ROB313 Intro to Learning from Data - Assignment 1

Objectives

The objective of this assignment is to familiarize one with the applications and varying implementations of the k-NN algorithm. In this respect, the k-NN algorithm will be used for both regression and classification of several datasets. Additionally, various parameters of the algorithm will be simulated to analyze the effect on the k-NN's performance (e.g. distance metrics, feature space dimensions). In doing so, one will also become familiar with the available linear algebra tools/libraries for various computations. The performance of the k-NN algorithm will be compared to the Linear Regression Model for deeper understanding. Altogether, this assignment serves to provide a first-hand experience in applying theoretical machine learning knowledge by implementing two introductory level algorithms

Code Structure and Strategies

The code structure was designed to be modular and easily understandable. To run the code for each question, the only modifications needed to be made to the file is to uncomment the corresponding section outlined in the main block. The remaining code is structured as follows:

*Q1: **ff_regression(**data**)** – 5-fold regression function**,** **ff_regression_test(**data**)** – regression test*

*Q2: **one_fold_classification(**data**)**, **classification_test(**data**)** – classification validation and test*

Questions 1 and 2 each share a common function called **run_model(**model_type, dataset**)** from which the Q1 and Q2 functions are called from, depending on the *model_type* parameter. The *run_model* function is called directly from the main block.

*Q3: **regression_performance(**data, method**)** – Modified k-NN algorithms*

This function is called many times to compute the Test RMSE for the provided dataset. However, the *method* parameter specifies what type of computation should take place (e.g. *method = 'a'* will compute predictions using the double loop over test points). This function is called directly from the main block.

*Q4: **linear_regression(**data, model_type**)** – Linear regression function*

This function is called directly from the main block and will compute predictions and Test RMSE/Test Accuracy Ratio using a linear model for either regression or classification datasets. The parameter, *model_type*, specified whether to run classification or regression.

The first strategy deployed was to make the code very modular. Thus, easily interchangeable functions are run directly from the main block for each question. The functions were designed to be generalizable so that they can be re-used for datasets with differing feature space dimensions. Furthermore, the final code has gone through several iterations to optimize the runtime of each function. Originally, performing regression and classification for the larger datasets would take several hours at a time. Changes such as inverting the loop structure of the function significantly decreased the overall runtime. For example, in Q1 and Q2, the distances between a given test point and all training points are computed only once, and then used to obtain the k-nearest neighbours for many of k values. Finally, print statements are neatly organized throughout the main block so that the results can be easily obtained by the operator when executing the code. Neatness and organization aided in the debugging process so that little time was wasted searching for incorrect results.

Q1 - k-NN Algorithm for Regression

1-a) Implement k-NN Algorithm for Regression Datasets and use 5-fold Cross Validation:

The results of running the k-NN regression on all regression datasets are provided below. A *Cross-Validation RMSE*, *Test RMSE*, *Estimated k, and preferred Distance Metric* is reported for each dataset. Note that for each dataset, k values on a range (1, 30) were simulated. From these results, we can see a correspondence between the dimensionality of the feature space to the estimated k value. Additionally, we see that datasets with close feature space dimensions prefer the same distance metric.

Table 1 – k-NN Regression Results

Dataset	**Estimated k**	**Distance Metric**	**CV RMSE**	**Test RMSE**
mauna_loa	2	L2-norm	0.0318041026269	0.440704890355
rosenbrock	2	L2-norm	0.330741584025	0.247598644133
pumadyn32nm	25	L-inf-norm	0.87221127042	0.832480142444

1-b) Cross-Validation Loss Curve and Prediction on Test Set for Mauna Loa (L2-norm)

Figure 1 illustrates the cross-validation loss (merging predictions from all splits) across increasing values of k for the Mauna Loa dataset. As expected, we see that the loss is minimized for a k value of 2. Th corresponds to the Mauna Loa result shown in *Table 1*. Further, we see that the loss rapidly increases for higher k values which suggests that computing an estimate with more neighbours negatively effects the accuracy of predictions.

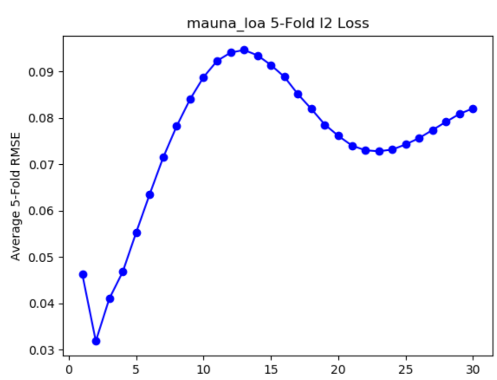

Figure 1 – Cross Validation Loss Curve for Mauna Loa

In *Figure 2*, we have the prediction of the k-NN regression on the test set for Mauna Loa. Notice that for all test points, the predicted values are identical. Intuitively, this makes sense as the Mauna Loa dataset represents data taken over *time*. The 1-dimensional feature space of time values for the test set contains data points that were collected after all the data points in the training and validation set, chronologically. Thus, the k nearest neighbours for all test points are the same, yielding identical predictions. It is for this reason that the Test RMSE shown in *Table 1* is significantly larger than the Cross-Validation RMSE for this particular dataset.

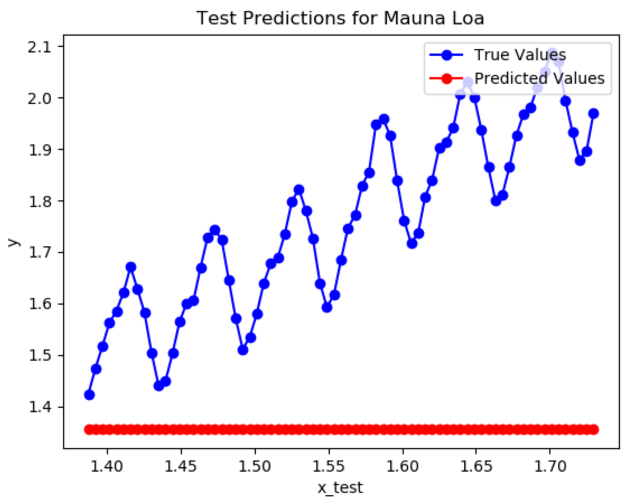

Figure 2 – Prediction on Test Set for Mauna Loa

Q2 - k-NN Algorithm for Classification

A similar k-NN algorithm was used as a classification model and was applied to two classification datasets. The results are summarized in *Table 2*, containing the *Estimated k, preferred Distance Metric, Validation Accuracy Ratio, and the Test Accuracy Ratio* for each dataset.

Table 2 – k-NN Classification Results

Dataset	**Estimated k**	**Distance Metric**	**Validation Ratio**	**Test Ratio**
iris	15	L1-norm	0.9032258064516129	1.0
mnist_small	1	L2-norm	0.95	0.959

Q3 - k-NN Performance with Varying Computational Modifications

Four versions of the k-NN algorithm were implemented to compute predictions on the rosenbrock (k=5, n_train = 5000) regression test set over increasing values of d (dimension of feature space). The four versions were: *brute force double loop (a), half-vectorized method (b), fully-vectorized method (c), k-d tree method (d)*. Using the L2-norm to compute distances, the performance metrics *Run-time and Test RMSE*, were tracked over d values on a range (2, 9). The quantities are summarized in *Table 3* and displayed in *Figures 3-4*.

Table 3 – k-NN Performance Results with Various Computational Modifications (plotted on Figures 3-4)

d	**Methods (a, b, c, d)**							
---	Double Loop (a)		Half-Vectorized (b)		Fully-Vectorized (c)		k-d Tree (d)	
---	Time [s]	RMSE	Time [s]	RMSE	Time [s]	RMSE	Time [s]	RMSE
2	172.363	0.267	2.505	0.267	0.157	0.267	0.003	0.267
3	106.918	0.379	2.873	0.379	0.155	0.379	0.005	0.379
4	109.243	0.420	2.899	0.420	0.171	0.420	0.007	0.420
5	305.698	0.521	3.007	0.521	0.158	0.521	0.011	0.521
6	290.247	0.611	3.012	0.611	0.164	0.611	0.018	0.611
7	129.156	0.689	3.485	0.689	0.170	0.689	0.029	0.689
8	310.322	0.747	6.338	0.747	0.204	0.747	0.174	0.747
9	153.278	0.800	3.904	0.800	0.180	0.800	0.064	0.800

Figure 3 depicts the prediction runtimes of all four methods over increasing feature space dimension. As we can see, the Double Loop method take significantly longer than the vectorized and k-d tree methods. Analyzing *Table 3*, we can see that the k-d tree implementation is consistently the best performer in terms of time, followed by the fully-vectorized and half-vectorized methods, although the k-d tree and fully-vectorized methods come quite close. An interesting observation is that the double loop method seems to be extremely sensitive to the size of the feature space, while the other three methods stay quite consistent in runtimes. Overall, vectorizing the code significantly reduced the runtime of the algorithm, and using a k-d data structure further improves the computational performance in comparison to the brute-force method.

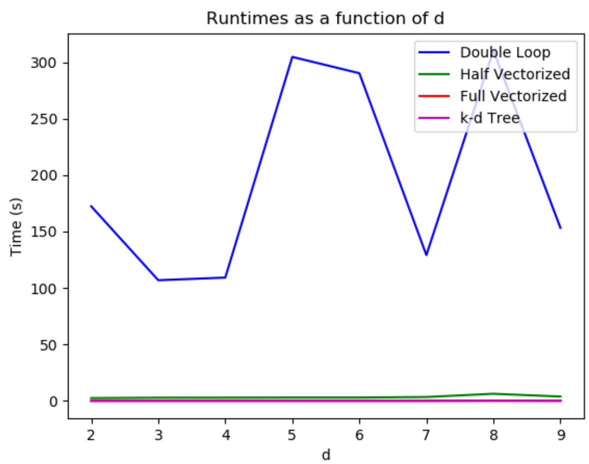

Figure 3 – Prediction Runtimes for all k-NN Methods

Figure 4 displays the Test RMSE values for all methods over the increasing feature space dimension. As expected, all versions of the k-NN algorithm predict the same values, and hence their test RMSE values should be identical. This corresponds to the overlapping curves below.

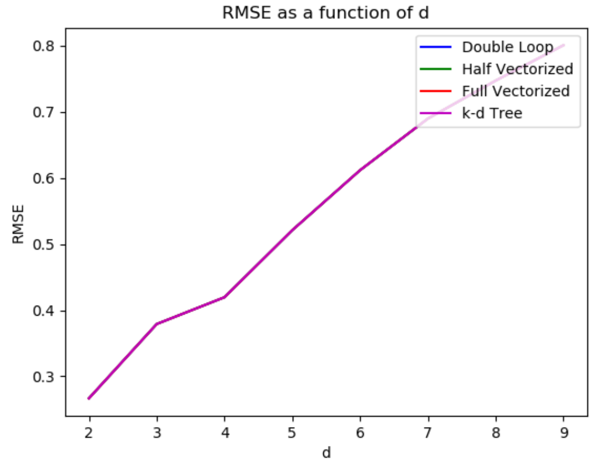

Figure 4 – Test RMSE for all k-NN Methods

Q4 - Linear Regression Model for Regression and Classification

Two versions of a linear regression model were implemented for regression and classification of all datasets. The test RMSE values (regression) and test accuracy ratios (classification) are reported in *Table 4*. Comparing the results of the linear model to the k-NN algorithm we see several things. For the regression sets, the test RMSE values suffered a significant increase for rosenbrock and a minor increase pumadyn32nm. However, the test RMSE value decreased for the Mauna Loa dataset. The result for Mauna Loa is expected, as the linear regression model is able to better fit the data with an increasing slope, which the k-NN algorithm was unable to do (see *Figure 2)*. For classification, the linear regression model slight underperformed in comparison to the k-NN algorithm for both iris and mnist_small datasets. Thus, the k-NN algorithm outperforms the linear regression model over all datasets except for Mauna Loa.

Table 4 – Linear Regression Model Test Results

Dataset	**Test RMSE**	**Test Accuracy Ratio**
mauna_loa	0.349388310499	---
rosenbrock	0.984087203069	---
pumadyn32nm	0.86225124366	---
iris	---	0.866666666667
mnist_small	---	0.855

```python
import numpy as np
import time
import math
from matplotlib import pyplot as plt
from data_utils import load_dataset
import heapq
from sklearn import neighbors

__author__ = 'Christopher Agia (1003243509)'
__date__ = 'February 12, 2019'


# Root Mean Squared Error Function
def rmse(y_test, y_estimates):
    return np.sqrt(np.average((y_test-y_estimates)**2))

# Norm Utility Functions
def l1_norm(x1, x2):
    return np.linalg.norm([x1-x2], ord=1)


def l2_norm(x1, x2):
    return np.linalg.norm([x1-x2], ord=2)


def linf_norm(x1, x2):
    return np.linalg.norm([x1-x2], ord=np.inf)


def ff_regression(x_train, x_valid, y_train, y_valid, distance_functions, k_list=None):

    assert((len(x_train)+len(x_valid)) == (len(y_train)+len(y_valid)))

    rmse_vals = {}
    error = []

    x_total = np.vstack([x_train, x_valid])
    y_total = np.vstack([y_train, y_valid])
    np.random.seed(5)
    np.random.shuffle(x_total)
    np.random.seed(5)
    np.random.shuffle(y_total)

    ff_length = len(x_total)//5

    #  Trial K values to try
    if not k_list:
        k_list = list(range(0, 30))

    # Iterate over 5-folds
    for i in range(5):

        # Obtain validation and train set associated with fold i
        y_valid = y_total[i * ff_length:(i + 1) * ff_length]
        y_train = np.vstack([y_total[:i * ff_length], y_total[(i + 1) * ff_length:]])
        x_valid = x_total[i * ff_length:(i + 1) * ff_length]
        x_train = np.vstack([x_total[:i * ff_length], x_total[(i + 1) * ff_length:]])

        # Loop over distance functions (e.g. l1_norm, l2_norm, linf_norm)
        for func in distance_functions:
            y_est = {}
            # Compute distances according to distance function for one validation point
            for j in range(ff_length):
                d = []
                for t in range(len(x_train)):
                    d.append((func(x_train[t], x_valid[j]), y_train[t]))

                # Sort distances to take the nearest k-values
                d.sort(key=lambda x: x[0])

                # Compute y_estimate for validation point j
                for k in k_list:
                    y = 0
                    for elem in d[:k+1]:
                        y += elem[1]

                    if k not in y_est:
                        y_est[k] = []

```

```
                    y_est[k].append(y/(k+1))

            # Compute root mean squared error for each k-value
            for k in k_list:
                if (func, k) not in rmse_vals:
                    rmse_vals[(func, k)] = []
                rmse_vals[(func, k)].append(rmse(y_valid, y_est[k]))


    # Error will contain 30 rmse-error values as per k for each function
    for func, k in rmse_vals:
        ff_error = sum(rmse_vals[(func, k)]) / 5
        error.append((k+1, func, ff_error))

    return error


def ff_regression_test(x_train, x_valid, x_test, y_train, y_valid, y_test, k, func, plot=False):

    # Create total training set (no validation)
    x_total = np.vstack([x_train, x_valid])
    y_total = np.vstack([y_train, y_valid])

    # Predictions for each test data point will be stored in this list
    predictions = []

    # Iterate over test points
    for elem in x_test:

        # Compute distances between all training points and test point
        d = []
        for i in range(len(x_total)):
            d.append((func(elem, x_total[i]), y_total[i]))

        # Sort in terms of increasing distance
        d.sort(key=lambda x: x[0])

        # Average over k nearest y values
        y_est = 0
        for item in d[:k]:
            y_est += item[1]
        avg = y_est/k

        # Append to predictions
        predictions.append(avg)

    test_error = rmse(y_test, predictions)

    if plot:
        # Predictions on test set
        plt.figure(2)
        plt.plot(x_test, y_test, '-bo', label='True Values')
        plt.plot(x_test, predictions, '-ro', label='Predicted Values')
        plt.title('Test Predictions for Mauna Loa')
        plt.xlabel('x_test')
        plt.ylabel('y')
        plt.legend(loc='upper right')
        plt.savefig('mauna_loa_prediction.png')

    return test_error


def one_fold_classification(x_train, y_train, x_valid, y_valid, distance_functions, k_list=None):

    assert ((len(x_train) + len(x_valid)) == (len(y_train) + len(y_valid)))

    tally = {}

    if not k_list:
        k_list = list(range(0, 30))

    # Loop over distance functions (e.g. l1_norm, l2_norm, linf_norm)
    for func in distance_functions:

        # Compute distances according to distance function for one validation point
        for j in range(len(x_valid)):
            d = []
            for t in range(len(x_train)):
                d.append((func(x_train[t], x_valid[j]), y_train[t]))

```

```
            # Sort distances to take the nearest k-values
            d.sort(key=lambda x: x[0])

            classes = {}
            # Identify k-nearest neighbours for x_valid[j]
            for k in k_list:
                classes[k] = []
                for elem in d[:k + 1]:
                    classes[k].append(elem[1])

            for k in k_list:
                occurance = {}
                for point in classes[k]:
                    if str(point) not in occurance:
                        occurance[str(point)] = (point, 0)
                    occurance[str(point)] = (point, occurance[str(point)][1] + 1)

                occur_list = list(occurance.values())
                occur_list.sort(key=lambda x: x[1], reverse=True)

                if np.all(occur_list[0][0] == y_valid[j]):
                    if (k + 1, func) not in tally:
                        tally[(k + 1, func)] = 0
                    tally[(k + 1, func)] += 1

    results = []
    for k, func in tally:
        ratio = tally[(k, func)]/len(y_valid)
        result = (k, func, ratio)
        results.append(result)

    return results


def classification_test(x_train, x_valid, x_test, y_train, y_valid, y_test, k, func):

    # Create total training set (no validation)
    x_total = np.vstack([x_train, x_valid])
    y_total = np.vstack([y_train, y_valid])

    tally = 0
    # Iterate over test points
    for i in range(len(x_test)):

        # Compute distances between all training points and test point
        d = []
        for j in range(len(x_total)):
            d.append((func(x_test[i], x_total[j]), y_total[j]))

        # Sort in terms of increasing distance
        d.sort(key=lambda x: x[0])

        occurance = {}
        for item in d[: k]:
            if str(item[1]) not in occurance:
                occurance[str(item[1])] = (item[1], 0)
            occurance[str(item[1])] = (item[1], occurance[str(item[1])][1] + 1)

        occur_list = list(occurance.values())
        occur_list.sort(key=lambda x: x[1], reverse=True)

        if np.all(occur_list[0][0] == y_test[i]):
            tally += 1

    return tally/len(x_test)


def run_model(model_type, dataset):

    functions = [l1_norm, l2_norm, linf_norm]

    if model_type == 'regression':

        if dataset == 'rosenbrock':
            x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset(dataset, n_train=1000, d=2)
        else:
            x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset(dataset)

        if dataset == 'mauna_loa':
            result = ff_regression(x_train, x_valid, y_train, y_valid, [l2_norm])
```

```python

            # Result = (k, Function, RMSE Error)
            result.sort(key=lambda x: x[0])

            k_vals = []
            error_vals = []
            for k, func, error in result:
                k_vals.append(k)
                error_vals.append(error)

            plt.figure(1)
            plt.plot(k_vals, error_vals, '-bo')
            plt.xlabel('k')
            plt.ylabel('Average 5-Fold RMSE')
            plt.title(dataset + ' 5-Fold l2 Loss')
            plt.savefig(dataset +'l2_loss.png')

            result.sort(key=lambda x: x[2])
            k_min = result[0][0]
            func = result[0][1]
            test_error = ff_regression_test(x_train, x_valid, x_test, y_train, y_valid, y_test, k_min, func, plot=True)

            print('-----------------------------------------------')
            print('Results for Mauna Loa with L2 Norm :')
            print('')
            print('Optimal k: ' + str(k_min))
            print('Optimal Distance Metric: ' + str(func))
            print('Five fold RMSE: ' + str(result[0][2]))
            print('Test RMSE: ' + str(test_error))
            print('')

        result = ff_regression(x_train, x_valid, y_train, y_valid, functions)
        result.sort(key=lambda x: x[2])
        k_min = result[0][0]
        func = result[0][1]
        test_error = ff_regression_test(x_train, x_valid, x_test, y_train, y_valid, y_test, k_min, func)

        return result[0][0], result[0][1], result[0][2], test_error

    elif model_type == 'classification':

        x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset(dataset)
        one_fold_result = one_fold_classification(x_train, y_train, x_valid, y_valid, functions)
        one_fold_result.sort(key=lambda x: x[2], reverse=True)
        # ff_result = (k, func, ratio)
        k_min = one_fold_result[0][0]
        func = one_fold_result[0][1]
        test_ratio = classification_test(x_train, x_valid, x_test, y_train, y_valid, y_test, k_min, func)

        return k_min, func, one_fold_result[0][2], test_ratio

    return 0


def regression_performance(x_total, x_test, y_total, y_test, k, f, method):

    start = time.time()

    # Predictions for each test data point will be stored in this list
    predictions = []

    # Brute-force Double For Loop Method
    if method == 'a':

        # Iterate over test points
        for elem in x_test:

            # Compute distances between all training points and test point
            d = []
            for j in range(len(x_total)):
                d.append((f(elem, x_total[j]), y_total[j]))

            # Sort in terms of increasing distance
            d.sort(key=lambda x: x[0])

            # Average over k nearest y values
            y_est = 0
            for item in d[:k]:
                y_est += item[1]
            avg = y_est/k
```

```python

            # Append to predictions
            predictions.append(avg)

        test_error = rmse(y_test, predictions)

    # Half-Vectorization Method
    elif method == 'b':

        for j in range(len(x_test)):
            d = np.sqrt(np.sum(np.square(x_total - x_test[j]), axis=1))
            k_nb = heapq.nsmallest(k, range(len(d)), d.take)
            predictions.append(np.array([(np.average(np.take(y_total, k_nb)))]))

        test_error = rmse(y_test, predictions)

    # Full Vectorization Method
    elif method == 'c':

        d = np.sqrt(-2 * np.dot(x_test, x_total.T) + np.sum(x_total ** 2, axis=1) + np.sum(x_test ** 2, axis=1)[:, np.newaxis])
        k_nb = np.argpartition(d, kth=k, axis=1)[:, : k]
        predictions = np.sum(y_total[k_nb], axis=1) / k

        test_error = rmse(y_test, predictions)

    # K-D Tree Method
    elif method == 'd':

        kdt = neighbors.KDTree(x_total)
        d, k_nb = kdt.query(x_test, k=k)
        predictions = np.sum(y_total[k_nb], axis=1) / k

        test_error = rmse(y_test, predictions)

    runtime = time.time() - start
    return runtime, test_error


def linear_regression(x_train, x_valid, x_test, y_train, y_valid, y_test, model_type):

    if model_type == 'regression':

        x_total = np.vstack([x_train, x_valid])
        y_total = np.vstack([y_train, y_valid])

        # Create X matrix
        X = np.ones((len(x_total), len(x_total[0]) + 1))
        X[:, 1:] = x_total

        # Compute SVD
        U, S, Vh = np.linalg.svd(X)

        # Invert Sigma
        sig = np.diag(S)
        filler = np.zeros([len(x_total)-len(S), len(S)])
        sig_inv = np.linalg.pinv(np.vstack([sig, filler]))

        # Compute weights and predictions
        w = np.dot(Vh.T, np.dot(sig_inv, np.dot(U.T, y_total)))

        X_test = np.ones((len(x_test), len(x_test[0]) + 1))
        X_test[:, 1:] = x_test
        predictions = np.dot(X_test, w)

        result = rmse(y_test, predictions)

    elif model_type == 'classification':

        x_total = np.vstack([x_train, x_valid])
        y_total = np.vstack([y_train, y_valid])

        # Expand X matrix
        X = np.ones([len(x_total), len(x_total[0]) + 1])
        X[:, 1:] = x_total

        # Convert to integer
        #y_test = 1 * y_test

        # Perform SVD
        U, S, Vh = np.linalg.svd(X)
```

```python

        # Expand Sigma Matrix
        sig = np.diag(S)
        filler = np.zeros([len(x_total) - len(S), len(S)])
        sig_inv = np.linalg.pinv(np.vstack([sig, filler]))

        # Compute weights
        w = np.dot(Vh.T, np.dot(sig_inv, np.dot(U.T, y_total)))

        # Create Test Matrix
        X_test = np.ones([len(x_test), len(x_test[0]) + 1])
        X_test[:, 1:] = x_test

        # find prediction accuracy
        predictions = np.argmax(np.dot(X_test, w), axis=1)
        y_test = np.argmax(1 * y_test, axis=1)

        result = (predictions == y_test).sum() / len(y_test)

    return result


if __name__ == '__main__':
    # All Dataset Names
    all_datasets = ['mauna_loa', 'rosenbrock', 'pumadyn32nm', 'iris', 'mnist_small']
    regression_sets = ['mauna_loa', 'rosenbrock', 'pumadyn32nm']
    classification_sets = ['iris', 'mnist_small']

    # ---------------------- Question 1 ----------------------

    # print('------------------ Overall Results for Question 1 -------------------')
    # print('')
    # for d_set in regression_sets:
    #     k_min, metric_min, ff_rmse, test_rmse = run_model('regression', d_set)
    #     print('-----------------------------------------------')
    #     print('Results for ' + d_set + ' :')
    #     print('')
    #     print('Optimal k: ' + str(k_min))
    #     print('Optimal Distance Metric: ' + str(metric_min))
    #     print('Five fold RMSE: ' + str(ff_rmse))
    #     print('Test RMSE: ' + str(test_rmse))
    #     print('')


    # ---------------------- Question 2 ----------------------

    # print('------------------ Overall Results for Question 2 -------------------')
    # print('')
    # for d_set in classification_sets:
    #     k_min, metric_min, max_ratio, test_ratio = run_model('classification', d_set)
    #     print('-----------------------------------------------')
    #     print('Results for ' + d_set + ' :')
    #     print('')
    #     print('Optimal k: ' + str(k_min))
    #     print('Optimal Distance Metric: ' + str(metric_min))
    #     print('Validation Ratio: ' + str(max_ratio))
    #     print('Test Ratio: ' + str(test_ratio))
    #     print('')


    # ---------------------- Question 3 ----------------------

    # result_table = {}
    # result_table['Double Loop'] = []
    # result_table['Half Vectorized'] = []
    # result_table['Full Vectorized'] = []
    # result_table['k-d Tree'] = []
    #
    # for ind_d in range(2, 10):
    #
    #     x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset('rosenbrock', n_train=5000, d=ind_d)
    #     x_total = np.vstack([x_train, x_valid])
    #     y_total = np.vstack([y_train, y_valid])
    #
    #     run_time, test_rmse = regression_performance(x_total, x_test, y_total, y_test, 5, l2_norm, 'a')
    #     result_table['Double Loop'].append((ind_d, run_time, test_rmse))
    #
    #     run_time, test_rmse = regression_performance(x_total, x_test, y_total, y_test, 5, l2_norm, 'b')
    #     result_table['Half Vectorized'].append((ind_d, run_time, test_rmse))
    #
```

```
#     run_time, test_rmse = regression_performance(x_total, x_test, y_total, y_test, 5, l2_norm, 'c')
#     result_table['Full Vectorized'].append((ind_d, run_time, test_rmse))
#
#     run_time, test_rmse = regression_performance(x_total, x_test, y_total, y_test, 5, l2_norm, 'd')
#     result_table['k-d Tree'].append((ind_d, run_time, test_rmse))
#
# print('------------------ Overall Results for Question 3 -------------------')
# print('')
#
# m = list(result_table.keys())
# for r in range(8):
#     print('--- Results for d = ' + str(r+2) + ' ---')
#     print('')
#     print('Times-- ' + m[0] + ': ' + str(result_table[m[0]][r][1]) + '  ' + m[1] + ': ' + str(result_table[m[1]][r][1]))
#     print('        ' + m[2] + ': ' + str(result_table[m[2]][r][1]) + '  ' + m[3] + ': ' + str(result_table[m[3]][r][1]))
#     print('RMSEs-- ' + m[0] + ': ' + str(result_table[m[0]][r][2]) + '  ' + m[1] + ': ' + str(result_table[m[1]][r][2]))
#     print('        ' + m[2] + ': ' + str(result_table[m[2]][r][2]) + '  ' + m[3] + ': ' + str(result_table[m[3]][r][2]))
#     print('')
#
# d_arr = list(range(2, 10))
# plt.figure(3)
# plt.title('Runtimes as a function of d')
# plt.xlabel('d')
# plt.ylabel('Time (s)')
# plt.figure(4)
# plt.title('RMSE as a function of d')
# plt.xlabel('d')
# plt.ylabel('RMSE')
# count = 0
#
# for m in result_table:
#
#     if count == 0:
#         tab = '-b'
#     elif count == 1:
#         tab = '-g'
#     elif count == 2:
#         tab = '-r'
#     else:
#         tab = '-m'
#     count += 1
#
#     runtimes = []
#     rmses = []
#     for i in range(len(result_table[m])):
#         runtimes.append(result_table[m][i][1])
#         rmses.append(result_table[m][i][2])
#
#     plt.figure(3)
#     plt.plot(d_arr, runtimes, tab, label=m)
#
#     plt.figure(4)
#     plt.plot(d_arr, rmses, tab, label=m)
#
# plt.figure(3)
# plt.legend(loc='upper right')
# plt.savefig('runtimes_vs_d.png')
#
# plt.figure(4)
# plt.legend(loc='upper right')
# plt.savefig('rmses_vs_d.png')

# ---------------------- Question 4 ----------------------

# print('------------------ Overall Results for Question 4 -------------------')
# print('')
# for d_set in regression_sets:
#
#     if d_set == 'rosenbrock':
#         x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset(d_set, n_train=1000, d=2)
#     else:
#         x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset(d_set)
#
#     final_rmse = linear_regression(x_train, x_valid, x_test, y_train, y_valid, y_test, 'regression')
#     print('Test RMSE for ' + d_set + ': ' + str(final_rmse))
#
#
# for d_set in classification_sets:
#
#     x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset(d_set)
```

```
#
#     final_ratio = linear_regression(x_train, x_valid, x_test, y_train, y_valid, y_test, 'classification')
#     print('Test Ratio for ' + d_set + ': ' + str(final_ratio))
```