In [38]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.preprocessing import StandardScaler
import random
import math
import warnings

warnings.filterwarnings("ignore")

%matplotlib inline
```

1a

In [39]:
```
import sklearn.model_selection as skl_model

admit_predict = pd.read_csv('Admission_Predict_Ver1.1.csv')

admit_predict.set_index('Serial No.', inplace= True)
admit_predict.head()

# LOR = letter of rec
# SOP = statement of purpose
```

Out[39]:

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

Features related to chance of admission are GRE score, TOEFL score, univesity rating, SOP. LOR, GCPA and Research.

In [26]:
```
rankings = admit_predict['Chance of Admit'].values

features = admit_predict.drop(columns='Chance of Admit').values
scaler = StandardScaler()
features_norm = scaler.fit_transform(features)
```

1b

In [27]:
```
def tanh(x):
    return np.tanh(x)

def tanh_grad(x):
    return 1-np.tanh(x)**2
```

In [40]:
```
class simple_perceptron():
    def __init__(self,input_dim,output_dim,learning_rate=0.01,activation=lam

        self.input_dim = input_dim
        self.output_dim = output_dim
        self.activation = activation
        # "The activation gradient is a measure of how sensitive the activat
        self.activation_grad = activation_grad
        self.lr = learning_rate
        ### initialize parameters ###
        self.weights = np.random.uniform(0, 1, (self.input_dim, self.output_
        self.biases = np.random.uniform(0, 0.05, self.output_dim)

    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 exp
        if not dim == self.input_dim:
            raise Exception("Expected input size %d, accepted %d!"%(self.inp
        ### Calculate logit and activation ###
        self.z = (X @ 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 conve
        if len(X.shape) == 1:
            X = X.reshape((-1,1))
        if len(y.shape) == 1:
            y = y.reshape((-1,1))
        self.predict(X)
        # subtracts true y values from predicted values (self.a)
        errors = (self.a - y) * self.activation_grad(self.z)
        # matric multiplication between transpose of errors and input data X
        weights_grad = errors.T.dot(X)
        # sums up errors along rows (axis 0)
        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.T


    def train_on_epoch(self,X,y,batch_size=32):
        # Every time select batch_size samples from the training set, until
        order = list(range(X.shape[0]))
        random.shuffle(order)
        n = 0
        while n < math.ceil(len(order)/batch_size)-1: # Parts that can fill
```

```
            self.fit(X[order[n*batch_size:(n+1)*batch_size]],y[order[n*batch
            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 = np.array(weights[0])
        self.biases = np.array(weights[1])
```

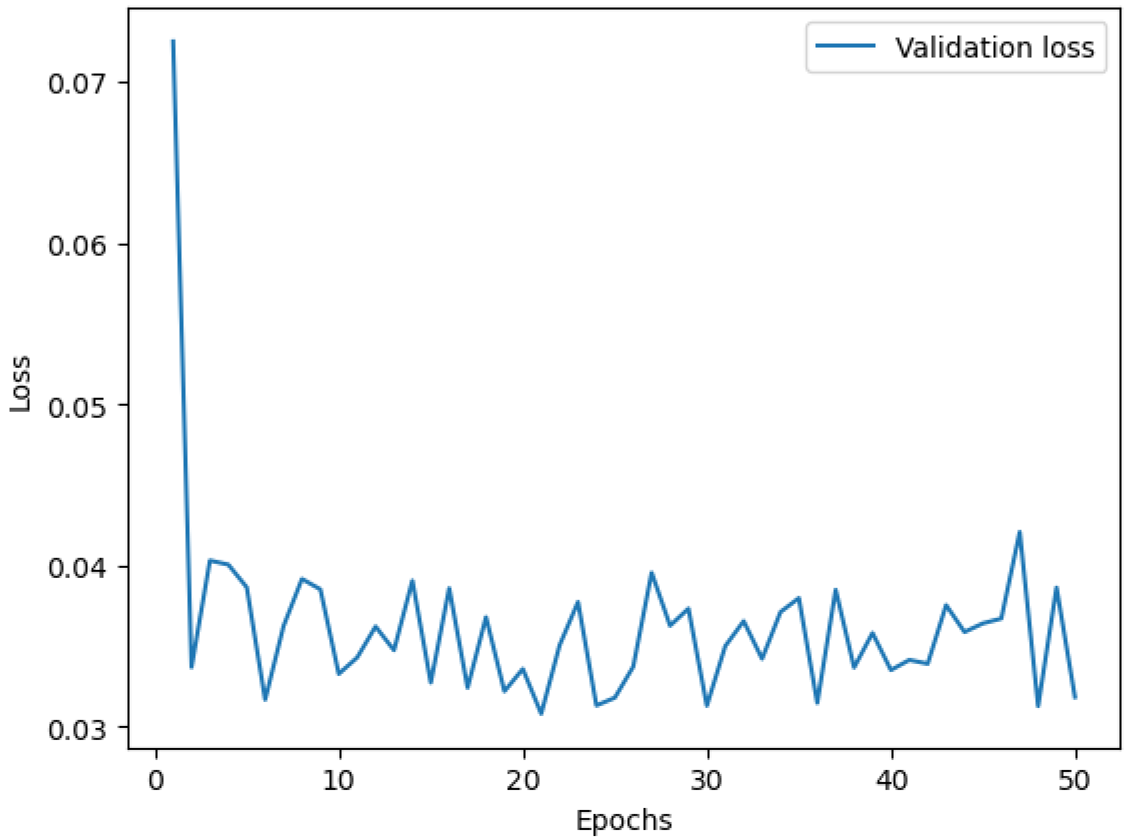
1c

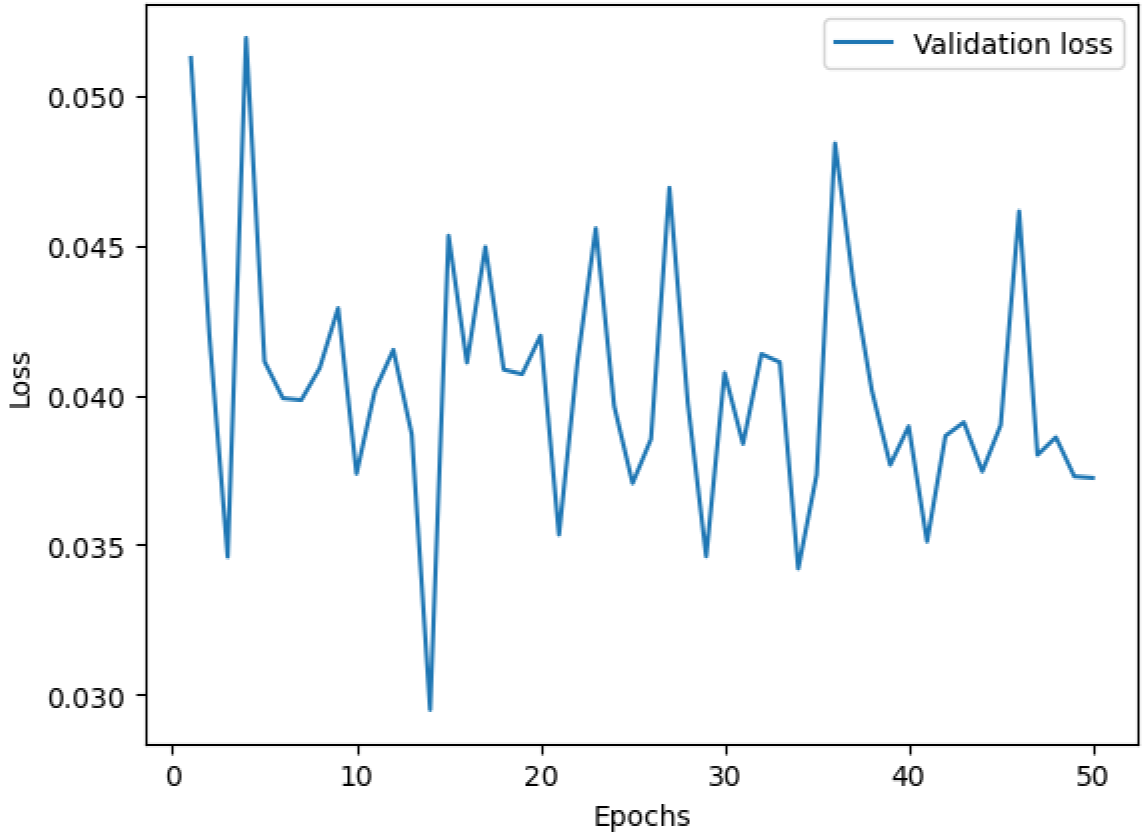
In [45]:
```
from sklearn.model_selection import train_test_split,KFold


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
        train_in, val_in, train_real, val_real = train_test_split(train_Xs,t

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

        # 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
```

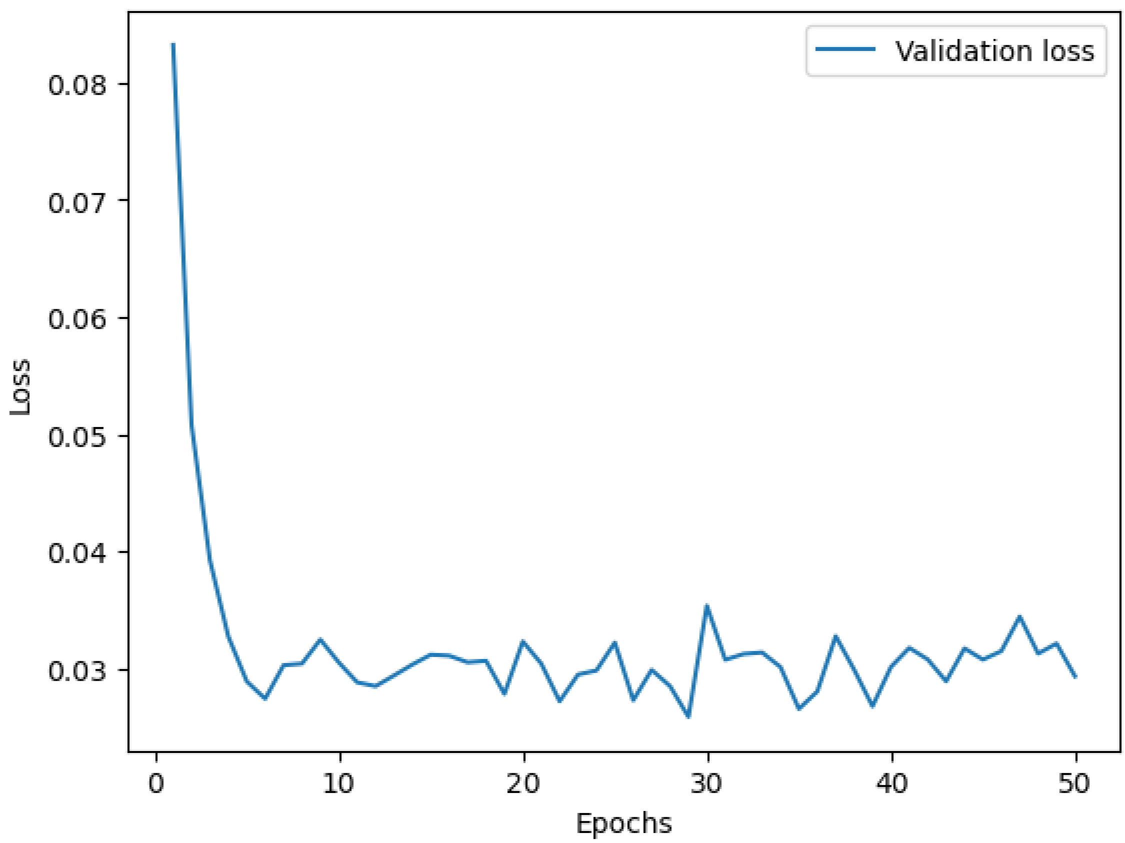
```
            model.train_on_epoch(train_in,train_real)
            val_err = model.evaluate(val_in,val_real)
            val_array.append(val_err)
            if val_err < lowest_val_err:
                lowest_val_err = val_err
                weights = model.get_weights()

        # The final number of epochs is when the minimum error in validatior
        final_epochs = val_array.index(min(val_array)) + 1 # +1 to consider
        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)

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

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

    def show_correlation(xs,ys):
        xs = model.predict(xs).reshape(-1,)
        ys = ys.reshape(-1)
        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)

    # return the last model
    return model
```

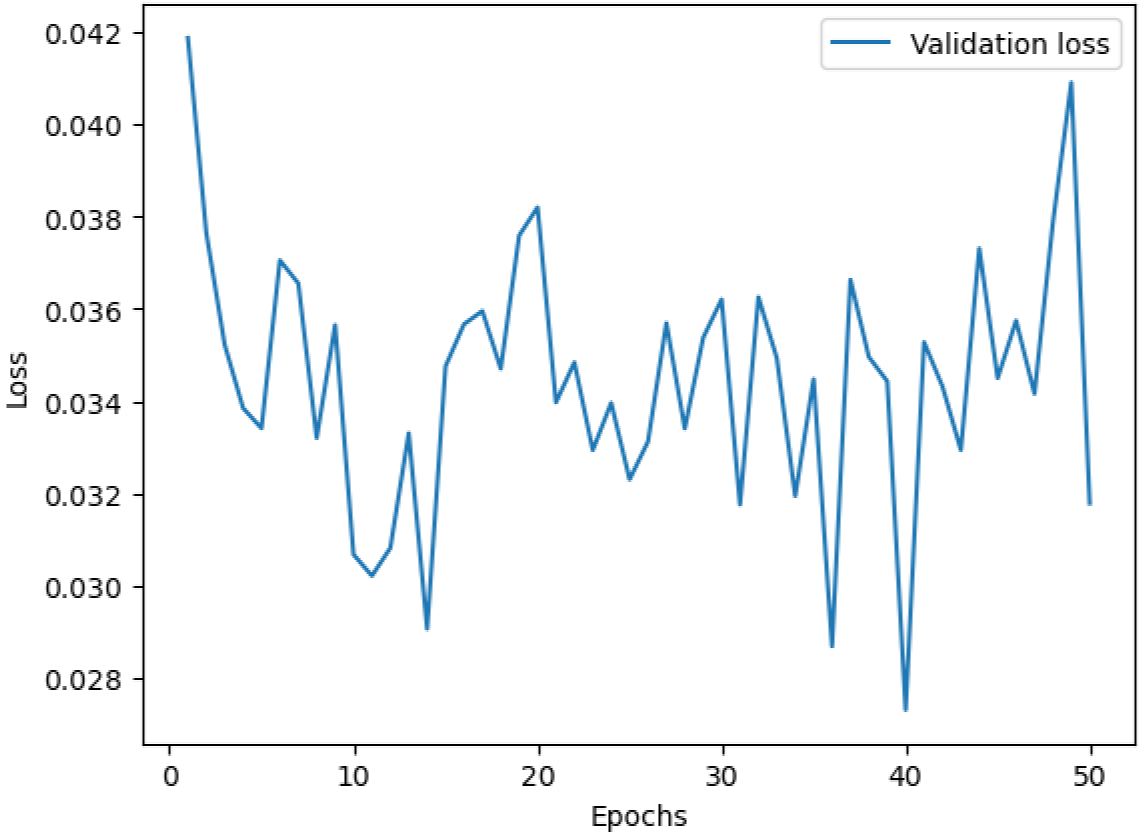
In [46]:
```
def tanh(x):
    return np.tanh(x)

def tanh_grad(x):
    return 1-np.tanh(x)**2

model = Kfold(5, features_norm, rankings, 50, learning_rate = 0.00001, draw_
print(model)
```

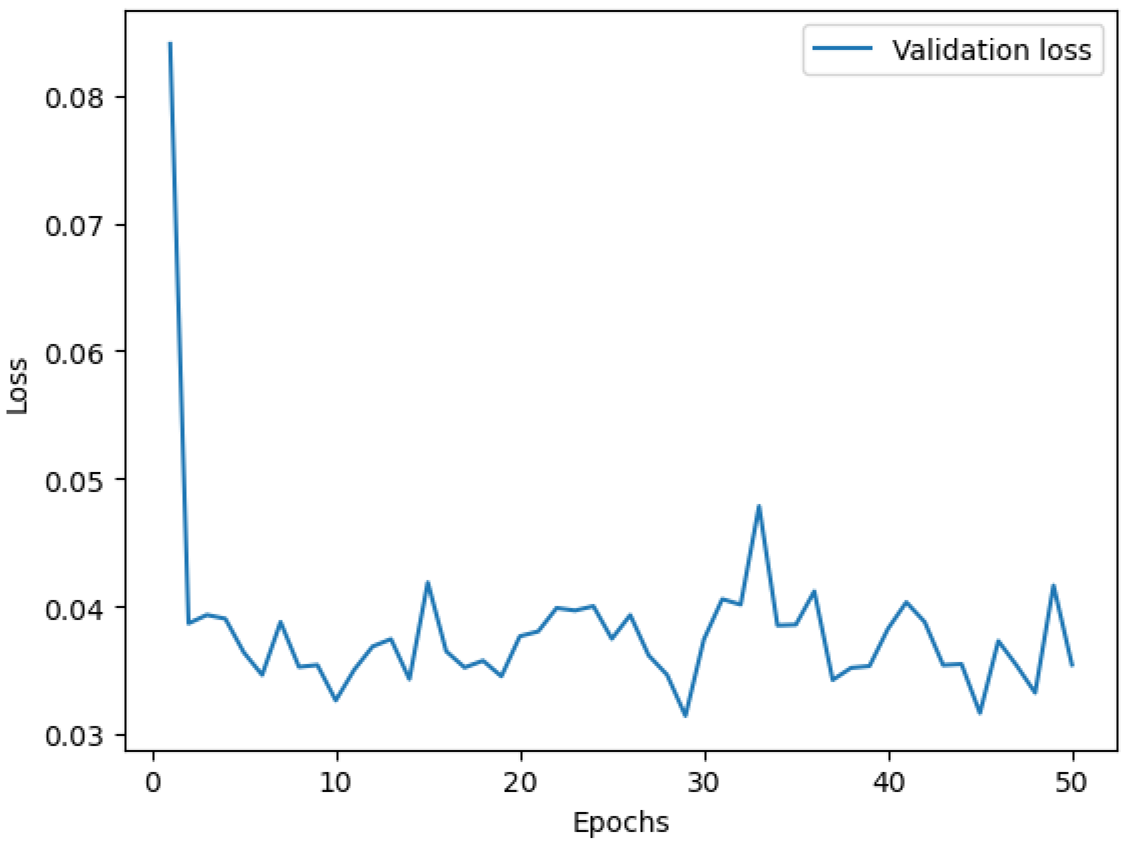
```
Number of epochs with lowest validation: 21
Train error: 0.03420285585867554
Test error: 0.02833616324911934
Number of epochs with lowest validation: 14
Train error: 0.03212957923900913
Test error: 0.03823247157429192
Number of epochs with lowest validation: 29
Train error: 0.03396143696543532
Test error: 0.03726124318468313
Number of epochs with lowest validation: 40
Train error: 0.03360303355729738
Test error: 0.03633624576843458
Number of epochs with lowest validation: 29
Train error: 0.035760189004682995
Test error: 0.027995262080891547
Final results:
Training error:0.033931+-0.001164
Testing error:0.033632+-0.004505
<__main__.simple_perceptron object at 0x13a7e6290>
```




Validation loss
0.050
0.045
0.040
0.035
0.030
Loss
0
10
20
30
40
50
Epochs


Validation loss
0.08
0.07
0.06
0.05
0.04
0.03
Loss
0
10
20
30
40
50
Epochs

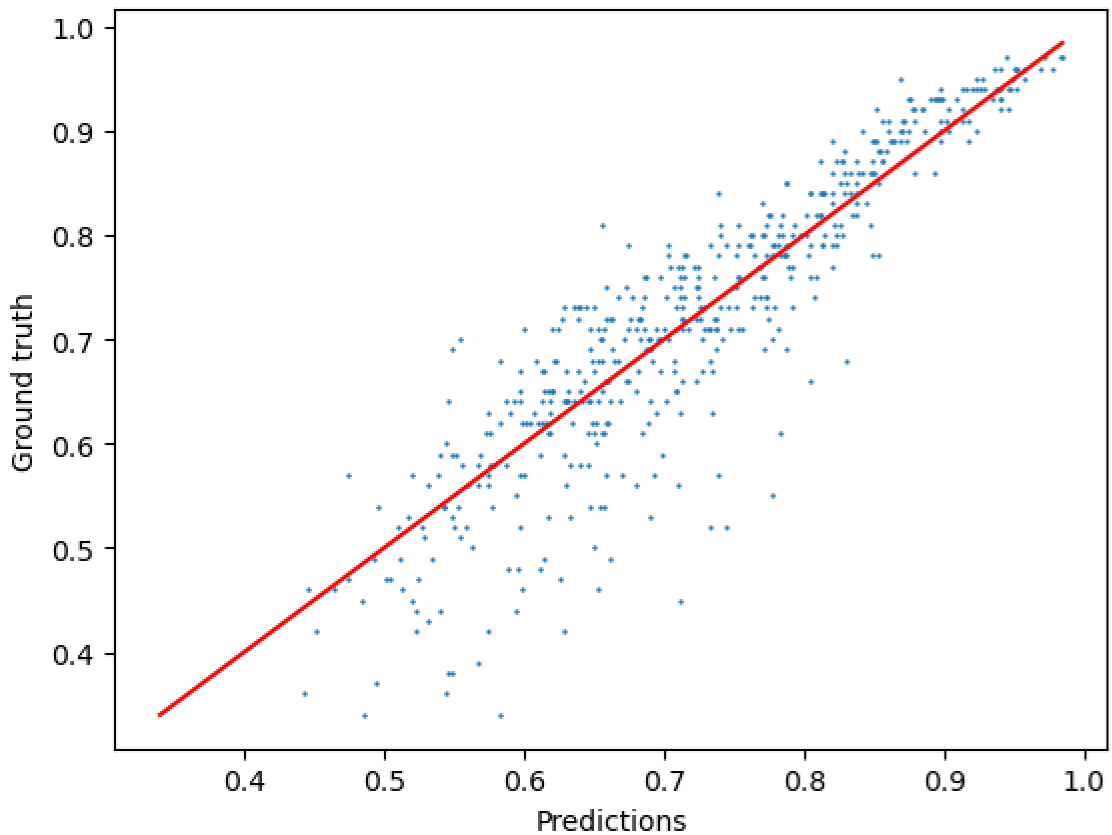




In [47]:
```
show_correlation(features_norm, rankings)
```

```
Correlation coefficient: 0.9057438961067454
```

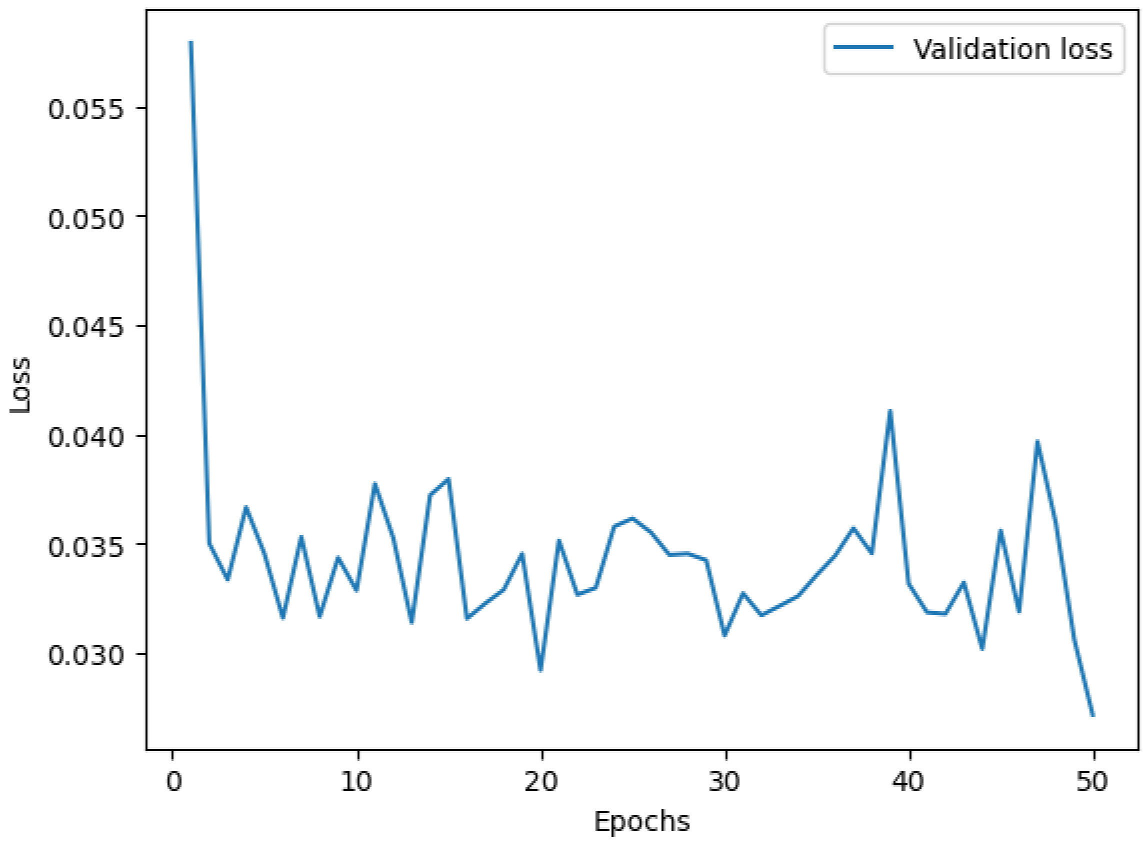
Out[47]: `<__main__.simple_perceptron at 0x13a7e6290>`

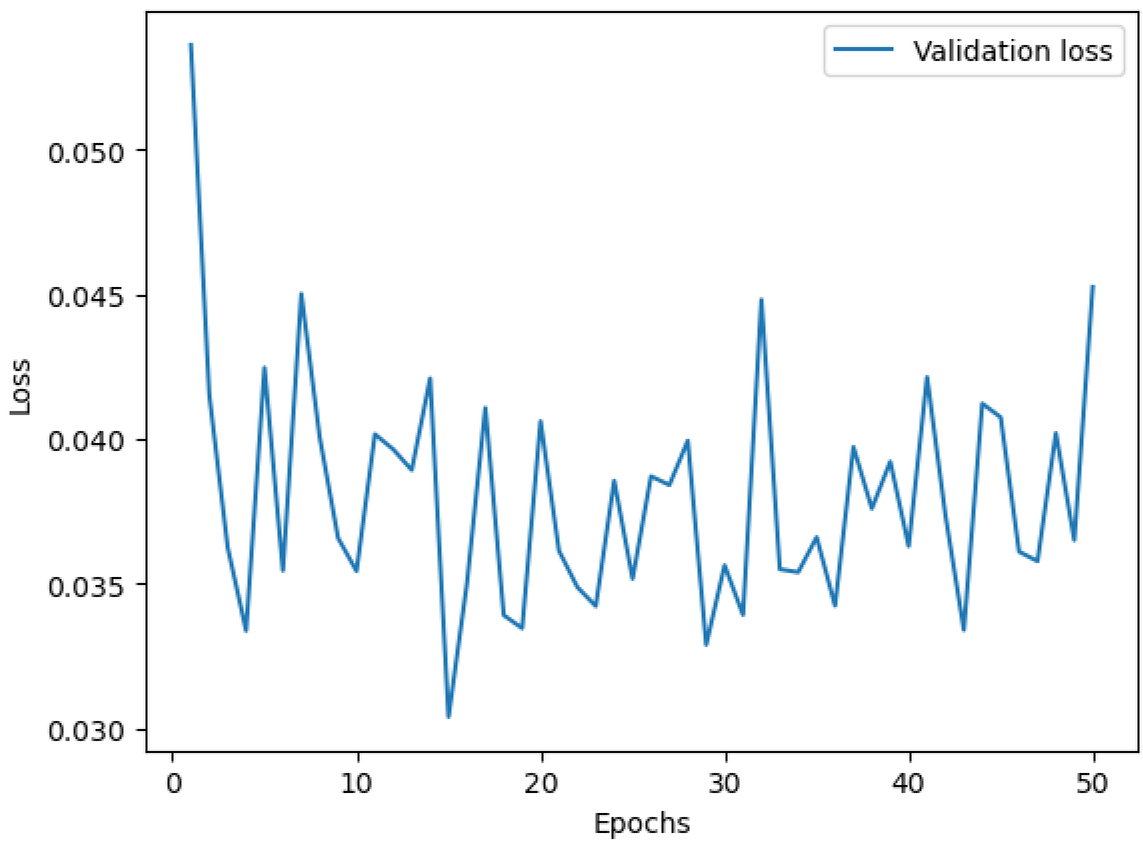


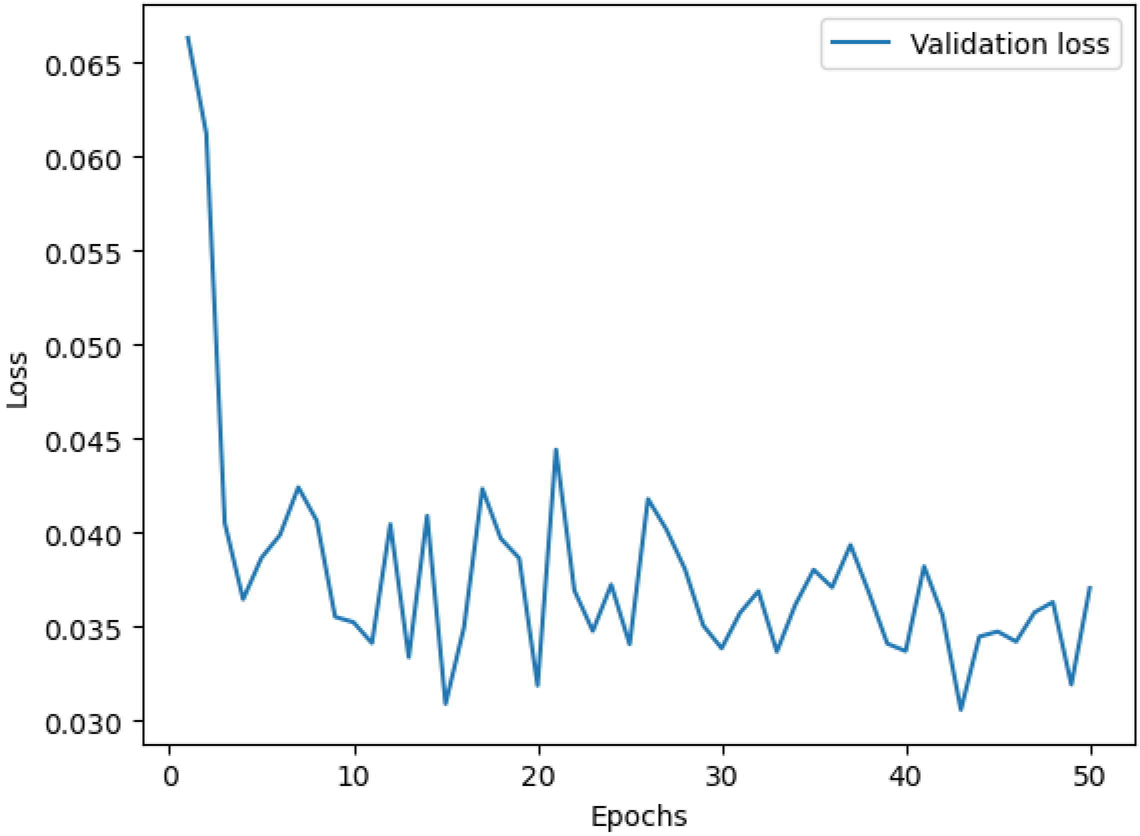
Yes, I think the features are a good indicator of getting into graduate school. Now we remove GRE Scores and test again.

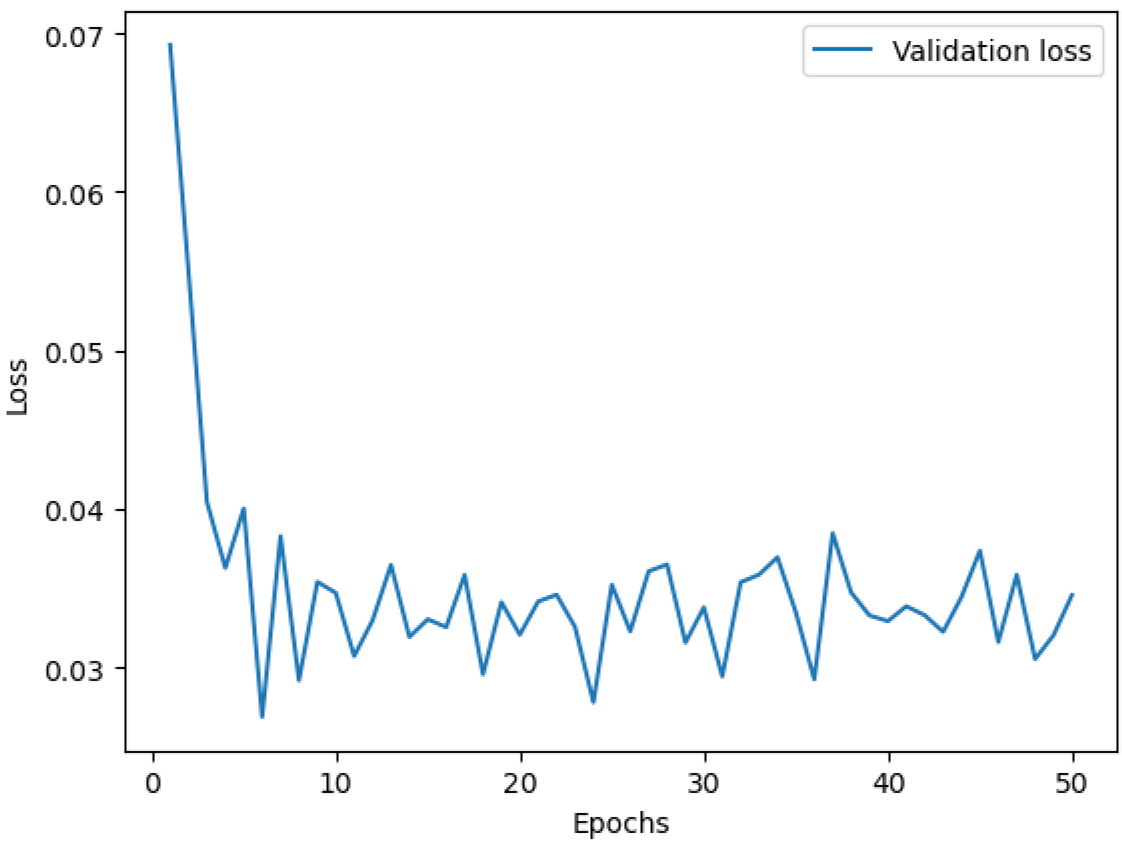
In [50]:
```
features_no_GRE = admit_predict.drop(columns=['Chance of Admit', 'GRE Score'
scaler = StandardScaler()
features_norm_no_GRE = scaler.fit_transform(features)

model2 = Kfold(5, features_norm_no_GRE, rankings, 50, learning_rate = 0.0000
print(model2)
```

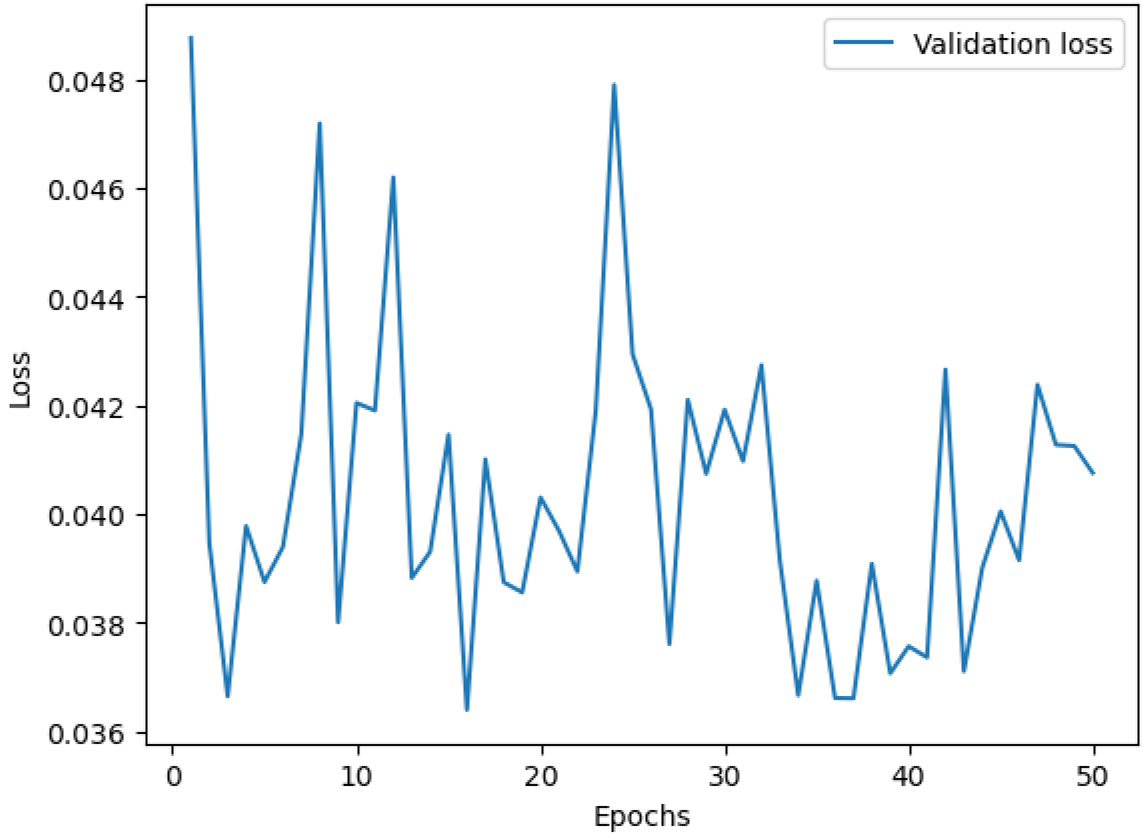
```
Number of epochs with lowest validation: 50
Train error: 0.029522150495362807
Test error: 0.02922318051672919
Number of epochs with lowest validation: 15
Train error: 0.044307049194561035
Test error: 0.0427414446593212
Number of epochs with lowest validation: 43
Train error: 0.03766941213285055
Test error: 0.034889763959733484
Number of epochs with lowest validation: 6
Train error: 0.03574767343904443
Test error: 0.039825314774456444
Number of epochs with lowest validation: 16
Train error: 0.03616130897572605
Test error: 0.03457722323799468
Final results:
Training error:0.036682+-0.004723
Testing error:0.036251+-0.004668
<__main__.simple_perceptron object at 0x13a28f5d0>
```


Validation loss
0.055
0.050
0.045
0.040
0.035
0.030
Loss
0
10
20
30
40
50
Epochs


Validation loss
0.050
0.045
0.040
0.035
0.030
Loss
0
10
20
30
40
50
Epochs


Validation loss
0.065
0.060
0.055
0.050
0.045
0.040
0.035
0.030
Loss
0
10
20
30
40
50
Epochs

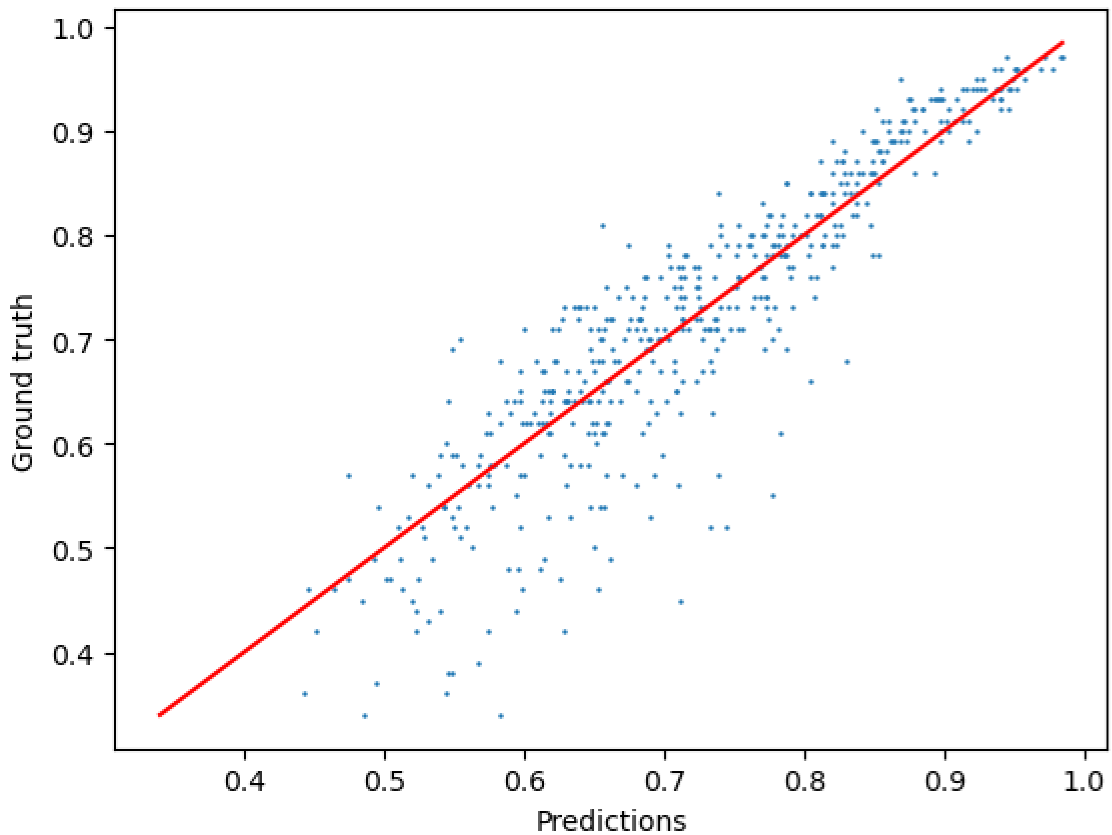

Validation loss
0.07
0.06
0.05
0.04
0.03
Loss
0
10
20
30
40
50
Epochs



In [51]:
```
show_correlation(features_norm_no_GRE, rankings)
```

```
Correlation coefficient: 0.9057438961067454
```

Out[51]:
```
<__main__.simple_perceptron at 0x13a7e6290>
```



I guess GRE Scores are not that important; the correlation graphs look more or less identical

2a

This dataset has a lot more categorical data than the previous one, which just contained numbers and binary values. There were more steps to prepare and separate the features and rankings.

In [52]:
```
titanic = pd.read_csv('titantic.csv')

titanic.set_index('PassengerId', inplace= True)
titanic = titanic.dropna()

titanic_numericals = titanic.drop(['Ticket', 'Name', 'Sex'], axis=1)
```

In [76]:
```
from sklearn.preprocessing import OneHotEncoder

# normalize numerical data
normalized_numerical = titanic_numericals.iloc[:, 1:6].apply(lambda x: (x -

# one hot encode categorical data
encoder = OneHotEncoder(handle_unknown='ignore')
encoded_categoricals = encoder.fit_transform(titanic[['Sex', 'Embarked']]).t
encoded_categoricals_df = pd.DataFrame(encoded_categoricals)

titanic_transformed = pd.concat([normalized_numerical, encoded_categoricals_
```

```
titanic_transformed = titanic_transformed.dropna()

#titanic_transformed.head()
```

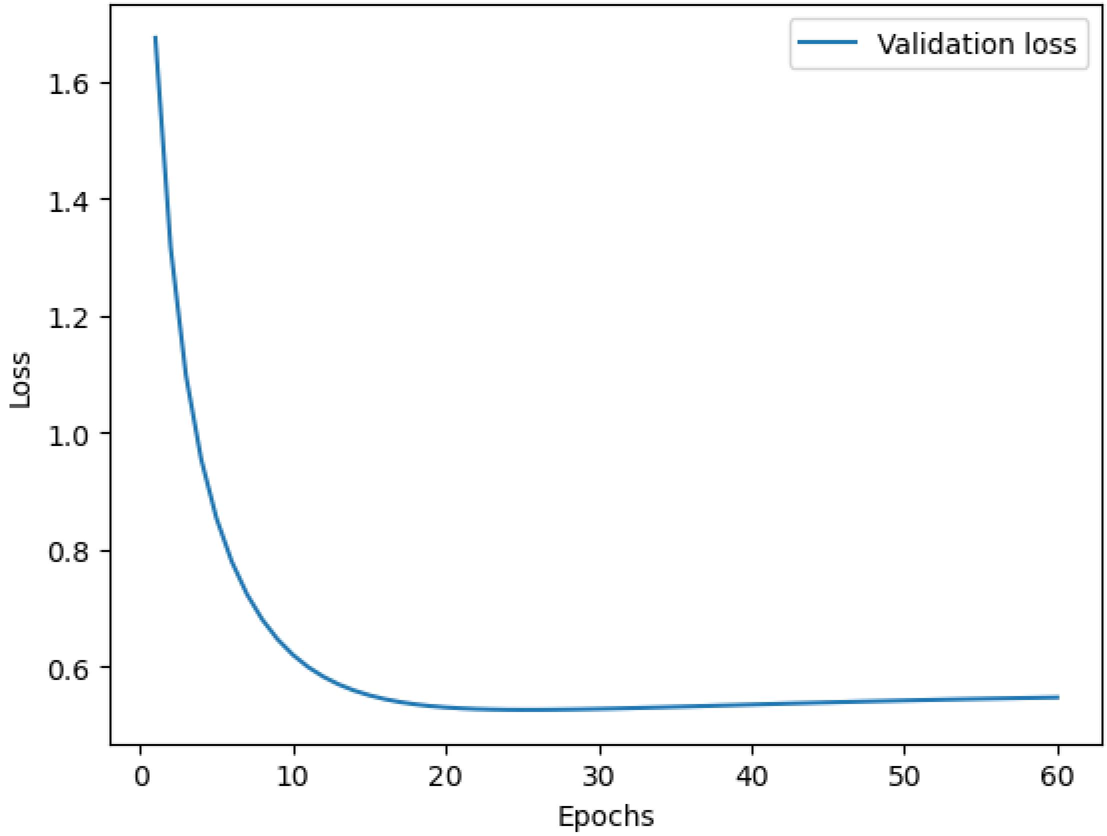
2b

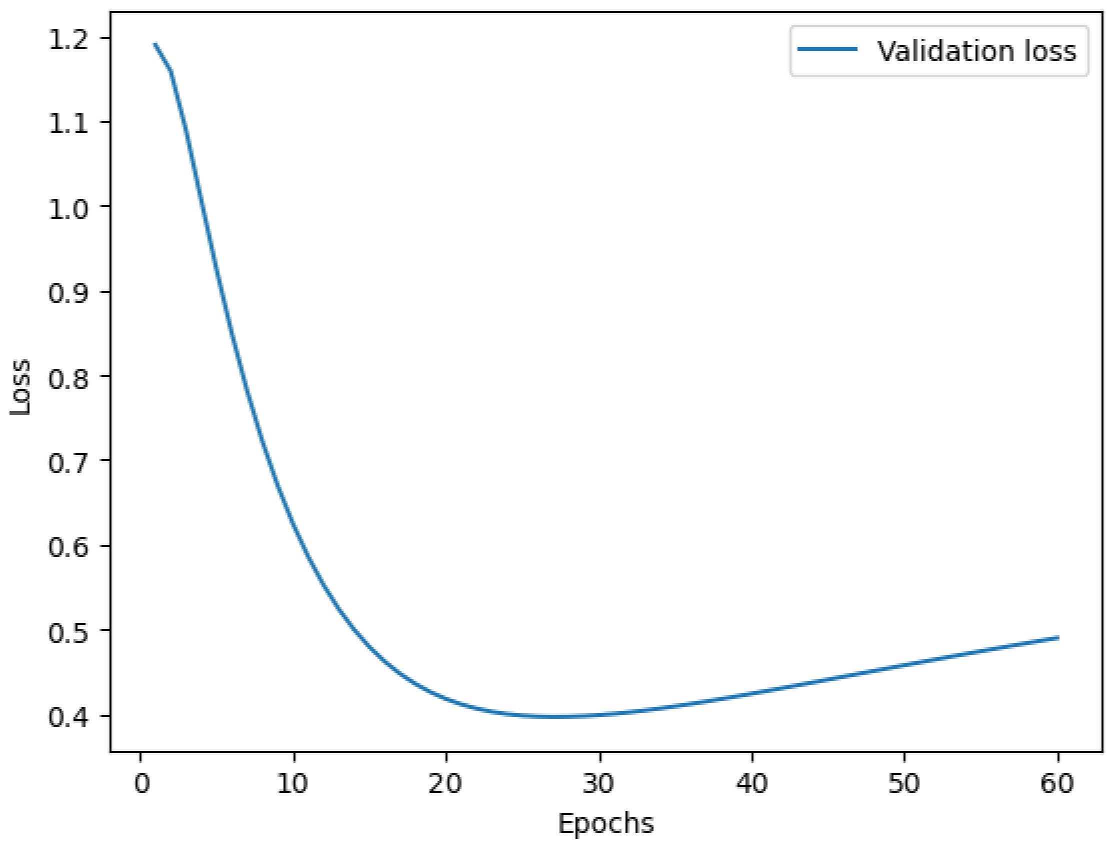
Use the simple perceptron model we developed in Q1. Use 80% of the data for training and 20% of the data for testing and do 5-fold validation. Can we predict who will survive? Play around with the features to determine which ones give you a better chance to get back to shore

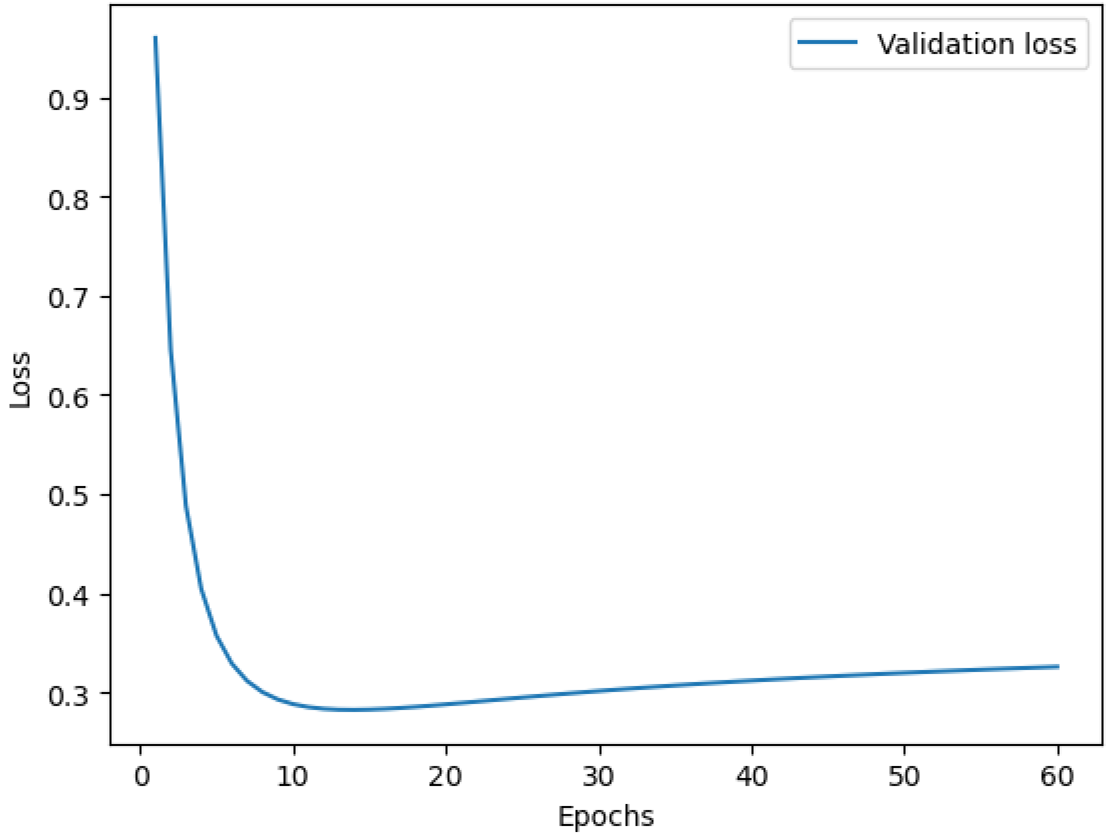
In [83]:
```
features = titanic_transformed.values
predictions = titanic['Survived'].values

Kfold(5, features, predictions, 60)
```

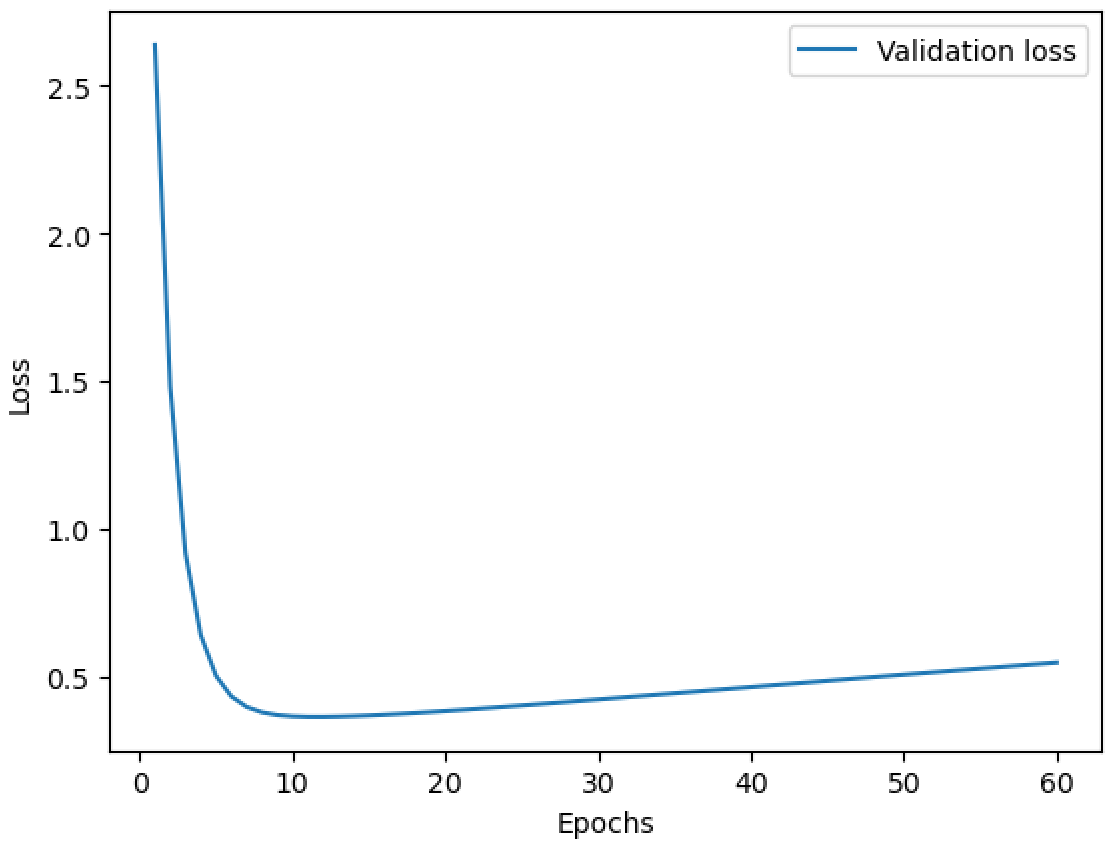
```
Number of epochs with lowest validation: 25
Train error: 0.39564764868540236
Test error: 0.31329868217727747
Number of epochs with lowest validation: 27
Train error: 0.3679870596334657
Test error: 0.4782723493771518
Number of epochs with lowest validation: 14
Train error: 0.34261610818235694
Test error: 0.40184007160530455
Number of epochs with lowest validation: 12
Train error: 0.3887770023802423
Test error: 0.366983306429162
Number of epochs with lowest validation: 3
Train error: 0.37021299484898945
Test error: 0.35849825710336575
Final results:
Training error:0.373048+-0.018536
Testing error:0.383779+-0.055034
```

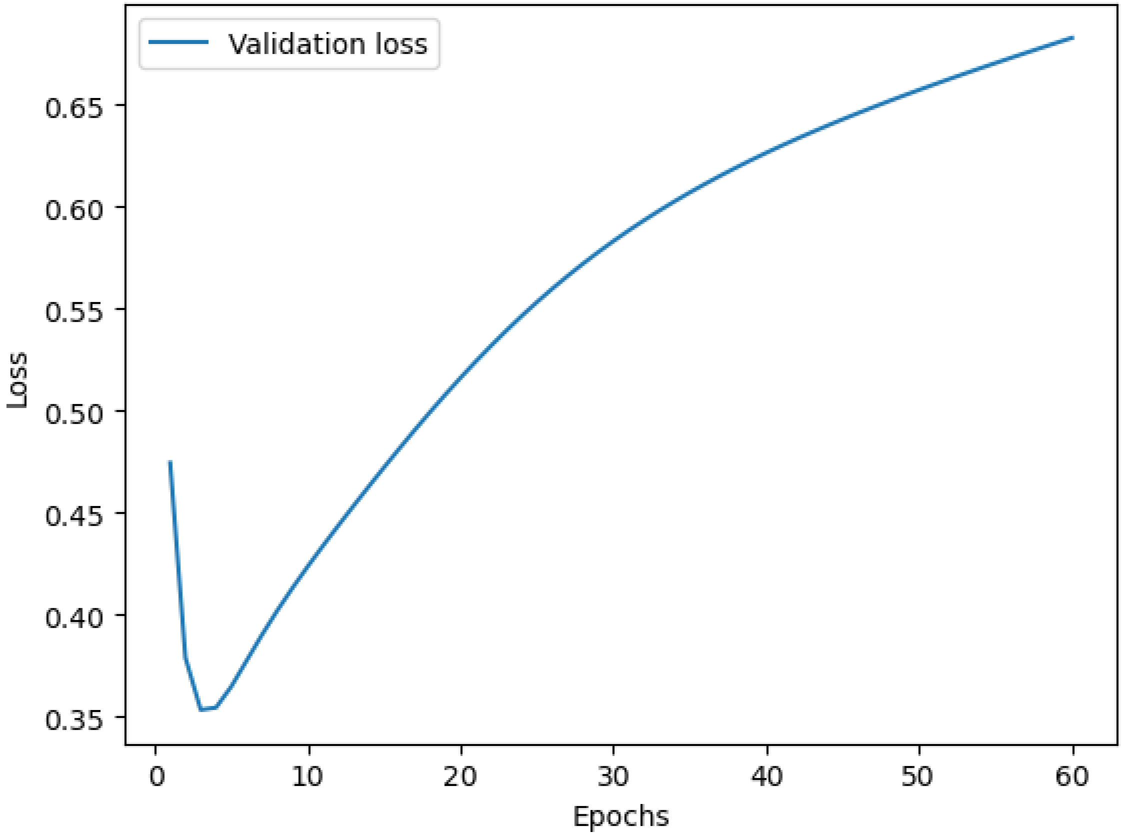
Out[83]:
```
<__main__.simple_perceptron at 0x13a4bab90>
```


Validation loss
1.6
1.4
1.2
1.0
0.8
0.6
Loss
0
10
20
30
40
50
60
Epochs


Validation loss
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
Loss
0
10
20
30
40
50
60
Epochs


Validation loss
Loss
Epochs
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
10
20
30
40
50
60


Validation loss
Loss
Epochs
0.5
1.0
1.5
2.0
2.5
0
10
20
30
40
50
60



In [ ]:
```
show_correlation(features, predictions)
```

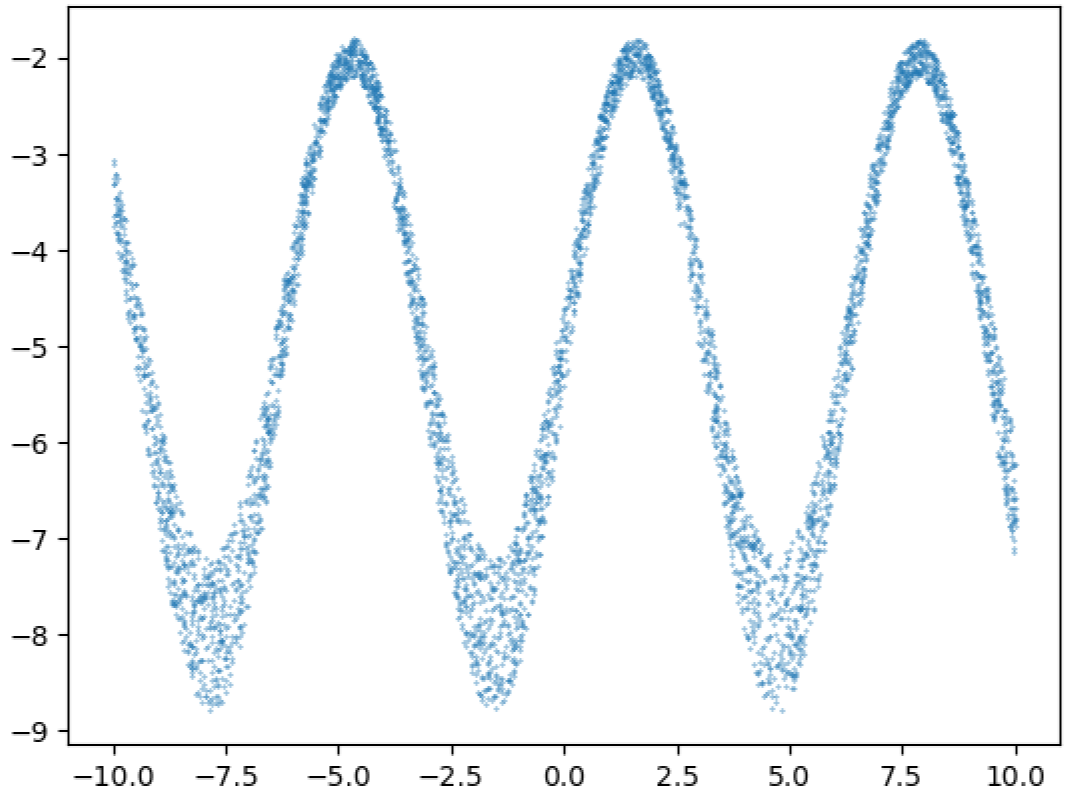
Yes, we can fairly accurately predict who will survive. Although I could not get my correlation graph to work, I say this based on the loss function. The largest determinant for survival was Sex, as it had the largest weight.

3a

In [64]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

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 [65]:
```
x, y = generate_data(5000, 0.1)
plt.scatter(x, y, s=0.1)
```

Out[65]: <matplotlib.collections.PathCollection at 0x13a4d2810>



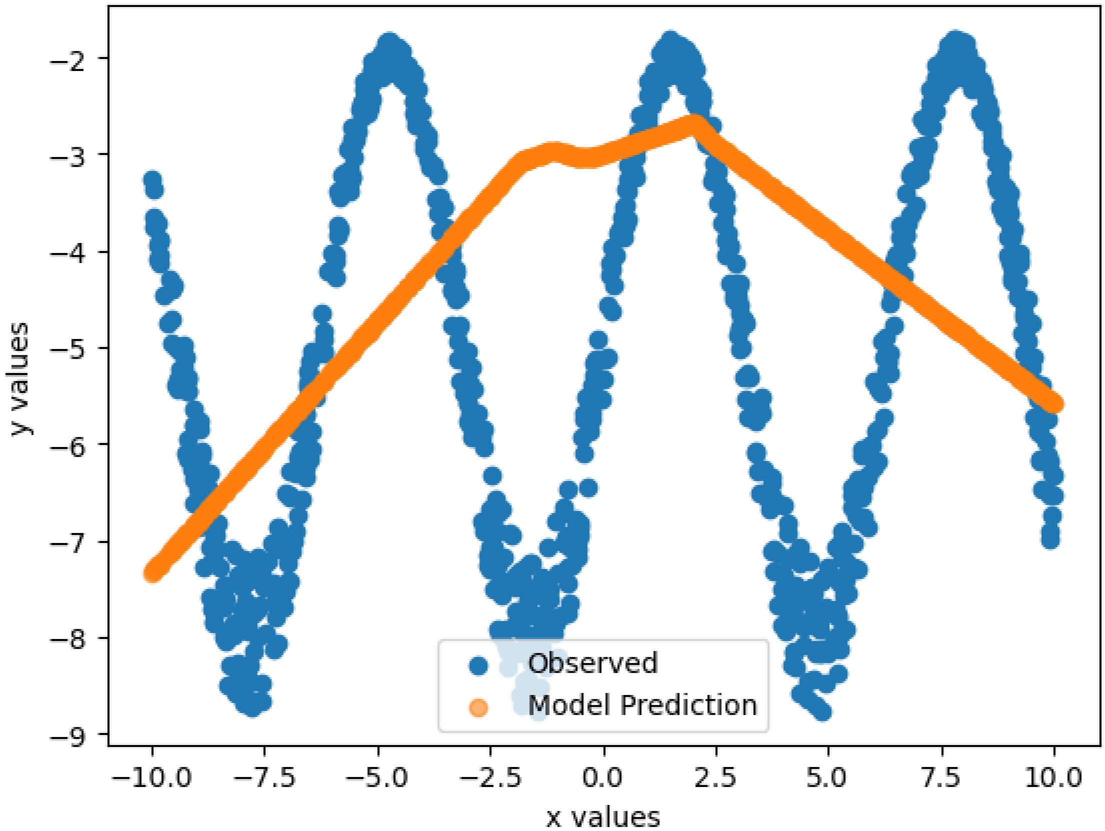
In [84]:
```
x, y = generate_data = (1000, 0.1)
```

3b

In [66]:
```
#from sklearn.neural_network import MLPRegressor
#from sklearn.model_selection import train_test_split, KFold

import numpy as np
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error

x, y = generate_data(1000, 0.1)

MLP = MLPRegressor(hidden_layer_sizes=(8,), random_state = 49)

mean_squared_error = -cross_val_score(MLP, x.reshape(-1, 1), y, cv = 5, scor

for fold, mse in enumerate(mean_squared_error):
    print(f'Fold {fold + 1}: MSE = {mse}')

MLP.fit(x.reshape(-1, 1), y)
y_pred = MLP.predict(x.reshape(-1, 1))

plt.figure()
plt.scatter(x, y, label='Observed')
plt.scatter(x, y_pred, label='Model Prediction', alpha=0.6)
plt.xlabel('x values')
plt.ylabel('y values')
```

```
plt.legend()
plt.show()
```

```
Fold 1: MSE = 6.557191509323998
Fold 2: MSE = 5.759326237994227
Fold 3: MSE = 6.967023793409838
Fold 4: MSE = 6.604969312598687
Fold 5: MSE = 6.710412821254925
```



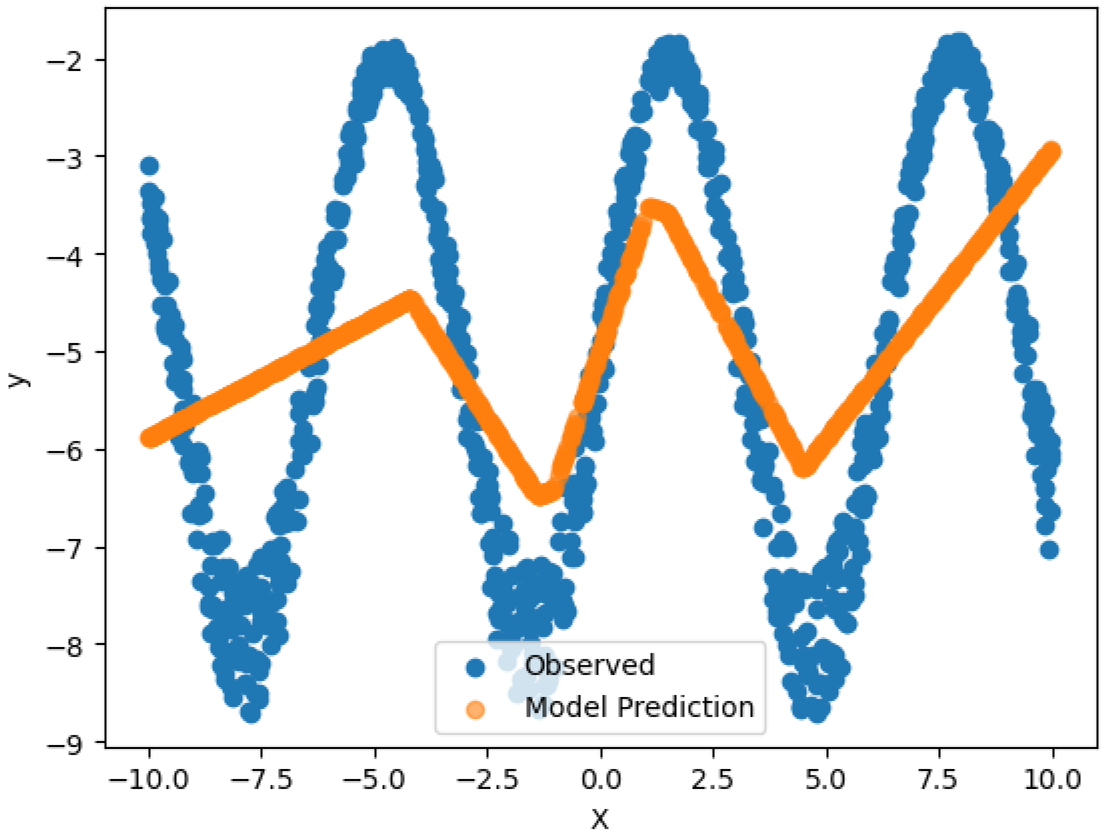
3c

The model prediction is not good at all. Let's add more hidden layers to see if that improved our predictions.

In [67]:
```
# second try with more hidden layers

import numpy as np
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error

x, y = generate_data(1000, 0.1)

MLP = MLPRegressor(hidden_layer_sizes=(1000,), random_state = 49)

mean_squared_error = -cross_val_score(MLP, x.reshape(-1, 1), y, cv = 5, scor

for fold, mse in enumerate(mean_squared_error):
    print(f'Fold {fold + 1}: MSE = {mse}')
```

```
MLP.fit(x.reshape(-1, 1), y)
y_pred = MLP.predict(x.reshape(-1, 1))

plt.figure()
plt.scatter(x, y, label='Observed')
plt.scatter(x, y_pred, label='Model Prediction', alpha=0.6)
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```

```
Fold 1: MSE = 3.0936014218491175
Fold 2: MSE = 2.6880131156572475
Fold 3: MSE = 3.0773700882193693
Fold 4: MSE = 3.228400144248916
Fold 5: MSE = 2.4915781161043653
```



I increased the number of hidden layers to 1000 and this greatly improved the model. However, it's still not a good fit to the data despite having very many hidden layers (1000). In real life, I don't know if this is even a realistic number of hidden layers to have in a model.