# CS 2230 Data Structures & Algorithms

# Final Project: Word Similarity

60 points

## Learning objectives

- Design, implement, and test an application based on a written specification
- Choose appropriate ADTs and efficient data structures for various tasks
- Use version control to collaborate on a coding project with another person

Unlike the previous programming assignments, we provide specification only at the level of the required behavior of the program, as opposed to at the level of the code or interfaces. It is your job to decide on the classes, interfaces, data structures, and algorithms to use to implement the program.

## Submissions

- April 24, 11:59pm: Milestone 1 in GitHub (no slip days)
  - benchmark: Finished Part 1
  - PROGRESS_REPORT_APRIL24.txt
- May 1, 11:59pm: Milestone 2 in GitHub (no slip days)
  - benchmark: finished Part 3, and draft of Part 4's written answers
  - PROGRESS_REPORT_MAY1.txt
- May 4, 11:59pm: Final version in GitHub (up to 2 slip days if at least 1 partner has them)
  - Finished all Parts

Wherever you see a  that means to write something in your project report. Name the file `project.pdf` and include it in the base of your GitHub repository.

Wherever you see a  that means to write some more Java code. Make sure files you create go within the `edu.uiowa.cs.similarity` package.

See Checklist.pdf for all of these deliverables in a list.

## Clone your repository

We will create a repository for your team. You can find it at

https://github.uiowa.edu/cs2230-fa17/project-team-NUM where NUM is an integer between 1 and 99, found from your Project Pair in ICON.

The course staff members also have read/write access to your repository.
There is nothing in your repository to begin with.

### First, have one partner do these steps

1. In NetBeans, clone https://github.uiowa.edu/cs2230-assignments/semantic-similarity.git
2. Team | Remote | Push...
3. Specify a Git Repository Location

**Remote Name:** mine
**Repository URL:** https://github.uiowa.edu/cs2230-sp18/project-team-NUM

- replace NUM with your team number

**Username:** your hawkid
**Password:** your hawkid password

### Second, the other partner can get the code directly

In NetBeans, clone https://github.uiowa.edu/cs2230-sp18/project-team-NUM

- replace NUM with your team number

## Introduction

In this project, you will write an application that understands the *semantic similarity*, or closeness of meaning, between two words. For example, the semantic similarity of "car" and "vehicle" is high, while the semantic similarity of "car" and "flower" is low. Semantic similarity has many uses, including finding synonyms, inferring sentiment, and inferring topic.

To compute the semantic similarity of two words, you will first compute each word's *semantic descriptor vector*. Then you will compute the closeness of two vectors using a similarity measure, such as cosine similarity or Euclidean distance.

### Semantic descriptor vector

Given a text with n unique words denoted by ($w_1$, $w_2$, ..., $w_n$) and a word w, let $desc_w$ be the semantic descriptor vector of w computed using the text. $desc_w$ is an n-dimensional vector. The i-th coordinate of $desc_w$ is the number of sentences in which both w and $w_i$ occur.

For example, suppose we are given the following text (the opening of Notes from the Underground by Fyodor Dostoyevsky, translated by Constance Garnett):

*I am a sick man. I am a spiteful man. I am an unattractive man. I believe my liver is diseased.*

*However, I know nothing at all about my disease, and do not know for certain what ails me.*

For this text, the dimensions of the semantic descriptor vectors will be

[I, am, a, sick, man, spiteful, an, unattractive, believe, my, liver, is, diseased, However, know, nothing, at, all, about, disease, and, do, not, for, certain, what, ails, me]

That is, 1 dimension for each unique word of the text.

The word "man" appears in the first three sentences. Its semantic descriptor vector would be:

[I=3, am=3, a=2, sick=1, man=0, spiteful=1, an=1, unattractive=1, believe=0, my=0, liver=0, is=0, diseased=0, However=0, know=0, nothing=0, at=0, all=0, about=0, disease=0, and=0, do=0, not=0, for=0, certain=0, what=0, ails=0, me=0]

The word "liver" occurs in the fourth sentence, so its semantic descriptor vector is:

[I=1, am=0, a=0, sick=0, man=0, spiteful=0, an=0, unattractive=0, believe=1, my=1, liver=0, is=1, diseased=1, However=0, know=0, nothing=0, at=0, all=0, about=0, disease=0, and=0, do=0, not=0, for=0, certain=0, what=0, ails=0, me=0]

## Cosine similarity

The cosine similarity between two vectors $u = (u_1, u_2, \ldots, u_N)$ and $v = (v_1, v_2, \ldots, v_N)$ is defined as:

$$sim(u,v) = \frac{u \cdot v}{|u||v|} = \frac{\sum_{i=1}^{N} u_i v_i}{\sqrt{(\sum_{i=1}^{N} u_i^2)(\sum_{i=1}^{N} v_i^2)}}$$

For example, the cosine similarity of "man" and "liver", given the semantic descriptors above, is

$$sim(v_{man}, v_{liver}) = \frac{3}{\sqrt{26 * 5}} = 0.2631 \ldots$$

## Indexing versus queries

This project highlights an important aspect of database systems. It often takes a long time (as in minutes or hours or days) to ***index*** a dataset. By index, we mean the database ingests the raw data and sorts and organizes it in a clever way. A familiar analogy is if you had to build the index at the back of your textbook that maps every vocabulary word to its page numbers - it takes a lot of computation to build this index, but it helps readers later. The goal of indexing is to make queries fast. Fast queries make the user happy!

## Getting started: Run the Main program

We've given you a bare bones starter file Main.java, containing the main() method and some basic input processing. Run the file. You'll see a single bracket in the console:

```
>
```

This bracket is known as a ***command prompt***. The console is waiting for a command from the user. Type `help` and press ENTER. It will print the menu of supported commands.

## Part 1: Processing and cleaning input (5 points)

To process a text file into useful semantic descriptor vectors, you need to 1) split the text into sentences (lists of words) and 2) clean up the words.

### Adding a command to index a file

Write code for the `index FILE` command.

Add the command to the while loop, such that when the user enters the command followed by a file path, the console prints "Indexing FILENAME". Here is a sample interaction.

```
> index C:\Users\me\similarity\cleanup_test.txt
Indexing C:\Users\me\similarity\cleanup_test.txt
> help
Supported commands:
help - Print the supported commands
quit - Quit this program
index FILE - Read in and index the file given by FILE
```

**For every command you add to your program:** you also need to add a help message to the printMenu() method. The help message should include (i) the command, (ii) its arguments, and (iii) a description. The exact formatting is not important as long as it has those 3 components.

You'll do more with the `index` command in the next steps.

### Reading text files into sentences

Write code to read the text file into sentences.

You should assume that the following punctuation always separates sentences: ".", "!", "?", and that is the only punctuation that separates sentences. You should assume that the only the following punctuation is present in the texts: [’,’, ’--’, ’:’, ’;’, ’"’, "’"]

We recommend that you use Scanner (https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html ) to read the file sentence by sentence. It allows you to specify a regular expression to use as a delimiter (see the Fish example in the documentation).

Now, the interaction looks like

```
1       > index C:\Users\me\similarity\cleanup_test.txt
2       Indexing C:\Users\me\similarity\cleanup_test.txt
3       >
```

Except that after line 2 is printed and before line 3 is printed, your program creates a list of the sentences in that file.

## Cleaning up the words

Write code to clean up the words.

Small differences in words will cause there to be lots of "unique" words that we wouldn't really consider unique. First, **capitalization**: if we don't do any cleanup then "Man" and "man" will be considered different words. You should convert all words to lower case.

Second, **root words**: if we don't do any cleanup then "glass" and "glasses" will be considered different words. You should find the roots of all words by using "stemming". To do so, use the PorterStemmer (https://opennlp.apache.org/docs/1.8.3/apidocs/opennlp-tools/opennlp/tools/stemmer/PorterStemmer.html ) from OpenNLP. See Appendix A for setting up OpenNLP.

Another problem is that common words (such as "a", "the", "is") may not add much information to our vectors. We call these words ***stop words***. We have provided a list of stop words in `stopword.txt`. You should remove any word that appears in file.

## Testing

We've provided a file `cleanup_test.txt` that you can try your input processing on. When we ran our input processing on this file, we got the following sentences.

Sentences
[[look, glum, night-cap], [feel, littl, breez], [ah], [whatev, mai, sai, good, aliv, dear, amédé]]
Num sentences
4

Your result may differ slightly but should certainly have words in the correct sentence, lower case (e.g. amédé not Amédé), no stop words (e.g., you), and stemmed words (e.g., littl not little).

## Adding a command to print the sentences

The upcoming Parts will require your program to print other information about the text. To control what information is printed, you will use commands.

Add a command to print the sentences and the number of sentences. The particular formatting is not important, as long as we can tell what words are in each sentence and how many sentences there are.

Here is are two example interactions up to this point.

*Example 1*

```
1        > index C:\Users\me\similarity\cleanup_test.txt
2        Indexing C:\Users\me\similarity\cleanup_test.txt
3        > sentences
4        [[look, glum, night-cap], [feel, littl, breez], [ah],
   [whatev, mai, sai, good, aliv, dear, amédé]]
5        Num sentences
6        4
7        > quit
```

**Important:** After line 2 is printed and before line 3, the program has already internally computed the sentences! The `sentences` command is only for printing out that list.

*Example 2*

```
> sentences
[]
Num sentences
0
>
```

In this example, there were no sentences because we haven't indexed any files.

## Part 2: build the semantic descriptor vectors (10 points)

Write more code that takes the sentences you generated from Part 1 and generate a semantic descriptor vector for each unique word.

You can use the provided file `vector_test.txt` that contains the text from the section "Semantic descriptor vector" to see if your program produces the same vectors as the example (except for *all* unique words, not just "man" and "liver").

**Change your index command** so that it generates the vectors before finishing.

### Command for printing vectors

Add another command to your program. This command should print every unique word along with its semantic descriptor vector.

You can pick the formatting of this output, as long as it is clear what the vector is for each word. **There are many 0's, so you may choose not to print the entries in the vectors that are 0, depending on what is convenient for you.**

Here is an example interaction

```
1       > index cleanup_test.txt
2       Indexing cleanup_test.txt
3       > vectors
4       {night-cap={{look=1.0, glum=1.0}}, feel={{breez=1.0,
   littl=1.0}}, amédé={{mai=1.0, aliv=1.0, sai=1.0, dear=1.0,
   whatev=1.0, good=1.0}}, littl={{breez=1.0, feel=1.0}},
   look={{night-cap=1.0, glum=1.0}}, glum={{night-cap=1.0,
   look=1.0}}, whatev={{mai=1.0, aliv=1.0, amédé=1.0, sai=1.0,
   dear=1.0, good=1.0}}, good={{mai=1.0, aliv=1.0, amédé=1.0,
   sai=1.0, dear=1.0, whatev=1.0}}, breez={{feel=1.0, littl=1.0}},
   mai={{aliv=1.0, amédé=1.0, sai=1.0, dear=1.0, whatev=1.0,
   good=1.0}}, aliv=D{{mai=1.0, amédé=1.0, sai=1.0, dear=1.0,
   whatev=1.0, good=1.0}}, sai={{mai=1.0, aliv=1.0, amédé=1.0,
   dear=1.0, whatev=1.0, good=1.0}}, dear={{mai=1.0, aliv=1.0,
   amédé=1.0, sai=1.0, whatev=1.0, good=1.0}}}
5       >
```

**Important:** After line 2 is printed and before line 3, the program has already internally computed the sentences and then the vectors! Like the `sentences` command, the `vectors` command only does the printing.

### Tips

- Don't write all your code in the Main class! For example, we **highly recommend** that you define a class to represent a vector. You need to decide how to represent the vector in a data structure. To future proof your code, you might also consider a vector interface, so that you can try swapping in different implementations of vector.

## Part 2b: Update vectors with multiple files (10 points)

It should be possible to improve the accuracy of meaning of our word vectors with more data!

Make sure that your program works when you want to index multiple files. When the index command is called on a second (or third or fourth...) file, the behavior is to add the new sentences and update the vectors.

Here is an example interaction:

```
> index cleanup_test.txt
Indexing cleanup_test.txt
> sentences
[[look, glum, night-cap], [feel, littl, breez], [ah], [whatev, mai, sai, good, aliv, dear, amédé]]
Num sentences
4
> index vector_test.txt
Indexing vector_test.txt
> vectors
{night-cap={{look=2.0, glum=2.0}}, feel={{breez=2.0, littl=2.0}}, amédé={{mai=2.0, aliv=2.0, sai=2.0, dear=2.0, whatev=2.0, good=2.0}}, littl={{breez=2.0, feel=2.0}}, look={{night-cap=2.0, glum=2.0}}, glum={{night-cap=2.0, look=2.0}}, whatev={{mai=2.0, aliv=2.0, amédé=2.0, sai=2.0, dear=2.0, good=2.0}}, good={{mai=2.0, aliv=2.0, amédé=2.0, sai=2.0, dear=2.0, whatev=2.0}}, breez={{feel=2.0, littl=2.0}}, mai={{aliv=2.0, amédé=2.0, sai=2.0, dear=2.0, whatev=2.0, good=2.0}}, aliv={{mai=2.0, amédé=2.0, sai=2.0, dear=2.0, whatev=2.0, good=2.0}}, sai={{mai=2.0, aliv=2.0, amédé=2.0, dear=2.0, whatev=2.0, good=2.0}}, dear={{mai=2.0, aliv=2.0, amédé=2.0, sai=2.0, whatev=2.0, good=2.0}}}
```

Note that for the sake of example we indexed the same file twice, but the intent is to index lots of different files.

## Part 3: Top-J similar words (10 points)

Write code to compute the Top-J similar words to a given word.

Time to finally use the semantic descriptor vectors for something interesting! In this Part, you will implement a Top-J similar words query. That is, given a text, a query word Q, and an integer J, your program will print the J most similar words to Q.

To compute the similarity, you'll need to create a method that calculates the cosine similarity of two semantic descriptor vectors (see Cosine similarity). The mathematical calculation requires real numbers, so be sure that this method returns a `double`.

### Command and output for TopJ

Your program needs another command that takes two arguments: a word `Q` and an integer `J`. The program will compute and print the `J` most similar words to Q, along with their similarity score. Or, if there is no vector for Q (because Q did not appear in the text), then prints an error message "Cannot compute top-J similarity to Q".

Here are is an example interaction to make sure you've done these steps correctly (again, exact formatting doesn't matter, as long as the information is clearly presented). Note that the order of equally similar words may be different. You should not return the query word itself (which always has cosine similarity of 1.0).

```
> index easy_sanity_test.txt
Indexing easy_sanity_test.txt
> topj cat 5
[Pair{wolf,0.8}, Pair{tiger,0.8}, Pair{dog,0.8}, Pair{fox,0.8},
Pair{squirrel,0.8}]
> topj cat 6
[Pair{wolf,0.8}, Pair{tiger,0.8}, Pair{dog,0.8}, Pair{fox,0.8},
Pair{squirrel,0.8}, Pair{ten,0.0}]
> topj elephant 3
Cannot compute top-J similarity to elephant
> quit
```

### Tips:

- You should stem Q before looking for its vector to make sure you don't accidentally reject the query.

## Part 4: Try it on bigger data! (10 points)

Now is a good time to see how your program fares on a large text. Project Gutenberg (http://www.gutenberg.org/ ) has thousands of public domain books that you can download as plain text files. Try these (in order of size):

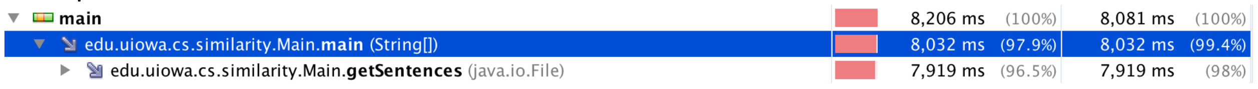
- http://www.gutenberg.org/ebooks/41
- http://www.gutenberg.org/ebooks/7178
- http://www.gutenberg.org/ebooks/2600

The time it takes to generate the semantic descriptors and run a Top-J query should be no more than ~2 minutes for the first two books. If it is taking significantly longer, see the following tips.

Try to improve your program's running time before continuing.

### Tips for slow programs

- Most words do not appear with every other word. Therefore, for really large texts, most of the entries in the vectors will be 0 (this data is called "sparse"). You should consider that fact when designing your data structure for vectors.
- Does your algorithm for the Top-J query theoretically run in O(N log N) time where N is the number of unique words in the text?
- You can see the breakdown of where time is spent by using NetBeans' profiler. Use the ***methods*** option when you *configure session*. Under **main**, you'll find how much time was spent in each method.



```
▼ main                                                       8,206 ms  (100%)    8,081 ms  (100%)
  ▼ edu.uiowa.cs.similarity.Main.main (String[])             8,032 ms  (97.9%)   8,032 ms  (99.4%)
    ▶ edu.uiowa.cs.similarity.Main.getSentences (java.io.File) 7,919 ms  (96.5%)   7,919 ms  (98%)
```

Write about the following:

Let N=the number of unique words in the text file
S=the maximum number of unique words any word appears with

a) What data structure did you finally use for vectors? What is the asymptotic memory usage of a vector? Of all of your vectors? Is this memory usage reasonable and why?
b) What algorithm did you finally use for cosine similarity? What is its asymptotic running time? Is this running time reasonable and why?
c) What algorithm did you finally use for the Top-J calculation? What is its asymptotic running time (might be in terms of J, too)? Is this running time reasonable and why?
d) What improvements did you make from your *original* code to make it run faster? Give an example of your running time measurements before and after the changes. Describe the information that informed your choices (asymptotic running time analysis, asymptotic memory analysis, and/or profiling).

## Part 5: More similarity measures (5 points)

There's many ways we can compare two vectors.

Implement two more similarity measures.

## (negative) Euclidean distance between vectors

$$sim_{euc}(v_1, v_2) = -\|v_1 - v_2\|$$

Example test:

$$w_1 = (1, 4, 1, 0, 0, 0)$$
$$w_2 = (3, 0, 0, 1, 1, 2)$$
$$sim_{euc}(w_1, w_2) = -5.1961524227$$

## (negative) Euclidean distance between normalized vectors

$$sim_{euc\ norm}(v_1, v_2) = -\left\| \frac{v_1}{|v_1|} - \frac{v_2}{|v_2|} \right\|$$

Example test:

$$w_1 = (1, 4, 1, 0, 0, 0)$$
$$w_2 = (3, 0, 0, 1, 1, 2)$$
$$sim_{euc\ norm}(w_1, w_2) = -1.27861316602$$

## Testing

Create JUnit tests for these two similarity measures. You can use the above examples as test cases. Note that `assertEquals` for doubles takes a third argument, epsilon, since comparing for exact equality of doubles is fragile. You can use epsilon=0.00001.

## Command for picking the measure

To tell your program which measure to use, add another command that can take the argument `cosine`, `euc`, or `eucnorm`. If the `measure` command is never given during the session, then the program should default to cosine similarity.

```
> topj cat 5
```

Uses cosine similarity

```
> measure euc
Similarity measure is negative euclidean distance
> topj cat 5
```

Uses negative Euclidean distance similarity

```
> measure eucnorm
Similarity measure is negative euclidean distance between norms
> topj cat 5
```

Uses negative Euclidean distance similarity between normalized vectors

**Good design**: Code that calls the similarity measures (e.g., the Top-J query code) should be reusable! Restructure your program, if needed, to promote reuse (hint: interface or generic type?). The litmus test is to ask yourself: "if I now add a *4th* similarity measure, do I need to change the Top-J code? If the answer is "yes", **then you are not done with this step**.

**Does the Top-J get better with more data?** Index one of the books. Then, run an interesting Top-J query. Then, *in the same session*, index *an additional* book (at this point your vectors reflect both books) and run the Top-J again. Report the results and comment on whether they changed.

**How does Top-J change with different measures?** Run your Top-J query above for all three similarity measures and report your findings. Discuss what is the same and different about the results for the 3 similarity measures.

## Part 6: Clustering similar words (10 points)

Next, we want to find *groups* of words that are similar to each other. The problem of ***clustering*** is to partition *n* points (in our case, a point is a semantic descriptor vector) into clusters of similar points. In k-means clustering, given a desired number of clusters, *k*, we find k ***cluster means*** (a center point) and assign each of the *n* points to the nearest mean.

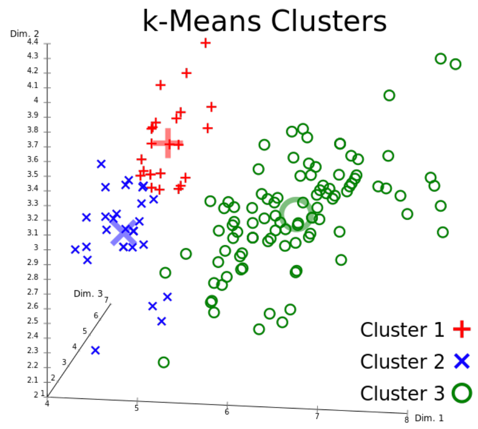


*Figure 1:* Illustration of k-means clusters where k=3 (meaning 3 clusters) and the number of dimensions is 3.

In general k-means clustering tries to figure out what the optimal choice of the cluster means is ("optimal" here means that the distance of each point from the mean is minimized). The ***k-means algorithm*** approximates a good set of means using *iterative refinement*. We start with random means, and each iteration moves the means to new places. We can stop when we have an iteration where no points switch between clusters or after a fixed number of iterations.

Here is the pseudocode, where it will always run for a fixed number of iterations `iters`.

```
means = generate k random points
for i in 0 to iters
     clusters = [[], [], [], ..., []] // k of these
     // assign points to clusters
     for p in points
          find m in means s.t. euclidean_distance(m, p) is minimized
          add p to the cluster for m
     // calculate new means
     for i in 0 to k
          means[i] = centroid of the points in clusters[i]
```

*Definition*: the centroid of a set of $r$ points $w_1, w_2, \ldots, w_r$ is $\frac{w_1+w_2+\cdots+w_r}{r}$

Implement k-means clustering using this algorithm.   To make sure the initial means are reasonable, initialize the k random points by randomly sampling k words without replacement from the database. **NOTE:** k-means only needs to be coded to use Euclidean distance, not the other similarity measures.

**Tips and examples for K-means algorithm:** https://piazza.com/class/jan1xtx1d7446x?cid=86

## Command for k-means

Add a new command that takes two arguments: an integer k  and an integer `iters` to tell the program to run k-means clustering using k clusters and running for `iters` iterations. When finished, it should print the words in each cluster. Here is an example interaction.

Run k-means clustering with k=3 and 2 iterations.

```
> index easy_sanity_test.txt
Indexing easy_sanity_test.txt
> kmeans 3 2
Cluster 0
nine,seven,two,three,eight,four,ten,five,on,
Cluster 1
six,
Cluster 2
banana,parslei,wolf,basil,appl,tomato,tiger,fox,squirrel,squash,cat,do
g,pumpkin,
>
```

Your results will vary, but you'll probably see (as above) that the categories are still mixed up. We may need to run for more iterations.

```
> kmeans 3 100
...
```

Run k-means clustering with k=3 and 100 iterations, yourself. Did it find the appropriate clusters? **Note that it might not because the clusters get stuck in a local optimum due to a poor initial random assignment of points to clusters**. However, you should be able to re-run kmeans and sometimes get the right clusters for this example, as long as the initial clusters are always different.

### Track the convergence

The k-means algorithm's iterative refinement attempts to minimize the average distance from each cluster mean to the points in the cluster. Enhance your k-means algorithm so that at the end of each iteration, it calculates the average Euclidean distance each point to its cluster's mean. Print one average for each iteration, along with the iteration number.

Run the following example again.

```
> index easy_sanity_test.txt
Indexing easy_sanity_test.txt
> kmeans 3 100
```

Using the output, create a scatterplot (using any program you like, e.g., Excel) showing the average distance vs. iteration. After how many iterations did the clustering seem to converge (i.e., not change anymore)?

## Extra credit (1 point)

Run k-means clustering on one or more of the larger texts. Run it once for at least 2 different values of k. For each value of k, create a scatterplot as above. Pick the number of iterations to be big enough that the algorithm converges. Write a description of your experiment and explain the results.

## Extra credit (3 points)

It can be hard to read the final output of k-means clustering when the number of points and/or clusters is large. To make the output more interpretable, use your Top-J code to pick J representative words for each cluster. The J representative words should be the words closest to the mean for that cluster. The idea is that J should be small, perhaps 2 to 4.

You may need to modify your Top-J code a bit so that it can be called directly using a vector (the mean) as the query instead of a word.

### Command for cluster representatives

Add a new command that takes an argument `J` to tell the program to print the final clusters using only the Top-J representatives for each cluster of the latest run of k-means.

A simple test:

```
> index easy_sanity_test.txt
Indexing easy_sanity_test.txt
> kmeans 3 100
...
> representatives 1
...
```

It should (often) print a number in one cluster, an animal in another cluster, and a food in the other cluster.

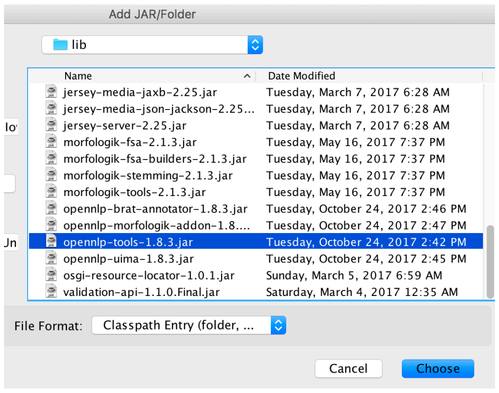
Run the k-means with representative words on one or more of the larger texts. Pick `J` between 2-4. Pick a reasonable k. Include a copy of your results and write about the interesting observations you find.

## Appendix A

### Add OpenNLP to your project

You will use OpenNLP Tools to stem (find roots of) words. Here are the steps to "install" it.

1. Download OpenNLP [https://opennlp.apache.org/download.html](https://opennlp.apache.org/download.html) and unzip it anywhere.
2. In your NetBeans project, right-click Libraries | Add JAR/folder
3. Navigate to where you unzipped the file, into the lib/ folder, and choose opennlp-tools-*.jar



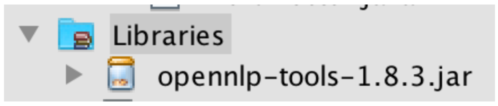
Add JAR/Folder

lib

| Name | Date Modified |
|---|---|
| jersey-media-jaxb-2.25.jar | Tuesday, March 7, 2017 6:28 AM |
| jersey-media-json-jackson-2.25… | Tuesday, March 7, 2017 6:28 AM |
| jersey-server-2.25.jar | Tuesday, March 7, 2017 6:28 AM |
| morfologik-fsa-2.1.3.jar | Tuesday, May 16, 2017 7:37 PM |
| morfologik-fsa-builders-2.1.3.jar | Tuesday, May 16, 2017 7:37 PM |
| morfologik-stemming-2.1.3.jar | Tuesday, May 16, 2017 7:37 PM |
| morfologik-tools-2.1.3.jar | Tuesday, May 16, 2017 7:37 PM |
| opennlp-brat-annotator-1.8.3.jar | Tuesday, October 24, 2017 2:46 PM |
| opennlp-morfologik-addon-1.8.… | Tuesday, October 24, 2017 2:47 PM |
| opennlp-tools-1.8.3.jar | Tuesday, October 24, 2017 2:42 PM |
| opennlp-uima-1.8.3.jar | Tuesday, October 24, 2017 2:45 PM |
| osgi-resource-locator-1.0.1.jar | Sunday, March 5, 2017 6:59 AM |
| validation-api-1.1.0.Final.jar | Saturday, March 4, 2017 12:35 AM |

File Format: Classpath Entry (folder, ...

1. You should now see that jar file under Libraries



2. You can now import the stemmer library into your Java file without any errors.

```
import opennlp.tools.stemmer.*;
```

## Acknowledgements

The concept for this assignment and some of the examples are drawn from SIGCSE.

Icon files:

- JAVA File by AomAm from the Noun Project
- Pencil by Adrien Coquet from the Noun Project