Modeling

Project 2: Modeling due 11:59 PM, Thursday, March 21, 2019

Introduction

Statistical models of text are one way to quantify how similar one piece of text is to another. Such models were used as evidence that the book *The Cuckoo's Calling* was written by J. K. Rowling (using the name Robert Galbraith) in the summer of 2013.

The comparative analysis of Rowling's works used surprisingly simple techniques to model the author's *style*. Possible "style signatures" could include (but are not limited to):

- the distribution of word lengths in a document – calculating the frequency of words of length one, length two, length three, etc. This is one of the metrics that was used in the Rowling case.
- the distribution of words used by an author – calculating the frequency of each unique word.
- the distribution of word *stems* used (e.g., "spam" and "spamming" would have the same stem) by an author – calculating the frequency of each unique stem.
- the distribution of sentence lengths used by an author – calculating the frequency of sentences of length one, length two, length three, etc.

In Modeling, you will create a text analysis model based on statistics to model, analyze, and score the similarity of text samples. By the end of the project, you will have developed a sophisticated tool that will allow you to identify a particular author or style of writing.

Installation and Handin

Project setup. For each project, there may be support files that you will need to complete the assignment. These can be copied to your home directory by using the `cs4_install` command in a CIT Terminal window. For this project, type the commands:

```
cs4_install modeling
```

There should now be a `modeling` folder within your `projects` directory. Using Terminal, you can move into the folders with the cd command:

```
cd ~/course/cs0040/projects/modeling
```

Project hand-in. When you are ready to submit the files for Project 2: Modeling, run:

```
cs4_handin modeling
```

from a CIT Terminal window from your `~/course/cs0040/projects/modeling` directory. The entire contents of `~/course/cs0040/projects/modeling` will be handed in. Check for a confirmation email to ensure that your assignment was correctly submitted using the `cs4_handin` command.

You can resubmit this assignment using the `cs4_handin` command at any time, but only your most recent submission with be graded.

Specification

In Modeling, you will develop a `TextModel` class that is able to perform statistical operations on a body of text. At a minimum, your final model should include the following five features:

- word frequencies
- word lengths
- stem frequencies
- sentence lengths
- **one other feature of your choice**. You might choose one of the features used in the analysis that revealed Rowling's authorship of *The Cuckoo's Calling*. Or you could choose something else entirely (e.g., something based on punctuation, sentiment analysis) – anything that you can compute/count using Python!

Checklist

While this project requires you to implement a lot of functions, a good sense of organization and pre-planning will make development substantially easier. To help, we've divided it up into five phases to help organize your workflow. We've first provided a checklist of the required functions you need to write (and you are welcome to write other helper functions for code cleanliness and readability).

Note that this checklist is intentionally underspecified in some places to keep it concise—after the checklist, we've provided high-level descriptions to help you in your implementation. **We highly recommend that you follow the checklist and the high-level descriptions in order to avoid major debugging issues later on.**

Phase 1: Setting Up The Model

In `model.py`, implement the following methods:

- ❏ **`__init__(self, model_name)`**: constructs a new TextModel object and initializes the following attributes:
  - ❏ `name` − a string that is a label for this text model
  - ❏ `words` − a dictionary that records the number of times each word appears in the text
  - ❏ `word_lengths` − a dictionary that records the number of times each word *length* appears
  - ❏ `stems` – a dictionary that records the number of times each word stem appears in the text
  - ❏ `sentence_lengths` – a dictionary that records the number of times each sentence length appears
  - ❏ another dictionary representing your extra chosen text analysis feature
- ❏ **`__repr__(self)`**: returns a string that includes the name of the model as well as the sizes of the dictionaries for each feature of the text

Phase 2: Parsing the Text

In `model.py`, implement the following functions and methods:

- ❏ **`clean_text(txt)`**: consumes a string `txt` and returns a list of the "cleaned" versions of the words in `txt`
- ❏ **`stem(word)`:** accepts a string as a parameter, returns the stem of `word`. The stem of a word is the root part of the word, which excludes any prefixes and suffixes.

    - ❏ This function does not have to work perfectly for all possible words and stems. For full credit, you should implement seven distinct cases and test them on many different words.
- ❏ **`add_string(self, s)`**: given a string `s`, updates the feature dictionaries initialized in the `TextModel` constructor
- ❏ **`add_file(self, file_name)`**: given the name of a file `file_name`, adds all of the text in the file identified by `file_name` to the `TextModel`

Phase 3: Model Persistence

In `model.py`, implement the following methods:

- ❏ **`save_model(self)`:** saves the `TextModel` object self by writing its various feature dictionaries to files
- ❏ **`read_model(self)`:** reads the stored dictionaries for the called `TextModel` object from their files (with the assumption those files have been generated by `save_model`) and assigns them to the attributes of the called `TextModel`

Phase 4: Performing Analysis

In `model.py`, implement the following functions and methods:

- ❏ **`compare_dictionaries(d1, d2)`:** consumes two feature dictionaries d1 and d2 as and computes and returns their log similarity score using the Naive Bayes scoring algorithm
- ❏ **`similarity_scores(self, other)`**: consumes another `TextModel` and computes and returns a *list* of log similarity scores measuring the similarity of self and other, one score for each type of feature
- ❏ **`classify(self, source1, source2)`**: consumes two other "source" `TextModel` objects (`source1` and `source2`) and returns which of those two `TextModels` is the more likely source of the called TextModel self
- ❏ **`compare(self, source1, source2)`**: identical to `classify`, but instead of returning a `TextModel`, prints out a "classification report" that contains the lists of similarity scores for `source1` and `source2` and some descriptive statement on which source the `TextModel` self most likely originated from

Phase 5: Experiment

You don't have to implement any functions for this phase of the project, but you still need to:

- Curate two bodies of text from which to create two "source" models
- Choose four different documents (not used in the creation of your source models) that you would be interested in classifying against your source models

- Edit the supplied run_experiements function in the stencil code to:
    - Create a `TextModel` instance for each of your two chosen bodies of text
    - Classify your four documents using each of the two new `TextModels`
    - Output meaningful data using the `compare` function from Phase 4
- Write a `technical_report.txt` file (approximately five paragraphs) containing answers to the following questions:
    - What text features did you use? What was the additional feature you chose?
    - What approach(es) did you use for your **classify** method? Why did you choose this approach?
    - Which source bodies of text did you choose? Which new texts/bodies of text did you choose to compare against the sources? Why did you choose those sources and documents?
    - What were the results of your comparisons?
    - How well do you think your text classification program works? How could it be improved?

Phase 1: Setting up the Model

In this part, you'll be editing the provided `TextModel` class in `model.py` to add functions that allow you to read in text files and extract meaningful output.

__init__(self, model_name)

If you're having trouble setting up your constructor method, we recommend you refer back to the lecture slides and your Markstrings code and observe how the `__init__` methods are defined there. Remember to initialize the following attributes listed in the Specification Checklist above:

1. `name` − a string that is a label for this text model, such as `'JKRowling'` or `'Shakespeare'`. This will be used in the filenames for saving and retrieving the model. Use the `model_name` that is passed in as a parameter.
2. `words` − a dictionary that records the number of times each word appears in the text.
3. `word_lengths` − a dictionary that records the number of times each word *length* appears.
4. `stems` – a dictionary that records the number of times each word stem appears in the text.
5. `sentence_lengths` – a dictionary that records the number of times each sentence length appears.
6. **Another dictionary representing your chosen extra text analysis feature**. You might choose one of the features used in the analysis that revealed Rowling's authorship of The Cuckoo's Calling. Or you could choose something else entirely (e.g., something

based on punctuation, sentiment analysis) – anything that you can compute/count using Python!

__repr__(self)

Write a method `__repr__(self)` that returns a string that includes the name of the model as well as the sizes of the dictionaries for each feature of the text.

For example, if a `TextModel` named "J. K. Rowling" has been set-up, the return value of this method may look like:

```
>>> some_model = TextModel("J. K. Rowling")
# ... a bunch of operations take place here
>>> print(some_model)
text model name: J. K. Rowling
  number of words: 2103
  number of word lengths: 17
  number of stems: 12
  number of sentence lengths: 23
```

The numbers shown above are just for demonstration purposes, and should be the length of each of the dictionaries you initialized in `__init__`. Initially, these dictionaries should be empty. However, in Phase 2, we'll develop methods to add meaningful information to these dictionaries.

Note: Remember that the `__repr__` method should create a single string and return it. You should not use the print function in this method. Additionally, since the returned string should have multiple lines, you will need to add in the newline character (`'\n'`). The stencil contains a stub method. Fill it out.

Checkpoint

At this point, you will be able to test that your instance variables were initialized correctly in `__init__` and your implementation of `__repr__` by calling print on a `TextModel` instance. For example:

```
>>> model = TextModel('A. Poor Righter')
>>> print(model)
text model name: A. Poor Righter
  number of words: 0
  number of word lengths: 0
```

```
  number of stems: 0
  number of sentence lengths: 0
```

Phase 2: Parsing the Text

Now that you have a base `TextModel` class and a way to read text from files, you will implement several helper functions to perform string parsing operations and accumulate statistics on a given body of text. Then, you will extend the `TextModel` class with methods that apply these functions so that you can add a body of text to a `TextModel` and automatically update the feature dictionaries you setup in the `TextModel` constructor.

Functions Outside of TextModel

Implement the following functions **outside of the `TextModel` class:**

clean_text(txt)

Write a function named `clean_text(txt)` that takes a body of text `s` as a parameter, and returns a list containing the words in `txt` with all punctuation and special characters removed. Duplicates should remain in the list.

The extent to which you clean the text is up to you. While you should at least remove common punctuation symbols, you can decide how many other symbols you want to remove. (You may find it helpful to use the Python string method `replace`.) You may also find it helpful to perform other operations on the string, such as converting all of the characters to lowercase (which you can do using the Python string method `lower`). For example:

```
>>> clean_text("This has LOTS of punctuation--let's clean this!")
["this", "has", "lots", "of", "punctuation", "lets", "clean", "this"]
```

Note: You should not use recursion for this function, since for large files you will run out of memory from too many recursive method calls.

stem(word)

Write a function named `stem(word)` that accepts a string as a parameter. The function should then return the *stem* of `word`. The stem of a word is the root part of the word, which excludes any prefixes and suffixes. For example:

```
>>> stem('party')
```

```
'parti'
>>> stem('parties')
'parti'
>>> stem('love')
'lov'
>>> stem('loving')
'lov'
```

Notes:

- This problem is open-ended, and the number of different cases that stem is able to handle is up to you. Please ask on Piazza for clarifications on expected output. For full credit, your function should handle at least seven distinct cases, each of which works on multiple words.
- The stem of a word is not necessarily a word itself!
- This function does not have to work perfectly for all possible words and stems. Instead, you should define a multitude of cases for stems that work for many words, as we will discuss in lecture. For full credit, you should implement seven distinct cases and test them on many different words.

Methods for TextModel

Implement the following two functions **as `TextModel` methods**:

add_string(self, s)

Write a method `add_string(self, s)` that adds a string of text s to the model by augmenting the feature dictionaries defined in the constructor. It should *not* explicitly return a value.

Don't forget to also update the extra dictionary you initialized in `__init__` representing the extra text analysis feature you chose in Part 1.

We **highly recommend** that you develop helper functions to perform each of the text processing operations you need to update the `TextModel`'s feature dictionaries. **You are also expected to apply the `clean_text` and stem methods as specified above.** See the Checkpoint below on the expected result of `add_string`.

add_file(self, file_name)

Write a method `add_file(self, file_name)` that adds all of the text in the file identified by filename to the model by augmenting the feature dictionaries defined in the constructor. It should *not* explicitly return a value.

You can use the `open` method to open a file and read its contents. When you open the file for reading, you should specify two additional arguments as follows:

```
f = open(<some file name>, 'r', encoding='utf8', errors='ignore')
```

These *encoding* and *errors* arguments should allow Python to handle special characters (e.g., “smart quotes”) that may be present in your text files.

Checkpoint

At this point, you should be able to add bodies of text to `TextModel` instances using your add_string method, and verify that the statistic dictionaries are updated correctly by printing the `TextModel` (thereby invoking the `__repr__` method you created in Part 1) and accessing the instance variables.

```
>>> model = TextModel('A. Poor Righter')
>>> model.add_string("The partiers love the pizza party.")
>>> print(model)
text model name: A. Poor Righter
  number of words: 5
  number of word lengths: 4
  number of stems: 4
  number of sentence lengths: 1
>>> model.words
{'party': 1, 'partiers': 1, 'pizza': 1, 'love': 1, 'the': 2}
>>> model.word_lengths
{8: 1, 3: 2, 4: 1, 5: 2}
>>> model.stems
{'parti': 2, 'the': 2, 'pizza': 1, 'lov': 1}
>>> model.sentence_lengths
{6: 1}
```

Phase 3: Model Persistence

Creating a text model can require a lot of computational power and time. Therefore, once we have created a model, we want to be able to save it for later use. The easiest way to do this is to write each of the feature dictionaries to a different file so that we can read them back in at a later time. In this part of the project, you will add methods to your `TextModel` class that allow you to save and retrieve a text model in this way.

save_model(self)

Write a method `save_model(self)` that saves the `TextModel` object `self` by writing its various feature dictionaries to files. There will be one file written for each feature dictionary.

In order to identify which model and dictionary is stored in a given file, you should use the `name` attribute concatenated with the name of the feature dictionary. For example, if `name` is `'JKR'` (for J. K. Rowling), then we would suggest using the filenames:

- `'JKR_words'`
- `'JKR_word_lengths'`
- `'JKR_stems'`
- `'JKR_sentence_lengths'`

In general, the filenames are `self.name + '_' +` *`name_of_dictionary`*. Taking this approach will ensure that you don't overwrite one model's dictionary files when you go to save another model.

Below, we've provided a code snippet of a function that writes a dictionary to a file:

```
def sample_file_write(filename):
    """
    A function that demonstrates how to write a Python dictionary to an
    easily-readable file.
    """
    d = {'test': 1, 'foo': 42}   # Create a sample dictionary.
    f = open(filename, 'w')      # Open file for writing.
    f.write(str(d))              # Writes the dictionary to the file.
    f.close()                    # Close the file.
```

Notice that the file is opened for *writing* by using a second parameter of `'w'` in the `open` function call. In addition, we write to the file by using the file handle's `write` method on a string representation of the dictionary.

read_model(self)

Write a method `read_model(self)` that reads the stored dictionaries for the called `TextModel` object from their files and assigns them to the attributes of the called `TextModel`. This is the complementary method to `save_model`, and you should assume that the necessary files have filenames that follow the naming scheme used in `save_model`.

Remember that you can use the `dict` and `eval` functions to convert a string that represents a dictionary to an actual dictionary object.

Below, we've provided a code snippet of a function that reads a dictionary from a file (which was put into the file by the `sample_file_write` method above). This function converts this string (which is a string that *looks like* a dictionary) to an actual dictionary object. The conversion is performed using a combination of two built-in functions: `dict`, the constructor for dictionary objects; and `eval`, which evaluates a string as if it were an expression:

```
def sample_file_read(filename):
    """
    A function that demonstrates how to read a Python dictionary from
    a file.
    """
    f = open(filename, 'r')    # Open for reading.
    d_str = f.read()           # Read string that represents a dict.
    f.close()
    d = dict(eval(d_str))      # Convert the string to a dictionary.
    print("Inside the newly-read dictionary, d, we have:")
    print(d)
```

Checkpoint

At this point, you should be able to add bodies of text to `TextModel` instances using your `add_string` method, save the generated statistic dictionaries using `save_model`, and load them back into a new `TextModel` instance using the `read_model` method. For example:

```
# Create a model for a simple text, and save the resulting model.
>>> model = TextModel('A. Poor Righter')
>>> model.add_string("The partiers love the pizza party.")
>>> model.save_model()
# Create a new TextModel object with the same name as the original one,
# and assign it to a new variable.
>>> model2 = TextModel('A. Poor Righter')
# Read the dictionaries that were saved for the original model,
# and use them as the dictionaries of `model2`.
>>> model2.read_model()
>>> print(model2)
  text model name: A. Poor Righter
    number of words: 5
```

```
    number of word lengths: 4
>>> model2.words
{'party': 1, 'partiers': 1, 'pizza': 1, 'love': 1, 'the': 2}
>>> model2.word_lengths
{8: 1, 3: 2, 4: 1, 5: 2}
```

Phase 4: Performing Analysis

In this part of the project, you will first implement the Naive Bayes scoring algorithm covered in lecture that will allow you to compare bodies of text. This algorithm will produce a numeric *similarity score* that measures how similar one body of text is to another, based on one type of feature (e.g., word lengths). You will then compute scores of this type for all five of the features, and use them to classify a piece of text as being more likely to come from one source than another.

Naive Bayes Example

To illustrate how the Bayesian scoring algorithm works, let's assume that the only features we care about are the individual word counts found in a text.

As you have already done in your `TextModel` class, we can use a Python dictionary to model the word counts. The dictionary's keys are words, and the value for a given word is the number of times that it appears in the text (i.e. its count).

For example, let's assume that we have two text documents:

- a *source* text (which we are pretending was written by Shakespeare!) that has the following dictionary:
  `shakespeare_dict = {'love': 50, 'spell': 8, 'thou': 42}`
- This document has 100 words in all: 50 occurrences of the word "love," 8 of "spell," and 42 of "thou."
- a *mystery* text (author unknown) whose dictionary looks like this:
  `mystery_dict = {'love': 3, 'thou': 1, 'potter': 2, 'spam': 4}`
- This document has 10 words in all: three occurrences of "love," one of "thou," two of "potter," and four of "spam."

The Bayesian similarity score between these two texts attempts to measure the likelihood that the ten words in the mystery text come from the same class of text as the 100 words in the source text. (A given class of text could be based on a particular author or publication, or on other characteristics of the texts in question.)

To calculate the score, we first take each word in the mystery text and compute a probability for it that is based on the number of times that it occurs in the source text. If a word in the mystery text doesn't occur at all in the source text (which would lead to a probability of 0), we instead compute a probability that is based on a "default" word count of 0.5. This will allow us to avoid multiplying by 0 when we compute the final score.

Here are the probabilities for the words in our mystery text:

- "love" has a probability of 50/100 or 0.5 (it occurs 50 times out of the 100 words in the source text)
- "thou" has a probability of 42/100 or 0.42
- "potter" has a probability of 0/100, but we change it to 0.5/100 or 0.005 to avoid a probability of 0
- "spam" has a probability of 0/100, but we change it to 0.5/100 or 0.005

Important: These probabilities have denominators of 100 because the source text has 100 words in it. The denominators should *not* always be 100!

To compute the similarity score of the mystery text, we need to compute a product in which a given word's probability is multiplied by itself *n* times, where *n* is the number of times that the word appears in the mystery text. In this case, we would do the following:

```
#           3 "love"   1 "thou"  2 "potter"         4 "spam"
sim_score = (.5*.5*.5) * (.42) * (.005*.005) * (.005*.005*.005*.005)
```

This similarity score is very small! In practice, these very small values are hard to work with, and they can become so small that Python's floating-point values cannot accurately represent them! Therefore, instead of using the probabilities themselves, we will use the *logs* of the probabilities. The log operation transforms multiplications into additions (and exponents into multiplication), so our log-based similarity score would be:

```
log_sim_score = 3*log(.5) + 1*log(.42) + 2*log(.005) + 4*log(.005)
```

This results in a more manageable value of around -34.737. (Note that Python's math.log function uses the natural log (of base *e*) by default, which is fine for our purposes.)

The resulting similarity score gives us a measure of how similar the mystery text is to the source text. To classify a new mystery text, we compute similarity scores between it and a collection of known texts in order to determine which of the known texts is most likely to be related to the

mystery text. For example, let's say that we also have the following model for texts by J.K. Rowling:

```
jkr = {'love': 25, 'spell': 275, 'potter': 700}
```

Note: there are a total of 1000 words in this model.

We can compute a similarity score for the mystery text and the jkr texts in the same way that we did above. We get the following probabilities that the words in the mystery text came from the jkr texts:

- "love" has a probability of 25/1000 or 0.025
- "thou" has a probability of 0.5/1000 or 0.0005 (using the default value of 0.5 for the count, because "thou" does not appear in the jkr texts)
- "potter" has a probability of 700/1000 of 0.7
- "spam" has a probability of 0.5/1000 or 0.0005

Thus, the non-log similarity score for 3 occurrences of "love", 1 occurrence of "thou", 2 occurrences of "potter", and 4 occurrences of "spam" would be:

```
sim_score = (.025*.025*.025) * (.0005) * (.7*.7) *
(.0005*.0005*.0005*.0005)
```

This value is also very small! Using logs:

```
log_sim_score = 3*log(.025) + 1*log(.0005) + 2*log(.7) + 4*log(.0005)
```

Now the similarity score is approximately -49.784. This value is less than the value that we computed when comparing the mystery text to the Shakespeare text. Therefore, we can conclude that the mystery text is more likely to have come from Shakespeare than from J.K. Rowling.

compare_dictionaries(d1, d2)

Write a function (**not a method, so it should be outside the TextModel class**) named `compare_dictionaries`. It should take two feature dictionaries `d1` and `d2` as inputs, and it should compute and return their log similarity score. Below is some pseudocode for what you will need to do:

1. Start the score at zero.

2. Let `total` be the total number of words in `d1` – not only distinct items, but all of the repetitions of all the items as well. (For example, `total` for our example `shakespeare_dict` would be 100.)
3. For each item in `d2`:
    a. Check if the item is in `d1`.
    b. If so, add the log of the probability that the item would be chosen at random from everything in `d1`, multiplied by the number of times it appears in `d2`.
    c. If not, add the log of the default probability (0.5 / `total`), multiplied by the number of times the item appears in `d2`.
4. Return the resulting score.

similarity_scores(self, other)

Write a method `similarity_scores(self, other)` that consumes another `TextModel` and computes and returns a *list* of log similarity scores measuring the similarity of self and other – one score for each type of feature (words, word lengths, stems, sentence lengths, and your additional feature).

The `compare_dictionaries` function you wrote will be helpful in your implementation for this function, as it returns the log similarity score between two dictionaries.

Important: In order for your statistics to be calculated correctly, whenever a `TextModel` instance calls the compare_dictionaries function, the dictionary belonging to that instance self should be the second parameter of the call. For example:

```
word_score = compare_dictionaries(other.words, self.words)
```

classify(self, source1, source2)

Write a method `classify(self, source1, source2)` that compares the called `TextModel` object `self` to two other "source" `TextModel` objects (`source1` and `source2`) and returns which of these two `TextModels` is the more likely source of the called `TextModel` self.

You should begin by calling `similarity_scores` twice:

```
scores1 = self.similarity_scores(source1)
scores2 = self.similarity_scores(source2)
```

You should then use these two lists of scores to determine whether the called `TextModel` is more likely to have come from `source1` or `source2`. **You are free to do this in any way you**

choose, but note that you will have to document your approach later in the project in Phase 5.

Example Approach 1: Best of Five

One way to do this is to compare corresponding pairs of scores, and determine which of the source `TextModels` has the larger number of higher scores.

For example, imagine that the two sets of scores are the following:

```
scores1 = [-34.737, -25.132, -55.312, -10.715, -47.125]
scores2 = [-49.091, -21.071, -60.154, -16.502, -43.675]
```

`scores1` has a higher score for three of the features (the ones in positions 0, 2, and 3 of the lists), while `scores2` has a higher score for only two of the features (the ones in positions 1 and 4). Thus, we conclude that self is more likely to have come from `source1`.

Example Approach 2: Weighted Sum

Another approach to using the two lists of scores is computing a weighted sum of the scores in each list, which can be done by doing something like this:

```
weighted_sum1 = 10 * scores1[0] + 5 * scores1[1] + 7 * scores1[2] + ...
weighted_sum2 = 10 * scores2[0] + 5 * scores2[1] + 7 * scores2[2] + ...
```

You could then base your classification on which source's weighted sum is larger. One advantage of this approach is that it allows you to adjust the relative importance of the different features – giving certain features a larger impact on the classification.

compare(self, source1, source2)

Write a method `compare` that prints out a "classification report" that contains the lists of similarity scores for `source1` and `source2` and some descriptive statement on which source the `TextModel` self most likely originated from.

This method does not need to return anything. As long as the lists of similarity scores and the final statement are in the message printed by compare, you are free to format your message as you see fit. For example, this is how the TAs formatted their output:

```
>>> mystery.compare(source1, source2)
scores for source1: [-16.394, -9.92, -15.701, -1.386, -1.386]
```

```
scores for source2: [-17.087, -15.008, -17.087, -1.386, -3.466]
mystery is more likely to have come from source1
```

Some of your similarity scores will be different than ours (e.g., the third scores, which depend on how you stem the words, and the fifth scores, which depend on which additional feature you include). Our conclusion is based on a pairwise comparison of the scores: because `source1` has a larger number of higher scores, it is chosen as the more likely source. If you use the lists of scores in another way, you may come to a different conclusion, which is fine!

Checkpoint

At this point, you should have developed a sophisticated `TextModel` class that can perform similarity comparisons between different bodies of text. For example:

```
>>> source1 = TextModel('source1')
>>> source1.add_string('It is interesting that she is interested.')

>>> source2 = TextModel('source2')
>>> source2.add_string('I am very, very excited about this!')

>>> mystery = TextModel('mystery')
>>> mystery.add_string('Is he interested? No, but I am.')
>>> mystery.compare(source1, source2)
scores for source1: [-16.394, -9.92, -15.701, -1.386, -1.386]
scores for source2: [-17.087, -15.008, -17.087, -1.386, -3.466]
mystery is more likely to have come from source1
```

Phase 5: Experiment

Now that your `TextModel` class is complete and you have tested its ability to compare texts, you should choose several bodies of text from which you can create models and compute similarity scores.

Curating TextModel Sources

Choose two bodies of text from which you will create two "source" `TextModels`. In the example we provided at the start of Phase 4, we chose William Shakespeare and J. K. Rowling texts for our source models, and then selected a new mystery text to compare them against. For this part, you should similarly choose two bodies of text for your source models, and in the next part you will choose new texts to compare them against.

Note that we say *bodies* of text, because a given text model can be based on more than one text document. For example, if you want to build a text model for *New York Times* articles, you should base it on multiple articles from the *Times*. (Another reason for combining multiple documents from the same source is that models based on too small a source text tend to be brittle—they depend too much on the idiosyncrasies of that source.)

You should choose two bodies of text that allow for meaningful comparisons. For example:

- works by Shakespeare vs. works by J.K. Rowling
- articles from the *New York Times* vs. articles from the *Wall Street Journal*
- scripts from *Friends* vs. scripts from *How I Met Your Mother*
- two styles of writing (e.g., articles written for scientific journals vs. articles written for popular magazines, or articles written for two different sections of a given publication)

You are welcome to choose whatever texts you like, but as a starting point, here is a link to a text file containing the complete works of William Shakespeare. You should not use this file as it is. You should download it, open it, and remove the text at the beginning and end that explains the file and provides additional information. This is true of *any* text file(s) that you use – you should inspect them and perform whatever human pre-processing is necessary to clean the file before handling it computationally.

Note: We encourage you to *leave out* at least one text from each body of text when creating your models. This will allow you to use it for testing. For example, if your two bodies of text are a collection of articles from the *New York Times* and a collection of articles from the *Wall Street Journal*, you can use most of the articles from a given collection to build its text model, but leave out one article from each collection so that you can perform tests to see if the *Times* article that you left out when building your source models is really more similar to the other *Times* articles than it is to the *WSJ* articles.

Choosing Documents to Compare

Once you have your two source models, **you should choose at least four new text documents (texts not used in the creation of your source models) that you would be interested in classifying according to your source models.**

For example, you could see if:

- your first year seminar paper is more like works by Shakespeare or Rowling
- *Providence Journal* is more like the *New York Times* or the *Wall Street Journal* (or perhaps the *Daily Mail*)
- Bart Simpson is more like Chandler Bing or Barney Stinson

Be creative! For each text/collection of texts that you want to classify, you will again need to obtain one or more text files, pre-process them, and create a TextModel object from them. You should then invoke the compare method on that TextModel to see which of your models is the more likely source.

Using run_experiments() to Run an Experiment

Edit the `run_experiements` function in the stencil code to:

- Create a `TextModel` instance for each of your two chosen bodies of text
- Classify your four documents using each of the two new `TextModels`
- Output meaningful data using the compare function in the `TextModel` instances

For example, this is an example of how we implemented part of our `run_experiments` function:

```
def run_experiments():
    source1 = TextModel('rowling')
    source1.add_file('rowling_source_text.txt')

    source2 = TextModel('shakespeare')
    source2.add_file('shakespeare_source_text.txt')

    new1 = TextModel('seminar')
    new1.add_file('seminar_source_text.txt')
    new1.compare(source1, source2)

    # more code goes here for your other three models
```

You should replace the model names and file names in the provided code with the names of your models and text files. Don't forget that you can use more than one file to build a given model, in which case you would call `add_file` multiple times for that model. Appropriate use of your `save_model` and `read_model` methods will even allow you to efficiently work with really large texts.

Important: Be sure and hand-in everything (including all your text files) needed to let the graders execute your `run_experiments` function.

technical_report.txt

In a file named `technical_report.txt`, include a report containing answers to the following questions:

- What text features did you use? What was the additional feature you chose?
- What approach(es) did you use for your ***classify*** method? Why did you choose this approach?
- Which source bodies of text did you choose? Which new texts/bodies of text did you choose to compare against the sources? Why did you choose those sources and documents?
- What were the results of your comparisons?
- How well do you think your text classification program works? How could it be improved?

Extra Credit

For up to 5 points of extra credit, you may extend your TextModel to include other text analysis features (anything outside of sentence length, word length, word counts, stem counts, and your additional chosen feature is fair game).

Document all extra functionality in your `technical_report.txt`, along with a description of why you have chosen to implement it. In particular, we will be looking for extra features that are tailored towards the specific texts you have chosen to analyze.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this document or have any concerns about this and any other CS4 document by posting on Piazza or filling out our anonymous feedback form.