CHEM277B Homework 5

Trevor Oldham

Problem 1

(A)

We are given the probabilities as follows:

P(+|M) = 0.95 - the probability that a person with the marker has a positive test.

P(-|~M) = 0.95 - the probability that a person without the marker tests negative.

P(M) = 0.01 - the probability that a person has the marker.

We can calculate the probability of P(-|M), P(+|M), and P(~M).

P(~M) = 0.99 - the probability that a person does not have the marker

P(-|M) = (0.01)(0.05) = 0.0005 - the probability that a person has the marker but has a negative test

P(+|~M) = (0.99)(0.05) = 0.0495 - the probability that a person does not have the marker but has a positive test

(B)

We can then use Bayes Theorem to calculate P(M|+) which is the probability that a person with a positive test actually has the marker.

$$P(M|+) = \frac{P(+|M)P(M)}{P(+)}$$

P(+) = (0.01)(0.95) + (0.99)(0.05) = 0.059

$$P(M|+) = \frac{(0.95)(0.01)}{0.059} = 0.161$$

So given a positive test, the person has a 16% chance of actually having the marker.

(C)

If the probability of having the marker were increased to 0.10, what would be the probability of P(M|+)?

We are given the probabilities as follows:

P(+|M) = 0.95 - the probability that a person with the marker has a positive test.

P(-|~M) = 0.95 - the probability that a person without the marker tests negative.

P(M) = 0.1 - the probability that a person has the marker.

We can calculate the probability of P(-|M), P(+|M), and P(~M).

P(~M) = 0.90 - the probability that a person does not have the marker

P(-|M) = (0.10)(0.05) = 0.005 - the probability that a person has the marker but has a negative test

P(+|~M) = (0.90)(0.05) = 0.045 - the probability that a person does not have the marker but has a positive test

$$P(M|+) = \frac{(0.95)(0.10)}{0.14} = 0.679$$

This shows that the probability of correctly identifying people with the marker rises to 67.9 percent when the incidence of the marker is increased by ten times.

Problem 2

(A)

```
In [1]:  import pandas as pd
         import math
         import numpy as np
         wines = pd.read_csv('wines.csv')
         display(wines)
```

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols.1	Proantho-cyanins	Col intensi
0	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.(
1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.:
2	14.83	1.64	2.17	14.0	97	2.80	2.98	0.29	1.98	5.:
3	14.12	1.48	2.32	16.8	95	2.20	2.43	0.26	1.57	5.(
4	13.75	1.73	2.41	16.0	89	2.60	2.76	0.29	1.81	5.(
...	...	...	...	...	...	...	...	...	...	
173	13.40	4.60	2.86	25.0	112	1.98	0.96	0.27	1.11	8.:
174	13.27	4.28	2.26	20.0	120	1.59	0.69	0.43	1.35	10.:
175	13.17	2.59	2.37	20.0	120	1.65	0.68	0.53	1.46	9.:
176	14.13	4.10	2.74	24.5	96	2.05	0.76	0.56	1.35	9.:
177	12.25	1.73	2.12	19.0	80	1.65	2.03	0.37	1.63	3..

178 rows × 15 columns

```
In [2]:   wines = wines.drop(columns = ['Start assignment'])
          wines_normalized = (wines - wines.mean()) / wines.std()
          wines_normalized['ranking'] = wines['ranking']
          wines_normalized
```

Out[2]:

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenol
0	1.514341	-0.560668	0.231400	-1.166303	1.908522	0.806722	1.031908	-0.6577
1	0.294868	0.227053	1.835226	0.450674	1.278379	0.806722	0.661485	0.2261
2	2.253415	-0.623328	-0.716315	-1.645408	-0.191954	0.806722	0.951817	-0.5773
3	1.378844	-0.766550	-0.169557	-0.806975	-0.331985	-0.151973	0.401188	-0.818₄
4	0.923081	-0.542765	0.158499	-1.046527	-0.752080	0.487157	0.731565	-0.5773
...	...	...	...	...	...	...	...	
173	0.491955	2.026281	1.798775	1.648436	0.858284	-0.503494	-1.070491	-0.7380
174	0.331822	1.739837	-0.388260	0.151234	1.418411	-1.126646	-1.340800	0.5475
175	0.208643	0.227053	0.012696	0.151234	1.418411	-1.030776	-1.350811	1.3510
176	1.391162	1.578712	1.361368	1.498716	-0.261969	-0.391646	-1.270720	1.592₁
177	-0.924604	-0.542765	-0.898568	-0.148206	-1.382223	-1.030776	0.000731	0.0654

178 rows × 14 columns

We choose to use the Naive Bayes Classifier method which calculates the gaussian distribution which can tell us the probability of P(wine attribute x | classifier).

$$P(x_j|c) = \frac{1}{\sqrt{2\pi\sigma_{jc}^2}}exp(-\frac{x_j - m_{jc}}{2\sigma_{jc}^2})$$

where σ_{jc} is the standard deviation of the j'th feature for a given class c and m_{jc} is the mean of the j'th feature in class c.

To calculate the probability of an alcohol content of 13% given class 1, we first get the mean and standard deviation of the alcohol content for class 1.

In [3]:
```
means_df = wines.groupby('ranking').mean()
display(means_df)

std_df = wines.groupby('ranking').std()
display(std_df)
```

ranking	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phen
1	13.744746	2.010678	2.455593	17.037288	106.338983	2.840169	2.982373	0.290
2	12.278732	1.932676	2.244789	20.238028	94.549296	2.258873	2.080845	0.363
3	13.153750	3.333750	2.437083	21.416667	99.312500	1.678750	0.781458	0.441

ranking	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols
1	0.462125	0.688549	0.227166	2.546322	10.498949	0.338961	0.397494	0.07004
2	0.537964	1.015569	0.315467	3.349770	16.753497	0.545361	0.705701	0.12396
3	0.530241	1.087906	0.184690	2.258161	10.890473	0.356971	0.293504	0.12414

Then we plug in the values of mean and stddev to calculate P(alcohol % = 13 | class 1)

$$P(alcohol = 13 | class1) = \frac{1}{\sqrt{2\pi(0.462125)^2}} exp(-\frac{13 - 13.744746}{2(0.462125)^2})$$

$$P(alcohol = 13 | class1) = 0.160$$

```
In [4]:  def gaussian(x, mean, std):
             return (1/(2*math.pi*std)**(1/2))*math.exp((-(x - mean)**2)/(2*std**

         p = gaussian(13,13.744746,0.462125)
         p

Out[4]:  0.16016435168863044
```

```
In [5]:  class NaiveBayesClassifier():
             def __init__(self):
                 self.type_indices={}    # store the indices of wines that belong to
                 self.type_stats={}      # store the mean and std of each cultivar
                 self.ndata = 0
                 self.trained=False

             @staticmethod
             def gaussian(x,mean,std):
                 return (1/(2*math.pi*std)**(1/2))*math.exp((-(x - mean)**2)/(2*std**

             @staticmethod
             def calculate_statistics(x_values):
                 # Returns a list with length of input features. Each element is a tu
```

```python
        n_feats=x_values.shape[1]
        return [(np.average(x_values[:,n]),np.std(x_values[:,n])) for n in r

    @staticmethod
    def calculate_prob(x_input,stats):
        """Calculate the probability that the input features belong to a spe
        x_input: np.array shape(nfeatures)
        stats: list of tuple [(mean1,std1),(means2,std2),...]
        """
        init_prob = 1
        for i in range(len(x_input[1])):
            init_prob = init_prob * NaiveBayesClassifier.gaussian(x_input[i]
        return init_prob

    def fit(self,xs,ys):
        # Train the classifier by calculating the statistics of different fe
        self.ndata = len(ys)
        for y in set(ys):
            type_filter= (ys==y)
            self.type_indices[y]=type_filter
            self.type_stats[y]=self.calculate_statistics(xs[type_filter])
        self.trained=True


    def predict(self,xs):
        # Do the prediction by outputing the class that has highest probabil
        if (xs.shape[1])>1:
            print("Only accepts one sample at a time!")
        if self.trained:
            guess=None
            max_prob=0
            # P(C|X) = P(X|C)*P(C) / sum_i(P(X|C_i)*P(C_i)) (deniminator for
            for y_type in self.type_stats:
                p_type = (np.sum([self.type_indices[y_type] == True]))/len(se
                prob= NaiveBayesClassifier.calculate_prob(xs, self.type_stat
                if prob>max_prob:
                    max_prob=prob
                    guess=y_type
            return guess
        else:
            print("Please train the classifier first!")
```

```python
In [ ]:
```

```python
In [6]: model = NaiveBayesClassifier()
```

```python
In [7]: x_1 = wines_normalized.iloc[0]
        x_1 = x_1.to_numpy().reshape(-1, 1)
        x_1.shape
```

```
Out[7]: (14, 1)
```

In [ ]:

In [8]:
```python
class_1 = wines_normalized[wines_normalized['ranking'] == 1].to_numpy()
class_2 = wines_normalized[wines_normalized['ranking'] == 2].to_numpy()
class_3 = wines_normalized[wines_normalized['ranking'] == 3].to_numpy()
stats_1 = model.calculate_statistics(class_1)
stats_2 = model.calculate_statistics(class_2)
stats_3 = model.calculate_statistics(class_3)
len(stats_1)
```

Out[8]: 14

In [9]:
```python
model.calculate_prob(x_1, stats_1)
```

Out[9]: 0.30308370896130926

In [10]:
```python
model.fit(wines_normalized.drop(columns=['ranking']).to_numpy(), wines_norma
```

In [11]:
```python
print(model.predict(x_1))
```

1

(B)

Divide the normalized features into three sets, each set uses 2/3 of the data for training and 1/3 of the data for testing.

In [12]:
```python
from sklearn.model_selection import train_test_split,KFold
```

In [13]:
```python
def calculate_accuracy(model,xs,ys):
    y_pred=np.zeros_like(ys)
    for idx,x in enumerate(xs):
        x = x.reshape(-1, 1)
        y_pred[idx]=model.predict(x)
    return np.sum(ys==y_pred)/len(ys)
```

In [14]:
```python
def Kfold(k,Xs, ys):
    # 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]

        model = NaiveBayesClassifier()

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

        model.fit(train_in, train_real)

        print("The accuracy of this fold is ", calculate_accuracy(model, val

    return
```

The Kfold function splits the data into three sets and runs the function calculate_accuracy to output the ratio of successful matches using the Naive Bayes Classifier. Here it returns a value of 70 percent accuracy on the first fold and 62.5% accuracy on the second fold, and 72.5% on the third fold, which is less accurate than the clustering method used in HW2 but still very effective.

In [15]:
```python
np.random.seed(0)
x_values = wines_normalized.drop(columns = ['ranking']).to_numpy()
print(x_values.shape)
print(x_values[0].shape)
y_values = wines_normalized['ranking'].to_numpy()
Kfold(3, x_values, y_values)
```

```
(178, 13)
(13,)
The accuracy of this fold is  0.7
The accuracy of this fold is  0.625
The accuracy of this fold is  0.725
```

Problem 3

(A)

```
In [16]:   import numpy as np
           import matplotlib.pyplot as plt
           from mpl_toolkits.mplot3d import Axes3D
           %matplotlib notebook

           def generate_X(number):
               xs=(np.random.random(number)*2-1)*2
               ys=(np.random.random(number)*2-1)*2
               return np.hstack([xs.reshape(-1,1),ys.reshape(-1,1)])

           def generate_data(number,stochascity=0.05):
               X=generate_X(number)
               xs=X[:,0]
               ys=X[:,1]
               fs=(1-xs)**2+10*(ys-xs**2)**2
               stochastic_ratio=(np.random.random(number)*2-1)*stochascity+1
               return np.hstack([xs.reshape(-1,1),ys.reshape(-1,1)]),fs*stochastic_rati
```

```
In [17]:   from torch import nn
           import torch

           class MLP(nn.Module):
               def __init__(self):
                   super(MLP, self).__init__()
                   self.layers = nn.Sequential(
                       nn.Linear(13, 3),
                       nn.Softmax()
                   )

               def forward(self, x):
                   return self.layers(x)
```

The values of x and y are taken from the wines dataframe and converted to torch.tensor, then a new MLP object is created and we call the predict() function on the x values to output the prediction.

In [18]:
```python
np.random.seed(0)
x_values = wines_normalized.drop(columns = ['ranking']).to_numpy()
print(x_values.shape)
y_values = wines_normalized['ranking'].to_numpy()
y_values = y_values - 1


x_values = torch.tensor(x_values, dtype=torch.float32)
y_values = torch.tensor(y_values, dtype=torch.float32)

print(x_values.shape)
print(y_values.shape)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
(178, 13)
torch.Size([178, 13])
torch.Size([178])
```

In [19]:
```python
#Pass the data through the network without backpropagation and print the out
net = MLP()
layers = net.forward(x_values)
print(layers)
```

```
tensor([[0.4359, 0.3438, 0.2203],
        [0.2319, 0.2892, 0.4789],
        [0.4531, 0.3826, 0.1643],
        [0.6256, 0.2235, 0.1509],
        [0.5477, 0.2894, 0.1630],
        [0.4221, 0.2766, 0.3014],
        [0.2981, 0.4604, 0.2415],
        [0.4767, 0.2785, 0.2448],
        [0.4840, 0.2502, 0.2658],
        [0.5116, 0.2968, 0.1916],
        [0.5055, 0.3087, 0.1857],
        [0.3790, 0.3067, 0.3143],
        [0.3217, 0.3176, 0.3608],
        [0.3961, 0.3250, 0.2789],
        [0.3947, 0.3402, 0.2651],
        [0.2510, 0.5161, 0.2328],
        [0.3872, 0.3747, 0.2381],
        [0.5847, 0.2610, 0.1543],
        [0.2491, 0.5267, 0.2241],
        [0.2966, 0.4504, 0.2530],
        [0.2556, 0.4124, 0.3320],
        [0.0968, 0.7001, 0.2031],
        [0.1731, 0.7220, 0.1049],
        [0.3955, 0.2735, 0.3311],
        [0.2536, 0.3128, 0.4336],
        [0.3493, 0.3407, 0.3099],
        [0.1854, 0.4650, 0.3497],
        [0.2785, 0.4578, 0.2637],
        [0.3195, 0.3194, 0.3611],
```

```
            [0.3682, 0.3372, 0.2947],
            [0.0956, 0.6706, 0.2338],
            [0.1695, 0.2207, 0.6099],
            [0.1257, 0.7030, 0.1713],
            [0.2860, 0.5468, 0.1672],
            [0.1474, 0.5632, 0.2894],
            [0.0541, 0.7276, 0.2184],
            [0.1320, 0.5192, 0.3488],
            [0.2287, 0.3576, 0.4136],
            [0.1361, 0.7054, 0.1586],
            [0.1626, 0.4699, 0.3676],
            [0.2620, 0.2872, 0.4508],
            [0.2021, 0.4367, 0.3612],
            [0.2943, 0.2316, 0.4740],
            [0.3130, 0.1430, 0.5440],
            [0.2288, 0.1756, 0.5956],
            [0.4125, 0.1499, 0.4376],
            [0.2233, 0.1467, 0.6300],
            [0.3562, 0.0840, 0.5598],
            [0.4303, 0.0740, 0.4957],
            [0.2658, 0.1344, 0.5999],
            [0.1061, 0.1616, 0.7323],
            [0.3131, 0.1323, 0.5546],
            [0.3512, 0.1279, 0.5209],
            [0.3144, 0.1154, 0.5703],
            [0.4192, 0.1299, 0.4509],
            [0.4100, 0.1117, 0.4783],
            [0.3917, 0.0731, 0.5351],
            [0.3007, 0.1215, 0.5778],
            [0.4312, 0.3975, 0.1713],
            [0.3782, 0.3641, 0.2577],
            [0.4067, 0.3908, 0.2025],
            [0.4397, 0.3703, 0.1899],
            [0.3519, 0.5325, 0.1156],
            [0.3393, 0.3736, 0.2871],
            [0.3930, 0.2374, 0.3696],
            [0.2554, 0.2224, 0.5221],
            [0.3901, 0.3049, 0.3051],
            [0.3785, 0.3669, 0.2546],
            [0.4105, 0.3884, 0.2011],
            [0.4489, 0.2559, 0.2953],
            [0.3820, 0.4405, 0.1775],
            [0.3028, 0.3127, 0.3844],
            [0.3022, 0.3116, 0.3862],
            [0.4895, 0.1667, 0.3438],
            [0.3318, 0.4742, 0.1940],
            [0.2918, 0.5994, 0.1088],
            [0.4357, 0.2945, 0.2698],
            [0.4120, 0.3401, 0.2479],
            [0.2195, 0.5904, 0.1901],
            [0.4208, 0.3545, 0.2247],
            [0.2283, 0.4213, 0.3504],
            [0.4178, 0.3768, 0.2055],
```

```
        [0.2824, 0.3687, 0.3489],
        [0.4304, 0.2240, 0.3457],
        [0.2835, 0.5500, 0.1664],
        [0.1347, 0.6776, 0.1877],
        [0.1874, 0.6226, 0.1900],
        [0.3864, 0.4036, 0.2100],
        [0.3552, 0.2846, 0.3602],
        [0.2994, 0.4013, 0.2992],
        [0.3653, 0.3312, 0.3036],
        [0.2143, 0.4924, 0.2933],
        [0.4231, 0.3523, 0.2246],
        [0.2258, 0.3015, 0.4727],
        [0.4120, 0.3635, 0.2246],
        [0.1589, 0.5871, 0.2541],
        [0.1590, 0.5309, 0.3101],
        [0.3086, 0.2504, 0.4410],
        [0.2984, 0.4245, 0.2771],
        [0.3465, 0.2635, 0.3900],
        [0.3018, 0.3971, 0.3011],
        [0.1557, 0.4988, 0.3455],
        [0.1903, 0.2945, 0.5153],
        [0.2951, 0.1655, 0.5393],
        [0.2993, 0.1155, 0.5851],
        [0.3574, 0.1051, 0.5375],
        [0.5369, 0.0939, 0.3692],
        [0.3532, 0.1362, 0.5106],
        [0.3589, 0.1586, 0.4825],
        [0.2868, 0.1408, 0.5724],
        [0.2749, 0.1127, 0.6125],
        [0.1766, 0.1800, 0.6434],
        [0.3052, 0.1334, 0.5614],
        [0.3404, 0.1448, 0.5148],
        [0.4463, 0.0995, 0.4542],
        [0.3085, 0.1565, 0.5350],
        [0.3466, 0.1395, 0.5139],
        [0.3615, 0.3906, 0.2479],
        [0.6469, 0.1997, 0.1534],
        [0.5769, 0.1722, 0.2509],
        [0.5196, 0.3407, 0.1396],
        [0.4087, 0.4766, 0.1147],
        [0.4722, 0.2803, 0.2475],
        [0.5088, 0.3474, 0.1438],
        [0.3662, 0.2903, 0.3434],
        [0.5410, 0.2787, 0.1803],
        [0.4588, 0.2856, 0.2556],
        [0.4855, 0.3292, 0.1853],
        [0.4152, 0.3287, 0.2561],
        [0.2203, 0.5307, 0.2489],
        [0.5089, 0.2463, 0.2448],
        [0.4208, 0.3467, 0.2326],
        [0.4621, 0.3733, 0.1646],
        [0.3050, 0.3904, 0.3046],
        [0.3111, 0.4188, 0.2701],
```

```
                         [0.3882, 0.3455, 0.2663],
                         [0.2200, 0.4240, 0.3560],
                         [0.1933, 0.5343, 0.2724],
                         [0.2470, 0.3374, 0.4156],
                         [0.1738, 0.6256, 0.2007],
                         [0.2070, 0.3704, 0.4227],
                         [0.1522, 0.5125, 0.3352],
                         [0.2389, 0.5175, 0.2435],
                         [0.2114, 0.3941, 0.3945],
                         [0.3639, 0.2146, 0.4215],
                         [0.2789, 0.4208, 0.3003],
                         [0.3431, 0.3056, 0.3513],
                         [0.1141, 0.5502, 0.3357],
                         [0.1688, 0.6564, 0.1748],
                         [0.1023, 0.7295, 0.1682],
                         [0.3626, 0.2789, 0.3585],
                         [0.3926, 0.2833, 0.3241],
                         [0.1990, 0.4057, 0.3953],
                         [0.2252, 0.4291, 0.3457],
                         [0.1198, 0.3751, 0.5051],
                         [0.2249, 0.1832, 0.5919],
                         [0.1653, 0.4540, 0.3807],
                         [0.2162, 0.4496, 0.3342],
                         [0.2162, 0.2785, 0.5053],
                         [0.2360, 0.3401, 0.4239],
                         [0.3321, 0.2245, 0.4434],
                         [0.2229, 0.0973, 0.6798],
                         [0.3724, 0.1514, 0.4762],
                         [0.3241, 0.0883, 0.5876],
                         [0.2190, 0.1120, 0.6690],
                         [0.2174, 0.1418, 0.6408],
                         [0.2947, 0.2035, 0.5018],
                         [0.2693, 0.2728, 0.4579],
                         [0.3165, 0.1209, 0.5625],
                         [0.2089, 0.1946, 0.5964],
                         [0.1967, 0.2228, 0.5804],
                         [0.4590, 0.1078, 0.4332],
                         [0.3545, 0.1049, 0.5406],
                         [0.2189, 0.0617, 0.7194],
                         [0.3244, 0.1218, 0.5537],
                         [0.2973, 0.1654, 0.5373],
                         [0.2566, 0.0769, 0.6664],
                         [0.4089, 0.3617, 0.2294]], grad_fn=<SoftmaxBackward0>)
/Users/trevor/opt/miniconda3/envs/msse-python/lib/python3.9/site-packages/to
rch/nn/modules/container.py:204: UserWarning: Implicit dimension choice for
softmax has been deprecated. Change the call to include dim=X as an argument
.
  input = module(input)
```

By feeding foward the initial training data we get one (n x 3) matrix which holds the probability of the datapoint falling into one of three categories. The probabilities add up to one because of the softmax function. Without the softmax function, I tried the ReLU function which returned values which were either positive or negative and did not reflect the probability of each class.

(B)

A function train_and_val() is used to train the MLP object based on the x values from the wines dataframe. This function uses three-fold validation to train the MLP using 2/3 of the values as training data and then 1/3 of the values for test data. The training continues for a 500 epochs and the epoch with the lowest loss is reported. Finally the best weights are saved and loaded into the model which is returned to the user.

In [20]:
```python
# you can use this framework to do training and validation
def train_and_val(model,Xs,ys,epochs,draw_curve=True):
    """
    Parameters
    --------------
    model: a PyTorch model
    train_X: np.array shape(ndata,nfeatures)
    train_y: np.array shape(ndata)
    epochs: int
    draw_curve: bool
    """
    ### Define your loss function, optimizer. Convert data to torch tensor #
    loss_func = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)


    total_num=len(Xs)
    kf=KFold(n_splits=3,shuffle=True)

    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]

        best_loss = float('inf')
        best_weights = []

        ### Split training examples further into training and validation ###

        val_array=[]
```

```python
        for i in range(epochs):
            ### Compute the loss and do backpropagation ###

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

            train_X = torch.tensor(train_in, dtype=torch.float32)
            train_y = torch.tensor(train_real, dtype=torch.long)
            test_X = torch.tensor(val_in, dtype=torch.float32)
            test_y = torch.tensor(val_real, dtype=torch.long)

            order=list(range(train_X.shape[0]))
            np.random.shuffle(order)
            batch_size = 1
            n=0
            while n<math.ceil(len(order)/batch_size)-1: # Parts that can fil
                pred = model.forward(train_X[order[n*batch_size:(n+1)*batch_
                loss = loss_func(pred, train_y[order[n*batch_size:(n+1)*batc

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                n+=1
            # Parts that cannot fill one batch
            pred = model.forward(train_X[order[n*batch_size:]])
            loss = loss_func(pred, train_y[order[n*batch_size:(n+1)*batch_si

            #print("training loss ", loss)

            ##set optimizer grad to zero, important,before step()
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            ### compute validation loss and keep track of the lowest val los
            pred = model.forward(test_X)
            loss = loss_func(pred, test_y)

            if loss.item() < best_loss:
                best_weights = model.state_dict()
                best_loss = loss.item()

            val_array.append(loss.item())


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

    model.load_state_dict(best_weights)
```

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

    return model
```

In [23]:
```
#running the train and validate function
net = MLP()
train_and_val(net, x_values ,y_values,epochs=200,draw_curve=True)
```

<ipython-input-20-250d04ec446e>:41: UserWarning: To copy construct from a te
nsor, it is recommended to use sourceTensor.clone().detach() or sourceTensor
.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTenso
r).
  train_X = torch.tensor(train_in, dtype=torch.float32)
<ipython-input-20-250d04ec446e>:42: UserWarning: To copy construct from a te
nsor, it is recommended to use sourceTensor.clone().detach() or sourceTensor
.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTenso
r).
  train_y = torch.tensor(train_real, dtype=torch.long)
<ipython-input-20-250d04ec446e>:43: UserWarning: To copy construct from a te
nsor, it is recommended to use sourceTensor.clone().detach() or sourceTensor
.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTenso
r).
  test_X = torch.tensor(val_in, dtype=torch.float32)
<ipython-input-20-250d04ec446e>:44: UserWarning: To copy construct from a te
nsor, it is recommended to use sourceTensor.clone().detach() or sourceTensor
.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTenso
r).
  test_y = torch.tensor(val_real, dtype=torch.long)
/Users/trevor/opt/miniconda3/envs/msse-python/lib/python3.9/site-packages/to
rch/nn/modules/container.py:204: UserWarning: Implicit dimension choice for
softmax has been deprecated. Change the call to include dim=X as an argument
.
  input = module(input)
Number of epochs with lowest validation: 135
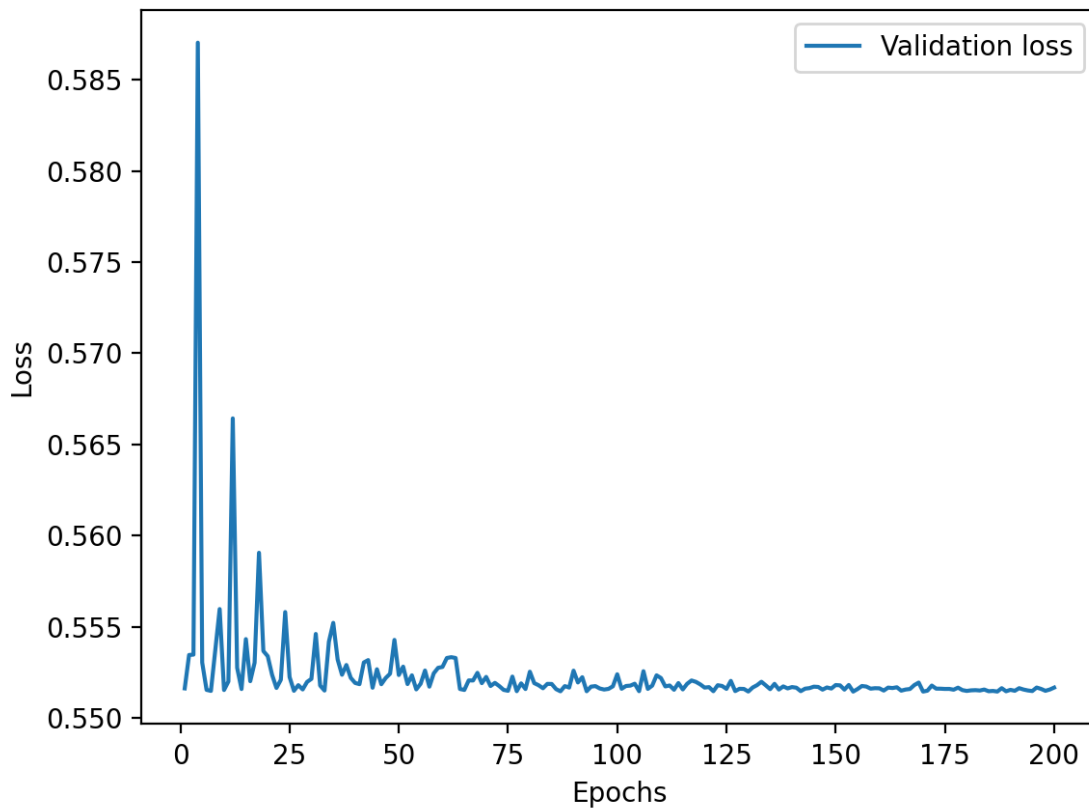Number of epochs with lowest validation: 152
Number of epochs with lowest validation: 154

Out[23]:
```
MLP(
  (layers): Sequential(
    (0): Linear(in_features=13, out_features=3, bias=True)
    (1): Softmax(dim=None)
  )
)
```

In [ ]:

In [ ]: