Predicting blood pressure in diabetics with non-linear regression models

1. Introduction

Diabetes is a disease which is generally caused by a person's inability to produce enough insulin; however, it may also be due to the cells not being able to respond to insulin accordingly. The disease affects millions of people worldwide which can be very deadly if left untreated. While diabetes mainly affects the blood sugar level inside the body, it can also have an impact on the blood pressure levels of the person. As diabetic people are more susceptible to infections and thus also are more likely to contract illnesses as well. Hence, by being able to forecast blood pressure levels, it could reveal other underlying problems that a diabetic person may have that is not immediately obvious from visual inspection.

This report will look at the problem more closely. Section 2 explores the dataset to understand and get a more intuitive idea about the data. Afterwards, section 3 will discuss about data splits and models used and as well as what types of error functions were used. Section 4 looks at the results of each model and how they compare by looking at the errors which they produce, and the best model will be picked. After analysing the results, section 5 will conclude the report.

2. Problem Formulation

As stated earlier, the goal of this machine learning model is to predict the blood pressure levels of diabetic individuals. Hence, the data point for training the model will be *individuals who have been medically already tested for diabetes* – regardless of the type of diabetes. The label, which we are interested in, will be the *blood pressure values [Float]* (of the diabetic individuals) which the model will try to predict. The features used by which the model will be trying to predict the blood pressure levels will be the following: *Age [Float], Sex [Float], BMI [Float], Total serum cholesterol [Float], Total cholesterol [Float], Possibly log of serum triglycerides level [Float]* and *Blood sugar level [Float]*.

2.1 The Dataset

The dataset used to train the model is from scikit-learn's diabetes dataset [1] from their toy dataset library that consists only of diabetic patients. From scikit's website they have linked that the original dataset was acquired through NCSU. [2]

3. Methods

3.1 Feature Selection and Pre-processing

By constructing a heatmap of the correlation of the features to blood pressure, it showed that there was high correlation between some of the features, namely total serum cholesterol with low-density lipoproteins and total cholesterol with high-density lipoproteins. To save computing resources, both low-density and high-density lipoproteins were taken out of the training set. All the other features were considered to have some significance in predicting the blood pressure's outcome and thus were included in training the models.

The dataset contains a total amount of 442 data points with no null values and all of them seemed to be eligible to be used without any modification as there didn't seem to be any serious outliers in the data either. As all the values from the dataset have already been "mean centered and scaled by the standard deviation times n_samples" [1] there was no need for further pre-processing.

Training, validation, and test sets

The training and validation were split up using the "train_test_split" method from the sklearn library. After trying out different values, it seemed that the optimal train/validation/test split was 80/10/10 as it led to the least amount of overfitting. The same split is used for all models for the sake of consistency when comparing the results. The one split data was used over cross validation to save computing resources, as well as to simplify the process.

3.2 Polynomial Regression

The first machine learning method which was implemented was Polynomial Regression [3] using the PolynomialFeatures from sklearn's library. Polynomial regression was used as there are a lot of features, and thus there isn't a linear relationship between the features and the label. Hence, a non-linear model is needed which polynomial regression can provide.

The polynomial regression was tried for the degrees: 2,4,3,5,10 and would then be compared with one another which one yields the best results.

3.3 MLP Regression

The second machine learning model which will be implemented is the MLPRegressor from sklearn's library. The MLP regression is different from polynomial regression as it uses deep learning which utilizes neural networks in order to predict. The MLP neural network uses neurons and hidden layers. Inside these hidden layers, there are hidden neuron activations as well as activation functions which introduces non-linearity to the network. The activation function used in this case was ReLU [5] as it is a very commonly used. For this

case, the number of layers used will be 1,2,4,6,8,10 while the number of neurons will be 10 as this was seen as optimal through testing.

3.4 Loss function

The loss function used for both models will be the mean squared error (MSE). This loss functions seems to be optimal since when exploring the data, there didn't seem any noticeable outliers, additionally as all the values have all been centered and scaled, the effect of an outlier would be reduced significantly. Additionally, this model is straightforward to implement for the models used. MSE is given by the following equation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where n = number of data points, y_i = observed values, $\hat{y}_i$ = predicted values.

4. Results

The MSE errors of both MLP and polynomial regression for the different numbers of layers and degrees are shown below in Figure 1.

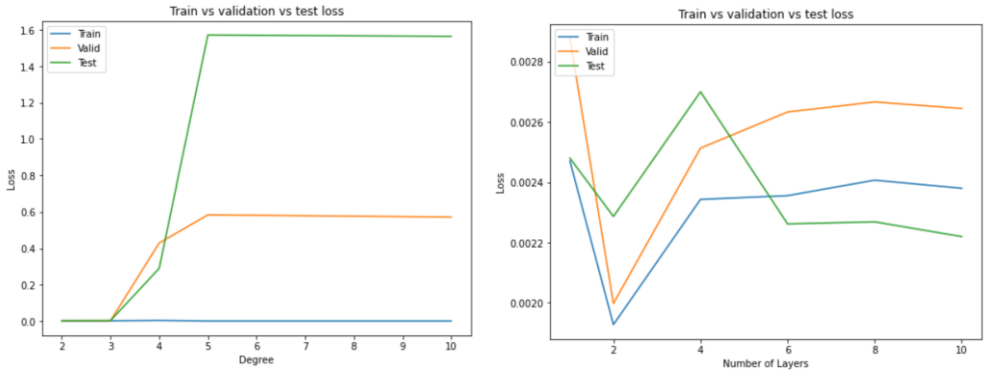


Figure 1: Left-side, MSE for polynomial regression with different degrees; Right-side, MSE for MLP Regressor with different number of layers.

As seen from Figure 1, polynomial regression starts to overfit drastically for degree > 3. Other than that, it is hard to tell the values of degrees 2 and 3 in polynomial regression.

As for MLP regression, from the graph, it shows that at 2 layers, the model fits the data the best as all training and validation error are at their lowest point while also not overfitting. Additionally, the test error is also at one of its lower points which indicates that the model is quite successful at predicting new data.

After 2 layers, the model starts to act worse as the training loss increases with the number of layers and the model starts to act unknowingly.

Split	Polynomial Regressor with degree 2 MSE	Polynomial Regressor with degree 3 MSE	MLP Regressor with 2 layers MSE
Training	0.0014106140	0.0010708729	0.0019278035
Validation	0.0017843381	0.0017144056	0.0019979312
Test	0.0016405080	0.0023833148	0.0022864456

Table 2: MSE of Polynomial Regression degree 2 & 3 and MLP Regressor with 2 layers

As degrees 2 and 3 are hard to visualize in Figure 1, at Table 2, when comparing polynomial regressor at degree 2 and 3, both training and validation error is higher in degree 2 than in degree 3. However, from the test error, at degree 2 (0.00164) the error is considerably lower than in degree 3 (0.00238). Thus, while the polynomial regressor might work better at degree 3 for the existing data, but when trying to predict on new data, degree 2 performs better. All things considered, as the models are aimed to predicting new values of blood pressure, polynomial regressor with degree 2 should be considered the better option.

When comparing polynomial regressor with degree 2 and MLP Regressor with 2 layers, Table 2 shows that the polynomial regressor has lower MSE for all training, validation, and test errors. Hence not only does polynomial of degree 2 predict better than MLP on existing data, but also on new data. Thus, the best model that should be chose is the **Polynomial Regressor with degree 2** with a test MSE of 0.00164.

5. Conclusions

The final test MSE of the chose model polynomial regression with degree 2 is 0.00164, however, in relation to the data, the error is still considerable as the standard deviation of the blood pressure is 0.047619 with most values in the range of -0.005 to 0.005 (25% to 75%). In retrospect, it would have been better to get data which hasn't been centered yet or scaled as the errors would be easier to read and comprehend.

There are still many improvements to the machine learning model which could be done in the future. One of them being that more features should be used which are more relevant to affecting the blood pressure of a person. This would for example, be the size of major arteries, family history of known heart diseases, other medical complications, and such. As there are very many factors which can affect blood pressure outside diabetes, more relevant features should have first been considered while increasing the amount of data points. Also, for an area as vast and complicated as blood pressure, the count of the data points should have also been a lot greater.

Additionally, other models should also be tried out. A more complex deep learning model is possibly needed if the feature count would increase as models such as polynomial regression could become computationally more expensive and have greater inaccuracies.

6. Bibliography:

[1] Scikit-learn, Toy datasets, https://scikit-learn.org/stable/datasets/toy_dataset.html, Date of Access 31.3.2022

[2] NCSU, Diabetes Data, https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html, Date of Access 31.3.2022

[3] Scikit-learn, PolynomialFeatures, https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html, Date of Access 31.3.2022

[4] Scikit-learn, MLPRegressor, https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html, Date of Access 31.3.2022

[5] A. F. Agarap, *"Deep Learning using Rectified Linear Units (ReLU)", arXiv:1803.08375v2* (2019)

7. Code

Blood pressure prediction using non-linear machine learning models

April 4, 2022

[1]:
```
%config Completer.use_jedi = False # enables autocomplete
from sklearn.datasets import load_diabetes
import pandas as pd
import numpy as np
import seaborn as sns
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split, KFold
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import PolynomialFeatures
from sklearn.tree import DecisionTreeRegressor
from sklearn.neural_network import MLPRegressor
```

[2]:
```
df = load_diabetes(as_frame=True)['frame'].drop(['target'], axis=1)
df.head()
```

[2]:

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019908	-0.017646
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068330	-0.092204
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	-0.002592	0.002864	-0.025930
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022692	-0.009362
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031991	-0.046641

[3]:
```
df.describe()
```

[3]:

	age	sex	bmi	bp	s1
count	4.420000e+02	4.420000e+02	4.420000e+02	4.420000e+02	4.420000e+02
mean	-3.634285e-16	1.308343e-16	-8.045349e-16	1.281655e-16	-8.835316e-17

```
std    4.761905e-02  4.761905e-02  4.761905e-02  4.761905e-02  4.761905e-02
min   -1.072256e-01 -4.464164e-02 -9.027530e-02 -1.123996e-01 -1.267807e-01
25%   -3.729927e-02 -4.464164e-02 -3.422907e-02 -3.665645e-02 -3.424784e-02
50%    5.383060e-03 -4.464164e-02 -7.283766e-03 -5.670611e-03 -4.320866e-03
75%    3.807591e-02  5.068012e-02  3.124802e-02  3.564384e-02  2.835801e-02
max    1.107267e-01  5.068012e-02  1.705552e-01  1.320442e-01  1.539137e-01

                 s2            s3            s4            s5            s6
count  4.420000e+02  4.420000e+02  4.420000e+02  4.420000e+02  4.420000e+02
mean   1.327024e-16 -4.574646e-16  3.777301e-16 -3.830854e-16 -3.412882e-16
std    4.761905e-02  4.761905e-02  4.761905e-02  4.761905e-02  4.761905e-02
min   -1.156131e-01 -1.023071e-01 -7.639450e-02 -1.260974e-01 -1.377672e-01
25%   -3.035840e-02 -3.511716e-02 -3.949338e-02 -3.324879e-02 -3.317903e-02
50%   -3.819065e-03 -6.584468e-03 -2.592262e-03 -1.947634e-03 -1.077698e-03
75%    2.984439e-02  2.931150e-02  3.430886e-02  3.243323e-02  2.791705e-02
max    1.987880e-01  1.811791e-01  1.852344e-01  1.335990e-01  1.356118e-01
```

[4]:
```
df.corr()
```

[4]:
```
          age       sex       bmi        bp        s1        s2        s3  \
age  1.000000  0.173737  0.185085  0.335427  0.260061  0.219243 -0.075181
sex  0.173737  1.000000  0.088161  0.241013  0.035277  0.142637 -0.379090
bmi  0.185085  0.088161  1.000000  0.395415  0.249777  0.261170 -0.366811
bp   0.335427  0.241013  0.395415  1.000000  0.242470  0.185558 -0.178761
s1   0.260061  0.035277  0.249777  0.242470  1.000000  0.896663  0.051519
s2   0.219243  0.142637  0.261170  0.185558  0.896663  1.000000 -0.196455
s3  -0.075181 -0.379090 -0.366811 -0.178761  0.051519 -0.196455  1.000000
s4   0.203841  0.332115  0.413807  0.257653  0.542207  0.659817 -0.738493
s5   0.270777  0.149918  0.446159  0.393478  0.515501  0.318353 -0.398577
s6   0.301731  0.208133  0.388680  0.390429  0.325717  0.290600 -0.273697

          s4        s5        s6
age  0.203841  0.270777  0.301731
sex  0.332115  0.149918  0.208133
bmi  0.413807  0.446159  0.388680
bp   0.257653  0.393478  0.390429
s1   0.542207  0.515501  0.325717
s2   0.659817  0.318353  0.290600
s3  -0.738493 -0.398577 -0.273697
s4   1.000000  0.617857  0.417212
s5   0.617857  1.000000  0.464670
s6   0.417212  0.464670  1.000000
```

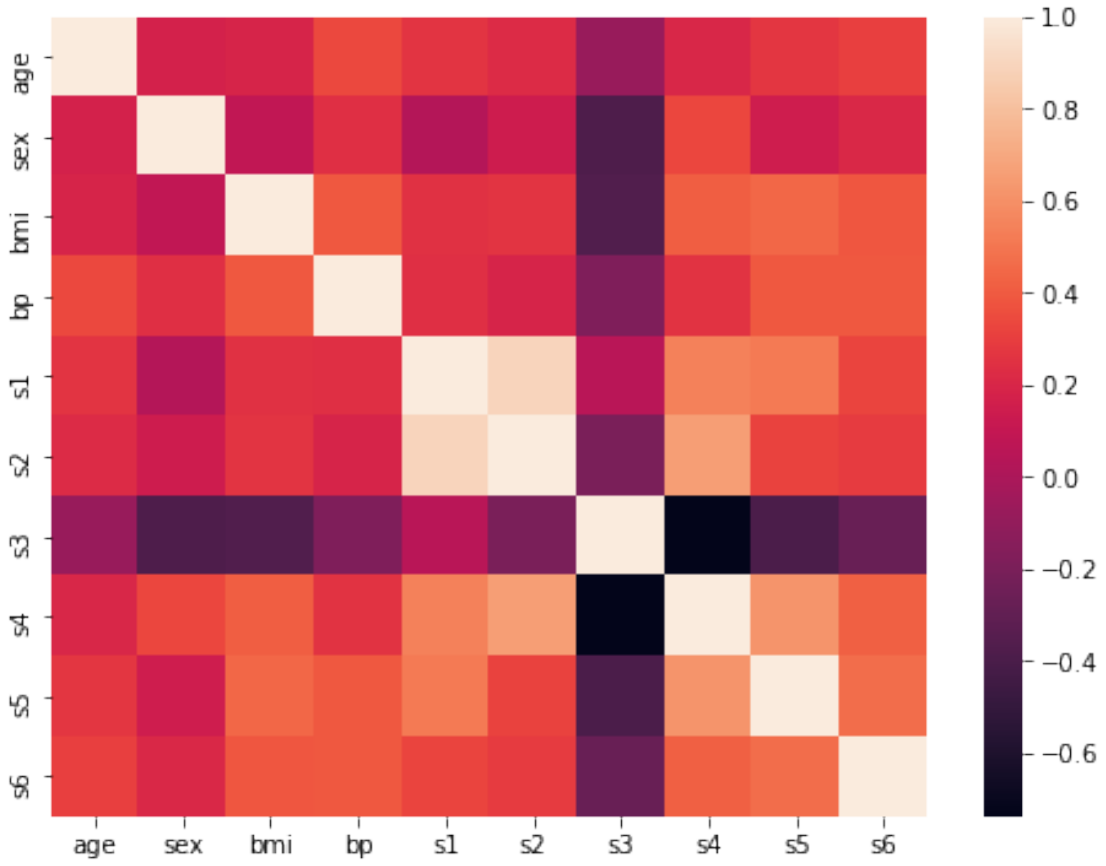
[5]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 442 entries, 0 to 441
```

```
Data columns (total 10 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   age     442 non-null    float64
 1   sex     442 non-null    float64
 2   bmi     442 non-null    float64
 3   bp      442 non-null    float64
 4   s1      442 non-null    float64
 5   s2      442 non-null    float64
 6   s3      442 non-null    float64
 7   s4      442 non-null    float64
 8   s5      442 non-null    float64
 9   s6      442 non-null    float64
dtypes: float64(10)
memory usage: 34.7 KB
```

[6]:
```
fig_dims = (8,6)
fig, ax = plt.subplots(figsize=fig_dims)
sns.heatmap(df.corr(), ax=ax)
plt.show
```

[6]: <function matplotlib.pyplot.show(close=None, block=None)>



[7]:
```
X = df[['age', 'sex', 'bmi', 's1', 's4', 's5', 's6']] #remove s2 and s3 due to␣
 ↪high feature correlation with s1 and s4
y = df[['bp']].to_numpy().reshape(-1)
```

[8]:
```
#splitting ddata to training, validation and test splits
X_train, X_rem, y_train, y_rem = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=1)
X_val, X_test, y_val, y_test = train_test_split(X_rem, y_rem, test_size=0.5,␣
 ↪random_state=1)
```

[9]:
```
degrees=[2,3,4,5,10]
tr_error=[0,0,0,0,0]
val_error=[0,0,0,0,0]
test_error=[0,0,0,0,0]

for i, degree in enumerate(degrees):

    poly = PolynomialFeatures(degree=degree)
    X_train_poly = poly.fit_transform(X_train)
    lr = LinearRegression(fit_intercept=True)
    lr.fit(X_train_poly, y_train)
#     train
    y_pred_train = lr.predict(X_train_poly)
    tr_error[i] = mean_squared_error(y_train, y_pred_train)
#     val
    X_val_poly = poly.fit_transform(X_val)
    y_pred_val = lr.predict(X_val_poly)
    val_error[i] = mean_squared_error(y_val, y_pred_val)
#     test
    X_test_poly = poly.fit_transform(X_test)
    y_pred_test = lr.predict(X_test_poly)
    test_error[i] = mean_squared_error(y_test, y_pred_test)



print("\ntraining errors for degrees [2,3,4,5,10]:\n",tr_error)
print('\n')
print("validation errors for degrees [2,3,4,5,10]:\n",val_error)
print('\n')
print("test errors for degrees [2,3,4,5,10]:\n",test_error)
print('\n')
print("Difference between training and validation error in degree 2 (best):\n",␣
 ↪np.amin(np.subtract(val_error, tr_error)))
```

```
training errors for degrees [2,3,4,5,10]:
 [0.001410614017787662, 0.0010708729054799354, 0.0030201299770942925,
5.135354863634959e-28, 1.2490211490988054e-25]


validation errors for degrees [2,3,4,5,10]:
 [0.0017843381211670771, 0.001714405608605392, 0.42980072168927863,
0.5846283012656586, 0.5725205426483448]


test errors for degrees [2,3,4,5,10]:
 [0.0016405080230570389, 0.0023833148713588654, 0.2901620338719492,
1.5723619551673176, 1.565186440679949]


Difference between training and validation error in degree 2 (best):
 0.00037372410337941503
```

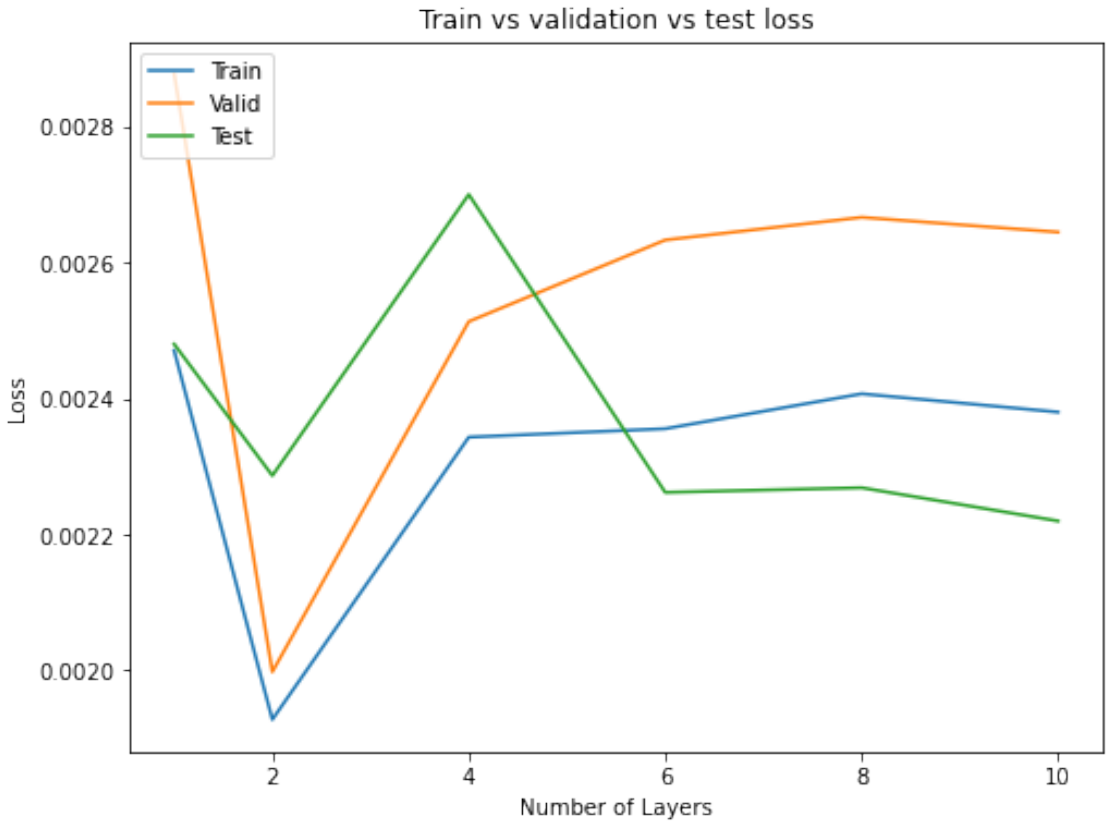
[10]:
```
plt.figure(figsize=(8, 6))


plt.plot(degrees, tr_error, label = 'Train')
plt.plot(degrees, val_error, label = 'Valid')
plt.plot(degrees, test_error,label = 'Test')
plt.legend(loc = 'upper left')

plt.xlabel('Degree')
plt.ylabel('Loss')
plt.title('Train vs validation vs test loss')
plt.show()
```

[11]:
```
num_layers = [1,2,4,6,8,10]
num_neurons = 10

mlp_tr_errors = []
mlp_val_errors = []
mlp_test_errors = []

for i, num in enumerate(num_layers):
    hiden_layer_sizes = tuple([num_neurons]*num)

    mlp_regr = MLPRegressor(hidden_layer_sizes=hiden_layer_sizes, max_iter=500,␣
 ↪random_state=1)
    mlp_regr.fit(X_train, y_train)
#     train
    y_mlp_pred_train = mlp_regr.predict(X_train)
    mlp_tr_error = mean_squared_error(y_train, y_mlp_pred_train)
#     val
    y_mlp_pred_val = mlp_regr.predict(X_val)
    mlp_val_error = mean_squared_error(y_val, y_mlp_pred_val)
#     test
```

```
    y_mlp_pred_test = mlp_regr.predict(X_test)
    mlp_test_error = mean_squared_error(y_test, y_mlp_pred_test)

    mlp_tr_errors.append(mlp_tr_error)
    mlp_val_errors.append(mlp_val_error)
    mlp_test_errors.append(mlp_test_error)

print("\ntraining error for num_layers[1,2,4,6,8,10]:\n",mlp_tr_errors)
print("\nvalidatoin error for num_layers[1,2,4,6,8,10]:\n",mlp_val_errors)
print("\ntest error for num_layers[1,2,4,6,8,10]:\n",mlp_test_errors)
```

```
training error for num_layers[1,2,4,6,8,10]:
 [0.002470443487171304, 0.0019278035462463267, 0.002343036671108225,
0.0023553871504723993, 0.0024067789160648113, 0.002379838107348869]

validatoin error for num_layers[1,2,4,6,8,10]:
 [0.002877597663565093, 0.0019979312935338995, 0.0025130200838093526,
0.0026329240644531984, 0.0026662623391704317, 0.002644517327250895]

test error for num_layers[1,2,4,6,8,10]:
 [0.0024798421613049628, 0.0022864456484247087, 0.002699887995185276,
0.002261791101676557, 0.002268598838949037, 0.0022197119231000748]
```

[12]:
```
plt.figure(figsize=(8, 6))

plt.plot(num_layers, mlp_tr_errors, label = 'Train')
plt.plot(num_layers, mlp_val_errors,label = 'Valid')
plt.plot(num_layers, mlp_test_errors,label = 'Test')
plt.legend(loc = 'upper left')

plt.xlabel('Number of Layers')
plt.ylabel('Loss')
plt.title('Train vs validation vs test loss')
plt.show()
```


Train vs validation vs test loss
Train
Valid
Test
0.0028
0.0026
0.0024
0.0022
0.0020
Loss
2
4
6
8
10
Number of Layers