Lucas Summers

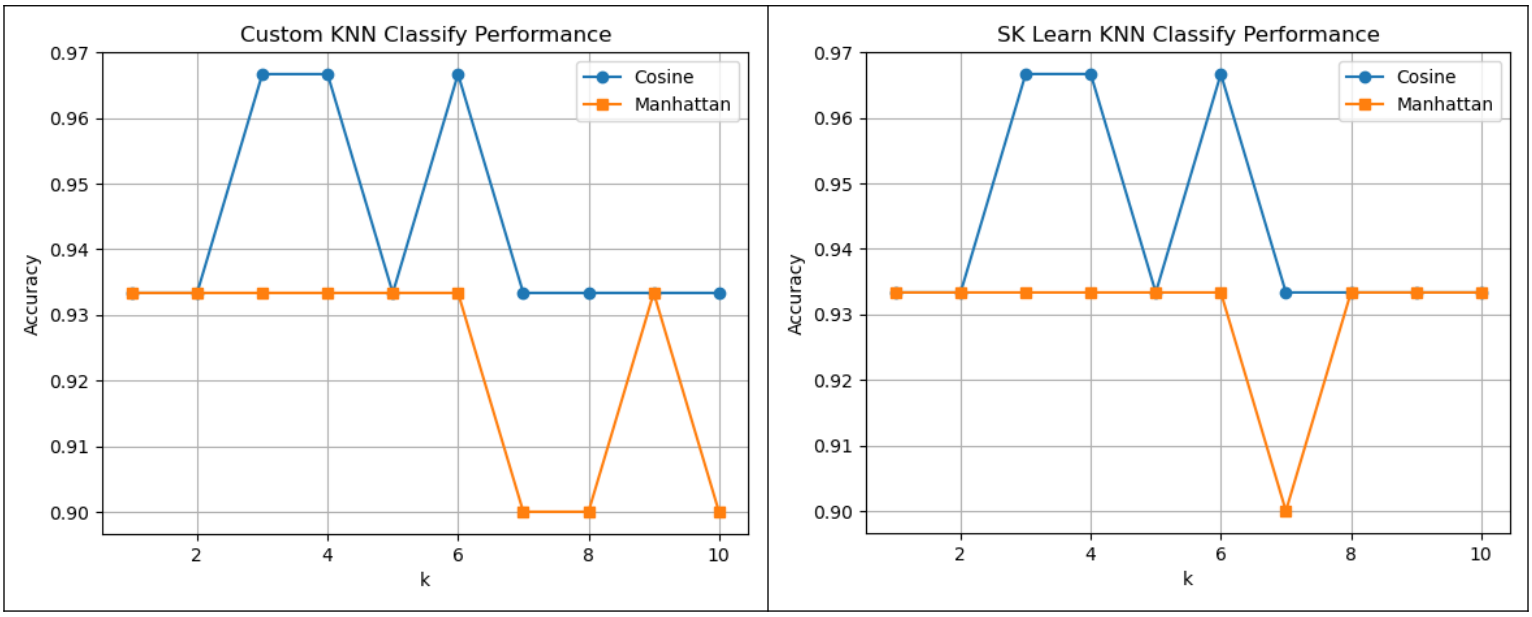
lsumme01@calpoly.edu

CSC 466 Lab 1

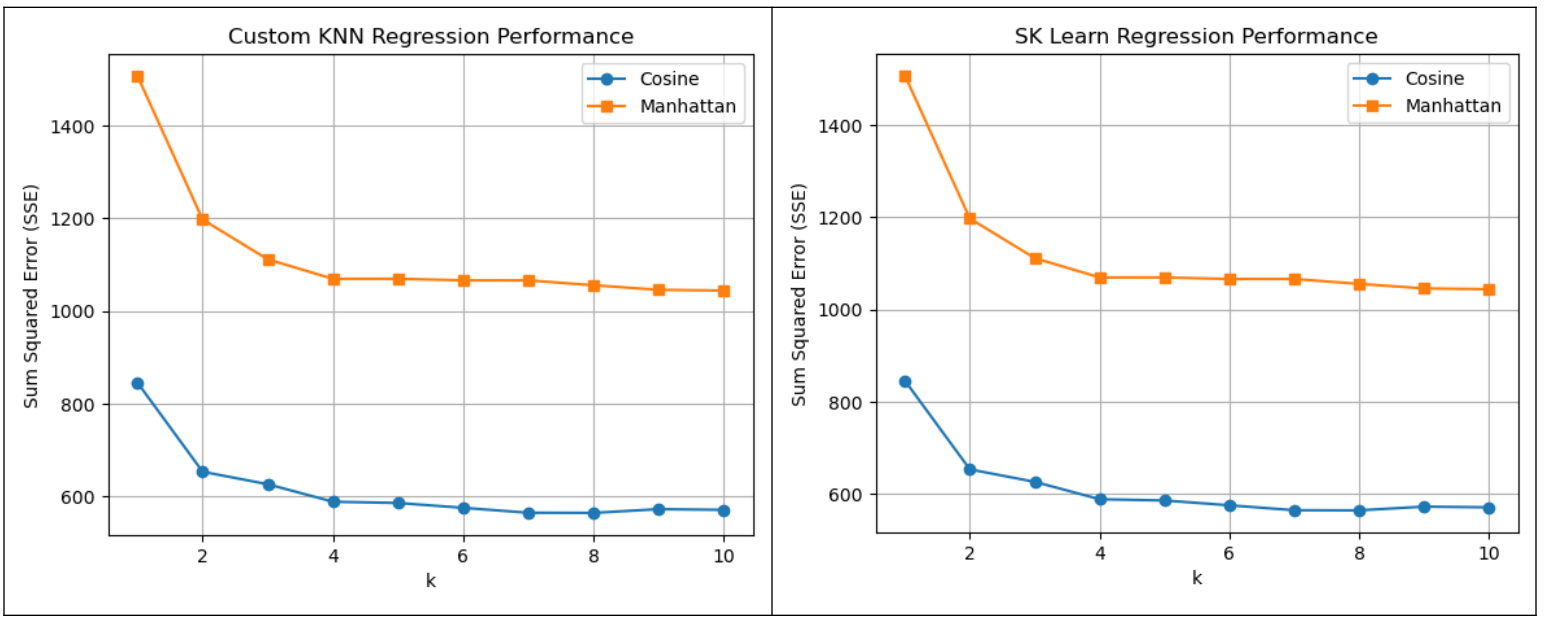
Implementation Notes:

First I implemented a **split_train_test** function which creates an 80-20 split of the data uses **np.random.shuffle()** and calculating an index to split at in order to create the desired size for each set. It also separates the features from the labels in both the training and test set. Note a random seed defaulting to 0 is kept to make the sets identical for all runs. My **grid_search** method loops through each value of k from 1 to the provided max_k and calls **evaluate_knn** for both cosine and Manhattan metrics. It then finds and reports the best model for either regression or classification by finding the lowest sum squared error or highest accuracy. All the results are also returned as a list of python dictionaries, which store all the relevant data about the specific model, such as the metric, k value, and the calculated evaluation metrics. This is useful for plotting the results later using the **plot_results** function. The **evaluate_knn** method loops through each datapoint in the test set, and uses the provided KNN function (either run_knn() or run_knn_sk()), model (either "regression" or "classify"), and metric (either "manhattan" or "cosine") to get a prediction, which is then stored in an array. Based on if we are doing regression or classification, we calculate the correct evaluation metrics with the prediction array and test labels (ground truth), returning them in a dictionary that is then inserted into the results dictionary. Each evaluation metric is its own function that takes a ground truth array (**test**) and a predictions array (**pred**) and returns the calculated metric. Finally, we have two KNN functions that can be used by **evaluate_knn()** to predict. The **run_knn** function is a custom version of the KNN algorithm. First, based on the provided metric it calculates the cosine similarity or Manhattan distance between the training set and the provided datapoint. Using **np.argsort()** to sort the data in ascending order. For Manhattan distance, we take the first k values from this sorted data to get the k nearest neighbors, and for cosine similarity we take the last k values. For classification, we then create a python Counter() and update the counts for each label of each nearest neighbor, then return the label with the highest count by using **most_common(1).** This also handles tiebreakers internally in a uniform way. For regression, we sum all the values for the nearest neighbors and divide by k to get an average. The **run_knn_sk** function has the same input and output as **run_knn(),** but uses sklearn's built in classes, either **KneighborsClassifier** for classification or **KneighborsRegressor** for regression**.** It then uses **knn.fit(train, labels)** and then **knn.predict(d)** to generate a prediction from our provided datapoint. Note, I needed to use **d = d.reshape(1, -1)** in order for the predictions to work correctly.

KNN Classification Results (Iris dataset):



KNN Regression Results (Cali Housing dataset ~ first 5000 samples):



Results Analysis:

The graphs above show the results of running grid search with KNN for both cosine similarity (cosine distance in the case of SK Learn implementation) and Manhattan distance metrics and for values of k ranging from 1 to 10. I chose the max k value as 10 because I found values above this did not provide

any better results and only created a longer runtime and harder to read graph. The custom KNN algorithm and the KNN algorithm provided by sklearn produced very similar results, with slight differences in the classification graphs probably due to different handling of tiebreakers. I was surprised by this similarity because of the difference of cosine similarity metric in the custom algorithm versus cosine distance metric used in the sklearn implementation. Despite this, both algorithms with grid search found the same best model (i.e. the same k value and metric) for both classification and regression on each dataset. Both best models used cosine similarity metric, suggesting its much more effective at KNN predictions over Manhattan distance for both classification and regression. This was especially noticeable with the graph of the regression results above.

The best model for classification of the Iris dataset was selected as having the highest accuracy (in the case of a tie the lowest k value was selected):

```
k: 3
metric: cosine
accuracy: 0.967
```

This suggests the model is very accurate on the data, probably due to the very small dataset as well as the relative simplicity of the data.

The best model for classification of the California Housing dataset was selected as having the lowest sum squared error (SSE) (in the case of a tie the lowest k value was selected):

```
k: 8
metric: cosine
sse: 564.14
mse: 0.564
mae: 0.533
```

These numbers suggest a reasonably accurate model for the data given. The MAE suggest the model's predictions are off by only about 0.53 units of the target variable, or about $53,000 in terms of the Median House Price. The small differences between MAE and MSE also indicate that the model is not overly affected by large outliers.