

▼ Machine-Learning Based Natural Language Processing

Lab 1 - Text Processing and Sentiment Analysis

Outline

In this session, we will teach you how to approach processing a text-based dataset, with the goal of training a classifier, based on classical machine learning techniques. We've chosen **sentiment analysis** as an application.

We will perform some preprocessing on data, see how to transform textual data into feature vectors, how to train several algorithms, as well as discuss evaluating the trained models. We will make use of the sklearn machine learning library in python.

These are the topics we will cover:

- **Part 1 - Python for data (pre-)processing**
 - Basic data analytics
 - Text processing
- **Part 2 - Sentiment Analysis**
 - Data Exploration
 - Text processing
 - Feature Extraction - **Exercise 1**
 - Classification Models - **Exercise 2**
 - Evaluation
 - Regularization - **Exercise 3**
 - Kaggle competition (*)

(*) As a **bonus**, we'll have a friendly kaggle competition to see who can get the highest performance for sentiment prediction on a held out test

▼ Part 1 - Python for data (pre)processing

Python is the most widely used programming language among data scientists and machine learning engineers, next to R, Matlab and SQL (although the latter is not a programming language in the strict sense). It is very readable and development times are usually quite fast, which makes it the perfect language to do rapid prototyping and to toy around with your ideas. It has also become an entire ecosystem on its own among data scientist: a plethora of GitHub data science repositories exist, which you can use in your own projects; if you have some data science problem, chances are people have already worked on it in the past and have open-sourced their code. And finally, Python is the language of choice if you want to experiment with deep learning -- the latest 'big leap' in machine learning -- since the most mature and most widely used frameworks PyTorch, Tensorflow and Keras provide very easy-to-use Python interfaces.

In this lab session, we will explore Python for (advanced) data processing and data analytics. We will focus on core Python modules and programming paradigms. The motivation is that, once you have good knowledge of the core Python modules, external modules such as NumPy, SciPy, Pandas, Seaborn... become much easier to learn and work with.

Today, we will use the *Hillary Clinton and Donald Trump 2016 US Presidential Election Twitter dataset*, an open dataset provided by Kaggle, containing ca. 6500 tweets. We will cover the following topics:

- Read the dataset in memory
- Calculate basic analytics on the dataset
- Make visual plots of dataset metrics
- Perform text cleaning on the tweets
- Identify distinct words in the tweets
- Create a basic AI that can auto-generate a tweet

Remark: this lab session is self-paced, and we hugely encourage that you go out on the internet and become big friends with *Google* and *StackOverflow*. No one has complete knowledge of the entire Python core, let alone knowing the most efficient methods to achieve a certain result. If you have a basic Python problem (e.g., "how would I initialize a list of all zeros with a given length?") you will definitely and easily find multiple answers on the internet.

▼ 1. Basic data analytics

We have provided the dataset in csv format. Pandas will be used to read the dataset into memory. Pandas is a convenient module to visualize tabular datasets, and we will use it now in the beginning of this lab session to get a quick look at the data. Read the csv file using Pandas and visualize it in this notebook. You can also take a look at the official dataset website for more details on the dataset:

<https://www.kaggle.com/benhamner/clinton-trump-tweets>

```
!wget "https://www.dropbox.com/s/octyjwtn4gww7gm/data.zip?dl=1" -O data.zip
!unzip -o data.zip
!rm data.zip
```

```
--2020-02-25 11:57:29-- https://www.dropbox.com/s/octyjwtn4gww7gm/data.zip?dl=1
Resolving www.dropbox.com (www.dropbox.com)... 162.125.1.1, 2620:100:6016:1::a27d:101
Connecting to www.dropbox.com (www.dropbox.com)|162.125.1.1|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: /s/dl/octyjwtn4gww7gm/data.zip [following]
--2020-02-25 11:57:29-- https://www.dropbox.com/s/dl/octyjwtn4gww7gm/data.zip
Reusing existing connection to www.dropbox.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://ucfb66988ee433d3cf91fa47532a.dl.dropboxusercontent.com/cd/0/get/Ayyd8M3XwbmDBwYIRp0AgHXwlffa9PcdmzXdIck0x8aHc;
--2020-02-25 11:57:29-- https://ucfb66988ee433d3cf91fa47532a.dl.dropboxusercontent.com/cd/0/get/Ayyd8M3XwbmDBwYIRp0AgHXwlffa9Pc
Resolving ucfb66988ee433d3cf91fa47532a.dl.dropboxusercontent.com (ucfb66988ee433d3cf91fa47532a.dl.dropboxusercontent.com)... 162.125.1.1
Connecting to ucfb66988ee433d3cf91fa47532a.dl.dropboxusercontent.com (ucfb66988ee433d3cf91fa47532a.dl.dropboxusercontent.com)|162.125.1.1|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 977015 (954K) [application/binary]
Saving to: 'data.zip'
```

```
data.zip          100%[=====>] 954.12K  --.-KB/s    in 0.04s
```

```
2020-02-25 11:57:29 (21.4 MB/s) - 'data.zip' saved [977015/977015]
```

```
Archive: data.zip
  inflating: test.csv
  inflating: train.csv
  inflating: tweets_stripped.csv
```

```
import pandas

# read the data
dataset = pandas.read_csv('tweets_stripped.csv', sep=',', header=0)

# show the 100th datapoint
dataset.iloc[100]
```

```

id 780592630585499648
handle realDonaldTrump
text Hillary's policies have made America less safe...
is_retweet True
original_author TeamTrump
time 2016-09-27T02:19:20
in_reply_to_screen_name NaN
in_reply_to_status_id NaN
in_reply_to_user_id NaN
is_quote_status False
lang en
retweet_count 4741
favorite_count 10739
longitude NaN
latitude NaN
place_id NaN
place_full_name NaN
place_name NaN
place_type NaN
place_country_code NaN
place_country NaN
place_contained_within NaN
place_attributes NaN
place_bounding_box NaN
source_url http://twitter.com
truncated False
Name: 100, dtype: object
```

We will now use Python's own built-in `csv` module to read and process the csv file. To store the tweets in memory, we will write our own Tweet class in which we can store the data in a structured fashion. Study the code of the Tweet class below; we have already written the constructor

`__init__` method that stores all necessary data as attributes in the object.

Complete the `__str__()` function that can be used to "pretty print" a Tweet object (e.g., print its ID, handle and text in a nice format).

```
class Tweet:
    def __init__(self, tweet_id, handle, text, is_retweet, original_author, time, lang, retweet_count, favorite_count):
        self.tweet_id = tweet_id
        self.handle = handle
        self.text = text
        self.is_retweet = is_retweet == 'True'
        self.original_author = original_author
        self.time = time
        self.lang = lang
        self.retweet_count = int(retweet_count)
        self.favorite_count = int(favorite_count)

    def __str__(self):
        return self.tweet_id + ' / ' + self.handle + ": " + self.text
```

Now use the `csv` module to read the csv dataset row by row. For each row, create a Tweet object and store it in a list. This list will be our dataset. Use the official Python docs as inspiration: <https://docs.python.org/3.6/library/csv.html>.

```
import csv

dataset = []

with open('tweets_stripped.csv', 'r') as data_file:
    datareader = csv.reader(data_file, delimiter=',', quotechar='"')
    for i, row in enumerate(datareader):
        if i > 0:
            t = Tweet(row[0], row[1], row[2], row[3], row[4], row[5], row[10], row[11], row[12])
            dataset.append(t)
```

▼ 1.1 Warm-up questions

Again, as a reminder, Google and StackOverflow are your friends!

Question: How many tweets are there in the dataset?

```
## Answer Q1
print(len(dataset))
```


 6444

Question: How many of these tweets are retweets? Give an absolute number and percentage. Also: what percentage of tweets are from Hillary and from Trump?

For this question you can use for-loops, but Python has a very compact and powerful syntax called a *list comprehension*. Toy around with the examples below, and then tackle the question.

Alternatively, use a pandas dataframe (see, e.g., <https://cmdlinetips.com/2018/02/how-to-subset-pandas-dataframe-based-on-values-of-a-column/>)

```
a = [1,2,3,4,5,6]
print(a)
b = [(x + 1) for x in a]
print(b)
c = [x**2 for x in range(10)]
print(c)
print(sum(c))
```

 [1, 2, 3, 4, 5, 6]
[2, 3, 4, 5, 6, 7]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
285

```
## Answer Q2
retweets = [tweet for tweet in dataset if tweet.is_retweet]
print(len(retweets))
print(len(retweets)/len(dataset))

hillary = [tweet for tweet in dataset if tweet.handle == 'HillaryClinton']
```

```

hillary = [tweet for tweet in dataset if tweet.handle == 'hillaryclinton']
print(len(hillary))
print(len(hillary)/len(dataset))

trump = [tweet for tweet in dataset if tweet.handle == 'realDonaldTrump']
print(len(trump))
print(len(trump)/len(dataset))

```

```

722
0.11204220980757294
3226
0.5006207324643079
3218
0.4993792675356921

```

Question 3. Which tweet was the most popular in terms of retweets? And what is the top 3 most popular tweets? Once found, you can look up the original tweet on Twitter by filling in the correct handle and id in the following URL: <https://twitter.com/{handle}/status/{id}>

For this question, you can also use for loops to find the most popular tweet. Python, however, has good support for sorting datastructures. Take a look below how the built-in `sorted()` function works. (Alternatively, use pandas `sort_values()` on the pandas dataframe.)

```

a = ['bbbbb', 'ee', 'aaa', 'cccc', 'd']
print(sorted(a))
print(sorted(a, reverse=True))
print(sorted(a, key=lambda x:len(x)))

```

```

['aaa', 'bbbbb', 'cccc', 'd', 'ee']
['ee', 'd', 'cccc', 'bbbbb', 'aaa']
['d', 'ee', 'aaa', 'cccc', 'bbbbb']


```

```

## Answer Q3
top = sorted(dataset, key=lambda x:x.retweet_count, reverse=True)[0]
print(top)

top3 = sorted(dataset, key=lambda x:x.retweet_count, reverse=True)[0:3]
for tweet in top3:
    print(tweet)


```

 740973710593654784 / HillaryClinton: Delete your account. <https://t.co/Oa92sncRQY>
740973710593654784 / HillaryClinton: Delete your account. <https://t.co/Oa92sncRQY>
741007091947556864 /realDonaldTrump: How long did it take your staff of 823 people to think that up--and where are your 33,000
755788382618390529 /realDonaldTrump: The media is spending more time doing a forensic analysis of Melania's speech than the FBI

Question: Who (Trump or Hillary) gets the most retweets on average? And based on the median value?

```
## Answer Q4
from statistics import mean, median
print(mean(tweet.retweet_count for tweet in hillary))
print(mean(tweet.retweet_count for tweet in trump))

print(median(tweet.retweet_count for tweet in hillary))
print(median(tweet.retweet_count for tweet in trump))
```

 2983.64724116553
5812.216904909882
1723.5
4469.5

▼ 1.2 Data visualisation

We'll now illustrate some visualizations of the data with Matplotlib and Pyplot. Matplotlib is the fundamental visualisation tool in Python, and Pyplot is an abstraction layer on top of Matplotlib in such a way that Matlab users feel right at home. These modules are, however, not a part of the core Python framework and should be installed separately.

Below we show the needed imports. In order to visualise the plots in the notebook, we use a jupyter-specific syntax `%matplotlib inline` to tell the engine to render the images right within the notebook.

(Note: As an alternative to matplotlib, especially for people familiar with [ggplot](#) in R, you may want to explore, e.g., the [plotnine library](#) in python.)

```
import matplotlib
import matplotlib.pyplot as plt
```



```
import matplotlib.pyplot as plt
```

```
%matplotlib inline
```

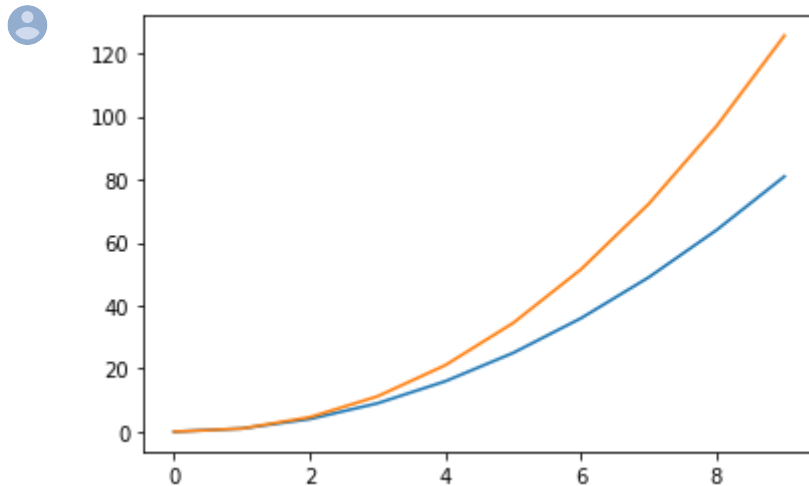
Below we show some basic plotting examples that you can play with, but you must realise that plotting is always a matter of trial and error, and requires frequent searches on Google and StackOverflow for existing recipes that you can use.

▼ Line plots

```
import math

x = range(10)
y = [i**2 for i in x]
yy = [i**2.2 for i in x]

plt.clf()          # clear figure
plt.plot(x, y)     # draw a plot
plt.plot(x, yy)    # draw another plot
plt.show()         # show the figure
```

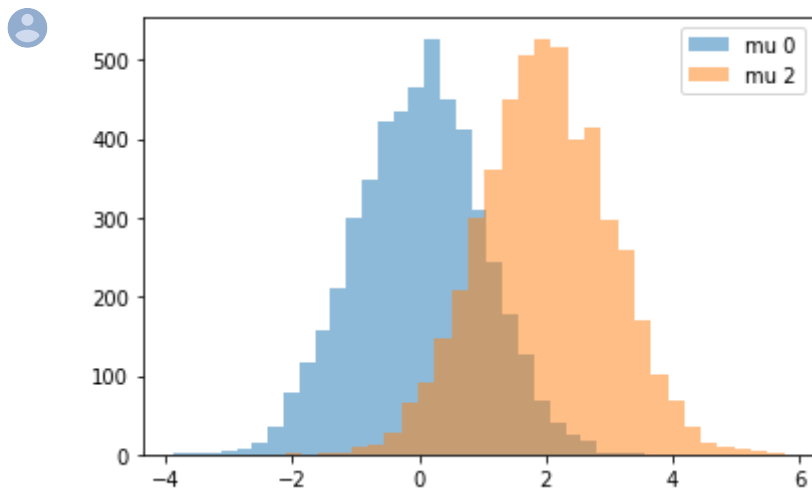


▼ Histograms

```
import random

d = [random.gauss(0, 1) for i in range(5000)]
e = [random.gauss(2, 1) for i in range(5000)]

plt.clf()
# draw histogram, give them labels, set opacity to 0.5
plt.hist(d, bins=30, label='mu 0', alpha=0.5)
plt.hist(e, bins=30, label='mu 2', alpha=0.5)
# draw legend
plt.legend()
plt.show()
```

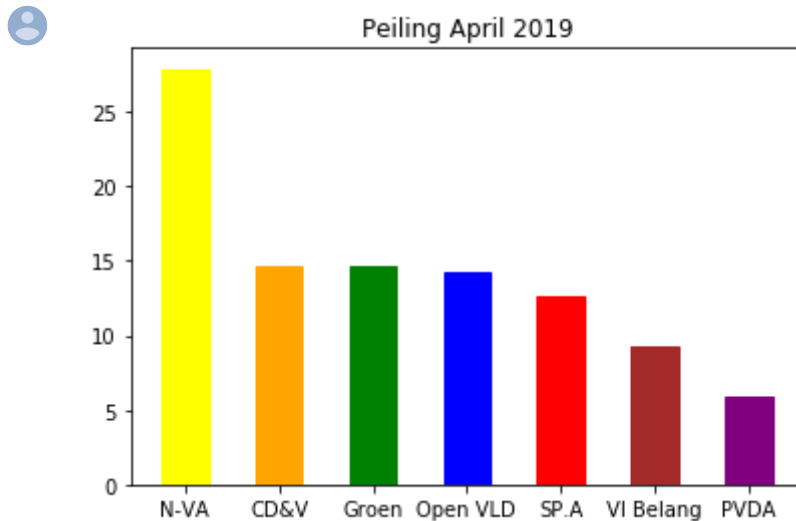


▼ Bar charts

```
x_values = ['N-VA', 'CD&V', 'Groen', 'Open VLD', 'SP.A', 'Vl Belang', 'PVDA']
y_values = [27.9, 14.7, 14.6, 14.2, 12.7, 9.3, 5.9]
colors = ['yellow', 'orange', 'green', 'blue', 'red', 'brown', 'purple']

plt.clf()
# set the plot title
plt.title("Peiling April 2019")
```

```
# create a bar chart
barlist = plt.bar(x=x_values, width=0.5, height=y_values)
# set the colors of each bar
for i in range(len(x_values)):
    barlist[i].set_color(colors[i])
plt.show()
```



▼ Visualisation exercises

Again, Google and StackOverflow are your friends!

Example 1. Generate a line plot of the tweet volume per each month for each candidate.

To get the month in which a tweet was sent, take a look at Python's powerful `datetime` module!

```
import datetime as dt

# Code for Example 1

tweet_volume = {
    'HillaryClinton': [0]*12,
    'realDonaldTrump': [0]*12
```

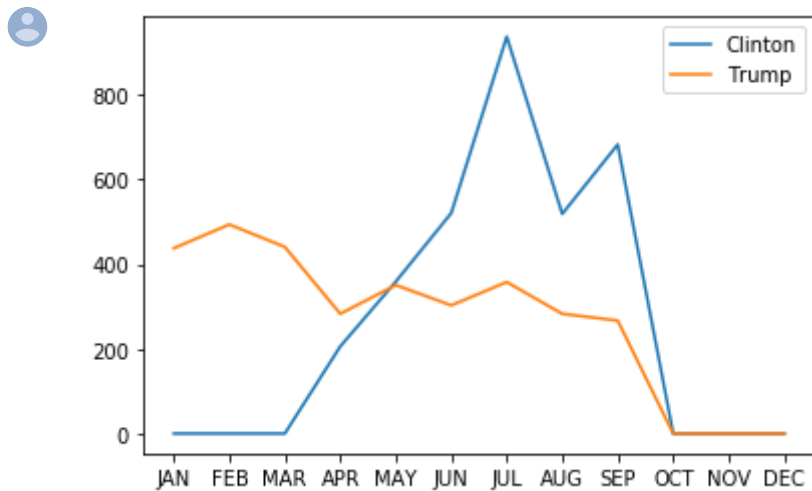
```

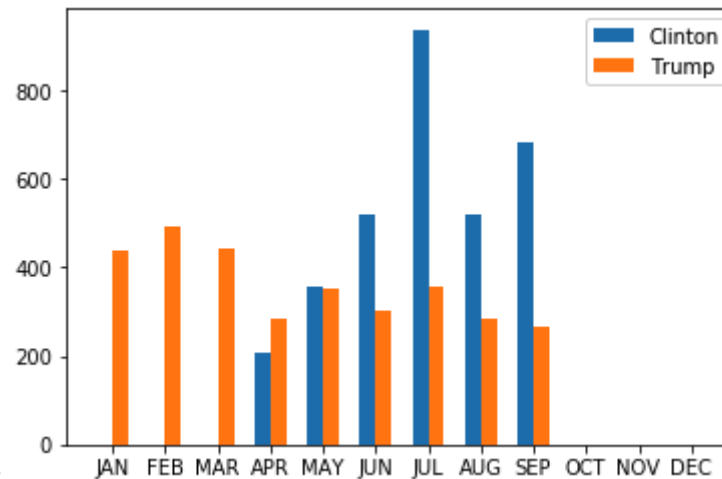
}

for t in dataset:
    tweet_month = dt.datetime.strptime(t.time, '%Y-%m-%dT%H:%M:%S').month - 1
    tweet_volume[t.handle][tweet_month] += 1

plt.clf()
plt.plot(tweet_volume['HillaryClinton'], label='Clinton')
plt.plot(tweet_volume['realDonaldTrump'], label='Trump')
plt.xticks(range(0,12), ['JAN', 'FEB', 'MAR', 'APR', 'MAY', 'JUN', 'JUL', 'AUG', 'SEP', 'OCT', 'NOV', 'DEC'])
plt.legend()
plt.show()

```





Example 2. Do the same, but now use a bar chart. Example:

You will notice that preparing the x-values is quite cumbersome and not flexible enough. That's why we will use histograms in the next assignment.

```
# Example 2:
```

```
x_values = range(0, 12)
```

```
plt.clf()
```

```
plt.bar(x=x_values, width=0.3, height=tweet_volume['HillaryClinton'], label='Clinton')
```

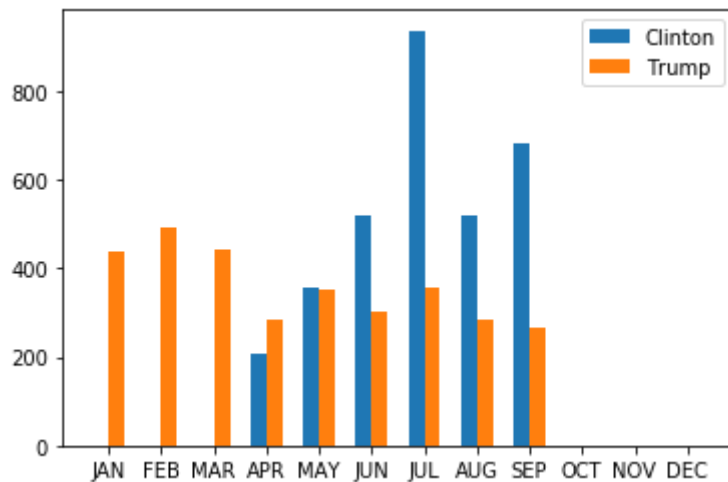
```
plt.bar(x=[k+0.3 for k in x_values], width=0.3, height=tweet_volume['realDonaldTrump'], label='Trump')
```

```
plt.xticks([k+0.15 for k in x_values], ['JAN', 'FEB', 'MAR', 'APR', 'MAY', 'JUN', 'JUL', 'AUG', 'SEP', 'OCT', 'NOV', 'DEC'])
```

```
plt.legend()
```

```
plt.show()
```






Example 3: Maybe dividing the time axis on a monthly level is too coarse. Histograms have the powerful capacity of easily tuning the number of bins, so that we can go as coarse or as fine-grained as we want. We now create a histogram of tweet volume as a function of time, per candidate.

```
# Example 3:

tweet_dates = {
    'HillaryClinton': [],
    'realDonaldTrump': []
}

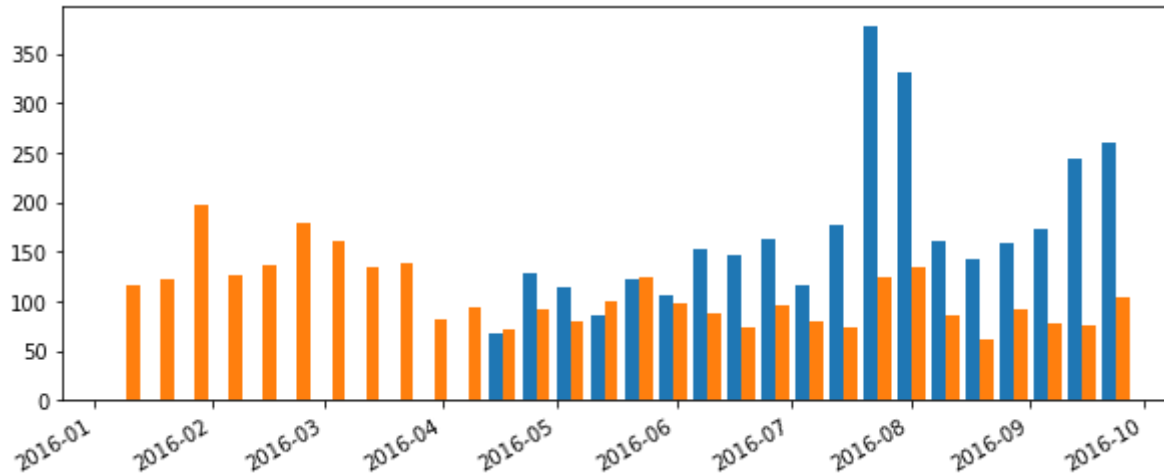
for t in dataset:
    tweet_date = dt.datetime.strptime(t.time, '%Y-%m-%dT%H:%M:%S')
    tweet_dates[t.handle].append(tweet_date)

plt.clf()
plt.figure(figsize=(10,4))
plt.hist([tweet_dates['HillaryClinton'], tweet_dates['realDonaldTrump']], bins=30)
plt.gcf().autofmt_xdate()
plt.show()
```

 /usr/local/lib/python3.6/dist-packages/pandas/plotting/_matplotlib/converter.py:103: FutureWarning: Using an implicitly register

To register the converters:

```
>>> from pandas.plotting import register_matplotlib_converters
>>> register_matplotlib_converters()
warnings.warn(msg, FutureWarning)
<Figure size 432x288 with 0 Axes>
```



Question: What are the Twitter accounts that Hillary Clinton retweets? Create a bar chart for the accounts that were retweeted at least 5 times. Create the same graph for Trump. Who is retweeting most?

```
## Answer Q4 (part 1: Hillary Clinton)
accounts = {
    'HillaryClinton': [],
    'realDonaldTrump': []
}
for tweet in hillary:
    if tweet.is_retweet:
        accounts['HillaryClinton'].append(tweet.original_author)
for tweet in trump:
    if tweet.is_retweet:
        accounts['realDonaldTrump'].append(tweet.original_author)
```

```

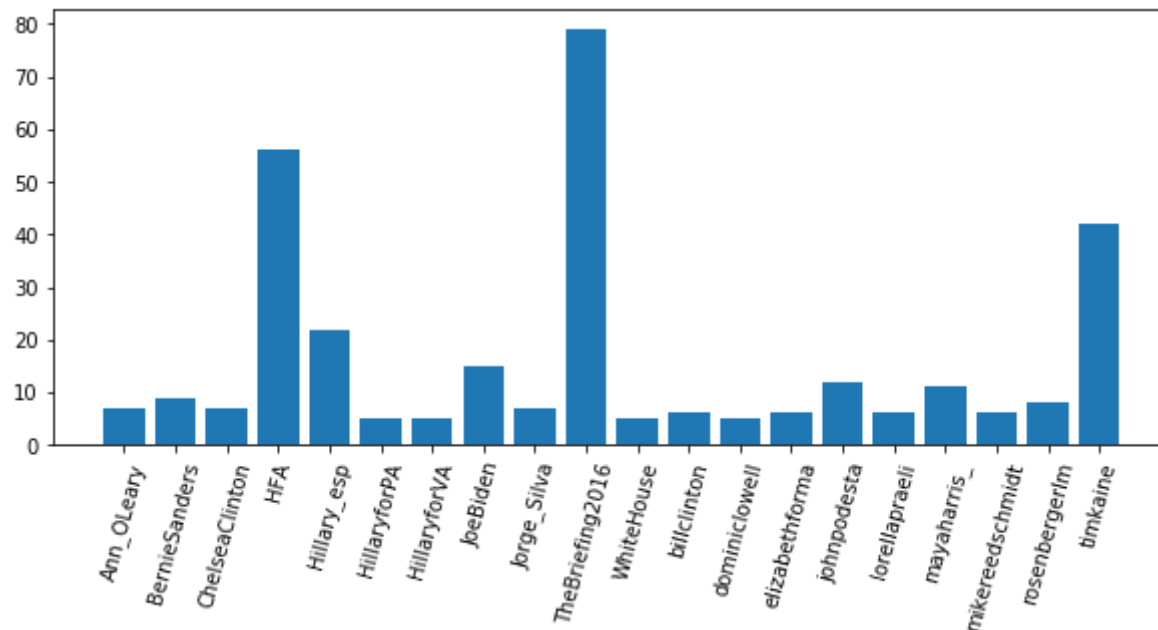
from itertools import groupby
freqs_hillary = {key:len(list(group)) for key, group in groupby(sorted(accounts['HillaryClinton']))}
freqs_hillary = {key: value for key, value in freqs_hillary.items() if value >= 5}
freqs_trump = {key:len(list(group)) for key, group in groupby(sorted(accounts['realDonaldTrump']))}
freqs_trump = {key: value for key, value in freqs_trump.items() if value >= 5}

plt.clf()
plt.figure(figsize=(10,4))
plt.bar(freqs_hillary.keys(), freqs_hillary.values())
plt.xticks(rotation=75)
plt.show()

```



<Figure size 432x288 with 0 Axes>



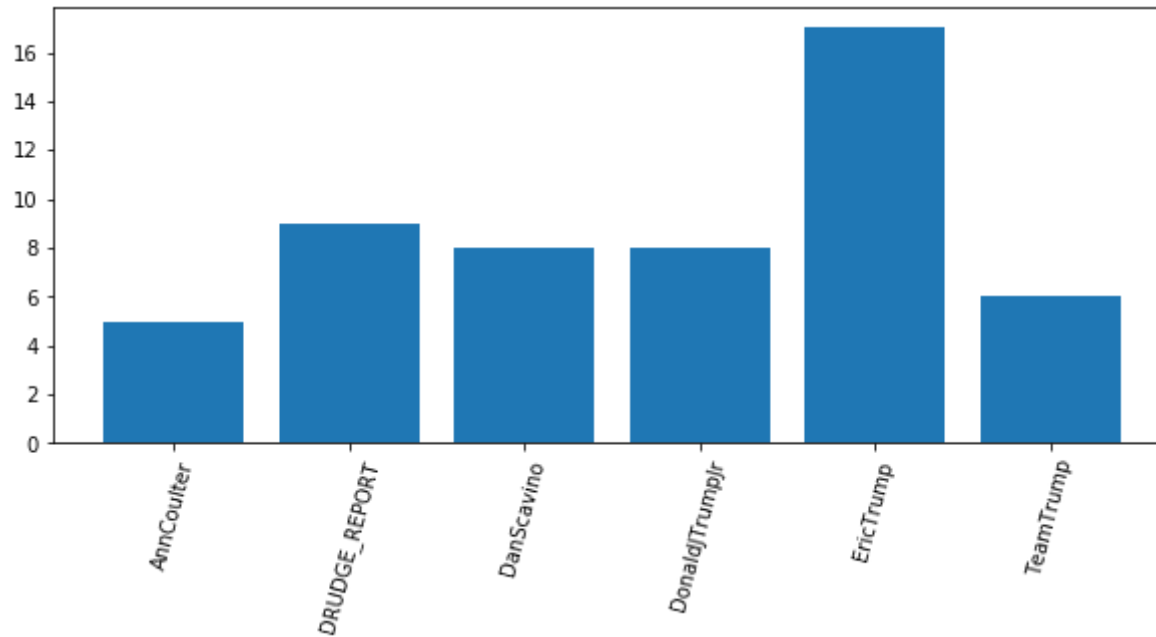
```

## Answer Q4 (part 2: Donald Trump)
plt.clf()
plt.figure(figsize=(10,4))
plt.bar(freqs_trump.keys(), freqs_trump.values())
plt.xticks(rotation=75)
plt.show()

```




<Figure size 432x288 with 0 Axes>



```
## Answer Q4 (part 3: Who is retweeting the most?)  
print(len(accounts['HillaryClinton']))  
print(len(accounts['realDonaldTrump']))
```



597
125

▼ 2. Text Processing

2.1 Cleaning and preprocessing

In the remainder of the lab session, we will focus on textual data. Performing analytics, business intelligence, machine learning, etc. on (written) text is called *Natural Language Processing (NLP)* in academics. In the next few exercises we will learn the necessary skills and steps in

preparing and transforming text into useful data that can be used in a wide variety of powerful machine learning applications.

2.1.1 Theory

The very first step in NLP is *preprocessing*, that is, preparing the raw textual data such that we get rid of (most of the) noise and retain the most informative signal. Consider for example the following tweet, and how we can apply multiple possible preprocessing steps to arrive at a noise-free tweet. **All steps are optional** and one should always consider the application at hand to determine which preprocessing steps are needed!

HEY @UGentstudent! This isn't a clean #NLP tweet :-D <http://www.ugent.be>

Step 1 - Convert all characters to lowercase. By doing this, the words '*This*' and '*this*' become the same, which is what we want.

hey @ugentstudent! this isn't a clean #nlp tweet :-d <http://www.ugent.be>

Step 2 - Apply normalization, for example:

- Replacing all numbers by a single character, e.g. 0.
- Replacing all characters with accents by their clean counterpart, e.g. é becomes e.
- Expanding popular contractions such as "*isn't*" to "*is not*".
- ...

hey @UGentstudent! this is not a clean #nlp tweet :-D <http://www.ugent.be>

Step 3 - Remove unwanted words, punctuation, URLs, whitespaces... For example, we could choose to only retain alphanumerical characters, but this always depends on the application at hand. In our case, we will want to keep hashtags and mentions, but remove URLs and emoticons/emojis.

hey @UGentstudent! this is not a clean #nlp tweet

Step 4 - Splitting the text into separate words. This is a process called *tokenization* and there exist many different methods of tokenizing a text. The most simple method finds all whitespaces and uses them to split the text. More advanced tokenizers also take into account punctuation and other textual markers.

```
['hey', '@UGentstudent', '!', 'this', 'is', 'not', 'a', 'clean', '#nlp', 'tweet']
```

Step 5 - Remove stop words. These are words that hardly contribute to the meaning of a text, such as "the", "a", "an", "is", "and", "our", etc. There are lists available of common stop words in English (e.g. <https://gist.github.com/sebleier/554280>). Some lists include words such as "not", but such words can shape the semantic meaning of a text, so always be careful which words get removed from your text.

```
['hey', '@UGentstudent', '!', 'not', 'clean', '#nlp', 'tweet']
```

Step 6 - In some applications (such a document classification) it can also be useful to apply *stemming* to the text (<https://en.wikipedia.org/wiki/Stemming>). Stemming essentially means that words are normalized, such that for example "fishing", "fished" and "fisher" all reduce to the same stem "fish". There exist many stemming algorithms (such as Porter stemming), but we will not cover stemming in this lab session.

▼ 2.1.2 Regular expressions

URLs appear very often in tweets and contribute almost no semantic message. It is essentially noise, and we therefore want to remove them. For this purpose, we will use regular expressions, a powerful string processing tool and syntax that is available in almost all high-level programming languages. Regular expressions (regex in short) can be very daunting to work with and can be a complete course on its own! Luckily there are useful online tools such as RegexpPlanet (www.regexpplanet.com) and StackOverflow that can help you in the process.

As an exemplary exercise, we will remove all non alphanumeric characters from our tweet using regexes. Consider the following examples, play with them, and after that, try to come up with a way of removing all non-alphanumeric characters from our tweet (but keeping the spaces).

```
import re

print(re.sub(r'de', '', 'abcde'))      #abc
print(re.sub(r'[ae]', '', 'abcde'))    #bcd
print(re.sub(r'[a-c]', '', 'abcde'))   #de
print(re.sub(r'[a-bd-e]', '', 'abcde')) #c
print(re.sub(r'^a-b', '', 'abcde'))    #ab
print(re.sub(r'a*', '', 'aaaabbc'))     #bbc
print(re.sub(r'a?bbb', '', 'aabbbbc')) #abc
print(re.sub(r'a+bbb', '', 'aabbbbc'))  #bc
```

```
print(re.sub(r'^[a-e]', '_', 'AaBbCcDdEe')) #_a_b_c_d_e
```

```
abc
bcd
de
c
ab
bbc
abc
bc
_a_b_c_d_e
```

```
original_tweet = "HEY @UGentstudent! This isn't a clean #NLP tweet :-D http://www.ugent.be"
print(re.sub(r'^[a-zA-Z0-9\s]', '', original_tweet))
print(re.sub(r'^\w\s]', '', original_tweet))
```

```
HEY UGentstudent This isnt a clean NLP tweet D httpwwwugentbe
HEY UGentstudent This isnt a clean NLP tweet D httpwwwugentbe
```

As stated above, we could teach hours on regular expressions alone. A nice layout of what regexes can do in Python, is given in the official documentation: <https://docs.python.org/3/howto/regex.html>. Some useful shortcuts are given there: `\s` stands for any whitespace character, `\d` stands for a digit, `\w` stands for any alphanumeric character, `.` stands for any character.

Here is a regular expression that can be used to remove URLs from a piece of text, and we test it on the original tweet above:

```
original_tweet = "HEY @UGentstudent! This isn't a clean #NLP tweet :-D http://www.ugent.be"
url_regex = r'https?://\S+'
# or (more complex and more general):
# url_regex = r'\w+:\/\/{2}[\d\w-]+(\.([\d\w-]+)(?:\.[^\/\s/]*))*'

re.sub(url_regex, '', original_tweet)
```

```
"HEY @UGentstudent! This isn't a clean #NLP tweet :-D "
```

The following example uses a regular expression to replace all numbers (i.e. one or more digits) with a single character '0' in a given text.

```
number_regex = r'\d+'

re.sub(number_regex, '0', '1, 2, 3 up to 45678.90')
```

👤 '0, 0, 0 up to 0.0'

Now, we find all non-alphanumeric characters in the whole dataset. For this purpose, we use the `set` datastructure.

```
punct = set()
for t in dataset:
    purged_tweet = re.sub(r'\w', '', t.text).strip()
    if len(purged_tweet) > 0:
        for character in purged_tweet:
            punct.add(character)

print(punct)
```

👤 {'\n', '[', ' ', ']', '👍', '👎', '*', '-', '□', '□', ',', '!', '👁', '\u200b', '😡', '❤', '...', '🍷', '@', '👊', '👂', ' '}

▼ 2. Cleaning tweets

We will now extend the `Tweet` class by writing a new function that performs all necessary text cleaning for that tweet. We will save the cleaned text as a new variable in the object.

Question: Copy the code of the `Tweet` class above and write a new method `clean(self)`. This method will perform a series of cleaning operations on the tweet's text. The resulting cleaned text is saved as a new attribute `self.cleaned_text` of that class. Do the following cleaning operations in given order:

1. Convert the original text to lowercase.
2. Remove URLs.
3. Replace all numbers by 0.
4. Remove punctuation, but **be careful not to lose whitespaces, #, @, ' (single quote)**. For this you can use the set of non-alphanumeric characters you have found above! Since a lot of punctuation characters start with a backslash, it is easier to use the built-in `replace()`

function on a Python string to remove a punctuation character.

5. Replace all whitespace with a single space.

6. Make sure the whitespaces at the beginning and end of the tweet are removed (look at the `strip()` function).

Afterwards, import the CSV file again, and perform cleaning on all tweets. Take a look at the resulting cleaned texts and fix any mistakes: data cleaning is always an iterative process in which you improve your cleaning algorithm step by step.

```
punct = {'👤', '[', ']', '~', '👉', '👈', '🏠', '- ', '👤', 'u', '"""', '±', '\u200a', ';', '👉', ' ', '!', '%', ',', '👉', '®', '👉',  
  
punct.remove(' ') # keep spaces  
punct.remove('#') # keep hashtags  
punct.remove('@') # keep mentions  
punct.remove('\') # keep single quotes (in order to retain I'm, isn't, etc.)  
  
class Tweet:  
    def __init__(self, tweet_id, handle, text, is_retweet, original_author, time, lang, retweet_count, favorite_count):  
        self.tweet_id = tweet_id  
        self.handle = handle  
        self.text = text  
        self.is_retweet = is_retweet == 'True'  
        self.original_author = original_author  
        self.time = time  
        self.lang = lang  
        self.retweet_count = int(retweet_count)  
        self.favorite_count = int(favorite_count)  
  
    def __str__(self):  
        return self.tweet_id + ' / ' + self.handle + ": " + self.text  
  
    def clean(self):  
        ## Answer: add function to clean tweets:  
        self.cleaned_text = self.text.lower()  
  
        url_regex = r'https?:\/\/\S+'  
        self.cleaned_text = re.sub(url_regex, '', self.cleaned_text)
```

```

number_regex = r'\d+'
self.cleaned_text = re.sub(number_regex, '0', self.cleaned_text)

for punctuation in punct:
    self.cleaned_text = self.cleaned_text.replace(punctuation, ' ')

self.cleaned_text = re.sub('\s+', ' ', self.cleaned_text)

```

```

dataset = []

with open('tweets_stripped.csv', 'r') as data_file:
    datareader = csv.reader(data_file, delimiter=',', quotechar='"')
    for i, row in enumerate(datareader):
        if i > 0:
            t = Tweet(row[0], row[1], row[2], row[3], row[4], row[5], row[10], row[11], row[12])
            t.clean()
            dataset.append(t)

```

```

for t in dataset[:10]:
    print(t.cleaned_text)

```



the question in this election who can put the plans into action that will make your life better
 last night donald trump said not paying taxes was smart you know what i call it unpatriotic
 couldn't be more proud of @hillaryclinton her vision and command during last night's debate showed that she's ready to be our ne
 if we stand together there's nothing we can't do make sure you're ready to vote
 both candidates were asked about how they'd confront racial injustice only one had a real answer
 join me for a 0pm rally tomorrow at the mid america center in council bluffs iowa tickets
 this election is too important to sit out go to and make sure you're registered #nationalvoterregistrationday h
 when donald trump goes low register to vote
 once again we will have a government of by and for the people join the movement today
 0 has trump offered a single proposal to reduce the friction of starting a business @hillaryclinton has

▼ 3. Tokenization

Tokenization is the process of splitting a text into a sequence of separate words. It is an essential step in NLP, since words are the most basic building blocks that provide a meaning or sentiment to any text. Natural language models often therefore often work with words as input features (although in some models the characters itself are used instead of words).

The most basic type of tokenization is to split the text whenever a whitespace occurs. Take a look at the `split()` function and use it to tokenize the given text:

```
tweet_text = "couldn't be more proud of @hillaryclinton her vision and command during last night's debate showed that she's ready to  
tweet_tokens = tweet_text.split(" ")  
print(tweet_tokens)
```

```
["couldn't", 'be', 'more', 'proud', 'of', '@hillaryclinton', 'her', 'vision', 'and', 'command', 'during', 'last', "night's", 'de
```

It works pretty well already, but some tokens such as *"couldn't"* are ideally expanded to *"could"* and *"not"*. NLTK is Python's Natural Language ToolKit that contains many methods and algorithms to tokenize a text. For example:


```
import nltk  
nltk.download('punkt')  
  
from nltk.tokenize import word_tokenize  
  
tweet_tokens = word_tokenize(tweet_text)  
print(tweet_tokens)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...  
[nltk_data]   Package punkt is already up-to-date!  
['could', "n't", 'be', 'more', 'proud', 'of', '@', 'hillaryclinton', 'her', 'vision', 'and', 'command', 'during', 'last', 'night
```

Question: Take a look at the official NLTK documentation (<https://www.nltk.org/api/nltk.tokenize.html>) and you will notice that it contains a specific Twitter-aware tokenizer. Use it to tokenize the tweet.


```
from nltk.tokenize import TweetTokenizer

## Answer: tokenize tweet with TweetTokenizer
tknzs = TweetTokenizer()
tknzs.tokenize(tweet_text)
```



```
["couldn't",
 'be',
 'more',
 'proud',
 'of',
 '@hillaryclinton',
 'her',
 'vision',
 'and',
 'command',
 'during',
 'last',
 "night's",
 'debate',
 'showed',
 'that',
 "she's",
 'ready',
 'to',
 'be',
 'our',
 'next',
 '@potus']
```

One reason to use Python for machine learning (and NLP) is that thousands of fellow computer scientists and researchers are using it as well for that purpose. Chances are that whatever problem you are trying to tackle, someone has done it already. Regarding tweet tokenization, there are numerous GitHub projects to be found that have solved this problem, some better or more user-friendly than others. We have had pretty good experience with the following GitHub repository: <https://github.com/erikavaris/tokenizer>.

Question: Take a look at the install instructions, install the package, and after that use the package to tokenize the tweet. Also, the package allows for text normalization, thereby expanding "*couldn't*" into "*could*" and "*not*". You can find out how by taking a look at the documentation.

Answer:

```
!pip install git+https://github.com/erikavaris/tokenizer.git
```

```
from tokenizer import tokenizer
```

```
T = tokenizer.TweetTokenizer(regularize=True)
```

```
T.tokenize(tweet_text)
```



```
Collecting git+https://github.com/erikavaris/tokenizer.git
  Cloning https://github.com/erikavaris/tokenizer.git to /tmp/pip-req-build-zpysa2i6
  Running command git clone -q https://github.com/erikavaris/tokenizer.git /tmp/pip-req-build-zpysa2i6
Requirement already satisfied (use --upgrade to upgrade): tokenizer==1.0.1 from git+https://github.com/erikavaris/tokenizer.git
Requirement already satisfied: nltk in /usr/local/lib/python3.6/dist-packages (from tokenizer==1.0.1) (3.2.5)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from nltk->tokenizer==1.0.1) (1.12.0)
Building wheels for collected packages: tokenizer
  Building wheel for tokenizer (setup.py) ... done
  Created wheel for tokenizer: filename=tokenizer-1.0.1-cp36-none-any.whl size=12978 sha256=f94ee034f4038481eb009587c4e19386cf2t
  Stored in directory: /tmp/pip-ephem-wheel-cache-f2tjdn8t/wheels/47/e7/9b/7f3e9b2989a5600b42ffbcc0fd8687562b1738f585f6bd92fa
Successfully built tokenizer
['could',
 'not',
 'be',
 'more',
 'proud',
 'of',
 '@hillaryclinton',
 'her',
 'vision',
 'and',
 'command',
 'during',
 'last',
 "night's",
 'debate',
 'showed',
 'that',
 'she',
 'is',
 'ready',
 'to',
 'be',
 'our',
 'next',
 '@potus']
```

Keep in mind that tokenization is never perfect, and should be tuned to your application at hand!


```

        for punctuation in punct:
            self.cleaned_text = self.cleaned_text.replace(punctuation, ' ')

        self.cleaned_text = re.sub('\s+', ' ', self.cleaned_text)

    def tokenize(self):
        self.tokens = T.tokenize(self.cleaned_text)

```

Now, test the tokenization on the tweet dataset:


```

dataset = []

with open('tweets_stripped.csv', 'r') as data_file:
    datareader = csv.reader(data_file, delimiter=',', quotechar='"')
    for i, row in enumerate(datareader):
        if i > 0:
            t = Tweet(row[0], row[1], row[2], row[3], row[4], row[5], row[10], row[11], row[12])
            t.clean()
            t.tokenize()
            dataset.append(t)

for t in dataset[:3]:
    print(t.tokens)

```

 ['the', 'question', 'in', 'this', 'election', 'who', 'can', 'put', 'the', 'plans', 'into', 'action', 'that', 'will', 'make', 'you', 'know', 'what', 'i', 'call', 'it', 'last', 'night', 'donald', 'trump', 'said', 'not', 'paying', 'taxes', 'was', 'smart', 'you', 'know', 'what', 'i', 'call', 'it', 'could', 'not', 'be', 'more', 'proud', 'of', '@hillaryclinton', 'her', 'vision', 'and', 'command', 'during', 'last', "night's"]

▼ 4. Building a vocabulary

For machine learning models it is important that datapoints have a numerical representation. Many machine learning models are vector-based, and each class of data has is assigned to its own dimension in these vectors. What this means for us now, is that each word should be mapped to its own unique positive integer value. For example, consider a dictionary of the entire English language, then each word can be mapped to the


position of that word in the dictionary. The sentence "*proud of hillary*" can then be represented as a sequence of indices, e.g. [243, 5, 8723]. In other words, the text is transformed into a mathematical representation, which opens the gate to all sorts of powerful machine learning methods and models. Such models will be explored in the next lab session, which is why we cover vocabulary building here.

We will now build a vocabulary for the Twitter dataset. This vocabulary will be used throughout the remainder of the lab session.

Question: Use the `set` datastructure to find all unique words in the dataset, thereby using the tokenized tweets. How big is the vocabulary?

```
## Answer:
voc = set()
for tweet in dataset:
    voc.update(tweet.tokens)

print(len(voc))
```

 9258

Question: Now create a dictionary `word_to_ix` which maps a word to a unique index (from 0 up to the length of the vocabulary). Also, create a dictionary `ix_to_word` which does the opposite mapping. Check if you can translate a word to an index, and use that index to find the original word again. What is the theoretical time complexity of a dictionary lookup?

```
## Answer part 1: create word_to_ix and ix_to_word
word_to_ix = {}
ix_to_word = {}
for ix, token in enumerate(voc):
    word_to_ix[token] = ix
    ix_to_word[ix] = token
```

```
## Answer part 2: test these on some words (e.g., 'hillary', ...)
print(word_to_ix['hillary'])
print(ix_to_word[1509])
print(ix_to_word[0])
print(word_to_ix['@stephencurry0'])
```

```
print("The theoretical time complexity of a dictionary lookup is constant: O(c)")
```



5036

towers

found

8418

The theoretical time complexity of a dictionary lookup is constant: O(c)

Some additional questions and exercises for which you can use the precalculated tokens or the vocabulary. Feel free to play around with the data as you like.

Question 1. Which words are used most by Trump and Hillary? Create a bar chart for each candidate to illustrate this.

Question 2. Which Twitter accounts are mentioned most by Trump and Hillary?

*Tip: to get better results for the first question, take a look at NLTK on how to **ignore common English stopwords**.*

Tip: for question Q2, try out the use of `defaultdict`, a dictionary for which new keys are automatically assigned some default value (here: `int 0`)

```
## Answer to Q1
nltk.download('stopwords')
from nltk.corpus import stopwords

stop_words = set(stopwords.words('english'))

tokens_hillary = []
for tweet in dataset:
    if tweet.handle == 'HillaryClinton':
        tokens_hillary.extend([w for w in tweet.tokens if not w in stop_words])
freqs_hillary = {key:len(list(group)) for key, group in groupby(sorted(tokens_hillary))}
freqs_hillary = {key: value for key, value in freqs_hillary.items() if value >= 100}

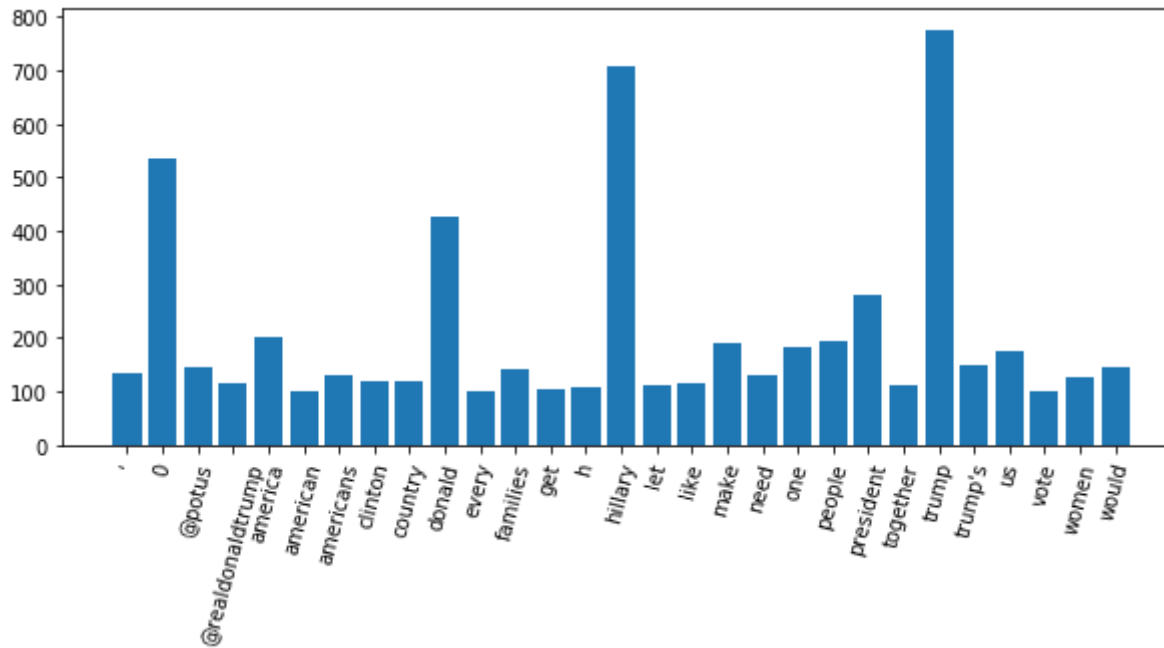
plt.clf()
plt.figure(figsize=(10,4))
plt.bar(freqs_hillary.keys(), freqs_hillary.values())
plt.xticks(rotation=75)
plt.show()
```

```
tokens_trump = []
for tweet in dataset:
    if tweet.handle == 'realDonaldTrump':
        tokens_trump.extend([w for w in tweet.tokens if not w in stop_words])
freqs_trump = {key:len(list(group)) for key, group in groupby(sorted(tokens_trump))}
freqs_trump = {key: value for key, value in freqs_trump.items() if value >= 100}

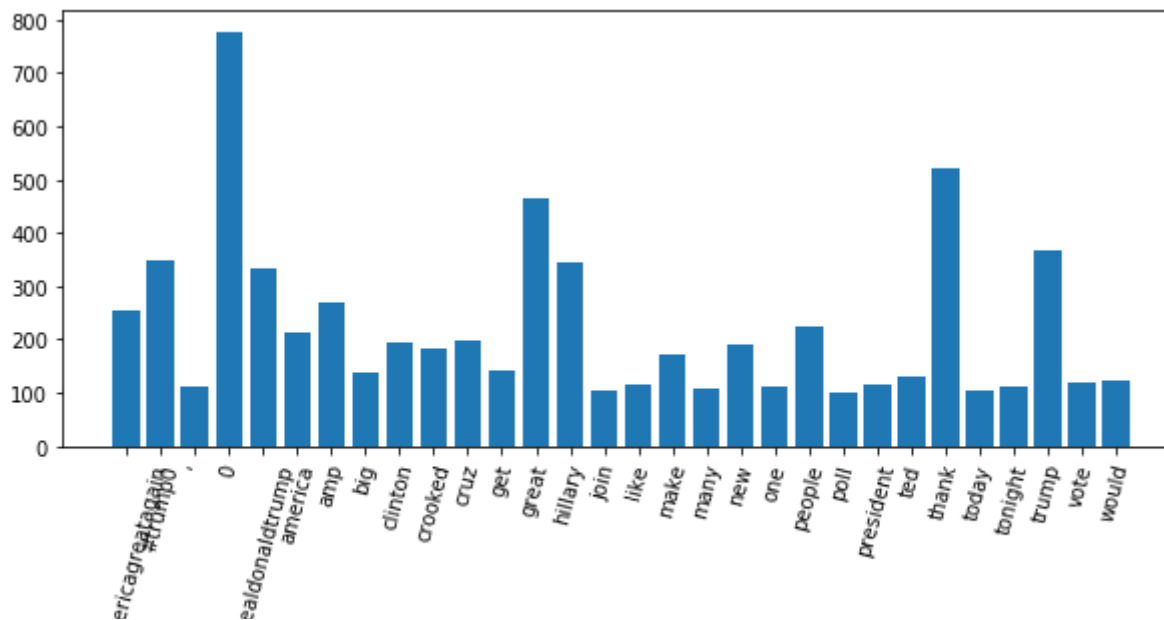
plt.clf()
plt.figure(figsize=(10,4))
plt.bar(freqs_trump.keys(), freqs_trump.values())
plt.xticks(rotation=75)
plt.show()
```



[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Package stopwords is already up-to-date!
<Figure size 432x288 with 0 Axes>



<Figure size 432x288 with 0 Axes>



#makeam

@h

```
## Answer to Q2
from collections import defaultdict

hillary_accounts = defaultdict(lambda: 0)
trump_accounts = defaultdict(lambda: 0)

for tweet in dataset:
    if tweet.handle == 'HillaryClinton':
        for w in tweet.tokens:
            if w[0] == '@':
                hillary_accounts[w] += 1
    if tweet.handle == 'realDonaldTrump':
        for w in tweet.tokens:
            if w[0] == '@':
                trump_accounts[w] += 1

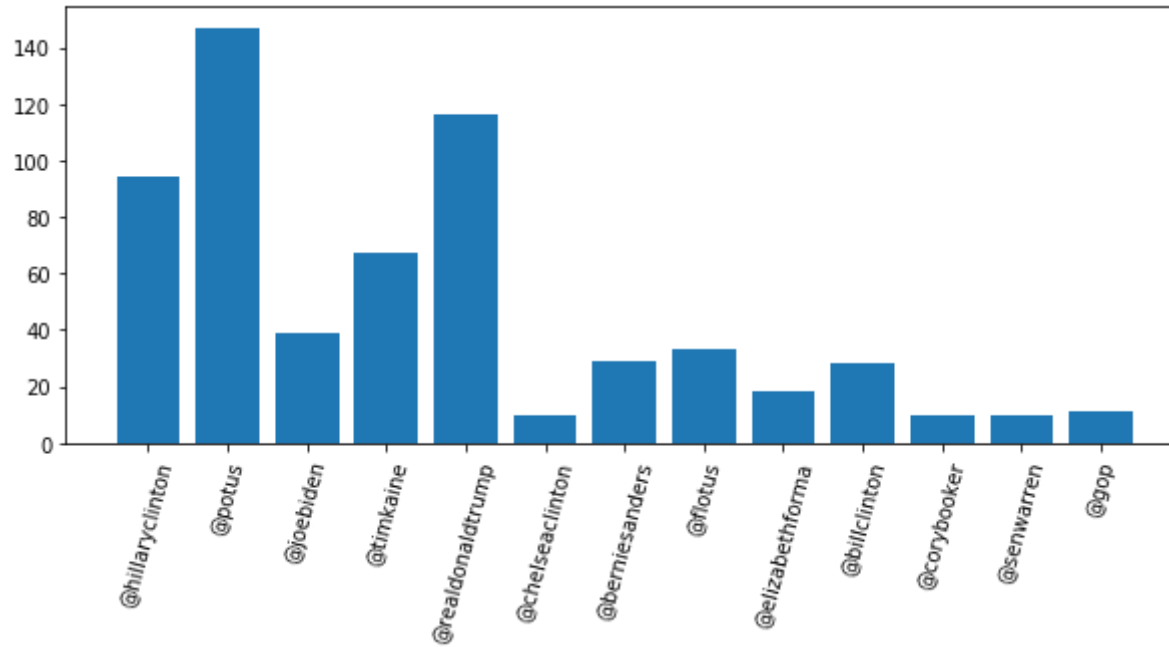
hillary_accounts = {key: value for key, value in hillary_accounts.items() if value >= 10}
trump_accounts = {key: value for key, value in trump_accounts.items() if value >= 10}

plt.clf()
plt.figure(figsize=(10,4))
plt.bar(hillary_accounts.keys(), hillary_accounts.values())
plt.xticks(rotation=75)
plt.show()

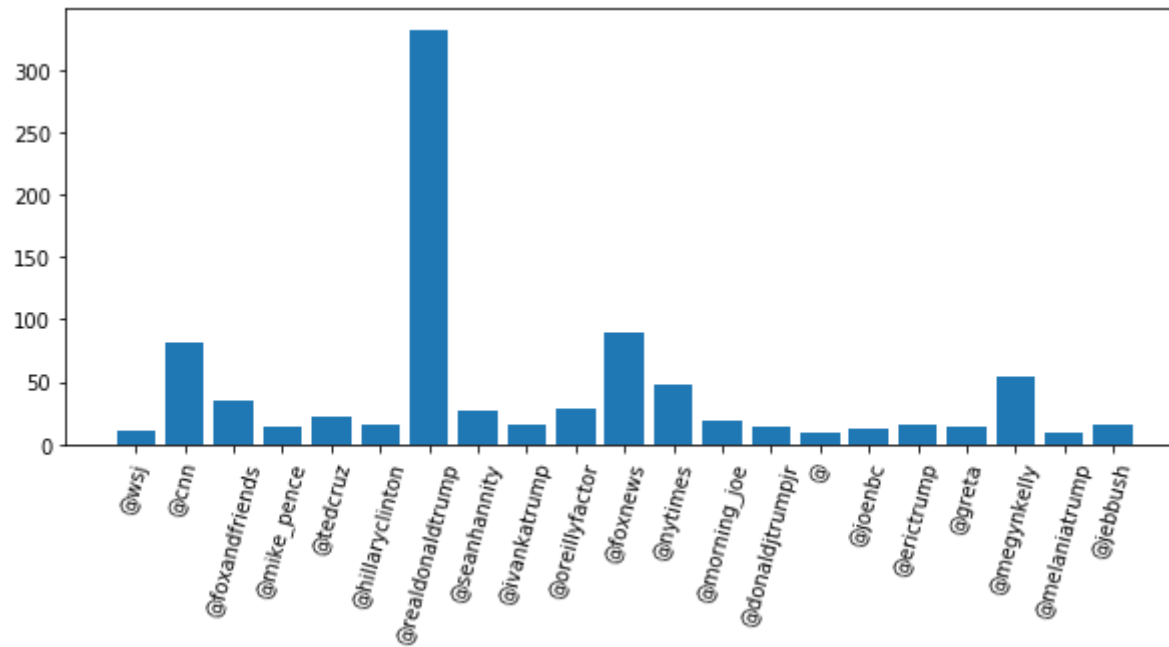
plt.clf()
plt.figure(figsize=(10,4))
plt.bar(trump_accounts.keys(), trump_accounts.values())
plt.xticks(rotation=75)
plt.show()
```



<Figure size 432x288 with 0 Axes>



<Figure size 432x288 with 0 Axes>



At this point, we are ready to explore more advanced predictive analytics on texts. Therefore, make sure to be up to speed with all text preprocessing concepts that were covered here.

Rule-based systems are among the very first AI systems. They can execute a task by following a set of rules that were predetermined by a human. For example: a very simple rule for a basic thermostat would be to start heating when the temperature drops below 20 degrees, and to switch off when the temperature rises above 22 degrees. Rule-based systems can be powerful, but are not flexible, can become really complex and it is very time consuming to determine a proper set of rules.

Nevertheless, it is always a good exercise to create and experiment with a rule-based AI system. As a matter of exercise, we will build a simple classifier that will predict whether a tweet comes from Donald Trump or from Hillary Clinton based on the principle of **majority voting**. For this, use the following steps. Feel free to tweak the rules of the game!

Question 3. For all words in the dataset, count how many times Trump and Hillary use that word. E.g. Trump uses the word 'great' 5623 times, and Hillary only 820 times. It gives you the quantities $N_{H,t}$ and $N_{T,t}$ for resp. the number of times Hillary and Trump use token t .

```
## Answer to Question 3
hillary_word_count = defaultdict(lambda: 0)
trump_word_count = defaultdict(lambda: 0)

for tweet in dataset:
    if tweet.handle == 'HillaryClinton':
        for w in tweet.tokens:
            hillary_word_count[w] += 1
    if tweet.handle == 'realDonaldTrump':
        for w in tweet.tokens:
            trump_word_count[w] += 1
print(trump_word_count['great'])
print(hillary_word_count['great'])
hillary_word_count_filtered = {key: value for key, value in hillary_word_count.items() if value >= 250}
trump_word_count_filtered = {key: value for key, value in trump_word_count.items() if value >= 250}

plt.clf()
plt.figure(figsize=(10,4))
plt.bar(hillary_word_count_filtered.keys(), hillary_word_count_filtered.values())
```

```
plt.xticks(rotation=75)
plt.show()

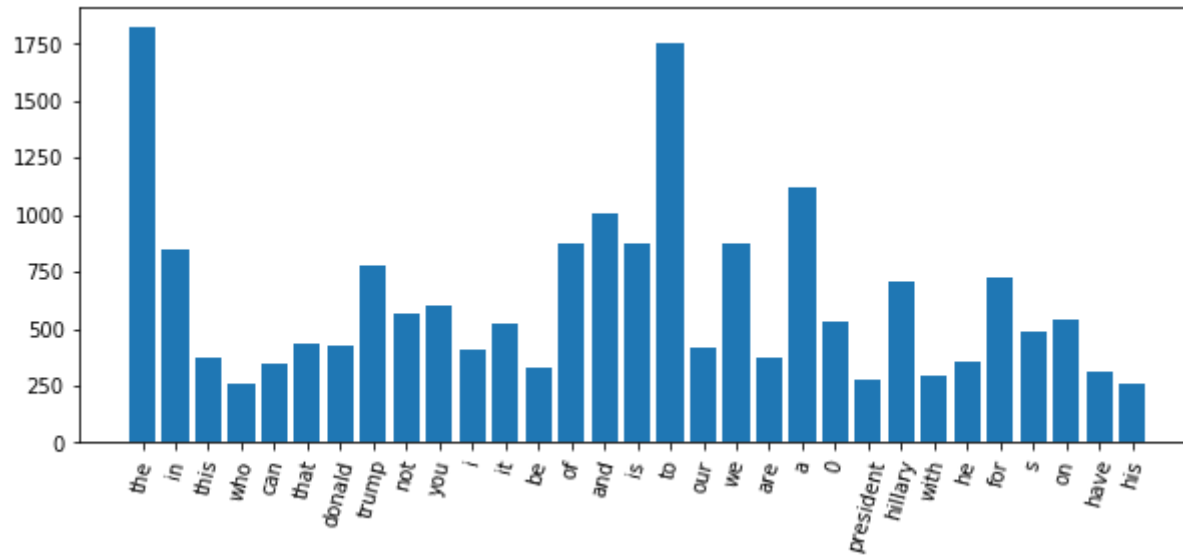
plt.clf()
plt.figure(figsize=(10,4))
plt.bar(trump_word_count_filtered.keys(), trump_word_count_filtered.values())
plt.xticks(rotation=75)
plt.show()
```



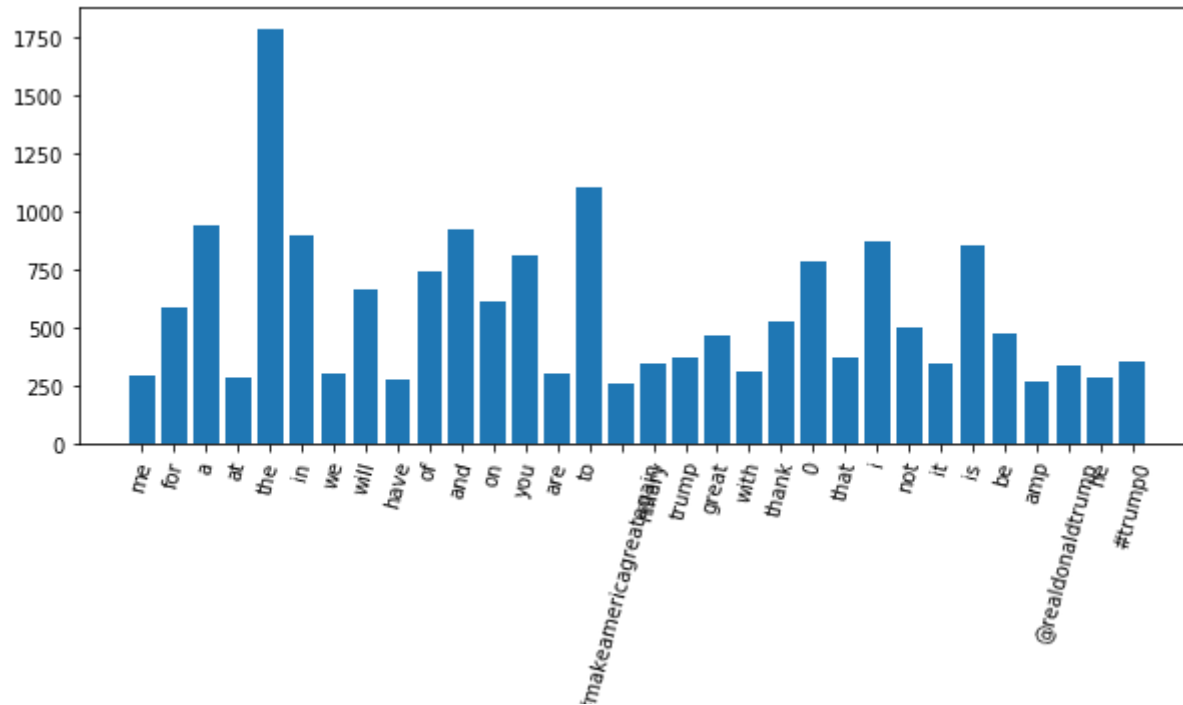
466

68

<Figure size 432x288 with 0 Axes>



<Figure size 432x288 with 0 Axes>



Question 4. Now take a new tweet (not in the dataset) and for every word (token) in that tweet, perform majority voting. If $N_{H,t} > N_{T,t}$, a vote is given to Hillary for that tweet. Otherwise, Trump gets a vote. Whoever gets the most votes for the given tweet, is the winner, and is therefore also the predicted author of the tweet. Test this on the tweet given below.

```
tweet_to_classify = "make america great again"

## Answer to Question 4
points_hillary = 0
points_trump = 0
for w in T.tokenize(tweet_to_classify):
    if hillary_word_count[w] > trump_word_count[w]:
        points_hillary += 1
    else:
        points_trump += 1

print(points_hillary)
print(points_trump)
```



1
3

▼ Part 2 - Sentiment Analysis

▼ Machine Learning in Python



Scikit-Learn

- Machine learning library written in **Python**

- **Simple and efficient**, for both experts and non-experts
- Classical, **well-established machine learning algorithms**
- Shipped with [documentation](#) and [examples](#)

```
# if necessary, install the needed packages by uncommenting the following lines:
# ! conda install numpy scipy scikit-learn jupyter matplotlib -y
# ! conda install python-graphviz -y
# ! pip install pydotplus

import warnings
warnings.filterwarnings('ignore')

# Optimized operations on arrays and lists
import numpy as np

# Data processing
import pandas as pd

# General machine Learning
import sklearn

# Visualization
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

▼ Data

▼ IMDB Movie Reviews



when `_star wars_` came out some twenty years ago , the image of traveling throughout the stars has become a commonplace image . [...]
when han solo goes light speed , the stars change to bright lines , going towards the viewer in lines that converge at an invisible point .
cool .
`_october sky_` offers a much simpler image—that of a single white dot , traveling horizontally across the night sky . [...]



“ snake eyes ” is the most aggravating kind of movie : the kind that shows so much potential then becomes unbelievably disappointing .
it’s not just because this is a brian depalma film , and since he’s a great director and one who’s films are always greeted with at least some fanfare .
and it’s not even because this was a film starring nicolas cage and since he gives a brauvara performance , this film is hardly worth his talents .

```
from google.colab import drive
drive.mount('/content/drive')
```



Drive already mounted at /content/drive; to attempt to forcibly remount, call `drive.mount("/content/drive", force_remount=True)`

Question:

- Load in the IMDB reviews stored as csv file `train.csv` using `pandas.read_csv()`
- Print the column headers of the csv file

Answer:

```
data = pandas.read_csv('train.csv')
for col in data.columns:
    print(col)
```



```
id
label
rating
text
```

▼ Data Exploration

Questions:

- How many reviews are included in the collection?
- How many different tokens are used in the reviews?
- What is the average rating of all the reviews?
- What is the average length of a review?


Note: alle text has been tokenized and seperated using white space, thus, calling `.split()` on text is sufficient for tokenization in this tutorial.

```
## Answer:
print(len(data.index))

tokens = set()
for d in data.text:
    for w in d.split():
        tokens.add(w)
print(len(tokens))

print(data.rating.mean())

data.text_len = data.text.apply(len)
print(data.text_len.mean())
```

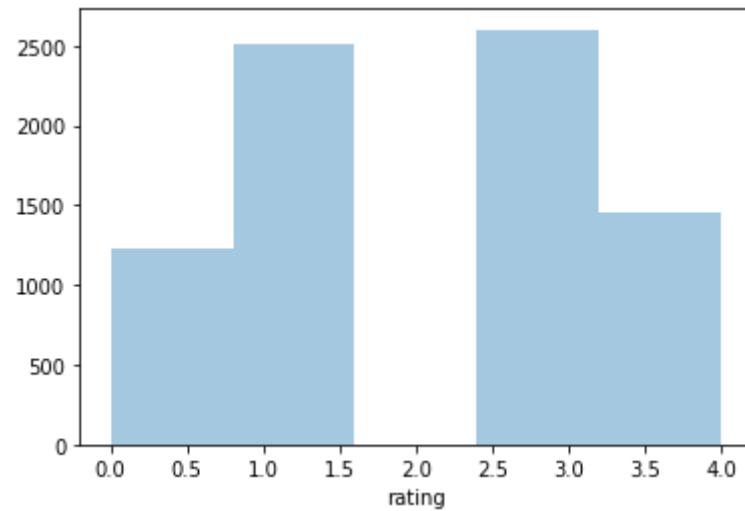
 7792
17354
2.0690451745379876
103.92415297741273

We can now take a look at the rating distribution:

```
sns.distplot(data['rating'], 5, kde=False)
```



<matplotlib.axes._subplots.AxesSubplot at 0x7f326bac1128>

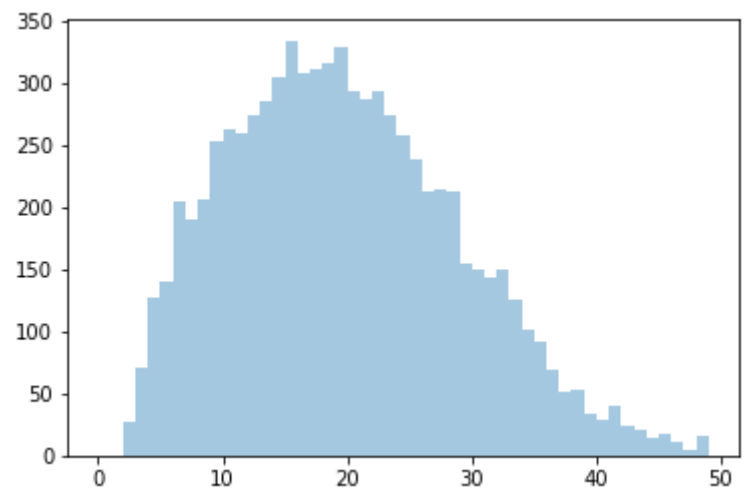


```
print(len([len(x.split()) for x in data['text']]))  
sns.distplot([len(x.split()) for x in data['text']], range(0,50,1), kde=False)
```



7792


<matplotlib.axes._subplots.AxesSubplot at 0x7f326c96a208>



Question:

- How is the vocabulary distributed? (Make a histogram of the word counts.)

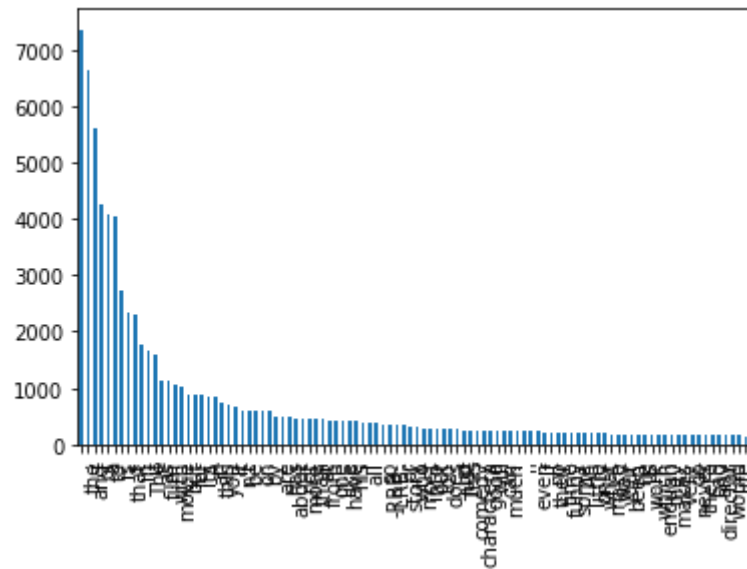
```
## Answer
word_counts = data.text.apply(lambda x: x.split()).explode().value_counts()
print(word_counts)
word_counts.nlargest(100).plot(kind='bar')
```



```

.          7368
,          6630
the        5595
and        4268
of         4095
...
Ado         1
untrained   1
adultery    1
ideological  1
systematically 1
Name: text, Length: 17354, dtype: int64
<matplotlib.axes._subplots.AxesSubplot at 0x7f326b03f7f0>

```



As an illustration, below is code to visual the word frequencies in a word cloud:

```
# Illustration: plot frequencies in a word cloud

# if necessary, install the needed packages by uncommenting the following line:
! pip install wordcloud

from wordcloud import WordCloud, STOPWORDS

# Store text from pandas data frame in list
text = ' '.join(data['text'][data['label']==1]) # join all reviews in training set
print('Example text:\n', data['text'][0])

# limit word count
wordcount = 1000
# stop words
stopwords = set(STOPWORDS)
stopwords.add("br")
print('\nEnglish stopwords:\n', ', '.join(sorted(list(stopwords))))

# setup word cloud
wc = WordCloud(scale=3, background_color="white", max_words=wordcount, stopwords=stopwords, width=800, height=600)

# generate word cloud
wc.generate(text)

# show
print('\nWordcloud:')
plt.figure(figsize=(12,8))
plt.imshow(wc, interpolation='bilinear')
plt.axis("off")
plt.show()
```



Requirement already satisfied: wordcloud in /usr/local/lib/python3.6/dist-packages (1.5.0)

Requirement already satisfied: pillow in /usr/local/lib/python3.6/dist-packages (from wordcloud) (6.2.2)

Requirement already satisfied: numpy>=1.6.1 in /usr/local/lib/python3.6/dist-packages (from wordcloud) (1.17.5)

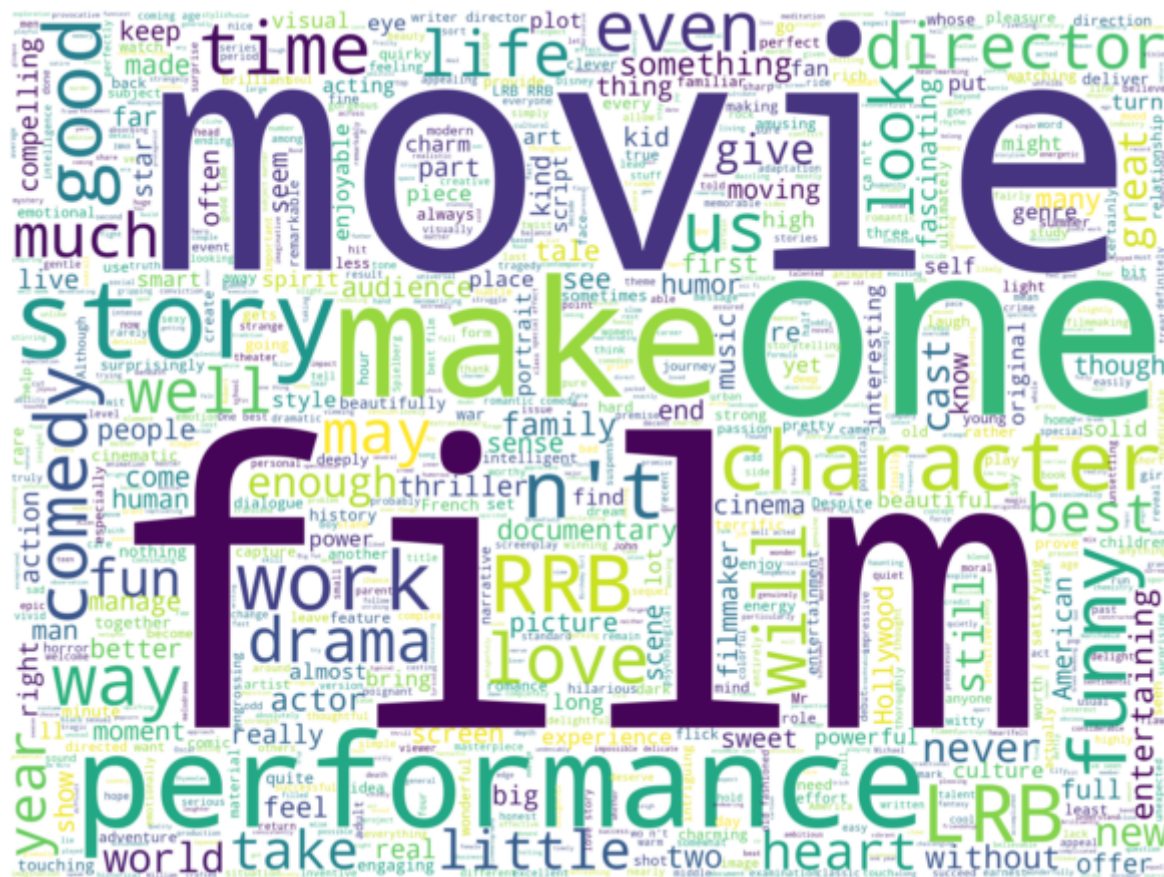
Example text:

The Rock is destined to be the 21st Century 's new `` Conan '' and that he 's going to make a splash even greater than Arnold 's

English stopwords:

a, about, above, after, again, against, all, also, am, an, and, any, are, aren't, as, at, be, because, been, before, being, be:

Wordcloud:



Supervised Machine Learning

Data comes as a finite learning set $\mathcal{L} = (X, y)$ where

- Input samples are given as an array X of shape `n_samples` \times `n_features`, taking their values in \mathcal{X} ;
- Output values are given as an array y , taking *symbolic* values in \mathcal{Y} .

The goal of supervised classification is to build a function $f : \mathcal{X} \mapsto \mathcal{Y}$ minimizing the expected error over the training data:

$$Err(\varphi) = \mathbb{E}_{X,Y} \{\ell(Y, \varphi(X))\}$$

where ℓ is a loss function, e.g., the zero-one loss for classification $\ell_{01}(Y, \hat{Y}) = 1(Y \neq \hat{Y})$.

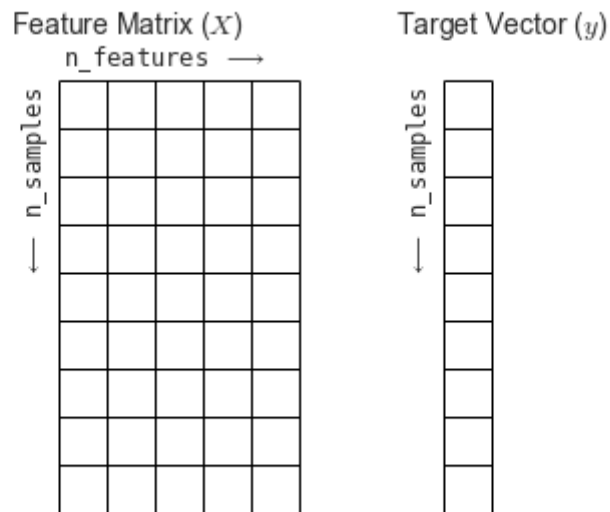
Applications

- Diagnosing disease from symptoms;
- Recognising cats in pictures;
- Identifying body parts with Kinect cameras;
- ...

▼ Feature Extraction

Now that we have the data, we need to build some sort of **feature representation** of our data. One of the simplest things we can do is to represent each sentence as a bag of its words. As part of determining what constitutes a word (or "token"), we'll have to choose how to tokenize the data. Let's do the simplest thing for now and just split on whitespace. More sophisticated methods might use a tokenizer from an outside library, such as NLTK or SpaCy.

The most intuitive way to do so is to use a **Bags of Words** representation using using [CountVectorizer](#)



Assign a fixed integer id to each word occurring in any document of the training set (for instance by building a dictionary from words to integer indices). For each document i , count the number of occurrences of each word w and store it in $X[i, j]$ (row i and column j in the feature matrix X) as the value of feature j where j is the index of word w in the dictionary. The bags of words representation implies that $n_features$ is the number of distinct words in the corpus: this number is typically larger than 100,000.

Fortunately, most values in X will be zeros since for a given document less than a few thousand distinct words will be used. For this reason we say that bags of words are typically high-dimensional **sparse** datasets. We can save a lot of memory by only storing the non-zero parts of the feature vectors in memory.

`scipy.sparse` matrices are data structures that do exactly this, and `scikit-learn` has built-in support for these structures

Below, we

- Import a `CountVectorizer` from the `sklearn.feature_extraction.text` package.
- Create a `CountVectorizer` object named `tokenizer`

[See documentation for parameters of this object](#)

```
# creating a tokenizer
```



```
# Creating a tokenizer

from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

tokenizer = CountVectorizer()
```

▼ Exercise 1

- The `fit()` method of the vectorizer object learns a vocabulary dictionary of all tokens in the raw documents. Fit the tokenizer using `tokenizer.fit()`. Explore the tokenizer object (e.g. `dir(tokenizer)`) and the fit function (e.g., `help(tokenizer.fit)`)
- Transform text from the training collection using `tokenizer.transform(data['text'])` into a vector named `x`
- Print the dimensions of the feature-matrix, `x`
- Print the tokenizer dictionary using `tokenizer.vocabulary_`
- Show the feature representation of the first document: what features are non-zero?
- Finetune the tokenizer:
 - Specify the tokenizer to strip accents.
 - Create a tokenizer that learns unigrams as well as bigrams, fit the data, and print the number of features.
 - Reduce the tokenizer to remove stopwords, only retain tokens that occur 2 or more times, and print the number of different words
 - Reduce to only the 5.000 most often occurring tokens.

```
## Answer to exercise 1
tokenizer.fit(data.text)
print(dir(tokenizer))
print(help(tokenizer.fit))

X = tokenizer.transform(data.text)
print(X.shape)

print(tokenizer.vocabulary_)
```

```
print(X[0])
```

```
tokenizer2 = CountVectorizer(strip_accents='ascii', ngram_range=(1,2), stop_words='english', min_df=2, max_features=5000)  
X = tokenizer2.fit_transform(data.text)  
print(X.shape)  
print(X[0])
```



```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getstate__', '__help__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Help on method fit in module sklearn.feature_extraction.text:

fit(raw_documents, y=None) method of sklearn.feature_extraction.text.CountVectorizer instance

Learn a vocabulary dictionary of all tokens in the raw documents.

Parameters

raw_documents : iterable

An iterable which yields either str, unicode or file objects.

Returns

self

None

(7792, 14599)

{'the': 12948, 'rock': 10854, 'is': 6924, 'destined': 3455, 'to': 13133, 'be': 1153, '21st': 74, 'century': 2050, 'new': 8660,

(0, 74) 1

(0, 601) 1

(0, 781) 1

(0, 1153) 1

(0, 2050) 1

(0, 2308) 1

(0, 2599) 1

(0, 3128) 1

(0, 3455) 1

(0, 4470) 1

(0, 5585) 1

(0, 5693) 1

(0, 5969) 1

(0, 6924) 1

(0, 7002) 1

(0, 7845) 1

(0, 8660) 1

(0, 8964) 1

(0, 10854) 1

(0, 11178) 1

(0, 11301) 1

(0, 12070) 1

(0, 12264) 1

(0, 12943) 1

(0, 12947)	1
(0, 12948)	2
(0, 13133)	2
(0, 13881)	1
(7792, 5000)	
(0, 3578)	1
(0, 1036)	1
(0, 21)	1
(0, 628)	1
(0, 2809)	1
(0, 1720)	1
(0, 2482)	1
(0, 4085)	1
(0, 1762)	1
(0, 256)	1
(0, 3707)	1
(0, 2139)	1
(0, 4682)	1
(0, 4140)	1
(0, 22)	1
(0, 1721)	1

▼ Classification Models

The goal of supervised classification is to build a function $f : \mathcal{X} \mapsto \mathcal{Y}$ minimizing

$$Err(f) = \mathbb{E}_{X,Y} \{\ell(Y, f(X))\}$$

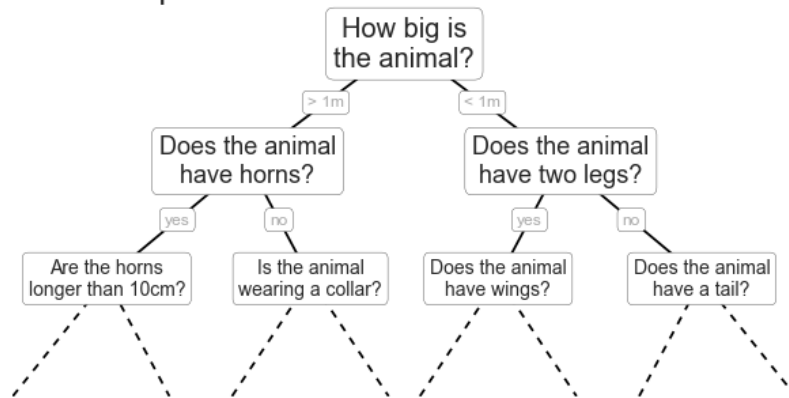
where ℓ is a loss function, e.g., the zero-one loss for classification $\ell_{01}(Y, \hat{Y}) = 1(Y \neq \hat{Y})$.

▼ Decision Tree

Decision Trees are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a token occurs in a review or not), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.

[A visual introduction to decision trees](#)

Example Decision Tree: Animal Classification



- Import the `DecisionTreeClassifier` from the `sklearn.tree` module. [See documentation](#)
- Create a `DecisionTreeClassifier` object named `dt_clf`; limit the number of leaf nodes to 5 and use information gain as criterion.

```
## Answer
from sklearn.tree import DecisionTreeClassifier


dt_clf = DecisionTreeClassifier(max_leaf_nodes=5, criterion='entropy')
```

In order to learn the "best" parameters for our model based on the training data, use scikit-learn's `dt_clf.fit(features, vectors)` method. Inside this method, the parameters are according to some loss function (see slides).

- Store sentiment labels in a `y` variable
- Fit the classifier object calling the `fit()` method

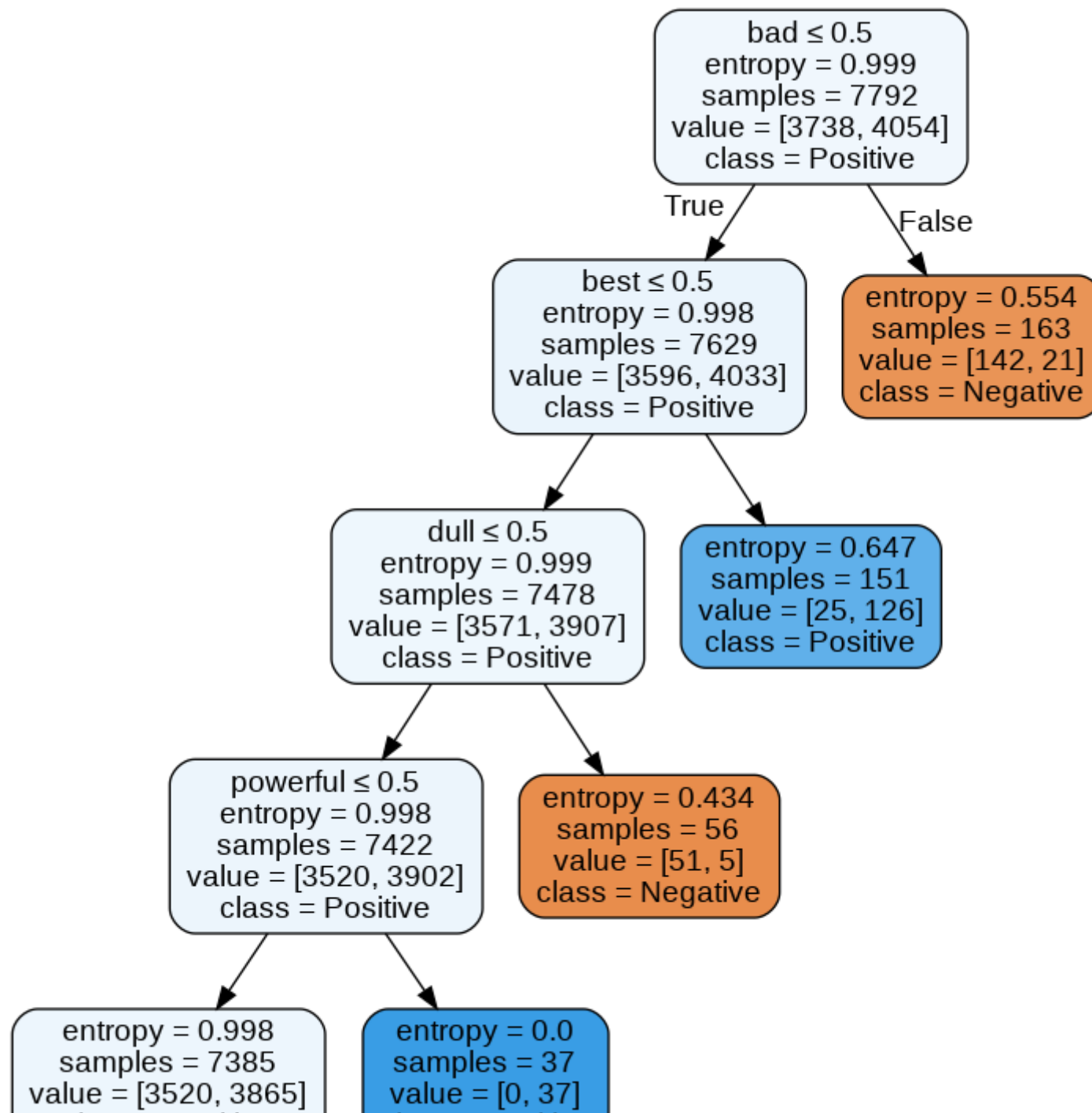
Answer

```
y = data.label  
dt_clf.fit(X, y)
```

 DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='entropy',
max_depth=None, max_features=None, max_leaf_nodes=5,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, presort='deprecated',
random_state=None, splitter='best')

```
from sklearn.tree import export_graphviz  
from IPython.display import Image  
import pydotplus  
import graphviz  
  
dot_data = export_graphviz(dt_clf,  
                           feature_names=list(tokenizer2.get_feature_names()),  
                           filled=True,  
                           rounded=True,  
                           special_characters=True, class_names=['Negative', 'Positive'])  
graph = pydotplus.graphviz.graph_from_dot_data(dot_data)  
Image(graph.create_png())
```





class = Positive

class = Positive

What feature is tested first? Does this decision correspond with your intuition? What about the 2nd test?

<< Insert your answer here in Markdown >>

First, there is looked at the number of times the word 'bad' appears. My intuition says that this is a very good first word to look at. If bad appears one or more times, the sample is predicted to be negative. If bad does not appear, there is looked at 'best'. If it appears one or more times, the sample is classified to be positive. This is again a good and intuitive word to look at.

Naive Bayes Classification

Naive Bayes classifiers are built on Bayesian classification methods. These rely on Bayes's theorem, which is an equation describing the relationship of **conditional probabilities** of statistical quantities. In Bayesian classification, we're interested in finding the probability of a label y given some observed features, which we can write as $P(y \mid x)$. Bayes's theorem tells us how to express this in terms of quantities we can compute more directly:

$$P(y \mid x) = \frac{P(x \mid y)P(y)}{P(x)}$$

If we are trying to decide between negative and positive—then one way to make this decision is to compute the ratio of the posterior probabilities for each label:

$$\operatorname{argmax}_{y \in \{negative, positive\}} P(y|x)$$

All we need now is some model by which we can compute $P(x | y)$ for each label. Such a model is called a *generative model* because it specifies the hypothetical random process that generates the data. Specifying this generative model for each label is the main piece of the training of such a Bayesian classifier. The general version of such a training step is a very difficult task, but we can make it simpler through the use of some simplifying assumptions about the form of this model.

This is where the "naive" in "naive Bayes" comes in: if we make very naive assumptions about the generative model for each label (i.e., the independence of co-occurring words), we can find a rough approximation of the generative model for each class, and then proceed with the Bayesian classification. Different types of naive Bayes classifiers rest on different naive assumptions about the data, and we will examine a few of these in the following sections.

▼ Logistic Regression

Logistic regression models the probability $P(y|x; w)$ that a review, x , falls into a specific category $y \in \{0, 1\}$ (negative, positive). Using a vector of weights, w .

If $P(y = 1|x; w) > 0.5$ then class positive is selected

- For Bag-of-Words vector

$$x = [x_1, x_2, \dots, x_{|V|}], x \in \mathbb{R}^{|V|}$$

- Parameter vector (One per class)

$$w = [w_1, w_2, \dots, w_{|V|}], w \in \mathbb{R}^{|V|}$$

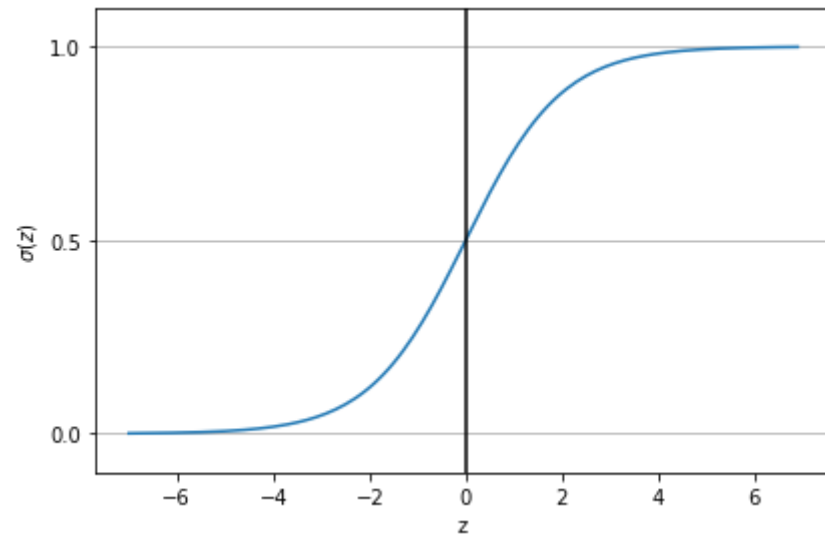
- Linear scoring function f

$$f_w(x) = w_1 x_1 + w_2 x_2 + \dots + w_{|V|} x_{|V|} = w^T x \in \mathbb{R}$$

- Sigmoid Function

$$\sigma(z) = \frac{1}{1 + e^{-z}} \in [0, 1]$$

As z goes from $-\infty$ to ∞ , $\sigma(z)$ goes from 0 to 1



$$P(y = \text{positive} | x; w) = \sigma(f_w(x))$$

$$= \frac{1}{1 + e^{-(w^T x)}}$$

$$\log P(y = \text{positive} | x; w) = -\log(1 + e^{-(w^T x)})$$

Learning is formulated as the maximizing the likelihood of the correct answer or minimizing the negative log of the likelihood (probability) of the correct answer

$$\min_{w \in R^{|V|}} \sum_i^N \log(1 + e^{-(w^T x)})$$

if $\sigma(f(x)) > 0.5$ then class $y = \text{positive}$ is selected

▼ Exercise 2

- Import the `BernoulliNB` from the `sklearn.naive_bayes` module. [See documentation](#)
- Create a new `BernoulliNB` object named `nb_clf`
- Fit the model on the data
- Which are the most significant tokens for both classes? One way to measure the importance of tokens to compute the ratio of the posterior probabilities for each label. Print the feature-weights and feature names learned by the classifier using `tokenizer.get_feature_names()` and `nb_clf.feature_log_prob_`.

$$\frac{P(y = \text{positive} \mid \mathbf{x})}{P(y = \text{negative} \mid \mathbf{x})}$$
$$\log(P(y = \text{positive} \mid \mathbf{x})) - \log(P(y = \text{negative} \mid \mathbf{x}))$$

```
## Answer
from sklearn.naive_bayes import BernoulliNB
```

```
nb_clf = BernoulliNB()
nb_clf.fit(X, y)
print(tokenizer2.get_feature_names())
print(nb_clf.feature_log_prob_[0])
import numpy as np
print(tokenizer2.get_feature_names()[np.argmax(nb_clf.feature_log_prob_[0] - nb_clf.feature_log_prob_[1])])
print(tokenizer2.get_feature_names()[np.argmax(nb_clf.feature_log_prob_[1] - nb_clf.feature_log_prob_[0])])
```


```
['10', '10 minutes', '100', '100 minutes', '101', '11', '12', '12 year', '13', '15', '15 years', '19', '1970s', '1984', '19th',
[-5.45425217 -6.14739935 -6.43508142 ... -6.43508142 -7.53369371
 -6.43508142]
suffers
powerful
```

- Import the `LogisticRegression` from the `sklearn.linear_model` module. [See documentation](#)
- Create a new `LogisticRegression` object named `lr_clf`
- Fit the model on the data
- Print the feature-weights, w , and tokens learned by the classifier using `tokenizer.get_feature_names()` and `lr_clf.coef_`. Which are the most important ones for both classes? Weights provide insights into important features, which tokens are weighted strongly for both classes?

Answer:

```
from sklearn.linear_model import LogisticRegression
```

```
lr_clf = LogisticRegression()
lr_clf.fit(X, y)
print(tokenizer2.get_feature_names())
print(lr_clf.coef_[0])
print(tokenizer2.get_feature_names()[np.argmax(lr_clf.coef_[0])])
print(tokenizer2.get_feature_names()[np.argmin(lr_clf.coef_[0])])
```

 ['10', '10 minutes', '100', '100 minutes', '101', '11', '12', '12 year', '13', '15', '15 years', '19', '1970s', '1984', '19th',
[-0.80286967 -0.03835489 -0.32759231 ... -0.5928834 -0.22934727
0.16630262]
powerful
worst

We now have a trained sentiment analysis model.

In the next section we'll evaluate our models.

▼ Evaluation

Once the model is trained, the main task of supervised machine learning is to evaluate it based on what it says about new data that was not part of the training set. In Scikit-Learn, this can be done using the `predict()` method. For the sake of this example, our "new data" will be a grid of How well does our model do? Let's define a function to see our model's accuracy on some data split and see how well we fit the training data. We'll make use of the `clf.predict()` interface for generating predictions.

▼ Metrics: Accuracy, Precision, Recall, F_1

		<i>gold standard labels</i>		
		gold positive	gold negative	
<i>system output labels</i>	system positive	true positive	false positive	$\text{precision} = \frac{tp}{tp+fp}$
	system negative	false negative	true negative	
		$\text{recall} = \frac{tp}{tp+fn}$		$\text{accuracy} = \frac{tp+tn}{tp+fp+tn+fn}$

$$F = \frac{2 * Precision * Recall}{Precision + Recall}$$

- Import all the metrics mentioned above from the `sklearn.metrics` module

```
## Answer: import metrics
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report
```

▼ Overfitting

- Evaluate accuracy by a classifier on the same data it was trained on using `clf.predict()` and `classification_report`

Answer:

```
clfs = [dt_clf, nb_clf, lr_clf]
for clf in clfs:
    print(classification_report(clf.predict(X), y))
```



	precision	recall	f1-score	support
0	0.05	0.88	0.10	219
1	0.99	0.53	0.69	7573
accuracy			0.54	7792
macro avg	0.52	0.71	0.40	7792
weighted avg	0.97	0.54	0.68	7792
	precision	recall	f1-score	support
0	0.89	0.86	0.88	3865
1	0.87	0.90	0.88	3927
accuracy			0.88	7792
macro avg	0.88	0.88	0.88	7792
weighted avg	0.88	0.88	0.88	7792
	precision	recall	f1-score	support
0	0.94	0.93	0.93	3782
1	0.93	0.94	0.94	4010
accuracy			0.94	7792
macro avg	0.94	0.93	0.93	7792
weighted avg	0.94	0.94	0.94	7792

Issue: the training error is a **biased** estimate of the generalization error.

Solution: Divide \mathcal{L} into disjoint parts called training and test sets (a common choice is using 80% for training and 20% for test).

- Use the training set for fitting the model;
- Use the test set for evaluation only, thereby yielding an unbiased estimate.


This could be done by hand, but it is more convenient to use the `train_test_split` utility function from `sklearn.model_selection`.

- import the `train_test_split` of the `sklearn.model_selection` module
- Create a `X_train`, `X_test`, `y_train`, `y_test` from the data collection
- Print the shapes of the new arrays

Answer:

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=69)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

 (6233, 5000)
(1559, 5000)
(6233,)
(1559,)

- How well do we do on held-out data? Evaluate accuracy by a Logistic regression classifier on the test data using `clf.predict(X_test)`

Answer:

```
for clf in clfs:
    clf.fit(X_train, y_train)
    print(classification_report(clf.predict(X_test), y_test))
```



	precision	recall	f1-score	support
0	0.02	0.82	0.05	22
1	1.00	0.53	0.70	1537
accuracy			0.54	1559
macro avg	0.51	0.68	0.37	1559
weighted avg	0.98	0.54	0.69	1559

	precision	recall	f1-score	support
0	0.78	0.76	0.77	756
1	0.78	0.80	0.79	803
accuracy			0.78	1559
macro avg	0.78	0.78	0.78	1559
weighted avg	0.78	0.78	0.78	1559

	precision	recall	f1-score	support
0	0.77	0.76	0.76	749
1	0.78	0.79	0.79	810
accuracy			0.77	1559
macro avg	0.77	0.77	0.77	1559
weighted avg	0.77	0.77	0.77	1559

We see a big drop, ~20 accuracy, on held-out data, so we overfit the training data. We can go back and revise our approach (e.g. by playing around with the different parameters for the [logistic regression classifier](#)) and re-fitting on the training data, and then see how well we do on the held-out validation data.

By doing this, however, we'll be fitting to the validation data. At some point, we'll want to evaluate one completely new data. Which is what the test split is for. The test split should be used as sparingly as possible!

```
# print(classification_report(y_val, clf.predict(X_val)))
```

▼ Regularization

When models have enough flexibility to nearly perfectly account for the fine features in the data, they can learn to very accurately describe the training data. The model's precise form can become more reflective of the particular noise properties of the training data (which will be different in held-out data) than the intrinsic properties of whatever process generated that data (i.e., the actual 'signal' in the data, which will be present in held-out data as well).

Such a model is said to **overfit** the data: that is, it has so much model flexibility that the model ends up accounting for random errors as well as the underlying data distribution; another way of saying this is that the model has high **variance**, and it may poorly perform on unseen data.

A model is said to **underfit** the data when it does not have enough model flexibility to suitably account for all the features in the data; another way of saying this is that the model has high **bias**.

Overfitting can be countered in many ways. In this tutorial we focus on the technique of **regularization** of a logistic regression classifier. Learning in Logistic Regression is formulated as the optimization of

$$\underbrace{\min_{w \in \mathbb{R}^{|V|}} \sum_i^N \log(1 + e^{-\sigma(w^T x)})}_{Loss}$$

When a linear model overfits, weights tend to become very large. One way to counter this, is to penalize large weights by adding an additional component to the loss function called a regularization.

$$\underbrace{\min_{w \in R^{|V|}} \sum_i^N \log(1 + e^{-(\sigma(w^T x))})}_{\text{Loss}} + \underbrace{\frac{1}{C} ||w||^2}_{\text{Regularization}}$$

Answer:

Same as code provided below?

In words: the higher the value of C, the less regularization.

```
from sklearn.model_selection import validation_curve
```

```
for C in [0.1, 0.5, 1., 5.]:
```

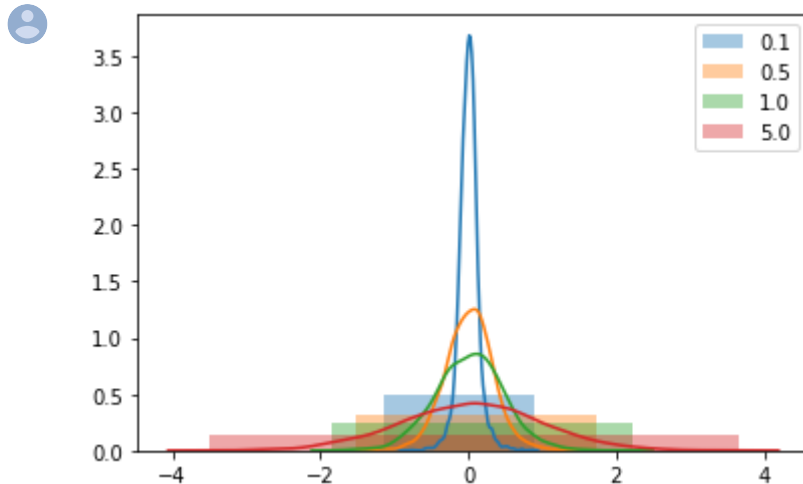
```
    clf = LogisticRegression(C=C)
```

```
    clf.fit(X_train, y_train)
```

```
    sns.distplot(clf.coef_, len(clf.coef_), kde=True, label=str(C))
```

```
plt.legend()
```

```
plt.show()
```



▼ The Bias-Variance trade-off

Fundamentally, the question of "the best model" is about finding a sweet spot in the tradeoff between *bias* and *variance*.

- **Under-fitting, High Bias:** the model is too simple and does not capture the true relation between X and Y.
- **Over-fitting, High Variance:** the model is too specific to the training set and does not generalize.

```
from sklearn.model_selection import validation_curve
from sklearn.ensemble import RandomForestClassifier

# Evaluate parameter range in CV
param_range = [0.001,0.01,0.1,0.4,0.6,1,1.2,1.4,1.6,2]
param_name = "C"
clf = LogisticRegression()

train_scores, test_scores = validation_curve(
    clf, X_train, y_train,
    param_name=param_name,
    param_range=param_range, cv=2, n_jobs=-1, scoring="accuracy")

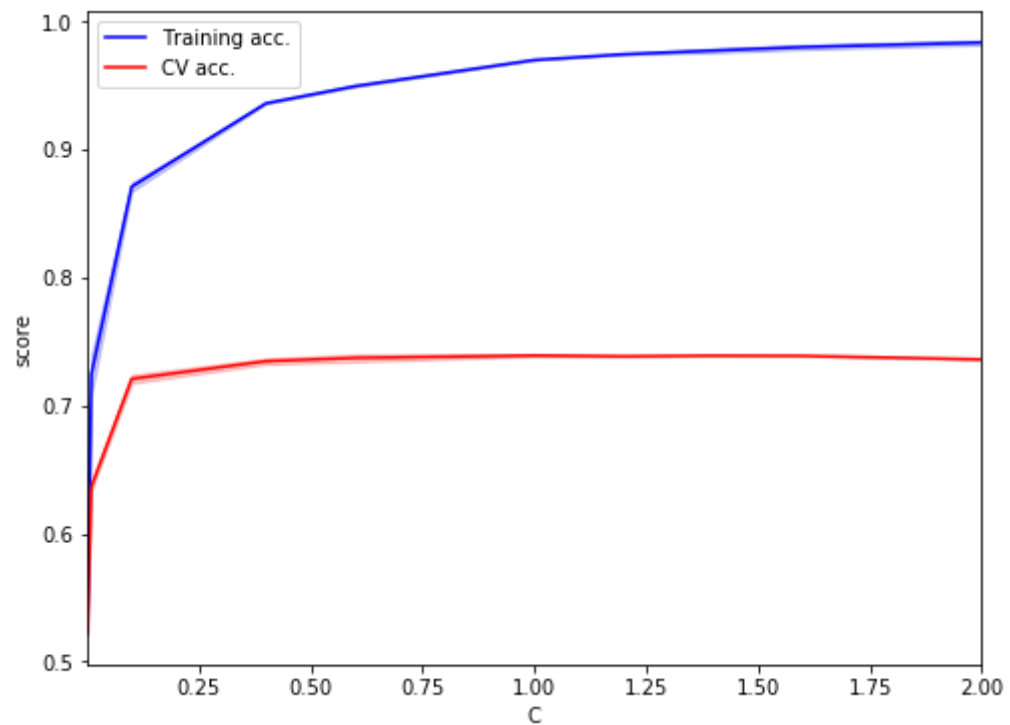
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Plot parameter VS estimated error
plt.figure(figsize=(8,6))
plt.xlabel(param_name)
plt.ylabel("score")
plt.xlim(min(param_range), max(param_range))
plt.plot(param_range, train_scores_mean, color="blue", label="Training acc.")
plt.fill_between(param_range,
                 train_scores_mean + train_scores_std,
                 train_scores_mean - train_scores_std,
                 alpha=0.2, color="blue")
plt.plot(param_range, test_scores_mean, color="red", label="CV acc.")
plt.fill_between(param_range,
                 test_scores_mean + test_scores_std,
                 test_scores_mean - test_scores_std,
                 alpha=0.2, color="red")
```

```
plt.legend(loc="best")
```

```
print(test_scores)
```

```
[[0.5210138  0.52214377]  
 [0.63394289 0.63928113]  
 [0.72409368 0.71726573]  
 [0.73211421 0.73684211]  
 [0.73403914 0.74005135]  
 [0.73820982 0.73973042]  
 [0.737889   0.73876765]  
 [0.73885146 0.73876765]  
 [0.73853064 0.73876765]  
 [0.73628489 0.73555841]]
```





▼ Cross Validation

When evaluating different settings (**hyperparameters**) for estimators, such as the C setting that must be manually set for a logistic regression, there is still a risk of overfitting on the test set because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set can **leak** into the model and evaluation metrics no longer report on generalization performance. To solve this

problem, the data not used for training is split into a **validation set** and the actual **test set**. Training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

A solution to this problem is a procedure called **cross-validation** (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, called k-fold CV, the training set is split into k smaller sets (other approaches are described below, but generally follow the same principles). The following procedure is followed for each of the k “folds”:

A model is trained using all but one of the folds as training data; the resulting model is evaluated on the remaining part of the data (i.e., it is used as a validation set to compute a performance measure such as accuracy). The performance measure reported by k-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set), which is a major advantage in problems such as inverse inference where the number of samples is very small.



Answer


```
from sklearn.model_selection import cross_val_score
```

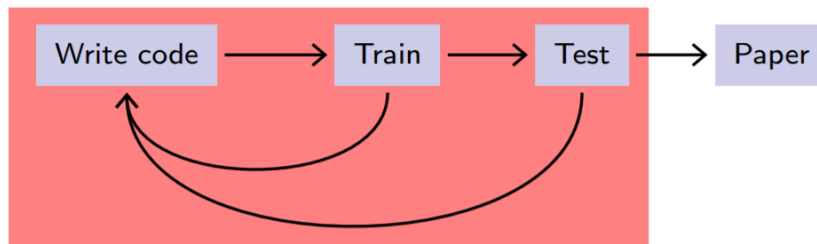
```
clf = LogisticRegression()
```

```
scores = cross_val_score(clf, X_train, y_train, cv=5)
```

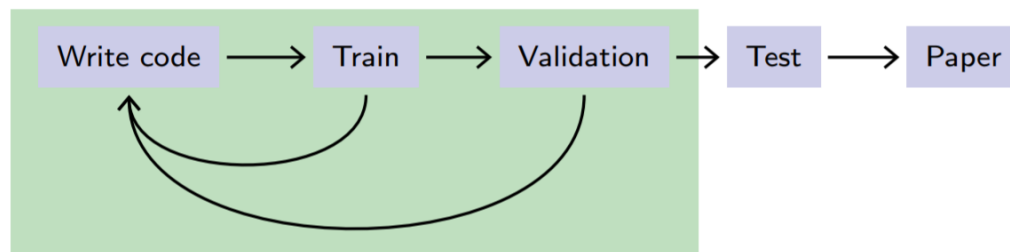
```
print(scores)
```

```
print(np.mean(scores))
```

 [0.74899759 0.75220529 0.76583801 0.76645265 0.76484751]
0.7596682117338434



This should be avoided at all costs. The standard strategy is to have a separate validation set for the tuning.



Summary: Beware of bias when you estimate model performance:

- Training score is often an optimistic estimate of the true performance;
- **The same data should not be used both for training and evaluation.**

▼ Exercise 3

- Experiment with other measures for regularization of the models discussed during the session
 - What is L_1 and L_2 regularization and what is the effect on the weights when changing the `penalty` parameter of the `LogisticRegression` object to `l1`?
 - What is the effect of the `max_features` parameter of the tokenizer?
- Which of the models performs best on the validation data? Discover and import other models from the `sklearn` package ([KNeighborsClassifier](#), [Random Forests](#), ...)
- Can you think of ways to combine predictions by different classifiers?
- A held-out collection of reviews is stored in `data/test.csv`. In the remaining time, try to maximize your model's performance on the CV splits without evaluating on it (until the end of class). How you go about that is completely open (feature engineering, modeling, optimization, etc.).

```
## Answer
# L1 regularization
clf = LogisticRegression(penalty='l1', solver='saga')

train_scores, test_scores = validation_curve(
    clf, X_train, y_train,
    param_name=param_name,
    param_range=param_range, cv=2, n_jobs=-1, scoring="accuracy")

train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
```



```

# Plot parameter VS estimated error
plt.figure(figsize=(8,6))
plt.xlabel(param_name)
plt.ylabel("score")
plt.xlim(min(param_range), max(param_range))
plt.plot(param_range, train_scores_mean, color="blue", label="Training acc.")
plt.fill_between(param_range,
                 train_scores_mean + train_scores_std,
                 train_scores_mean - train_scores_std,
                 alpha=0.2, color="blue")
plt.plot(param_range, test_scores_mean, color="red", label="CV acc.")
plt.fill_between(param_range,
                 test_scores_mean + test_scores_std,
                 test_scores_mean - test_scores_std,
                 alpha=0.2, color="red")
plt.legend(loc="best")

print(test_scores)

# 2500 features (The model is less complex and cannot fit the train data that well. The train accuracy is 90 % i.s.o. 95 %)
tokenizer2500 = CountVectorizer(strip_accents='ascii', ngram_range=(1,2), stop_words='english', min_df=2, max_features=2500)
X = tokenizer2500.fit_transform(data.text)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=69)

clf = LogisticRegression(penalty='l1', solver='saga')

train_scores, test_scores = validation_curve(
    clf, X_train, y_train,
    param_name=param_name,
    param_range=param_range, cv=2, n_jobs=-1, scoring="accuracy")

train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

```

```

# Plot parameter VS estimated error
plt.figure(figsize=(8,6))
plt.xlabel(param_name)
plt.ylabel("score")
plt.xlim(min(param_range), max(param_range))
plt.plot(param_range, train_scores_mean, color="blue", label="Training acc.")
plt.fill_between(param_range,
                 train_scores_mean + train_scores_std,
                 train_scores_mean - train_scores_std,
                 alpha=0.2, color="blue")
plt.plot(param_range, test_scores_mean, color="red", label="CV acc.")
plt.fill_between(param_range,
                 test_scores_mean + test_scores_std,
                 test_scores_mean - test_scores_std,
                 alpha=0.2, color="red")
plt.legend(loc="best")

print(test_scores)

# 1000 features (The model is more complex and can overfit more to the training data. the train accuracy goes to 100 % i.s.o. 95%)
tokenizer10000 = CountVectorizer(strip_accents='ascii', ngram_range=(1,2), stop_words='english', min_df=2, max_features=10000)
X = tokenizer10000.fit_transform(data.text)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=69)

clf = LogisticRegression(penalty='l1', solver='saga')

train_scores, test_scores = validation_curve(
    clf, X_train, y_train,
    param_name=param_name,
    param_range=param_range, cv=2, n_jobs=-1, scoring="accuracy")

train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

```

```
# Plot parameter VS estimated error
plt.figure(figsize=(8,6))
plt.xlabel(param_name)
plt.ylabel("score")
plt.xlim(min(param_range), max(param_range))
plt.plot(param_range, train_scores_mean, color="blue", label="Training acc.")
plt.fill_between(param_range,
                 train_scores_mean + train_scores_std,
                 train_scores_mean - train_scores_std,
                 alpha=0.2, color="blue")
plt.plot(param_range, test_scores_mean, color="red", label="CV acc.")
plt.fill_between(param_range,
                 test_scores_mean + test_scores_std,
                 test_scores_mean - test_scores_std,
                 alpha=0.2, color="red")
plt.legend(loc="best")

print(test_scores)
```

▼ Bonus: Kaggle Competition

- Congratulations! You have made your first sentiment classifier and predicted sentiment for a held-out collection of movie reviews. The ground truth for the held-out data is stored on the online data science platform Kaggle.
- What is the performance of your model? How do you rank among your peers?
- Now try to improve your model by adding new models, vectorizers, additional data,... (talk to us or go looking online to get ideas)
- Think of ways to include knowledge from external sources in your model.
- Submit your predictions to the in-class [Kaggle Competition](#). This competition is private and will be removed. We will keep it open up until the deadline for submitting this notebook (i.e., next Sunday at 11:59PM).
- Feel free to comment on any successful submissions, or provide insights into which methods worked better than others. However, this remains optional.

Keep in mind, the limit of submissions is set at 20 per day!

```

## Answer
# L1 regularization
clf = LogisticRegression(penalty='l1', solver='saga')

train_scores, test_scores = validation_curve(
    clf, X_train, y_train,
    param_name=param_name,
    param_range=param_range, cv=2, n_jobs=-1, scoring="accuracy")

train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Plot parameter VS estimated error
plt.figure(figsize=(8,6))
plt.xlabel(param_name)
plt.ylabel("score")
plt.xlim(min(param_range), max(param_range))
plt.plot(param_range, train_scores_mean, color="blue", label="Training acc.")
plt.fill_between(param_range,
                 train_scores_mean + train_scores_std,
                 train_scores_mean - train_scores_std,
                 alpha=0.2, color="blue")
plt.plot(param_range, test_scores_mean, color="red", label="CV acc.")
plt.fill_between(param_range,
                 test_scores_mean + test_scores_std,
                 test_scores_mean - test_scores_std,
                 alpha=0.2, color="red")
plt.legend(loc="best")

print(test_scores)

# 2500 features (The model is less complex and cannot fit the train data that well. The train accuracy is 90 % i.s.o. 95 %)
tokenizer2500 = CountVectorizer(strip_accents='ascii', ngram_range=(1,2), stop_words='english', min_df=2, max_features=2500)
X = tokenizer2500.fit_transform(data.text)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=69)

```

```

clf = LogisticRegression(penalty='l1', solver='saga')

train_scores, test_scores = validation_curve(
    clf, X_train, y_train,
    param_name=param_name,
    param_range=param_range, cv=2, n_jobs=-1, scoring="accuracy")

train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Plot parameter VS estimated error
plt.figure(figsize=(8,6))
plt.xlabel(param_name)
plt.ylabel("score")
plt.xlim(min(param_range), max(param_range))
plt.plot(param_range, train_scores_mean, color="blue", label="Training acc.")
plt.fill_between(param_range,
                 train_scores_mean + train_scores_std,
                 train_scores_mean - train_scores_std,
                 alpha=0.2, color="blue")
plt.plot(param_range, test_scores_mean, color="red", label="CV acc.")
plt.fill_between(param_range,
                 test_scores_mean + test_scores_std,
                 test_scores_mean - test_scores_std,
                 alpha=0.2, color="red")
plt.legend(loc="best")

print(test_scores)

# 1000 features (The model is more complex and can overfit more to the training data. the train accuracy goes to 100 % i.s.o. 95%)
tokenizer10000 = CountVectorizer(strip_accents='ascii', ngram_range=(1,2), stop_words='english', min_df=2, max_features=10000)
X = tokenizer10000.fit_transform(data.text)

```

X_train X_test y_train y_test train_test_split(X, y, test_size=0.2, random_state=60)

```

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=69)

clf = LogisticRegression(penalty='l1', solver='saga')

train_scores, test_scores = validation_curve(
    clf, X_train, y_train,
    param_name=param_name,
    param_range=param_range, cv=2, n_jobs=-1, scoring="accuracy")

train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Plot parameter VS estimated error
plt.figure(figsize=(8,6))
plt.xlabel(param_name)
plt.ylabel("score")
plt.xlim(min(param_range), max(param_range))
plt.plot(param_range, train_scores_mean, color="blue", label="Training acc.")
plt.fill_between(param_range,
                 train_scores_mean + train_scores_std,
                 train_scores_mean - train_scores_std,
                 alpha=0.2, color="blue")
plt.plot(param_range, test_scores_mean, color="red", label="CV acc.")
plt.fill_between(param_range,
                 test_scores_mean + test_scores_std,
                 test_scores_mean - test_scores_std,
                 alpha=0.2, color="red")
plt.legend(loc="best")

print(test_scores)

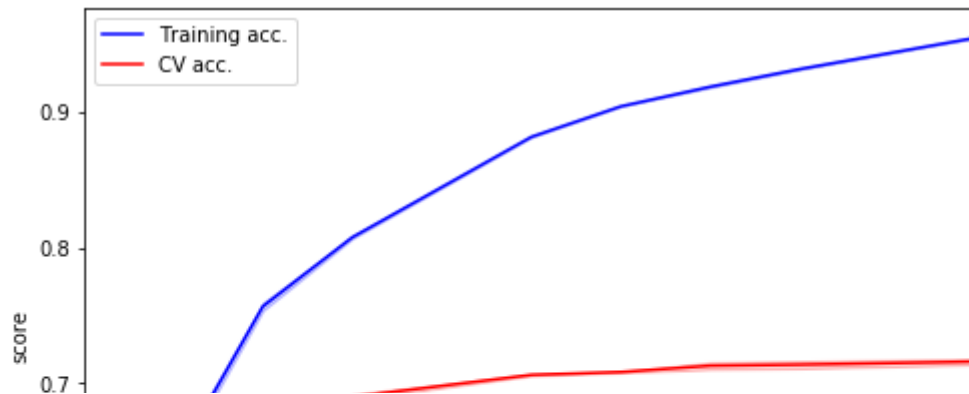
```

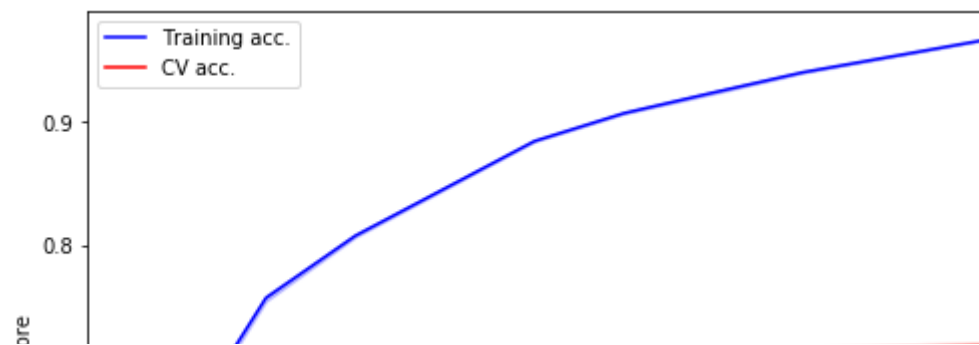
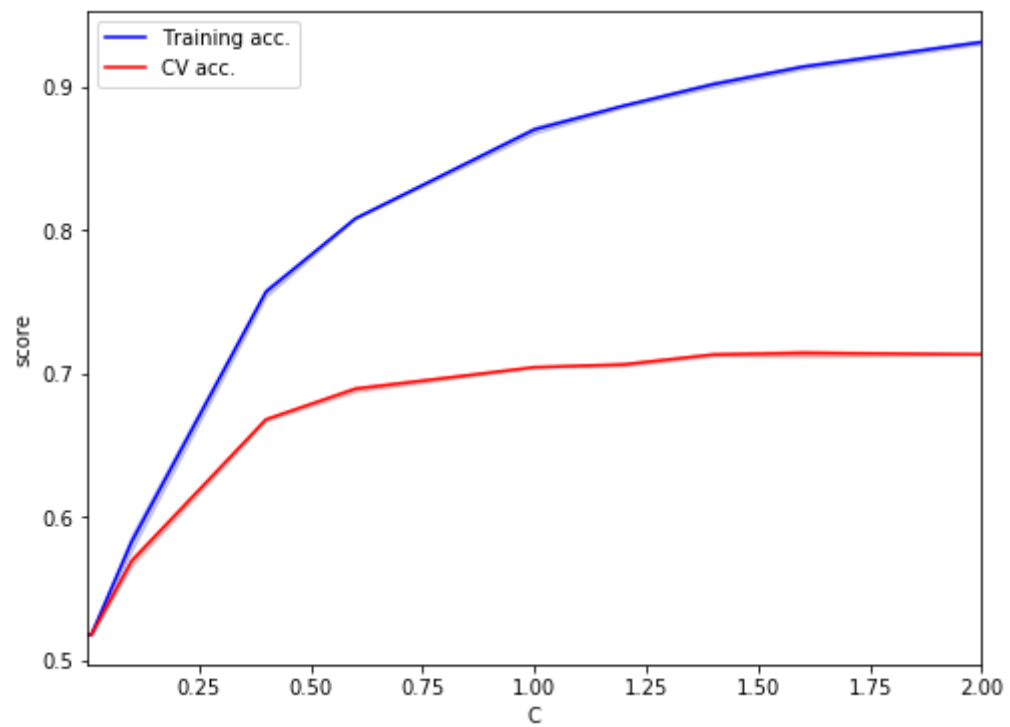
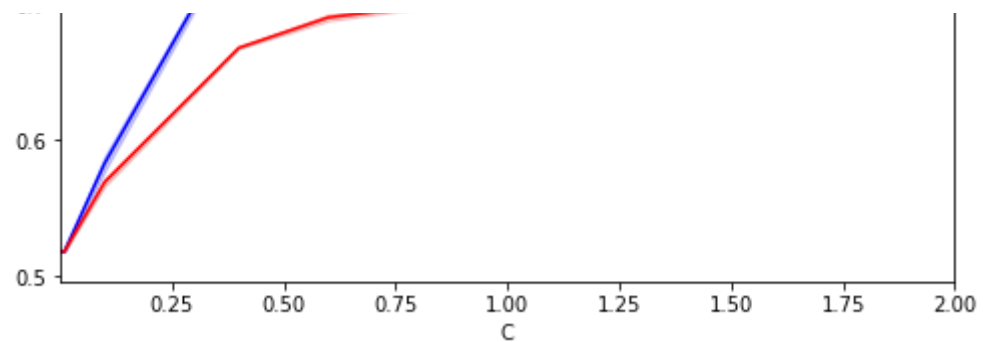


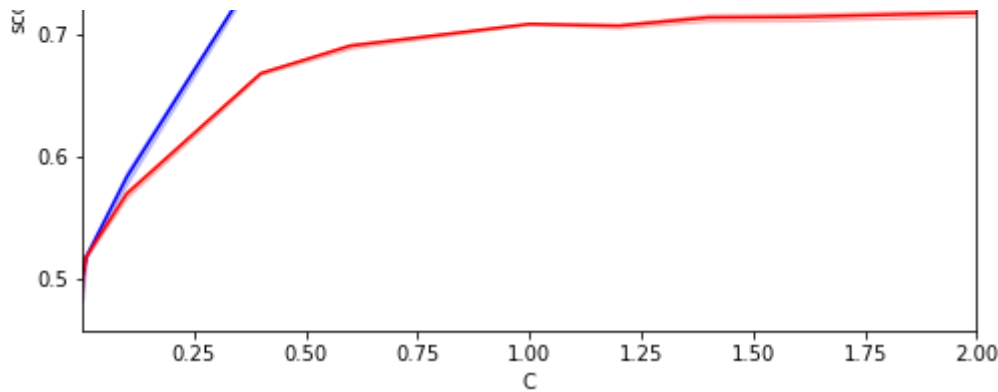
```

[[0.5181264  0.51797176]
 [0.5181264  0.51797176]
 [0.5662496  0.57188703]
 [0.66827077 0.66720154]
 [0.69233237 0.68774069]
 [0.70548604 0.70699615]
 [0.70773179 0.70892169]
 [0.71575233 0.71020539]
 [0.71671479 0.71084724]
 [0.7183189  0.71373556]]
[[0.5181264  0.51797176]
 [0.5181264  0.51797176]
 [0.5662496  0.57220796]
 [0.66859159 0.66720154]
 [0.69104909 0.68741977]
 [0.70452358 0.70410783]
 [0.70516522 0.70731707]
 [0.71414822 0.71213094]
 [0.71639397 0.71181001]
 [0.71318576 0.71341463]]
[[0.5181264  0.48202824]
 [0.5181264  0.51797176]
 [0.5662496  0.57220796]
 [0.66859159 0.66688062]
 [0.69233237 0.68806162]
 [0.70773179 0.70827985]
 [0.70837344 0.70474968]
 [0.71639397 0.71020539]
 [0.71703561 0.71052632]
 [0.72024382 0.71437741]]

```







▼ Bonus: Kaggle Competition

- Congratulations! You have made your first sentiment classifier and predicted sentiment for a held-out collection of movie reviews. The ground truth for the held-out data is stored on the online data science platform Kaggle.
- What is the performance of your model? How do you rank among your peers?
- Now try to improve your model by adding new models, vectorizers, additional data,... (talk to us or go looking online to get ideas)
- Think of ways to include knowledge from external sources in your model.
- Submit your predictions to the in-class [Kaggle Competition](#). This competition is private and will be removed. We will keep it open up until the deadline for submitting this notebook (i.e., next Sunday at 11:59PM).
- Feel free to comment on any successful submissions, or provide insights into which methods worked better than others. However, this remains optional.

Keep in mind, the limit of submissions is set at 20 per day!

Tutorial based on

- [NYU Course - Introduction to Natural Language Understanding - Samuel Bowman](#)
- [Introduction to Scikit Learn - Gilles Louppe](#)