Chem277B: Machine Learning Algorithms

Homework assignment #5: Regression

In [3]:
```python
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split, KFold
```

1. Baye’s Theorem.

(a) From the given data, the categories of testing results and their probabilities within proportion have been summarized in the table below:

Category	Probability within Proportion	Kidney Disease Positive	Marker	Proportion
P[+|M]	0.95	$+$	$+$	0.95 * 0.01
P[-|M]	(1-0.95)	$-$	$+$	(1-0.95) * 0.01
P[+|not M]	(1-0.95)	$+$	$-$	(1-0.95) * 0.99
P[-|not M]	0.95	$-$	$-$	0.95 * 0.99

Hence the quantities for the questions are:

(a1) P[-|M] = (1-0.95) = 0.05 within its proportion, the absolute probability is 0.05 * 0.01 = 0.0005

(a2) P[+|not M] = (1-0.95) = 0.05 within its proportion, the absolute probability is 0.05 * 0.99 = 0.0495

(a3) P[not M] = (1-0.95) *0.99 + 0.95* 0.99 = 0.99, or 1 - 0.01 = 0.99

(b) Using the Baye's Theorem, we try to differentiate between positive marker + positive test and positive marker + negative test. Hence the calculation is defined as:

$$P[M|+] = \frac{P[+|M] * P[M]}{P[+|M] * P[M] + P[+|notM] * P[notM]}$$

$$= \frac{0.95 \times 0.01}{(0.95 \times 0.01) + (0.05 \times 0.99)} = 0.161$$

Hence the chance of testing positive and actually have the marker is 16.1%. It warrants additional testing to confirm.

(c) When P[M] = 0.10, the categories and probabilities become the following:

Category	Probability within Proportion	Kidney Disease Positive	Marker	Proportion
P[+|M]	0.95	$+$	$+$	0.95 * 0.10
P[-|M]	(1-0.95)	$-$	$+$	(1-0.95) * 0.10
P[+|not M]	(1-0.95)	$+$	$-$	(1-0.95) * 0.90
P[-|not M]	0.95	$-$	$-$	0.95 * 0.90

Hence with the new frequency, the individual who test positive actually has the marker is:

$$P[M|+]' = \frac{P[+|M]' * P[M]'}{P[+|M]' * P[M]' + P[+|notM]' * P[notM]'}$$

$$= \frac{0.95 \times 0.1}{(0.95 \times 0.1) + (0.05 \times 0.9)} = 0.679$$

2. Gaussian Naive Bayes.

(a) The finished codes are shown below.

I chose Gaussian distribution because it's a widely used normal probability distributions in statistics, data analysis and visualization.

The Gaussian distribution has a bell curve with mean and standard deviation, hence suitable for modeling many real-world phenomena.

With the finished function, I calculated that a wine from cultivar 1 has a 23.24% probability of containinh 13% Alcohol.

In [4]:
```
class NaiveBayesClassifier():
    def __init__(self):
        self.type_indices={}    # store the indices of wines that belong to each cultivar as a boolean array of length
        self.type_stats={}      # store the mean and std of each cultivar
        self.ndata = 0
```

```python
        self.trained=False

    @staticmethod
    def gaussian(x,mean,std):
        exponent = -(x - mean)**2 / (2 * std**2)

        return (np.exp(exponent) / (np.sqrt(2 * np.pi) * std))

    @staticmethod
    def calculate_statistics(x_values):
        # Returns a list with length of input features. Each element is a tuple, with the input feature's average and s
        n_feats=x_values.shape[1]
        return [(np.average(x_values[:,n]),np.std(x_values[:,n])) for n in range(n_feats)]

    @staticmethod
    def calculate_prob(x_input,stats):
        """Calculate the probability that the input features belong to a specific class(P(X|C)), defined by the statist
        x_input: np.array shape(nfeatures)
        stats: list of tuple [(mean1,std1),(means2,std2),...]
        """
        init_prob = 1
        for i in range(len(x_input)):
            mean, std = stats[i]
            init_prob *= NaiveBayesClassifier.gaussian(x_input[i], mean, std)
        return init_prob

    def fit(self,xs,ys):
        # Train the classifier by calculating the statistics of different features in each class
        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 probability
        if len(xs.shape)>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 normalization only, can be ignored)
            for y_type in self.type_stats:
                pre = sum(self.type_indices[y_type]) / self.ndata
                prob= self.calculate_prob(xs, self.type_stats[y_type]) * pre
                if prob>max_prob:
                    max_prob=prob
                    guess=y_type
            return guess
```

```
        else:
            print("Please train the classifier first!")

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

In [5]:
```
# Import wines.csv
wines = pd.read_csv('wines.csv')
wines.head()
```

Out[5]:

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols.1	Proantho-cyanins	Color intensity	Hue	OD280 315	Proline	Start assignment	ranking
0	14.23	1.71	2.43	15.6	127	2.8	3.06	0.28	2.29	5.64	1.04	3.92	1065	1	1
1	13.24	2.59	2.87	21.0	118	2.8	2.69	0.39	1.82	4.32	1.04	2.93	735	1	1
2	14.83	1.64	2.17	14.0	97	2.8	2.98	0.29	1.98	5.20	1.08	2.85	1045	1	1
3	14.12	1.48	2.32	16.8	95	2.2	2.43	0.26	1.57	5.00	1.17	2.82	1280	1	1
4	13.75	1.73	2.41	16.0	89	2.6	2.76	0.29	1.81	5.60	1.15	2.90	1320	1	1

In [6]:
```
# Define the instance from the Naive Bayes Classifier
nbc = NaiveBayesClassifier()

# Fit the Naive Bayes Classifier
nbc.fit(wines.loc[:, 'Alcohol %':'Proline'].values, wines.loc[:, 'ranking'].values)

# First get the stats for cultivar 1
type_stats = nbc.type_stats[1]

# Then calculate the probability of alcohol% = 13
probability = nbc.gaussian(13, type_stats[0][0], type_stats[0][1])

print(f"A wine from cultivar 1 has a {round(probability*100, 2)}% probability of containinh 13% Alcohol")
```

```
A wine from cultivar 1 has a 23.24% probability of containinh 13% Alcohol
```

(b) After 3-fold training, I can achieve close to 100% accuracy in very short term. The Naive Baye's method performs much better and much faster than the simulated annealing method.

In [7]:
```
# First normalize the wines dataframe
wines_norm = wines.loc[:, 'Alcohol %':'Proline']
wines_norm = (wines_norm - np.mean(wines_norm, axis=0)) / np.std(wines_norm, axis=0)
```

```
wines_norm = wines_norm.merge(wines[['Start assignment','ranking']], left_index=True, right_index=True)
wines_norm.head()
```

Out[7]:

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols.1	Proantho-cyanins	Color intensity	Hue	OD280 315	Proline
0	1.518613	-0.562250	0.232053	-1.169593	1.913905	0.808997	1.034819	-0.659563	1.224884	0.251717	0.362177	1.847920	1.01300
1	0.295700	0.227694	1.840403	0.451946	1.281985	0.808997	0.663351	0.226796	0.401404	-0.319276	0.362177	0.449601	-0.03787
2	2.259772	-0.625086	-0.718336	-1.650049	-0.192495	0.808997	0.954502	-0.578985	0.681738	0.061386	0.537671	0.336606	0.94931
3	1.382733	-0.768712	-0.170035	-0.809251	-0.332922	-0.152402	0.402320	-0.820719	-0.036617	-0.025128	0.932531	0.294232	1.69767
4	0.925685	-0.544297	0.158946	-1.049479	-0.754202	0.488531	0.733629	-0.578985	0.383884	0.234414	0.844785	0.407228	1.82505

In [8]:

```
# Divide the normalized wines data into 3-fold training and testing groups
# and use 2/3 training and 1/3 testing for the three divisions
kf = KFold(n_splits=3, shuffle=True)
xs = wines.loc[:, 'Alcohol %':'Proline'].values
ys = wines.loc[:, 'ranking'].values
nbc = NaiveBayesClassifier()
accuracy = []

for train_index, test_index in kf.split(xs):
    x_train, x_test = xs[train_index], xs[test_index]
    y_train, y_test = ys[train_index], ys[test_index]

    # train the classifier
    nbc.fit(x_train,y_train)
    accuracy.append(calculate_accuracy(nbc,x_test,y_test))
    print(f'Accuracy: {calculate_accuracy(nbc,x_test,y_test)}')
print(f'Average accuracy after 3-fold training is {np.array(accuracy).mean()}')
```

```
Accuracy: 0.9833333333333333
Accuracy: 0.9322033898305084
Accuracy: 0.9830508474576272
Average accuracy after 3-fold training is 0.9661958568738229
```

3. Softmax and Cross Entropy Loss.

(a) I did one PyTorch model without softmax and one PyTorch model with softmax. The output without softmax is a cluster of large positive or negative values. The output with softmax is more like probabilities that sum up to 1.

In [9]:

```
# First convert the features and labels to PyTorch tensors
pytorch_features = torch.tensor(wines.loc[:, 'Alcohol %':'Proline'].values , dtype=torch.float32)
pytorch_labels = torch.tensor(wines.loc[:, 'ranking'].values, dtype=torch.int64)
```

```python
# Then define a pytorch model without softmax
model_no_softmax = nn.Sequential(
    nn.Linear(pytorch_features.shape[1], len(np.unique(pytorch_labels)))
)

# Then pass the data through the model once without backpropagation
outputs_no_softmax = model_no_softmax(pytorch_features)

# Finally print out the outputs_no_softmax
print(outputs_no_softmax)
```

```
tensor([[-4.5661e+01, -1.3691e+01, -1.7287e+02],
        [-3.8358e+01, -4.7560e+00, -1.1983e+02],
        [-3.7913e+01, -1.7992e+01, -1.6826e+02],
        [-4.0346e+01, -2.5418e+01, -2.0616e+02],
        [-3.9303e+01, -2.7642e+01, -2.1222e+02],
        [-4.6351e+01, -2.1539e+01, -2.0694e+02],
        [-4.1492e+01, -5.2367e+00, -1.2725e+02],
        [-3.6758e+01, -1.7042e+01, -1.6418e+02],
        [-3.4566e+01, -1.1880e+01, -1.3688e+02],
        [-4.0340e+01, -2.5281e+01, -2.0730e+02],
        [-3.7320e+01, -1.7710e+01, -1.6708e+02],
        [-4.9081e+01, -1.7736e+01, -2.0063e+02],
        [-3.8823e+01, -1.0421e+01, -1.4267e+02],
        [-3.8718e+01, -1.9284e+01, -1.7850e+02],
        [-3.9538e+01, -1.8746e+01, -1.7665e+02],
        [-3.8050e+01, -1.1400e+01, -1.4363e+02],
        [-3.8947e+01, -1.7775e+01, -1.7098e+02],
        [-3.9717e+01, -2.5400e+01, -2.0357e+02],
        [-4.3327e+01, -2.0096e+01, -1.9217e+02],
        [-4.3692e+01, -2.3754e+01, -2.0740e+02],
        [-3.3863e+01, -5.3006e+00, -1.1086e+02],
        [-2.6572e+01, -6.6408e-01, -6.8693e+01],
        [-2.5513e+01, -5.0173e+00, -8.1033e+01],
        [-3.6483e+01, -1.1233e+01, -1.4137e+02],
        [-3.3833e+01,  2.1536e+00, -8.2306e+01],
        [-2.6807e+01, -8.2227e+00, -1.0236e+02],
        [-2.9793e+01, -2.4009e+00, -8.5069e+01],
        [-3.0256e+01,  8.2731e-01, -7.4071e+01],
        [-2.8945e+01, -8.5113e+00, -1.1031e+02],
        [-2.4809e+01, -9.2888e+00, -1.0138e+02],
        [-5.3220e+01, -3.3267e+00, -1.5456e+02],
        [-4.1035e+01,  1.7825e+00, -1.0353e+02],
        [-2.6666e+01,  6.0689e-01, -6.5912e+01],
        [-3.3164e+01, -7.2037e+00, -1.1552e+02],
        [-2.6325e+01,  7.8301e-02, -6.8174e+01],
        [-3.3421e+01, -9.2280e-01, -9.2022e+01],
        [-2.6889e+01, -6.3643e-01, -7.1323e+01],
        [-2.5043e+01,  1.6478e-01, -6.3227e+01],
        [-2.5847e+01, -2.9022e-02, -6.6891e+01],
        [-3.1516e+01, -4.8608e+00, -1.0194e+02],
        [-2.4972e+01,  1.3769e+00, -6.1246e+01],
        [-2.5443e+01,  1.9359e+00, -5.6399e+01],
        [-2.6197e+01, -5.6942e+00, -9.3593e+01],
        [-2.9933e+01, -3.2600e+00, -9.1049e+01],
        [-3.2797e+01, -2.4039e+00, -9.7599e+01],
        [-2.7745e+01, -4.8797e+00, -9.3400e+01],
        [-3.1232e+01, -2.9310e+00, -9.6214e+01],
        [-2.9239e+01, -1.5035e+00, -8.4516e+01],
        [-2.3179e+01, -9.4961e-01, -6.6196e+01],
        [-3.2858e+01, -7.6090e-01, -8.9076e+01],
```

```
[-3.2474e+01,  3.4966e+00, -7.0152e+01],
[-2.5830e+01, -2.4536e+00, -7.6732e+01],
[-3.4043e+01, -4.7178e+00, -1.1024e+02],
[-3.3932e+01, -5.7373e+00, -1.1218e+02],
[-2.8227e+01, -9.0485e+00, -1.0985e+02],
[-2.8914e+01, -8.0324e+00, -1.0596e+02],
[-3.0905e+01, -8.5678e+00, -1.1858e+02],
[-3.3078e+01, -8.1377e+00, -1.2107e+02],
[-3.9062e+01, -1.7007e+01, -1.6958e+02],
[-4.7420e+01, -2.8837e+01, -2.3870e+02],
[-4.7050e+01, -2.7868e+01, -2.3378e+02],
[-3.7636e+01, -1.8309e+01, -1.6853e+02],
[-3.8030e+01, -2.2234e+01, -1.8445e+02],
[-4.3009e+01, -1.7953e+01, -1.8281e+02],
[-3.4556e+01, -8.2571e+00, -1.2451e+02],
[-4.1093e+01, -6.7942e+00, -1.3564e+02],
[-3.8296e+01, -1.2237e+01, -1.4823e+02],
[-4.1512e+01, -2.5034e+01, -2.0733e+02],
[-3.8831e+01, -1.4875e+01, -1.6010e+02],
[-4.1581e+01, -1.7072e+01, -1.7730e+02],
[-3.7959e+01, -1.6594e+01, -1.6493e+02],
[-4.1614e+01, -3.4797e+00, -1.2344e+02],
[-3.3782e+01, -5.2349e+00, -1.1005e+02],
[-4.1155e+01, -1.6132e+01, -1.7438e+02],
[-3.7831e+01, -1.5779e+01, -1.5905e+02],
[-3.7727e+01, -2.2391e+01, -1.8482e+02],
[-4.6558e+01, -2.4895e+01, -2.2207e+02],
[-4.3166e+01, -1.5005e+01, -1.7185e+02],
[-2.9064e+01, -2.1932e+00, -8.4970e+01],
[-3.1233e+01, -5.5775e+00, -1.0240e+02],
[-2.9904e+01,  4.7128e+00, -5.8815e+01],
[-2.5089e+01, -5.0025e+00, -8.2814e+01],
[-2.5729e+01, -7.5196e-01, -6.6891e+01],
[-2.6614e+01, -2.3542e+00, -7.6911e+01],
[-2.5890e+01, -9.1863e-02, -6.3847e+01],
[-4.4030e+01, -2.1521e+00, -1.2335e+02],
[-2.4553e+01,  3.8125e+00, -4.6189e+01],
[-3.0154e+01, -9.6549e+00, -1.1546e+02],
[-2.8079e+01, -4.5186e+00, -9.1888e+01],
[-2.5699e+01, -2.9299e+00, -7.7862e+01],
[-2.5376e+01, -3.9857e+00, -8.0157e+01],
[-2.5014e+01,  4.2057e+00, -4.7636e+01],
[-2.8773e+01, -4.4003e+00, -9.1542e+01],
[-2.9337e+01,  1.2278e+00, -7.2015e+01],
[-2.9348e+01, -8.3176e+00, -1.0915e+02],
[-2.6414e+01,  3.6831e+00, -5.1729e+01],
[-3.2016e+01, -7.1346e+00, -1.1076e+02],
[-3.2612e+01, -3.0588e+00, -9.8778e+01],
[-2.7294e+01, -2.6138e+00, -8.1114e+01],
[-2.3699e+01,  5.1893e-01, -5.9989e+01],
```

```
[-2.8457e+01, -3.9164e+00, -9.1628e+01],
[-2.5084e+01,  1.0339e+00, -5.7621e+01],
[-2.7501e+01, -1.2250e+00, -7.6370e+01],
[-3.1365e+01, -8.9508e-01, -8.6172e+01],
[-2.9951e+01, -8.2420e+00, -1.1598e+02],
[-3.0261e+01, -4.0829e+00, -9.7378e+01],
[-3.0902e+01, -1.1009e+01, -1.2589e+02],
[-2.8451e+01, -2.7472e+00, -8.8603e+01],
[-3.5097e+01, -9.8720e+00, -1.3439e+02],
[-2.9114e+01, -7.0893e+00, -1.0467e+02],
[-3.3338e+01, -1.3145e+01, -1.4229e+02],
[-2.6432e+01, -3.3355e+00, -8.3826e+01],
[-3.2102e+01, -1.6366e+00, -9.2795e+01],
[-3.3893e+01, -4.6300e+00, -1.0944e+02],
[-2.6287e+01, -3.4038e+00, -8.3482e+01],
[-2.8937e+01, -1.4140e+00, -8.2849e+01],
[-2.4941e+01, -2.9993e+00, -7.5593e+01],
[-4.0207e+01, -2.1728e+01, -1.9101e+02],
[-4.0797e+01, -2.5484e+01, -2.0760e+02],
[-4.7120e+01, -2.1216e+01, -2.0965e+02],
[-4.5961e+01, -3.0861e+01, -2.4327e+02],
[-4.5887e+01, -3.2694e+01, -2.4852e+02],
[-4.4670e+01, -2.3668e+01, -2.1143e+02],
[-4.8799e+01, -3.6045e+01, -2.7029e+02],
[-3.9647e+01, -8.3097e+00, -1.3694e+02],
[-3.8778e+01, -1.6707e+01, -1.6737e+02],
[-3.8783e+01, -2.2924e+01, -1.9256e+02],
[-4.6010e+01, -3.1110e+01, -2.4426e+02],
[-3.6364e+01, -1.3887e+01, -1.4890e+02],
[-3.9248e+01, -7.2897e+00, -1.2925e+02],
[-3.5316e+01, -1.8272e+01, -1.6652e+02],
[-3.9109e+01, -1.7290e+01, -1.7153e+02],
[-4.2714e+01, -2.3537e+01, -2.0301e+02],
[-4.3049e+01, -1.7428e+01, -1.8151e+02],
[-4.1883e+01, -1.2383e+01, -1.5731e+02],
[-4.1861e+01, -2.3900e+01, -2.0473e+02],
[-2.9833e+01,  7.6664e-01, -7.3580e+01],
[-3.2753e+01, -6.6852e+00, -1.1007e+02],
[-3.7247e+01, -6.0068e+00, -1.2248e+02],
[-4.7356e+01,  1.5584e+00, -1.1925e+02],
[-4.7095e+01, -9.1325e+00, -1.6182e+02],
[-3.6639e+01, -1.1945e+01, -1.4453e+02],
[-2.9088e+01,  1.3332e+00, -7.0122e+01],
[-3.0210e+01,  1.1201e+00, -7.5722e+01],
[-2.6079e+01, -3.6662e+00, -8.2860e+01],
[-2.8006e+01, -2.0993e+00, -8.0897e+01],
[-2.6025e+01, -1.6696e+00, -7.3420e+01],
[-2.8370e+01,  4.1912e+00, -5.7227e+01],
[-2.6492e+01, -7.9709e-01, -6.9775e+01],
[-3.0302e+01, -7.8865e+00, -1.0711e+02],
```

```
        [-2.5326e+01,  3.3629e+00, -5.1772e+01],
        [-2.6120e+01, -3.2871e+00, -7.9219e+01],
        [-2.5439e+01,  3.0640e+00, -5.3523e+01],
        [-3.0784e+01,  5.4397e+00, -5.7700e+01],
        [-3.4280e+01,  2.4063e+00, -7.6883e+01],
        [-2.8654e+01,  4.2772e+00, -6.0093e+01],
        [-2.4173e+01,  2.7400e-01, -6.1546e+01],
        [-2.5307e+01,  6.6712e-01, -6.1938e+01],
        [-3.8186e+01, -8.0562e-01, -1.0344e+02],
        [-2.8584e+01, -7.7505e+00, -1.0517e+02],
        [-3.0867e+01, -7.7628e+00, -1.1226e+02],
        [-2.8262e+01, -1.1577e+00, -8.3253e+01],
        [-3.7245e+01, -9.5428e+00, -1.3832e+02],
        [-2.7080e+01, -6.8124e+00, -1.0027e+02],
        [-3.4871e+01,  2.2189e+00, -8.1707e+01],
        [-3.1473e+01,  6.0737e-01, -7.8050e+01],
        [-3.0747e+01, -7.2281e+00, -1.0843e+02],
        [-3.2382e+01, -5.1479e+00, -1.0411e+02],
        [-3.0099e+01, -8.7073e+00, -1.1630e+02],
        [-2.9919e+01, -8.1043e+00, -1.0625e+02],
        [-2.7449e+01, -7.8829e+00, -9.9681e+01],
        [-2.7839e+01, -6.8794e+00, -9.8818e+01],
        [-3.3616e+01, -8.4037e+00, -1.2130e+02],
        [-3.3476e+01, -2.9543e+00, -1.0206e+02],
        [-3.8444e+01, -8.0253e+00, -1.3495e+02],
        [-3.8981e+01, -8.3581e+00, -1.3623e+02],
        [-2.8324e+01, -3.8579e+00, -8.9997e+01],
        [-2.5720e+01, -4.4445e+00, -8.2650e+01]], grad_fn=<AddmmBackward0>)
```

In [10]:

```
# Second is to define a pytorch model with softmax
model_softmax = nn.Sequential(
    nn.Linear(pytorch_features.shape[1], len(np.unique(pytorch_labels))),
    nn.Softmax(dim=1)
)

# Then pass the data through the model once without backpropagation
outputs_softmax = model_softmax(pytorch_features)

# Finally print out the outputs_softmax
print(outputs_softmax)
```

```
tensor([[0.0000e+00, 0.0000e+00, 1.0000e+00],
        [2.6625e-44, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [5.5717e-41, 0.0000e+00, 1.0000e+00],
        [1.4148e-25, 0.0000e+00, 1.0000e+00],
        [4.8433e-31, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [2.7850e-30, 0.0000e+00, 1.0000e+00],
        [3.9550e-37, 0.0000e+00, 1.0000e+00],
        [2.0296e-31, 0.0000e+00, 1.0000e+00],
        [2.4643e-27, 0.0000e+00, 1.0000e+00],
        [1.7187e-40, 0.0000e+00, 1.0000e+00],
        [6.6914e-37, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [3.0440e-37, 0.0000e+00, 1.0000e+00],
        [9.9116e-25, 0.0000e+00, 1.0000e+00],
        [5.7453e-43, 0.0000e+00, 1.0000e+00],
        [8.0556e-25, 0.0000e+00, 1.0000e+00],
        [6.8606e-34, 0.0000e+00, 1.0000e+00],
        [8.3917e-26, 0.0000e+00, 1.0000e+00],
        [5.7988e-23, 0.0000e+00, 1.0000e+00],
        [3.5372e-24, 0.0000e+00, 1.0000e+00],
        [1.1272e-37, 0.0000e+00, 1.0000e+00],
        [4.6792e-23, 0.0000e+00, 1.0000e+00],
        [4.5052e-20, 0.0000e+00, 1.0000e+00],
        [1.3924e-34, 0.0000e+00, 1.0000e+00],
        [3.3056e-33, 0.0000e+00, 1.0000e+00],
        [6.9034e-36, 0.0000e+00, 1.0000e+00],
        [5.4652e-35, 0.0000e+00, 1.0000e+00],
        [4.7762e-35, 0.0000e+00, 1.0000e+00],
        [6.5500e-31, 0.0000e+00, 1.0000e+00],
        [5.0583e-25, 0.0000e+00, 1.0000e+00],
        [2.0625e-33, 0.0000e+00, 1.0000e+00],
```

```
[3.7782e-25, 0.0000e+00, 1.0000e+00],
[2.3203e-29, 0.0000e+00, 1.0000e+00],
[5.7599e-41, 0.0000e+00, 1.0000e+00],
[5.5926e-42, 0.0000e+00, 1.0000e+00],
[1.2585e-41, 0.0000e+00, 1.0000e+00],
[5.0368e-40, 0.0000e+00, 1.0000e+00],
[9.8091e-45, 0.0000e+00, 1.0000e+00],
[7.0065e-45, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[1.8094e-41, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[0.0000e+00, 0.0000e+00, 1.0000e+00],
[7.0640e-32, 0.0000e+00, 1.0000e+00],
[4.1946e-38, 0.0000e+00, 1.0000e+00],
[1.8752e-21, 0.0000e+00, 1.0000e+00],
[6.1560e-31, 0.0000e+00, 1.0000e+00],
[2.4091e-24, 0.0000e+00, 1.0000e+00],
[4.5798e-28, 0.0000e+00, 1.0000e+00],
[3.6937e-24, 0.0000e+00, 1.0000e+00],
[1.4013e-45, 0.0000e+00, 1.0000e+00],
[6.3287e-17, 0.0000e+00, 1.0000e+00],
[4.0357e-43, 0.0000e+00, 1.0000e+00],
[4.7233e-33, 0.0000e+00, 1.0000e+00],
[8.8366e-29, 0.0000e+00, 1.0000e+00],
[1.8378e-29, 0.0000e+00, 1.0000e+00],
[7.5938e-18, 0.0000e+00, 1.0000e+00],
[8.3742e-34, 0.0000e+00, 1.0000e+00],
[2.0709e-26, 0.0000e+00, 1.0000e+00],
[2.3012e-40, 0.0000e+00, 1.0000e+00],
[6.0951e-19, 0.0000e+00, 1.0000e+00],
[8.7543e-41, 0.0000e+00, 1.0000e+00],
[1.5860e-36, 0.0000e+00, 1.0000e+00],
[1.3954e-29, 0.0000e+00, 1.0000e+00],
[7.8614e-23, 0.0000e+00, 1.0000e+00],
```

```
        [6.9135e-34, 0.0000e+00, 1.0000e+00],
        [3.1958e-21, 0.0000e+00, 1.0000e+00],
        [5.0499e-27, 0.0000e+00, 1.0000e+00],
        [6.3788e-32, 0.0000e+00, 1.0000e+00],
        [7.0485e-43, 0.0000e+00, 1.0000e+00],
        [6.0346e-36, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [3.4574e-33, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [3.9107e-39, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [2.3284e-31, 0.0000e+00, 1.0000e+00],
        [3.5325e-34, 0.0000e+00, 1.0000e+00],
        [9.5175e-41, 0.0000e+00, 1.0000e+00],
        [3.2268e-31, 0.0000e+00, 1.0000e+00],
        [8.4276e-31, 0.0000e+00, 1.0000e+00],
        [1.0630e-28, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [1.7625e-27, 0.0000e+00, 1.0000e+00],
        [6.1886e-41, 0.0000e+00, 1.0000e+00],
        [4.2039e-45, 0.0000e+00, 1.0000e+00],
        [1.5274e-43, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [3.6396e-26, 0.0000e+00, 1.0000e+00],
        [9.4135e-28, 0.0000e+00, 1.0000e+00],
        [8.5595e-31, 0.0000e+00, 1.0000e+00],
        [2.9163e-29, 0.0000e+00, 1.0000e+00],
        [8.9918e-27, 0.0000e+00, 1.0000e+00],
        [3.9542e-21, 0.0000e+00, 1.0000e+00],
        [3.4725e-26, 0.0000e+00, 1.0000e+00],
        [5.9788e-40, 0.0000e+00, 1.0000e+00],
```

```
        [6.2455e-19, 0.0000e+00, 1.0000e+00],
        [5.3601e-29, 0.0000e+00, 1.0000e+00],
        [1.6891e-19, 0.0000e+00, 1.0000e+00],
        [1.4225e-20, 0.0000e+00, 1.0000e+00],
        [8.4847e-28, 0.0000e+00, 1.0000e+00],
        [1.7840e-21, 0.0000e+00, 1.0000e+00],
        [4.7263e-23, 0.0000e+00, 1.0000e+00],
        [8.5201e-23, 0.0000e+00, 1.0000e+00],
        [6.5661e-38, 0.0000e+00, 1.0000e+00],
        [4.5691e-39, 0.0000e+00, 1.0000e+00],
        [5.9541e-42, 0.0000e+00, 1.0000e+00],
        [1.6407e-30, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [1.1606e-37, 0.0000e+00, 1.0000e+00],
        [4.2509e-30, 0.0000e+00, 1.0000e+00],
        [3.0425e-29, 0.0000e+00, 1.0000e+00],
        [3.8607e-41, 0.0000e+00, 1.0000e+00],
        [1.6956e-38, 0.0000e+00, 1.0000e+00],
        [1.4574e-43, 0.0000e+00, 1.0000e+00],
        [8.6224e-40, 0.0000e+00, 1.0000e+00],
        [1.8716e-37, 0.0000e+00, 1.0000e+00],
        [3.9599e-37, 0.0000e+00, 1.0000e+00],
        [8.4078e-45, 0.0000e+00, 1.0000e+00],
        [8.3640e-38, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00],
        [1.3066e-33, 0.0000e+00, 1.0000e+00],
        [7.8519e-31, 0.0000e+00, 1.0000e+00]], grad_fn=<SoftmaxBackward0>)
```

(b) The finished codes are listed below. I unfortunately always encounter an error of dead kernel when trying to evaluate the train_and_val function in Jupyter Notebook. Hence I ran all the codes in Google Colab and got some interesting results.

In [11]:
```python
def train_and_val(model, train_X, train_y, epochs, draw_curve=True):
    """
    Train and validate a PyTorch model using cross-entropy loss.

    Parameters
    ----------
    model : PyTorch model
        The model to train.
    train_X : numpy.ndarray
        The input training data of shape (n_samples, n_features).
    train_y : numpy.ndarray
        The target training data of shape (n_samples,).
    epochs : int
        The number of training epochs.
    draw_curve : bool, optional
        Whether to draw a validation loss curve (the default is True).
```

```
    Returns
    -------
    float
        The final validation loss.
    """
    # Convert data to torch tensor
    Xs = torch.tensor(train_X).float()
    ys = torch.tensor(train_y).long()

    # Define Kfolds
    kf = KFold(n_splits=3, shuffle=True)
    for train_index, test_index in kf.split(Xs):
        train_X, test_X = Xs[train_index], Xs[test_index]
        train_y, test_y = ys[train_index], ys[test_index]

        # Subtract one from labels to make them 0-based
        train_y -= 1
        test_y -= 1

        # Split training examples further into training and validation
        train_X, val_X, train_y, val_y = train_test_split(train_X, train_y, test_size=0.2)

        # Define loss function and optimizer
        loss_fn = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(), lr=0.001)

        # Keep track of the lowest validation loss
        lowest_val_loss = np.inf

        # Train the model
        val_array = []
        for epoch in range(epochs):
            # Compute training loss and update model parameters
            optimizer.zero_grad()
            train_out = model(train_X)
            train_loss = loss_fn(train_out, train_y)
            train_loss.backward()
            optimizer.step()

            # Compute validation loss and keep track of the lowest validation loss
            val_out = model(val_X)
            val_loss = loss_fn(val_out, val_y)
            val_array.append(val_loss.item())
            if val_loss < lowest_val_loss:
                lowest_val_loss = val_loss
                torch.save(model.state_dict(), 'model.pt')

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

```
        # Recover the model weight
        model.load_state_dict(torch.load('model.pt'))
        model.eval()

        # Compute test accuracy
        test_out = model(test_X)
        test_loss = loss_fn(test_out, test_y)
        test_preds = torch.argmax(test_out, dim=1).numpy()
        test_acc = np.mean(test_preds == test_y.numpy())
        print("Test accuracy:", test_acc)

        # Plot the validation loss curve
        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()

    return lowest_val_loss.item()
```

For evaluation of the function, I used the 2 models generated above, `model_no_softmax` and `model_softmax`. I also created two additional models, one with softmax as the last layer activation function and one with ReLu as the last layer activation function. Their performance in terms of convergence and accuracy of prediction as shown below. In general it seems `model_no_softmax`, the simpliest model gives the best prediction performance. My guess is the other models are over-fitting?

In [ ]:
```
train_and_val(model_no_softmax, pytorch_features, pytorch_labels, 1000, draw_curve=True)
```

In [ ]:
```
train_and_val(model_softmax, pytorch_features, pytorch_labels, 1000, draw_curve=True)
```

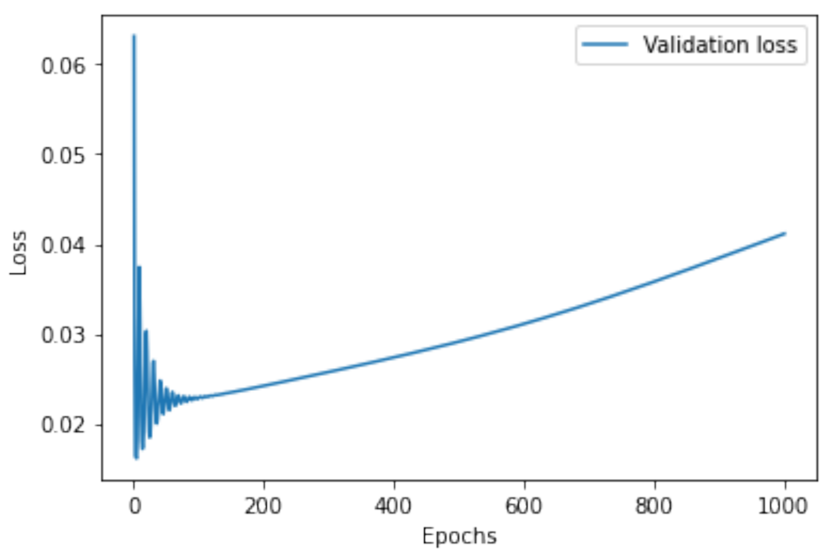
In [ ]:
```
class Classifier_softmax(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_classes):
        super(Classifier_softmax, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = nn.functional.relu(out)
        out = self.fc2(out)
        out = nn.functional.softmax(out, dim=1)
        return out

wine_classifier = Classifier_softmax(pytorch_features.shape[1], len(np.unique(pytorch_labels)), 3)

train_and_val(wine_classifier, pytorch_features, pytorch_labels, 1000, draw_curve=True)
```

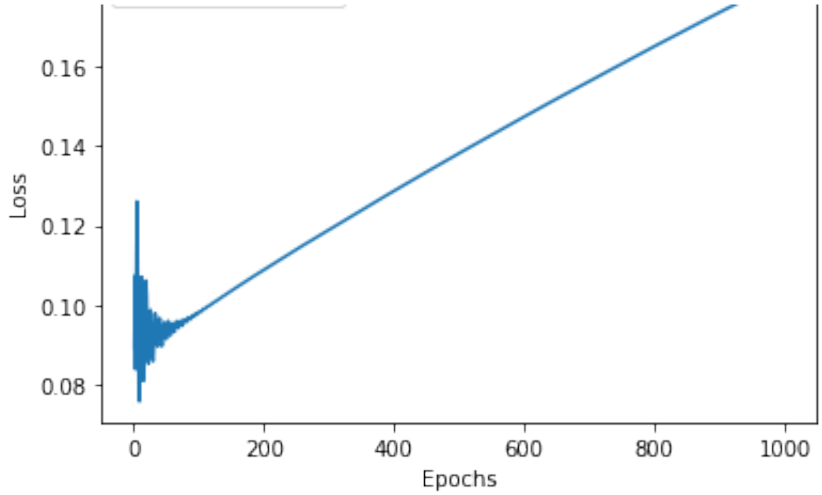
In [ ]:
```
class Classifier_Relu(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_classes):
        super(Classifier_Relu, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

wine_classifier_2 = Classifier_Relu(pytorch_features.shape[1], len(np.unique(pytorch_labels)), 3)

train_and_val(wine_classifier_2, pytorch_features, pytorch_labels, 1000, draw_curve=True)
```

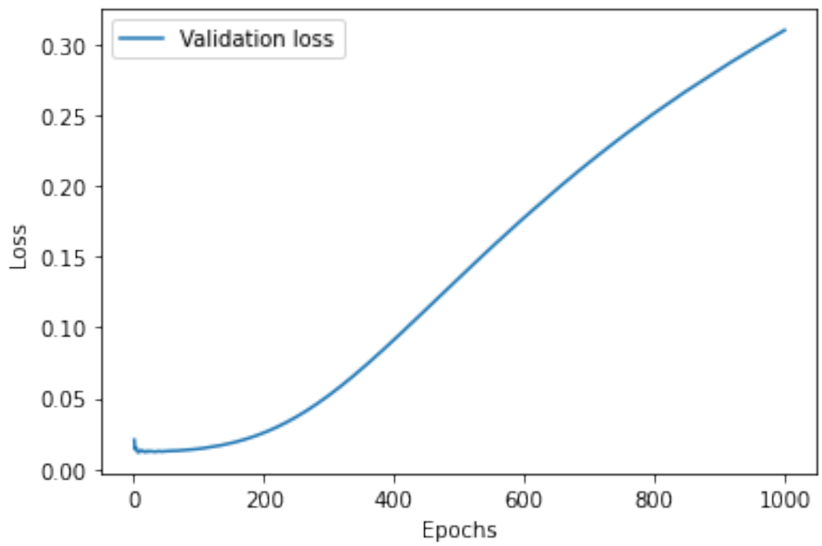
```
        # Plot the validation loss curve
        if draw_curve:
            plt.figure()
            plt.plot(np.arange(len(val_ar
            plt.xlabel('Epochs')
            plt.ylabel('Loss')
            plt.legend()

    return lowest_val_loss.item()
```

```
train_and_val(model_no_softmax,·pytorch_f
```

```
<ipython-input-97-f0eea764c926>:24: UserWarning: To copy construct from a ten
  Xs = torch.tensor(train_X).float()
<ipython-input-97-f0eea764c926>:25: UserWarning: To copy construct from a ten
  ys = torch.tensor(train_y).long()
Number of epochs with lowest validation: 5
Test accuracy: 0.9666666666666667
Number of epochs with lowest validation: 9
Test accuracy: 0.9830508474576272
Number of epochs with lowest validation: 7
Test accuracy: 1.0
0.011702748946845531
```

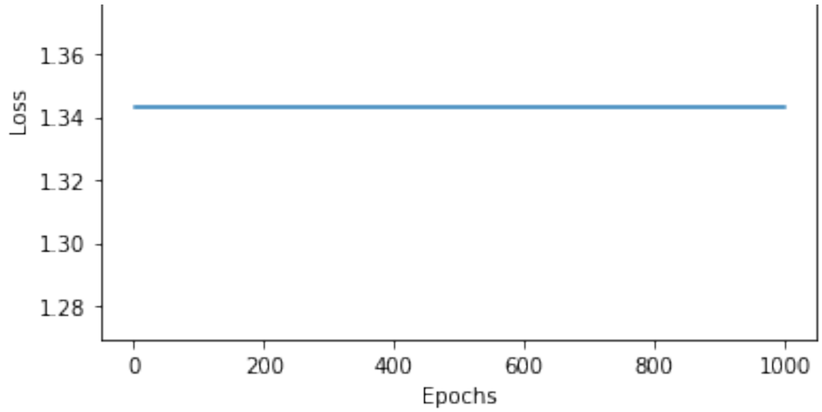


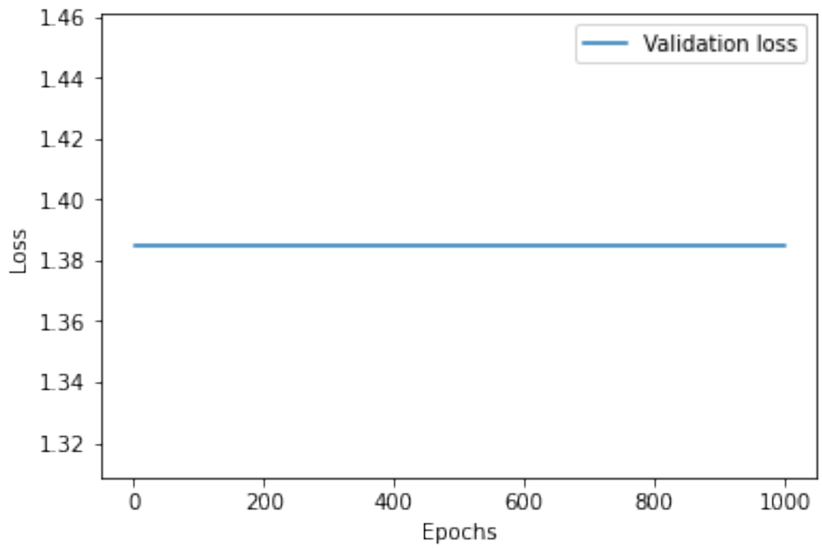


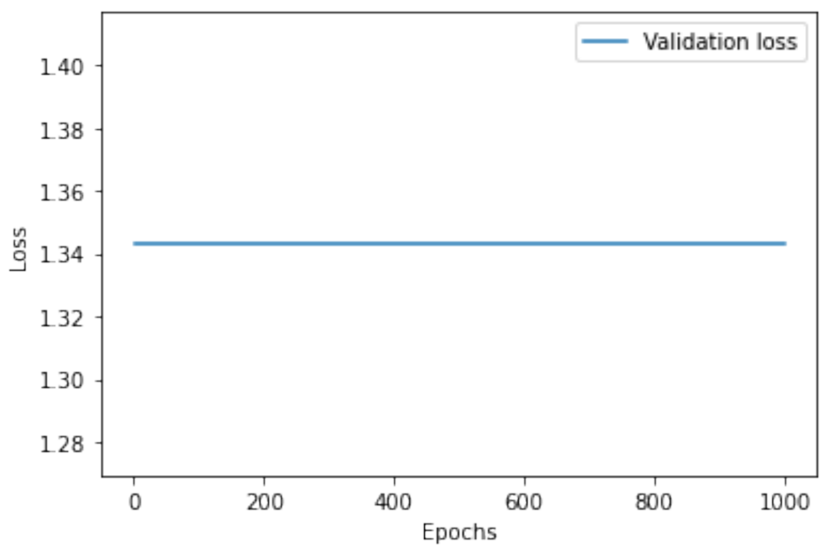



```
train_and_val(model_softmax,·pytorch_feat
```

```
<ipython-input-71-f0eea764c926>:24: UserWarning: To copy construct from a ten
  Xs = torch.tensor(train_X).float()
<ipython-input-71-f0eea764c926>:25: UserWarning: To copy construct from a ten
  ys = torch.tensor(train_y).long()
Number of epochs with lowest validation: 1
Test accuracy: 0.36666666666666664
Number of epochs with lowest validation: 1
Test accuracy: 0.2542372881355932
Number of epochs with lowest validation: 1
Test accuracy: 0.1864406779661017
1.3431113958358765
```


1.36
1.34
1.32
1.30
1.28
Loss
0
200
400
600
800
1000
Epochs


Validation loss
1.46
1.44
1.42
1.40
1.38
1.36
1.34
1.32
Loss
0
200
400
600
800
1000
Epochs

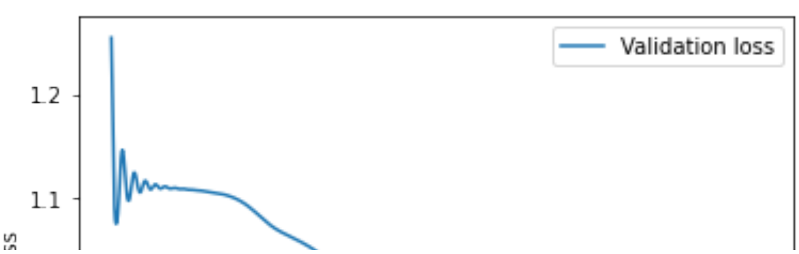

Validation loss
1.40
1.38
1.36
1.34
1.32
1.30
1.28
Loss
0
200
400
600
800
1000
Epochs

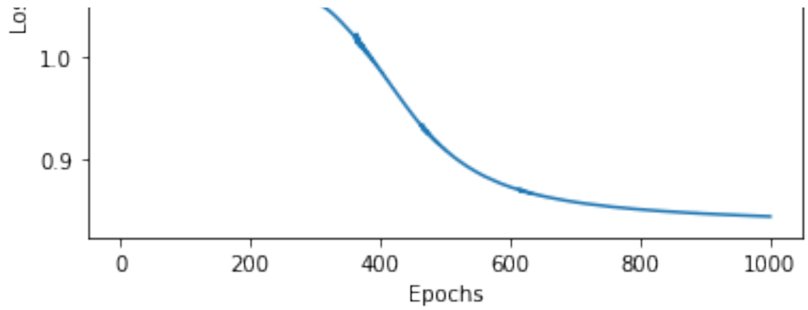
```
class Classifier_softmax(nn.Module):
    def __init__(self, input_dim, hidden_
        super(Classifier_softmax, self)._
        self.fc1 = nn.Linear(input_dim, h
        self.fc2 = nn.Linear(hidden_dim,

    def forward(self, x):
        out = self.fc1(x)
        out = nn.functional.relu(out)
        out = self.fc2(out)
        out = nn.functional.softmax(out,
        return out
```

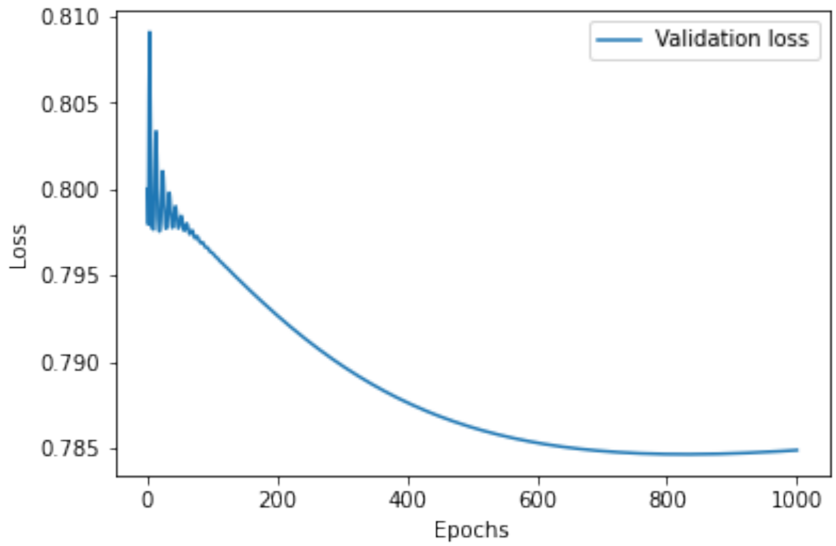
```
wine_classifier = Classifier_softmax(pyto
```

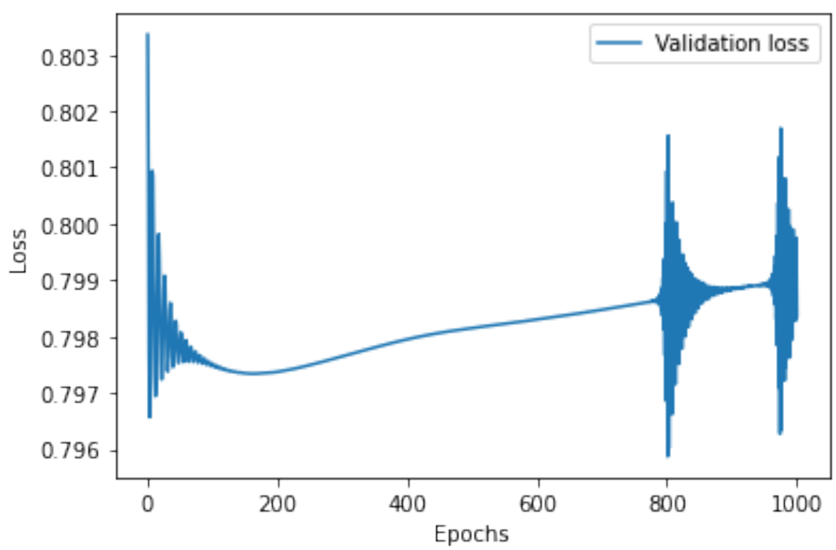
```
train_and_val(wine_classifier,·pytorch_fe
```

```
<ipython-input-71-f0eea764c926>:24: UserWarning: To copy construct from a ten
  Xs = torch.tensor(train_X).float()
<ipython-input-71-f0eea764c926>:25: UserWarning: To copy construct from a ten
  ys = torch.tensor(train_y).long()
Number of epochs with lowest validation: 1000
Test accuracy: 0.6166666666666667
Number of epochs with lowest validation: 828
Test accuracy: 0.6101694915254238
Number of epochs with lowest validation: 801
Test accuracy: 0.7288135593220338
0.7958707809448242
```




Lo
1.0
0.9
0
200
400
600
800
1000
Epochs


Validation loss
Loss
0.810
0.805
0.800
0.795
0.790
0.785
0
200
400
600
800
1000
Epochs

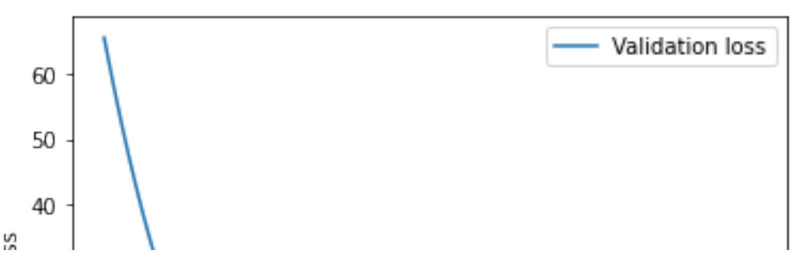

Validation loss
Loss
0.803
0.802
0.801
0.800
0.799
0.798
0.797
0.796
0
200
400
600
800
1000
Epochs

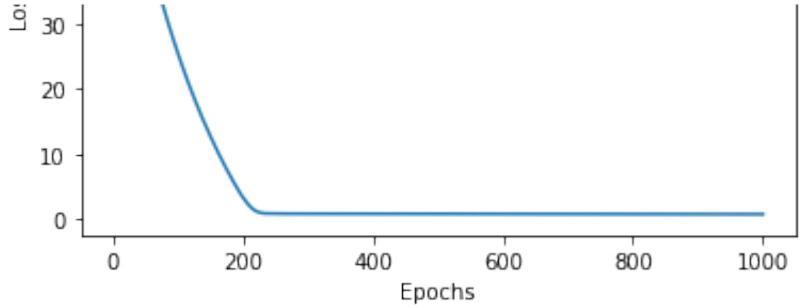
```
class Classifier_Relu(nn.Module):
    def __init__(self, input_dim, hidden_
        super(Classifier_Relu, self).__in
        self.fc1 = nn.Linear(input_dim, h
        self.fc2 = nn.Linear(hidden_dim,
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

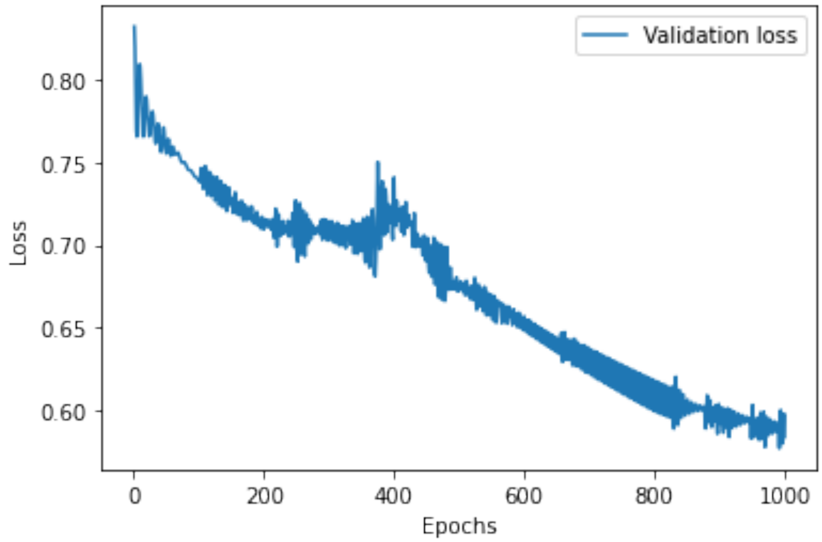
```
wine_classifier_2 = Classifier_Relu(pytor
```

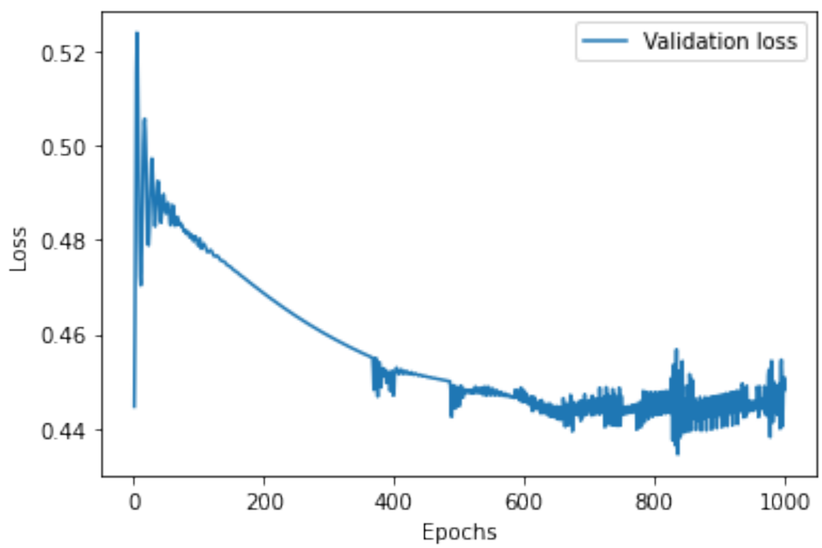
```
train_and_val(wine_classifier_2, pytorch_
```

```
<ipython-input-71-f0eea764c926>:24: UserWarning: To copy construct from a ten
  Xs = torch.tensor(train_X).float()
<ipython-input-71-f0eea764c926>:25: UserWarning: To copy construct from a ten
  ys = torch.tensor(train_y).long()
Number of epochs with lowest validation: 1000
Test accuracy: 0.6166666666666667
Number of epochs with lowest validation: 992
Test accuracy: 0.6949152542372882
Number of epochs with lowest validation: 836
Test accuracy: 0.711864406779661
0.4345228970050812
```




Lo
30
20
10
0
0
200
400
600
800
1000
Epochs


Validation loss
0.80
0.75
0.70
0.65
0.60
Loss
0
200
400
600
800
1000
Epochs


Validation loss
0.52
0.50
0.48
0.46
0.44
Loss
0
200
400
600
800
1000
Epochs