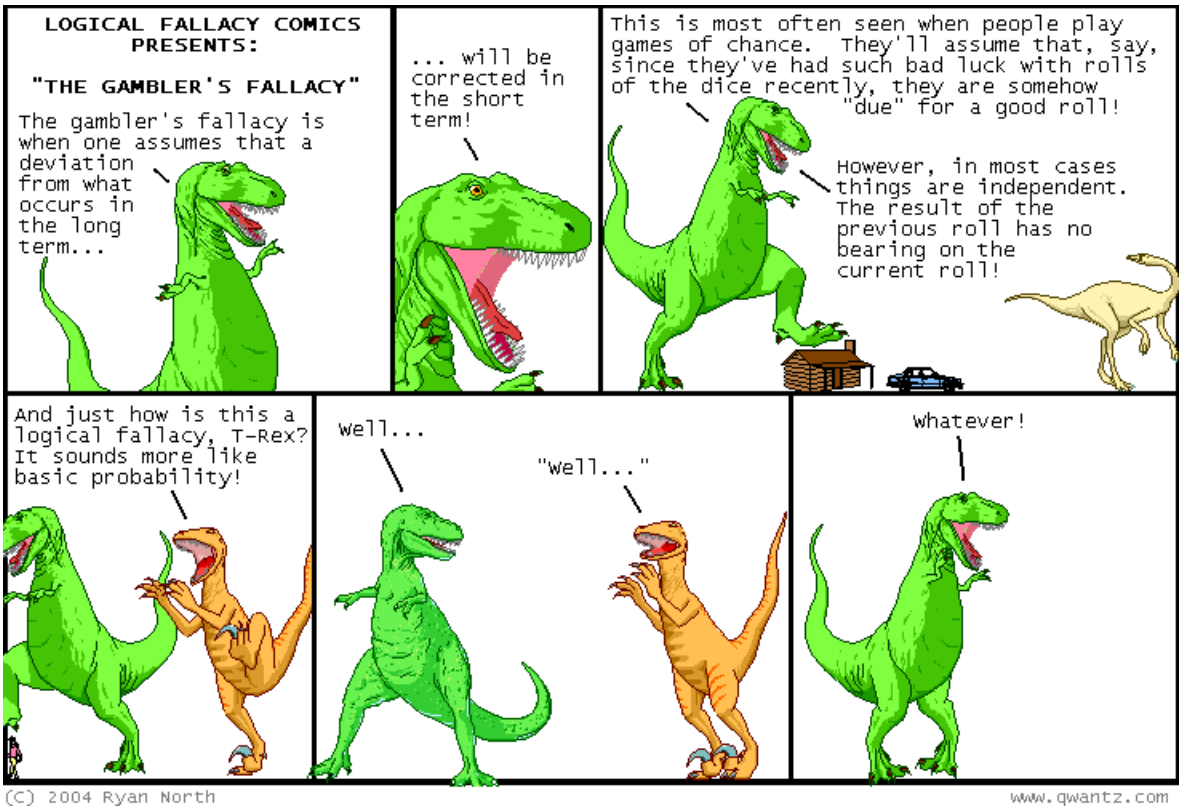
CS311 - Assignment 5
Due: Friday, April 12 by 6pm



http://www.empiricalzeal.com/2012/12/21/what-does-randomness-look-like/

For our last assignment we will are going to be coding up the k-means algorithm. I've given you a fair amount of code to deal with the nitty-gritty details so that you can focus on the algorithm itself and on evaluating different variations.

You may, and I encourage you to, work in pairs on this assignment. As always, read through the **entire** document before starting.

1 Starter Code

A starter is available for this assignment that has both code and data at:

```
/home/dkauchak/PUBLIC/cs311/assign5/
```

The starter code contains four different files:

- `kmeans.py`: This is skeleton code for a KMeans clusterer. You will be filling in the details of some of the methods and may also be adding your own. Do NOT remove or change the headers for any of the methods defined in this file.
- `kmeansGUI.py`: This is a graphical interface for visualizing your KMeans algorithm for 2-D clustering problems. You will not make any changes to this code, however, the code will not work until you have filled in all the details for your KMeans class.
- `datapoint.py`: This file contains classes for storing the data that will be clustered by your k-means algorithm. The data is stored as a sparse vector using a dictionary as a mapping from the feature/dimension to the value of that feature. You can create new data points manually by passing in a dictionary to the constructor and the class supports most of the basic operations you'll need, e.g.

```
>>> from datapoint import *
>>> point1 = DataPoint(dict(x=1, y=2))
>>> print point1
{'y': 2, 'x': 1}
>>> point2 = DataPoint(dict(y=2,x=1))
>>> print point2
{'y': 2, 'x': 1}
>>> point1 == point2
True
>>> point3 = DataPoint(dict(x=1, y=2, z=15))
>>> print point3
{'y': 2, 'x': 1, 'z': 15}
>>> point1 != point3
True
```

  Look at the code for the `DataPoint` class and make sure you understand at a high-level what functionality is available to you. In particular, you'll likely need to us the following methods:

  - `get_val`
  - `add_data_counts`
  - `divide_by_constant`
  - `cosine_distance`
  - `euclidean`

  Also in this file is a `WordDataPoint` class which you'll use to cluster documents. It extends the `DataPoint` class, so it has the same functionality and can be used anywhere a `DataPoint` is expected, however, it allows you to instantiate a data point from a list of words:

```
>>> words = "I like to eat bananas with more bananas".split()
>>> words
['I', 'like', 'to', 'eat', 'bananas', 'with', 'more', 'bananas']
>>> word_point = WordDataPoint(words)
>>> print word_point
Counts: {'bananas': 2, 'I': 1, 'to': 1, 'more': 1, 'with': 1, 'eat': 1, 'like': 1}
>>> word_point.get_val("bananas")
2
```

Finally, at the end of this file is the function `gen_random_data` that's useful for generating random data for testing your clustering algorithm on (more on this later).

- `DataReader.py`: The same DataReader file/class that you used for the NB classification assignment. We will use this to read similar text data for clustering. See the NB handout for details on how to use this class.

2 Getting to know your `DataPoint`

To get you comfortable with the `DataPoint` class, the first thing you need to do is fill in the `euclidean` method which will compute the Euclidean distance between this point and another point. For example:

```
>>> point1 = DataPoint(dict(x=2, y=3))
>>> point2 = DataPoint(dict(x=4, y=6))
>>> point3 = DataPoint(dict(x=2, y=3, z=2))
>>> point1.euclidean(point2)
# some number---work through the example by hand
>>> point2.euclidean(point1)
# the same number :)
>> point1.euclidean(point3)
# some number that is NOT zero
```

When you think you have it working, work through a few examples by hand and make sure this is working correctly. One of the keys to good programming (and to saving yourself a lot of headaches and time) is to develop incrementally, checking and testing your code as you go.

3 k-means

Now you should have a fully functional `DataPoint` class. If you haven't done so yet, make sure you look through all of the methods in that class that are available to you.

The next step is to implement your k-means clusterer. Specifically, read through the documentation then fill in the details for the following methods in `kmeans.py`:

- `__init__`
- `cluster`
- `assign_to_centers`
- `recalculate_centers`
- `get_clusters`
- `get_centers`

This shouldn't be a ton of coding, but make sure you understand what each of the functions above are supposed to be doing.

You'll need to keep track of two key things in the `KMeans` class: 1) the clusters and 2) the cluster centers. For the cluster centers, you can just use `DataPoint`s since this will allow you to do distance calculations, etc. For the clusters, I recommend that your represent each cluster as a list of `DataPoints` and then you can store all of the clusters in a single list. You may do it another way, however, representing your data this way will make your life easier since this is the format that is expected to be returned from the `get_clusters` method.

I'd recommend getting a barebones clustering algorithm working using Euclidean distance and a fixed number of iterations then go back and add handling of cosine distance and checking for convergence.

For the first pass of this you should pick random points to initialize your cluster centers.

4 Visualizing k-means

Assuming you've implemented everything in the `KMeans` class, you should now be able to construct a new `KMeans` object using some data and then cluster it using the `cluster` method. Try out some simple examples and see if you get the results.

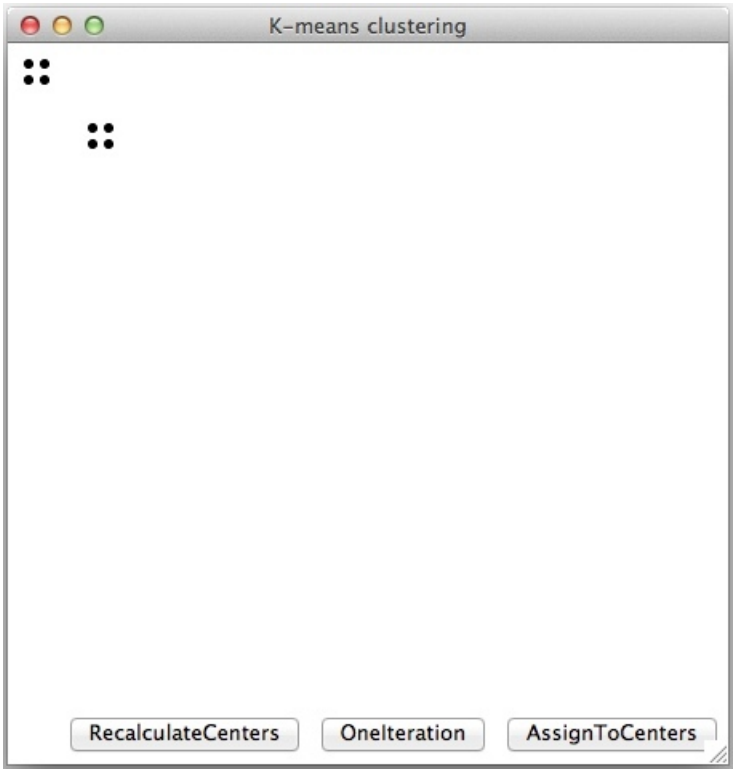
To help you with understanding how k-means works (and for debugging your implementation), i've also put together a GUI that will allow you to cluster points in two dimensions.

Try out a simple example:

```
from datapoint import *
from kmeans import *
from kmeansGUI import *

data = []

data.append(DataPoint(dict(x=1, y=1)))
data.append(DataPoint(dict(x=1, y=2)))
data.append(DataPoint(dict(x=2, y=1)))
```

```
data.append(DataPoint(dict(x=2, y=2)))
data.append(DataPoint(dict(x=5, y=5)))
data.append(DataPoint(dict(x=5, y=6)))
data.append(DataPoint(dict(x=6, y=5)))
data.append(DataPoint(dict(x=6, y=6)))

clusterer = KMeans(data, 2)

KMeansGUI(data, clusterer)
```

If you put this in a file and run this it should pop open a window that looks something like:



In the upper left are the 8 points in our data set and on the bottom are three buttons. There are two ways of running you k-means algorithm from the GUI. If you want to see each of the steps happen individually, you can alternate between clicking `AssignToCenters` and `RecalculateCenters`. This can be useful when trying to make sure your algorithm does the right thing since it allows you two run each step individually.

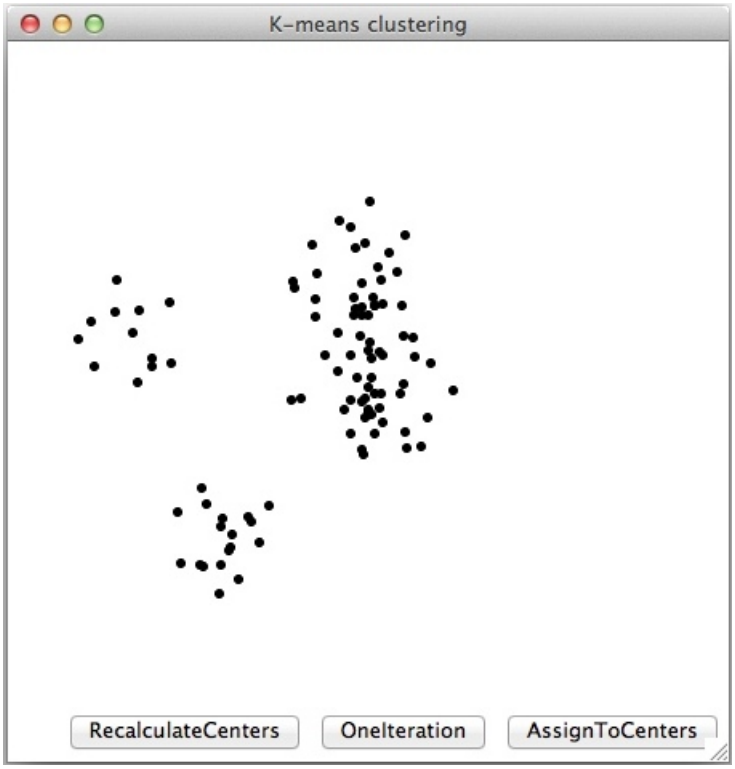
If you just want to see the algorithm work, then you can use the `OneIteration` button, which will do one iteration of the k-means algorithm (i.e. assigning to cluster centers and then recalculating centers).

This example isn't too interesting since often the clusters are right after the first iteration. To try out a more interesting example, I've provided a function in the `DataPoint.py` file that will generate

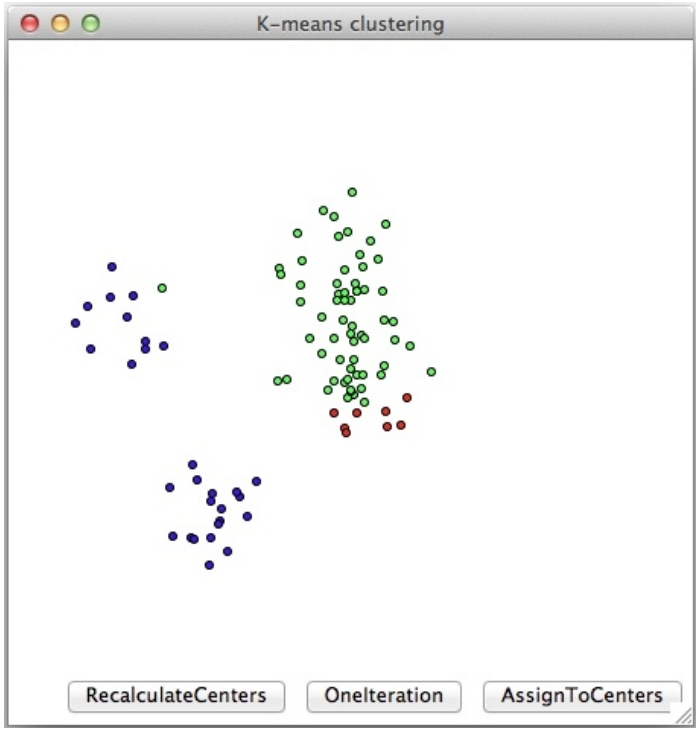
random data around some number of randomly chosen cluster centers. For example, to generate 100 points around 4 cluster centers and then display it in the GUI:

```
from datapoint import *
from kmeans import *
from kmeansGUI import *

data = gen_random_data(4, 100)

clusterer = KMeans(data, 3)

KMeansGUI(data, clusterer)
```

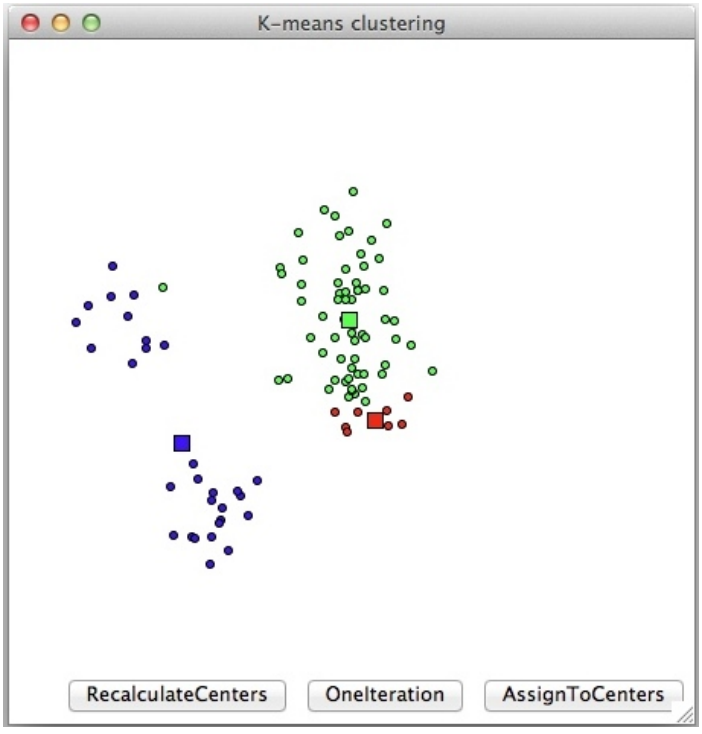
for example you might see the following (note the points are random, so you'll most likely see something different):



If I click on the "AssignToCenters" button, the points get assigned to centers and will be colored appropriately:

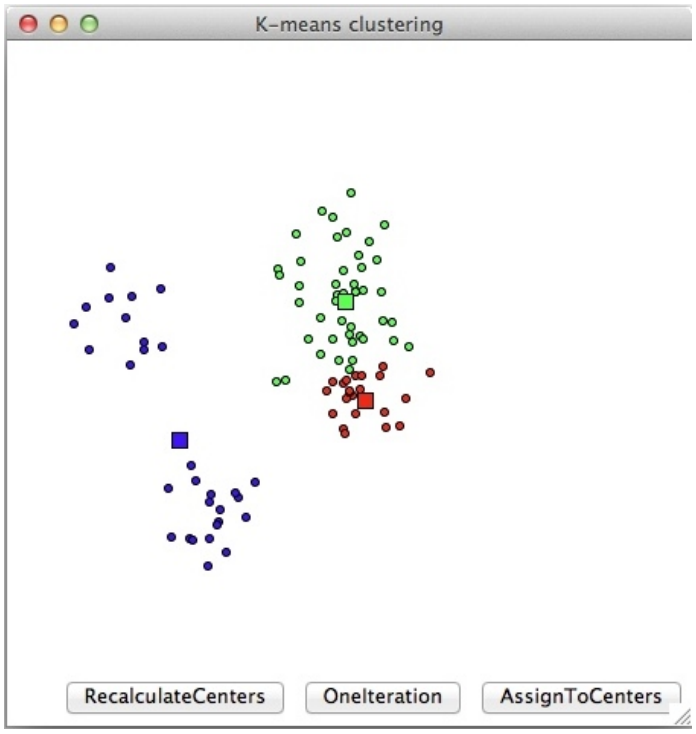


If I now click on the "RecalculateCenters" button the centers will be calculated and shown:

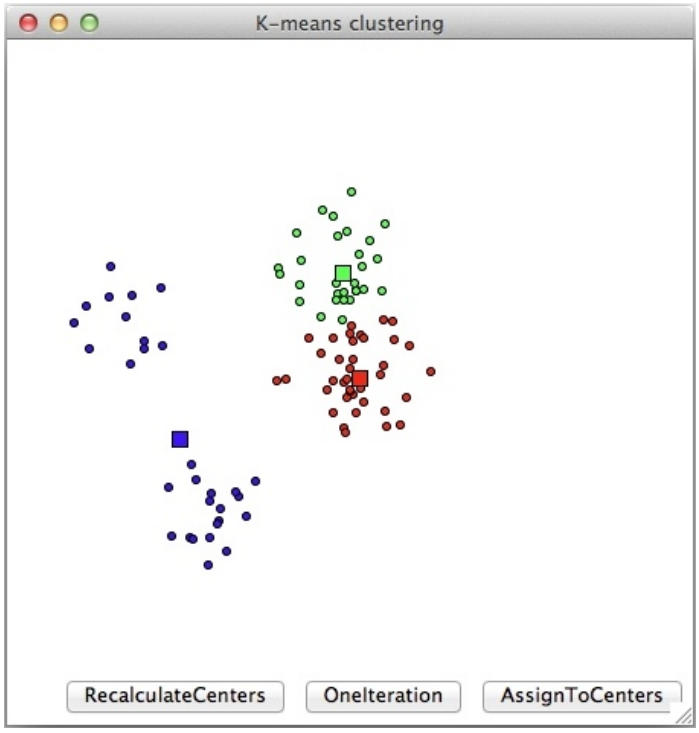


I could continue to alternate pushing these buttons to continue k-means, or I can just press the

"OneIteration" button to do a full iteration:



Eventually, if I keep pressing the button the solution converges:



5 Clustering Documents

Besides just clustering numerical points, we can also cluster "documents". For example, you can cluster the following sentences:

```
from datapoint import *
from kmeans import *

data = []

data.append(WordDataPoint("I like bananas".split()))
data.append(WordDataPoint("bananas like me".split()))
data.append(WordDataPoint("you like bananas".split()))
data.append(WordDataPoint("I hate apples".split()))
data.append(WordDataPoint("apples hate me".split()))
data.append(WordDataPoint("you hate apples".split()))

# generally use the cosine distance for words
clusterer = KMeans(data, 2, KMeans.COSINE)
clusterer.cluster()
clusters = clusterer.get_clusters()

# print out the clusters
for cluster in clusters:
  print "-" * 25

  for point in cluster:
    print point
```

Most of the time, you should see a hate/apples cluster and a like/bananas cluster (although not always, so try it a couple of times if it doesn't work).

Once you're sure that your clustering algorithm is working, you're now ready to cluster some real data. I pre-processed the movie review data to be more amenable for clustering. In particular,

1. I removed punctuation and any words that weren't alphanumeric.
2. I removed very frequent words that are not content bearing (e.g. "a" or "the").
3. I then counted all of the word occurrences and only kept the top 5% most frequent words. For clustering, rare words don't help us much since they don't help much with similarity and they can make the clustering computationally expensive.

I've put this revised data set in a file called `movies.condensed.data` and put it in the `data` directory within the assignment starter.

This file has the same format as the old movies file and so we can use the `DataReader` class to read in a the data set and then cluster it:

```
data = []

reader = DataReader("movies.condensed.data")

for label, tokens in reader:
  # pass the label to the WordData constructor to keep track of it for later use
  data.append(WordDataPoint(tokens, label=label))

# k=five clusters
clusterer = KMeans(data, 5, KMeans.COSINE)
clusters = clusterer.cluster()

for cluster in clusters:
  print "-" * 25

  for point in cluster:
    print point.get_label() + ": " + str(point)
```

There are 13,591 documents in this data set, so it may take a couple of minutes to cluster, however, it shouldn't take much longer than that if you've implemented everything correctly.

6 Evaluation

If all is working well, you should have seen 5 clusters printed out with the data points in each and their positive/negative label. But, how good are we doing?

Create a file called `eval.py` and add to it code to calculate two evaluation measures:

- Calculate the average purity of the clustering of the documents (we'll just do purity for this assignment since there are only two classes). To calculate the average purity calculate the purity of each class and then average it across classes.
- Calculate the *weighted* average purity. Calculate the purity for each clutter and then average the clusters, but weight the average based on how many points are in the cluster. So if cluster A has purity 0.5 with 10 points and cluster B has purity 0.8 with 20 points, the average purity would be 0.65, but the weighted average purity would be 0.70.

7 Experimentation

Now let's experiment! Create a file called `evaluation_results` (it can be a .txt, .doc or .pdf) and experiment with you k-means algorithm and some variations on the algorithm. I'd like you to

treat this like a mini-research paper where you analyze some variations, presents the results in a reasonable format and then interpret the results. This will be good practice for you final project.

Your paper *must* include:

- Baseline evaluation scores on the movie data set for both evaluation metrics using the basic k-means algorithm.
- Implement at least one other approach for initializing k-means and provide evaluation results on the movies data set.
- Experiment with at least one other k-mean component/variation and present results. For example:
  - Examine how the quality of the clustering improves for each iteration of the algorithm running.
  - Randomly restart many times and compare the results (e.g. worst, best and average).
  - Be creative :)
- Your writeup must include at least 2 figures/table, should be < 2 pages long and for every experiment that you present results you should include a short paragraph introducing the experiment (i.e. what you did) and then analyzing/interpreting the results.
- If you want, feel free to experiment with other data sets. I'm happy to help you preprocess them if that would be useful.

The writeup will be a non-trivial portion of your grade, so make sure you spend some time working on it. See what you can figure out about how the algorithm performs!

When you're done

When you're all done, follow the directions on the course web page for submitting your work. Make sure that your code compiles, that your files are named as specified and that all your functions have the same name and number of parameters.

If you worked with a partner, put both people's last names on the submitted directory, but only submit one copy.

What to submit

- `datapoint.py`: Your revised file with the `euclidean` method implemented.
- `kmeans.py`: Your k-means implementation.
- `eval.py`: You evaluation metric implementation.
- `evaluation_results`: Your experimentation writeup.

Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each file
- Each class and method should have a short "docstring"
- If anything is complicated, put a short note in there to help the graders out if there are any issues.

There are many possible ways to approach this problem, which makes code style and comments very important here so that the grader and I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.