CS 598: Practical Statistical Learning Homework 1

Netid: pushpit2

Question 1 KNN

a) Write a function myknn(xtest, xtrain, ytrain, k)

I have implemented the KNN algorithm from scratch. The method **myknn** can be called from the outside with training & test data and value for k. It then internally uses l2NormAvgOutput method to find the k nearest neighbors for each of the data instance in test set and then return the mean of the Y values for those k nearest neighbors (When KNNs are used for classification, the mode of the Y values of K nearest neighbors are taken, but here as we are using KNN for regression, hence taking mean of the Y values seems most accurate to me in order to do prediction in continuous space).

```
def l2NormAvgOutput(xTestInstance, xtrain, ytrain, k):
    dist = (xtrain - xTestInstance)**2
    dist = np.sum(dist, axis=1)
    dist = np.sqrt(dist)
    return np.mean(ytrain[np.argpartition(dist, k)[:k]])

def myknn(xtest, xtrain, ytrain, k):
    return np.apply_along_axis(l2NormAvgOutput, 1, xtest, xtrain, ytrain, k)
```

b) Generate 1000 observations:

- **Autoregressive covariance matrix**:
  - (i,j)th entry equal to $0.5^{|i-j|}$

```
cov = np.full(shape=[numOfIndependentVar, numOfIndependentVar], fill_value=1.0, dtype=float)
M,N = cov.shape
X, Y = np.ix_(np.arange(M), np.arange(N))
out = cov * (0.5 ** abs(X - Y))
```

- **Mean vector** (defaulted to vector given in the homework, but made it a param to the method, please see the full method at the end of this section):

```
# Default mean value as the per the homework document
mean=np.array([1, 2, 3, 4, 5])
```

- **Setting the seed:**
  - Seed is set to 1 before generating any observations, so that it should be used in generating the both X observations as well as the epsilon needed for Y.

```
# Setting seed as 1
np.random.seed(1)
```

- **X (independent variables generation)**:
  - Here I have used scipy.stats.multivariate_normal library. This helped in drawing a random sample from a multivariate normal distribution with the given mean vector and covariance matrix.

```
np.random.seed(1)
rv = multivariate_normal.rvs(mean=mean, cov=out,
size=totalNumOfObservations)
```

- **Y generation**:
  - Epsilon values are drawn from standard normal distribution (mean=0, std=1)

```
mu, sigma = 0, 1 # mean and standard deviation
epsilon = np.random.normal(mu, sigma, 1000)
y = rv[:, 0] + rv[:, 1] + ((rv[:, 2] - 2.5) ** 2) + epsilon
```

- **First 3 observations**

	X1	X2	X3	X4	X5	Y
0	-0.206456	0.625077	1.775050	2.406915	5.181349	0.019418
1	3.740236	4.405034	4.350167	4.961146	5.395203	12.697279
2	-0.869265	-0.281322	2.102837	3.417965	5.903901	-2.121640

- **Complete method:**

```
def generateObservations(numOfIndependentVar=5,
mean=np.array([1, 2, 3, 4, 5]), totalNumOfObservations=1000):
    cov = np.full(shape=[numOfIndependentVar,
numOfIndependentVar], fill_value=1.0, dtype=float)
    M,N = cov.shape
    X, Y = np.ix_(np.arange(M), np.arange(N))
    out = cov * (0.5 ** abs(X - Y))
    #Setting seed as 1
    np.random.seed(1)
```

```
    rv = multivariate_normal.rvs(mean=mean, cov=out, size=totalNumOfObservations)
    mu, sigma = 0, 1 # mean and standard deviation
    epsilon = np.random.normal(mu, sigma, 1000)
    y = rv[:, 0] + rv[:, 1] + ((rv[:, 2] - 2.5) ** 2) + epsilon
    ddf = pd.DataFrame(data=rv, columns=['X1', 'X2', 'X3', 'X4', 'X5'])
    ddf['Y'] = y
    print(ddf.head())
    print(rv[0:3, :], y[0:3])
    return rv, y
```

c) Use the first 400 observations of your data as the training data and the rest as testing data

- **Method to split the train and test data:**

```
def generateTestTrainData(X, Y, trainingSetSize=400):
    xtrain = X[:trainingSetSize, :]
    ytrain = Y[:trainingSetSize]
    xtest = X[trainingSetSize:, :]
    ytest = Y[trainingSetSize:]
    return xtrain, ytrain, xtest, ytest
```

- **Y values prediction and Mean squared error at k=5:**

```
# k = 5
yhat = myknn(xtest, xtrain, ytrain, 5)
error = np.sum((ytest-yhat) ** 2)/len(ytest)
print("K=%d, MeanSquaredError=%f"%(5, error))
```

Output:

```
K=5, MeanSquaredError=1.555822 (given that the seed is set to 1 as mentioned before)
```

d) Compare the prediction error of a linear model with your KNN model:

- The plot below shows the Mean squared error (MSE) of different KNN models generated using **k** values from (**1** to **100** [inclusive]). I have also fitted the training data to a linear regression model and the MSE for that model on test data is shown

as horizontal red line. The value for the MSE for Linear Model is `2.424736` `(given that the seed is set to 1 as mentioned before)`

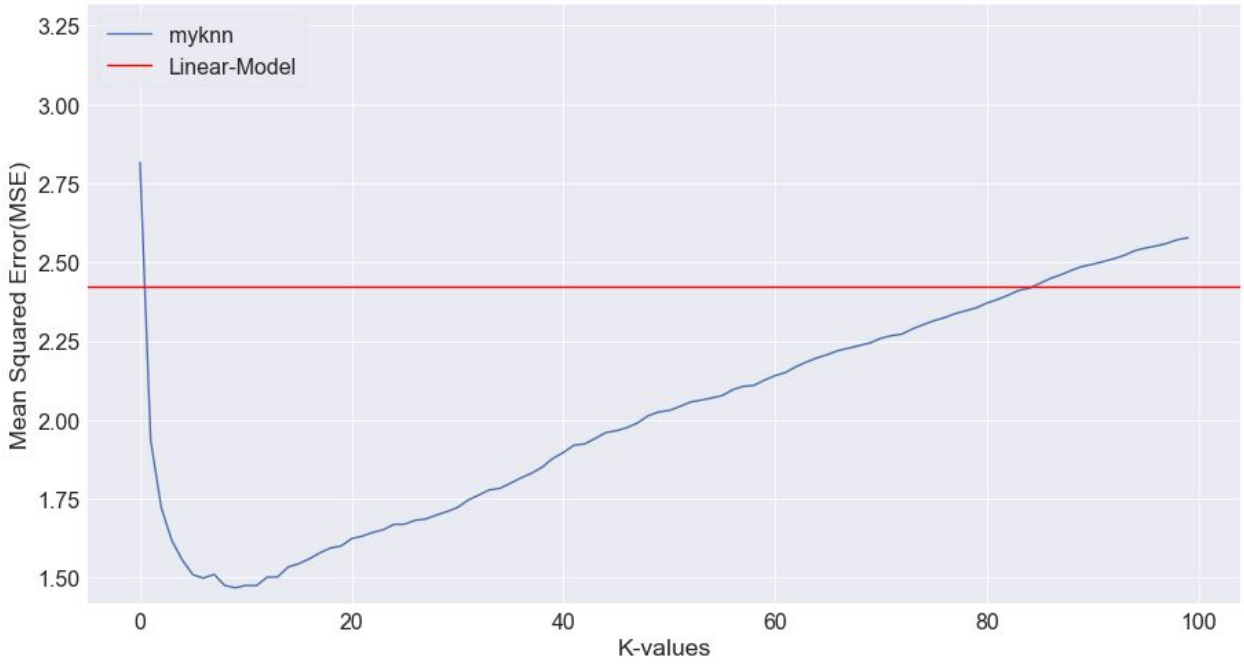

Fig. Mean squared error plot for KNN models at different K (MSE for the linear model is added as red horizontal line to demonstrate all results in a single figure)

- **Code:**

```
for k in range(101):
    yhat = myknn(xtest, xtrain, ytrain, k)
    error = np.sum((ytest-yhat) ** 2)/len(ytest)
    errors.append(error)

# Sklean-linear regression
lm = LinearRegression().fit(xtrain, ytrain)
yhat = lm.predict(xtest)
lm_error = np.sum((ytest-yhat) ** 2)/len(ytest)
print("Linear model MSE: %f" % lm_error)

# Plotting the results
plt.plot(errors[1:], label='myknn')
plt.axhline(y=lm_error, color='r', label='Linear-Model')
plt.legend(loc='upper left', frameon=True, fancybox=True)
plt.xlabel("K-values")
plt.ylabel("Mean Squared Error(MSE)")
```

```
plt.ylim(np.min(errors[1:]) - 0.05, np.max(errors[1:]) + 0.5)
plt.show()
```

Question 2 Linear Regression through Optimization

- **Custom function mylm_g(x, y, delta, epsilon, maxitr) to implement gradient descent based optimization version of linear regression**
    - This method works on the matrix, where each row represents one observation. I have made slight changes in the way we calculate the gradient descent to incorporate matrix calculations. The beta values are initialized to 0 for all parameters to initiate the gradient descent algorithm. For the check, to see if the new beta is within epsilon distance (10e-6) of the old one, I have used method (linalg.norm) present in numpy library to calculate the euclidean distance between two n-dimensional vectors (beta and betaNew). This code either runs till maxItr or till distance (between beta and betaNew) constraint is satisfied whichever comes earlier.

```
def mylm_g(X, Y, maxitr, delta=0.1, epsilon=10e-6):
    m = len(Y)
    beta = np.zeros([X.shape[1],])
    for i in range(maxitr):
        preds = X.dot(beta)
        error = preds - Y
        gradient = X.T.dot(error) / X.shape[0]
        betaNew = beta - delta * gradient
        dis = np.linalg.norm(betaNew - beta)
        beta = betaNew
        if dis < epsilon:
            print("FinalIteration: %d" % i)
            break
    return beta
```

- **Boston Housing data**
    - **Load the data:** I have used the link (provided on Piazza) to download the data, removed the columns [town, tract, medv], centered and scaled the data. And get the cmedv as Y values and rest of the columns as X.

```
def getData():
    url = 'https://maya-jha.github.io/data/BostonData.csv'
    df = pd.read_csv(url, index_col=0)
    df = df.drop(columns=['town', 'tract', 'medv'])
    df.head()
```

```
    Y = df['cmedv'].values
    X = df.loc[:, df.columns != 'cmedv'].values
    print(X.shape, Y.shape)
    X_scaled = preprocessing.scale(X)
    Y_scaled = preprocessing.scale(Y)
    print(X_scaled.shape, Y_scaled.shape)
    return X_scaled, Y_scaled
```

- **Running for different values of max-iteration:**
  - I have ran the mylm_g method for maxItr = [1, 1000] iterations and collected the estimated beta for each run. To compare it with the linear model, I have fitted the data to a LinearRegression model and collected the coefficient vector. Then I used the euclidean distance between all the beta(s) I have collected from mylm_g runs with this coefficient vector from the linear model (Please see the plot below).
  - Running for different maxIter values:

```
maxIter=1000
beta_gs = []
for itr in range(1, maxIter):
    beta_g = mylm_g(X, Y, itr)
    beta_gs.append(beta_g)
beta_gs = np.array(beta_gs)
```

  - Linear model:

```
lm = LinearRegression(fit_intercept=False).fit(X, Y)
print(lm.coef_)
```

  - Plotting the distance between estimated beta for different values of maxIteration and beta/coefficients estimated by linear model. By inspecting this plot it looks like the estimated beta values from our optimization method gotten very close to beta/coefficient estimated by linear model somewhere between 600-700 iterations. Please note that maxItr=870 is satisfying the constraint of betaNew-beta less than ϵ (10e-6) and the estimated beta is also very close to the coefficients of the linear mode(Please see the estimated beta vector and linear model coefficient vectors below)l:

```
def distance(X, Y):
    return np.linalg.norm(X - Y)

dt = np.apply_along_axis(distance, 1, beta_gs, lm.coef_)
plt.plot(dt[1:])
plt.xlabel("Max-iteration")
plt.ylabel("Euclidean distance between estimated beta and
coeficients from linear model ")
plt.ylim(np.min(dt[1:]) - 0.05, np.max(dt[1:]) + 0.05)
plt.show()
```

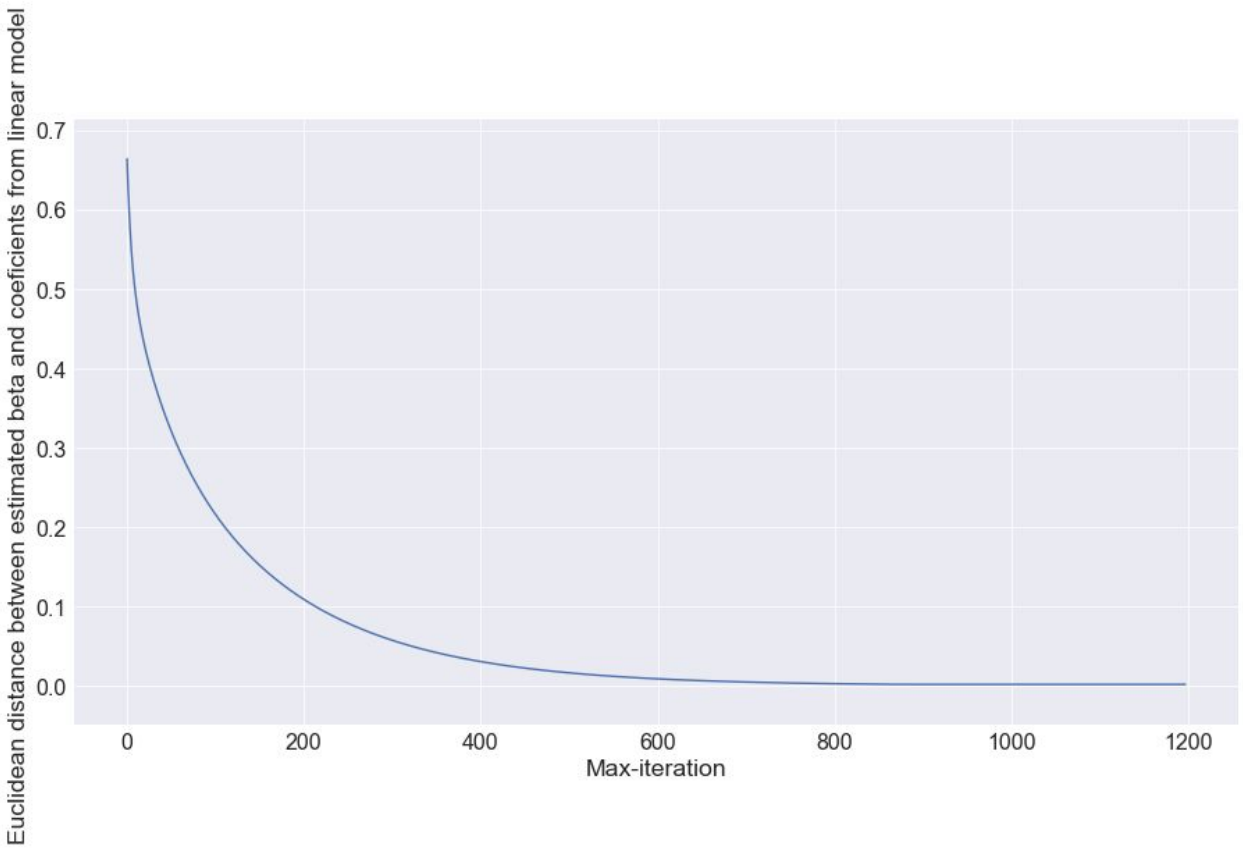

Fig: Distance between estimated beta and beta/coefficients estimated by linear model for different values of maxIteration.

Estimated beta (satisfying betaNew-betaOld < 10^{-6}):

```
Beta:    [-0.03234809    0.03021324  -0.0978601     0.11813833    0.01098584
0.07136175  -0.19961831    0.28729784    0.00752068  -0.32105484    0.28984483
-0.23540706  -0.20675364   0.09123043  -0.41793609]
```

Coefficients from Linear model:

```
Coeff:    [-0.03231644    0.03024509  -0.09793597    0.1182731     0.01139038
0.07131225  -0.19970377    0.28723281    0.00756485  -0.32103934    0.29085076
-0.23652616  -0.20680497   0.09123541  -0.41797282]
```

Bonus: The Non-scaled Version

- **Try this with your code, and comment on what you observed and explain why.**
    - I see the change in the scale of regression coefficients. The scale of the regression coefficients is different as now different predictors have varied scale and variability in values which make regression coefficient for predictors say with lower variance larger (which can also be seen by the proof below). As we are not using regularization, the scaling should not affect the model because scaling & centering will not affect their correlation with the dependent variable.
- **Can you think of a way to calculate the beta parameters on the original scale using the solution from the previous question:**

The beta values I am getting after applying the following transformation are very close to the coefficients I got on fitting Linear model to non-scaled data (Please see below).

(Y - mY) / sY = b1* (X1-mX1)/sX1 + b2*(X2 - mX2)/sX2	Eq1: Scaled and centered equation
Y = (b1 * sY / sX1) * X1 + (b2 * sY / sX2) * X2 + (mY - b1*mX1 * sY/sX1 - b2*mX2 * sY/sX2)	Eq2: Reorganized Eq1 to write in terms of original (non-scaled) predictors and dependent variable
(mY - b1*mX1 * sY/sX1 - b2*mX2 * sY/sX2) => a	This can be substituted as intercept a
Y = a + (b1 * sY / sX1)X1 + (b2 * sY / sX2) X2	Eq2 rewritten after substitution
Y = a + b1' * X1 + b2' * X2	Substitute non-scaled coefficients
⇒ b1' = (b1 * sY / sX1), b2' = (b2 * sY / sX2)	Q.E.D

Notations: m[Y/X1/X2] → mean([Y/X1/X2]), s[Y/X1/X2] → std([Y/X1/X2])

Note: I have used just two predictors here but the above proof can be generalized to any number of features(predictors), please see the code below where I applied this derived equation on Boston housing dataset.

Code:

```
url = 'https://maya-jha.github.io/data/BostonData.csv'
df = pd.read_csv(url, index_col=0)
```

```
df = df.drop(columns=['town', 'tract', 'medv'])
df.head()
df_new = df.loc[:, df.columns != 'cmedv']
df_new['cmedv'] = df['cmedv']
Y = df['cmedv'].values
X = df.loc[:, df.columns != 'cmedv'].values
all_data = df_new.values
lm = LinearRegression().fit(X, Y)
print("Non scaled linear model beta: ", lm.coef_)

# Taking the best beta from previous run
scaled_beta = beta_gs[871]

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(all_data)
std = scaler.scale_[0:15]
transformed_beta = (scaled_beta * scaler.scale_[15]) / std
print("Transformed beta: ", transformed_beta)
print("Distance from non-scaled linear model: ",
np.linalg.norm(transformed_beta - lm.coef_))
```

```
Non scaled linear model beta:  [-3.93520158e+00  4.49544140e+00
-1.04546949e-01   4.65647582e-02  1.52453460e-02   2.57801975e+00
-1.58245766e+01    3.75371170e+00  2.46765932e-03  -1.39992664e+00
3.06714509e-01  -1.28863292e-02  -8.77121177e-01    9.17619603e-03
-5.37441098e-01]

Transformed beta:  [-3.93905529e+00  4.49070794e+00 -1.04465953e-01
4.65116982e-02  1.47038993e-02    2.57980927e+00  -1.58178049e+01
3.75456150e+00  2.45325148e-03  -1.39999423e+00    3.05653719e-01
-1.28253592e-02 -8.76903477e-01  9.17569567e-03 -5.37393870e-01]
```