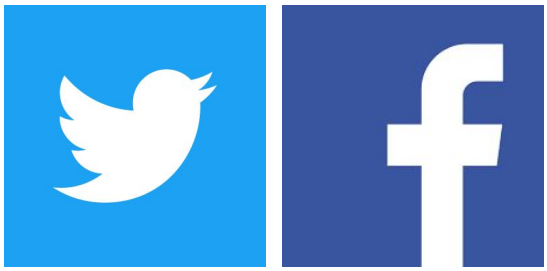
15-110 Hw6 - Social Media Analytics

Hw6 and its checks are organized differently from the other assignments. If you haven't already done so, you should read the **Hw6 General Guide** to understand how this assignment works.

Project Description



In this project, you will analyze a dataset that maps social media messages (tweets and facebook posts) made by politicians over a two-day period to information about the text's bias, partisanship, and message intent. You will use **sentiment analysis** to add additional data on message sentiment and compare politicians across several vectors.

Note: this project uses the **pandas** library to perform data analysis. You will need to install several libraries and learn about pandas dataframes to complete this project.

In the first week, you'll organize the data by adding several additional columns to the dataframe, to include information on name, location, position, sentiment, and hashtags. In the second week, you'll analyze political trends by comparing the data across different combinations of features, by state, region, and hashtag. In the third week, you'll visualize these comparisons using matplotlib.

Data source: https://www.kaggle.com/crowdflower/political-social-media-posts

Click on the following links to read the instructions for each week's assignment:

Check6-1 - due Monday 11/23 at noon EST

Check6-2 - due Friday 12/04 at noon EST

Hw6 - due Friday 12/11 at noon EST

Check6-1 - due Monday 11/23 at noon EST

In the first stage, you will organize the data for the project by installing necessary libraries, reformatting the CSV data into a pandas dataframe, and adding additional columns. This reformatting will make analysis easier in the following stage.

Step 0: Written Assignment **[45pts]**

In addition to completing the steps described below, there is a short written assignment on the week's material. You can find the written assignment on Gradescope, and linked on the Assignments page of the course website.

Step 1: Installations **[0pts]**

The first goal of the first stage is to make sure you are able to install the external libraries: pandas, nltk, matplotlib, and numpy. If you have installed them successfully, you should be able to run the starter code without any errors. After you have successfully installed all the libraries, you should proceed to Step 2.

This project will require a couple of installations for your personal machine. **Unfortunately, some of the required installations are not installed on the CMU cluster machines, so you need to own a laptop or desktop computer to complete this project.**

To install the libraries you need, we recommend that you use the pip tool included in your Python installation. This tool will manage the installation process for you, which is much easier than trying to install a module manually. To use pip, open Pyzo and run the following lines of code in the interpreter:

```
pip install numpy
pip install pandas
pip install matplotlib
pip install nltk
```

If an error message occurs, try googling it to find a solution. TAs can also help debug installation errors via Piazza or in office hours.

You can test whether the modules are correctly installed by running the following commands in your interpreter. If they do not give you an error, you're good to go!

```
import numpy
import pandas
import matplotlib
import nltk
```

Step 2: Learn `pandas` [0pts]

This project will use the pandas library to help do statistical analysis of the social media dataset. We have not taught the pandas library in class, so you'll need to do a bit of additional instruction to learn the new syntax.

Watch the linked video and review the linked code to learn the basics of pandas syntax. If you have questions, feel free to post them on Piazza.

Video: https://www.cs.cmu.edu/~110/hw/hw6_pandas.mp4
Code: https://www.cs.cmu.edu/~110/hw/hw6_pandas_examples.py

If you'd like to learn more, check out the pandas documentation: pandas.pydata.org

Step 3: Generate Dataframes **[5pts]**

Currently, our social media data is stored as a csv file. In order to use this data, we want to convert the csv into a data structure that's easy to use with Python. Luckily, pandas is just the library for the job!

Given a filename of a .csv in the data folder (which you should have downloaded with the starter file), write a function `makeDataFrame(filename)` which takes in a string filename of a CSV file and returns a pandas dataframe. If you aren't sure how to do this, check the videos linked above.

To test this function, run `testMakeDataFrame()`.

Note: to keep the starter file from becoming too crowded, all the tests have been moved to the file `hw6_social_tests.py`, which you can open and read while debugging. The functions in this file are called at the bottom of `hw6_social.py`. **If you change the**

filename of `hw6_social.py`, you will need to change the filename imported by `hw6_social_tests.py` in the first line of the file too.

Step 4: Parse Label **[15pts]**

Step 4 consists of several *helper functions* which are grouped together for a more general purpose. The ultimate goal is to parse the column 'label' (which describes a message's poster) and add four columns to our dataframe: Name, State, Position, and Region.

First, write a function `parseName(fromString)` that takes a string of the form:
`"From: <FirstName> <Lastname> (<Position> from <State>)"`
and returns the name (first and last) in a single string.

For example:
`parseName("From: Stephanie Rosenthal (Professor from Pennsylvania)")`
`-> "Stephanie Rosenthal"`

Next, write a function `parsePosition(fromString)` that takes a string of the form:
`"From: <FirstName> <Lastname> (<Position> from <State>)"`
and returns the position.

`parsePosition("From: Stephanie Rosenthal (Professor from Pennsylvania)") -> "Professor"`

Finally, write a function `parseState(fromString)` that takes a string of the form:
`"From: <FirstName> <Lastname> (<Position> from <State>)"`
and returns the state.

`parseState("From: Stephanie Rosenthal (Professor from Pennsylvania)")`
`-> "Pennsylvania"`

Hint: what comes before or after each of the pieces of information?

To test these functions, run `testParseName()`, `testParsePosition()`, and `testParseState()`.

Step 5: Parse Message **[15pts]**

Step 5 takes on a more complicated parsing task: given a social media message, what hashtags are included inside that message? Write the message `findHashtags(message)` which takes a string (a social media message) and returns a list of strings- the hashtags included in that message, in order of appearance.

To do this, you'll need to find all the starting places of hashtags (designed by the '#' character), and iterate over the characters that follow that starting index until you reach an 'end' character. We've provided a list of end characters in the global variable `endChars`; just check for whether each character is in that list, and stop when you reach one of them.

For example:
```
findHashtags("I am so #excited to watch #TheMandalorian! #starwars")
-> [ "#excited", "#TheMandalorian", "#starwars" ]
```

To test these functions, run `testFindHashtags()`.

Step 6: Find Region **[5pts]**

Write a function `getRegionFromState(stateDf, state)` that takes, `stateDf`, a dataframe, and a string of a state name to search for. and returns the corresponding region. `stateDf` has rows where each state is in the column `'state'` and its corresponding region of the US, like Northeast or South, is in the `'region'` column.

Often, when looking up values in a dataframe, you know the value of one column, and you want to look up the associated value of a different column (like knowing a person's name and wanting to look up their phone number). To do this, use the code pattern:

```
df.loc[df['known column name'] == 'known value to match', 'column name
to return']
```

Use this code to get the row associated with the state in the state dataframe by using the correct column name and state value. Then, get the value of this cell in the dataframe by using `row.values[0]`, and return this value.

To test this function, run `testGetRegionFromState()`.

Step 7: Add Columns to Dataframe **[15pts]:**

Write a function `addColumns(data, stateDf)` that takes in a dataframe `data` and a dataframe `stateDf`, and destructively adds five new columns to `data` based on the functions you wrote above. Return `None` at the end of the function.

1. In order to add columns to a dataframe, you must make lists of all the values to add (one for each new column). Create five new empty lists for names, positions, states, regions, and hashtags of the posters.
2. Uses `iterrows()` to iterate through each index and row in data. For each row, the function should:
    a. get the value in the column 'label' (the code looks the same as a dictionary with key 'label')
    b. call `parseName`, `parsePosition`, and `parseState` on that value.
    c. call `getRegionFromState` with `stateDf` and the state you parsed to get its region.
    d. get the value in the column 'text'
    e. call `findHashtags` on that value
    f. append each of the values to their respective list.
3. After the end of the for loop, set `data['name']`, `data['position']`, `data['state']`, `data['region']`, and `data['hashtags']` to the respective lists.
4. Return `None`

To test this function, run `testAddColumns()`.

You have now finished the first stage of the project. To see the dataframe you've created, run `runWeek1()`.

Check6-2 - due Friday 12/04 at noon EST

In this stage, you will dive deeper into actually analyzing the social media messages in the CSV. You'll group data by state and region to identify trends in types of messages, and you'll analyze different hashtags to determine which are most popular and what common sentiments are for different hashtags.

Before you start this second stage, go to the bottom of the starter file and uncomment the four test lines associated with Week 2.

Step 0: Written Assignment **[45pts]**

In addition to completing the steps described below, there is a short written assignment on the week's material. You can find the written assignment on Gradescope, and linked on the Assignments page of the course website.

Step 1: Identify Message Sentiment **[5pts]**

Edit the function `findSentiment(classifier, message)` to return `"positive"` if the classifier predicts the message is positive, `"negative"` if it is negative, and `"neutral"` if it is neutral. The first line of the code already runs a classifier on the message and receives a float score. If that score is less than -0.1, return `"negative"`; if greater than 0.1, return `"positive"`; and otherwise return `"neutral"`.

To test this function, run `testFindSentiment()`.

Note: you might be curious about how the sentiment classifier works. Sentiment Analysis infers whether the statement has a positive, negative, or neutral connotation from the words in the sentence. This is a really hard problem. Think about the statement "That's great!" You could use it to mean it really is great, or you can use it in response to something that is really horrible. The sentiment classifier is trained using thousands of example sentences in many different contexts in order to help it understand whether particular words or phrases are generally positive, negative, neutral (or could be used both positively and negatively). This classifier generates a number for the text given to it, where negative numbers are associated with negative sentiment, and positive numbers are associated with positive sentiment.

Step 2: Add a Sentiment Column **[5pts]**

Edit the function `addSentimentColumn(data)` to perform a similar function to `addColumns`, but with sentiment.

Like last time, you should create a new empty list of sentiments. Then iterate through each index and row of the dataframe `data`. For each row, get the value of the `'text'` column which represents the message, pass the message and the classifier to `findSentiment`, and append the result to the sentiments list. After the loop is finished, it should add the column `'sentiment'` to `data` (using the sentiments list) and return `None`.

To test this function, run `testAddSentimentColumn()`.

Step 3: Organize Data to Compare by State **[5pts]**

Next, we will write functions that analyze the dataframe and create dictionaries mapping states or regions to data about that state or region. We'll use these in Week 3 to create graphs that show trends in the data.

First, write the function `getDataCountByState(data, colName, dataToCount)` that takes the dataframe, a column name (like 'sentiment' or 'message'), and a column value (like 'negative' or 'attack'), and returns a dictionary that maps state names to the number of message that had that value in the listed column. For example, `getDataCountByState(data, "sentiment", "negative")` would return a dictionary mapping states to the number of messages from that state that had negative sentiment.

To solve this problem, recall how we can generate dictionaries that map values to counts. Do the same thing here, but use `row[colName]` and `row['state']` to access your values.

This function should also handle one special case. If both `colName` and `dataToCount` are empty strings (""), just return a dictionary mapping each state to the number of data points it has.

To test this function, run `testGetDataCountByState()`. We'll generate count dictionaries to count negative sentiments, attack messages, and partisan bias.

Step 4: Organize Data to Compare by Region **[10pts]**

Next, we want to organize data by region to find how many of *every* possible value occurs within a given column. Write a function `getDataForRegion(data, colName)` that takes a dataframe and a column name (a string) and returns a nested dictionary. The keys of the outer dictionary should be regions, which each map to an inner dictionary. The keys of the inner dictionary should be the different values that occur in the column for that region. For example, if we called `getDataForRegion(data, "message")`, the inner dictionaries may have the keys 'attack', 'constituency', 'information', and more! Each message type should map to the number of times it occurred in that region in the dataset.

Setting up a nested dictionary isn't too different from setting up a normal dictionary. The only big change is, if `d` is your outer dictionary, you'll need to set `d[key] = { }`. Then you can add new key-value pairs directly to `d[key]`.

To test this function, run `testGetDataForRegion()`.

Step 5: Get Hashtag Counts **[5pts]**

The next three steps will build towards one central goal: determining what the most popular hashtags in the dataset are, and whether they are used in an overall positive or negative sentiment.

First, write the function `getHashtagRates(data)` which takes the dataframe and returns a dictionary mapping hashtags (strings) to counts of how many times they're used in the entire dataset (integers).

To do this, you should loop over the dataframe and get the list in `row["hashtags"]` for each data point. Iterate over the hashtags in the list (if there are any) to update the number of times each hashtag has been seen in the overall dictionary.

To test this function, run `testGetHashtagRates()`.

Step 6: Find Most Common Hashtags **[15pts]**

Next, write the function `mostCommonHashtags(hashtags, count)` which takes the hashtag dictionary generated in the previous function and the number of top hashtags to find (integer) and returns a new dictionary mapping just the top-count hashtags to how

often they each occur. For example, `mostCommonHashtags(hashtags, 5)` would return a dictionary of five key-value pairs, the five most popular hashtags in the dataset.

One way to do this is to create an empty dictionary which will hold the known most-common hashtags. Then repeatedly search for the most common hashtag that is not already a key in that dictionary. Once you've gone through all the hashtags and found the one with the highest appearance rate, add it to the dictionary. Continue the process until the length of the dictionary is equal to `count`.

To test this function, run `testMostCommonHashtags()`.

Step 7: Get a Hashtag's Sentiment Score **[10pts]**

Finally, write the function `getHashtagSentiment(data, hashtag)` which takes the dataset and a hashtag (string) and returns a 'sentiment score' (float) for that hashtag.

You will need to iterate over the dataset to find every message (from `row['text']`) that contains that hashtag. For each message, get its sentiment value (in `row['sentiment']`). If it is 'positive', that maps to 1; 'negative' maps to -1; and 'neutral' maps to 0.

(How can we tell if a hashtag is in a message? Just call `findHashtags` on that message and check if the hashtag you're looking for is in that list!).

To get the sentiment score, average all the individual hashtag-message scores together, and divide them by the total number of messages that contained the hashtag. If this number is positive and close to 1, the messages were generally very positive; if it is negative and close to -1, they were generally very negative. A score close to 0 might indicate a neutral hashtag, or one which has a mix of positive and negative sentiments.

To test this function, run `testGetHashtagSentiment()`.

Now that you've written all the analysis functions, you can run `runWeek2()` to check out your results!

Hw6 - due Friday 12/11 at noon EST

In the final part of this project, you will visualize all of the data you have analyzed in the previous stages. To do this, you will use matplotlib to create graphs and charts.

Before you start this last stage, go to the bottom of the starter file and uncomment the two test lines associated with Week 3.

Step 0-A: Complete Check-in 1 **[20pts]**

If you got a perfect score on Hw6 Check-in 1 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check-in 1 and use it to update your Check-in 1 code. This is your chance to implement any features you might have missed before, and fix any code that isn't working.

Step 0-B: Complete Check-in 2 **[20pts]**

If you got a perfect score on Hw6 Check-in 2 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check-in 2 and use it to update your Check-in 2 code. This is your chance to implement any features you might have missed before, and fix any code that isn't working.

Step 1: Review Provided Code **[0pts]**

Drawing graphs with matplotlib requires a lot of setup code, and we want you to draw quite a few graphs, so we've provided a few functions for you to use, to simplify things. You will write some functions that call the functions we've provided.

You should definitely know what each of the following functions do, and we recommend that you look over their code at the bottom of the file quickly, to get a sense of how they work.

- **`sideBySideBarPlots:`** generates two bar charts side-by-side, for easy comparison. The x values come from xLabels, the y values come from valuesList, and labelList is used as a key.
- **scatterPlot:** generates a scatterplot of numerical data. xValues and yValues provide the position of each data point, and labels provide a descriptive term.

Step 2: Graph State Count Dictionaries **[15pts]**

Write a function `graphStateCounts(stateCounts, title)` which takes a dictionary mapping states to some kind of count (integer), and a title (string), and displays a bar chart of the data, returning nothing.

To make a bar chart, you'll want to use the matplotlib function `plt.bar()`. Make a list of the dictionary's keys and another list of the dictionary's values - the keys (states) will be the labels of the graph, and the values (numbers) will be used to determine how tall the bars should be.

If you use the default settings, the labels on the graph will all overlap. Use the function call `plt.xticks(rotation='vertical')` to change the label orientation to be vertical instead.

Check the Data Analysis II slides for examples of how to make a bar plot and add the title to the chart!

To test this function, check that the first graph produced by `runWeek3()` has 50 bars, that the largest bars are Texas and California, and that the smallest bars belong to Montana, Maine, and Hawaii. The second graph produced (narrowed down to only Facebook data) should have only 49 bars - South Dakota should be missing, as it only has Twitter data.

Step 3: Graph Top N States **[10pts]:**

Write a function `graphTopNStates(stateCounts, stateFeatureCounts, n, title)` which takes a dictionary mapping states to total number of messages, a dictionary mapping states to the number of messages that match a certain feature, an integer `n` for the number of top states to select, and a title, and displays a graph (returning nothing). The function should find the top `n` states that have the highest **rates** of a feature occurring overall to provide to the graph as data.

Note that a feature's rate is different from its count! If we just looked at counts, the biggest states (Texas and California) would always end up at the top. Instead, you should divide each state's feature count by the total number of messages (from `stateCounts`) to get the frequency rate for that feature.

To find the top `n` states, use a similar approach to what you did in `mostCommonHashtags` before. Just make sure to check rates instead of counts!

Once you've created the dictionary of the top states, call `graphStateCounts` with it to graph the top `n` states. Don't forget to include the provided title!

To test this function, check the third, fourth, and fifth graphs produced by `runWeek3()`. The third (graphing attack rates) should have five bars: Louisiana (which should be at around 0.14), Kentucky, Georgia, South Dakota, and Indiana. The fourth (graphing policy rates) should have five bars: Wyoming (which should be around 0.6), Maine, Alaska, Mississippi, and Michigan. The fifth (graphing nationally-focused messages) should have five bars: Vermont (which should be close to 1.0), Maryland, Connecticut, Mississippi, and Utah.

Step 4: Graph a Region Comparison **[20pts]:**

Write the function `graphRegionComparison(regionDicts, title)` which takes a nested region dictionary and a title and displays a graph (returning nothing). The function should take the region nested dictionary and create new lists of the data within that will be used to produce a side-by-side bar chart that can compare all four regions at once. You will need to make three lists: a list of **feature names**, a list of **region names**, and a list of **region-feature lists**.

To generate the list of feature names, iterate over the regions in the region dictionary, then iterate over the features in each region's inner dictionary. Construct a feature list over time, adding any features you find that are not yet in the list.

To generate the list of region names, simply iterate over the keys of the region dictionary and add those keys to a list.

To generate the list of region-feature values, iterate over the regions of the region dictionary and create a new temporary list for each region. This temporary list will hold the counts of how often each feature occurred in the region (with the value at index i corresponding to the feature at index i of the feature list). To produce this list, iterate

over the **feature list**. If the feature you're looking at is in the region's inner dictionary, add the count to the list; if it isn't, add 0 (as the feature never showed up). When the temporary list has been fully constructed, add it to the main region-feature list (which is a 2D list).

Once you have all three lists, call the function `sideBySideBarPlots(xLabels, labelList, valueLists, title)` to produce the side-by-side graph. The feature list should be the `xLabels`, the region list provides the `labelList`, and the region-feature list should be the `valueLists`. Don't forget to include the title as well!

To test this function, check the sixth and seventh graphs produced by `runWeek3()`. The sixth chart (message types by region) should show 9 clusters of four bars each, with the South region containing a much larger number of certain types of messages (policy, attack, personal, support, information, media) than the other three regions. The seventh chart (position by region) should show two groups of 4 bars, and should demonstrate that there are many more messages by Representatives than Senators (especially in the South).

Step 5: Graph Hashtag Sentiments to Frequencies **[15pts]:**

Finally, write the function `graphHashtagSentimentByFrequency(df)`. This function will use the functions you've built so far to generate a graph that compares hashtags' frequency (how often they occurred) to their average sentiment score.

Start by generating a dictionary mapping hashtags to their counts (with `getHashtagRates`), then get the the top 50 most common hashtags (with `mostCommonHashtags`).

You'll need to generate three lists - a list of hashtags, a list of frequencies, and a list of sentiment scores. You can build up these lists simultaneously by looping over the hashtags in the most-common dictionary (the dictionary gives you the hashtag and count, and you can call `getHashtagSentiment` to get the sentiment).

Once you have the three lists, call the function `scatterPlot(xValues, yValues, labels, title)` to draw the chart. (It may take a while, since the functions are not particularly efficient, but should not take more than a minute). You should use the frequencies for the x values, the sentiment for the y values, and the hashtags for the labels. You should also write your own title to describe the chart.

To test this function, check the eighth (and final) chart produced by `runWeek3()`. The scatter plot should have one hashtag all the way to the right (above 60 on the x axis) and the rest clustered between 0 and 30 on the x axis. The general trend should be that the messages are neutral to positive - few messages should be below -0.25. If you want to investigate the left side more closely, use the zoom feature built into matplotlib by clicking on the magnifying glass icon and drawing a rectangle around the part of the chart you want to zoom.

Optional final step: you'll notice that some data point labels overlap, so that you can't read them. This happens when two or more data points share the same frequency x sentiment pair. If you want to fix this, you can modify the `graphHashtagSentimentByFrequency` function to **merge** data points that share the same frequency x sentiment, by combining hashtags together into a single string. If only one data point/label is drawn, you'll be able to read all the hashtags easily.

Congratulations- now you're done! Feel free to try running some of the graphing functions or analysis functions on different aspects of the dataset, to investigate different political trends.