Chem277B: Machine Learning Algorithms

Homework assignment #4: Regression

In [110…

```
import numpy as np
from numpy import linalg as LA
import time
from pylab import *
import matplotlib.pyplot as plt
import math
import scipy
import pandas as pd
import numba
from sklearn.preprocessing import OneHotEncoder
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error
from mpl_toolkits.mplot3d import Axes3D
```

1. Linear regression using a simple perceptron.

(a) I think the features related to Chance of Admit are: GRE Score, TOEFL Score, University Rating, SOP, LOR, CGPA, Research.

The normalized table are stored in admission_norm.

In [19]:

```
admission = pd.read_csv('Admission_Predict_Ver1.1.csv')
admission.head()
```

Out[19]:

	Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
0	1	337	118	4	4.5	4.5	9.65	1	0.92
1	2	324	107	4	4.0	4.5	8.87	1	0.76
2	3	316	104	3	3.0	3.5	8.00	1	0.72
3	4	322	110	3	3.5	2.5	8.67	1	0.80
4	5	314	103	2	2.0	3.0	8.21	0	0.65

In [20]:

```
admission_norm = admission.iloc[:, 1:8]
admission_norm = (admission_norm - np.mean(admission_norm, axis=0)) / np.std(admission_norm, axis=0)
admission_norm.head()
```

Out[20]:

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research
0	1.819238	1.778865	0.775582	1.137360	1.098944	1.776806	0.886405
1	0.667148	-0.031601	0.775582	0.632315	1.098944	0.485859	0.886405
2	-0.041830	-0.525364	-0.099793	-0.377773	0.017306	-0.954043	0.886405
3	0.489904	0.462163	-0.099793	0.127271	-1.064332	0.154847	0.886405
4	-0.219074	-0.689952	-0.975168	-1.387862	-0.523513	-0.606480	-1.128152

(b) The simple perceptron class is presented as below.

The weights are initialized as a uniform distribution between 0 to 0.05.

I also used MSE to adjust weights and biases in fit and evaluate methods.

The predict method is also filled.

In [36]:

```
class simple_perceptron():
    def __init__(self,input_dim,output_dim,learning_rate=0.01,activation=lambda x:x,activation_grad=lambda x:1):

        self.input_dim=input_dim
        self.output_dim=output_dim
        self.activation=activation
        self.activation_grad=activation_grad
        self.lr=learning_rate
        ### initialize parameters ###
        self.weights=np.random.rand(input_dim, output_dim) * 0.05
        self.biases=np.random.rand(1, output_dim) * 0.05

    def predict(self,X):
        if len(X.shape)==1:
            X=X.reshape((-1,1))
        dim=X.shape[1]
        # Check that the dimension of accepted input data is the same as expected
        if not dim==self.input_dim:
            raise Exception("Expected input size %d, accepted %d!"%(self.input_dim,dim))
        ### Calculate logit and activation ###
        self.z = X.dot(self.weights) + self.biases             #shape(X.shape[0],1)
        self.a = self.activation(self.z)             #shape(X.shape[0],1)
        return self.a

    def fit(self,X,y):
        # Transform the single-sample data into 2-dimensional, for the convenience of matrix multiplication
        if len(X.shape)==1:
            X=X.reshape((-1,1))
        if len(y.shape)==1:
```

```
            y=y.reshape((-1,1))
        self.predict(X)
        errors=(self.a-y)*self.activation_grad(self.z)
        weights_grad=errors.T.dot(X)
        bias_grad=np.sum(errors,axis=0)
        ### Update weights and biases from the gradient ###
        self.weights -= self.lr * weights_grad.T
        self.biases -= self.lr * bias_grad


    def train_on_epoch(self,X,y,batch_size=32):
        # Every time select batch_size samples from the training set, until all data in the training set has been train
        order=list(range(X.shape[0]))
        np.random.shuffle(order)
        n=0
        while n<math.ceil(len(order)/batch_size)-1: # Parts that can fill one batch
            self.fit(X[order[n*batch_size:(n+1)*batch_size]],y[order[n*batch_size:(n+1)*batch_size]])
            n+=1
        # Parts that cannot fill one batch
        self.fit(X[order[n*batch_size:]],y[order[n*batch_size:]])

    def evaluate(self,X,y):
         # Transform the single-sample data into 2-dimensional
        if len(X.shape)==1:
            X=X.reshape((1,-1))
        if len(y.shape)==1:
            y=y.reshape((1,-1))
        ### means square error ###
        return np.mean((self.predict(X) - y)**2)

    def get_weights(self):
        return (self.weights,self.biases)

    def set_weights(self,weights):
        self.weights=weights[0]
        self.biases=weights[1]
```

(c) The filled KFold codes are listed below.

80% training data and 20% testing data are set using train_test_split class.

5-fold validation is also listed when calling the Kfold function.

After data training, all 5-fold data converged to an error of < 0.05. The correlation between predicted data y^hat and y is >0.9, indicating reasonable prediction.

After removing GRE scores,all 5-fold data still converged to an error of < 0.05. The newly acquired correlation between predicted data y^hat and y is >0.9, indicating reasonable prediction even without GRE scores.

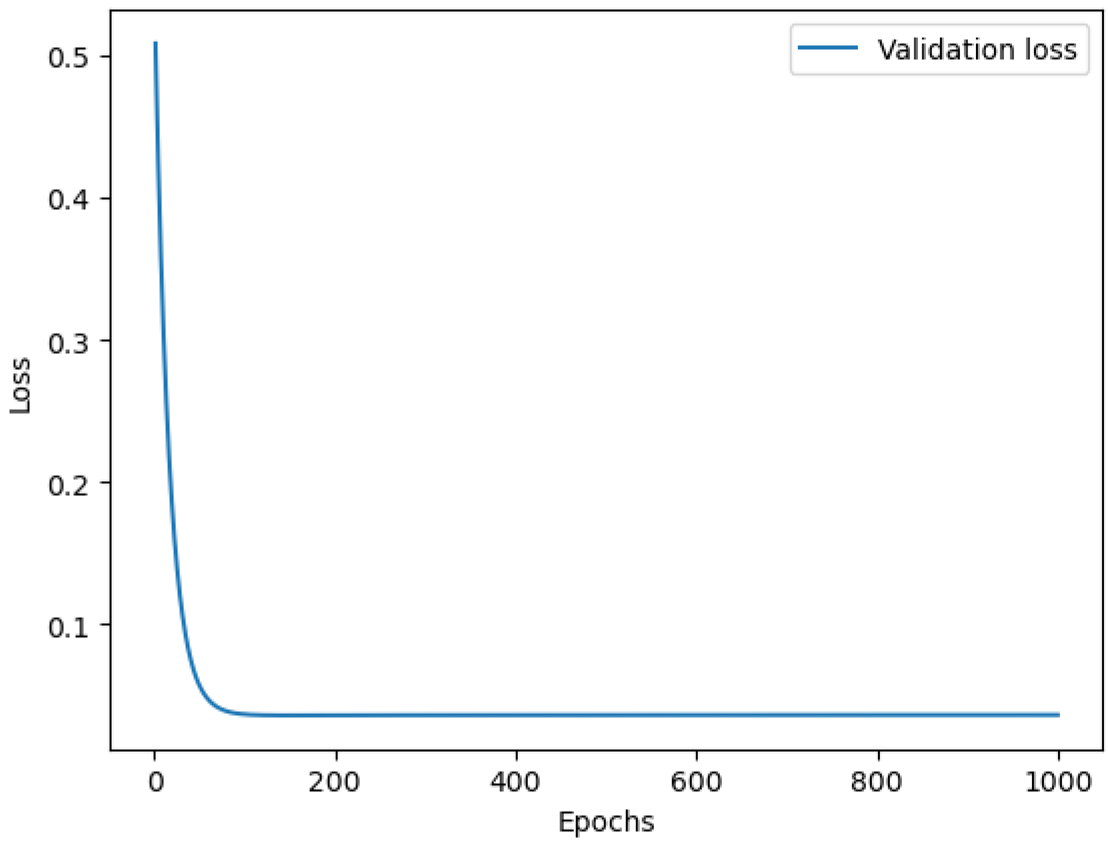
In [59]:

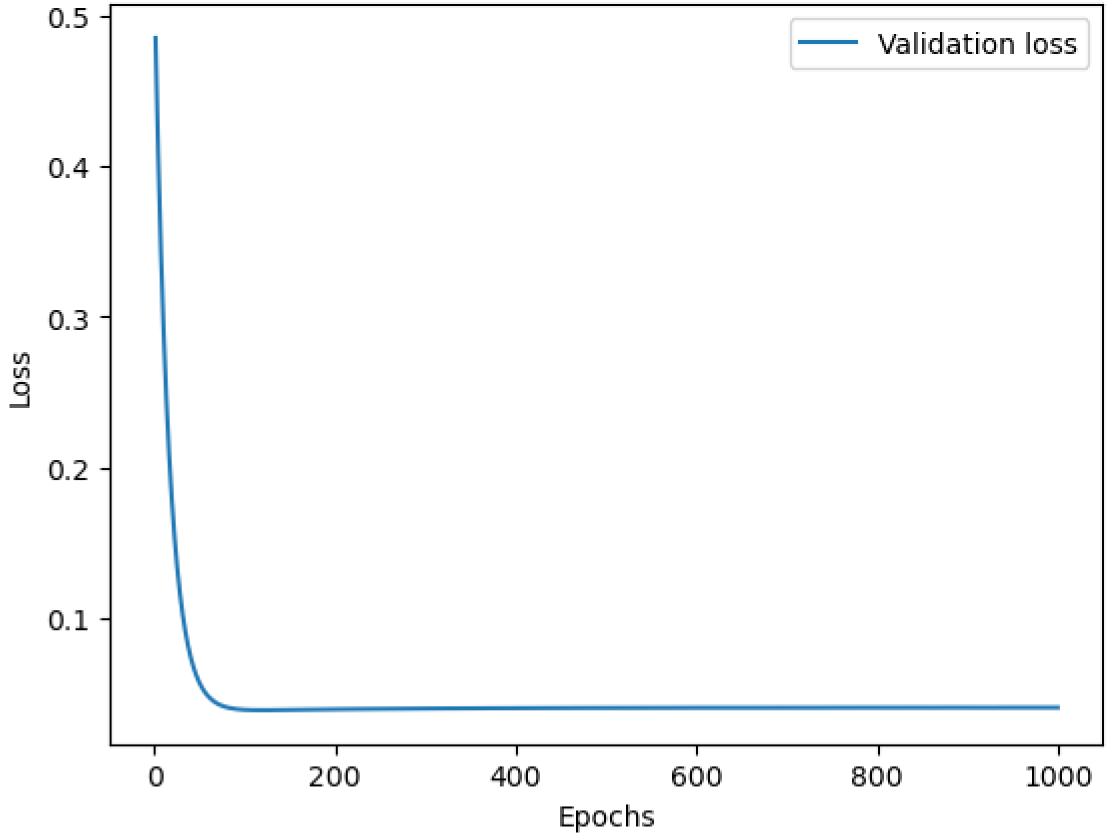
```python
from sklearn.model_selection import train_test_split,KFold

def Kfold(k,Xs,ys,epochs,learning_rate=0.0001,draw_curve=True):
    # The total number of examples for training the network
    total_num=len(Xs)

    # Built in K-fold function in Sci-Kit Learn
    kf=KFold(n_splits=k,shuffle=True)
    # record error for each model
    train_error_all=[]
    test_error_all=[]

    for train_selector,test_selector in kf.split(range(total_num)):
        ### Decide training examples and testing examples for this fold ###
        train_Xs=Xs[train_selector]
        test_Xs=Xs[test_selector]
        train_ys=ys[train_selector]
        test_ys=ys[test_selector]


        val_array=[]
        # Split training examples further into training and validation
        # 20% test data is set using test_size
        train_in,val_in,train_real,val_real=train_test_split(train_Xs,train_ys,test_size=0.2)

        ### Establish the model for simple perceptron here ###
        model=simple_perceptron(Xs.shape[1], 1, learning_rate=learning_rate)

        # Save the lowest weights, so that we can recover the best model
        weights = model.get_weights()
        lowest_val_err = np.inf
        for _ in range(epochs):
            # Train model on a number of epochs, and test performance in the validation set
            model.train_on_epoch(train_in,train_real)
            val_err = model.evaluate(val_in,val_real)
            val_array.append(val_err)
            if val_err < lowest_val_err:
                lowest_val_err = val_err
                weights = model.get_weights()

        # The final number of epochs is when the minimum error in validation set occurs
        final_epochs=np.argmin(val_array)+1                 #+1 for indexing
        print("Number of epochs with lowest validation:",final_epochs)
        # Recover the model weight
        model.set_weights(weights)

        # Report result for this fold
        train_error=model.evaluate(train_Xs, train_ys)
```

```
        train_error_all.append(train_error)
        test_error=model.evaluate(test_Xs, test_ys)
        test_error_all.append(test_error)
        print("Train error:",train_error)
        print("Test error:",test_error)
        pred = model.predict(Xs)

        if draw_curve:
            plt.figure()
            plt.plot(np.arange(len(val_array))+1,val_array,label='Validation loss')
            plt.xlabel('Epochs')
            plt.ylabel('Loss')
            plt.legend()

    print("Final results:")
    print("Training error:%f+-%f"%(np.average(train_error_all),np.std(train_error_all)))
    print("Testing error:%f+-%f"%(np.average(test_error_all),np.std(test_error_all)))

    # return the last model
    return model, pred

def show_correlation(xs,ys):
    plt.figure()
    plt.scatter(xs,ys,s=0.5)
    r = [np.min([np.min(xs),np.min(ys)]),np.max([np.max(xs),np.max(ys)])]
    plt.plot(r,r,'r')
    plt.xlabel("Predictions")
    plt.ylabel("Ground truth")
    corr=np.corrcoef([xs,ys])[1,0]
    print("Correlation coefficient:",corr)
```

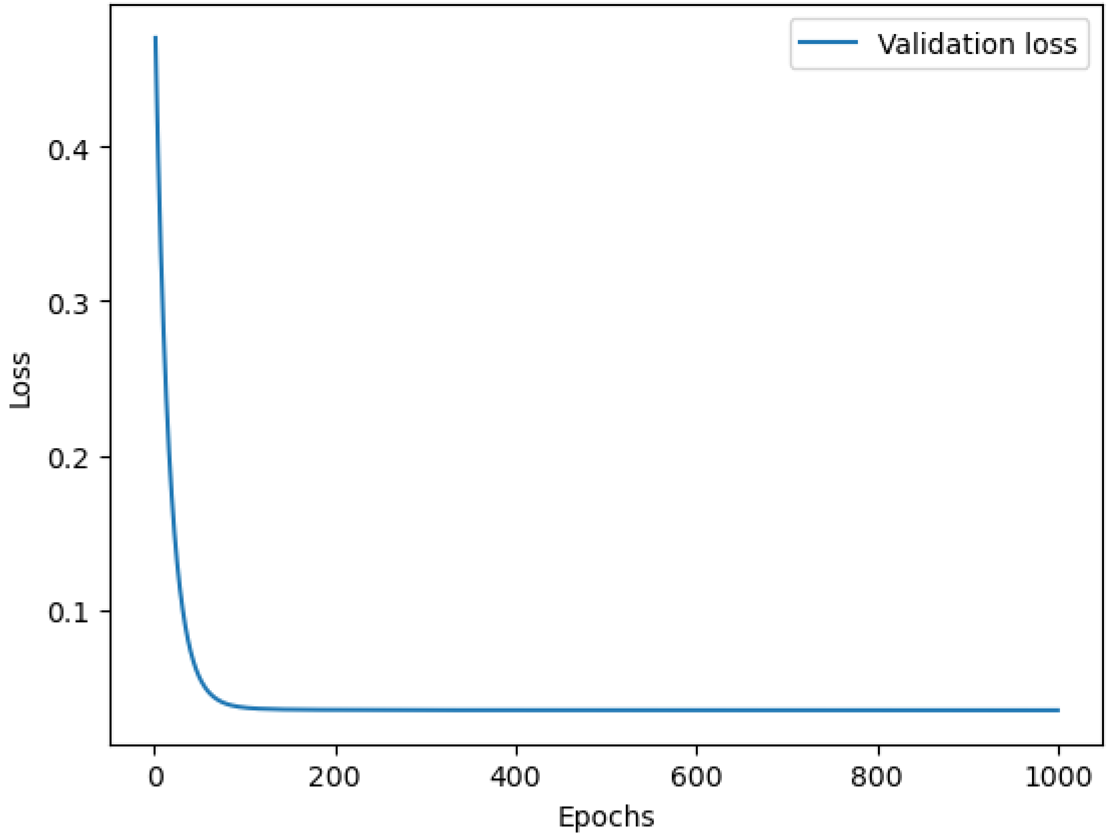
In [60]:
```
prediction_1c_1, pred = Kfold(5,admission_norm.values,admission['Chance of Admit '].to_numpy(),epochs=1000,learning_rat
```

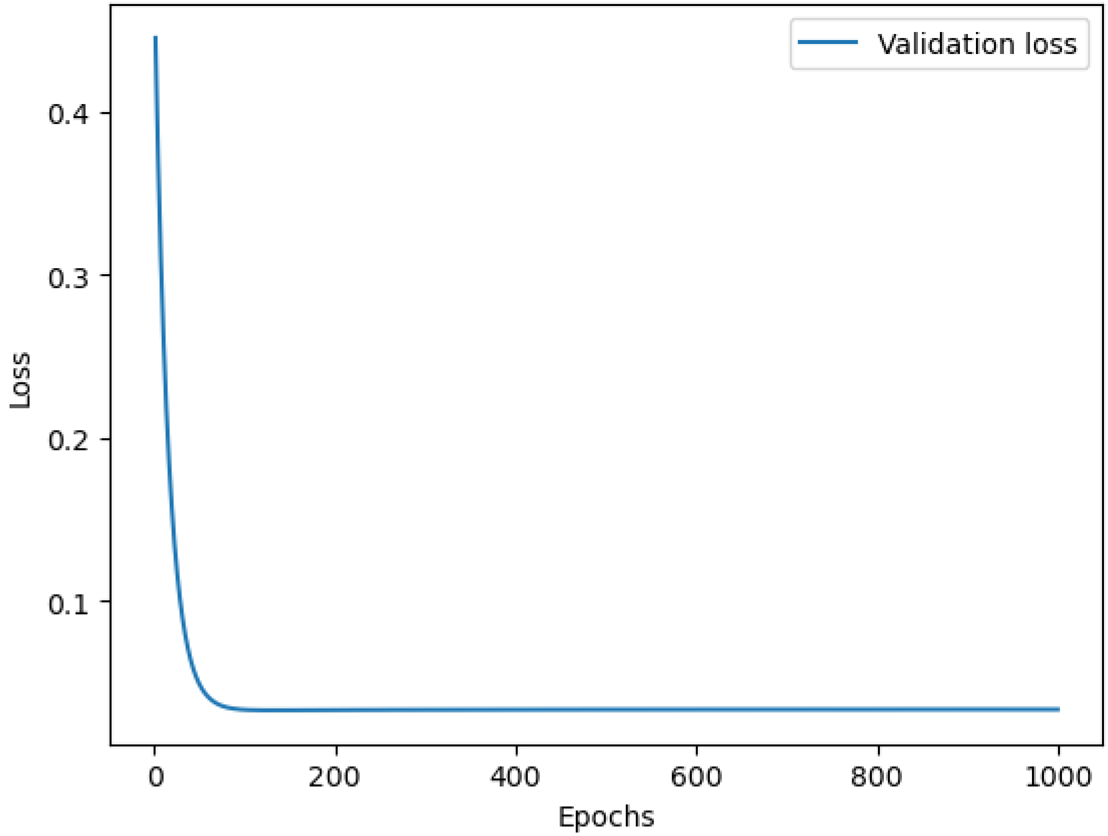
```
Number of epochs with lowest validation: 143
Train error: 0.035288333989616845
Test error: 0.03774393066804139
Number of epochs with lowest validation: 120
Train error: 0.038009206121829536
Test error: 0.028760743210197337
Number of epochs with lowest validation: 829
Train error: 0.03699164768384797
Test error: 0.035475453109607986
Number of epochs with lowest validation: 126
Train error: 0.03515604696126015
Test error: 0.03894520919341821
Number of epochs with lowest validation: 166
Train error: 0.03509131875609637
Test error: 0.03820913876176258
Final results:
Training error:0.036107+-0.001184
Testing error:0.035827+-0.003718
```



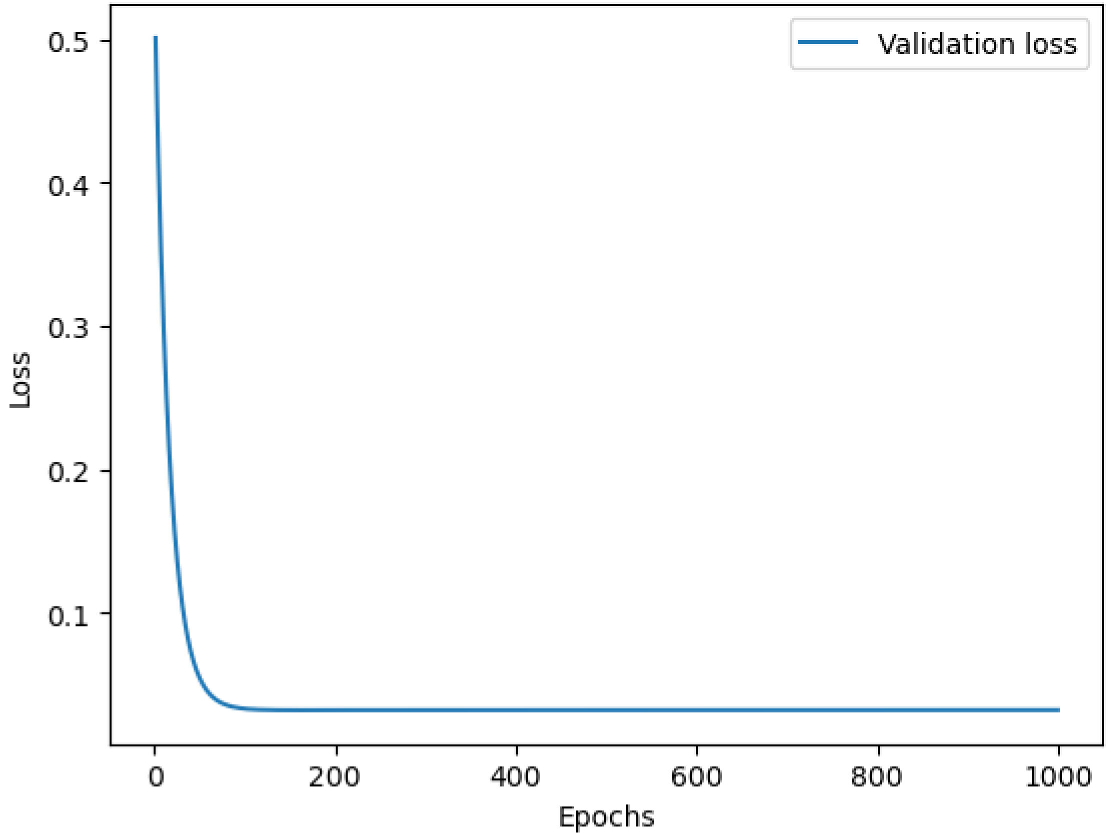


Validation loss
Loss
Epochs
0.5
0.4
0.3
0.2
0.1
0
200
400
600
800
1000


Validation loss
0.4
0.3
0.2
0.1
Loss
0
200
400
600
800
1000
Epochs

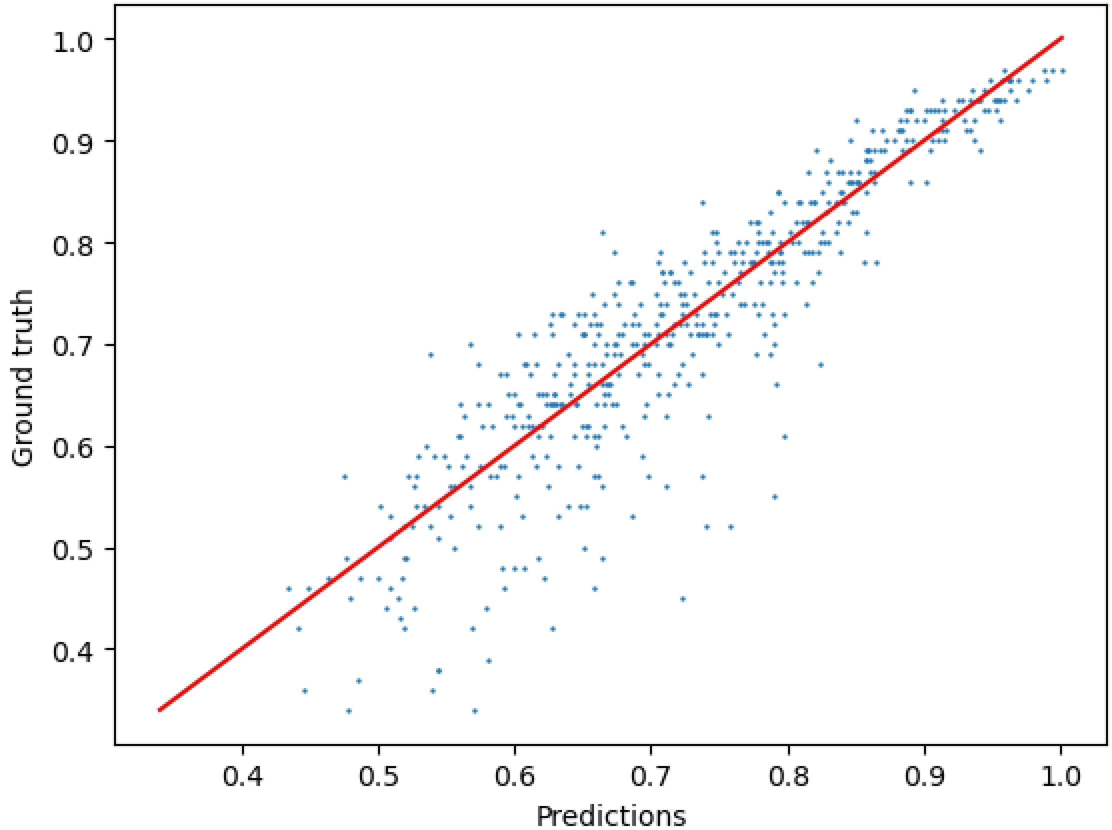


Validation loss
0.4
0.3
0.2
0.1
Loss
0
200
400
600
800
1000
Epochs



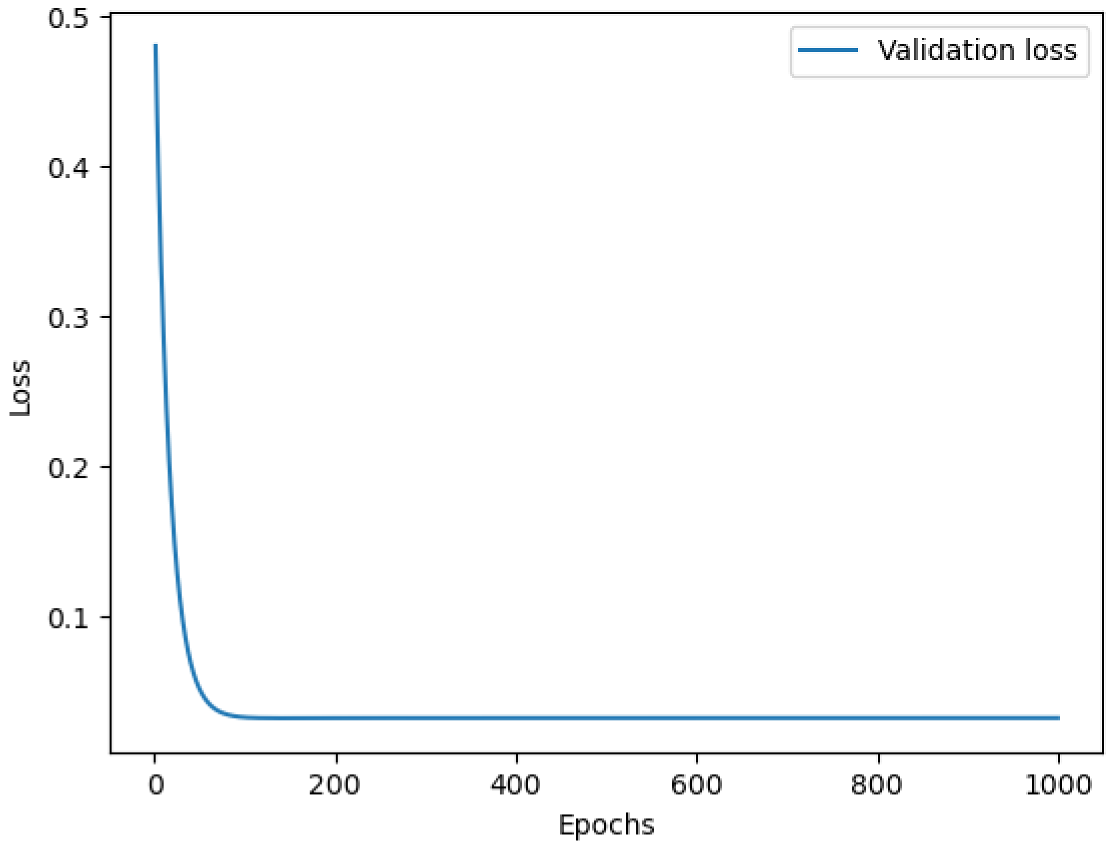
In [61]:
```
show_correlation(pred.reshape(-1,),admission['Chance of Admit '].to_numpy())
```

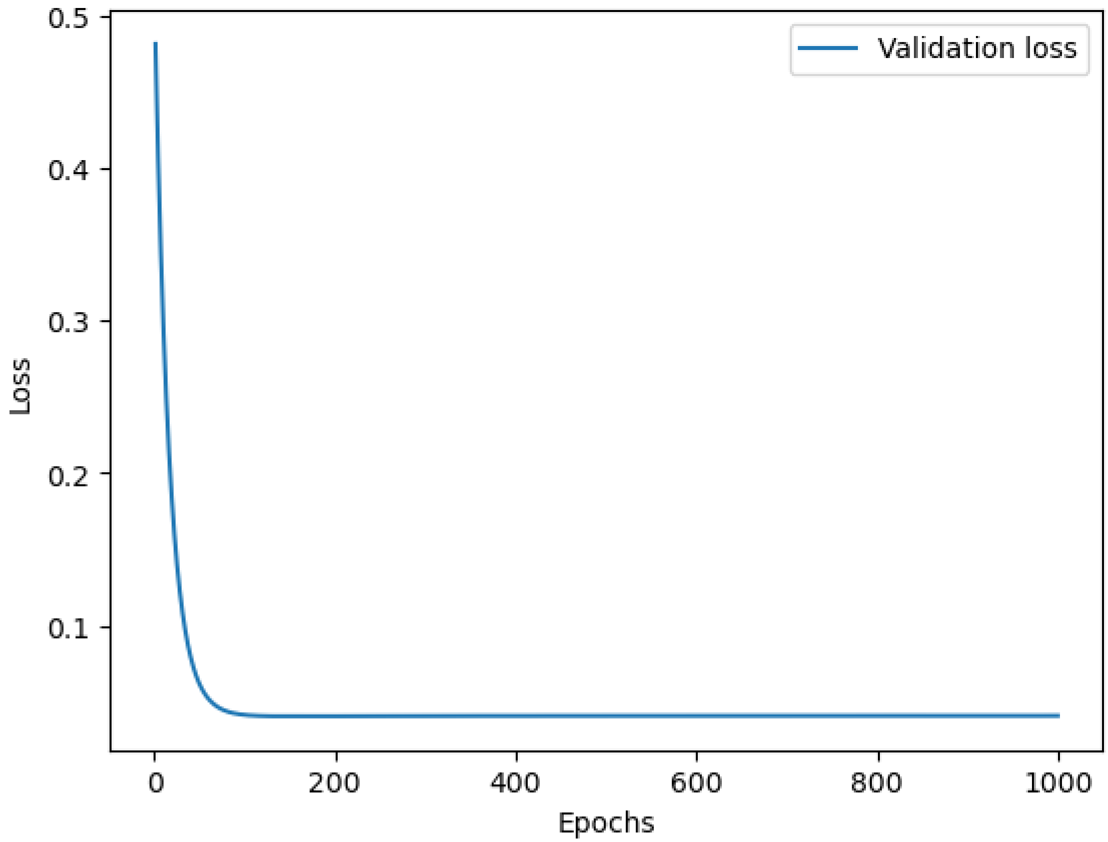
```
Correlation coefficient: 0.9060140619720447
```



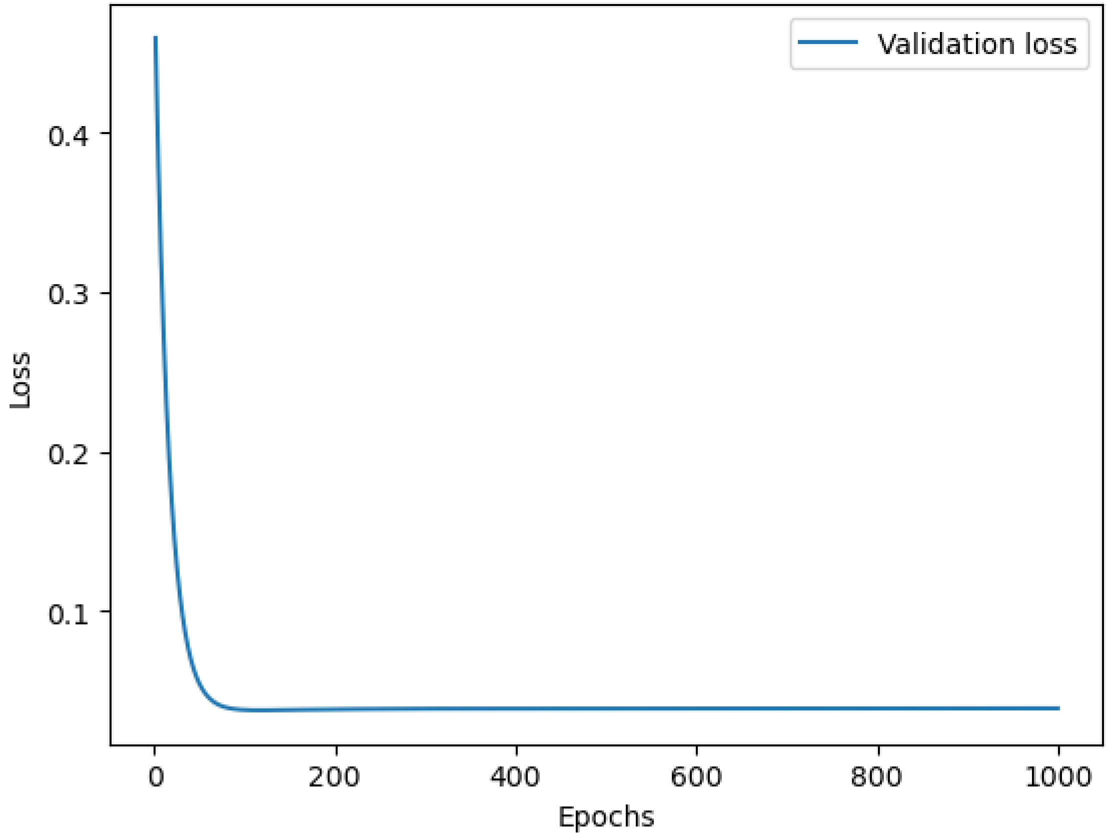
In [67]:
```
admission_norm_noGRE = admission_norm.drop(columns=['GRE Score'])
prediction_1c_2, pred_noGRE = Kfold(5,admission_norm_noGRE.values,admission['Chance of Admit '].to_numpy(),epochs=1000,
```

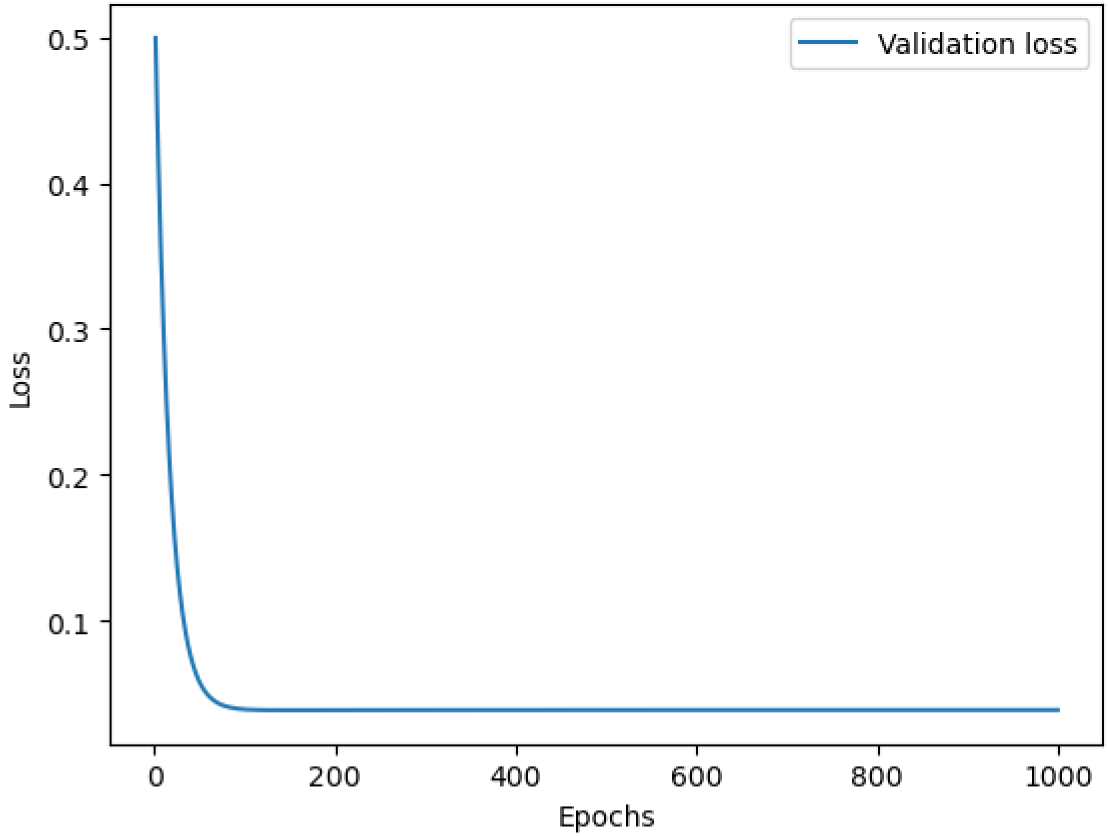
```
Number of epochs with lowest validation: 136
Train error: 0.036473156104575306
Test error: 0.03544032711569678
Number of epochs with lowest validation: 160
Train error: 0.033140733670702686
Test error: 0.04179780105260108
Number of epochs with lowest validation: 116
Train error: 0.03663507453834195
Test error: 0.035378555543794314
Number of epochs with lowest validation: 174
Train error: 0.03939639095572981
Test error: 0.027042724118376896
Number of epochs with lowest validation: 140
Train error: 0.03469088880819279
Test error: 0.03925920057935457
Final results:
Training error:0.036067+-0.002099
Testing error:0.035784+-0.004999
```


Validation loss
Loss
Epochs
0.5
0.4
0.3
0.2
0.1
0
200
400
600
800
1000

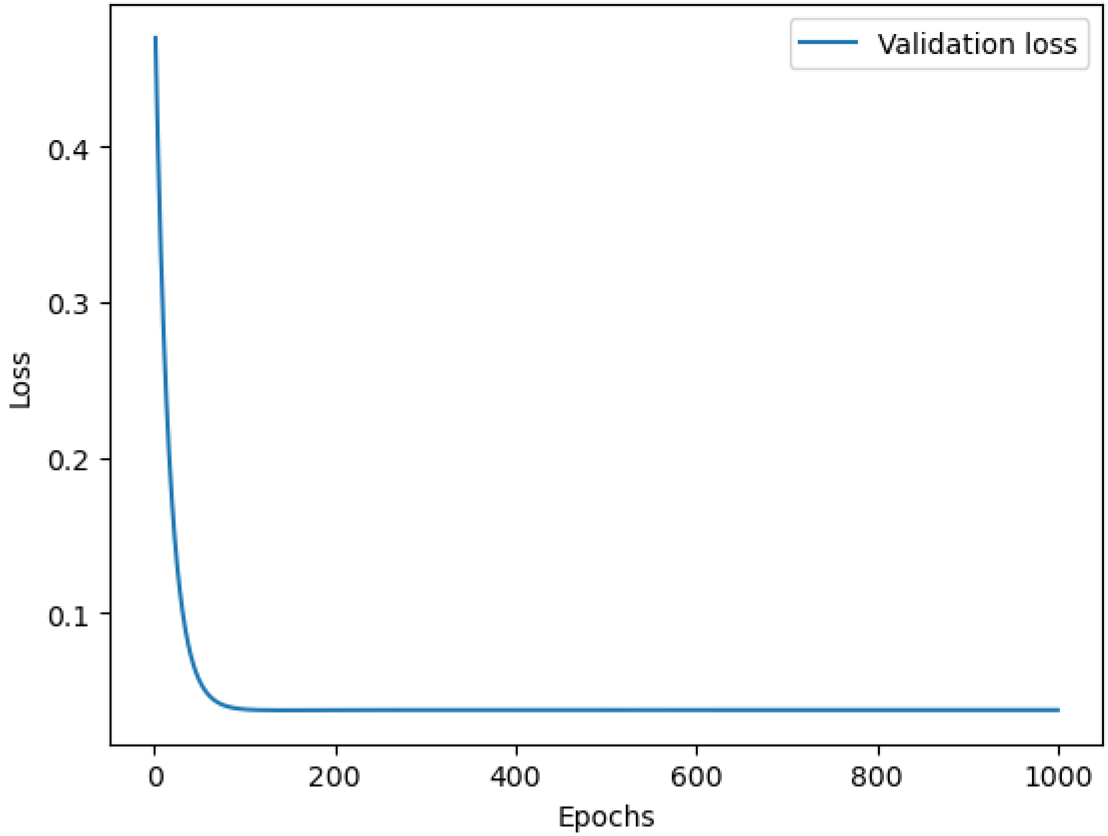


Validation loss
0.5
0.4
0.3
0.2
0.1
Loss
0
200
400
600
800
1000
Epochs


Validation loss
0.4
0.3
0.2
0.1
Loss
0
200
400
600
800
1000
Epochs

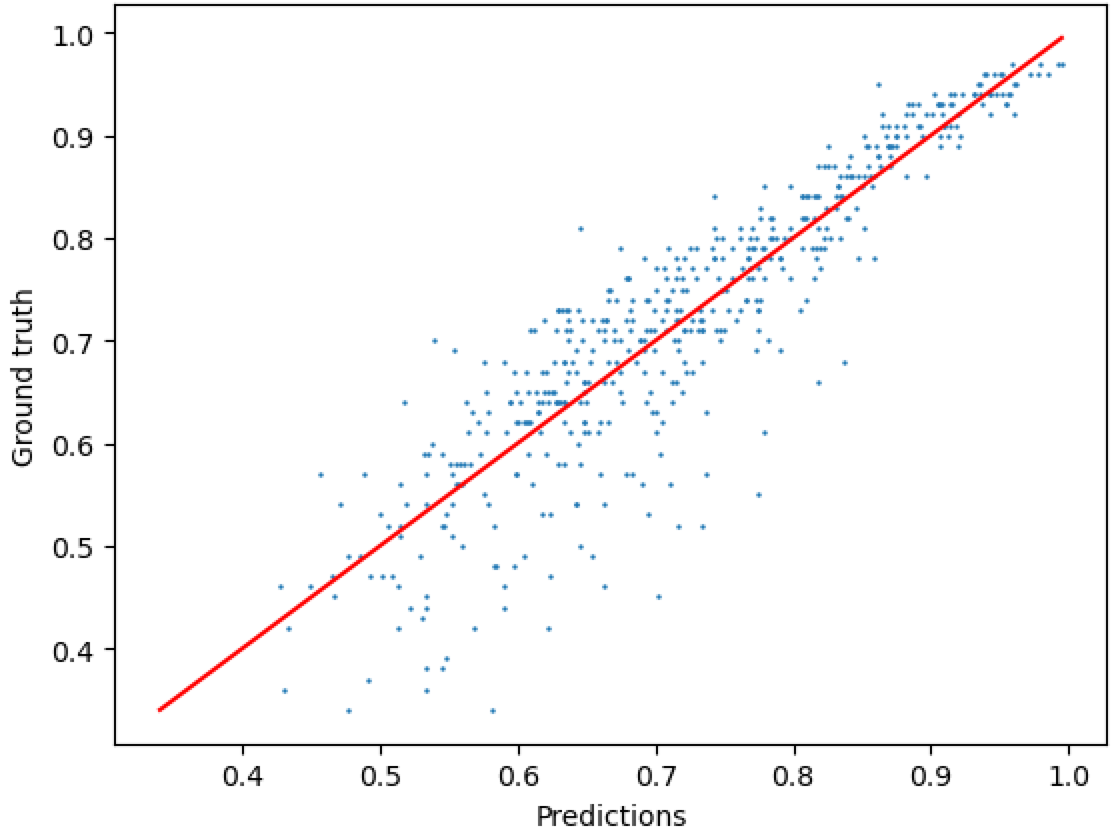


Validation loss
Loss
Epochs
0.5
0.4
0.3
0.2
0.1
0
200
400
600
800
1000



In [69]:
```
show_correlation(pred_noGRE.reshape(-1,),admission['Chance of Admit '].to_numpy())
```

```
Correlation coefficient: 0.9037401187805468
```



2. Logistic regression using a simple perceptron.

(a) By reviewing the titanic dataset, a lot of the data is qualitative categorical instead of quantitative continuous.

Hence we first selected categorical features and quantitative features out for separate processing.

For the categorical features, we used one-hot coding to categorize them into sub-columns as arrays of [0,1].

For the quantitative features, we normalized them using the standard procedures.

In the end we acquired a new dataframe with 183 rows and 12 columns/features.

In [93]:

```
titanic = pd.read_csv('titantic.csv')
titanic_filter = titanic.dropna()

# The features that are qualitative categorical and quantitative continuous
categ = ['Pclass', 'Sex', 'Embarked']
quant = ['Age', 'SibSp', 'Parch', 'Fare']
```

```
# First normalize the quantitative continuous features
titanic_filter[quant] = (titanic_filter[quant] - np.mean(titanic_filter[quant], axis=0)) / np.std(titanic_filter[quant]

# Use one-hot encoding to process the categorical features
enc = OneHotEncoder(handle_unknown='ignore')
enc.fit(titanic_filter[categ])
enc_array = enc.transform(titanic_filter[categ]).toarray()
enc_catego = pd.DataFrame(enc_array, columns = enc.get_feature_names_out(), index = titanic_filter.index)

titanic_filter_process = titanic_filter[quant].merge(enc_catego, left_index=True, right_index=True)
titanic_filter_process.head()
```

```
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_43407/1373871913.py:9: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a
-view-versus-a-copy
  titanic_filter[quant] = (titanic_filter[quant] - np.mean(titanic_filter[quant], axis=0)) / np.std(titanic_filter[quan
t], axis=0)
```

Out[93]:

	Age	SibSp	Parch	Fare	Pclass_1	Pclass_2	Pclass_3	Sex_female	Sex_male	Embarked_C	Embarked_Q	Embarked_S
1	0.149065	0.833628	-0.631730	-0.097180	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0
3	-0.043230	0.833628	-0.631730	-0.335997	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0
6	1.174636	-0.723044	-0.631730	-0.352250	1.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0
10	-2.030273	0.833628	0.697081	-0.814070	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0
11	1.431029	-0.723044	-0.631730	-0.684702	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0

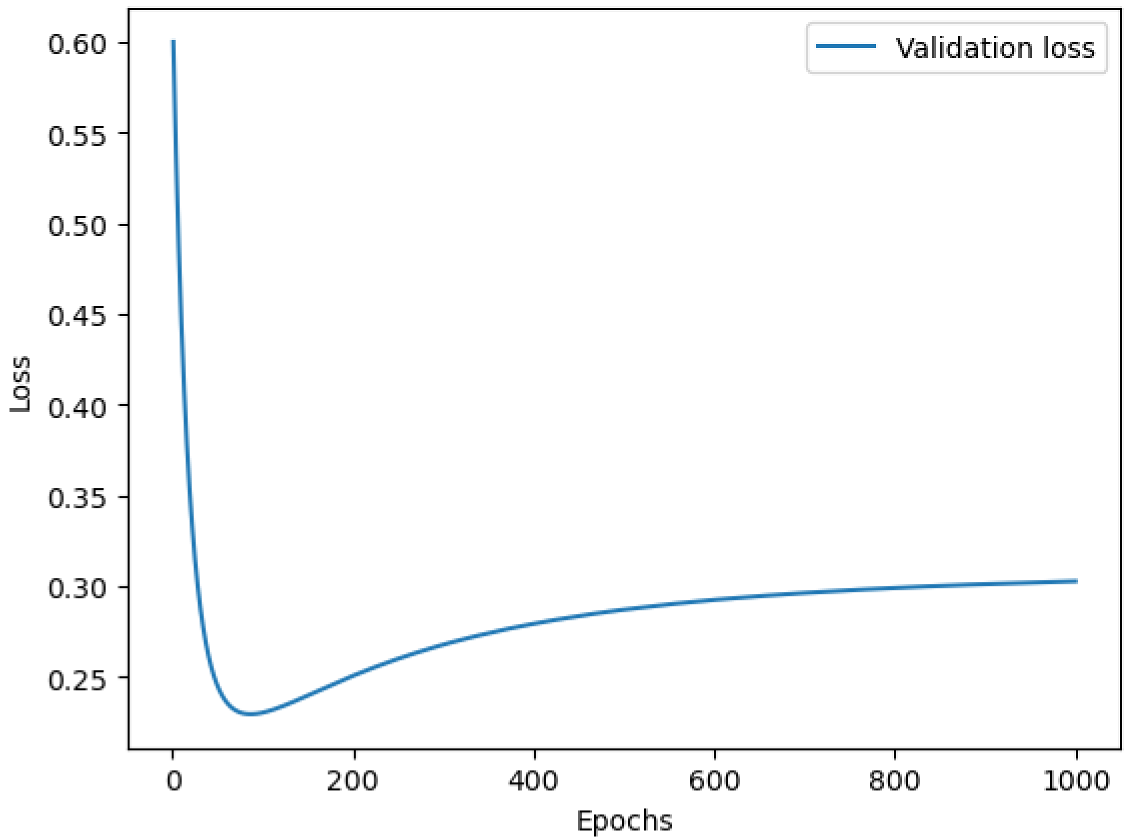
(b) I directly used the simple perceptron model from Q1 to predict the survival rate. The result is encouraging but I think the model needs to be adapted to yield two categories instead of probability. For now We end up with a training error of ~0.3 in all the 5 folds of training. The correlation between y^hat and y is ~0.57. This basically indicates weak correlation. When I looked at the prediction, I noticed that there's a threshold of 0.6. The model tends to predict survivor to be >0.6 and non-survivors to be <0.6. However the distribution is a little big and there are outliers.

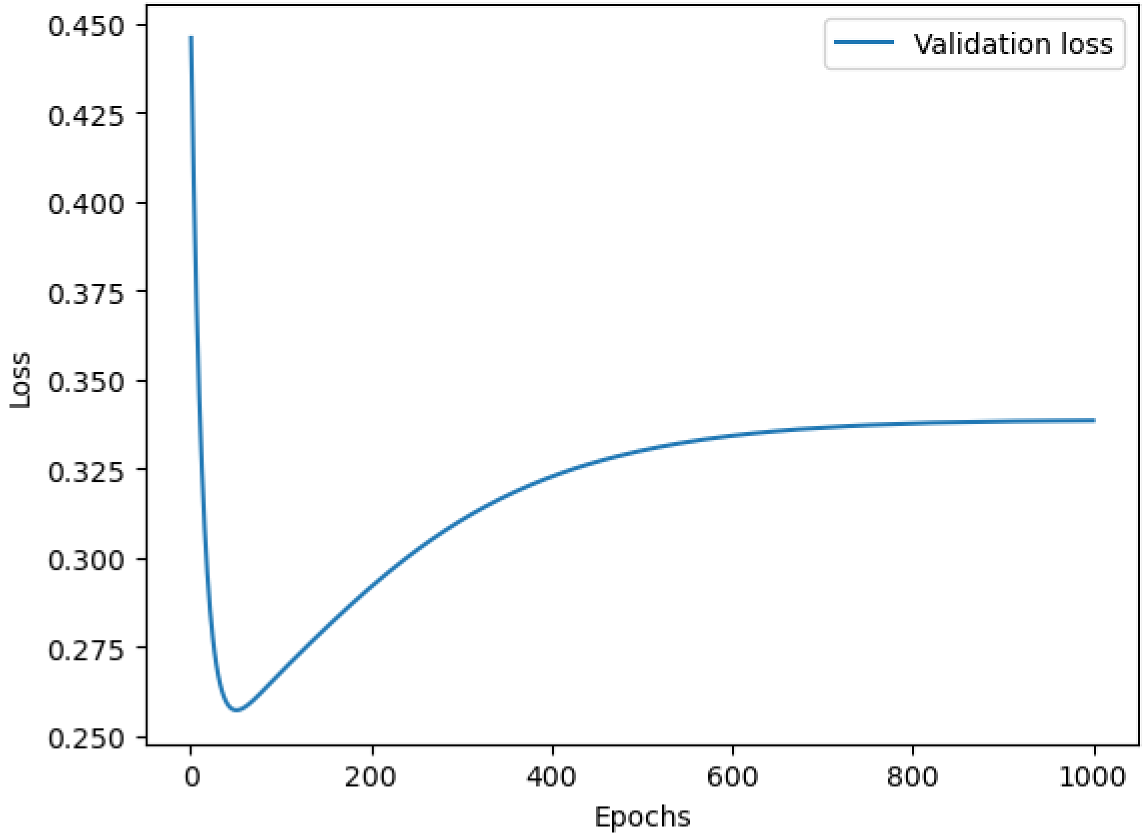
Also, when I played with all the 12 features, I noticed that 'Sex_female' has a correlation of 0.53, very close to the total correlation. I think this is the feature that shows the highest chance to survive.

In [98]:

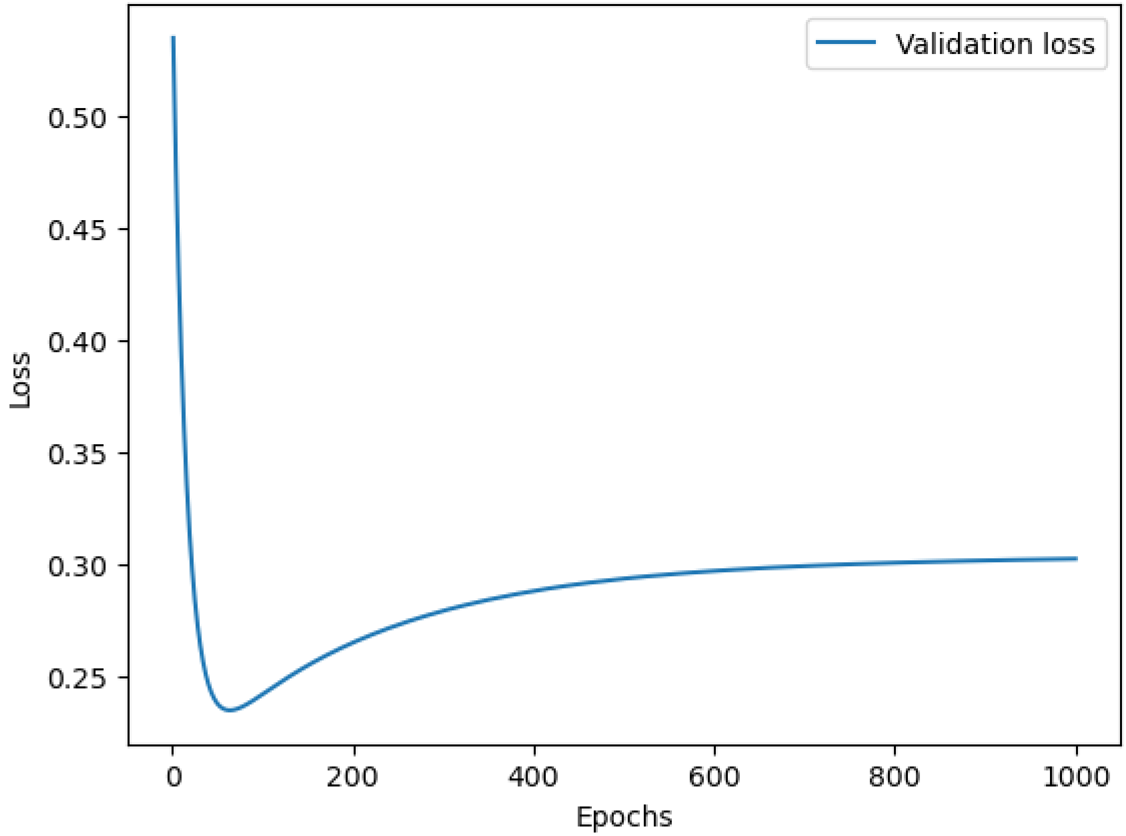
```
survival = titanic_filter['Survived']
prediction_2b_1, pred = Kfold(5,titanic_filter_process.values,survival.to_numpy(),epochs=1000,learning_rate=0.0001,draw
show_correlation(pred.reshape(-1), survival.to_numpy())
```

```
Number of epochs with lowest validation: 87
Train error: 0.30690014672135024
Test error: 0.31186969786597857
Number of epochs with lowest validation: 51
Train error: 0.3099211127626657
Test error: 0.29337907688318104
Number of epochs with lowest validation: 64
Train error: 0.2899474360100807
Test error: 0.3042574168369281
Number of epochs with lowest validation: 84
Train error: 0.2859447931708963
Test error: 0.29928765445946365
Number of epochs with lowest validation: 46
Train error: 0.2902439575066901
Test error: 0.304560249349907
Final results:
Training error:0.296591+-0.009816
Testing error:0.302671+-0.006140
Correlation coefficient: 0.5767837718637085
```

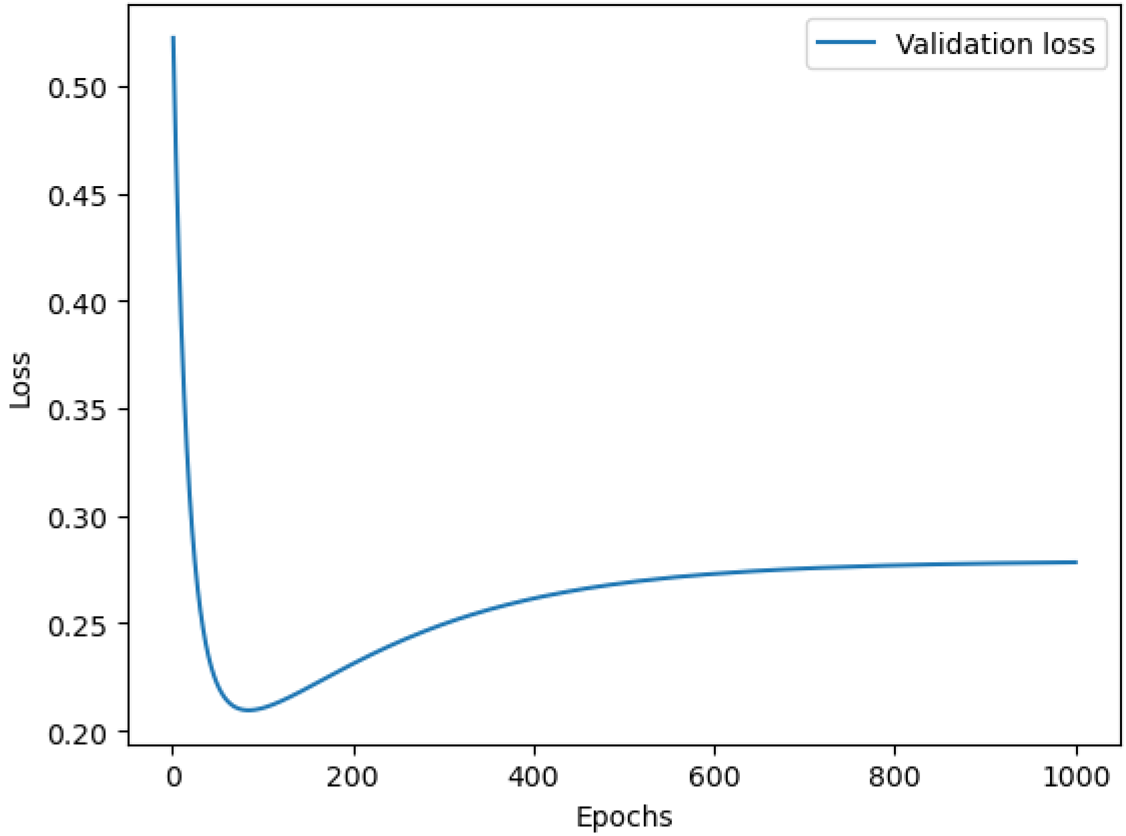




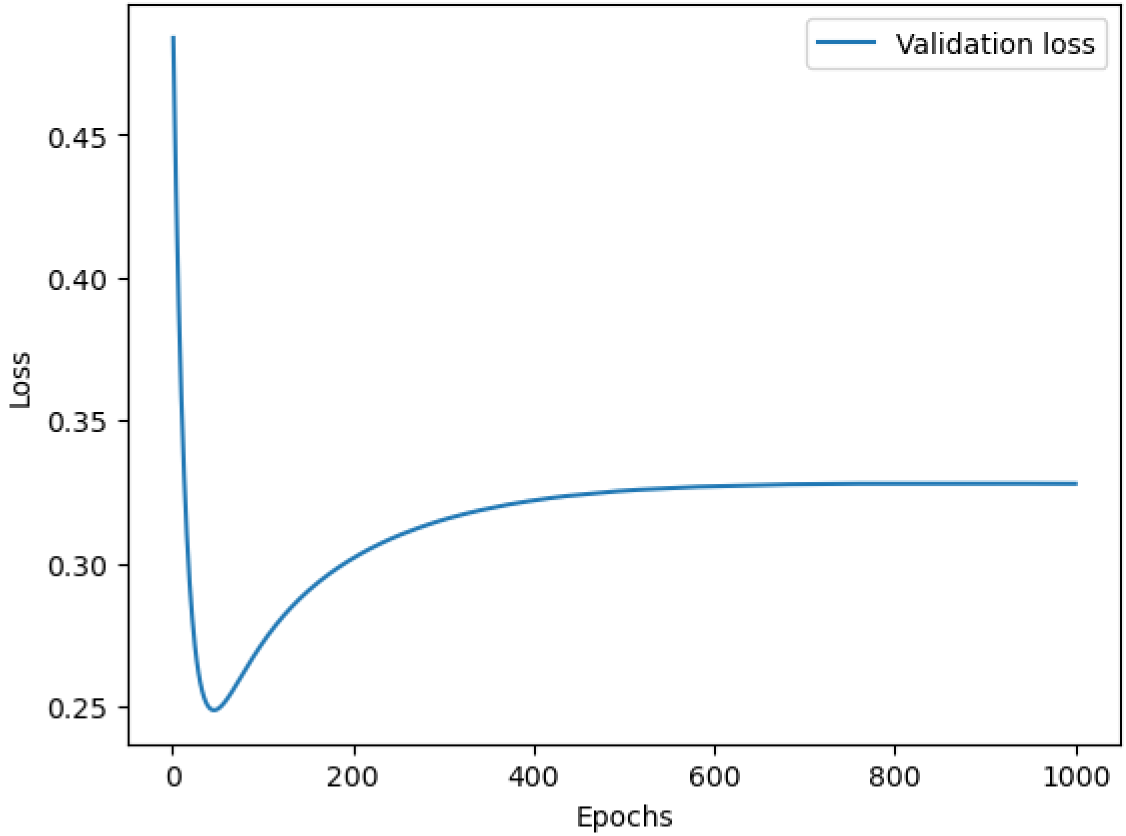
Validation loss
Loss
Epochs
0.450
0.425
0.400
0.375
0.350
0.325
0.300
0.275
0.250
0
200
400
600
800
1000



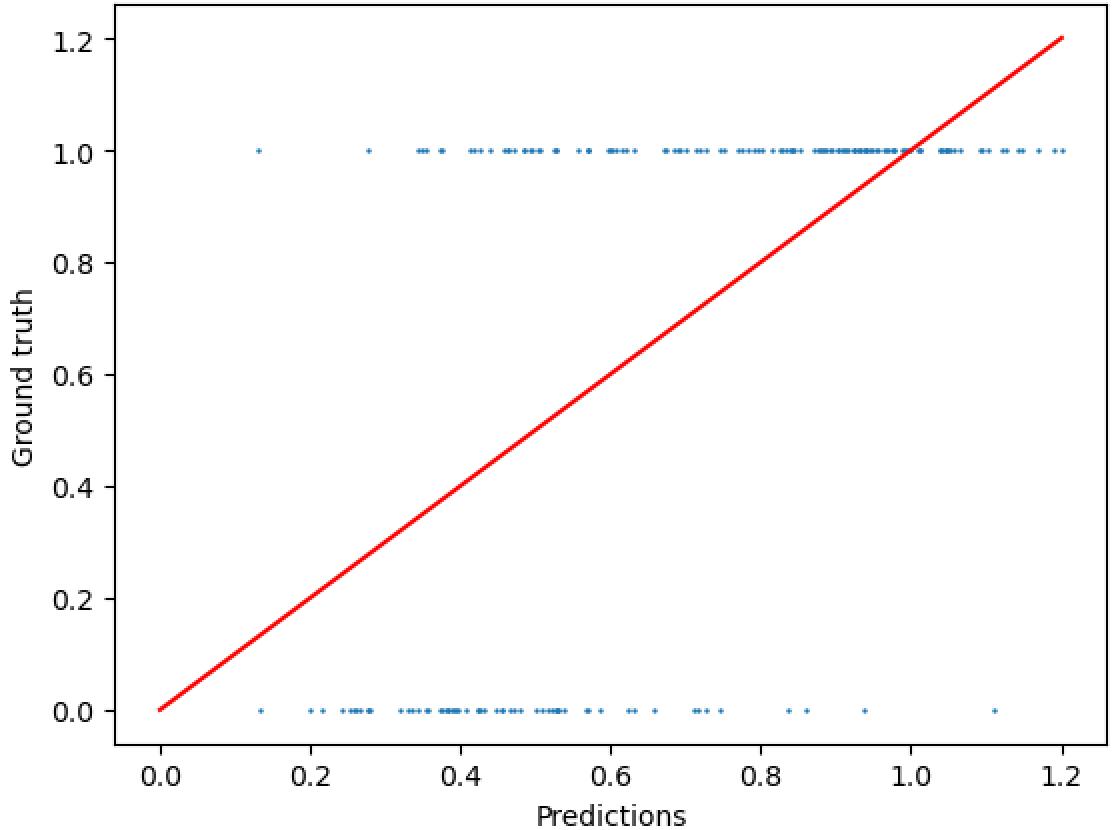
Validation loss
0.50
0.45
0.40
0.35
0.30
0.25
Loss
0
200
400
600
800
1000
Epochs



Validation loss
0.50
0.45
0.40
0.35
0.30
0.25
0.20
Loss
0
200
400
600
800
1000
Epochs



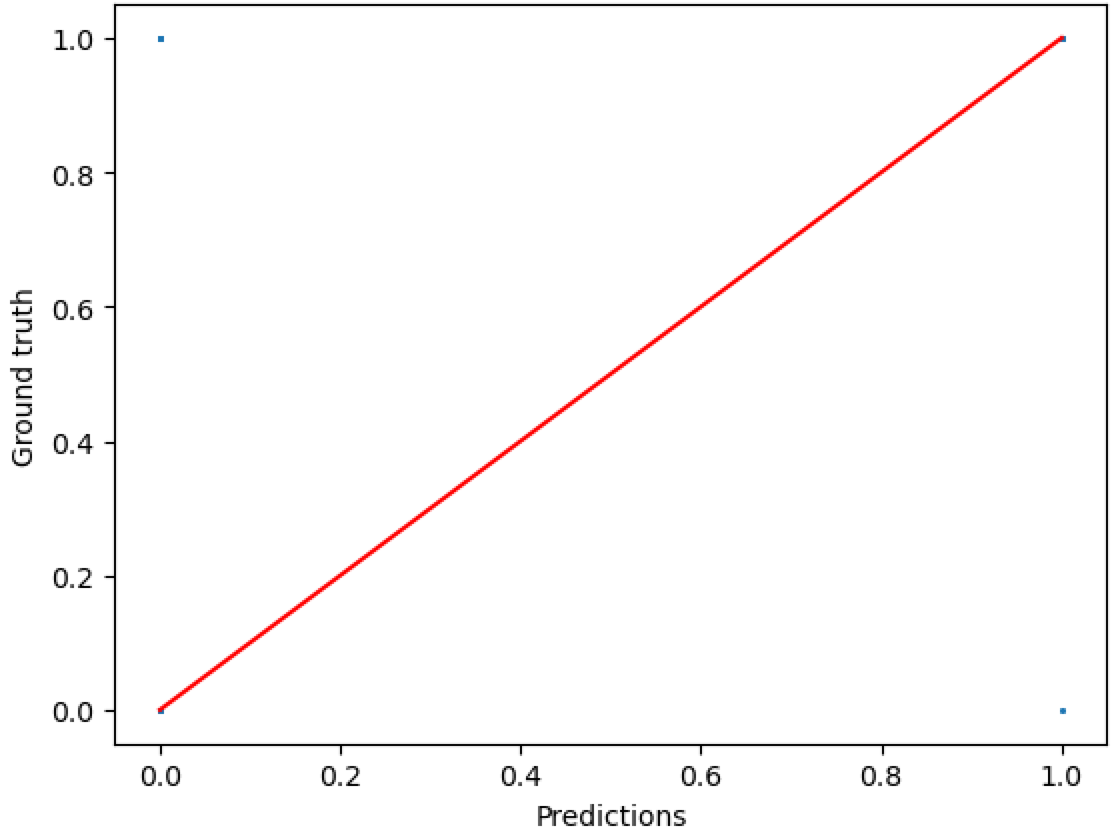
Validation loss
Loss
0.45
0.40
0.35
0.30
0.25
0
200
400
600
800
1000
Epochs



In [109…

```
show_correlation(titanic_filter_process['Sex_female'], survival.to_numpy())
```

```
Correlation coefficient: 0.5324179744538421
```



3. Nonlinear regression using a simple perceptron and a simple ANN.

(a) By using the Kfold algorithm developed in Q1, I was able to acquire a training set with very large errors, even though it sort of converged.

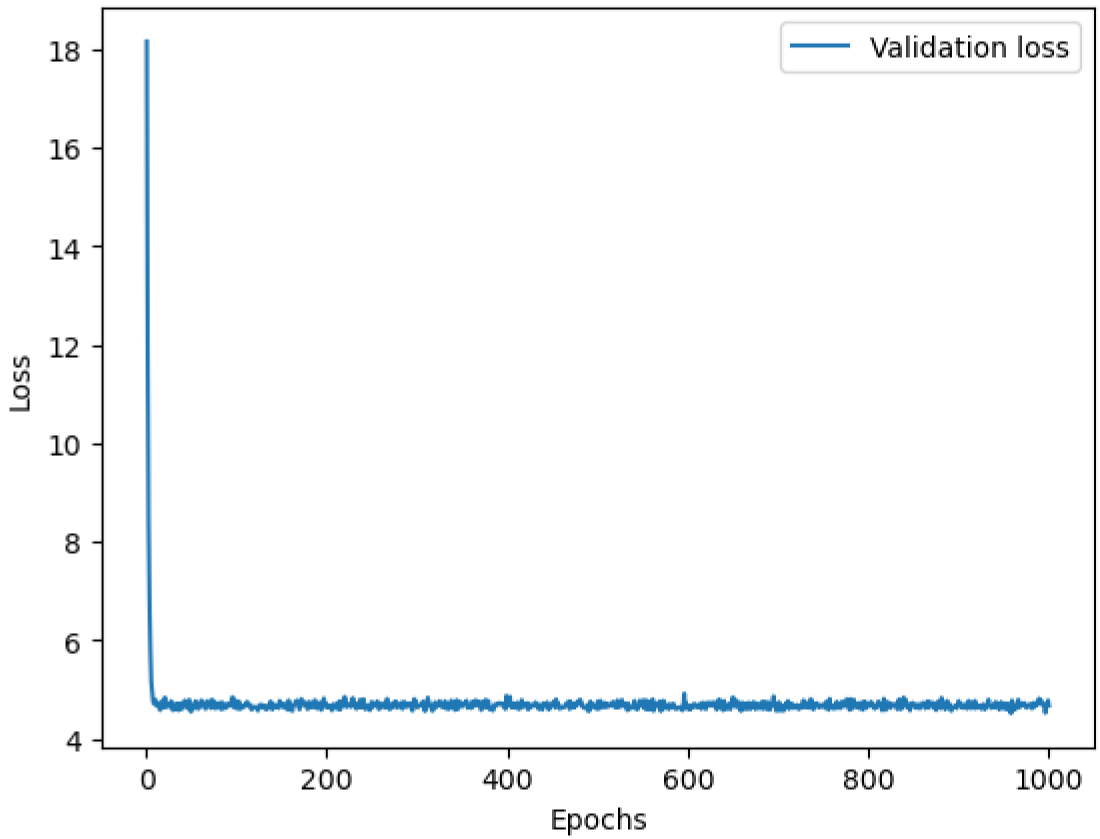
The correlation graph showed very poor correlation between y^hat and y values, with a coefficient of only 0.2.

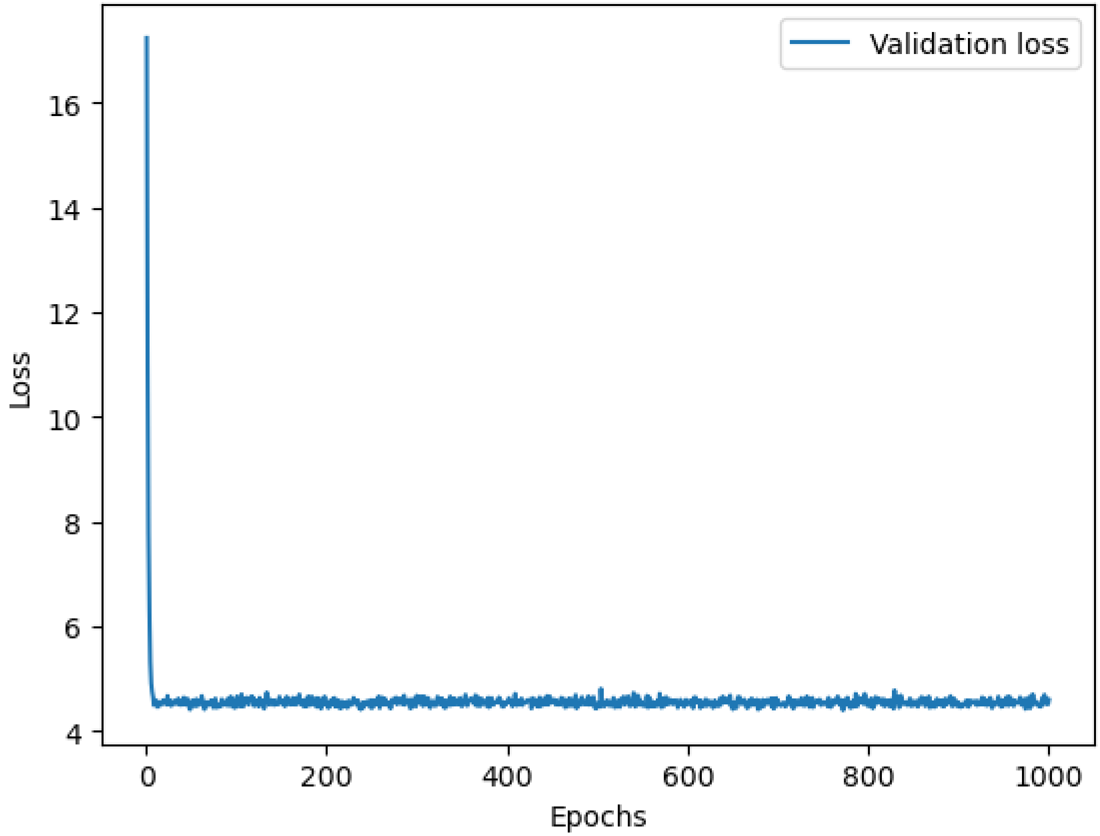
In [112…
```
def generate_X(number):
    xs=(np.random.random(number)*2-1)*10
    return xs

def generate_data(number,stochascity=0.05):
    xs=generate_X(number)
    fs=3*np.sin(xs)-5
    stochastic_ratio=(np.random.random(number)*2-1)*stochascity+1
    return xs,fs*stochastic_ratio
```

In [165…
```
x_train, y_train = generate_data(5000,0.1)
x_test, y_test = generate_data(1000,0.1)

x_train = x_train.reshape(-1,1)
```

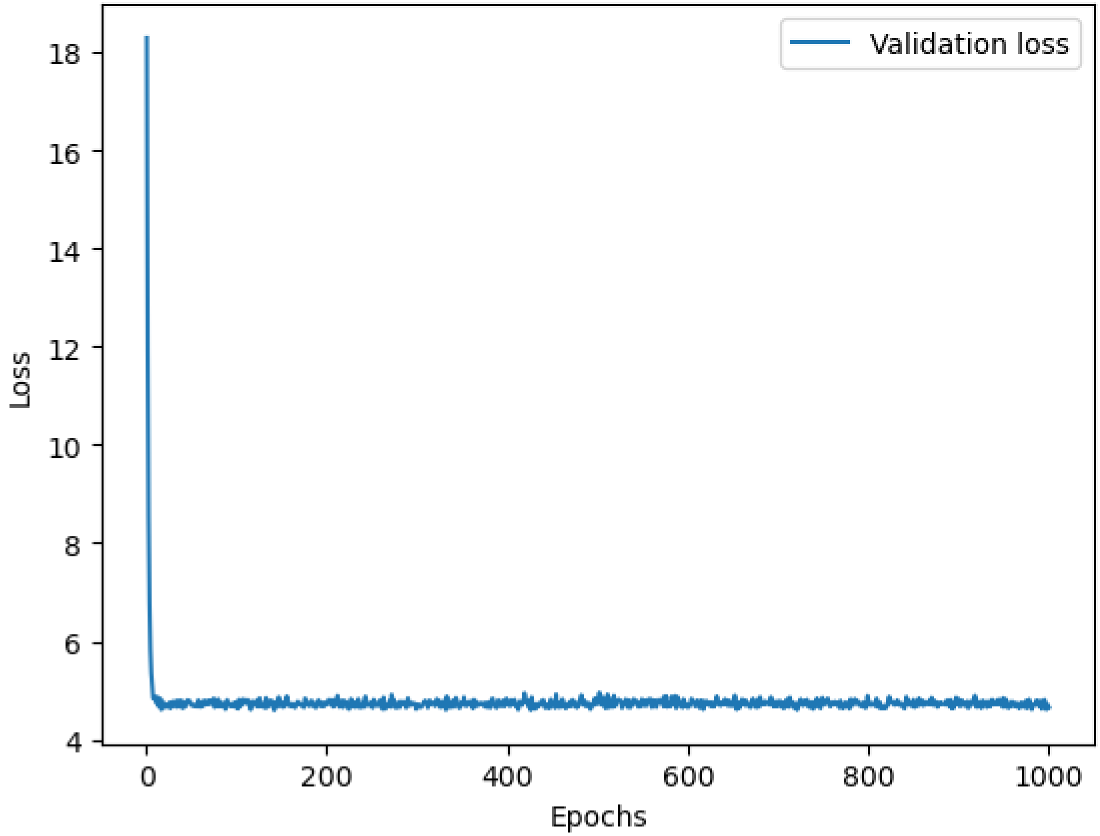
```python
prediction_3a_1, pred_3a = Kfold(5,x_train,y_train,epochs=1000,learning_rate=0.0001,draw_curve=True)
```

```
Number of epochs with lowest validation: 958
Train error: 4.645689289583773
Test error: 4.571847148330748
Number of epochs with lowest validation: 772
Train error: 4.676336261046478
Test error: 4.647797080822681
Number of epochs with lowest validation: 142
Train error: 4.534087989123831
Test error: 4.5332216090818
Number of epochs with lowest validation: 690
Train error: 4.6176076319842405
Test error: 4.594584636335373
Number of epochs with lowest validation: 141
Train error: 4.5785188142806765
Test error: 4.716145333157068
Final results:
Training error:4.610448+-0.049970
Testing error:4.612719+-0.063634
```

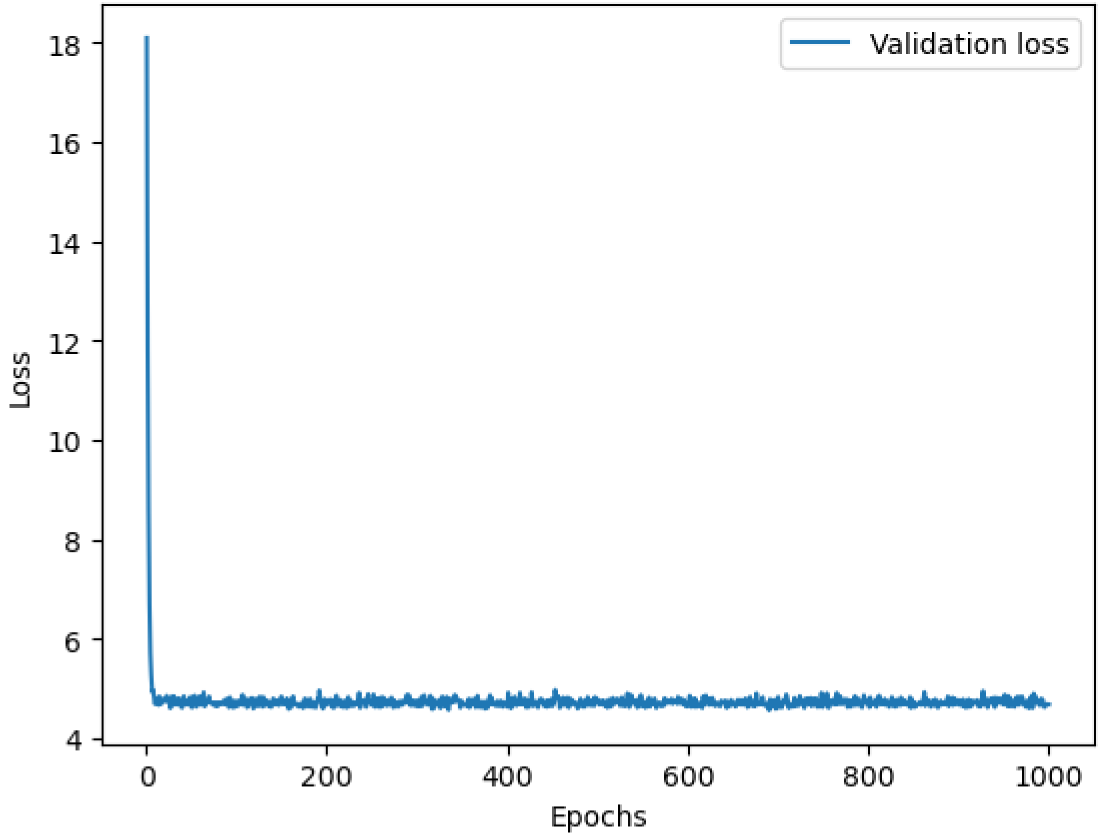


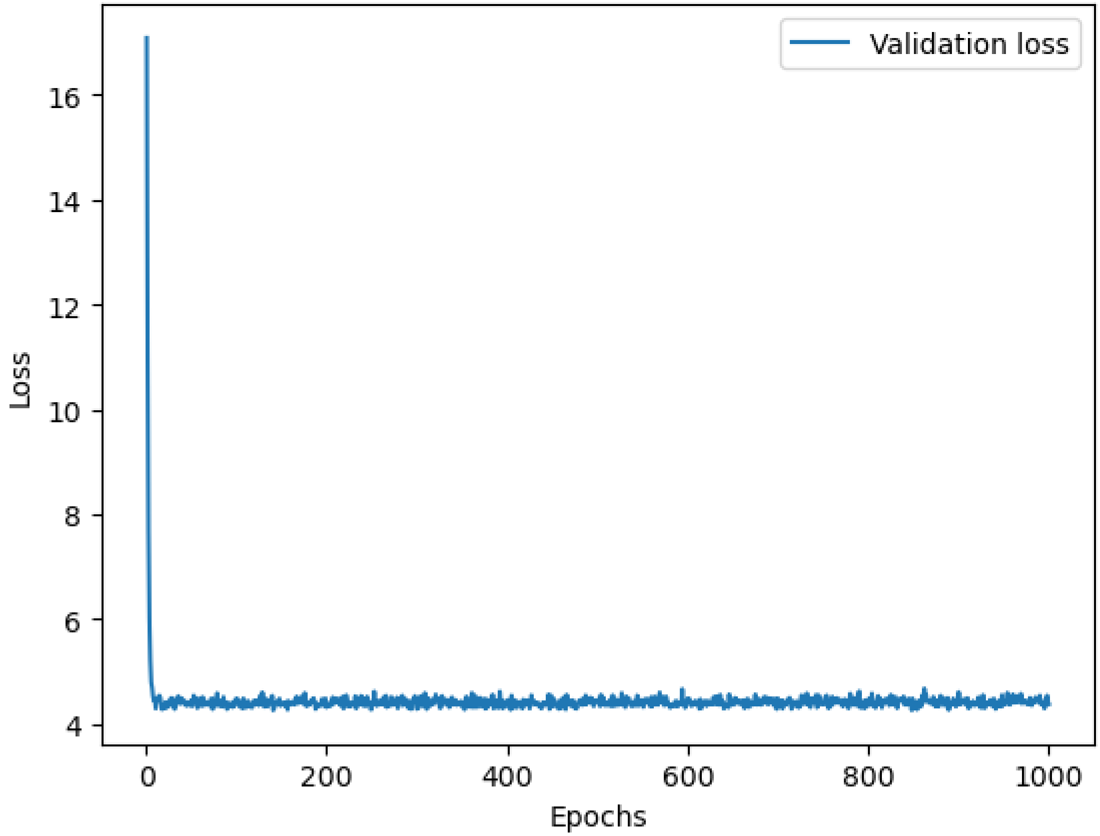


Validation loss
16
14
12
10
8
6
4
Loss
0
200
400
600
800
1000
Epochs



Validation loss
Loss
Epochs
18
16
14
12
10
8
6
4
0
200
400
600
800
1000

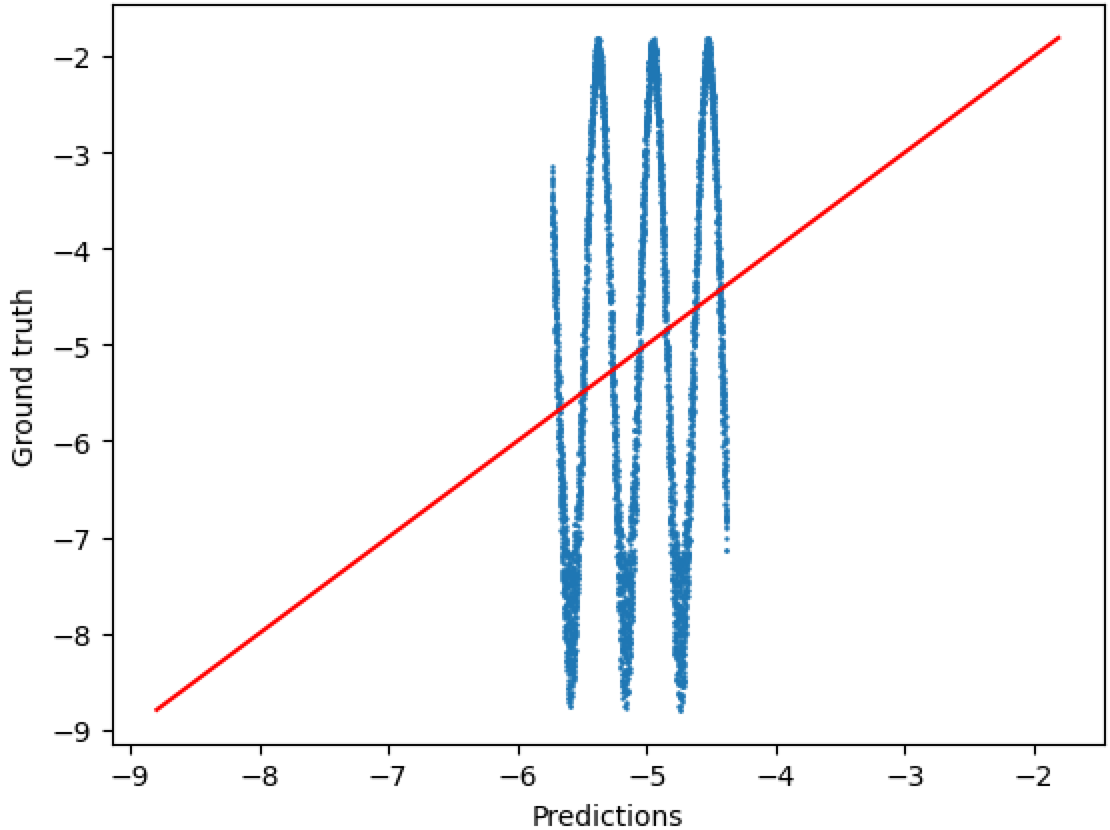

Validation loss
Loss
Epochs
18
16
14
12
10
8
6
4
0
200
400
600
800
1000



In [166…

```
show_correlation(pred_3a.reshape(-1), y_train.reshape(-1))
```

```
Correlation coefficient: 0.19540616619442897
```



(b) The KFold_NN codes are filled and presented below.

After the multi-layer perceptron training, the correlation improved from 0.2 to 0.3.

From the correlation image, the result also looks better.

In [167…

```
def KFold_NN(k,Xs,ys,hidden_layers,epochs=1000,lr=0.001,draw_curve=True):
    # The total number of examples for training the network
    total_num=len(Xs)

    # Built in K-fold function in Sci-Kit Learn
    kf=KFold(n_splits=k,shuffle=True)
    train_error_all=[]
    test_error_all=[]
    for train_selector,test_selector in kf.split(range(total_num)):
        # Decide training examples and testing examples for this fold
        train_Xs=Xs[train_selector]
        train_ys=ys[train_selector]
        test_Xs=Xs[test_selector]
        test_ys=ys[test_selector]
```

```
        # Establish the model here
        model = MLPRegressor(max_iter=epochs, activation='tanh', early_stopping=True,
                             validation_fraction=0.25, learning_rate='constant', learning_rate_init=lr,
                             hidden_layer_sizes=hidden_layers).fit(train_Xs, train_ys)

        ### Report result for this fold ##
        train_error = mean_squared_error(model.predict(train_Xs), train_ys)
        train_error_all.append(train_error)
        test_error = mean_squared_error(model.predict(test_Xs), test_ys)
        test_error_all.append(test_error)
        print("Train error:",train_error)
        print("Test error:",test_error)


    print("Final results:")
    print("Training error:%f+-%f"%(np.average(train_error_all),np.std(train_error_all)))
    print("Testing error:%f+-%f"%(np.average(test_error_all),np.std(test_error_all)))

    # return the last model
    return model
```

In [168…

```
x_train, y_train = generate_data(5000,0.1)
x_test, y_test = generate_data(1000,0.1)

x_train = x_train.reshape(-1,1)
y_train = y_train.reshape(-1)
prediction_3b_1 = KFold_NN(5,x_train,y_train,(8,),epochs=1000,lr=0.001)
```

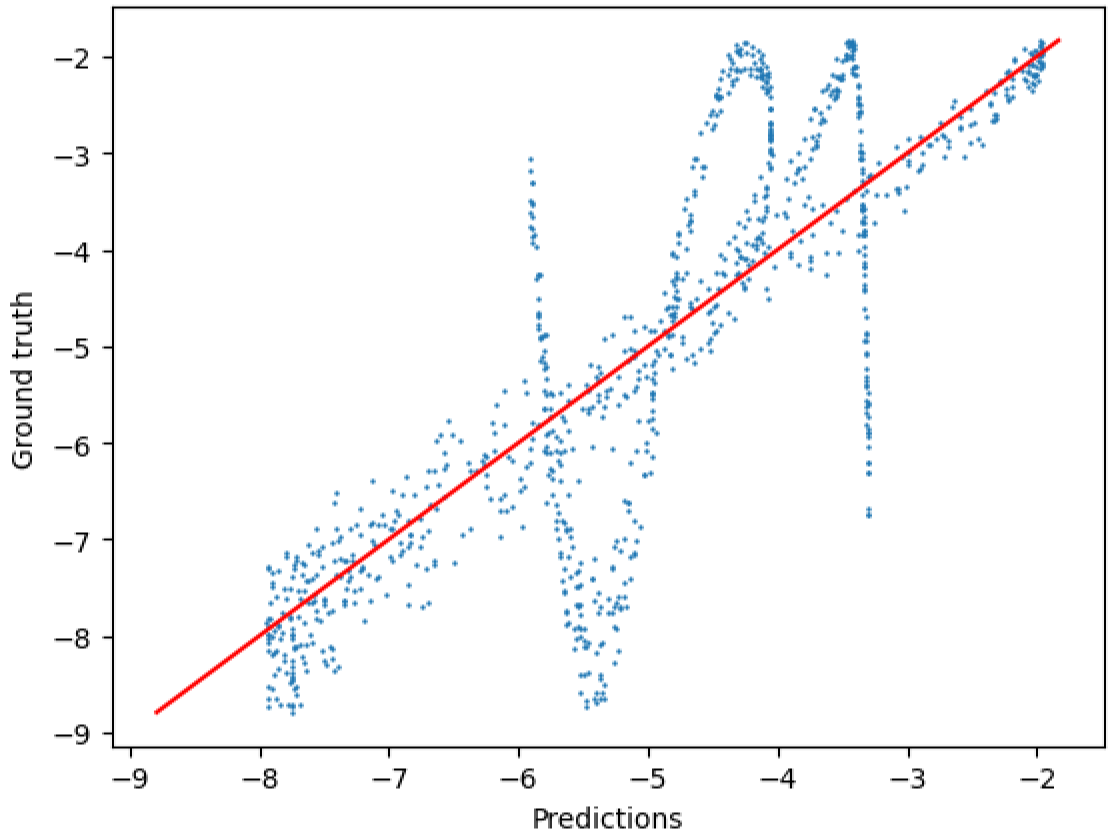
```
Train error: 3.4746710416673783
Test error: 3.5674557939772904
/opt/miniconda3/envs/qm-tools/lib/python3.10/site-packages/sklearn/neural_network/_multilayer_perceptron.py:684: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (1000) reached and the optimization hasn't converged yet.
  warnings.warn(
Train error: 0.4638955801650735
Test error: 0.4721935558431428
Train error: 2.442806899127254
Test error: 2.56751024037758
/opt/miniconda3/envs/qm-tools/lib/python3.10/site-packages/sklearn/neural_network/_multilayer_perceptron.py:684: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (1000) reached and the optimization hasn't converged yet.
  warnings.warn(
Train error: 3.0055132661496047
Test error: 2.9333740236695234
Train error: 1.377552352506833
Test error: 1.3868264402507384
Final results:
Training error:2.152888+-1.096538
Testing error:2.185472+-1.112506
```

```
/opt/miniconda3/envs/qm-tools/lib/python3.10/site-packages/sklearn/neural_network/_multilayer_perceptron.py:684: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (1000) reached and the optimization hasn't converged yet.
  warnings.warn(
```

In [169…

```
show_correlation(prediction_3b_1.predict(x_test.reshape(-1,1)).flatten(), y_test.reshape(-1).flatten())
```

```
Correlation coefficient: 0.8291204158676143
```



(c) After building a new cluster of hidden layers, the correlation significantly improved to > 0.9.

In [172…

```
x_train, y_train = generate_data(5000,0.1)
x_test, y_test = generate_data(1000,0.1)

x_train = x_train.reshape(-1,1)
y_train = y_train.reshape(-1)
prediction_3c_1 = KFold_NN(5,x_train,y_train,(5,5,2),epochs=1000,lr=0.001)
```

```
Train error: 2.378362748753247
Test error: 2.5391356085235866
Train error: 0.12706783321855275
Test error: 0.12028636716234348
Train error: 3.717991670570362
Test error: 3.939990781900907
Train error: 0.15190107891312032
Test error: 0.16958475141069057
Train error: 0.9577269144631534
Test error: 0.8084112803558121
Final results:
Training error:1.466610+-1.391532
Testing error:1.515482+-1.495598
```

In [173…

```
show_correlation(prediction_3c_1.predict(x_test.reshape(-1,1)), y_test.reshape(-1))
```

```
Correlation coefficient: 0.9084334515480132
```