18.065 Final Project

Comparing Regression Methods for Predicting COVID-19 Response

Aparna Gupte, Charvi Gopal, Stacia Johanna

1 Introduction

Supervised statistical learning problems, which involve building a model to predict an output from one or more inputs, occur in a variety of fields. The COVID-19 pandemic provides ongoing opportunities to understand relationships between the underlying variables using supervised statistical learning methods on publicly available datasets.

We compare the performance on COVID-19 data of two kinds of regression methods, Linear Regression and Neural Network, both with and without Principal Component Analysis (PCA). Linear models offer certain inference-related advantages in terms of real-world problems and are often surprisingly competitive against non-linear methods (James, 2017), so we evaluate both the Linear Regression method and non-linear Neural Network method. In addition, we examine the effects of shrinkage using Lasso Regression and dimension reduction using Principal Component Analysis (PCA).

2 Background

2.1 Principal Component Analysis

Principal Component Analysis (PCA) is a method used for dimensionality reduction through feature extraction. Due to the limited size of the training data (there are only about 200 countries), we examine the effect of reducing the number of features in preventing overfitting. This method performs the spectral decomposition PDP^{-1} of the positive semidefinite symmetric covariance matrix X^TX of the training feature matrix X to obtain a new orthogonal basis. Assume that the eigenvalues are in descending order is D. Then, the data matrix is transformed to get a new feature matrix $X^* = XP$ by projecting onto the principal components. We arbitrarily pick the number of principal components we want to consider and perform training on that subset of the new features. The disadvantage of using PCA is that it reduces the interpretability of the features.

2.2 Linear Regression

Linear regression is a supervised learning technique for predicting a dependent variable on the basis of one or more independent variables. Simple linear regression involves one predictor variable and multiple linear regression involves more than one. In this paper, we use multiple linear regression, which gives each predictor X_i a separate slope/coefficient β_i:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \varepsilon,$$

β_i here is the average effect on the dependent variable Y of a one unit increase in X_i, if the other predictors are held fixed. β_0 is the intercept term, the expected value of Y when $X = 0$. ε is a mean-zero random error term.

Our goal is to obtain coefficients β_i such that the resulting line is as close as possible to the data points. To measure closeness to the data points, we will use least squares. $e_i = y_i - \hat{y}_i$ represents the ith residual, which is the difference between the predicted and observed response values. The least squares method chooses coefficients β_i that minimize the residual sum of squares (RSS), which is defined as:

$$\text{RSS} = e_1^2 + e_2^2 + \ldots + e_n^2$$

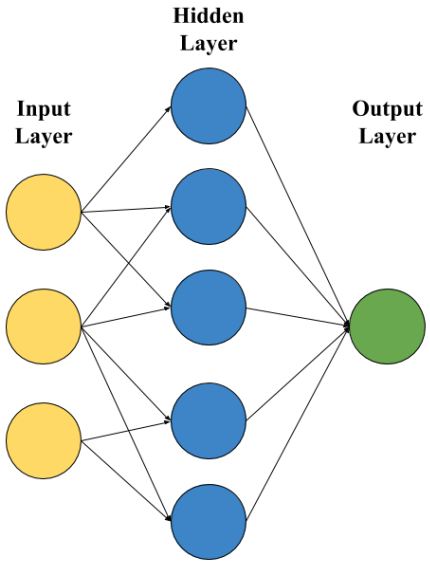
2.3 Lasso Regression

Lasso (Least Absolute Shrinkage and Selected Operator) Regression is a type of linear regression with L1 regularization. In addition to trying to minimize the squared error sum, it also tries to minimize the sum of the absolute values of the parameters of the model. The bias/ intercept term is usually excluded from this sum. It encourages the algorithm to learn sparser models, and is used to prevent overfitting the training data. Lasso regression tries to minimize the following loss function:

$$\sum_{i=1}^{n}(y_i - \sum_j x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

2.4 Neural Networks

Neural networks are a set of machine learning algorithms that are designed to perform some tasks by analyzing training examples. Some examples of the tasks where neural networks work well are speech recognition, image classification, etc. In the context of this paper, neural networks are used to predict a country's COVID-19 deaths per million based on the various factors of said country, such as Gross Domestic Product (GDP), land area, average temperature, etc.



In neural networks, the input nodes are connected to multiple nodes in a hidden layer. These nodes are called neurons. When a value goes past a node, it pasts through an activation function that exists on every node. The activation function converts the summed weighted input values into another value that will be produced as an output from that node. These outputs are then used as the input for the next layer, and so on.

To use the neural networks, training input data is fed into the bottom input layer. These datas are then fed-forwarded through the hidden layer, and to the output layer. Since there are many adjustable factors of neural networks, such as diverse activation layers, number of nodes in hidden layers, number of hidden layers, etc., the accuracy of these models are cross validated with the validation data. Then finally, the model which produced the best consistent result is used to predict the test data, and in this paper, it is compared with linear regression and PCA.

Deep learning is neural networks that have multiple hidden layers. While some deep learning models are used to train the data, we found that neural networks that use 1 hidden layer produced the best result for the COVID-19 data.

3 Data

The metric we seek to predict is the number of deaths per million between January 22 and April 21, 2020, as a measure of how well different countries respond to the COVID-19 pandemic. The features that we use as the input are:

- Gross Domestic Product (GDP) in billion US Dollar
- Population density in people per Km^2
- Median age
- Land area in Km^2
- Global health security index
- Humidity in %, calculated by averaging daily humidity from January 22, 2020 to March 21, 2020
- Temperature in Celsius, calculated by averaging daily humidity from January 22, 2020 to March 21, 2020
- Number of SARS-1 deaths

Since COVID-19 is an ongoing pandemic, there is no single dataset that includes all the features we need. This motivated us to compile data from multiple sources, which are detailed in the References. We were able to find data about countries' demographics, economic and healthcare

systems, climate, and past epidemic responses. However, we were not able to find adequate data on some other kinds of features, including the extent and efficiency of testing strategies, immigration and international travel restrictions, and the extent of lockdown and restriction of movement within a country.

4 Method

4.1 Data Cleaning

4.1.1 Filling in Missing Values

We ensure that our data does not have any missing values by filling gaps with data from other publicly available sources with similar values. In some cases, we can infer values. For example, for the feature about the number of SARS deaths, death data is available for only 37 countries, so we assume 0 deaths for other countries. For countries whose values we could not infer or find, we insert the mean of the feature's values as the missing value. Note that this does not change the mean of the feature's values.

4.1.2 Merging Values in Dataset

In the dataset, there are several types of data that we combine. For instance, we merge the total number of deaths in provinces or regions of the same countries into one. In the original data source, the number of deaths in countries such as China and Canada is split into the number of deaths in each province/state. We combine the number of deaths of all these provinces into just one country's total death value. This is done because there is no detailed data on the other features of these provinces. We perform a similar combination of rows for other features such as temperature and humidity.

4.1.3 Removing Rows and Columns

We started by having the number of rows in our dataset equal the number of rows of the John Hopkins COVID-19 deaths dataset. However, we found that some places do not have any data on the other features that we want to base our models upon. That is why we remove them from our dataset. These locations are cruise ships (Diamond Princess and MS Zaandam) and a Church jurisdiction (Holy See).

For the columns (features), we only remove literacy rate, as the data is too sparse. In addition, there is no literacy rate data for many developed countries, such as the United States, Canada, South Korea, etc., so we feel that this feature would not be representative of actual literacy values.

4.2 Data Pre-processing

We first randomize and split the data into training (60%), cross validation (20%) and testing (20%). We use the Python sklearn.preprocessing library to standardize the data so that the mean value of the new standardized features is 0 and so that the features are scaled down to roughly comparable values depending on their spread and distribution. We standardize only the features and not the targets, and use only the training data to compute the mean μ and standard deviation σ.

$$x_j^i = \frac{x_j^i - \mu}{\sigma}$$

4.3 Modeling

4.3.1 Principal Component Analysis

We use the Python sklearn.decomposition library, and fit the PCA model on the training data to obtain the principal components. We then project the training and validation feature matrices onto these principal components to obtain new feature matrices with 2, 4 and 6 principal components. These new features matrices are used in downstream regression tasks (linear regression, lasso regression and neural networks).

4.3.2 Linear Regression

We use `Pandas` for inputting the dataset CSV as a data frame and `Statsmodels` for performing linear regression on the data frame. We use the Ordinary Least Squares (OLS) function to minimize the difference between the predicted and observed values. We train the OLS model on four sets of training data: without PCA (8 features), PCA with 2 features, PCA with 4 features, and PCA with 6 features. We validate each of these models on the corresponding 4 sets of validation data and calculate the Root Mean Squared Error (RMSE), explained in Section 4.4. The code is available in Appendix 1.

4.3.3 Lasso Regression

We use the Python `sklearn.linear_model` library, a supervised learning library for linear models. We fit the model on the training data and use the trained model to predict the outputs on both the training and cross validation data to evaluate the model. We train on the feature matrices before performing PCA, as well as with 2, 4, 6 principal components after performing PCA. We perform training with a range of learning rates and choose the learning rate that gives us the least error.

4.3.4 Neural Network

For modelling the neural network, we use `Tensorflow 2` and `Keras` library. There are many parameters that can be tuned for neural network models. In this paper, here are the parameters that we adjusted: number of hidden layers, number of nodes, activation function, optimizer, training epochs, and batch size.

In Keras, there are many activation layers that are provided by the library. As the problem in this paper is a regression problem, the activation layers that Keras provides are Rectified Linear Unit (ReLU), sigmoid, and tanh.

For the optimizer, Keras provides SGD (stochastic gradient descent) and Adam that we can use for neural network models.

- Stochastic Gradient Descent (SGD)

  Gradient descent is an optimization algorithm which minimizes a function by iteratively moving in the direction of the negative gradient. SGD is a more efficient version of gradient descent, in which it only uses a subset of the example data. The function that defines the next step x_{k+1} after knowing x_k is $x_{k+1} = x_k - s_k \nabla_x l(x_k, v_i)$, where s_k is the step size and $\nabla_x l(x_k, v_i)$ is derivative of the loss term from sample v_i.

- Adam

  Adam is an adaptive learning rate optimizer for neural networks. It converges faster than SGD as it uses the previous gradients as a part of its calculation. The step direction at step k is defined as $D_k = \delta D_{k-1} + (1-\delta)\nabla L(x_k)$ and the step size at step k is defined as ${s_k}^2 = \beta {s_{k-1}}^2 + (1-\beta)\|\nabla L(x_k)\|^2$. Because it uses the knowledge of previous step direction and previous step size to calculate the next step, Adam generally has better performance than SGD.

4.4 Evaluation

To evaluate our model performance, we use root mean squared error (RMSE). RMSE measures the average magnitude of the error. It's the square root of the average of squared

differences between the actual value and the prediction value. Formally, RMSE is defined as $RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$.

5 Result and Analysis

5.1 Linear Regression

The following table shows RMSE values for linear regression without PCA (all 8 features) and with 6, 4, and 2 PCA components. For reference, the range of COVID deaths per million is 0 to 1178, so the best RMSE achieved, 56.8, is 4.8% of this range.

Method	Training RMSE	Validation RMSE
No PCA	145.5	117.6
PCA-6	146.3	99.2
PCA-4	146.6	67.3
PCA-2	147.0	56.8

These relatively high RMSEs, relative to the neural network results in 5.3, can perhaps be explained using correlation scores of each of the 8 features against COVID deaths per million. The highest correlation score, 0.139 for Global Health Security Index, is still quite low.

5.2 Lasso Regression

Lasso regression does not significantly reduce the RMSE, and we observe an increase in the validation RMSE with PCA, as compared to linear regression. This is not surprising, because the high RMSE for linear regression indicates that linear models will underfit the data, so trying to prevent overfitting will not improve the model.

Method	Training RMSE	Validation RMSE
No PCA	139.4	104.2
PCA - 6	140.2	93.7
PCA - 4	140.4	73.9
PCA - 2	140.9	74.8

5.3 Neural Network

Other than linear regression, we also use neural network and deep neural network models to predict COVID-19 death per million. As there are many parameters that can be adjusted, we try out many neural network models for the data. In this part of the section, the results on the tables are the RMSE of the validation data.

For the first iteration of trying out the model, we try different numbers of batch size, while keeping the others parameter constant. For this first iteration, we use 1 hidden layer with 10 nodes, ReLU as the activation function, Adam as the optimizer, and epochs of 100.

Batch Size	RMSE
10	43.57443452
20	43.16827883
30	**42.89406037**
40	43.17309088

From the result, a batch size of 30 has the best result, so we use that size for the other models.

After determining the best batch size for the model, we try out different activation functions. The functions that we try are ReLU, sigmoid, and tanh. This time, all the parameters are the same as above other than the number of nodes in 1 hidden layer is either 10 or 50 nodes.

Activation Function	10 nodes RMSE	50 nodes RMSE
ReLU	42.99406037	48.84912505
Sigmoid	**42.1649227**	**41.407048**
Tanh	42.64236212	42.03640911

From the result, sigmoid produces the best result. However, since ReLU usually also does well in neural networks, we still compare ReLU and sigmoid in the next models.

For the next iteration, we try changing the number of nodes in 1 hidden layer. We also compare both ReLU and Sigmoid at each of the new models. Below is a part of the result for trying out different numbers of nodes.

Number of Nodes	ReLU RMSE	**Sigmoid RMSE**
20	42.77291074	42.1060317
30	44.33159849	41.81477316
65	48.84912505	**41.30074224**
70	50.93284923	41.42433481

From the above table, we find that sigmoid with 65 nodes in 1 hidden layer gives the best RMSE of 41.3. Also, from this iteration, we are now sure that sigmoid works better than ReLU as it consistently gives lower RMSE for all numbers of nodes.

Then, we try out different numbers of hidden layers while also changing the number of nodes in the layers just in case that deeper neural networks with fewer nodes have a different result than shallower layers with more nodes. Similar to the previous iteration, we keep the number of epochs to 100, batch size to 30, optimization to Adam, activation function to sigmoid. The next table contains models which have the best result in their respective number of layers.

Model	RMSE
1 hidden layer, 65 nodes	**41.30074224**
2 hidden layers, 50 nodes	41.45275397
5 hidden layers, 40 nodes	41.43462044
10 hidden layers, 40 nodes	41.43386672
20 hidden layers, 40 nodes	41.40802013

All the results are pretty similar, but the model with 1 hidden layer has a slightly better RMSE. This might be because deeper layers with large numbers of nodes makes it overfit the training data, thus becoming worse in predicting the validation data.

For the optimizer, we try out different models with Adam and SGD. In almost all models, the models with SGD work consistently worse compared to those of Adam, in which the models with SGD have RMSE of over 60. This might be because Adam uses adaptive learning and thus has better convergence compared to models with SGD.

Therefore, among all the iterations, this model has the best result overall: 1 hidden layer with 65 nodes and activation function of sigmoid, optimizer of Adam, epochs of 100, and batch size of 30.

5.3.1 Neural Network with Principal Component Analysis

Similar to linear regression, we also trained the neural network models with datas produced from PCA. Since we know the neural network model with the best result from the previous section, now we will have that model to predict the test data and compare the result to that of the model trained on PCA data.

Model	RMSE Test Data
NN with no PCA	57.91357669
NN with 6-columns PCA	57.74835258
NN with 4-columns PCA	57.63444174
NN with 2-columns PCA	**57.47104704**

The results in this section are also quite similar; however, from the table, it shows that using PCA is consistently better than not using PCA. Also, PCA with fewer numbers of columns are also doing better than those of more columns. This result is expected because PCA chooses datas that contributes more to the death per million output and reduces down the number of unrelated and unnecessary data. PCA also helps the neural networks to converge better. Thus, neural networks that use 2 columns PCA have the smallest RMSE among all tested neural network models.

5.4 Comparison between Linear Regression and Neural Network

Models	RMSE on validation data
Linear regression	117.6
Lasso regression	104.2
Neural network	**41.30074224**

As we see from the above table comparing the RMSE result of validation data from linear regression, lasso regression, and neural network, it is conspicious that neural networks produce the best result by a huge margin. This can be a strong indication that the problem we are trying to solve is not linear, and that is why linear regression methods do not work too well on this problem.

6 Limitations

6.1 Inaccurate COVID-19 data

The metric we train on and predict is the number of COVID-19 deaths per million. In the US and abroad, there are several reasons to doubt the reported number of deaths. These include delays in reporting deaths due to operational issues, the lack of reporting of deaths outside hospitals, and inconsistent policies for attributing deaths to COVID-19 (O'Neil, 2020).

6.2 Missing Relevant Features

There are several datas that we think are relevant to the death per million people of a country but we could not obtain numerical data for. These include the extent of a lockdown within a country, the flight controls and immigration restrictions of each country, and testing availability. It is difficult to quantify the extent of a lockdown within a country without clear metrics. Moreover, very little of this data is available for the public, so we could not include these relevant features into our calculation.

7 Conclusion

The purpose of this project was to apply a variety of statistical learning techniques and use 8 features to predict how well or poorly a country responds to COVID-19. We found the nonlinear neural net method more effective than linear and lasso regression. Further projects we suggest include rerunning the models after more covid data is available, adding other kinds of features discussed earlier, and comparing data from multiple sources (including, perhaps, crowdsourced data from citizen science efforts) to verify the accuracy of government datasets.

References

Method

Bronshtein, A. (2017, September 18). A Quick Introduction to the NumPy Library. Retrieved May 10, 2020, from https://towardsdatascience.com/a-quick-introduction-to-the-numpy-library-6f61b7dee4db

Bronshtein, A. (2019, October 29). A Quick Introduction to the "Pandas" Python Library. Retrieved May 10, 2020, from https://towardsdatascience.com/a-quick-introduction-to-the-pandas-python-library-f1b678f34673

Bronshtein, A. (2019, December 27). Simple and Multiple Linear Regression in Python. Retrieved May 10, 2020, from https://towardsdatascience.com/simple-and-multiple-linear-regression-in-python-c928425168f9

Brownlee, J. (2020, April 23). How to Choose Loss Functions When Training Deep Learning Neural Networks. Retrieved May 10, 2020, from https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Brownlee, J. (2020, February 9). TensorFlow 2 Tutorial: Get Started in Deep Learning With tf.keras. Retrieved May 10, 2020, from https://machinelearningmastery.com/tensorflow-tutorial-deep-learning-with-tf-keras/

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2017). *An introduction to statistical learning: with applications in R*. Retrieved from https://faculty.marshall.usc.edu/gareth-james/ISL/ISLR%20Seventh%20Printing.pdf

O'Neil, C. (2020, April 13). 10 Reasons to Doubt the Covid-19 Data. Retrieved May 10, 2020, from https://www.bloomberg.com/opinion/articles/2020-04-13/ten-reasons-to-doubt-the-covid-19-data

Strang, G. (2019). *Linear algebra and learning from data*. Wellesley, MA: Wellesley-Cambridge Press

Galarnyk, M. (2017). *PCA Using Python (scikit-learn)*. Retrieved May 11, 2020, from https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60

Pedregosa et al. (2011). *Scikit-learn: Machine Learning in Python*. JMLR 12, pp. 2825-2830, 2011

Data

Novel Coronavirus (COVID-19) Cases Data - time_series_covid19_deaths_global.csv - Humanitarian Data Exchange. (n.d.). Retrieved April 22, 2020, from

https://data.humdata.org/dataset/novel-coronavirus-2019-ncov-cases/resource/61a3a172-9aa3-4d00-b9f4-1c94ed1c76fc
kp, D. (2020, February 26). SARS 2003 Outbreak Dataset. Retrieved April 22, 2020, from https://www.kaggle.com/imdevskp/sars-outbreak-2003-complete-dataset
Countries in the world by population (2020). (n.d.). Retrieved April 22, 2020, from https://www.worldometers.info/world-population/population-by-country/
2020 World Population by Country. (n.d.). Retrieved April 22, 2020, from https://worldpopulationreview.com/
real time world statistics. (n.d.). Retrieved April 22, 2020, from https://www.worldometers.info/
GDP. (n.d.). Retrieved April 22, 2020, from https://tradingeconomics.com/country-list/gdp
Winter, P. (2020, March 24). COVID19_Global_Weather_Data. Retrieved April 22, 2020, from https://www.kaggle.com/winterpierre91/covid19-global-weather-data

Appendix: Code

Principal Component Analysis

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

# Read the CSV file
all_data = pd.read_csv('all_data.csv')
x = all_data.filter(items = ['humidity', 'temp', 'ghsi', 'medianage', 'sars', 'landarea', 'gdp', 'popdens'])
x['popdens'] = pd.to_numeric(x['popdens'], errors='coerce')
y = all_data.filter(items = ['covidpercapita'])
x = x.values
y = y.values

train_val_x, test_x, train_val_y, test_y = train_test_split(x, y, test_size=0.2, random_state=0)
train_x, val_x, train_y, val_y = train_test_split(train_val_x, train_val_y, test_size=0.25, random_state=0)

# We do not standardize the targets
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(train_x)
train_x = scaler.transform(train_x)
val_x = scaler.transform(val_x)
test_x = scaler.transform(val_x)
```

```
# Standardized training data
np.savetxt(f'data//std_train_x.csv', train_x, delimiter = ',')
# Training targets
np.savetxt(f'data//train_y.csv', train_y, delimiter = ',')
# Normalized val data
np.savetxt(f'data//std_val_x.csv', val_x, delimiter = ',')
# Val targets
np.savetxt(f'data//val_y.csv', val_y, delimiter = ',')
# Normalized test data
np.savetxt(f'data//std_test_x.csv', test_x, delimiter = ',')
# Test targets
np.savetxt(f'data//test_y.csv', test_y, delimiter = ',')

from sklearn.decomposition import PCA# Make an instance of the Model
for n in [2, 4, 6]:
        pca = PCA(n)
        pca.fit(train_x)
        pca_train_x = pca.transform(train_x)
        pca_val_x = pca.transform(val_x)
        pca_test_x = pca.transform(test_x)
        explained_variance = pca.explained_variance_ratio_


        filename = f'data//pc{n}_ev{explained_variance}_train_x.csv'
        np.savetxt(filename, pca_train_x, delimiter = ',')

        filename = f'data//pc{n}_ev{explained_variance}_val_x.csv'
        np.savetxt(filename, pca_val_x, delimiter = ',')

        filename = f'data//pc{n}_ev{explained_variance}_test_x.csv'
        np.savetxt(filename, pca_test_x, delimiter = ',')
```

Linear Regression

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
from statsmodels.tools.eval_measures import rmse

# get training data
X_training = pd.read_csv("data/std_train_x.csv",header=None)
```

```
y_training = pd.read_csv("data/train_y.csv",header=None)

# train
model = sm.OLS(y_training, X_training).fit()

# get validation data
X_validation = pd.read_csv("data/std_val_x.csv",header=None)
y_validation = pd.read_csv("data/val_y.csv",header=None)

# validate
predictions = model.predict(X_validation)
rmse_pred = rmse(y_validation[0].values.tolist(), predictions.values.tolist())
print("the RMSE is:", rmse_pred)
```

Lasso Regression

```
from sklearn.linear_model import Lasso
from sklearn.metrics import r2_score, mean_squared_error
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

train_y = pd.read_csv("data//train_y.csv")
val_y = pd.read_csv("data//val_y.csv")
alpha = np.linspace(0.01,0.4,10)

for id in ['std', 'pc2', 'pc4', 'pc6']:
        train_x = pd.read_csv(f'data//{id}_train_x.csv')
        val_x = pd.read_csv(f'data//{id}_val_x.csv')
        r2_train =[]
        r2_val =[]
        norm = []
        rmse_train = []
        rmse_val = []
        for i in range(10):
                lasso = Lasso(alpha = alpha[i])
                lasso.fit(train_x, train_y)
                train_predict = lasso.predict(train_x)
                val_predict = lasso.predict(val_x)

                r2_train = np.append(r2_train, r2_score(train_predict,train_y))
                r2_val = np.append(r2_val, r2_score(val_predict,val_y))
```

```
            norm = np.append(norm,np.linalg.norm(lasso.coef_))
            rmse_train.append(mean_squared_error(train_predict, train_y)**0.5)
            rmse_val.append(mean_squared_error(val_predict, val_y)**0.5)
```

Neural Network

```
from pandas import read_csv
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow import keras
import numpy as np
import tensorflow as tf
import math
import statistics

np.random.seed(42)

# Process training set
train_X_path = './data/std_train_x.csv'
X_train_raw = read_csv(train_X_path, header=None)
X_train = tf.convert_to_tensor(np.asarray(X_train_raw), np.float64)

train_y_path = './data/train_y.csv'
y_train_raw = np.asarray(read_csv(train_y_path, header=None))
y_train = tf.convert_to_tensor(y_train_raw, np.float64)

# Process validation set
valid_X_path = './data/std_val_x.csv'
X_valid_raw = read_csv(valid_X_path, header=None)
X_valid = tf.convert_to_tensor(np.asarray(X_valid_raw), np.float32)

valid_y_path = './data/val_y.csv'
y_valid_raw = np.asarray(read_csv(valid_y_path, header=None))
y_valid = tf.convert_to_tensor(y_valid_raw, np.float32)

# Number of features without PCA
num_features = X_train.shape[1]
num_loop = 5
total = 0.0

for i in range(num_loop):
    # Define model
```

```
    model = Sequential()
    model.add(Dense(65, activation='sigmoid', kernel_initializer='he_normal',
input_shape=(num_features,)))
    model.add(Dense(1, activation='linear'))

    # Compile model
    model.compile(optimizer="adam", loss='mse')

    # Fit model
    history = model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=100, batch_size=10,
verbose=0)

    # Evaluate model with validation data
    error = model.evaluate(X_valid, y_valid, verbose=0)
    total += math.sqrt(error)

print(total/num_loop)
```