8.

         sum_a = sum(a_counter.values())
         sum_b = sum(b_counter.values())

         p_box_a = {}
         p_box_b = {}
         for color in a_counter:
             p_box_a[color] = float(a_counter[color]) / sum_a

         for color in b_counter:
             p_box_b[color] = float(b_counter[color]) / sum_b


         #### YOUR CODE ENDS HERE ####

         print(p_box_a)
         print(p_box_b)
```

```
{'orange': 0.23, 'green': 0.27, 'blue': 0.39, 'red': 0.1, 'yellow': 0.01}
{'red': 0.53, 'yellow': 0.25, 'green': 0.09, 'orange': 0.08, 'blue': 0.05}
```

Step 3: Now that we have the conditional probabilities for each color, we can apply Bayes rule to compute which box a ball of a certain color most likely came from. Fill in the blanks in this function.

As a reminder, the probability $P(box \mid color)$ is proportional to

$$P(box \mid color) \propto P(box) \times P(color \mid box)$$

```
In [16]: #### YOUR CODE STARTS HERE ####

         def likeliest_box(color):
```

```
    # The probability that someone picked a ball from box A or from box B
    # If we set both of these to 0.5, then this means that a box was chosen completely
at random
    # Modify these values to see how the likelihood of the two boxes changes.
    prior_box_a = 0.5
    prior_box_b = 0.5

    # P(box A | color) is proportional to P(color | box A) * P(box A)
    # Hint use the prior_box_a variable and the p_box_a counter from the cell above.
    ratio = p_box_a[color] * prior_box_a / (p_box_b[color] * prior_box_b)

    # Which of the two boxes is likelier? Complete the following if statement
    # such that likely_box is assigned Box A if the ball came most likely from Box A
    # and Box B if it most likely came from the other box.
    if ratio > 1:
        likely_box = "Box A"
    else:
        likely_box = "Box B"

    return likely_box

#### YOUR CODE ENDS HERE ####


print("Balls in Box A:")
print(Counter(box_a))
print("Balls in Box B:")
print(Counter(box_b))

print("")

colors = ["red", "green", "blue", "yellow", "orange"]

for color in colors:
    print("A %s ball most likely came from %s" % (color, likeliest_box(color)))
```

```
Balls in Box A:
Counter({'blue': 39, 'green': 27, 'orange': 23, 'red': 10, 'yellow': 1})
Balls in Box B:
Counter({'red': 53, 'yellow': 25, 'green': 9, 'orange': 8, 'blue': 5})

A red ball most likely came from Box B
A green ball most likely came from Box A
A blue ball most likely came from Box A
A yellow ball most likely came from Box B
A orange ball most likely came from Box A
```

2.2 Which box did a sequence of balls most likely come from?

Now we are interested in a different question. We know that someone drew several balls from a single box, but we don't know from which one of the two. In other words, given a **list** of balls, we want to determine their likeliest origin.

The conditional probabilities remain the same, so all you have to do for this exercise is rewrite the *likeliest_box* function below.

As a reminder, $P(box \mid color_1, color_2, color_3, ...)$ is proportional to

$$P(box \mid color_1, color_2, color_3, ...) \propto P(box) \times P(box \mid color_1) \times P(box \mid color_2) \times P(box \mid color_3) \times ...$$

In [17]:
```
#### YOUR CODE STARTS HERE ####
```

```
def likeliest_box(colors):
    # The probability that someone picked a ball from box A or from box B
    # If we set both of these to 0.5, then this means that a box was chosen completely
at random
    # Modify these values to see how the likelihood of the two boxes changes.
    prior_box_a = 0.5
    prior_box_b = 0.5

    # P(box A | color1, color2, color3, ...) is proportional to
    #      P(box A) * P(color1 | box A)  * P(color2 | box A) * P(color3 | box A) * ...
    # Hint use the prior_box_a variable and the p_box_a counter from above.

    p_box_a_colors = prior_box_a
    p_box_b_colors = prior_box_b

    for color in colors:
        p_box_a_colors *= p_box_a[color]
        p_box_b_colors *= p_box_b[color]

    ratio = p_box_a_colors / p_box_b_colors


    # Which of the two boxes is likelier? Complete the following if statement.
    if ratio > 1:
        likely_box = "Box A"
    else:
        likely_box = "Box B"

    return likely_box

#### YOUR CODE ENDS HERE ####


print("Balls in Box A:")
print(Counter(box_a))
print("Balls in Box B:")
print(Counter(box_b))

print("")

sequences = [["red"], ["green"], ["blue"], ["yellow"], ["orange"],
             ["red", "red", "green"],
             ["red", "green", "green"],
             ["blue", "red", "green", "yellow", "blue", "yellow", "yellow"],
             ["yellow", "orange"],
             ["yellow", "green"],
             ["yellow", "green", "green", "green", "green"]]

for seq in sequences:
    print("The sequence %s most likely came from %s" % (", ".join(seq),
likeliest_box(seq)))
```

```
Balls in Box A:
Counter({'blue': 39, 'green': 27, 'orange': 23, 'red': 10, 'yellow': 1})
Balls in Box B:
Counter({'red': 53, 'yellow': 25, 'green': 9, 'orange': 8, 'blue': 5})

The sequence red most likely came from Box B
The sequence green most likely came from Box A
The sequence blue most likely came from Box A
The sequence yellow most likely came from Box B
The sequence orange most likely came from Box A
The sequence red, red, green most likely came from Box B
The sequence red, green, green most likely came from Box A
The sequence blue, red, green, yellow, blue, yellow, yellow most likely came from Box
```

```
B
The sequence yellow, orange most likely came from Box B
The sequence yellow, green most likely came from Box B
The sequence yellow, green, green, green, green most likely came from Box A
```

2.3 Optional Challenge: More boxes!

Up until now, we always assumed that there were just 2 boxes. But all of this can be extended to more boxes as well!

Re-implement the computation of probabilities for four boxes and implement a new *likeliest_box* method that can deal with more than two boxes.

This is a very challenging and open-ended problem. Think about how you could solve this and feel free to talk this through with us before you start implementing it.

Hint: One useful function for *likeliest_box* might be the *argmax* function (https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html).

```
In [18]: from numpy import argmax

         number_of_boxes = 4

         boxes = lib.get_box_contents(n_boxes=number_of_boxes)

         for i, box in enumerate(boxes):
             print("Box %d has %d balls." % (i + 1, len(box)))


         #### YOUR CODE STARTS HERE ####

         counter_1 = Counter(boxes[0])
         counter_2 = Counter(boxes[1])
         counter_3 = Counter(boxes[2])
         counter_4 = Counter(boxes[3])

         # Estimate the conditional probabilities for each box.

         p_box_1 = {}
         p_box_2 = {}
         p_box_3 = {}
         p_box_4 = {}

         for color in counter_1:
             p_box_1[color] = float(counter_1[color]) / len(boxes[0])
         for color in counter_2:
             p_box_2[color] = float(counter_2[color]) / len(boxes[1])
         for color in counter_3:
             p_box_3[color] = float(counter_3[color]) / len(boxes[2])
         for color in counter_4:
             p_box_4[color] = float(counter_4[color]) / len(boxes[3])

         #### YOUR CODE ENDS HERE ####


         def likeliest_box(colors):
             #### YOUR CODE STARTS HERE ####

             prior_box_1 = 0.25
             prior_box_2 = 0.25
             prior_box_3 = 0.25
             prior_box_4 = 0.25

             scores = [0] * 4
             scores[0] = prior_box_1
```

```python
        scores[1] = prior_box_2
        scores[2] = prior_box_3
        scores[3] = prior_box_4

        for color in colors:
            scores[0] *= p_box_1[color]
            scores[1] *= p_box_2[color]
            scores[2] *= p_box_3[color]
            scores[3] *= p_box_4[color]

        print(scores)
        likeliest_box = argmax(scores)

        return "Box %d" % likeliest_box

        #### YOUR CODE ENDS HERE ####


    sequences = [["red"], ["green"], ["blue"], ["yellow"], ["orange"],
                 ["red", "red", "green"],
                 ["red", "green", "green"],
                 ["blue", "red", "green", "yellow", "blue", "yellow", "yellow"],
                 ["yellow", "orange"],
                 ["yellow", "green"],
                 ["yellow", "green", "green", "green", "green"]]

    for seq in sequences:
        print("The sequence %s most likely came from %s" % (", ".join(seq),
    likeliest_box(seq)))
```

```
Box 1 has 100 balls.
Box 2 has 100 balls.
Box 3 has 100 balls.
Box 4 has 100 balls.
[0.025, 0.1325, 0.0375, 0.0125]
The sequence red most likely came from Box 1
[0.0675, 0.0225, 0.0075, 0.0125]
The sequence green most likely came from Box 0
[0.0975, 0.0125, 0.0375, 0.0125]
The sequence blue most likely came from Box 0
[0.0025, 0.0625, 0.16, 0.0125]
The sequence yellow most likely came from Box 2
[0.0575, 0.02, 0.0075, 0.2]
The sequence orange most likely came from Box 3
[0.0006750000000000001, 0.00632025, 0.00016874999999999998, 3.1250000000000001e-05]
The sequence red, red, green most likely came from Box 1
[0.0018225000000000003, 0.00107325, 3.3749999999999994e-05, 3.1250000000000001e-05]
The sequence red, green, green most likely came from Box 0
[1.0266750000000003e-09, 4.658203125e-07, 6.635519999999999e-06,
1.9531250000000001e-10]
The sequence blue, red, green, yellow, blue, yellow, yellow most likely came from Box
2
[0.000575, 0.005, 0.0048, 0.010000000000000002]
The sequence yellow, orange most likely came from Box 3
[0.000675, 0.005625, 0.0048, 0.0006250000000000001]
The sequence yellow, green most likely came from Box 1
[1.3286025000000003e-05, 4.100624999999999e-06, 1.2959999999999997e-07,
7.812500000000003e-08]
The sequence yellow, green, green, green, green most likely came from Box 0
```

3.4 Lesson 4: Language Model

lesson4_languagemodel

April 29, 2018

1 Language Model Demo

Based on this demo: http://nlpforhackers.io/language-models/

1.0.1 Import modules and data

```
In [1]: import random
        from nltk import bigrams, trigrams
        from nltk.corpus import reuters, movie_reviews, shakespeare
        from nltk.tokenize import sent_tokenize, word_tokenize
        from collections import Counter, defaultdict
```

```
In [2]: # Choose a corpus: reuters, movie_reviews or shakespeare
        import nltk
        nltk.download('movie_reviews')
        corpus = movie_reviews

        if corpus == shakespeare:
            shakespeare_text = ''.join([''.join(corpus.xml(fileid).itertext()) for fileid in
        corpus.fileids()])
            words = word_tokenize(shakespeare_text)
            sents = [word_tokenize(sent) for sent in sent_tokenize(shakespeare_text)]
        else:
            words = corpus.words()
            sents = corpus.sents()

        # Lowercase everything
        words = [w.lower() for w in words]
        sents = [[w.lower() for w in sent] for sent in sents]
```

```
[nltk_data] Downloading package movie_reviews to
[nltk_data]     /Users/xiaoxing/nltk_data...
[nltk_data]   Package movie_reviews is already up-to-date!
```

1.0.2 Unigram language model

In this section, we will construct a language model based on unigrams (words).

```
In [3]: # Exercise 1. Fill in the blanks.


        # Step 1: Make a Counter from the list of words and call it "unigram_counts" (remember,
        this is easy to do!)
        unigram_counts = Counter(words)

        # Step 2: Get the total number of words and assign it to "total_count"
        total_count = sum(unigram_counts.values())
```

```
print("Total number of words in corpus: ", total_count)

# Print 10 most common words
print("\nTop 10 most common words: ")

for (word, count) in unigram_counts.most_common(n=10):
    print(word, count)
```

```
Total number of words in corpus:  1583820

Top 10 most common words:
, 77717
the 76529
. 65876
a 38106
and 35576
of 34123
to 31937
' 30585
is 25195
in 21822
```

In [4]:

```
# Exercise 2. Fill in the blanks.

# We have the Counter unigram_counts, which maps each word to its count.
# We want to construct the Counter unigram_probs, which maps each word to its
probability.


# Step 1: create an empty Counter called unigram_probs.
unigram_probs = {}

# Step 2: using a for-loop over unigram_counts, (this will iterate over the keys i.e.
words)
# calculate the appropriate fraction, and add the word -> fraction pair to
unigram_probs.
# Remember about integer division!
for word in unigram_counts:
    fraction = float(unigram_counts[word])/10
    unigram_probs[word] = fraction

# Check the probabilities add up to 1
print("Probabilities sum to: ", sum(unigram_probs.values()))

# Print 10 most common words
print("\nTop 10 most common words: ")
for (word, count) in Counter(unigram_probs).most_common(n=10):
    print(word, "%.5f" % count)
```

```
Probabilities sum to:  158382.0000001032

Top 10 most common words:
, 7771.70000
the 7652.90000
. 6587.60000
a 3810.60000
and 3557.60000
of 3412.30000
to 3193.70000
' 3058.50000
is 2519.50000
```

```
in 2182.20000
```

```
In [5]: # Print the probability of word "the", then try some other words.
        print(unigram_probs['the'])
```

```
7652.9
```

```
In [6]: # Generate 100 words of language using the unigram model.
        # Run this cell several times!

        text = []  # will be a list of generated words

        for _ in range(100):
            r = random.random()  # random number in [0,1]

            # Find the word whose "interval" contains r
            accumulator = .0
            for word, freq in unigram_probs.items():
                accumulator += freq
                if accumulator >= r:
                    text.append(word)
                    break

        print(' '.join(text))
```

```
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
```

1.0.3 Bigram language model

In this section, we'll build a language model based on bigrams (pairs of words).

```
In [7]: # Count how often each bigram occurs.

        # bigram_counts is a dictionary that maps w1 to a dictionary mapping w2 to the count for
        (w1, w2)
        bigram_counts = defaultdict(lambda: Counter())

        for sentence in sents:
            for w1, w2 in bigrams(sentence, pad_right=True, pad_left=True):
                bigram_counts[w1][w2] += 1
```

```
In [8]: # Print how often the bigram "of the" occurs. Try some other words following "of".
        print(bigram_counts['of']['the'])
```

```
8621
```

```
In [9]: # Transform the bigram counts to bigram probabilities
        bigram_probs = defaultdict(lambda: Counter())
        for w1 in bigram_counts:
            total_count = float(sum(bigram_counts[w1].values()))
            bigram_probs[w1] = Counter({w2: c / total_count for w2, c in
        bigram_counts[w1].items()})
```

```
In [10]: # Print the probability that 'the' follows 'of'
         print(bigram_probs['of']['the'])
```

```
0.25264484365384055
```

```
In [11]: # Print the top ten tokens most likely to follow 'fair', along with their probabilities.
         # Try some other words.
         prob_dist = bigram_probs['fair']
         for word, prob in prob_dist.most_common(10):
             print(word, "%.5f" % prob)
```

```
, 0.19048
to 0.15238
game 0.10476
. 0.04762
share 0.04762
amount 0.03810
enough 0.03810
bit 0.02857
warning 0.01905
town 0.00952
```

```
In [12]: # Generate text with bigram model.
         # Run this cell several times!

         text = [None]  # You can put your own starting word in here
         sentence_finished = False

         # Generate words until a None is generated
         while not sentence_finished:
             r = random.random()  # random number in [0,1]
             accumulator = .0
             latest_token = text[-1]
             prob_dist = bigram_probs[latest_token]  # prob dist of what token comes next

             # Find the word whose "interval" contains the random number r.
             for word, p in prob_dist.items():
                 accumulator += p
                 if accumulator >= r:
                     text.append(word)
                     break

             if text[-1] == None:
                 sentence_finished = True

         print(' '.join([t for t in text if t]))
```

```
s no genius , for me to drop down a half hour running away myself included as the cast
is being interesting about henstridge )
```

How does the bigram text compare to the unigram text?

1.0.4 Trigram language model

In this section, we'll build a language model based on trigrams (triples of words).

```
In [13]: # Count how often each trigram occurs.

         # trigram_counts maps (w1, w2) to a dictionary mapping w3 to the count for (w1, w2, w3)
         trigram_counts = defaultdict(lambda: Counter())

         for sentence in sents:
             for w1, w2, w3 in trigrams(sentence, pad_right=True, pad_left=True):
                 trigram_counts[(w1, w2)][w3] += 1
```

```
In [14]: # Print how often the trigram "I am not" occurs. Try some other trigrams.
         print(trigram_counts[('i', 'am')]['not'])
```

27

```
In [15]: # Transform the trigram counts to trigram probabilities
         trigram_probs = defaultdict(lambda: Counter())
         for w1_w2 in trigram_counts:
             total_count = float(sum(trigram_counts[w1_w2].values()))
             trigram_probs[w1_w2] = Counter({w3: c / total_count for w3, c in
         trigram_counts[w1_w2].items()})
```

```
In [16]: # Print the probability that 'not' follows 'i am'. Try some other combinations.
         print(trigram_probs[('i', 'am')]['not'])
```

0.16363636363636364

```
In [17]: # Print the top ten tokens most likely to follow 'i am', along with their probabilities.
         # Try some other pairs of words.
         prob_dist = trigram_probs[('i', 'am')]
         for word, prob in prob_dist.most_common(10):
             print(word, "%.5f" % prob)
```

```
not 0.16364
a 0.07273
sure 0.07273
the 0.03030
willing 0.02424
going 0.02424
, 0.02424
of 0.01818
glad 0.01818
thinking 0.01212
```

```
In [18]: # Generate text with trigram model.
         # Run this cell several times!

         text = [None, None]  # You can put your own first two words in here

         sentence_finished = False

         # Generate words until two consecutive Nones are generated
         while not sentence_finished:
             r = random.random()
             accumulator = .0
             latest_bigram = tuple(text[-2:])
             prob_dist = trigram_probs[latest_bigram]  # prob dist of what token comes next

             for word, p in prob_dist.items():
                 accumulator += p
                 if accumulator >= r:
                     text.append(word)
                     break

             if text[-2:] == [None, None]:
                 sentence_finished = True

         print(' '.join([t for t in text if t]))
```

```
instead of refreshing the audience with information dug up by the fact that margaret
does not borrow from a mile away .
```

How does the trigram text compare to the bigram text?

1.1 Extension exercise

N-gram language models can encounter the *sparsity problem*, especially if the data is small.

Suppose you train a trigram language model on a small amount of data (let's say the text of *The Hunger Games*), then use the language model to generate text.

On each step, you take the last two generated words (e.g. "may the") and lookup the probability distribution of what word is most likely to come next. But if your training data is small, perhaps there is only one example of the bigram "may the" in the training data (e.g. "may the odds be ever in your favor" in *The Hunger Games*). In that case, the next word will be *odds* with probability 1. This means that your language model always says "odds" after saying "may the".

1. Is the sparsity problem worse for unigram language models, bigram language models, trigram language models, or n-gram language models for n>3?
2. How might you fix this problem?
3. How might you fix this problem without access to more training data?

Try altering either the bigram or the trigram language model with your solution to question 3.

3.5 Lesson 5: Naive Bayes

lesson5_naivebayes

April 29, 2018

```
In [1]: # Run this every time you open the spreadsheet
        %load_ext autoreload
        %autoreload 2
        from collections import Counter
        import lib
```

1 Load and inspect the data

```
In [2]: # Load the data.
        # This function returns tweets and test_tweets, both lists of tweets
        tweets, test_tweets = lib.read_data()
```

2 Learn a Naive Bayes classifier

To construct our Naive Bayes classifier, we first need to calculate two things:

2.0.1 Prior probabilities of categories

We need to calculate $P(C_i)$ for each category $C_i \in \{\text{Energy}, \text{Food}, \text{Medical}, \text{Water}, \text{None}\}$.

We estimate $P(C_i)$ by $\frac{\text{\# tweets about } C_i}{\text{\# tweets}}$

2.0.2 Conditional probabilities of tokens

For each token (i.e. word) x_j and each category C_i, we need to calculate $P(x_j|C_i)$.

We estimate $P(x_j|C_i) = \frac{P(x_j \text{ and } C_i)}{P(C_i)}$ by $\frac{\text{\# tweets about } C_i \text{ containing } x_j}{\text{\# tweets about } C_i}$

```
In [3]: # Exercise 1, step-by-step version (challenge version is below).

        # The function below has two arguments: a list of tweets, and a category c
        # which is a string equal to one of "Energy", "Food", "Medical", "Water", "None".
        # The function should calculate the two things described above.
        # Fill in the blanks.


        def calc_probs(tweets, c):
            """
            Input:
                tweets: a list of tweets
                c: a string representing a category; one of "Energy", "Food", "Medical",
        "Water", "None".
            Returns:
                prob_c: the prior probability of category c
                token_probs: a Counter mapping each token to P(token|category c)
            """
```

```
    # Step 1: Calculate the total number of tweets
    num_tweets = len(tweets)

    # Step 2: Calculate the number of tweets that are about category c.
    # Save the answer to a variable called num_tweets_about_c.
    # Remember c is a string, and you can get the category of a tweet via tweet.category
    num_tweets_about_c = sum(map(lambda tweet: tweet.category == c, tweets))


    # Step 3: Calculate the probability of category c using the answers from Steps 1 and
2.
    # Hint: be careful when you divide two integers!
    prob_c = float(num_tweets_about_c)/num_tweets

    # Step 4: Create an empty Counter called token_counts.
    # (We will use it to map each token to the number of category-c tweets containing
that token.)
    token_counts = Counter()

    # Step 5 (tricky): Use a for-loop to iterate over the list of tweets.
    # Use an if-statement to check whether the tweet is in category c.
    # If it is, iterate over the tokens of the tweet (which you can access via
tweet.tokenSet) using a for-loop.
    # For each token, increment its count in token_counts.

    for tweeter in tweets:
        if tweeter.category == c:
            for token in tweeter.tokenSet:
                token_counts[token] += 1

    # Step 6: Create an empty Counter called token_probs.
    # (We will use it to map each token to P(token | category c),
    # i.e. the fraction of all category-c tweets that contain the token)
    token_probs = Counter()

    # Step 7: Now fill token_probs.
    # For each token->count in token_counts, you want to add token->fraction to
token_probs.
    # Use a for-loop over token_counts.
    # Remember that when you iterate over a dictionary/Counter, you access the keys.
    # You'll need to use the variable num_tweets_about_c.
    # Be careful when you divide integers!

    for token in token_counts:
        token_probs[token] = token_counts[token] / num_tweets_about_c

    print("Class %s has prior probability %.2f" % (c, prob_c))
    return prob_c, token_probs


prob_food, token_probs_food = calc_probs(tweets, "Food")
prob_water, token_probs_water = calc_probs(tweets, "Water")
prob_energy, token_probs_energy = calc_probs(tweets, "Energy")
prob_medical, token_probs_medical = calc_probs(tweets, "Medical")
prob_none, token_probs_none = calc_probs(tweets, "None")
```

```
Class Food has prior probability 0.47
Class Water has prior probability 0.09
Class Energy has prior probability 0.12
Class Medical has prior probability 0.04
Class None has prior probability 0.28
```

2.0.3 See what your model has learnt

In [4]:
```
# For each category c, print out the tokens that maximize P(c|token)

token_probs = {'Food': token_probs_food, 'Water': token_probs_water, 'Energy':
token_probs_energy,
              'Medical': token_probs_medical, 'None': token_probs_none}
prior_probs = {'Food': prob_food, 'Water': prob_water, 'Energy': prob_energy, 'Medical':
prob_medical,
              'None': prob_none}
lib.most_discriminative(tweets, token_probs, prior_probs)
```

```
MOST DISCRIMINATIVE TOKENS:

TOKEN                    P(Energy|token)
powers                   0.8029
dark                     0.8029
generator                0.7654
batteries                0.7559
class                    0.7534
sandysucks               0.7534
flashlights              0.7345
masks                    0.7334
11/3                     0.6736
cleaner                  0.6707

TOKEN                    P(Food|token)
canned                   0.9784
non-perishable           0.9767
serve                    0.9663
perishable               0.9562
cook                     0.9511
soup                     0.9489
sandwiches               0.9489
thanksgiving             0.9441
rice                     0.9441
meal                     0.9383

TOKEN                    P(Medical|token)
meds                     0.8229
aid                      0.8008
ointment                 0.7360
prescription             0.7360
ups                      0.7360
medicine                 0.7360
medications              0.7360
4t-5t                    0.7360
kits                     0.6596
pull                     0.6596

TOKEN                    P(None|token)
..                       0.9531
everyone                 0.8955
last                     0.8809
feel                     0.8809
im                       0.8618
irene                    0.8604
...                      0.8601
thing                    0.8314
wow                      0.8314
tropical                 0.8314
```

```
TOKEN               P(Water|token)
bottled             0.9059
gallon              0.8307
jugs                0.7970
water               0.7873
gallons             0.7266
pallets             0.6625
spring              0.6625
flood               0.6625
liter               0.6625
parks               0.6625
```

3 Build a Naive Bayes classifier

Now we've calculated $P(C_i)$ and $P(x_j|C_i)$, we can classify any tweet!

Given a tweet which is a set of tokens $\{x_1, ..., x_n\}$, the posterior probability of each category C_i is

$P(C_i|x_1, ..., x_n) \propto P(C_i) \times P(x_1|C_i) \times P(x_2|C_i) ... \times P(x_n|C_i)$

We just need to calculate this for each category then determine which is largest.

In [5]:
```
# Exercise 2.

# Complete this function that calculates the posterior probability of P(c|tweet).

def get_posterior_prob(tweet, prob_c, token_probs):
    """Calculate the posterior P(c|tweet).
    (Actually, calculate something proportional to it).

    Inputs:
        tweet: a tweet
        prob_c: the prior probability of category c
        token_probs: a Counter mapping each token P(token|c)
    Return:
        The posterior P(c|tweet).
    """

    ##### YOUR CODE STARTS HERE #####

    # Hint: first set posterior to prob_c, then use a for-loop over tweet.tokenSet
    # to repeatedly multiply posterior by P(token|c)

    posterior = prob_c
    for token in tweet.tokenSet:
        if token_probs[token] == 0:
            posterior *= 0.001
        else:
            posterior *= token_probs[token]

    ##### YOUR CODE ENDS HERE #####

    return posterior


# Now you've written the function, look at the output for P(Energy|"No power in
Riverdale").
# What's gone wrong?
# Try editing your function above to print out each token and token_probs[token].
# Can you see what went wrong? How might you fix it?
```

```
riverdale_tweet = lib.Tweet("No power in Riverdale", "Energy", "need")
print("P(Energy|'No power in Riverdale') = ", get_posterior_prob(riverdale_tweet,
prob_energy, token_probs_energy))
```

```
P(Energy|'No power in Riverdale') =  2.806001890359169e-06
```

In [6]:
```
# This cell defines the classification function, that takes a tweet
# and decides which category is most likely using the posteriors you just calculated.


# OPTIONAL EXERCISE (come back to it once you've reached the end of the notebook).
# Rewrite this function to be less repetitive i.e. don't repeat things 5 times.
# There are several possible solutions; you might want to use lists or dictionaries.
# You might also want to rewrite the earlier code that computed prob_food,
token_probs_food etc.


def classify_nb(tweet):
    """Classifies a tweet. Calculates the posterior P(c|tweet) for each category c,
    and returns the category with largest posterior.
    Input:
        tweet
    Output:
        string equal to most-likely category for this tweet
    """
    posterior_food_prob = get_posterior_prob(tweet, prob_food, token_probs_food)
    posterior_water_prob = get_posterior_prob(tweet, prob_water, token_probs_water)
    posterior_energy_prob = get_posterior_prob(tweet, prob_energy, token_probs_energy)
    posterior_medical_prob = get_posterior_prob(tweet, prob_medical,
token_probs_medical)
    posterior_none_prob = get_posterior_prob(tweet, prob_none, token_probs_none)

    max_posterior = max([posterior_food_prob, posterior_water_prob,
                         posterior_energy_prob, posterior_medical_prob,
                         posterior_none_prob])
    if posterior_food_prob == max_posterior:
        return 'Food'
    elif posterior_water_prob == max_posterior:
        return 'Water'
    elif posterior_energy_prob == max_posterior:
        return 'Energy'
    elif posterior_medical_prob == max_posterior:
        return 'Medical'
    else:
        return 'None'
```

3.1 Evaluate the Naive Bayes classifier

In [7]:
```
# Compare true labels and predicted labels in a table

predictions = [(tweet, classify_nb(tweet)) for tweet in test_tweets]  # a list of
(tweet, prediction) pairs
lib.show_predictions(predictions)
```

```
<IPython.core.display.HTML object>
```

In [8]:
```
# Get average F1 score for the test set

predictions = [(tweet, classify_nb(tweet)) for tweet in test_tweets]  # maps each test
tweet to its predicted label
lib.evaluate(predictions)
```

```
Energy
Precision:  50.0
Recall:  60.0
F1:  54.54545454545455

Food
Precision:  83.56164383561644
Recall:  94.57364341085271
F1:  88.72727272727272

Medical
Precision:  85.71428571428571
Recall:  46.15384615384615
F1:  60.0

None
Precision:  82.85714285714286
Recall:  73.41772151898734
F1:  77.85234899328859

Water
Precision:  80.0
Recall:  40.0
F1:  53.333333333333336

Average F1:  66.89168191986984
```

```
In [9]: # Get average F1 score for the TRAINING set.
        # Compare with average F1 for test set above. What's the reason for the difference?

        trainset_predictions = [(tweet, classify_nb(tweet))
                                for tweet in tweets]  # maps each training tweet to its
        predicted label
        lib.evaluate(trainset_predictions)
```

```
Energy
Precision:  91.33333333333333
Recall:  99.27536231884058
F1:  95.13888888888887

Food
Precision:  96.6355140186916
Recall:  97.91666666666667
F1:  97.27187206020695

Medical
Precision:  97.77777777777777
Recall:  100.0
F1:  98.87640449438202

None
Precision:  97.98657718120805
Recall:  94.49838187702265
F1:  96.21087314662273

Water
Precision:  100.0
Recall:  91.08910891089108
F1:  95.33678756476684
```

```
Average F1:  96.56696523097348
```

```
In [10]: lib.show_confusion_matrix(predictions)
```

```
<IPython.core.display.HTML object>
```

3.6 Lesson 6: Visualization

lesson6_visualization

April 29, 2018

```
In [1]: # Run this every time you open the spreadsheet
        %load_ext autoreload
        %autoreload 2
        from collections import Counter
        import lib
        import random
```

0.1 Optional Exercise: Add bigram capabilities to the classifier!

So far our Naive Bayes classifier scores an Average F1 score of 66.9% on the test set. Let's see if we can improve on that by incorporating bigrams!

```
In [2]: def add_bigrams(tweet):
        # Currently, tweet has an attribute called tweet.tokenList which is a list of tokens.
        # You want to add a new attribute to tweet called tweet.bigramList which is a list of
        bigrams.
        # Each bigram should be a pair of strings. You can define the bigram like this: bigram =
        (token1, token2).
        # In Python, this is called a tuple. You can read more about tuples here:
        https://www.programiz.com/python-programming/tuple

        ##### YOUR CODE STARTS HERE #####

            tweet.bigramList = [(tweet.tokenList[i], tweet.tokenList[i + 1]) for i in
        range(len(tweet.tokenList) - 1)]

        ##### YOUR CODE ENDS HERE #####


        tweets, test_tweets = lib.read_data()
        for tweet in tweets + test_tweets:
            add_bigrams(tweet)
        print("Checking if bigrams are correct...")
        for tweet in tweets + test_tweets:
            assert tweet._bigramList == tweet.bigramList, "Error in your implementation of the
        bigram list!"
        print("Bigrams are correct.\n")

        prior_probs, token_probs = lib.learn_nb(tweets)
        predictions = [(tweet, lib.classify_nb(tweet, prior_probs, token_probs)) for tweet in
        test_tweets]
        lib.evaluate(predictions)
```

```
Checking if bigrams are correct...
Bigrams are correct.

Energy
Precision:  60.0
Recall:  67.5
F1:  63.529411764705884
```

```
Food
Precision:  84.39716312056737
Recall:  92.24806201550388
F1:  88.14814814814815

Medical
Precision:  75.0
Recall:  46.15384615384615
F1:  57.14285714285714

None
Precision:  82.66666666666667
Recall:  78.48101265822785
F1:  80.51948051948052

Water
Precision:  83.33333333333333
Recall:  50.0
F1:  62.5

Average F1:  70.36797951503834
```

0.2 Re-run the classifier and get evaluation score

This notebook uses our implementation of the Naive Bayes classifier, but it's very similar to what you implemented yesterday. If you're interested in the details, take a look at the `learn_nb` and `classify_nb` functions in `lib.py` in the `sailors2017` directory.

```
In [3]: tweets, test_tweets = lib.read_data()
        prior_probs, token_probs = lib.learn_nb(tweets)
        predictions = [(tweet, lib.classify_nb(tweet, prior_probs, token_probs)) for tweet in
        test_tweets]
        lib.evaluate(predictions)
```

```
Energy
Precision:  60.0
Recall:  67.5
F1:  63.529411764705884

Food
Precision:  84.39716312056737
Recall:  92.24806201550388
F1:  88.14814814814815

Medical
Precision:  75.0
Recall:  46.15384615384615
F1:  57.14285714285714

None
Precision:  82.66666666666667
Recall:  78.48101265822785
F1:  80.51948051948052

Water
Precision:  83.33333333333333
```

```
Recall:  50.0
F1:  62.5

Average F1:  70.36797951503834
```

0.3 Inspecting the Classifier

After implementing and training a classifier, you often want to inspect what kind of things it has learned and how it is making predictions on individual examples. This can help you make sure that you implemented everything correctly and it might give you ideas on how to further improve the classifier.

0.3.1 Most discriminative words

Let's first look again at the most discriminative words for each category, i.e. the words that maximize P(category | word), for each category.

```
In [4]: lib.most_discriminative(tweets, token_probs, prior_probs)
```

```
MOST DISCRIMINATIVE TOKENS:

TOKEN                P(Energy|token)
dark                 0.8029
powers               0.8029
generator            0.7654
batteries            0.7559
class                0.7534
sandysucks           0.7534
flashlights          0.7345
masks                0.7334
11/3                 0.6736
cleaner              0.6707

TOKEN                P(Food|token)
canned               0.9784
non-perishable       0.9767
serve                0.9663
perishable           0.9562
cook                 0.9511
soup                 0.9489
sandwiches           0.9489
rice                 0.9441
thanksgiving         0.9441
meal                 0.9383

TOKEN                P(Medical|token)
meds                 0.8229
aid                  0.8008
ups                  0.7360
medications          0.7360
prescription         0.7360
4t-5t                0.7360
ointment             0.7360
medicine             0.7360
kits                 0.6596
pull                 0.6596
```

```
TOKEN               P(None|token)
..                  0.9531
everyone            0.8955
last                0.8809
feel                0.8809
im                  0.8618
irene               0.8604
...                 0.8601
tropical            0.8314
halloween           0.8314
finally             0.8314

TOKEN               P(Water|token)
bottled             0.9059
gallon              0.8307
jugs                0.7970
water               0.7873
gallons             0.7266
flood               0.6625
pallets             0.6625
spring              0.6625
feet                0.6625
parks               0.6625
```

These five lists show you which words are most predictive of the five categories. For example, the word *bottled* is a very strong indicator that the tweet is about water or the word *canned* is a very strong indicator that the tweet is about food.

Many of you used several of these words in your rule-based classifiers in week 1. It's reassuring (and exciting!) to see that the Naive Bayes classifier learned that these words are good indicators of the categories as well.

0.3.2 Confusion matrix

Another useful type of visualization is a so-called confusion matrix. A confusion matrix shows you for each true category *c* how many of the tweets in *c* were classified into the five different categories. (In this way it tells you which categories are "confused" for others by the classifier).

```
In [5]: lib.show_confusion_matrix(predictions)
```

```
<IPython.core.display.HTML object>
```

In the matrix, the **rows** correspond to the **true category** and the **columns** correspond to the **predicted category**.

For example, this matrix shows you that of all the 79 tweets in the category *None*, 13 were incorrectly classified as *Energy*, 3 as *Food*, and 1 as *Medical*. 62 of them were actually correctly classified as *None*.

0.3.3 Visualizing individual tweets

It can also be really useful to visualize the probabilities of each token in an individual tweet. This can help you understand why a classifier made a correct or wrong prediction. We've implemented a visualization for you so that you can use to inspect how the classifier works on individual tweets.

```
In [6]: # The following code visualizes a random tweet from the test data.
        # Hover your mouse over the words!

        random_tweet = random.choice(test_tweets)
        lib.visualize_tweet(random_tweet, prior_probs, token_probs)
```

```
<IPython.core.display.HTML object>
```

The color of each word tells you for which category $P(\text{token} \mid \text{category})$ is the highest. When you move the mouse over a word, it shows you the actual values of $P(\text{token} \mid \text{category})$ for each category that the classifier uses to make its predictions.

You can also have the classifier make a prediction on your own tweets. Change the text in `my_tweet` below and run the cell below to see what the classifier would predict.

```
In [7]: my_tweet = "I urgently need some bottled water."

        lib.visualize_tweet(lib.Tweet(my_tweet, "?", ""), prior_probs, token_probs)
```

```
<IPython.core.display.HTML object>
```

0.4 Error analysis: Figuring out remaining errors

Often, one wants to know in which scenarios a classifier makes mistakes. This can be really useful when you want to improve your classifier.

In this exercise, try to break the Naive Bayes classifier. Use the cell above and try to come up with a tweet which should be classified as *Food* but which is assigned a different category. Once you find such a tweet, use the visualization to figure out why the classifier gets this example wrong.

Repeat this exercise for all the other categories. Based on your observations, do you have any ideas on how to further improve the classifier?

```
In [10]: my_tweet = "Milks are the batteries powering the life."  # Food but assigned to energy.

         lib.visualize_tweet(lib.Tweet(my_tweet, "?", ""), prior_probs, token_probs)
```

```
<IPython.core.display.HTML object>
```

Bibliography

[Let15] Letian: *Three common models of Naive Bayes: Gaussian, Multinomial, Bernoulli.* http://www.letiantian.me/2014-10-12-three-models-of-naive-nayes/. Version: Mar 2015

[lr-] *1.1.11. Logistic regression.* http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

[PVG+11] Pedregosa, F. ; Varoquaux, G. ; Gramfort, A. ; Michel, V. ; Thirion, B. ; Grisel, O. ; Blondel, M. ; Prettenhofer, P. ; Weiss, R. ; Dubourg, V. ; Vanderplas, J. ; Passos, A. ; Cournapeau, D. ; Brucher, M. ; Perrot, M. ; Duchesnay, E.: Scikit-learn: Machine Learning in Python. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830

[twi18] *Twitter Terms of Service.* https://twitter.com/en/tos. Version: Apr 2018

[Wik18] Wikipedia contributors: *Naive Bayes classifier — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Naive_Bayes_classifier&oldid=838385521. Version: 2018. – [Online; accessed 29-April-2018]