Homework 6

[ 100 points - due by 11:59 pm, Sunday, March 26, 2017 ] *(the Sunday a week after Spring break…)*

Submit these files to the CS submission system at the usual place by 11:59. You may work on your own or with 1-2 partners on the programming portions of this assignment. (The reading/response is individual only.) Groups larger than 3, please split into smaller groups! Remember that partners need to work in the same physical location, share composition time equally (or each compose on their own machines) and be fully equal owners and producers of their work. *Have fun treeing (and foresting)!*  [cs35 homepage](https://www.cs.hmc.edu/~dodds/cs35/)

**Downloads**

Starter files to download -- grab them at the start of class & follow along:

* [The zip file, hw6.zip, to start this week's problems…](https://drive.google.com/open?id=0BwPWh-3AmiLxUXhnemNqLVhEeE0) [[ updated ]]
* [hw6pr5](https://github.com/ScriptingBeyondCS/CS-35/tree/master/week_6)

Deprecated import in hw6pr1.py: Line 21-22 should now be:

from gensim.models import KeyedVectors

m = KeyedVectors.load\_word2vec\_format(file\_name, binary=False)

As a result, comments in lines 24-28 are no longer accurate

**Submission**

**Overview** Please submit an archive named **hw6.zip** with the starter-files' filenames:

**hw6pr1.py** [**lab** problem #1] asks you to make sure the NLTK and TextBlob libraries are installed, to view a

short TextBlob tutorial, and to experiment with an "odd-one-out" algorithm with word2vec and matplotlib.

**hw6pr2.py** [**lab** problem #2] This problem asks you to use word2vec to create an analogy-checker and an

analogy-solver….

**hw6pr3.py** This problem combines word2vec's wordnetwork wtih TextBlob's part-of-speech and "lemmatizing"

to create one function that can paraphrase/reword a given sentence and another that can paraphrase an entire file

**hw6pr4.py** This problem asks you to classify movie reviews as positive or negative -- by designing a

combination of features from TextBlob's and NLTK's and word2vec's capabilities.

**hw6pr5.py** On a similar note, use a regressor to predict Amazon product reviews.

As usual, submit your reading response in its own spot at the [submission site](http://cicero.cs.hmc.edu/).

As always, extra-credit is available for posting code and a write-up of any one of these problems to your GitHub repository (be sure to let us know you've done this -- and provide a direct link)

**Problem 0**: Sentiment analysis through trees? [5 pts]

This homework's problem 0 is more *browsing* than *reading*... . Head to the [Stanford's Sentiment Analysis page](http://nlp.stanford.edu/sentiment/index.html) and read over the two summaries at left: *Deeply Moving...* and *Recursive Deep Models…* . Both refer to a sentiment-analysis project that builds trees out of sentences (from movie reviews) and then propagates positive or negative sentiment based on human-labeled examples (there are over 9,000 example trees in their database).

Then, try it out! Find a movie that you loved (or hated) and find (or write!) a part of a review of that movie. Go to the live demo page and paste in a few sentences from that review. The system will analyze the results. Finally, reflect on how the system does: do you agree with the sentiment estimated by the site? Find at least one piece of a sentence's parse tree that you agree with and one you don't, and note those in your response. Hover over particular words to get more information about them relative to the movie-sentiment model. Broadly speaking, reflect on how well or poorly such an approach might scale to assessing emotions *other than* positive/negative sentiments, e.g., anger, disappointment, determination, etc. As with each week's reading, responses should carefully considered, but need not be very long: a 4-5 sentence paragraph is wonderful.

**[Lab problem 1] Problem 1: One of these is not like the others...**

[25 pts; setup, lab walkthrough, and trying out word2vec]

* This problem asks you to run/alter the code in the **hw6pr1.py** file.
* [**Overall goals/tasks**] In this problem, you will compare sets of words in two ways:
  + Ussing word2vec’s **doesnt\_belong** to see which it thinks is *least* like the others
  + By using **matplotlib** to plot the words in a lower-dimensional (2d) space
  + Transforming large vectors of data into a lower-dimensional space is the heart of word2vec, deep learning, and feature engineering in general...
* [**Part 1**] Set up/**installing** libraries:
  + This week's assignment will use at least three libraries:
    - [NLTK](http://www.nltk.org/), the natural language toolkit, a large, widely-used library for natural language processing - this is included with Anaconda Python
    - The [gensim](https://radimrehurek.com/gensim/) library for word2vec
    - the [TextBlob](https://textblob.readthedocs.io/en/dev/) library for part-of-speech tagging, "lemmatizing," tokenizing, etc.
  + Install gensim with **conda install gensim** at the command-line
  + Install TextBlob by … (instructions follow)
  + Determining where conda via **which conda** (Mac) or **where conda** (Win)
  + Find the path to conda, e.g., mine were
    - **//anaconda/bin/conda** on the Mac
    - **C:\users\zdodds\Anaconda3\Scripts\conda.exe** on Windows
  + *Using that path without the* **conda***, run*  **PATH/pip install -U textblob** 
    - That is, for me, **//anaconda/bin/pip install -U textblob**
    - Or, on Win, **C:\users\zdodds\Anaconda3\Scripts\pip install -U textblob**
    - Your path will likely be different, to be sure!
    - By the way, **pip**  is an installation toolset for Python
  + Then, install the models and libraries with
    - **<PATH>python -m textblob.download\_corpora lite**
    - Again, for me, **<PATH>** was **//anaconda/bin/**
    - Or **C:\users\zdodds\Anaconda3\** (no **\Scripts** this time)
  + Phew!
* [**Part 2**] Try the libraries!
  + Make sure hw6pr1.py's functions run for you… this will check the install of the libraries… One reason hw6.zip was so large is that it contains the big word2vec model named **word2vec\_model.txt**. This has 300 numbers representing each of 43,981 words. Be sure that file stays in the same location as hw6pr1.py.
  + Try the four functions in the file - read them over and make sure you have a sense for what they're doing...
    - **m = read\_word2vec\_model()** This will read the word2vec model into  **m**
    - **most\_similar\_example(m)**  to show off **most\_similar**
    - **doesnt\_match\_example(m)**  to show off **doesnt\_match**
    - **textblob\_examples()**  to show off various language-processing functions
  + When those work (perhaps with juggling the libraries), go through the TextBlob introduction here:
    - <https://textblob.readthedocs.io/en/dev/quickstart.html>
  + It's quite short and to the point -- and has lots of things that are fun:
    - Tags (parts of speech)
    - Words and sentences (tokenizing)
    - Word stems (lemmatizing)
    - Singulars, plurals,
    - Definitions, Translations, etc.
* [**Part 3**] Try out PCA and **doesnt\_match** to experiment with word-outliers
  + The function visualize\_wordvecs is further down in the file… It shows off
    - PCA is a dimensionality-reduction approach that finds the "best" dimensions in a smaller space in which to represent data. It's explained well [at this site](http://setosa.io/ev/principal-component-analysis/).
    - The reduction from 200 dimensions to 2 dimensions with PCA (principal components analysis) Here's an example plot of a four-word wordlist. It chooses cereal as the outlier:



* + - Also, it uses **doesnt\_match** to find outliers from a list of words
  + **Your tasks** are to
    - Find two lists of four-or-more words (all in the model) where visualize\_wordvecs does a \_good\_ job of identifying an outlier - note them and the results there in the file.
    - Find two lists of four-or-more words (all in the model) where it's possible to see that visualize\_wordvecs has \_missed\_ the outlier (in some sense - you choose) Note these and the results there in the file, as well.
* [**Extra**] There is an example of a 3d PCA (using the iris data) at the bottom of the file. For extra-credit, adapt the word2vec's PCA to three dimensions, add labels, and show a word list that has outliers in each of two different dimensions! Include a screenshot and/or an image of your 3d words!

**[Lab problem 2] Problem 2: Analogies with word2vec...**

[25 pts; showing the central capabilities of word2vec] - can also be done in a *second lab...*

* This problem asks you to run/write code in the file **hw6pr2.py**
* [**Overall goals/tasks**] In class, we saw an example that, in part, helped word2vec create such traction: the example of “king - man + woman = queen.” For this problem, you will write a function to see how well word2vec does on *other* analogies, and then you will test it out on some analogies of your own devising.
* First, test out and read over the two helper functions at the top of the file. One checks if all of the words in a list are in the model provided. The other shows how to call word2vec's **most\_similar** function.
* Write a function generate\_analogy(word1, word2, word3, model) This function should ask word2vec to solve the analogy  
   word1:word :: word3:?  
  using the word2vec model *model*. It should do this by returning the *best* out of the list of the 100-best results for the appropriate call to **most\_similar** Here, you'll need to adapt the “king - man + woman = queen” example.
* Find two analogies beyond those from class where this works well -- and two where it doesn't work well -- and be sure to include those in your file (in the comment portion at the bottom of the file)
* Next, Write a function check\_analogy(word1, word2, word3, word4, model) This function should ask word2vec to grade or check the analogy  
   word1:word :: word3:word4  
  again using the word2vec model *model*. It should do this by returning a score based on where (or if) word4 appears in the appropriate call to **most\_similar** Specifically,
* It should determine where word4 appears in the top 100 (use topn=100) most-similar words
* If it \_doens't\_ appear in the top-100, it should give a score of 0
* If it \_does\_ appear, it should give a score between 1 and 100: the distance from the \_far\_ end of the list. Thus, a score of 100 means a perfect score. A score of 1 means that word4 was the 100th in the list (index 99) Try it out:
  + check\_analogy( "man", "king", "woman", "queen", m ) -> 100
  + check\_analogy( "woman", "man", "bicycle", "fish", m ) -> 0
  + check\_analogy( "woman", "man", "bicycle", "pedestrian", m ) -> 96
* To wrap up, create at least five analogies that perform at a variety of levels of "goodness" based on the check\_analogy scoring criterion -- share those (and any additional analysis) with us there in your file!

**Problem 3: Automatic rewording: *the paraphraser***

[50 pts; EC for anything above and beyond!]

* This problem asks you to run/write code in the file **hw6pr3.py**
* In this problem, you will use word2vec to automatically replace words with “similar” words. A very naive starting function is already provided…

(1) **Try** paraphrase\_sentence as it stands (it's quite bad...) E.g.,

**Try**: paraphrase\_sentence("Don't stop thinking about tomorrow!", m)

**Result**: ['Did', "n't", 'stopped', 'Thinking', 'just', 'tonight']

First, **change this** so that it returns (not prints) a string (the paraphrased sentence),

rather than the starter code it currently has (it prints a list) Thus, after the change:

**Try**: paraphrase\_sentence("Don't stop thinking about tomorrow!", m)

**Result**: "Did n't stopped Thinking just tonight" (as a return value)

* But this function, paraphrase\_sentence is bad, in part, because words are close to variants of *themselves*, e.g.,

+ stop is close to stopped

+ thinking is close to Thinking

(2) So, your task is to **add at least three things** that improve this performance (though it will necessarily still be far from perfect!) Choose at least one of these two ideas to implement:

#1: Either check for a different-first-letter OR use lemmatize to check if two words have the same stem/root - and \_don't\_ use that one! Checking the first letters is a bit easier, since lemmatize requires a part of speech (which requires context…)

+ The idea is to go past the first entries of the similarity list if they're *too* similar

#2: Use part-of-speech tagging or some other linguistic/syntactic reasoning to ensure that the similar word chose could be the same part of speech...

Then, choose two more ideas that use NLTK, TextBlob, or Python strings -- either to guard against bad substitutions OR to create specific substitutions you'd like, e.g., just some ideas:

+ the replacement word can't have the same first letter as the original

+ the replacement word is as long as possible (up to some score cutoff)

+ the replacement word is as \_short\_ as possible (again, up to some score cutoff...)

+ replace things with their antonyms some or all of the time

+ use spelling correction or translation in TextBlob in some cool way

+ use as many words as possible with the letter 'z' in them!

+ don't use the letter 'e' at all…

Or any others paraphrasing-styles you might like to consider!

(3) Share at least 4 examples of input/output sentence pairs that your paraphraser creates -- be sure to include at least one "very successful" one and at least one "very unsuccessful" ones

(4) Create a function paraphrase\_file that opens a plain-text file, reads its contents, tokenizes it into sentences, paraphrases all of the sentences, and writes out a new file containing the full, paraphrased contents with the word paraphrased in its name, e.g.,

+ paraphrase\_file( "test.txt", model ) should write out a file named "test\_paraphrased.txt" with paraphrased contents...

(5) Be sure to include an example file, both the input and your paraphraser's output -- and make a comment on what you chose and how it did!

(**Optional EC**) For extra-credit (up to +5 pts or more)

[**+2**] write a function that takes in a sentence, converts it (by calling the function above) and then compares the sentiment score (the polarity and/or subjectivity) before and after the paraphrasing

[**+3** or more beyond this] create another function that tries to create the most-positive or most-negative or most-subjective or least-subjective -- be sure to describe what your function does and share a couple of examples of its input/output...

**Problem 4: Automating Metacritic (or Rotten Tomatoes):**

**Calculating Movie Review *Sentiment***

[50 pts; and, EC for anything above and beyond!]

* This problem asks you to run/write code in the file **hw6pr4.py**
* For this problem, you'll be building a classifier for how positive/negative movie reviews are -- similar in spirit to Stanford's [Sentiment Analysis Project](http://nlp.stanford.edu/sentiment/index.html)
* To get started, run these commands to download the sourcetexts (corpora) needed for thse two machine-learning NLP examples:
  + import nltk
  + nltk.download('names')
  + nltk.download('movie\_reviews')
  + nltk.download('opinion\_lexicon')
* From there, you should start with the [hw6pr4.py starter file](https://drive.google.com/open?id=0BwPWh-3AmiLxdFM0SjRncE44REE) and
  + (1) read over, try out, and get to understand the gender-based name-classifier that we went over on 3/20 as an example
  + (2) read over, try out, and then improve the movie-review classifier for which there's a start in that same hw6pr4.py file. The challenge is to get the best performance you can on the test set. But there are a couple of catches:
    - Like in the names example, you should be careful about how you use the test set; use only the devtest set for analysis, to avoid overfitting!
    - You should use at least one feature from the TextBlob library (there are lots there to try out, e.g, parts of speech, sentence lengths or number of sentences, sentiment and subjectivity scores from there!)
    - For this problem, there’s no a limit to the number of features you use -- but beware that if you use *too* many features, you may start to add features that are specific to nuances of the devtest set, which won’t necessarily generalize well to the test set.
    - For this problem, the starter code doesn’t give you a way to visualize the errors your classifier is making. (This is the TODO line near the bottom of the file). Since that will be an important part of coming up with new features, though, you should write a routine -- again, modeled from the name example -- that shows features of mislabeled reviews. You can look up any review with movie\_review.raw(fileid), as well.
    - Extra-credit create a *graphical* summary of the features and their relationship with the review sentiment, e.g., scatterplot the numerical results from some of your features and then use the color of each scatterplot point to indicate if it was a pos. review or a negative one.
    - Also, you might try other ML algorithms, e.g., random forests.
  + At the bottom of the file, reflect on what features you tried, what worked, what didn’t, and how your classifier worked overall, both on the devtest and test sets! (Be sure to randomize your results by changing/removing the "seed" value to the random number generator.)
  + Anything over 60% (on the test set) is solid and anything approaching 70% is excellent (I think 70% is our record so far…). If you enjoyed working on the challenge of NLP, you could certainly consider it for a final project option…
  + More on final projects next week. Good luck with hw#6!
* **Warning!** Be sure to include your hw6pr4.py in your hw6.zip file when you submit!

**Problem 5: Predicting Product Reviews**

[??? pts]

* This problem asks you to run/write code in the file **hw6pr5.py**

Download the starter file [hw6pr5.py](https://github.com/ScriptingBeyondCS/CS-35/tree/master/week_6) and [app\_reviews.json](https://github.com/ScriptingBeyondCS/CS-35/tree/master/week_6). For this problem, you'll be building a RandomForestRegressor to analyze Android app reviews. 7,500 app reviews are stored in the provided **app\_reviews.json** file. Open this now and examine the contents of each review. Your task is to use this data to create a model that predicts the overall rating of an app (1.0 - 5.0) based on the other aspects of the review.

* + - Like in the names example, you should be careful about how you use the test set; use only the devtest set for analysis, to avoid overfitting!
    - Try to get above a 0.45 score for the ***test data*** (0.54 is the highest we've seen)!
    - Print the percentage of scores for the test data that were within 1 star and 0.5 stars
    - Use TextBlob to analyze the 'reviewText' and 'summary' categories like in the previous problem. Using the opinion set from last problem is encouraged, but also use some words that may not be positive or negative in general, but have a connotation in the context of app reviews (such as "freeze" or "works").
    - Take advantage of the other data features as well!
    - In addition to manipulating features, you can use cross validation to determine the best depth and number of estimators for your RFRegressor.
    - *The top (10?) testing scores in the class will receive extra credit!*
    - At the bottom of the file, reflect on what features you tried, what worked, what didn’t, and how your classifier worked overall, both on the devtest and test sets! (Be sure to randomize your results by changing/removing the "seed" value to the random number generator.)
    - It is highly encouraged that you generate a visual representation (some kind of print statement or graph) of your error (**on devtest**) to analyze what features are useful

**Extra-credit: Showing off your results…**

[up to +5 pts extra-credit...]

* As with each week, you're invited to include both your source code and a short write-up of one of the week's problems within your GitHub repo(s). Images and other visuals, of course, are welcome. If you do this, let us know (and provide a direct link :-)