18-752 Project Report
NO_2 Concentration Prediction

Siddhant Jagtap Dong Hyun Lee

Problem Statement:

To predict the Nitrogen Dioxide concentration in ppm (parts per million) by applying regression techniques on other quantities measured by an air quality sensor.

Data Collection:

A raw dataset titled 'Allegheny County Air Quality' was downloaded from the Western Pennsylvania Data Center website. It includes air quality data collected by 13 air quality sensors installed at different locations in the county over a span of 4 years. However, our project is based on data collected by the 'Parkway East' sensor from January 2016 to April 2020. Since the data was in raw format, significant pre-processing was done. This included removal of Nan or blank entries and normalization of data points by subtracting the mean and dividing by the standard deviation.

Data Visualization:

The data was initially visualized by plotting the target label 'NO2' against each of the individual features available in the dataset. It was observed that certain features display a relatively strong positive correlation with the target. 3-D plots of the target versus 2 input features were also plotted using Matlab. Interestingly, the plot of 'NO', 'NOX' and 'NO2' roughly resembles a plane. Please refer to Appendix C.1 for screenshots of the MATLAB Plots.

Investigations:

The following regression methods were applied to the above-mentioned dataset:

Method 1: Normal Discriminative Model

Initially, a normal discriminative model was trained using all 8 features present in the dataset. However, since this model did not perform well on the test dataset, another linear model was trained using only those features which had a strong positive correlation with the target 'NO2'. Therefore, features like 'CO', 'NO', 'NOX' and 'UVPM' were used for predicting the concentration of NO_2. 80% of the data points were used for 10-fold cross-validation and the remaining were used as test data points.

In order to train a model that can generalize well, ridge regression was implemented with an L2 regularization penalty λ. The optimal value of λ is the value of the regularization penalty at which validation loss is minimum (Appendix C.2). The performance of this model can be summarized as follows:

- Normalized MSE = 0.0171653240897406
- Optimal Regularization Penalty (λ) = 2.65

Furthermore, we also tried to analyze the dependence of the target on feature 'NOX' which displays very high positive correlation with the target. A high normalized test MSE was obtained when 'NOX' was not included in the training process. The results can be summarized as follows:

- Normalized MSE = 0.357989253150152
- Optimal Regularization Penalty (λ) = 442.08

Method 2: Regression using K Nearest Neighbors

Since the normal discriminative model did not work well with all the features, we tried to capture the non-linear structure of the dataset by applying a KNN regression model which is non-linear. The dataset was split into 75% for training, and 15% for validation and the remainder for testing. Hyperparameters like distance metric and value of K were varied to minimize the validation error. When all the 8 features were used, the results obtained can be summarized as follows:

- Manhattan Distance, K = 1: Normalized MSE = 0.239265
- Euclidean Distance, K = 1: Normalized MSE = 0.261311

As the normalized MSE was quite high with all the features, we tried to investigate further by applying the KNN regression algorithm using features included in the normal discriminative model. The results using just 4 features were as follows:

- Manhattan Distance, K = 1: Normalized MSE = 0.148939
- Euclidean Distance, K = 1: Normalized MSE = 0.147064

Please refer to Appendix C.3 for the plot of train/validation MSE vs K

Method 3: Regression Model using Feedforward Neural Network

The best way to capture the regression model for the air quality data is to use a simple feedforward neural network. By implementing a multi-layer perceptron, we could easily capture the regression model of nitrogen dioxide using air quality features. The structure of the neural network is simple with 3 linear layers and 300 hidden units in each layer. The ReLU activation function was used between linear layers to activate the hidden units. The ReLU activation function is used because it is a non-linear activation function and does not activate all the neurons at the same time.

The dataset was normalized using min max scaler (Appendix A.3). Also, it was split into 80% for training, 10% for the validation and the remainder for the testing. Since the goal was to capture the regression model, the model used MSE Loss function to calculate the error. For the optimizer, we used SGD (online algorithm) optimizer with learning rate 0.01. We did not use any of the learning rate schedulers to train the neural network. We trained for 200 epochs with a batch size of 32.

- Normalized MSE for the training = 0.011
- Normalized MSE for the validation = 0.0189
- Normalized MSE for the testing = 0.00706 (Appendix C.4 for the Predictions vs Data plot)

Please refer to Appendix C.4 for the plot of train/validation MSE loss vs Epoch plot

Conclusion:

The problem was solved successfully as a test MSE of less than 10% was obtained with at least one of the methods. We would like to list a few key observations we made through the course of this project. Firstly, the normal discriminative model (without any feature transformations) cannot capture the non-linearities in the dataset. Secondly, the KNN regression model suffers from the curse of dimensionality. Furthermore, the neural network gave the lowest normalized test MSE and this value decreased with increasing the number of layers in the network. Last but not the least, performing data normalization is a precursor to all the above-mentioned regression methods.

Appendix

A. Mathematical Equations

A.1. Normal Discriminative Model:

$$L(x, y) = \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2$$

$$\text{where } h_\theta x_i = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 + \theta_4 x_4^4$$

$$L(x, y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

Loss Function

$$\text{If } \lambda > 0,$$

$$\theta = \left(X^T X + \lambda \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \right)^{-1} X^T y$$

L2 Regularization

A.2. Regression using K Nearest Neighbors:

$$d = |\mathbf{x} - \mathbf{y}| = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2} .$$

Euclidean Distance Formula

$$D = \sum_{i=1}^{n} |x_i - y_i|$$

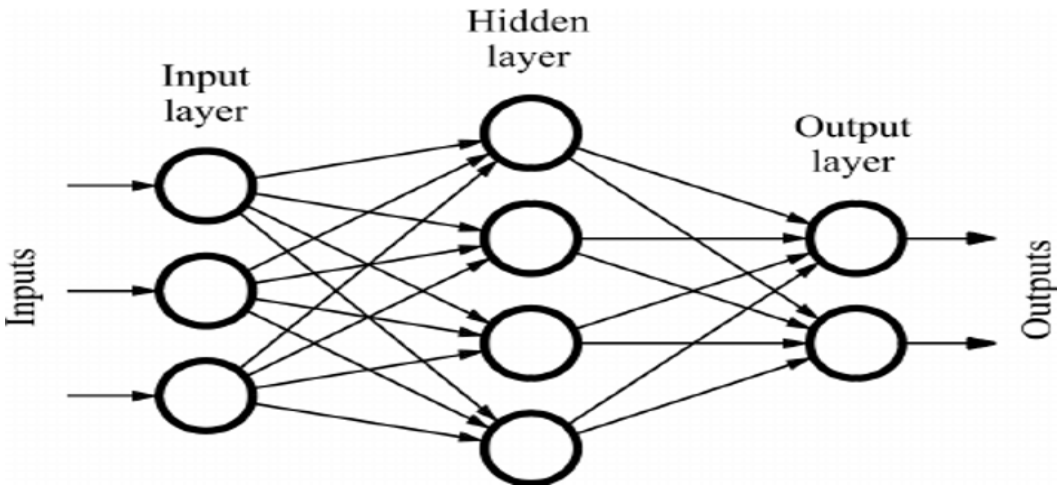
Manhattan Distance Formula

A.3. Regression using Feedforward Neural Network:

$$z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Min Max Scaler Equation

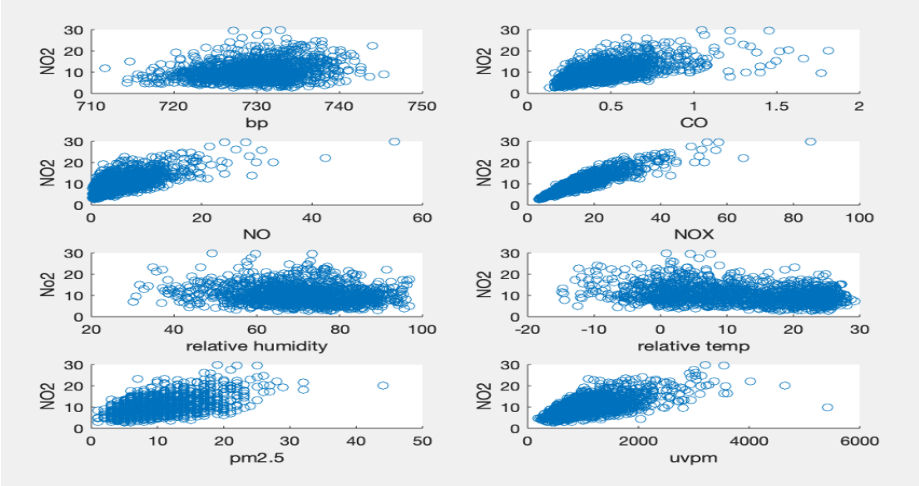
B. Diagrams:



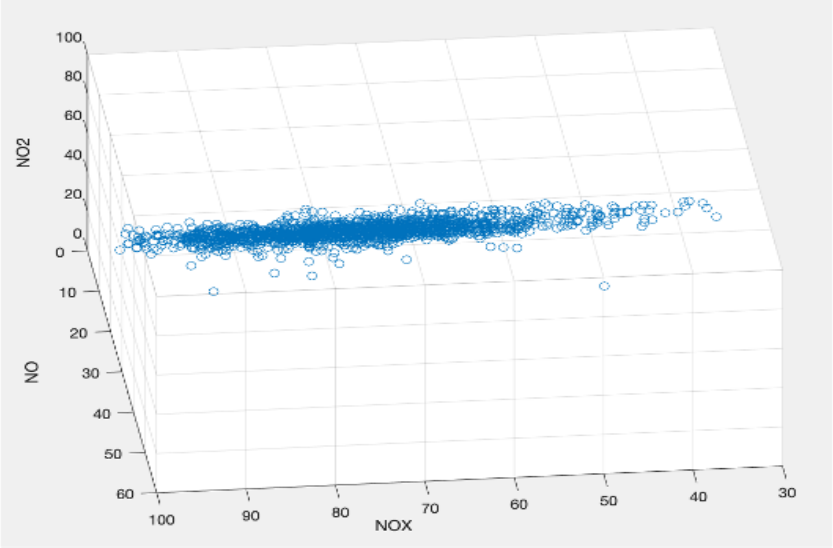
3 Hidden Layer 300 Hidden Units Per Layer

C. Plots and Charts:

C.1. Data Visualization:

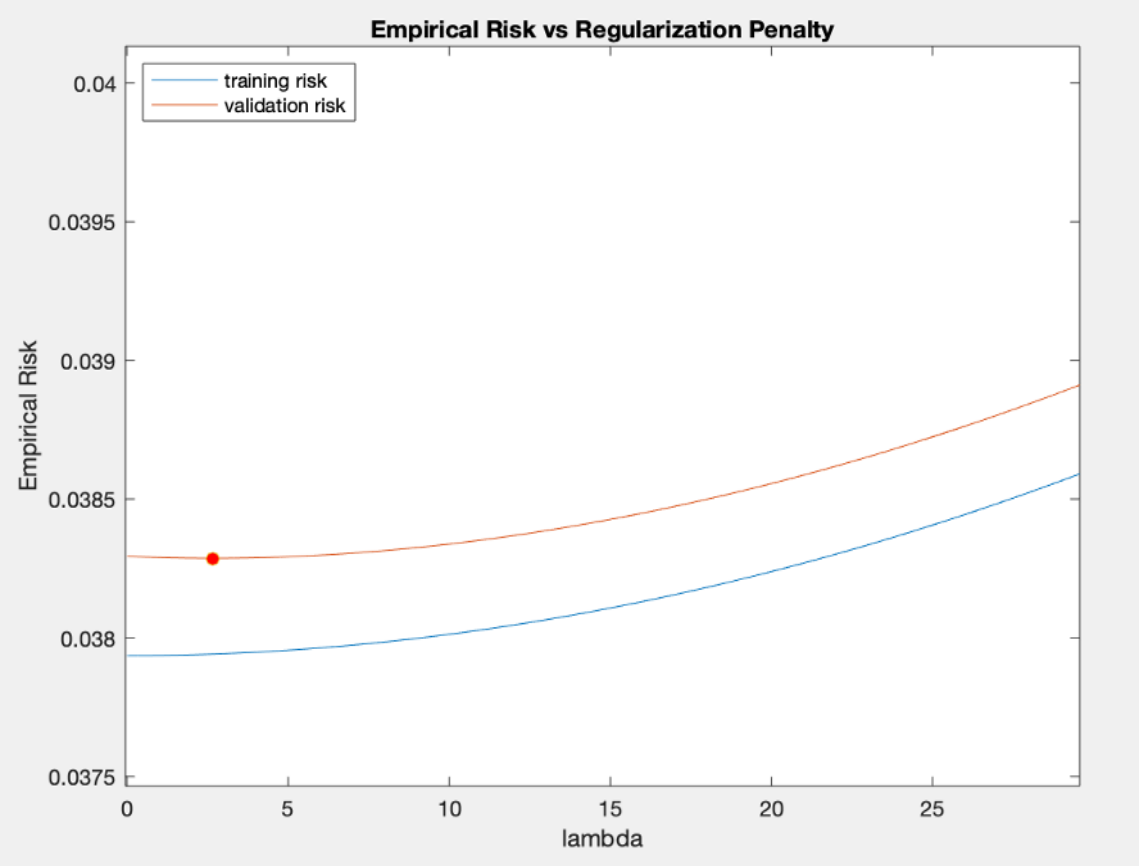


2-D plots of target v.s. individual inputs



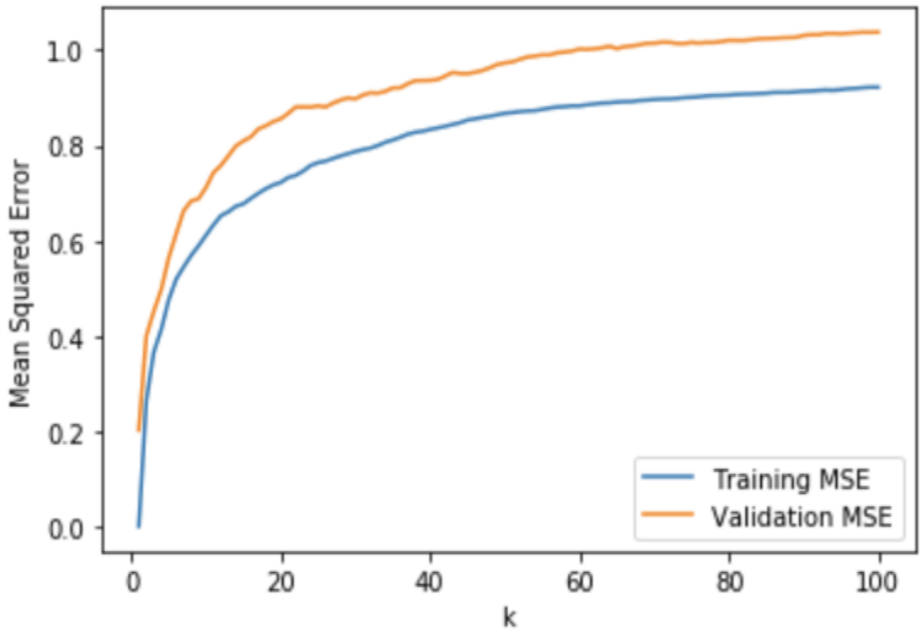
3-D plot of ‘NO2’ v.s. ‘NO’ and ‘NOX’

C.2. Normal Discriminative Model:



Empirical Risk v.s. Regularization Penalty

C.3. Regression using K Nearest Neighbors Regression:

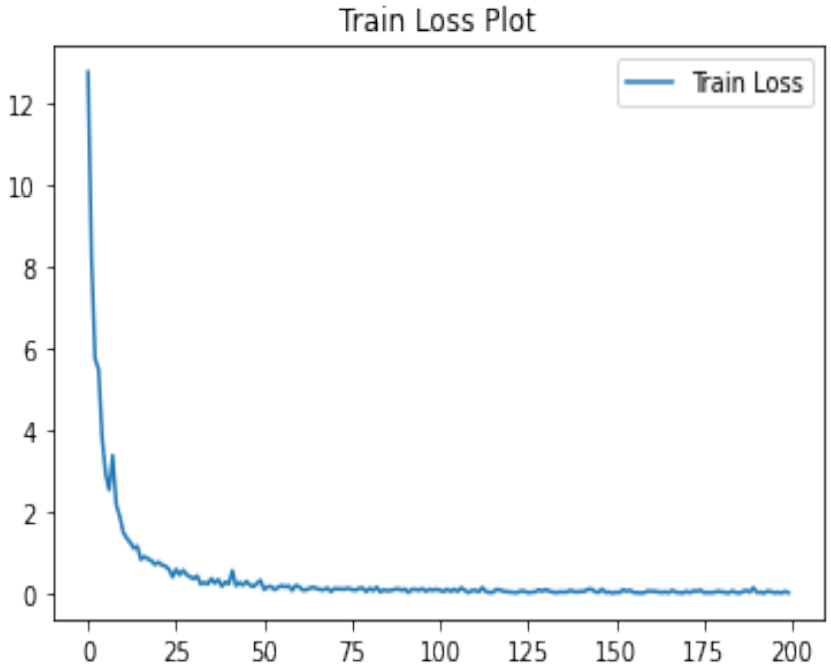


MSE v.s. k (For 8 features and Manhattan Distance)

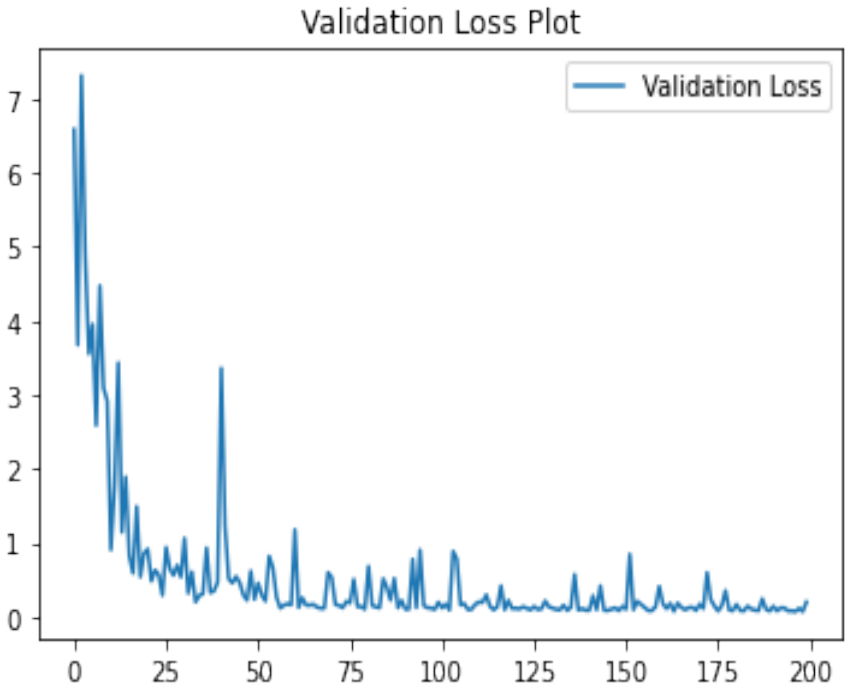
C.4. Regression using Feedforward Neural Network:

The FeedForward Neural Network MSE Loss VS Epoch plot

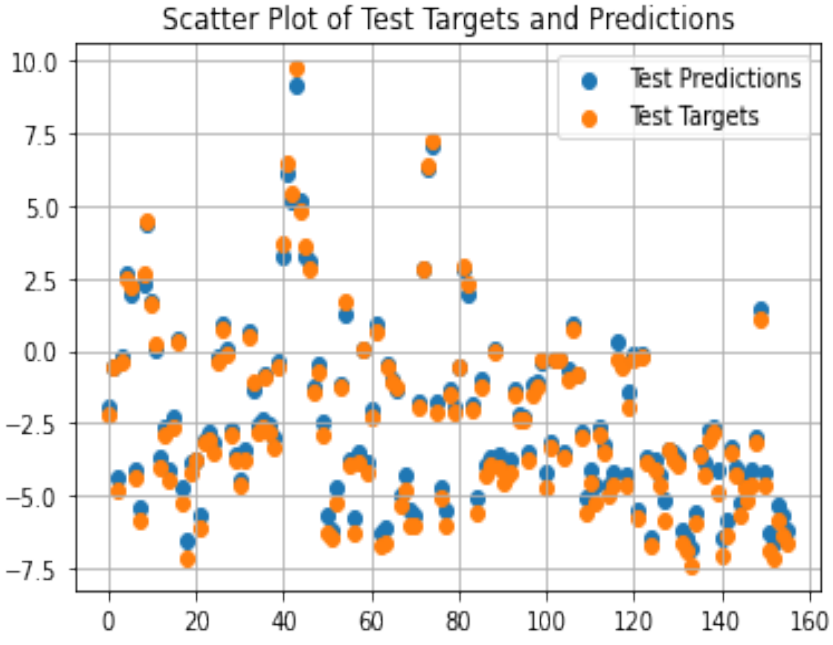
Train:



Validation:



Test Predictions vs Test Data Plot



D. Software Code:

D.1. Normal Discriminative Model (MATLAB):

```
clack;
clear all;
close all;

d_matrix = readmatrix('csv_file_path');
feature_matrix = [d_matrix(:,2:3) d_matrix(:,5) d_matrix(:,9)];
label_column = d_matrix(:,4);

% Split dataset into 2 sets for cross-validation and training:
n_samples = size(d_matrix,1);
prop_crossvalid = (ceil(0.8*n_samples)-4);
prop_test = n_samples - (prop_crossvalid);

% Feature Matrices:
cvalid_features = [ones(prop_crossvalid,1) feature_matrix(1:prop_crossvalid,:)];
test_features = feature_matrix((prop_crossvalid+1):size(feature_matrix,1),:);
test_features = [ones(size(test_features,1),1) test_features];

% Label Columns:
cvalid_labels = label_column(1:prop_crossvalid,:);
test_labels = label_column((1+prop_crossvalid):size(label_column,1),:);

% Performing 10 fold cross-validation for different L2 regularization penalties:
k = 10;
fold_size = size(cvalid_labels,1)/k;
lambda = 0:0.01:1000;
D = eye(size(cvalid_features,2));
D(1,1)=0;
train_mse_list = zeros(1,size(lambda,2));
crossvalid_mse_list = zeros(1,size(lambda,2));
theta_mle_collection = [];

% Training using 9 folds each time:
for i = 1:size(lambda,2)
    % First Fold:
    train_f1 = cvalid_features(1:(9*fold_size),:);
    train_l1 = cvalid_labels(1:(9*fold_size),:);
    valid_f1 = cvalid_features((9*fold_size+1):(10*fold_size),:);
    valid_l1 = cvalid_labels((9*fold_size+1):(10*fold_size),:);
```

```
    theta_mle1 = (inv(train_f1'*train_f1 + lambda(1,i)*D))*train_f1'*train_l1;
    valid_p1 = valid_f1*theta_mle1;
    valid_mse_1 = (1/size(valid_l1,1))*(sum((valid_l1 - valid_p1).^2));
    train_mse_1 = (1/size(train_l1,1))*(sum((train_l1 - (train_f1*theta_mle1)).^2));
    % Second Fold:
    train_f2 = [cvalid_features(1:(8*fold_size),:);cvalid_features((9*fold_size
+1):(10*fold_size),:)];
    train_l2 = [cvalid_labels(1:(8*fold_size),:);cvalid_labels((9*fold_size +1):(10*fold_size),:)];
    valid_f2 = cvalid_features((8*fold_size+1):(9*fold_size),:);
    valid_l2 = cvalid_labels((8*fold_size+1):(9*fold_size),:);
    theta_mle2 = (inv(train_f2'*train_f2 + lambda(1,i)*D))*train_f2'*train_l2;
    valid_p2 = valid_f2*theta_mle2;
    valid_mse_2 = (1/size(valid_l2,1))*(sum((valid_l2 - valid_p2).^2));
    train_mse_2 = (1/size(train_l2,1))*(sum((train_l2 - (train_f2*theta_mle2)).^2));
    % Third Fold:
    train_f3 = [cvalid_features(1:(7*fold_size),:);cvalid_features((8*fold_size
+1):(10*fold_size),:)];
    train_l3 = [cvalid_labels(1:(7*fold_size),:);cvalid_labels((8*fold_size +1):(10*fold_size),:)];
    valid_f3 = cvalid_features((7*fold_size+1):(8*fold_size),:);
    valid_l3 = cvalid_labels((7*fold_size+1):(8*fold_size),:);
    theta_mle3 = (inv(train_f3'*train_f3 + lambda(1,i)*D))*train_f3'*train_l3;
    valid_p3 = valid_f3*theta_mle3;
    valid_mse_3 = (1/size(valid_l3,1))*(sum((valid_l3 - valid_p3).^2));
    train_mse_3 = (1/size(train_l3,1))*(sum((train_l3 - (train_f3*theta_mle3)).^2));
    % Fourth Fold:
    train_f4 = [cvalid_features(1:(6*fold_size),:);cvalid_features((7*fold_size
+1):(10*fold_size),:)];
    train_l4 = [cvalid_labels(1:(6*fold_size),:);cvalid_labels((7*fold_size +1):(10*fold_size),:)];
    valid_f4 = cvalid_features((6*fold_size+1):(7*fold_size),:);
    valid_l4 = cvalid_labels((6*fold_size+1):(7*fold_size),:);
    theta_mle4 = (inv(train_f4'*train_f4 + lambda(1,i)*D))*train_f4'*train_l4;
    valid_p4 = valid_f4*theta_mle4;
    valid_mse_4 = (1/size(valid_l4,1))*(sum((valid_l4 - valid_p4).^2));
    train_mse_4 = (1/size(train_l4,1))*(sum((train_l4 - (train_f4*theta_mle4)).^2));
    % Fifth Fold:
    train_f5 = [cvalid_features(1:(5*fold_size),:);cvalid_features((6*fold_size
+1):(10*fold_size),:)];
    train_l5 = [cvalid_labels(1:(5*fold_size),:);cvalid_labels((6*fold_size +1):(10*fold_size),:)];
    valid_f5 = cvalid_features((5*fold_size+1):(6*fold_size),:);
    valid_l5 = cvalid_labels((5*fold_size+1):(6*fold_size),:);
    theta_mle5 = (inv(train_f5'*train_f5 + lambda(1,i)*D))*train_f5'*train_l5;
    valid_p5 = valid_f5*theta_mle5;
```

```
    valid_mse_5 = (1/size(valid_l5,1))*(sum((valid_l5 - valid_p5).^2));
    train_mse_5 = (1/size(train_l5,1))*(sum((train_l5 - (train_f5*theta_mle5)).^2));
    % Sixth Fold:
    train_f6 = [cvalid_features(1:(4*fold_size),:);cvalid_features((5*fold_size
+1):(10*fold_size),:)];
    train_l6 = [cvalid_labels(1:(4*fold_size),:);cvalid_labels((5*fold_size +1):(10*fold_size),:)];
    valid_f6 = cvalid_features((4*fold_size+1):(5*fold_size),:);
    valid_l6 = cvalid_labels((4*fold_size+1):(5*fold_size),:);
    theta_mle6 = (inv(train_f6'*train_f6 + lambda(1,i)*D))*train_f6'*train_l6;
    valid_p6 = valid_f6*theta_mle6;
    valid_mse_6 = (1/size(valid_l6,1))*(sum((valid_l6 - valid_p6).^2));
    train_mse_6 = (1/size(train_l6,1))*(sum((train_l6 - (train_f6*theta_mle6)).^2));
    % Seventh Fold:
    train_f7 = [cvalid_features(1:(3*fold_size),:);cvalid_features((4*fold_size
+1):(10*fold_size),:)];
    train_l7 = [cvalid_labels(1:(3*fold_size),:);cvalid_labels((4*fold_size +1):(10*fold_size),:)];
    valid_f7 = cvalid_features((3*fold_size+1):(4*fold_size),:);
    valid_l7 = cvalid_labels((3*fold_size+1):(4*fold_size),:);
    theta_mle7 = (inv(train_f7'*train_f7 + lambda(1,i)*D))*train_f7'*train_l7;
    valid_p7 = valid_f7*theta_mle7;
    valid_mse_7 = (1/size(valid_l7,1))*(sum((valid_l7 - valid_p7).^2));
    train_mse_7 = (1/size(train_l7,1))*(sum((train_l7 - (train_f7*theta_mle7)).^2));
    % Eighth Fold:
    train_f8 = [cvalid_features(1:(2*fold_size),:);cvalid_features((3*fold_size
+1):(10*fold_size),:)];
    train_l8 = [cvalid_labels(1:(2*fold_size),:);cvalid_labels((3*fold_size +1):(10*fold_size),:)];
    valid_f8 = cvalid_features((2*fold_size+1):(3*fold_size),:);
    valid_l8 = cvalid_labels((2*fold_size+1):(3*fold_size),:);
    theta_mle8 = (inv(train_f8'*train_f8 + lambda(1,i)*D))*train_f8'*train_l8;
    valid_p8 = valid_f8*theta_mle8;
    valid_mse_8 = (1/size(valid_l8,1))*(sum((valid_l8 - valid_p8).^2));
    train_mse_8 = (1/size(train_l8,1))*(sum((train_l8 - (train_f8*theta_mle8)).^2));
    % Ninth Fold:
    train_f9 = [cvalid_features(1:(1*fold_size),:);cvalid_features((2*fold_size
+1):(10*fold_size),:)];
    train_l9 = [cvalid_labels(1:(1*fold_size),:);cvalid_labels((2*fold_size +1):(10*fold_size),:)];
    valid_f9 = cvalid_features((1*fold_size+1):(2*fold_size),:);
    valid_l9 = cvalid_labels((1*fold_size+1):(2*fold_size),:);
    theta_mle9 = (inv(train_f9'*train_f9 + lambda(1,i)*D))*train_f9'*train_l9;
    valid_p9 = valid_f9*theta_mle9;
    valid_mse_9 = (1/size(valid_l9,1))*(sum((valid_l9 - valid_p9).^2));
    train_mse_9 = (1/size(train_l9,1))*(sum((train_l9 - (train_f9*theta_mle9)).^2));
```

```
    % Last Fold:
    train_f10 = cvalid_features((fold_size+1):(10*fold_size),:);
    train_l10 = cvalid_labels((fold_size+1):(10*fold_size),:);
    valid_f10 = cvalid_features(1:(fold_size),:);
    valid_l10 = cvalid_labels(1:(fold_size),:);
    theta_mle10 = (inv(train_f10'*train_f10 + lambda(1,i)*D))*train_f10'*train_l10;
    valid_p10 = valid_f10*theta_mle10;
    valid_mse_10 = (1/size(valid_l10,1))*(sum((valid_l10 - valid_p10).^2));
    train_mse_10 = (1/size(train_l10,1))*(sum((train_l10 - (train_f10*theta_mle10)).^2));

    % Calculate avg. valid MSE:
    avg_valid_mse = (valid_mse_1 + valid_mse_2 + valid_mse_3 + valid_mse_4 + valid_mse_5
+ valid_mse_6 + valid_mse_7 + valid_mse_8 + valid_mse_9 + valid_mse_10)/10;
    avg_train_mse = (train_mse_1 + train_mse_2 + train_mse_3 + train_mse_4 + train_mse_5 +
train_mse_6 + train_mse_7 + train_mse_8 + train_mse_9 + train_mse_10)/10;
    theta_mle_avg = (0.1)*(theta_mle1 + theta_mle2 + theta_mle3 + theta_mle4 + theta_mle5 +
theta_mle6 + theta_mle7 + theta_mle8 + theta_mle9 + theta_mle10);
    theta_mle_collection = [theta_mle_collection theta_mle_avg];
    train_mse_list(1,i) = avg_train_mse;
    crossvalid_mse_list(1,i) = avg_valid_mse;
end

% From plot, select sweet-spot as lambda with lowest validation risk:
index = find(crossvalid_mse_list == min(crossvalid_mse_list));
theta_final = theta_mle_collection(:,index);
index_train = find(train_mse_list == min(train_mse_list));
plot(lambda,train_mse_list);
hold on;
plot(lambda,crossvalid_mse_list);
hold on;
plot(lambda(1,index),min(crossvalid_mse_list),'o','MarkerFaceColor','red');
xlabel('lambda');
ylabel('Empirical Risk');
legend({'training risk', 'validation risk'},'Location','northwest');
title('Empirical Risk vs Regularization Penalty');

% Making Predictions on Test dataset:
test_mse = (1/size(test_features,1))*(sum((test_labels - (test_features*theta_final)).^2));
normal_test_mse = (1/var(test_labels))*test_mse;

% Printing optimal lambda value:
fprintf('Optimal regularization penalty : %d \n', lambda(1,index));
```

```
% Printing testing risk:
fprintf('Normalized Test risk = %d \n',normal_test_mse);
```

D.2. Regression using KNN(Python):

```
# In[ ]:
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import copy

# In[ ]:
# Loading the csv file:
df = pd.read_csv('csv_file_path'')
# Normalize the dataset
df = (df-df.mean())/df.std()

# In[ ]:
# Splitting data into train, validation and test:
split_train = round(0.7*(df.shape[0]))
df_train = df.iloc[:split_train,:]
split_valid = round(0.15*(df.shape[0]))
df_valid = df.iloc[split_train:(split_train+split_valid),:]
df_test = df.iloc[(split_train+split_valid):,:]

# In[ ]:
# Splitting training set into features and labels:
train_features = df_train[['co','no','nox','uvpm']]
#train_features = df_train[['bp','co','no','nox','out_rh','out_t','pm25t','uvpm']]
train_labels = df_train[['no2']]

# In[ ]:
# Splitting validation set into features and labels:
valid_features = df_valid[['co','no','nox','uvpm']]
#valid_features = df_valid[['bp','co','no','nox','out_rh','out_t','pm25t','uvpm']]
valid_labels = df_valid[['no2']]

# In[ ]:
# Splitting test set into features and labels:
test_features = df_test[['co','no','nox','uvpm']]
#test_features = df_test[['bp','co','no','nox','out_rh','out_t','pm25t','uvpm']]
```

```
test_labels = df_test[['no2']]

# In[ ]:
# Function
# Calculate distances of a point from each training point and store them in list of lists:
# Distance Metric used is Euclidean Distance, can be changed to Manhattan by uncommenting
def cal_dist(valid_features,train_features):
    list_valid_dist = []
    for i in range(valid_features.shape[0]):
        list_dist_point = []
        arr1 = (valid_features.iloc[i,:]).to_numpy()
        for j in range(df_train.shape[0]):
            arr2 = (train_features.iloc[j,:]).to_numpy()
            #dis = np.linalg.norm(arr1-arr2)    # For Euclidean distance
            dis = np.sum(np.abs(arr1-arr2))     # For Manhattan distance
            list_dist_point.append(dis)
        list_valid_dist.append(list_dist_point)
    return list_valid_dist

# In[ ]:
# Function
# Find K nearest neighbours and storing their indices:
def cal_knn(list_valid_dist,k):
    list_k = copy.deepcopy(list_valid_dist)
    k_n_list = []
    for dist_list in list_k:
        list_np = []
        for j in range(k):
            min_index = dist_list.index(min(dist_list))
            list_np.append(min_index)
            dist_list.remove(dist_list[min_index])
        k_n_list.append(list_np)
    return k_n_list

# In[ ]:
# Function
# Predicting on given dataset set:
# Prediction is made by calculating average of labels of k nearest training points:
def predict(valid_labels,train_labels,k_n_list,k):
    valid_preds = np.zeros((valid_labels.shape[0],1))
    train_np_labels = train_labels.to_numpy()
    for i in range(valid_preds.shape[0]):
```

```
        sum = 0
        for ind in k_n_list[i]:
            sum = sum + train_np_labels[ind,0]
        valid_preds[i,0] = sum/k
    return valid_preds

# In[ ]:
# Function
# Calculating MSE:
def cal_mse(valid_preds, valid_labels):
    valid_labels_np = valid_labels.to_numpy()
    sqr_sum = np.sum(np.square((valid_labels_np - valid_preds)))
    valid_mse = sqr_sum/(valid_labels_np.shape[0])
    return valid_mse

# In[ ]:
# Calculating distances for both train and validation set:
train_dist = cal_dist(train_features,train_features)
valid_dist = cal_dist(valid_features,train_features)

# In[ ]:
# Running knn regression on training and validation data and saving the MSE values in a list:
train_mse_list = []
valid_mse_list = []
for k in range(1,101):
    # Calculating Training MSE:
    train_knn = cal_knn(train_dist,k)
    preds_train = predict(train_labels,train_labels,train_knn,k)
    mse_train = cal_mse(preds_train, train_labels)
    train_mse_list.append(mse_train)
    # Calculating Validation MSE:
    valid_knn = cal_knn(valid_dist,k)
    preds_valid = predict(valid_labels,train_labels,valid_knn,k)
    mse_valid = cal_mse(preds_valid, valid_labels)
    valid_mse_list.append(mse_valid)

# In[ ]:
# Plotting training MSE and validation MSE against k:
k_range = list(range(1,101))
plt.plot(k_range,train_mse_list,label= 'Training MSE')
plt.plot(k_range,valid_mse_list,label = 'Validation MSE')
plt.xlabel('k')
```

```
plt.ylabel('Mean Squared Error')
plt.legend()
plt.show()

# In[ ]:
k_final = 1

# In[ ]:
# Testing on test data:
test_dist = cal_dist(test_features,train_features)
test_knn = cal_knn(test_dist,k_final)
preds_test = predict(test_labels,train_labels,test_knn,k_final)
mse_test = cal_mse(preds_test, test_labels)
print(mse_test)
print(mse_test/test_labels.var())
```

D.3. Regression using Feedforward Neural Network (Python):

```
# In[ ]:
import torch
import csv
import os,sys
from torch import nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, TensorDataset
import numpy as np
import time
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')

# In[ ]:
bp = 4
no = 5
no2 = 6
nox = 7
out_rh = 8
out_t = 9
pm25t = 10
uvpm = 14
with open('airquality17-20.csv', 'r') as csvfile:
    bp_col = []
```

```
    no_col = []
    no2_col = []
    nox_col = []
    out_rh_col = []
    out_t_col = []
    pm25t_col = []
    uvpm_col = []
    for line in csvfile.readlines():
        array = line.split(',')
        array = ['0' if len(x)<1 else x for x in array]
        bp_col.append(array[bp])
        no_col.append(array[no])
        no2_col.append(array[no2])
        nox_col.append(array[nox])
        out_rh_col.append(array[out_rh])
        out_t_col.append(array[out_t])
        pm25t_col.append(array[pm25t])
        uvpm_col.append(array[uvpm].rstrip("\n"))

del bp_col[0]
del no_col[0]
del no2_col[0]
del nox_col[0]
del out_rh_col[0]
del out_t_col[0]
del pm25t_col[0]
del uvpm_col[0]
uvpm_col[366] = '0'
uvpm_col[367] = '0'
bp_col = list(map(float,bp_col))
no_col = list(map(float,no_col))
no2_col = list(map(float, no2_col))
nox_col = list(map(float, nox_col))
out_rh_col = list(map(float, out_rh_col))
out_t_col = list(map(float, out_t_col))
pm25t_col = list(map(float, pm25t_col))
uvpm_col = list(map(float, uvpm_col))

# In[ ]:
data = [[bp, no, nox, out_rh, out_t, pm25t, uvpm] for bp, no,nox,out_rh, out_t,pm25t, uvpm in
zip(bp_col, no_col, nox_col, out_rh_col, out_t_col, pm25t_col, uvpm_col)]
```

```
# In[ ]:
normalized = MinMaxScaler(feature_range=(-10,10))
data = normalized.fit_transform(data)
no2_col = np.asarray(no2_col)
no2_col = normalized.fit_transform(no2_col.reshape(-1,1))

# In[ ]:
data_test_len = int(len(data)*0.10)
data_train_len = len(data) - data_test_len
data_train = data[:data_train_len]
data_test = data[data_train_len:]
target_train = no2_col[:data_train_len-data_test_len]
target_val = no2_col[data_train_len-data_test_len:data_train_len]
target_test = no2_col[data_train_len:]
data_train = data[:data_train_len-data_test_len]
data_val = data[data_train_len - data_test_len:data_train_len]

# In[ ]:
class Dataset(Dataset):
    def __init__(self, feature, target):
        self.feature = torch.FloatTensor(feature)
        self.target = torch.FloatTensor(target)
    def __len__(self):
        return len(self.feature)

    def __getitem__(self, index):
        x = self.feature[index]
        y = self.target[index]
        return x,y

# In[ ]:
train_dataset = Dataset(data_train, target_train)
val_dataset = Dataset(data_val, target_val)
test_dataset = Dataset(data_test, target_test)
train_dataloader = DataLoader(dataset=train_dataset, batch_size = 32, shuffle = True)
val_dataloader = DataLoader(dataset=val_dataset, batch_size = 32, shuffle = True)
test_dataloader = DataLoader(dataset=test_dataset, batch_size = 32, shuffle = False)

# In[ ]:
class simple_model(nn.Module):
    def __init__(self,in_feat,out_feat):
        super(simple_model, self).__init__()
```

```
        self.first_layer = nn.Linear(in_feat,300)
        self.activation1 = nn.ReLU()
        self.second_layer =nn.Linear(300,300)
        #self.dropout = nn.Dropout(0.2)
        self.activation2 = nn.ReLU()
        self.third_layer = nn.Linear(300,300)
        self.activation3 = nn.ReLU()
        self.fourth_layer = nn.Linear(300,1)

    def forward(self,x):
        out = self.first_layer(x)
        out = self.activation1(out)
        out = self.second_layer(out)
        #out = self.dropout(out)
        out = self.activation2(out)
        out = self.third_layer(out)
        out = self.activation3(out)
        out = self.fourth_layer(out)
        return out

# In[ ]:
model = simple_model(7,1)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
criterion = nn.MSELoss().to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=1e-6)
model.to(device)

# In[ ]:
def train_model(model, train_loader, criterion, optimizer):
    model.train()
    train_loss = []
    start_time = time.time()
    for index, (feature,target) in enumerate(train_loader):
        optimizer.zero_grad()
        feature = feature.to(device)
        target = target.view(-1,1)
        target = target.to(device)
        out = model(feature)
        loss = criterion(out,target)
        var_target = target.var(dim=0)
        var_target = var_target.to("cpu")
        var_target = var_target.numpy()
```

```
            loss_item = loss.item()/var_target
            train_loss.append(loss_item)
            loss.backward()
            optimizer.step()
        end_time = time.time()
        avg_loss = np.mean(train_loss)
        print('Training Loss:', avg_loss, 'Time: ',end_time-start_time)
        return avg_loss

    # In[ ]:
    def test_model(model, test_loader, criterion, optimizer):
        with torch.no_grad():
            model.eval()
            val_loss = []
            total_predictions = 0
            total_correct = 0
            for index, (feature,target) in enumerate(test_loader):
                feature = feature.to(device)
                target = target.view(-1,1)
                target = target.to(device)
                out = model(feature)
                total_predictions += target.size(0)
                #out = (out * (10**2)).round()/(10**2)
                #target = (target * (10**2)).round()/(10**2)
                total_correct += (out.round()== target.round()).sum().item()
                loss = criterion(out, target).detach()
                var_target = target.var(dim=0)
                var_target = var_target.to("cpu")
                var_target = var_target.numpy()
                loss_item = loss.item()/var_target
                val_loss.append(loss_item)
            avg_loss = np.mean(val_loss)
            acc = (total_correct/total_predictions)*100
            print('Testing(Val) Loss: ', avg_loss)
            print('Testing(Val) Accuracy: ', acc, '%')
            return avg_loss, acc

    # In[ ]:
    num_epoch = 200
    train_loss_list = []
    val_loss_list = []
    val_acc_list = []
```

```
for i in range(num_epoch):
    print("Starting Epoch: ", i)
    train_loss = train_model(model, train_dataloader, criterion, optimizer)
    val_loss, val_acc = test_model(model, val_dataloader, criterion, optimizer)
    train_loss_list.append(train_loss)
    val_loss_list.append(val_loss)
    val_acc_list.append(val_acc)
    print('='*25)

# In[ ]:
plt.title("Validation Loss Plot")
plt.plot(val_loss_list, label="Validation Loss")
plt.legend()
plt.show()
plt.title("Validation Accuracy Plot")
plt.plot(val_acc_list, label="Valicdation accuracy")
plt.legend()
plt.show()
plt.title("Train Loss Plot")
plt.plot(train_loss_list, label = "Train Loss")
plt.legend()
plt.show()

# In[ ]:
def test_model_out(model, test_loader, criterion, optimizer):
    with torch.no_grad():
        model.eval()
        out_list = []
        target_list = []
        loss_list = []
        for index, (feature,target) in enumerate(test_loader):
            feature = feature.to(device)
            target = target.to(device)
            out = model(feature)
            loss = criterion(out, target).detach()
            out = out.view(-1)
            out = out.to("cpu")
            out = out.numpy()
            out_list.extend(out)
            var_target = target.var(dim=0)
            var_target = var_target.to("cpu")
            var_target = var_target.numpy()
```

```
            loss_item = loss.item()/var_target
            loss_list.append(loss_item)
            target = target.to("cpu")
            target = target.view(-1)
            target = target.numpy()
            target_list.extend(target)
        return out_list, target_list, np.mean(loss_list)

# In[ ]:
out_list, target_list, test_loss = test_model_out(model,test_dataloader, criterion, optimizer)
print(test_loss)

# In[ ]:
plt.title("Test Predictions Vs. Test Targets")
plt.plot(out_list,target_list)
plt.autoscale(axis='x')
plt.show()

# In[ ]:
x = np.arange(0,len(out_list),1)
plt.title("Scatter Plot of Test Targets and Predictions")
plt.scatter(x,out_list, label ="Test Predictions")
plt.scatter(x,target_list, label ="Test Targets" )
plt.grid(True)
plt.legend()
plt.show()

# In[ ]:
print(len(out_list))

# In[ ]:
x = np.arange(0,len(out_list),1)

# In[ ]:
print(len(x))
```