Text ID OLD

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/TextClouds on 3/22/2107

This was NOT used in fall 2015

Instead you want this page for the up-to-date TextID project

This project offers you an opportunity to use Python to develop a statistical model of text that will allow you to "ID" an author or style -- with varying degrees of success -- from samples of text...

Milestone requirements for Text ID

You should have a TextModel class along with these methods, working at least to the extent described:

- a working constructor and `__repr__`: a version of each is provided in the description
- `readTextFromFile(self, filename)`: should take in a filename (a string) and should return all of the text in that file
- `addTextFromString(self, s)`: should use the text in the input string `s` to add to the self.sentencelengths, self.words, and self.wordlengths dictionaries (the self.stems dictionary should be working for the final deliverable, but does not need to work for the milestone)
- `printAllDictionaries(self)`: this method simply prints all of the `TextModel`'s dictionaries (it's especially helpful for debugging)
- `saveModelToFile(self)`: you need this method to write all of the `TextModel`' dictionaries to a file. The file should be named using the string in `self.name` with the file extension `.txt`
- `readModelFromFile(self)`: This method should open the file (named as described above) and read the dictionaries into the text model

Be sure to zip up the folder containing your Text ID project file (or files) and name it `textid_milestone.zip` -- then submit that in the right spot for Hw#12!

Background

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. Details on that story and the analysis done appear at this link.

The comparative analysis of those works used simple models -- perhaps surprisingly simple -- of an "author's style," for example, the distribution of word lengths in a document. The four "text features" that we ask you to implement for this project are

- The distribution of words used by an author
- The distribution of word lengths (noted above)
- The distribution of word-stems used (e.g., "spam" and "spamming" would have the same stem)
- The distribution of sentence-lengths used
- *plus*, the project asks you to include one more "text feature" of your own design
    - for example, you could use punctuation in some way
    - or compute something else about the text - it could be the number of words ending in "ness," something about capitalization, or anything else you can compute or count using Python...

Perhaps the *most-common and most-successful application* of this kind of textual analysis is Spam detection and filtering.

Getting started: the `TextModel` overview

To create your Bayesian classifier, you will extend your Markov text model with a more complete statistical model of a body of text.

All of the examples above have used words as features and word-frequency dictionaries as models. Here, we provide a constructor to get you started with the following empty dictionaries

- `self.words`
- `self.wordlengths`
- `self.stems`
- `self.sentencelengths`
- remember that you'll need one more dictionary for a text-feature of your own design!

Computing word lengths from words won't be a problem. However, computing stems and sentence lengths are an additional part of the challenge. Some starting functions and other hints are below.

`TextModel` starter code

To get started, here is *part* of a constructor, `__init__` and string-conversion function, `__repr__`:

```
class TextModel:

    def __init__(self, name):
        """ the constructor for the TextModel class
            all dictionaries are started at empty
        """
        self.name = name
        self.words = {}   # starts empty
        self.wordlengths = {}
        self.stems = {}
        self.sentencelengths = {}
        # you will want another dictionary for your text feature


    def __repr__(self):
        """ this method creates the string version of TextModel
objects
        """
        s  = "\nModel name: " + str(self.name) + "\n"
        s += "    n. of words: " + str(len(self.words))  + "\n"
        s += "    n. of word lengths: " +
str(len(self.wordlengths))  + "\n"
        s += "    n. of sentence lengths: " +
str(len(self.sentencelengths))  + "\n"
        s += "    n. of stems: " + str(len(self.stems))  + "\n"
        # you will likely want another line for your custom
text-feature
        return s
```

In addition, here are several methods you should implement for the *milestone* part of the project:

- `addTextFromString(self, s)`: should use the text in the input string `s` to add to the self.sentencelengths, self.words, and self.wordlengths dictionaries (the self.stems dictionary should be working for the final deliverable, but does not need to work for the milestone)
- `readTextFromFile(self, filename)`: should take in a filename (a string) and should return all of the text in that file
- `printAllDictionaries(self)`: this method simply prints all of the `TextModel`'s dictionaries (it's especially helpful for debugging)
- `saveModelToFile(self)`: you need this method to write all of the `TextModel`' dictionaries to a file. The file should be named using the string in `self.name` with the file extension `.txt`
- `readModelFromFile(self)`: This method should open the file (named as described above) and read the dictionaries into the text model
- you should create a method called `cleanText(self, text)`: use this as a starting point!

```
def clean(self,w):
    for p in string.punctuation:
        w = w.replace(p,'')
    return w.lower()
```

You will need `import string` at the top!

Hints and other methods

One of the best approaches to all new software projects is to make small examples work at first.

Here is a start for the `addTextFromString` method. Certainly you'll want to expand this.

```
def addTextFromString(self, s):
    """ analyzes the string s and adds its pieces
        to all of the dictionaries in this text model
    """
    LoW = s.split()
    for w in LoW:
        if w not in self.words: self.words[w] = 1
        else: self.words[w] += 1
```

Throughout the project, you may end up improving and expanding this function, but it's important to get it working early at a basic level, i.e., adding words to the `self.words` dictionary.

For the milestone, you only need to have the `self.words` and `self.wordlengths` dictionaries working. Here is a test to make sure you're on track. *Note that this example has all five dictionaries working, but for the milestone you only need words, wordlengths, and sentencelengths working!*

```
          >>> T = TextModel( "test" )
          >>> T.addTextFromString( "This sentence has five
words." )
          >>> print T
          Model name: test
            n. of words: 5
            n. of word lengths: 4
            n. of sentence lengths: 1
            n. of stems: 5
            n. of punctuation marks: 1  # for my implementation
- your may vary!

          >>> T.words
          {'this': 1, 'has': 1, 'five': 1, 'words': 1,
'sentence': 1}

          >>> T.wordlengths
          {8: 1, 3: 1, 4: 2, 5: 1}

          >>> T.sentencelengths
          {5: 1}

          >>> T.stems
          {'sentenc': 1, 'word': 1, 'ha': 1, 'five': 1, 'thi':
1}
          # I used the porter stemmer - see below

          >>> T.punc
          {'.': 1}

          # dictionaries don't have an order, so different
orderings are OK
```

Do be sure to run the above tests or others like it, to make sure your code is working thus far... !

- **`readTextFromFile(self, filename)`**
  augments the model -- again, all of the dictionaries -- with the text in file `filename`. Your best bet here is to open the file, read all the text into a string, and then call `self.addTextFromString` on that string! That way, all of the functionality you add to `addTextFromString` will be used by this file-handler, as well! You may want to refer to your Markov text-generation code to remind you how file-handling works!

  Note that this is very different than the `readModelFromFile` method: `readTextFromFile` reads a plain-text file with prose in order to model it. The `readModelFromFile` will read in from a file that you write a list of Python dictionaries, i.e., an already-created model.

- **`cleanText(self, s)`**
  "cleans" the text in string `s` in a reasonable way. This does not have to be perfect -- in fact, it can't be. But, it should make sure punctuation is removed (*after* counting sentences, to be sure!) It should also ensure that a list of words (all lowercase) is returned. Some people write two "cleaning" functions -- one to clean the full string of text and another to break it into words and clean the words. It's up to you.

  Helpful hints:
  Hint #1: You will find the built-in string methods `replace` and `lower` helpful -- try this example out in your Python shell:

```
s = "Ms. Rowling writes."
print s
s = s.replace( "Ms.", "Ms" ) # replace "Ms." with "Ms"
print s
s = s.lower()
print s
```

  Warning! Do ***not*** use the function `remAll` that we wrote using recursion -- you'll run out of recursive stack on large files!!

- **`saveModelToFile(self)`**
  This method should create a list of all of `self`'s dictionaries and save them to a file that uses the string `self.name` as its name, with `.txt` as its file extension. See the next method for a set of example tests... .

Here is a small function to use as a starter - but you'll definitely need to adapt it to make it a working method of your `TextModel` class:

```
def saveListOfTwoDs():
    """ starter function - be sure to adapt! """
    # for now, L is a list of two dictionaries
    # you will want it to be a list of _all_ your dictionaries!
    L = [ {"a":97, "b": 98},  {"spam":42} ]

    # we open the file into which to store L
    f = open( "thefile.txt", "w" ) # "w" == "write"

    # print the list L into the file, then close it
    print >> f, L
    f.close()
    # that's it!
```

You should be able to test this with `saveList()`, and it should make a short file named `thefile.txt` containing that list of two dictionaries.

- **readModelFromFile(self)**
  This is the complementary method to `saveModelToFile` method, above. Here is a starting point. Again, make sure this works -- and then you'll need to adapt it to your `TextModel` class:

```
def readListOfTwoDs():
    """ starter function - matches saveList() """
    # open the file for reading
    f = open( "thefile.txt", "r" ) # "r" == "read"

    # we read _all_ of the contents into the string data
    data = f.read()
    f.close()

    # Python magic! We _eval_ the string to get the list of 2 dictionaries
    L = eval(data)

    # We then unpack into two different names
    d1, d2 = L  # gives each of the two a different name
    print "d1 is", d1
    print "d2 is", d2
    print 'd1["b"] is', d1["b"]
    # that's it!
```

Run this *readList* function *after* you run *saveList*.

Additional methods for the final portion of the project...

For the final version, you will need to implement methods that

- find stems from words
- compare two dictionaries (for example, with a method called `compare_two_dictionaries(self, d_test, d_model)`
- compare two `TextModel` objects (by comparing their corresponding dictionaries pairwise)
  below, this is called `compute_similarity(self, model2)`
- output the results in a readable way (this is up to you!)
- Then, we ask you to use your text-modeler and -matcher to compare at least four texts:
  - you'll create at least two text-models, e.g., Rowling and Shakespeare
  - you'll find two ***test*** texts of your own choice, e.g., your writing or others...
  - you should then determine which of the (two or more) text-models your test-texts are more similar to...
  - you'll write up the results (briefly) in a file named final.txt

Here are guidelines for this part of the project.

- **`stem(self, word)`**
  should return the stem (a string) from the input `word`. This will be used to populate the `self.stems` dictionary (for the final, not necessarily the milestone). You may write your own stemming function, in which case we ask you to implement at least 12 different stemming rules. (There are many more than that!) We realize - as you do! - that no stemming algorithm will be perfect. Don't worry about this. If you would rather use an off-the-shelf stemming algorithm, you may download and use the *Porter stemmer* from this site. ***How do I use that file?*** To get started, I'd suggest these steps:
  - download that file and save it as `ps.py` *in the same folder that you're working*
  - at the top of your project file, import that file with the line `import ps`

    - you need to create a `PorterStemmer` object. Use a line such as `self.ps = ps.PorterStemmer()` within the `__init__` method of your `TextModel` class
    - finally, you need a way to call it -- ***here's where you write*** `stem(self,word)`
    - Within your `stem(self,word)` method, you should call the Porter stemmer function as follows: `self.ps.stem(word, 0, len(word)-1)`. Note that the original version requires passing in the start and end indices of the word, but your `stem` method can include those things instead of forcing the user to always generate them.

  From here, you should be able to test your `stem` method as follows:

    - Note that you might put these lines at the bottom of your file for easy reloading...
    - `t = TextModel( 'test' )`
    - `print t.stem( "singing" )`
    - `print t.stem( "sing" )`
    - The stemmer does pretty well on these two words -- despite their rules being quite different!

  Now you're ready to incorporate it into your program! You can also write your own parser, but the Porter stemmer is considered one of the best -- it's one of the earliest and most widely-used stemmers. The examples on this page use the Porter stemmer.

- You will also need **`compare_two_dictionaries( self, d_test, d_model )`** and **`compare_with_two_models( self, model1, model2 )`**. An example motivates these and then provides an outline:

Motivating example for `compare_two_dictionaries`

If our features are words, then our model of *all words* in a text model will be a Python dictionary: the dictionary's keys are each word and each value is the number of times the key (word) appears in the document. Here is a contrived example of a model dictionary, named **d_model**. This would arise from a 100-word document if 50 of those 100 words had been "love" and 8 had been "spell" and 42 had been "thou"

d_model = { "love":50, "spell":8, "thou":42 }

Next, you want to compare some new *test* text against this model. Suppose a test document gave you a test dictionary named `d_test`, with 10 words, as follows:

d_test = { "love":3, "thou":1, "potter":2, "spam":4 }

Our Bayesian similarity score between **d_test** and **d_model** will be the probability of the 10 `d_test` words arising from the model of `d_model`:

- "love" has 50/100 probability of occurring (each of three times)
- "thou" has 42/100 probability of occurring (once)
- "potter" and "spam" don't appear

We avoid giving "potter" and "spam" probabilities of 0 - multiplying by 0 would remove all of the information in the score! Instead, we give them a "default" frequency of 1, so that each will act as if it had probability 1/100 of arising from the **d_model** model. (You can adapt this... .) If each word was independent, we would multiply each of the probabilities:

prob = (.5*.5*.5)*(.42)*(.01*.01)*(.01*.01*.01*.01)

which is very small, about 0.0000000000000525. This computes the probability in theory... . In practice, however, those very small values are hard to work with -- and they can get so small that Python's floating-point values can't hold them (they "underflow")!

So, instead of outright multiplication of those probabilities, we take the log of each. (At the top of your file write `from math import log`.) The log function transforms multiplications into additions (and powers into multiplication), so our log-probability will be

log_prob = 3*log(.5) + 1*log(.42) + 2*log(.01) + 4*log(.01)

which results in the unusual, but more manageable, value of around -30. (I think of it as a probability of 10^-30.) **Note on logs** Using logs helps avoid the values getting too small, but what do those negative numbers mean? If your log-probability score is -5, that means the probability has (about) 5 decimal places, i.e., it's about 0.00001. If another log-probability score was -10, then *that* second probability is about 0.00000000001. That latter probability is *much* less likely than the former one. It's in those *comparisons* of likelihood that this technique works well. More precisely, its e^-30, because Python uses the natural log (log-base-e) by default. If you want log-base-10, no worries: use `math.log(value,10)`. You'll see that it's about -13.3 for this example, matching the 13 zeros in the small probability above. Since this is about making comparisons, however, either way is OK!

Your task is to implement this and return the **log_prob** for the `d_test` and `d_model`. Here's a pseudocode outline:

- Start the `log_prob` at 0.

- Let `model_total` be the total number of items in `d_model` -- not only distinct items, but all of the repetitions of all of the items, added up. (You'll need a loop!)
- Then, for each item in `d_test`...
- Check if that item is in `d1` at all...
- If so, add the log of the probability of that item (that it would be chosen at random from everything in `d_model`), times its number of appearances in `d2`.
- If not, add the same thing, but use, for example, `1.0` as the numerator. Be sure to still multiply by its frequency in `d_test` and be sure you use floating-point division for all of your probabilities!
- Then, you should return the resulting `log_prob` score

Try it out!

Try this test (reflecting the above example) of your `compare_two_dictionaries` method:

```
t = TextModel( "test" )
d_test = { "love":3, "thou":1, "potter":2, "spam":4 }
d_model = { "love":50, "spell":8, "thou":42 }
log_prob_score = t.compare_two_dictionaries( d_test, d_model )
print "log_prob_score is", log_prob_score
```

here is my result:

```
log_prob_score is -30.5779632253
```

which is log-base-e (the default, natural log). For log-base-10, it is about -13.27

Final step: comparing a test text against two models:

`compare_with_two_models(self,m1,m2)`

The ultimate goal of our comparisons is to

- create two (or more) models, e.g., (Shakespeare vs. Rowling)

- compare a new, *"test"* text against those two (or more) models to see which one it's (most) close to...

To do this, you will need to write a method **`compare_with_two_models( self, model1, model2 )`** .

Here's how it will work at the level of Python calls:

- you'll need to create two (or more) foundational models, for example, one named `WS` (Shakespeare) and one named `JKR`
- you will also need to create a third `TextModel` object -- but this one contain be the ***test*** data -- this of this third one as the "test text"
- So, suppose you've created this and named it `testtext`
- Then, you would call
- `testtext.compare_with_two_models( WS, JKR )  # WS is m1,  JKR is m2`

  in order to compute - and print - the results of the comparisons.

- Inside the method `compare_with_two_models` you will need to
  - scale the dictionaries-to-be-compared so that they're the same size (see below)
  - compute the ten similarity scores between the dictionaries to be compared:
  - 5 scores will be between `testtext`'s five dictionaries and `m1`'s five dictionaries
  - 5 scores will be between `testtext`'s five dictionaries and `m2`'s five dictionaries
- You should be sure to print all 10 similarity scores - they're really log-probability scores - with labels so that a reader known which one is which...
- Then, you'll have the rest of the code decide which of the two models, `m1` or `m2`, was more similar to `testtext`
- Determining which of the two models is the better match is up to you. Some possibilities include
  - boiling down the five scores to a single number -- a sum or a weighted sum -- and then simply comparing those two numbers
  - using the five scores as "votes." Since there are an odd number of scores, one of the two models must get more votes (more scores that are closer to it) -- and you could choose that one as the winner.
  - combinations are welcome:
    - for example, you could give weighted votes to each of the scores, e.g., words are "worth more" than punctuation...

Important: *scaling* the model dictionaries so that they're the same size...

You may notice that *different numbers* of items in models will influence the similarity score.

As an example, consider the test dictionary (of only two words):

d_test = { "love":1, "spam":1 }

And consider two models the ***seem*** to be identical, they're just different sizes:

d_model1 = { "love":5, "potter":5 } and d_model2 = { "love":50, "potter":50 }

If you run the following two tests, you'll see that `d_test` gets a different score against each:

```
TestText = TextModel( "cs5hw" ); TestText.addTextFromString(
"spam alien." )
M1 = TextModel( "m1" );  M1.addTextFromString( "love "*5 + "spam
"*4 + "spam." )
M2 = TextModel( "m2" );  M2.addTextFromString( "love "*50 +
"spam "*49 + "spam." )

# a test WITHOUT scaling (different scores! ... though they
_shouldn't_ be different)
print "unscaled vs. 1: ",
TestText.compare_two_dictionaries(TestText.words, M1.words)
print "unscaled vs. 2: ",
TestText.compare_two_dictionaries(TestText.words, M2.words)
```

Here are my results - notive that they're different, even though we would really want them to be the same!

```
unscaled vs. 1:  -2.99573227355
unscaled vs. 2:  -5.29831736655
```

(The above numbers are base-e; the base-10 numbers are about -1.3 and -2.3) The ***reason*** tha tthey're different is that `spam`, which appears in neither of the models, gets a much higher probability (1/10) in M1 than it does in M2 (1/100). Next, you'll fix this!

Fixing the scaling problem with different-size dictionaries...

To fix the problem noted above, the right thing to do is to *scale* both dictionaries to the "same size" - without changing the relative frequency of words in each.

Here are two methods to help with this -- feel free to copy these into your `TextModel` class:

```
    def get_denominator(self, d):
        """ gets the total of values in d """
        return 1.0*sum(d.values())

    def scale(self, d, new_denominator):
        """ returns a NEW dictionary with its values
            scaled so that they sum to new_denominator
            must have non-zero old denominator and
                      non-zero new_denominator!
        """
        new_d = {}  # the new dictionary to be built...

        old_den = self.get_denominator(d)
        if old_den == 0.0 or new_denominator == 0.0:
            print "can't scale a dictionary with"
            print "old or new denominators == 0.0"
            print "old_den ==", old_den
            print "new_den ==", new_denominator
            return new_d  # returns the empty dictionary

        multiplier = new_denominator*1.0/old_den
        for k in d: # for each key in the old dictionary, d
            new_d[k] = d[k]*multiplier

        return new_d
```

To use them, you would

- find the size (the "denominator") for each of your two model dictionaries
- find the larger of the two (with `max`)
- scale them both to the larger value
- use the scaled dictionaries instead of the originals

Try it out - use this scaled version of the previous example:

```
TestText = TextModel( "cs5hw" ); TestText.addTextFromString(
"spam alien." )
M1 = TextModel( "m1" );  M1.addTextFromString( "love "*5 + "spam
"*4 + "spam." )
M2 = TextModel( "m2" );  M2.addTextFromString( "love "*50 +
"spam "*49 + "spam." )
```

```
# Now, WITH the scaling: should give the same (further-from-
zero) scores...
den1 = TestText.get_denominator(M1.words)
den2 = TestText.get_denominator(M2.words)
den_max = max( den1, den2 )
scaledwds1 = TestText.scale(M1.words, den_max)
scaledwds2 = TestText.scale(M2.words, den_max)
print "scaled vs. 1: ",
TestText.compare_two_dictionaries(TestText.words, scaledwds1)
print "scaled vs. 2: ",
TestText.compare_two_dictionaries(TestText.words, scaledwds2)
```

which should result with this: and about -2.3 if using log-base-10

```
scaled vs. 1:  -5.29831736655
scaled vs. 2:  -5.29831736655
```

You ***should*** use this scaling in order to make your comparisons more meaningful -- otherwise the smaller dictionaries tend to win every time, simply because they weight unseen words as more likely.

Testing and an example...

Here is a small example -- some of your dictionaries may be different, but it provides a rough outline of what your output from `compare_with_two_models` could look like. For the record, we've printed all of our models' dictionaries below the output of `compare_with_two_models` - you won't need to do this (unless, perhaps, it's part of your debugging...).

First, the three inputs - two models and one testtext:

```
M1 = TextModel( "m1 (shorta)" ); M1.addTextFromString( "This is
a short sentence. A is a short word." )
M2 = TextModel( "m2 (gohens)" ); M2.addTextFromString( "Go hens,
Go. Go, go hens!! Go, indeed!" )
testtext = TextModel( "test" ); testtext.addTextFromString(  "A
short a is indeed short." )

# here is how to make the call to compare_with_two_models:
testtext.compare_with_two_models( M1, M2 )
```

Here is our output - your formatting may be different, but this shows one possible method for displaying the results. The log-base-10 results are between -4 and -6 for the first few scores...)

```
Scoring the testtext with name: test
   against the model with name: m1 (shorta)
 & against the model with name: m2 (gohens)

Comparing words which yield the scores
   vs. m1 (shorta) :  -9.53884443895
   vs. m2 (gohens) :  -13.5923670067

Comparing wordlengths which yield the scores
   vs. m1 (shorta) :  -9.53884443895
   vs. m2 (gohens) :  -11.7597855429

Comparing stems which yield the scores
   vs. m1 (shorta) :  -9.53884443895
   vs. m2 (gohens) :  -13.5923670067

Comparing sentencelengths which yield the scores
   vs. m1 (shorta) :  -1.09861228867
   vs. m2 (gohens) :  -1.09861228867

Comparing punc which yield the scores
   vs. m1 (shorta) :  0.0
   vs. m2 (gohens) :  -1.94591014906


Tallying votes:
  Model m1 (shorta): 5 votes
  Model m2 (gohens): 0 votes

+++   Thus, the testtext has features closer to model m1
(shorta)
```

Here are the full models we obtained. Your algorithms may count punctuation or other features slightly differently:

```
Model name: test
    n. of words: 4
    n. of word lengths: 4
    n. of sentence lengths: 1
    n. of stems: 4
    n. of punctuation marks: 1

words :  {'a': 2, 'indeed': 1, 'is': 1, 'short': 2}
```

```
wordlengths :  {1: 2, 2: 1, 5: 2, 6: 1}
stems :  {'a': 2, 'inde': 1, 'is': 1, 'short': 2}
sentencelengths :  {6: 1}
punc :  {'.': 1}


Model name: m1 (shorta)
    n. of words: 6
    n. of word lengths: 5
    n. of sentence lengths: 1
    n. of stems: 6
    n. of punctuation marks: 1

words :  {'a': 3, 'short': 2, 'word': 1, 'sentence': 1, 'this': 1, 'is': 2}
wordlengths :  {8: 1, 1: 3, 2: 2, 4: 2, 5: 2}
stems :  {'a': 3, 'short': 2, 'word': 1, 'sentenc': 1, 'is': 2, 'thi': 1}
sentencelengths :  {5: 2}
punc :  {'.': 2}


Model name: m2 (gohens)
    n. of words: 3
    n. of word lengths: 3
    n. of sentence lengths: 2
    n. of stems: 3
    n. of punctuation marks: 3

words :  {'go': 5, 'indeed': 1, 'hens': 2}
wordlengths :  {2: 5, 4: 2, 6: 1}
stems :  {'go': 5, 'inde': 1, 'hen': 2}
sentencelengths :  {2: 1, 3: 2}
punc :  {'!': 3, ',': 3, '.': 1}
```

Use your text-modeler!

Once your `TextModel` class is complete and you've tested its ability to compute match scores, you should choose two or more bodies of text from which to create models. For example, our demo used JK Rowling and Shakespeare, but you should choose two of your own "foundational" models.

You are welcome to choose whatever source texts you might like, but if you do want to use Shakespeare, here is a `.txt` file containing the complete works of Shakespeare. ***Don't use this file as-is***, however -- you should take a look at it and remove the front/back matter that accompanies Shakespeare's words (text that explains the file, its origin, etc.) This is true of any source file(s) you use -- you should be sure to look them over and do whatever human pre-processing is appropriate (and feasible) before handling it computationally... .

Just as examples of comparisons you could make, one could imagine

- actually comparing Shakespeare and J. K. Rowling
- comparing NYTimes and WSJournal articles
- comparing Big Bang Theory and Arrested Development
- choosing a more abstract comparison (one writing style vs. another)

- choosing a more concrete comparison (Sheldon vs. Leonard)

Once you have two "foundational" models -- representing two authors or artists, or styles (or Big Bang Theory characters), you should run at least two texts against each of them:

- you should run ***one*** of the original texts against each, in order to make sure that it scores ***better*** against the correct model than against the other one!
- you should also run an ***unrelated*** text against each -- and see which it resembles more closely

For example "unrelated" texts, you could see if

- your Writ1 paper (or thesis) is more like W.S. or J.K.R.
- the Chicago Tribune is more like the NYT or the WSJ
- Bart Simpson is more like Michael Bluth or Raj Koothrappali

or any other comparison you would like to brainstorm... .

Your analysis...

After running your test (or tests), you should write up, in a `final.txt` file, an explanation of

- the two or more categories you used as your "foundational" models
- the other two (or more) test texts you used to compare against them
- the results of all of those tests and comparisons - how well did it do? What kinds of similarities were most pronounced
- any additional tests you ran (more are great!)
- please include the detailed scores of at least one of the tests, printed out and formatted in an easy-to-read manner, as well (help the graders!)

Also, be sure to describe how the graders can run your tests -- you will want to make this as easy as possible to do, so write helper functions that will help the graders do so! Include in your final.zip archive whatever files (if they're not too big) are needed to create your models and try it out... .

Submission

Be sure to submit everything in a zip file named `textid_final.zip` to the *final* spot in the submission system...

Congratulations on building a Bayesian text-classification system (and on writing more like Rowling than Shakespeare!)

Some additional background on *Bayesian classification*

This project's basic algorithm is known as a Naive Bayes Classifier. Despite the "naive" in its name, this classifier has been hugely successful in distinguishing spam from non-spam ("ham") emails and, in different forms, it is used for many classification problems.

The approach boils down to computing the likelihood score of a set of new text features, given a dictionary of those features' appearances in the original text. The reason that the algorithm is called "naive" is that we make the assumption that each feature is *independent*. Thus, we assume that the appearance of the word `spell` does not depend on the appearance of the word `potter` -- and that this independence holds for all pairs of words and pairs of features throughout the text. This assumption is certainly not true, but that turns out not to matter in many situations!

With this assumption, Wikipedia derives the algorithm and summarizes it in a form I'd describe as less-than-illuminating:

$$\text{classify}(f_1, \ldots, f_n) = \operatorname*{argmax}_c \, p(C = c) \prod_{i=1}^{n} p(F_i = f_i | C = c).$$

Expressing this idea computationally is actually more natural than this notation - and is the purpose of this project. The central function for computing similarity - and the implementation of that formula above - is in `compare_two_dictionaries( self, d_test, d_model )`

Other helps...

Some folks end up with text files that are not-quite-all-plain-text... The additional characters are troublesome, so there is a way to get rid of them.

This `remove_non_ascii` is a small function that removes all non-plain-ascii characters from a string s. It includes a little bit of testing code, as well:

```
# -*- coding: utf-8 -*-
import string

def remove_non_ascii( s ):
    """ removes non-plain-ascii characters
        returns a copy of s with no such chars...
    """
    new_s = ''
    for c in s:
        if c in string.printable:
            new_s = new_s + c
    return new_s


# for testing the above function:
s = "This\nhas\n\xe2\x80\x99 some crazy chars."
new_s = remove_non_ascii( s )
print "    s is", repr(s)
print "new_s is", repr(new_s)
```

Extras...

There are lots of additional directions you might consider, if you'd like to take your Text ID project further.

If you do try one of these extras, be sure to note it in your `final.txt` file -- along with any instructions/analysis that go along with it!

More features/feature analysis

You can create/invent one or more features different than the four required above. Then, using texts with *known classification* analyze which features are better at classifying than others (at least for your datasets).

Present your analysis in your final.txt file.

Hierarchical models

Build a *hierarchical* model in which an object of `TextModel` contains a list of `TextModel`s!

This would be used, for example, to divide a single model of "Shakespeare" into two sub-models, perhaps

- Shakespeare comedies
- Shakespeare trajedies

More finely, it could be used to divide a model of "Romeo and Juliet" into a "Romeo" submodel and a "Juliet" submodel.

Similar subdivisions could be made for other sorts of source texts, as well.