Kenton Carrier
CS 460G
Homework 2

User-Based Collaborative Filtering

For this portion of the assignment, I made several functions that facilitate the KNN method. I first made a method to read the provided datasets and and format them into arrays. Since the final column is inconsequential to our calculations, I chose to remove it in the readFile function to save memory. I decided to use Euclidean distance for my measure of similarity because it was the easiest to implement. For this I had to import the math library for the square root function.

Using these utility functions, I broke the remaining process of KNN prediction into three stages: finding neighbors, predicting individual rating, and applying predictions to the whole dataset. The first function loops through each row in the training data and calculates the Euclidean distance of the parameter row to each row in the training set. Once this is finished, the function sorts the rows by distance and returns the parameter k number of rows with the shortest distances, the “neighbors”. The prediction function uses the neighbors functions to get the k nearest neighbors to a single row; using the neighbors, it creates an array of only their associated ratings and finds the maximum of the set after sorting by the count of the rating’s occurances in the set. This is the returned rating prediction. The kNearest function facilitates predicting on the entire test dataset by looping through each row of the test data and appending the prediction to an array. The array of results is returned from kNearest for error and accuracy calculations.

My mean square error function implements the equation $\frac{1}{n}\Sigma_{i=1}^{n}(Y_i - \widehat{Y}_i)^2$. After running the kNearest function to generate an array of predictions, I pass this array and the test data into my MSE function to get my mean squared error for an group of predictions. I had some challenges when creating this function since the run time for the kNearest function on the entire dataset is so long. Since I had to wait for kNearest to run before I could run MSE, there were several times that I waited almost 10 minutes to find a simple syntax error.

I ended up calculating that the mean squared error for K = 3 with my predictions to be approximately 2.545. This varied from each run, but it usually was around this value each time.

Cross Validation

I decided to use a cross-fold method for cross-validation to preserve run time. Dividing the training set into folds divided my runtime by the number of folds in almost every test. I wrote two functions to facilitate this process: one to split the data into random folds and one to run my KNN method on each fold. The splitData function creates an array to be populated with the arrays representing each fold and an array that is a copy of the provided dataset. The fold size is calculated by dividing the length of the dataset by the number of desired folds, k, provided as a parameter and casting the result as an integer for rounding. Then, the function loops k times, creating an array for the fold each loop. While the size of the new fold array is less than the calculated fold size, the function generates a random index in the dataset, appends the row from the copy dataset at that index to the fold array, and removes that row from the copy dataset. Once the fold array's size has reached the calculated fold size, the loop appends the fold to the array populated with each fold. Once all loops have finished, the resulting array of folds is returned.

The second function uses splitData, kNearest, and meanSquareError functions to facilitate the entire cross-validation process for a given k value. The dataset and k value are passed into the testKValue function; the function begins by splitting the data into folds and storing the fold arrays. I used 5 folds for my analysis because it had the best runtime without significant loss in accuracy. The function also creates an array to store the MSE value of each fold test for calculating the average later. Looping through each fold, the function creates a training dataset array from all the folds, and, after removing the current testing fold, casts the array as a two dimensional array instead of a three dimensional array. Then, looping through the current fold, the function appends each row to a testing dataset array to be passed into the kNearest function. Using the newly created training and testing datasets, testKValue passes these datasets and the provided k value to find the predicted ratings. These predictions and the test dataset are used to calculate the mean square error for that fold; the error is appended to the error array. This loops repeats for each fold, and once the loop concludes, the errors are averaged and returned by the function.

My main script runs a for loop for k-values 1-5 and prints the returned average mean squared error for each k value. After running, it several times, it seems that a k value of 5 results in the lowest error, as shown in the table below:

K-Value	Average Mean Standard Error
1	2.332
2	2.495
3	2.767
4	2.524
5	2.326

5 being the best k value for the algorithm is not that surprising, but what did surprise me is how the lower values performed better than the middle values. It makes sense than with larger k's, there is more information to base the prediction on making it more accurate; I would think that there would be more cases where only one neighbor resulted in more errors, but it seems that having two or three neighbors in the prediction caused more outliers to be present and skew the predictions. I would like to try in the future to see if the number of folds affect the k value calculations and if they could cause the best k value to change.