Project 2

Important Information

1. Due electronically by **11:59PM, Tuesday, October 31, 2017**. One late period may be used to extend this deadline to 11:59pm on November 2, 2017. No submissions will be accepted after November 2.
2. Programming projects must be completed individually. You may discuss algorithms with others, but the coding should be done alone. You must explicitly name everyone with whom you discussed this project in the header comments of your code and in the collaboration attestation in the project quiz. Students must abide by the terms of the **Stanford Honor Code. (https://communitystandards.stanford.edu/student-conduct-process/honor-code-and-fundamental-standard)**
3. Remember to consult **Piazza (https://piazza.com/class/j66r25o2adf4vp?)**, as many common questions will be asked and answered there.
4. Prior to beginning the assignment please read the **Code Policy (https://canvas.stanford.edu/courses/68991/pages/code-policy)**.
5. Do not use scikit-learn. We want you to implement these algorithms by hand, and furthermore, scikit-learn is not installed on corn by default.
6. Pay careful attention to instructions for naming and formatting output files. You will not receive full credit if you do not follow these instructions.
7. You will need to write additional code to answer some of the quiz questions. This code does not need to be submitted, and will not be graded. The code you submit should not perform these additional tasks. Source code must run **exactly** as specified below. We will be running your code on test data on the corn servers.
8. We will not provide answers for you to use in debugging this project. We released answers for Project 1 because that project is very difficult to test and debug de novo. If you understand these algorithms, you will be able to create your own test cases to check the correctness of your code. That being said, be sure to actually write such test cases to make sure your code works. These are simple algorithms, but several students in the past have made mistakes that could easily have been caught if they had tested their code more thoroughly.

Files to Download

You can download all files required for the project, including the quiz questions and template for submitting your answers, **here (https://canvas.stanford.edu/courses/68991/files/2003039/download)**. These files are described in detail in the following sections.

Introduction

In this project you will be implementing both K-nearest neighbors (a supervised machine learning algorithm) and K-means (an unsupervised machine learning algorithm). You will be applying your algorithms to analyze gene expression microarray data, but the programs you write will be general-purpose and can be applied to any data in which each sample can be represented by a numerical vector.

Analysis of microarray data can be on the experiment level (cluster/classify experiments), at the gene level (cluster/classify genes), or both (see **biclustering (http://en.wikipedia.org/wiki/Bicluster)** if interested). For this project, you will perform k-nearest neighbors to classify microarray experiments based on gene expression. You will also perform K-means clustering on genes, clustering together genes with similar expression profiles over a multitude of experimental conditions. You will not be implementing bi-clustering for this project.

K nearest neighbors (KNN)

For the KNN algorithm, you start with a training set of data points with class labels. The idea is that if you are given an unlabeled data point, you can make a prediction about what class it belongs to based on the class of the data points in the training set that it is "closest" to. The choice of distance metric depends on the data set you are working with. We will be using Euclidean distance as our distance metric.

In this assignment we will only be working with data points assigned to two classes (positive or negative), but it is a good exercise to think about how you would approach the problem if there were 3 or more classes.

Assessing the quality of a classifier

To assess the performance accuracy of a supervised learning algorithm, we typically use an approach known as n-fold cross-validation. In this strategy we:

- Divide the training set into n groups randomly. Divide the positive set into n groups and the negative set into n groups and then combine them so that each of the final n groups have equal proportions of positive and negative samples. Think carefully about what happens when the positives and/or negatives are not perfectly divisible by n. Consider how you can keep the proportions of positive and negative samples as equal as possible across the n groups without any group being much larger than the others.
- We use 1 of the n subsets as our test set and the other n-1 as our training set. We repeat this step n times.
- We assess the test accuracy (sensitivity, specificity, or a combination of the two) by taking a weighted average over all n folds. The easiest way to do this is by summing the number of TP, FP, TN, FN over the n folds and using these totals to calculate the sensitivity, specificity or accuracy.

When dividing the data into n groups, you cannot use a deterministic partition but rather **must** divide the data randomly. This is because you can often get artifacts due to the ordering of the data. If you deterministically divided the data into n groups then there may be one test set which would drastically affect your results.

Some useful definitions:

- Sensitivity = TP/(TP+FN)
- Specificity = TN/(TN+FP)
- Accuracy = (TP+TN)/total

Where TP = number of true positives, FN = number of false negatives, FP = the number of false positives, and TN = the number of true negatives in the test set.

Data for KNN

The data set you are given for developing and testing your algorithm, as well as completing the quiz questions, contains gene expression data from cancer patients. Expression data was taken from 72 patients with acute leukemia, to see if expression alone can differentiate between acute lymphoblastic leukemia (ALL) and acute myeloid leukemia (AML). This is an important task because clinical symptoms are similar for these two diseases, but the response to treatment is quite variable between the two cancer types. For this assignment we will consider the ALL patients to be in the positive class and AML patients to be in the negative class.

Normalized gene expression data can be found in ALL.dat for the 44 ALL patients and in AML.dat for the 28 AML patients. These files are tab-delimited, with each column representing a different patient and each row representing a different gene. The genes are in the same order in both of these files. A description of the 7129 genes can be found in human_genes.txt.

The experiments that were used to generate this data are described in detail in the paper:

T.R. Golub, D.K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J.P. Mesirov, H. Coller, M. Loh, J.R. Downing, M.A. Caligiuri, C.D. Bloomfield, and E.S. Lander, D. **Molecular Classification of Cancer: Class Discovery and Class Prediction by Gene Expression (https://canvas.stanford.edu/courses/68991/files/2003084/download)** . Science. 99:531-537.

Implementation notes

KNN classifier

You will implement an algorithm that, given an expression vector for an unclassified patient, will find the K "closest" expression vectors (based on Euclidean distance) from the classified patients and let them vote on what class the unclassified patient belongs to. Specifically, if p percent or more of the K neighbors are from the "positive" class, then you will classify the unknown patient as belonging to the "positive" class, otherwise you will classify it as belonging to the "negative" class. Remember that for the cancer data set you are given, we are considering ALL to be the "positive" class and AML to be the "negative" class. You will then assess the performance of your algorithm using n-fold cross validation.

n-fold cross validation

- These notes are written specifically for the case when n=4 in order to be as clear as possible, but remember that your code must work for any n. In the case that the total number of samples is not perfectly divisible by n, extra samples should be spread among all n groups such that every group has no more than one negative and one positive sample more than the others.

1. Randomize the positive data and divide it into 4 equally (or as close to equal as possible) sized sets.
2. Randomize the negative data and divide it into 4 equally (or as close to equal as possible) sized sets.
3. Pair up the positive and negative data sets so that you have 4 pairs that contain approximately equal proportions of positive and negative examples. Each pair consists of one positive data set, and one negative data set.
4. Now take three of the four pairs and combine them to make the training set. Run KNN on this training set using the remaining one (out of the four) pair as a test set (i.e. pretend you don't know the classification of the items in the test set, let your algorithm predict the class for each item, then see if the prediction was correct).
5. Repeat the previous step three more times. Each time one of the four pairs is left out as the test set, and the remaining 3 pairs are used as the training set.

- You must only perform the randomize and division steps one time at the beginning of your implementation of n-fold cross-validation.

KNN input

Your scripts are expected to work for any data set if the input files are properly formatted. You may assume that any positive and negative data files we test your code on will be formatted just like the AML.dat and ALL.dat files. Specifically, you can assume:

- the files are tab-delimited
- columns correspond to samples (for example patients) and rows correspond to the values used to represent the samples (for example gene expression values)
- the rows in each of the 2 files are in the same order
- the values have all been normalized, meaning they can be used to compare samples in a meaningful way

Your script should be run with the following command line arguments:

```
knn.py pos neg k p n
```

where *pos* is the data file for positive samples, *neg* is the data file for negative samples, *k* is the number of neighbors to be considered, *p* is the minimum fraction of neighbors needed in order to classify a sample as positive, and "n" is the number of folds for n-fold validation. To be efficient, the recommended runtime is under 20 seconds.

KNN output

After running knn and assessing performance using n-fold validation, your script will report some performance statistics. Output from your script should be written to a file called *knn.out* **exactly** in the format shown below. *k* and *n* are integers, all other float values should be rounded to exactly 2 decimal places. Please do not output any debugging statements to <stdout> or knn.out in the final submitted version of your code.

```
k: 30
p: 0.50
n: 3
accuracy: 0.45
sensitivity: 0.00
specificity: 1.00
```

Note that these numbers are made up only for the purpose of demonstrating the correct format, you should not use these values for debugging your code.

K-means

The K-means algorithm is useful when you have only unlabeled data points. You are looking for patterns in the data that would indicate that there are subsets, or clusters, of the data that may be considered similar to each other. This is an unsupervised learning algorithm because there are no "correct" classifications to which you can compare the results of your algorithm.

You usually do not know how many clusters are present in your data, or even if there are any clusters. However, as discussed in lecture, the K-means algorithm requires that you specify the number of clusters *k* in advance. There are statistical methods that can be applied to identify the optimal number of clusters in your data, but these are beyond the scope of this class. For the purposes of this assignment, you will assume that the optimal number of clusters for the data set has already been determined and you will run K-means with a pre-determined value for *k*.

K-means is just the first step in an unsupervised learning problem. Once you find clusters, you would want to investigate the key features responsible for driving the observed clustering, determine how reproducible the clustering is, and try to understand if there is a true biological reason underlying the cluster results. You will very briefly consider some of these questions while working through the quiz questions.

Data for K-means

There are 2 data sets you will use for developing and testing your algorithm, as well as completing the quiz questions. A data set consists of:

- a data file with each row corresponding to a gene and each column corresponding to one experimental condition.
- a centroid file with each row corresponding to a centroid (cluster center) and each column corresponding to one experimental condition.

Test Data

The first data set is made-up data to give you an opportunity to visualize in low dimensions what the algorithm is doing

(and to help with debugging your code).

- data file: testdata.dat contains 9 data points in 2 conditions
- centroid file: testdata_centroids.dat contains 3 centroids

Experimental Data

The second data set is experimental data generated by measuring gene expression in yeast under different conditions.

- data file: yeast.dat contains 2467 data points (gene expression measurements) in 79 conditions.
- centroid file: yeast_centroids.dat contains 3 centroids

Yeast were exposed to 79 different environmental conditions and the expression of 2467 genes was measured. For example, some of these conditions were starvation (causing the yeast to form spores), changing the sugar supply (causing the yeast to ferment rather than respire), and synchronizing the cells to force them to pass through the stages of cell division at the same time. The experiments are described in detail in the paper:

Eisen, M.B., Spellman, P.T., Brown, P.O., Botstein, D. Cluster analysis and display of genome-wide expression patterns. PNAS. 95:14863-14868.

The file yeast_experiments.txt lists the experiments to which each column corresponds.

The file yeast_gene_names.txt contains the common gene names and a short functional annotation for the 2467 genes.

Implementation notes

K-means algorithm

1. Initialize *k* centroids. If a centroid file is provided, these are the starting centroids. If not, generate *k* centroids randomly (see details for initializing centroids below).
2. For each data point, assign it to a cluster such that the Euclidean distance from the data point to the centroid is minimized.
3. For each cluster, move the centroid to be at the center of all points that belong to that cluster.
4. Iterate steps 2 and 3 until convergence or after a certain number of iterations (see details for stopping conditions below).

Initializing centroids

Centroids can be picked at random or specified by the user. If a centroid file is provided, the first *k* centroids in the file should be used. If no centroid file is provided, *k* centroids should be generated randomly. There are various strategies that can be used to generate random centroids. Any solution that meets the following criteria is reasonable:

- The centroids are generated randomly, not deterministically. If your script is run multiple times without specifying a centroid file, different centroids should be generated each time.
- The centroids must fall within the range of the data. For example, if you have 2-dimensional data and the X values of the data all fall between 0 and 10 and the Y values of the data all fall between 5 and 8, the random centroids must also have X and Y values within these respective ranges.

While in general the numbering/naming of the clusters is arbitrary, in order to simplify grading of this assignment **you must number the clusters based on the order they are entered in the centroid file, starting at 1**. The first centroid in the file will be centroid 1, the 2nd will be centroid 2, etc. When centroids are generated randomly, they should be numbered 1 to k.

Stopping conditions

- The algorithm has converged when no data points change clusters.

- The *max.it* argument specifies the maximum number of iterations we allow before cutting off if convergence is not achieved. It allows you to get "pretty good" results in cases when it is taking a long time to converge.
- Do not rely on the maximum iteration parameter to stop the program. If k-means converges before the maximum number of iterations is reached, it should stop.

K-means input

Your scripts are expected to work for any data set if the input files are properly formatted. You may assume that any data or centroid files we test your code on will be formatted just like the test and experimental files provided.

Your script should be run with the following command line arguments:

```
kmeans.py k expression.dat max.it centroids.txt
```

where *k* is the number of centroids, *expression.dat* is the tab delimited file containing expression data, *max.it* is the maximum number of iterations allowed, and *centroids.txt* is an optional parameter specifying a file from which initial centroids should be read. To be efficient, the recommended runtime is under 20 seconds.

K-means output

When your program reaches a stopping condition, cluster assignments should be written to a tab-delimited file called *kmeans.out*. The first column contains the genes listed by index as ordered in the input data file (starting at 1). The first gene must be gene 1, the 2nd gene 2, etc. The second column should contain the number of the cluster the gene is assigned to.

For example (K=2):

```
1    1
2    1
3    1
4    1
5    2
6    2
7    2
```

and the number of iterations completed should be written to <stdout> as shown below:

```
iterations: 45
```

Iteration 0 is when the data points are first assigned to their nearest centroid. Iteration 1 is the first time the centroids are updated based on the initial gene assignments. If the assignments do not change from the original ones after the first centroid update, then you would report *iterations: 1*. If the assignments change once after the first centroid move and then converge, you would report *iterations: 2*, etc.

Quiz

You will run your code to answer the questions in the quiz on Canvas. In order to answer some questions you may have to write additional functions. Any code you write for the sole purpose of answering the quiz questions should not be submitted.

Submission Instructions

1. Take the quiz on Canvas.
2. Submit your scripts knn.py and kmeans.py. Remember, we expect well-commented code. If you have created re-usable utility modules that are required to run these scripts, you should submit these as well.

Please be sure you have followed all of the project directives (pay particular attention to the input and output instructions) as a portion of your grade is based on your compliance with these directives.

This is the same process you used to submit project 1, but detailed steps are provided below in case you need to refer to them.

Log in to a corn machine:

```
ssh sunetid@corn.stanford.edu (mailto:sunetid@cardinal.stanford.edu)
```

Place all of your relevant files into one directory. Do not include irrelevant files; please only include those listed above under submission contents. In particular, do not submit the data.

cd into your submit folder.

```
cd ~/biomedin214/p2/submit
```

Run the class submission script:

```
/usr/class/biomedin214/bin/submit p2
```

Note the space before p2. Be sure to run the script from within your submission folder.

You may submit multiple times; simply re-run the script. Each new submission overwrites the previous submission. Your submission date will be the final submission received, and late periods will be charged accordingly.

The submit script will run a format check on your code to make sure it runs properly and output files conform to project specifications. If you'd like to run this format check without submitting, run:

```
/usr/class/biomedin214/bin/check p2
```