HW2

February 3, 2022

1 About HW2

For this HW, we mainly try to understand the KNN method in both classification and regression settings and use it to perform several real data examples. Tuning the model will help us understand the bias-variance trade-off. A slightly more challenging task is to code a KNN method yourself. For that question, you cannot use any additional package to assist the calculation.

1.1 Custom Functions for HW2

- plot_helper: Helper function to do repetivie plotting

[1]:
```
def plot_helper(train_accs, test_accs, ks, title, ylabel = '% Error'):
    fig, ax = plt.subplots(figsize=(5, 3))
    ax.plot(ks, np.array(train_accs), label="Training Error")
    ax.plot(ks, np.array(test_accs), label="Testing Error")
    ax.set_title(title)
    ax.legend(loc='lower right')
    ax.set_ylabel(ylabel)
    plt.show()
```

2 Question 1 [40 Points] KNN Classification (Diabetes)

Load the Pima Indians Diabetes Database (PimaIndiansDiabetes) from the mlbench package. It also randomly splits the data into training and testing. You should preserve this split in the analysis.

2.1 Loading Data

I modified the code provided as follows to save the whole data and the train/test splits to csv to bring into python:

"'{r pressure, echo=FALSE} #install.packages("mlbench") # run this line if you don't have the package library(mlbench) data(PimaIndiansDiabetes)

set.seed(2) trainid = sample(1:nrow(PimaIndiansDiabetes), nrow(PimaIndiansDiabetes)/2) Diab.train = PimaIndiansDiabetes[trainid,] Diab.test = PimaIndiansDiabetes[-trainid,]

write.csv(PimaIndiansDiabetes,"/Users/harrisnisar/Documents/Stat 542/HW2/data/PimaIndiansDiabetes.csv", row.names =TRUE) write.csv(Diab.train,"/Users/harrisnisar/Docume
542/HW2/data/PimaIndiansDiabetesTrain.csv", row.names =TRUE)

write.csv(Diab.test,"/Users/harrisnisar/Documents/Stat 542/HW2/data/PimaIndiansDiabetesTest.csv", row.names =TRUE) "'

2.2 Reading Data

I used the following python code to read in the csv files that I saved:

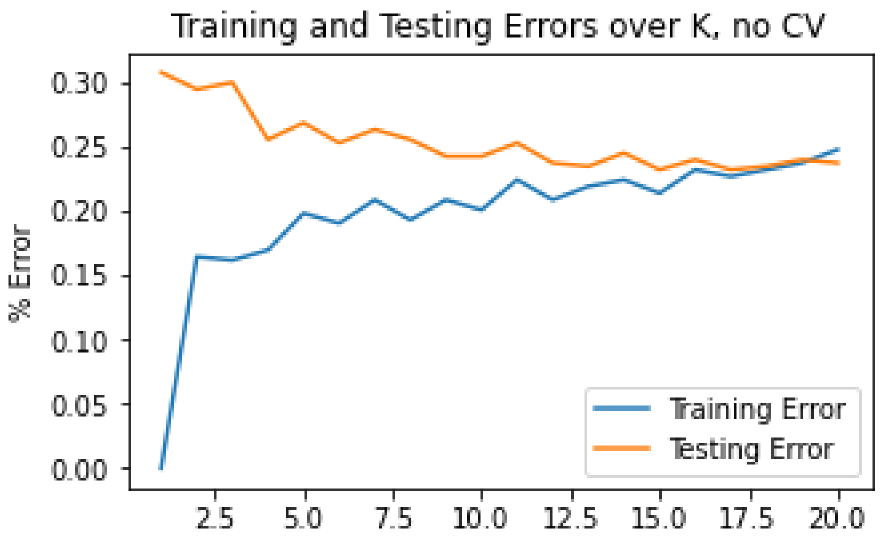
[2]:
```
import numpy as np
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
plt.rcParams['figure.figsize'] = [3, 2]
# Load data into pandas for pre-processing
pima_train_df = pd.read_csv('./data/PimaIndiansDiabetesTrain.csv', index_col=0)
pima_test_df = pd.read_csv('./data/PimaIndiansDiabetesTest.csv', index_col=0)
pima_train_df.loc[(pima_train_df.diabetes == 'pos'),'diabetes']=1
pima_train_df.loc[(pima_train_df.diabetes == 'neg'),'diabetes']=0
pima_test_df.loc[(pima_test_df.diabetes == 'pos'),'diabetes']=1
pima_test_df.loc[(pima_test_df.diabetes == 'neg'),'diabetes']=0
pima_train_matrix = np.array(pima_train_df, dtype=float)
pima_train_X = pima_train_matrix[:,:-1]
pima_train_Y = pima_train_matrix[:,-1]
pima_train_X = (pima_train_X - np.mean(pima_train_X, axis=0))/np.
 ↪std(pima_train_X, axis = 0)
pima_test_matrix = np.array(pima_test_df, dtype=float)
pima_test_X = pima_test_matrix[:,:-1]
pima_test_Y = pima_test_matrix[:,-1]
pima_test_X = (pima_test_X - np.mean(pima_test_X, axis=0))/np.std(pima_test_X,␣
 ↪axis = 0)
```

Use a grid of k values (every integer) from 1 to 20.

a) [10 pts] Fit a KNN model using `Diab.train` and calculate both training and testing errors. For the testing error, use `Diab.test`. Plot the two errors against the corresponding k values. Make sure that you differentiate them using different colors/shapes and add proper legends. Sklearn's KNeighborsClassifier was used. This gives us a score function to calcualte the accuracy of predictions. I used 1-output_from_score to calcualte the error.

[3]:
```
ks = np.arange(1,21, dtype=int)
train_results_1 = []
test_results_1 = []
for k in ks:
    neigh = KNeighborsClassifier(n_neighbors=k)
    neigh.fit(pima_train_X, pima_train_Y)
    train_results_1.append(neigh.score(pima_train_X, pima_train_Y))
    test_results_1.append(neigh.score(pima_test_X, pima_test_Y))
```

```
train_results_1 = 1-np.array(train_results_1)
test_results_1 = 1-np.array(test_results_1)
plot_helper(train_results_1, test_results_1, ks, "Training and Testing Errors␣
↪over K, no CV")
best_index = np.argmin(test_results_1)
best_k1 = ks[best_index]
best_score1 = 1-test_results_1[best_index]
print(f'No CV found {best_k1} (df={pima_test_X.shape[0]/best_k1}) to be best k␣
↪with testing error: {100.0-best_score1*100.0}%')
```



```
No CV found 15 (df=25.6) to be best k with testing error: 23.177083333333343%
```

b) [15 pts] Does the plot match (approximately) our intuition of the bias-variance trade-off in terms of having a U-shaped error? What is the optimal k value based on this result? For the optimal `k`, what is the corresponding degrees-of-freedom and its error?

The figure above shows the training error (blue) and testing error (orange) of the KNN model fit for various k's, where k represents the number of neighbors used. As expted when k = 1 (1NN), the training error is 0 and the testing error is maximum. This is a model that is overfit (performs perfect on training, but bad on testing). This means the model has a low bias and high variance. As k increases, we see that the training error increases rapidly while the testing error come down. The best k was found to be 15 (23.17% testing error, $df = \frac{n}{k} = 25.6$). After this, increasing k will probably underfit the model. Since we are looking at more neighbors, the variance of our model is low, while the bias will be high. The model does not learn the associations in the data. This will result in the testing error increasing as we increase k, matching our intuition of the bias-variance tradeoff having a U-shaped eror (testing error starts high, then goes low, then increases again).

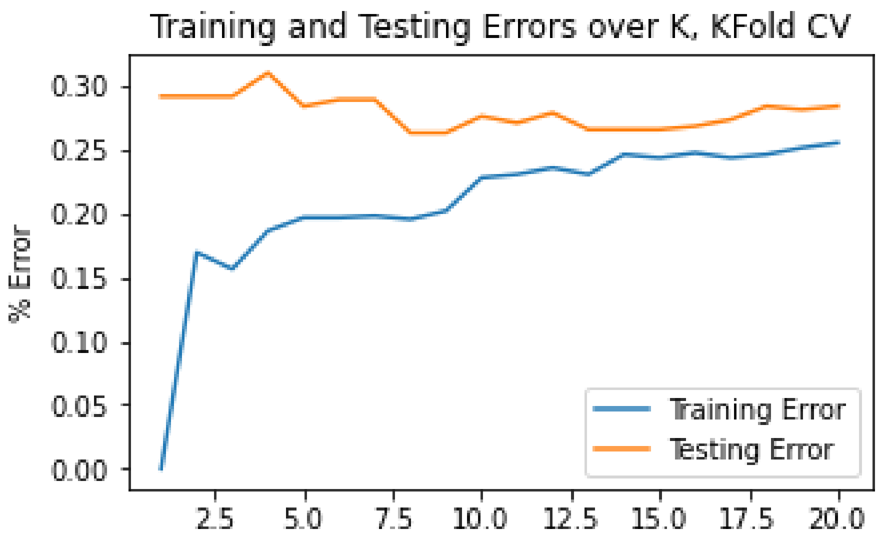
c) [15 pts] Suppose we do not have access to `Diab.test` data. Thus, we need to further split the training data into train and validation data to tune `k`. For this question, use the `caret` package to complete the tuning. You are required to

- Train the **knn** model with cross-validation using the `train()` function.
  - Specify the type of cross-validation using the `trainControl()` function. We need to use three-fold cross-validation.
  - Specify a grid of tuning parameters. This can be done using `expand.grid(k = c(1:20))`.
- Report the best parameter with its error. Compare it with your `k` in b).

Since this report is generated with Python, sklearn's KFold module was used as an alternate for the `caret` package. This module allows us to generate the splits we require to perform KFold cross validation. Essentially, for every k, we generate these splits. We iterate over these splits and use 2 of them for training and 1 for validation until all splits have been used for validation.

```
[4]: k_scores = {}
kf = KFold(n_splits=3, random_state=None, shuffle=False)
# for all k's
for k in ks:
    train_scores = []
    test_scores = []
    # for all splits, fit the model and store train and test scores
    for train_index, test_index in kf.split(pima_train_X):
        X_train, X_test = pima_train_X[train_index], pima_train_X[test_index]
        Y_train, Y_test = pima_train_Y[train_index], pima_train_Y[test_index]
        neigh = KNeighborsClassifier(n_neighbors=k)
        neigh.fit(X_train, Y_train)
        train_scores.append(neigh.score(X_train, Y_train))
        test_scores.append(neigh.score(X_test, Y_test))
    # store the averaged train and test scores across all splits
    result = {'train': np.mean(train_scores), 'test': np.mean(test_scores)}
    k_scores[k] = result

train_results_2 = [1-k_scores[k]['train'] for k in k_scores]
test_results_2 = [1-k_scores[k]['test'] for k in k_scores]
plot_helper(train_results_2, test_results_2, ks, 'Training and Testing Errors␣
 ↪over K, KFold CV')
best_index_2 = np.argmin(test_results_2)
best_k2 = ks[best_index_2]
best_score2 = 1-test_results_2[best_index_2]
print(f'KFold CV found {best_k2} (df={pima_test_X.shape[0]/best_k2}) to be best␣
 ↪k with testing error: {100.0-best_score2*100.0}%')
```



```
KFold CV found 8 (df=48.0) to be best k with testing error: 26.302083333333343%
```

When using KFold CV, the best value to use for k was found to be 8 (as opposed to 15 without CV). This gives a testing error of 26.3% which is higher than without CV (23.17%). This makes sense because KFold CV should give a more realistic idea of what our testing performance will be.

3 Question 2 [40 Points] Write your own KNN for regression

a. [10 pts] Generate $p = 5$ independent standard Normal covariates X_1, X_2, X_3, X_4, X_5 of $n = 1000$ independent observations. Then, generate Y from the regression model

$$Y = X_1 + 0.5 \times X_2 - X_3 + \epsilon,$$

with i.i.d. standard normal error ϵ. Make sure to set a random seed 1 for reproducibility.

- Use a KNN implementation from an existing package. Report the mean squared error (MSE) for your prediction with `k = 5`. Use the first 500 observations as the training data and the rest as testing data. Predict the Y values using your KNN function with `k = 5`. Mean squared error is

$$\frac{1}{N}\sum_i (y_i - \widehat{y_i})^2$$

. This question also helps you validate your own function in b). a) and b) are expected have similar (possibly not identical) results.
- Hints: this is a **regression** problem instead of a classification one.

Since this report is generated with Python, KNeighborsRegressor from sklearn was used as the implementation from an existing package.

[5]:
```
# generating the data
n = 1000
p = 5
```

```
k = 5
data = []
np.random.seed(1)
for _ in range(p):
    data.append(np.random.normal(size=(n)))
X = np.array(data).T
beta = np.array([1,0.5,-1,0,0])
epsilon = np.random.normal(size=(n))
y = X @ beta + epsilon
X_train = X[:int(n/2),:]
Y_train = y[:int(n/2)]
X_test = X[int(n/2):,:]
Y_test = y[int(n/2):]

# custom helper function to find mse
def mse(actual, preds):
    return np.sum(np.power((actual-preds),2)) / len(preds)

# model fitting
from sklearn.neighbors import KNeighborsRegressor
neigh = KNeighborsRegressor(n_neighbors=k)
neigh.fit(X_train, Y_train)
Y_train_pred = neigh.predict(X_train)
Y_test_pred = neigh.predict(X_test)
mse_train = mse(Y_train, Y_train_pred)
mse_test = mse(Y_test, Y_test_pred)
print(f'Training MSE: {mse_train}\nTesting MSE: {mse_test}')
```

```
Training MSE: 0.9614093562145081
Testing MSE: 1.2345716288770519
```

The training MSE was 0.961 and the testing MSE (prediction MSE) was 1.235. As expected, the model does better on the training data than the testing data. After all, it is fitted on the training data.

b. [30 pts] For this question, you **cannot** use (load) any additional `R` package. Write your own function `myknn(xtrain, ytrain, xtest, k)` that fits a KNN model and predict multiple target points `xtest`. The function should return a variable `ytest`.

- Here, `xtrain` is the training dataset covariate value, `ytrain` is the training data outcome, and `k` is the number of nearest neighbors. `ytest` is the prediction on `xtest`.
- Use Euclidean distance to calculate the closeness between two points.
- Test your code by reporting the mean square error on the testing data.

[6]:
```
class MyKNN:
    def __init__(self, k):
        self.k = k

    def fit(self, X, Y):
        self.X = X
        self.Y = Y

    def predict(self, X_to_preds):
        # 1. for every point to predict
        preds = []
        for X_to_pred in X_to_preds:
            # 2. calculate the distance between that point and all training␣
↪points
            distances = np.linalg.norm((X_to_pred - self.X), axis=1)
            # 3. find k smallest distances (https://stackoverflow.com/questions/
↪34226400/find-the-index-of-the-k-smallest-values-of-a-numpy-array_)
            idx = np.argpartition(distances, k)
            smallest_idx = idx[:k]
            # 4. calculate the mean of the label of those closest points
            preds.append(np.mean(self.Y[smallest_idx]))
        return np.array(preds)

myKNN = MyKNN(k)
myKNN.fit(X_train, Y_train)
Y_train_pred = myKNN.predict(X_train)
Y_test_pred = myKNN.predict(X_test)
mse_train = mse(Y_train, Y_train_pred)
mse_test = mse(Y_test, Y_test_pred)
print(f'Training MSE: {mse_train}\nTesting MSE: {mse_test}')
```

```
Training MSE: 0.9614093562145081
Testing MSE: 1.2345716288770519
```

The training MSE was 0.961 and the testing MSE (prediction MSE) was 1.235. These are the same results as the sklearn implementation, a good sign that the code is behaving as expected.

4 Question 3 [30 Points] Curse of Dimensionality

Let's consider a high-dimensional setting. Keep the data-generating model the same as question 2. In addition to the outcomes and covariates from question 2, we will also generate 95 more noisy variables to make p = 100. In this question, you can use a KNN function from any existing package.

We consider two different settings to generate that additional set of 95 covariates. Make sure to set random seeds for reproducibility.

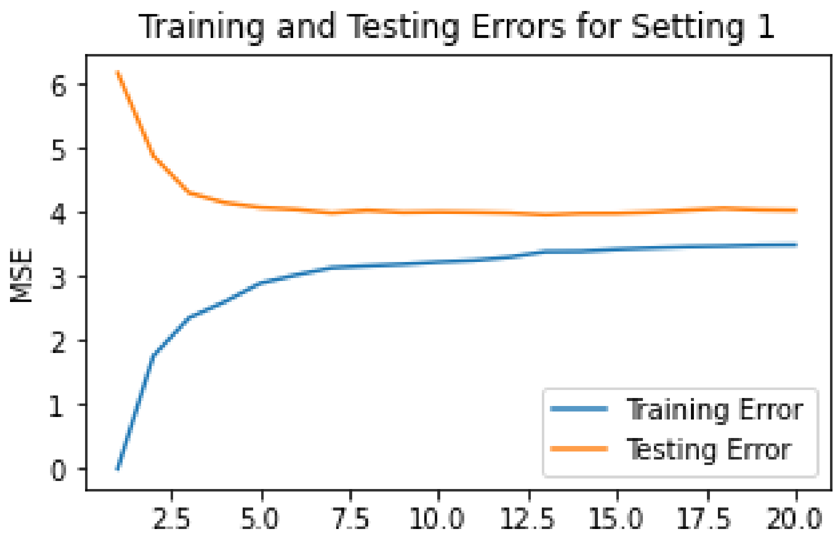
Fit KNN in both settings (with the total of 100 covariates) and select the best k value. Answer the following questions

- Generate another 95-dimensional covariates with all independent standard Gaussian entries.

[7]:
```
# generate covariates and add them to what already have
more_covariates = []
for _ in range(95):
    more_covariates.append(np.random.normal(size=(n)))
more_covariates = np.array(more_covariates).T
X1 = np.hstack((X, more_covariates))
# generate beta and noise to generate y
beta1 = np.concatenate((beta, np.zeros(95)))
np.random.seed(1)
epsilon1 = np.random.normal(size=(1000))
Y1 = X1 @ beta1 + epsilon1
#split
X1_train = X1[:int(n/2),:]
Y1_train = Y1[:int(n/2)]
X1_test = X1[int(n/2):,:]
Y1_test = Y1[int(n/2):]
#experiment
train_errors1 = []
test_errors1 = []

for k in ks:
    neigh1 = KNeighborsRegressor(n_neighbors=k)
    neigh1.fit(X1_train, Y1_train)
    Y_train_pred = neigh1.predict(X1_train)
    Y_test_pred = neigh1.predict(X1_test)
    train_errors1.append(mse(Y_train_pred, Y1_train))
    test_errors1.append(mse(Y_test_pred, Y1_test))

plot_helper(train_errors1, test_errors1, ks, "Training and Testing Errors for␣
 ↪Setting 1", "MSE")
best_index_1 = np.argmin(test_errors1)
best_k1 = ks[best_index_1]
best_score1 = test_errors1[best_index_1]
print(f'Setting 1 found {best_k1} to be best k with testing error:␣
 ↪{best_score1}')
```



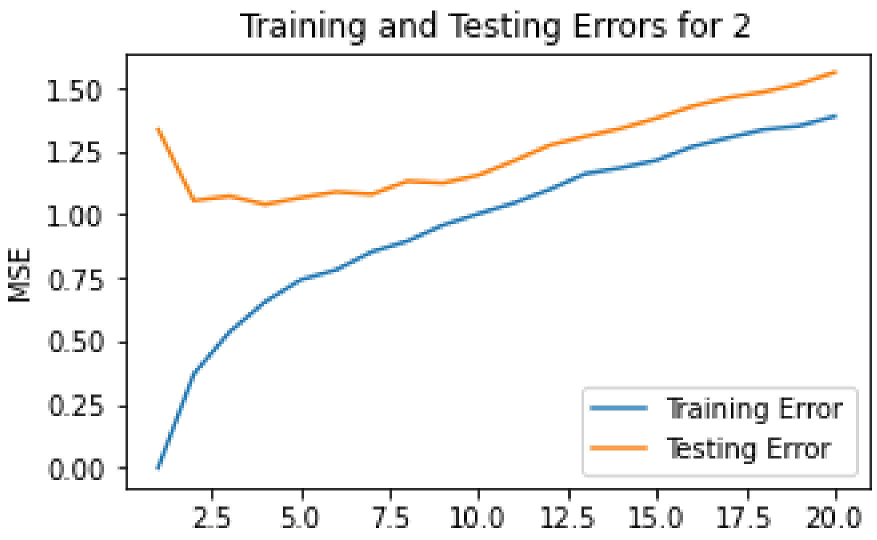
```
Setting 1 found 13 to be best k with testing error: 3.9615008868605694
```

- Generate another 95-dimensional covariates using the formula $X^T A$, where X is the original 5-dimensional vector, and A is a 5×95 dimensional (fixed) matrix that remains the same for all observations. You should generate A just once using i.i.d. uniform $[0, 1]$ entries and then apply A to your current 5-dimensional data.

[8]:
```python
# generate covariates and add them to what already have
A = np.random.uniform(size=(5,95))
more_covariates2 = X @ A
X2 = np.hstack((X, more_covariates2))
# generate beta and noise to generate y
beta2 = np.concatenate((beta, np.zeros(95)))
np.random.seed(1)
epsilon2 = np.random.normal(size=(1000))
Y2 = X2 @ beta2 + epsilon2
# splits
X2_train = X2[:int(n/2),:]
Y2_train = Y2[:int(n/2)]
X2_test = X2[int(n/2):,:]
Y2_test = Y2[int(n/2):]
# experiment
train_errors2 = []
test_errors2 = []

for k in ks:
    neigh2 = KNeighborsRegressor(n_neighbors=k)
    neigh2.fit(X2_train, Y2_train)
    Y_train_pred = neigh2.predict(X2_train)
    Y_test_pred = neigh2.predict(X2_test)
    train_errors2.append(mse(Y_train_pred, Y2_train))
    test_errors2.append(mse(Y_test_pred, Y2_test))

plot_helper(train_errors2, test_errors2, ks, "Training and Testing Errors for␣
↪2", "MSE")
best_index_2 = np.argmin(test_errors2)
best_k2 = ks[best_index_2]
best_score2 = test_errors2[best_index_2]
print(f'Setting 2 found {best_k2} to be best k with testing error:␣
↪{best_score2}')
```



```
Setting 2 found 4 to be best k with testing error: 1.0404314402313848
```

a. [10 pts] For the first setting, what is the best k and the best mean squared error for prediction? Setting 1 found 15 to be best k with mse of 3.961 for prediction.

b. [10 pts] For the second setting, what is the best k and the best mean squared error for prediction? Setting 2 found 9 to be best k with mse of 1.04 for prediction.

c. [10 pts] In which setting kNN performs better? Why? Setting 2 performed better. This is because the first setting introduced new variables, while the second setting introduced variables that were simply linear combinations of the first 5. The first setting therefore suffered from the curse of dimensionality. Put another way, as we increase our variables and go towards a higher dimensional space, it becomes harder and harder to discern the closeness of one point with another (the points seem all close to each other). This is the very basis of KNN. We assign labels based on the closest points. If all points in higher dimesnsions are similar in distance, we cannot do this. The second setting does not suffer from this because, while we do add 95 new covariates, they are simply linear combinations of the first covariates. This means that the data has some underlying structure (manifold) that is easier to learn.

This is very similar to why KNN does not perform poorly when classifying MNIST data.