ROB313 Assignment 1

Alvin Pane [1004281118]

February 16, 2020

Objectives

The objective of this assignment is to study the effectiveness of various approaches to the k-NN algorithm. This assignment applies k-NN in both regression and classification to analyze k-NN's performance when parameters such as distance metrics and dimension are varied. By necessity, this will require an understanding of the implementations of popular Python libraries Numpy and SciKit. The performance of k-NN will then be compared to a linear regression model. In summary, this assignment serves as an introduction to real-world applications of basic machine learning algorithms.

Code Structure

The code was designed in a way to optimize the experience for the person running the code. The main strategy was to make the code modular. This was achieved by defining many different functions, each responsible for handling a small task. The functions were intended to be as general as possible so they could be called multiple times with different inputs. This was in the interest of space efficiency. The code uses print statements very frequently, to aid in debugging and to present data to the user in an intuitive manner. The main section then calls functions depending on what question is to be answered. There are four variables: Q1, Q2, Q3, and Q4, all initialized to False. To run a question, simply set the variable equal to True.

- Q1 calls the *regressionmodel()* function, which calls the *regression()* and *test()* functions. *Regression()* performs the 5-fold regression test and determines the root mean square error, while *test* computes the test error and plots test predictions against actual values.
- Q2 calls *classificationmodel()*. From there, *classification()* and *classifytest()* are called to determine the optimal k value, distance metric, and the test accuracy.
- Q3 uses a for-loop to iterate over values of d from 2 to 9, each iteration calling the *qthree()* function, which returns the runtime and test RMSE for each d value using the kd-Tree method. These are then appended to a results array and plotted against the list of d values. Next, *rosenbrockperform()* is called to determine the runtime of the brute force approach when calculating test RMSE at a specified d value. Early iterations of this code had *rosenbrockperform()* called in a for-loop at all values of d, but because the brute force method is so inefficient, the computation time for this section was on the order of hours.
- Q4 runs *SVDregress()* on all model types. Based on whether the model is of type regression or classification, this function performs SVD based linear regression and returns the test RMSE or the test accuracy.

Q1

The results for k-NN regression on the regression datasets (mauna-loa, rosenbrock, and pumadyn32nm) are shown in Table 1 below. In the interest of keeping computing time low, only K values between 1 and 25 were considered in simulation.
Table 1: Results when Applying K-NN Algorithm to Regression Datasets

Dataset	K	Best Metric	Test RMSE	5-Fold RMSE
Mauna-Loa	2	L1 Norm	0.4407	0.0318
Rosenbrock	2	L2 Norm	0.2476	0.3307
Pumadyn32nm	24	L2 Norm	0.8546	0.8951

Q1 also asks to plot the cross-validation prediction curves against the test set for different k values, as well as the RMSE for the dataset Mauna Loa with the L2 distance metric. These results are shown in Figure 1 and Figure 2, respectively.

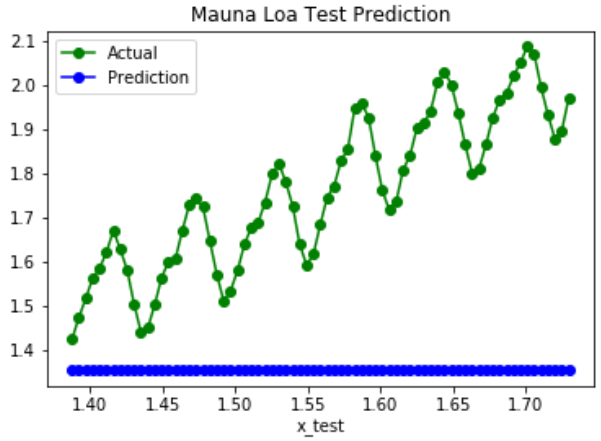

Figure 1: Mauna Loa Prediction against the Test Set, Actual and Predicted Values

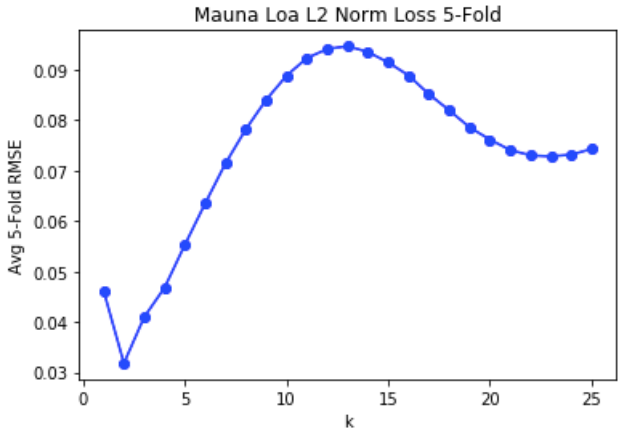

Figure 2: Mauna Loa L2 Loss at Varying K

As expected from Table 1, Figure 2 illustrates how the loss is minimized at k=2, hence why the algorithm selected it as the optimal k. As k is increased, there is a steep linear increase in the 5-fold loss, this suggests that for this dataset, considering a greater number of neighbors does not positively impact the accuracy of predictions. In Figure 1, we see that the predicted values are constant across k. This would seem to suggest the nearest neighbors are identical at all test points, leading to similar predictions.

Q2:

The results for the modified k-NN algorithm on the classification datasets (Iris and Mnist_small) are shown in Table 2 below. In the interest of keeping computing time low, only K values between 1 and 25 were considered in simulation.

Table 2: Results when Applying Modified K-NN Algorithm to Classification Datasets

Dataset	K	Best Metric	Validation Accuracy	Test Accuracy
Iris	25	L2 Norm	0.90323	1.0
Mnist_small	25	L2 Norm	0.9001	0.945

Q3:

This section analyzes how the KD-Tree method compares to the brute-force method in computing predictions on the rosenbrock dataset, with k=5 and n_train = 5000. The L2 distance metric is used. Figure 3 and Figure 4 summarize the runtimes and RMSE as d is varied in the KD-Tree method.

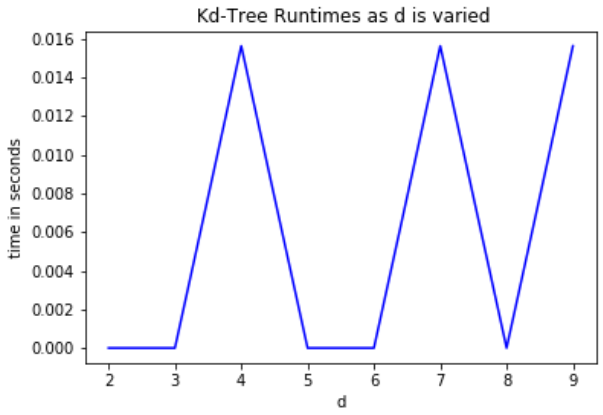

Figure 3: Runtimes vs d for the KD-Tree Method

Figure 3 shows just how efficient the performance of the KD-Tree method is. For d=9, the runtime was just 0.016s, as compared to 179.25s for the brute force method. This is conclusive in stating that the brute force method is inferior to KD-trees in terms of time. This may be due to the fact that the brute force method employs nested for-loops, which are very sensitive to the size of the datasets, causing the code to execute over long periods of time. Interestingly, there does not seem to be a direct correlation between the dimension d, and the runtime of the KD-Tree method. This may mean that the method can be extended to larger dimensions without significant runtime penalties. The consequence of an increased dimension will be a greater RMSE loss, however. As expected, both methods computed nearly identical test RMSEs.

Q4:

A linear regression model based on SVD was implemented for the regression and classification datasets. The results for this implementation are summarized in Table 3. As compared to Table

1, the regression sets had their RMSEs increased, with the exception of Mauna Loa which decreased. The RMSE for Rosenbrock was drastically increased by the linear regression. A similar result exist for the classification sets when comparing with Table 2, the test accuracy was decreased for all classification datasets. This leads to the conclusion that the k-NN algorithm is superior to the linear regression model in all datasets with the exception of Mauna Loa.

Table 3: Summary of Linear Regression Model Results

Dataset	Test RMSE
Mauna Loa	0.349388
Rosenbrock	0.984087
Pumadyn32nm	0.8622512
---	Test Accuracy
Iris	0.86666
Mnist_small	0.8570

Appendix A: Python Script

In [4]:

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

# ALVIN PANE [1004281118]

# ROB313

Q1 = False
Q2 = True
Q3 = False
Q4 = False

# Error function root mean square
def rms_err(y_t, y_e):
    return np.sqrt(np.average((y_t-y_e)**2))

# distance norms

#manhattan distance
def l1norm(x1, x2):
    return np.linalg.norm([x1-x2], ord=1)

#euclidean distance
def l2norm(x1, x2):
    return np.linalg.norm([x1-x2], ord=2)

def classification(x_train, y_train, x_valid, y_valid, distance_
functions, k_list=None):

    kvals = list(range(0, 25))
    result = []
    # iterate over l2 and l1 norm
    for func in distance_functions:

        for j in range(len(x_valid)):  #distances for valid poin
t
            dist = []
```

```
            for t in range(len(x_train)):
                dist.append((func(x_train[t], x_valid[j]), y_tra
in[t]))

            dist.sort(key=lambda x: x[0]) #sort

            class_ = {}
            for k in kvals:
                class_[k] = []
                for elem in dist[:k + 1]:    #for nearest distan
ces, append to class list
                    class_[k].append(elem[1])

            for k in kvals:
                inst = {}
                for p in class_[k]:
                    if str(p) not in inst:    #instances of poin
t p in k
                        inst[str(p)] = (p, 0)
                    inst[str(p)] = (p, inst[str(p)][1] + 1)

                occlist = list(inst.values())
                occlist.sort(key=lambda x: x[1], reverse=True)
#sort reverse
                count = {}

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

    for k, func in count:
        r = count[(k, func)]/len(y_valid) #avg
        res = (k, func, r)
        result.append(res)

    return result


def regression(x_train, x_valid, y_train, y_valid, distance_func
tions):

    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)
    kvals = list(range(0, 25))
    rvals = {} #rmse = rvals
    error = []
    foldsize = len(x_total)//5

    for i in range(5):

        # split into training and validation sets, partition lis
ts based on size of each fold
        y_valid = y_total[i * foldsize : (i + 1) * foldsize]
        y_train = np.vstack([y_total[:i * foldsize], y_total[(i
+ 1) * foldsize:]])
        x_valid = x_total[i * foldsize:(i + 1) * foldsize]
        x_train = np.vstack([x_total[:i * foldsize], x_total[(i
+ 1) * foldsize:]])
        kvals = list(range(0, 25))
        #print(y_valid)
        # run for both l1 and l2
        for func in distance_functions:
            y_est = {}

            #iterate over fold
            for f in range(foldsize):
                dist = []
                for t in range(len(x_train)):
                    dist.append((func(x_train[t], x_valid[f]), y
_train[t]))

                # sort distances, anon lambda func
                dist.sort(key=lambda x: x[0])

                # iterate over possible kvals from 0 to 20 and d
etermine y estimate
                for k in kvals:
                    z = 0
                    for elem in dist[:k+1]:
                        z += elem[1]

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

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


            # rms err for k value, and append to rvals list
            for k in kvals:
                if (func, k) not in rvals:
                    rvals[(func, k)] = []
                rvals[(func, k)].append(rms_err(y_valid, y_est[k
]))
                #print(y_est[k])

                #print(rms_err(y_valid, y_est[k]))
    #print('rvals')
    #print(rvals)
    for func, k in rvals:
        regerr = sum(rvals[(func, k)]) / 5   #avg over 5 folds
        error.append((k+1, func, regerr))    #append to err list

    return error

def test(x_train, x_valid, x_test, y_train, y_valid, y_test, k,
func, plot=False):
    x_total = np.vstack([x_train, x_valid])
    y_total = np.vstack([y_train, y_valid])
    p = []
    for elem in x_test:
        # append distances between test and training points
        dist = []
        for i in range(len(x_total)):
            dist.append((func(elem, x_total[i]), y_total[i]))
        dist.sort(key=lambda x: x[0])
        y_est = 0
        for item in dist[:k]:
            y_est += item[1]
        avg = y_est/k    #average y
        p.append(avg)
    if plot:
        plt.figure(2)
        plt.plot(x_test, y_test, '-go', label='Actual')
        plt.plot(x_test, p, '-bo', label='Prediction')
        plt.title('Mauna Loa Test Prediction')
        plt.xlabel('x_test')
        plt.legend(loc='upper left')
        plt.savefig('ml_prediction.png')
```

```
    testerror = rms_err(y_test, p)
    return testerror

#model regression
def regressionmodel(dataset):


    if dataset == 'rosenbrock':
            start = time.time()
            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 = regression(x_train, x_valid, y_train, y_valid,
[l2norm])

        result.sort(key=lambda x: x[0])
        #print(result)
        k_vals = []
        error_vals = []

        for k, func, error in result:
            k_vals.append(k)
            error_vals.append(error)
            #print(error_vals)

        plt.figure(1)
        plt.plot(k_vals, error_vals, '-bo')
        plt.xlabel('k')
        plt.ylabel('Avg 5-Fold RMSE')
        plt.title('Mauna Loa L2 Norm Loss 5-Fold')
        plt.savefig('l2loss.png')

        result.sort(key=lambda x: x[2])
        k_min = result[0][0]
        method = result[0][1]
        testerror = test(x_train, x_valid, x_test, y_train, y_va
lid, y_test, k_min, method, plot=True)

        print('************************')
        print('Mauna Loa with L2 Norm :')
```

```
        print('')

        print('Optimal k: ' + str(k_min))
        print('Test RMS Error: ' + str(testerror))
        print('RMS Error 5 fold: ' + str(result[0][2]))
        print('************************')


    result = regression(x_train, x_valid, y_train, y_valid, func
tions)
    result.sort(key=lambda x: x[2])
    k_min = result[0][0]
    pref_function = result[0][1]
    testerror = test(x_train, x_valid, x_test, y_train, y_valid,
y_test, k_min, pref_function)
    #print(test_error)

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

#model classify
def classificationmodel(dataset):

    functions = [l1norm, l2norm]

    x_train, x_valid, x_test, y_train, y_valid, y_test = load_da
taset(dataset)
    onefresult = classification(x_train, y_train, x_valid, y_val
id, functions)
    onefresult.sort(key=lambda x: x[2], reverse=True)
    k_min = onefresult[0][0]
    func = onefresult[0][1]
    test_acc = classifytest(x_train, x_valid, x_test, y_train, y
_valid, y_test, k_min, func)

    return k_min, func, onefresult[0][2], test_acc


def classifytest(x_train, x_valid, x_test, y_train, y_valid, y_t
est, k, func):

    x_total = np.vstack([x_train, x_valid])
    y_total = np.vstack([y_train, y_valid])
    count = 0

    for i in range(len(x_test)):
```

```
        dist = []
        for j in range(len(x_total)):
            dist.append((func(x_test[i], x_total[j]), y_total[j]
))

        dist.sort(key=lambda x: x[0])

        inst = {}
        for q in dist[: k]:
            if str(q[1]) not in inst:
                inst[str(q[1])] = (q[1], 0)
            inst[str(q[1])] = (q[1], inst[str(q[1])][1] + 1)

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

        if np.all(occlist[0][0] == y_test[i]):
            count += 1

    return count/len(x_test)


def SVDregress(x_train, x_valid, x_test, y_train, y_valid, y_tes
t, model_type):

    if model_type == 'regression':

        x_total = np.vstack([x_train, x_valid])
        y_total = np.vstack([y_train, y_valid])
        X = np.ones((len(x_total), len(x_total[0]) + 1))
        X[:, 1:] = x_total

        #SVD
        U, S, D = np.linalg.svd(X)
        sigma = np.diag(S)
        fillzeros = np.zeros([len(x_total)-len(S), len(S)])
        sig_inverse = np.linalg.pinv(np.vstack([sigma, fillzeros
]))
        w = np.dot(D.T, np.dot(sig_inverse, np.dot(U.T, y_total)
))
        X_test = np.ones((len(x_test), len(x_test[0]) + 1))
        X_test[:, 1:] = x_test
        prediction = np.dot(X_test, w)
```

```
        result = rms_err(y_test, prediction)


    elif model_type == 'classification':

        x_total = np.vstack([x_train, x_valid])
        y_total = np.vstack([y_train, y_valid])
        X = np.ones([len(x_total), len(x_total[0]) + 1])
        X[:, 1:] = x_total

        U, S, Vh = np.linalg.svd(X)
        sig = np.diag(S)
        fillzeros = np.zeros([len(x_total) - len(S), len(S)])
        sig_inv = np.linalg.pinv(np.vstack([sig, fillzeros]))
        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

        #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

def qthree(x_total, x_test, y_total, y_test, k):

    start_time = time.time()
    p = []

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

    test_error = rms_err(y_test, predictions)

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

def rosenbrockperform():

    start_time = time.time()
    x_train, x_valid, x_test, y_train, y_valid, y_test = load_da
taset('rosenbrock', n_train=5000, d=9)
    k_min = 2
```

```
    testerror = test(x_train, x_valid, x_test, y_train, y_valid,
y_test, k_min, l2norm)
        #print(test_error)
    runtime = time.time() - start_time
        #return runtime, testerror
    print('The runtime for Rosenbrock using Brute force method f
or d=9 is ' + str(runtime) +' with test error =' + str(testerror
))

if __name__ == '__main__':
    Q1_sets = ['mauna_loa', 'rosenbrock', 'pumadyn32nm']  #regre
ssion
    Q2_sets = ['iris', 'mnist_small'] #classification


if Q1:

    print('Question 1')
    print('')
    for datas in Q1_sets:
        k_min, metric_min, fivef_rmse, test_rmse = regressionmo
del(datas)
        print('*************************')
        print( datas + ' :')
        print('')
        print('Optimal k: ' + str(k_min))
        print('Best Metric: ' + str(metric_min))
        print('Test Root Mean Square Error: ' + str(test_rmse))
        print('RMSE for 5 fold: ' + str(fivef_rmse))

        print('')


if Q2:

    print('Question 2')
    print('')
    for d_set in Q2_sets:
        k_min, metric_min, max_ratio, test_ratio = classificatio
nmodel(d_set)
        print('************************************')
        print('Results for ' + d_set + ' :')
        print('')
        print('Optimal k: ' + str(k_min))
        print('Optimal Distance Metric: ' + str(metric_min))
```

```
        print('Validation Accuracy is ' + str(max_ratio))

        print('Test Accuracy is ' + str(test_ratio))
        print('')

if Q3:

    print('Question 3')
    print('')
    print('************************************')
    result = {}
    result['k-d Tree'] = []

    rosenbrockperform()
    for d in range(2, 10):

        x_train, x_valid, x_test, y_train, y_valid, y_test = loa
d_dataset('rosenbrock', n_train=5000, d=d)
        x_total = np.vstack([x_train, x_valid])
        y_total = np.vstack([y_train, y_valid])

        runtime, test_rmse = qthree(x_total, x_test, y_total, y_
test, 5)
        result['k-d Tree'].append((d, runtime, test_rmse))
        print('Runtime for Rosebrock with KD Tree Method for d='
+ str(d)+ ' is ' + str(runtime))

    print(' ')

    m = list(result.keys())

    dvals = list(range(2, 10))
    plt.figure(3)
    plt.title(' Kd-Tree Runtimes as d is varied')
    plt.xlabel('d')
    plt.ylabel('time in seconds')

    for m in result:

        runtimes = []
        rmses = []
        for i in range(len(result[m])):
            runtimes.append(result[m][i][1])
            rmses.append(result[m][i][2])
```

```
        plt.figure(3)
        plt.plot(dvals, runtimes, '-b', label=m)

    plt.figure(3)
    plt.savefig('runtimesd.png')


if Q4:

    print('Question 4')
    print('')
    print('************************************')
    for d_set in Q1_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)

        testrmse = SVDregress(x_train, x_valid, x_test, y_train
, y_valid, y_test, 'regression')
        print('Test RMSE for ' + d_set + ': ' + str(testrmse))


    for d_set in Q2_sets:

        x_train, x_valid, x_test, y_train, y_valid, y_test = lo
ad_dataset(d_set)

        final_ratio = SVDregress(x_train, x_valid, x_test, y_tr
ain, y_valid, y_test, 'classification')
        print('Test Accuracy for ' + d_set + ': ' + str(final_r
atio))
```