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	-0.818
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	1.592
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	0.29(
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	0.44

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
```

```
        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!")
```

In [ ]:

In [6]:

```
model = NaiveBayesClassifier()
```

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]:
```
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]:
```
model.calculate_prob(x_1, stats_1)
```

Out[9]:
```
0.30308370896130926
```

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

In [11]:
```
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]:
```
from sklearn.model_selection import train_test_split,KFold
```

In [13]:
```
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]:
```
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]:
```
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]:
```
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]:
```
#Pass the data through the network without backpropagation and print the out
net = MLP()
layers = net.forward(x_values)
print(layers)
```

```
tensor([[0.3371, 0.2487, 0.4142],
        [0.4018, 0.3080, 0.2902],
        [0.3002, 0.3086, 0.3912],
        [0.2872, 0.3382, 0.3746],
        [0.3708, 0.2641, 0.3651],
        [0.3306, 0.2221, 0.4474],
        [0.3296, 0.3522, 0.3182],
        [0.3572, 0.3821, 0.2607],
        [0.3474, 0.4611, 0.1915],
        [0.2900, 0.3272, 0.3828],
        [0.3542, 0.3461, 0.2997],
        [0.2500, 0.1772, 0.5727],
        [0.3106, 0.2587, 0.4306],
        [0.3251, 0.3473, 0.3276],
        [0.3939, 0.2282, 0.3778],
        [0.3447, 0.3781, 0.2772],
        [0.3884, 0.3218, 0.2898],
        [0.4220, 0.3284, 0.2495],
        [0.3736, 0.1394, 0.4870],
        [0.3779, 0.2235, 0.3986],
        [0.1665, 0.3490, 0.4845],
        [0.3006, 0.6256, 0.0738],
        [0.2555, 0.6469, 0.0976],
        [0.1847, 0.5629, 0.2524],
        [0.2607, 0.4171, 0.3222],
        [0.2074, 0.7197, 0.0729],
        [0.2707, 0.6227, 0.1065],
        [0.2000, 0.6760, 0.1239],
        [0.2917, 0.5386, 0.1697],
```

```
        [0.2156, 0.7067, 0.0778],
        [0.2946, 0.4621, 0.2433],
        [0.2835, 0.5056, 0.2110],
        [0.4239, 0.5042, 0.0719],
        [0.2545, 0.6003, 0.1452],
        [0.2181, 0.6655, 0.1164],
        [0.5211, 0.4035, 0.0754],
        [0.2196, 0.6734, 0.1070],
        [0.2306, 0.6470, 0.1224],
        [0.2352, 0.7218, 0.0430],
        [0.4025, 0.4451, 0.1524],
        [0.5487, 0.3307, 0.1206],
        [0.1697, 0.7701, 0.0602],
        [0.4368, 0.3793, 0.1838],
        [0.1974, 0.5530, 0.2496],
        [0.3122, 0.3842, 0.3036],
        [0.2751, 0.2462, 0.4786],
        [0.3391, 0.2919, 0.3691],
        [0.2731, 0.3412, 0.3857],
        [0.3018, 0.2799, 0.4183],
        [0.2951, 0.2276, 0.4773],
        [0.2862, 0.4634, 0.2504],
        [0.3774, 0.1751, 0.4475],
        [0.2712, 0.1942, 0.5346],
        [0.3791, 0.1694, 0.4514],
        [0.3420, 0.1904, 0.4676],
        [0.2719, 0.2065, 0.5217],
        [0.3709, 0.0883, 0.5408],
        [0.3692, 0.2084, 0.4224],
        [0.2935, 0.2470, 0.4594],
        [0.4063, 0.1096, 0.4842],
        [0.3160, 0.1350, 0.5490],
        [0.3651, 0.3083, 0.3266],
        [0.3910, 0.2308, 0.3782],
        [0.3108, 0.2738, 0.4154],
        [0.5359, 0.2550, 0.2090],
        [0.3797, 0.3746, 0.2457],
        [0.3905, 0.3251, 0.2844],
        [0.4171, 0.3657, 0.2172],
        [0.3067, 0.3840, 0.3093],
        [0.3121, 0.3350, 0.3529],
        [0.2628, 0.4309, 0.3063],
        [0.4434, 0.1351, 0.4216],
        [0.4339, 0.2553, 0.3108],
        [0.3894, 0.1406, 0.4700],
        [0.3601, 0.3479, 0.2920],
        [0.4623, 0.3081, 0.2296],
        [0.3269, 0.1373, 0.5359],
        [0.2808, 0.2936, 0.4256],
        [0.1390, 0.5533, 0.3078],
        [0.1694, 0.6010, 0.2296],
        [0.2102, 0.6243, 0.1655],
        [0.2000, 0.6977, 0.1023],
```

```
[0.2529, 0.6995, 0.0476],
[0.1704, 0.7231, 0.1065],
[0.1526, 0.7398, 0.1076],
[0.2134, 0.5261, 0.2605],
[0.1623, 0.7837, 0.0540],
[0.2872, 0.5876, 0.1252],
[0.2401, 0.6939, 0.0660],
[0.2298, 0.6166, 0.1536],
[0.1728, 0.6690, 0.1583],
[0.3438, 0.5773, 0.0789],
[0.1556, 0.7233, 0.1211],
[0.3230, 0.4986, 0.1784],
[0.2427, 0.6377, 0.1196],
[0.1938, 0.7117, 0.0944],
[0.3537, 0.5510, 0.0953],
[0.3279, 0.3573, 0.3149],
[0.1898, 0.7289, 0.0814],
[0.2602, 0.4087, 0.3311],
[0.3460, 0.5191, 0.1349],
[0.2263, 0.6612, 0.1125],
[0.2500, 0.6598, 0.0902],
[0.2463, 0.4113, 0.3424],
[0.3547, 0.2043, 0.4409],
[0.2723, 0.2817, 0.4460],
[0.2488, 0.3036, 0.4477],
[0.4138, 0.2067, 0.3795],
[0.2688, 0.2840, 0.4472],
[0.3293, 0.2149, 0.4557],
[0.3362, 0.2275, 0.4363],
[0.3934, 0.1933, 0.4133],
[0.2921, 0.2529, 0.4550],
[0.2560, 0.2322, 0.5118],
[0.3370, 0.2862, 0.3768],
[0.2615, 0.3273, 0.4112],
[0.2385, 0.2574, 0.5041],
[0.5106, 0.2605, 0.2289],
[0.3502, 0.2114, 0.4384],
[0.2833, 0.1662, 0.5505],
[0.4388, 0.2512, 0.3100],
[0.4503, 0.1295, 0.4203],
[0.3593, 0.1707, 0.4701],
[0.3364, 0.1259, 0.5377],
[0.4115, 0.2213, 0.3672],
[0.3445, 0.3635, 0.2920],
[0.3434, 0.2230, 0.4336],
[0.3768, 0.2484, 0.3748],
[0.3501, 0.4440, 0.2060],
[0.3634, 0.3138, 0.3228],
[0.4214, 0.2831, 0.2955],
[0.4975, 0.1835, 0.3190],
[0.3630, 0.2294, 0.4076],
[0.4148, 0.3271, 0.2581],
[0.3410, 0.2542, 0.4048],
```

```
        [0.3880, 0.1940, 0.4180],
        [0.1686, 0.4149, 0.4165],
        [0.3067, 0.5188, 0.1745],
        [0.1498, 0.2826, 0.5677],
        [0.2116, 0.6233, 0.1651],
        [0.3216, 0.5693, 0.1091],
        [0.3599, 0.4994, 0.1407],
        [0.2105, 0.6157, 0.1738],
        [0.4464, 0.4343, 0.1193],
        [0.3403, 0.3984, 0.2612],
        [0.1886, 0.7290, 0.0825],
        [0.2123, 0.6675, 0.1202],
        [0.3476, 0.5422, 0.1102],
        [0.2370, 0.6644, 0.0986],
        [0.2970, 0.5906, 0.1125],
        [0.2217, 0.6018, 0.1765],
        [0.1724, 0.7026, 0.1250],
        [0.2441, 0.6508, 0.1051],
        [0.1623, 0.7514, 0.0863],
        [0.3585, 0.5315, 0.1099],
        [0.4274, 0.4565, 0.1161],
        [0.5550, 0.3926, 0.0524],
        [0.2546, 0.6335, 0.1120],
        [0.1552, 0.4635, 0.3813],
        [0.2231, 0.3389, 0.4380],
        [0.2178, 0.2211, 0.5611],
        [0.4661, 0.1909, 0.3430],
        [0.3259, 0.1642, 0.5099],
        [0.3870, 0.1579, 0.4551],
        [0.2692, 0.3534, 0.3773],
        [0.2929, 0.3045, 0.4026],
        [0.3056, 0.1310, 0.5634],
        [0.1847, 0.3670, 0.4483],
        [0.4019, 0.1242, 0.4739],
        [0.4377, 0.2144, 0.3479],
        [0.4157, 0.2263, 0.3580],
        [0.2724, 0.3001, 0.4275],
        [0.3247, 0.2977, 0.3776],
        [0.4625, 0.1637, 0.3738],
        [0.2777, 0.0944, 0.6279],
        [0.2141, 0.1382, 0.6476],
        [0.3811, 0.1552, 0.4637],
        [0.2154, 0.6727, 0.1119]], 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]:
```
# 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=[]
```

```
    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)
```

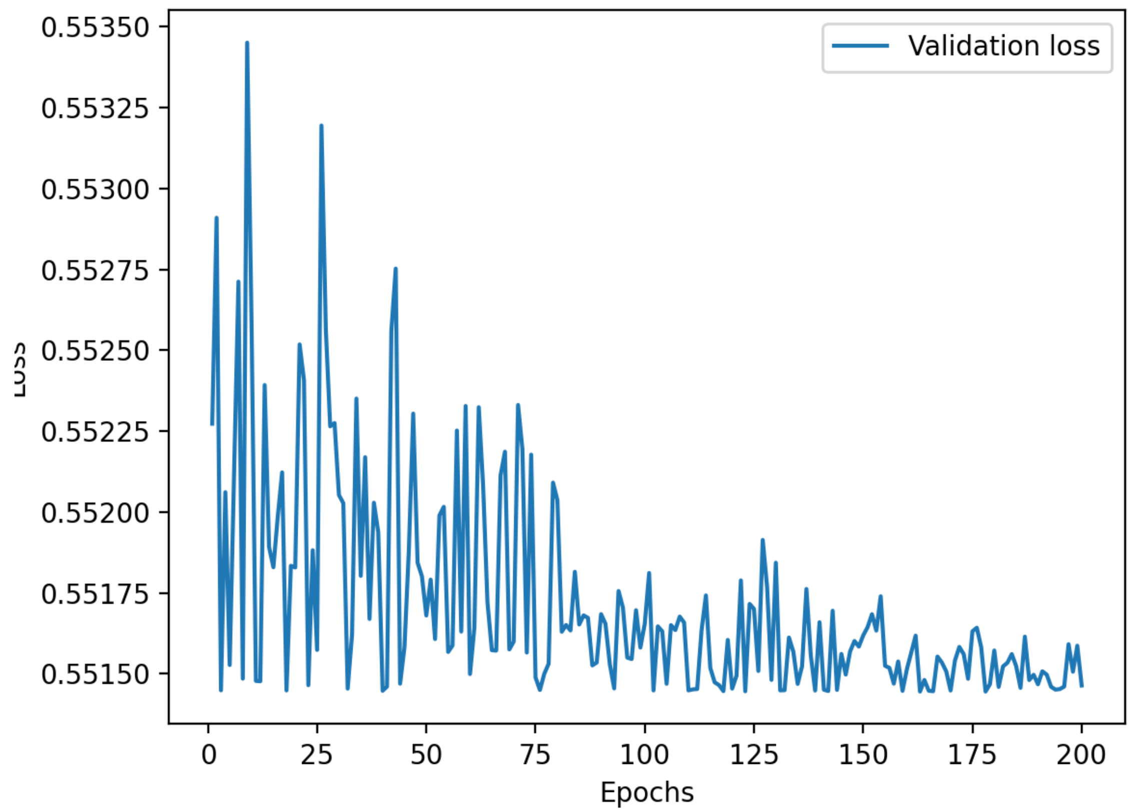
```
    if draw_curve:
        plt.figure()
        plt.plot(np.arange(len(val_array))+1,val_array,label='Validation los
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()

    return model
```

In [21]:

```
#running the train and validate function
net = MLP()
model = 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)
Number of epochs with lowest validation: 172
Number of epochs with lowest validation: 106
Number of epochs with lowest validation: 163
```



Use the MLP prediction to classify the wines according to cultivar.

In [22]:
```
def calculate_accuracy_mlp(model,xs,ys):
    y_pred=np.zeros_like(ys)
    count = 0
    for idx,x in enumerate(xs):
        #x = x.reshape(-1, 1)
        y_pred[idx]=torch.argmax(model.forward(x))

        if (ys[idx] == y_pred[idx]):
            count += 1
    print("The ground truth Y values: ", ys)
    print("The MLP prediction Y values: ", y_pred)
    print("Proportion of correctly classified values", count/len(xs))
    return count/len(xs)

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)
```

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

In [23]:
```
calculate_accuracy_mlp(model, x_values, y_values)
```

```
The ground truth Y values:  tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1.,
        1., 1., 1., 1., 1., 1., 1., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
2.,
        2., 2., 2., 2., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
        0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2., 2.,
2.,
        2., 2., 2., 2., 2., 2., 2., 2., 2., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1.,
1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2.,
2.,
        2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 1.])
The MLP prediction Y values:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 1.]
Proportion of correctly classified values 1.0
/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)
```

Out[23]:
```
1.0
```

We find that the MLP model can correctly classify each wine perfectly, which is a surprise.

In [ ]: