# Question 1

## (a)

The probability of someone being negative, given that they have the marker P[-|M]= 1 - 0.95 = 0.05. The probability that someone is positive, given that they do not have the parker P[+|not M] = 1 - 0.95 = 0.05 The probability that someone does not have the marker P[not M] = 1 - 0.01 = 0.99

## (b)

We need to find P[M | +] = P[+ | M] * P[M] / P[+] = 0.95 * 0.01 / (0.95 * 0.01 + 0.05 * 0.99) = 0.16

You should not be worried about having the genetic marker if you have kidney disease. But the chance is not insignificant. P[M] helps account for this result - the raw probability of the marker occurring in the population is already low.

## (c)

Redoing our math, 0.95 * 0.1 / (0.95 * 0.1 + 0.05 * 0.9) = 0.68

The marker is now highly prevalent in the population. So it is likely that someone with kidney disease does have the genetic marker.

# Question 2

## (a)

We can represent the P(x | C) as a finite product of all the Gaussian probability densities P($x_i$ | C)for each feature $i$ (in this case, 13 features).

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [ ]:
```
# Note: you can modify the template, define new attributes or functions as you want
class NaiveBayesClassifier():
    def __init__(self):
        # classes
        self.cls = []
        # statistics of features that belongs to different classes
        self.cls_stats = []
        # prior probablity, i.e. P(C)
        self.prior_prob = []

        self.trained = False

    @staticmethod
    def gaussian(X, mean, std):
        """
        Gaussain probability distribution function

        Parameters
        ----------
        X: numpy.ndarray
            Input data, shape (n_samples, n_features)
        mean: numpy.ndarray
            Average of each feature, shape (n_features,)
        std: numpy.ndarray
            Standard deviation of each feature, shape (n_features,)

        Returns
        -------
        prob: numpy.ndarray
            Probability of each feature, shape (n_samples, n_features)
        """
        g = np.exp(-0.5 * ((X - mean) / std)**2) / (std * np.sqrt(2 * np.pi))
        return g

    @staticmethod
    def calculate_statistics(X):
        """
        Calculate the average and standard deviation of each feature based on the i

        Parameters
        ----------
        X: numpy.ndarray
            Input data, shape (n_samples, n_features)

        Returns
        -------
        mean: numpy.ndarray
            Average of each feature, shape (n_features,)
        std: numpy.ndarray
            Standard deviation of each feature, shape (n_features,)
        """
        mean = np.mean(X, axis=0)
        std = np.std(X, axis=0)
        return mean, std

    def calculate_prob(self, X, mean, std):
        """
        Calculate the prior probability that the input features belong to a specifi
        i.e. P(X | C) = \prod_i P(X_i | C)
        which is defined by the statistics of features in that class.

        Gaussain probability distribution function

        Parameters
        ----------
        X: numpy.ndarray
            Input data, shape (n_samples, n_features)
        mean: numpy.ndarray
            Average of each feature in the speific class, shape (n_features,)
        std: numpy.ndarray
            Standard deviation of each feature in the specific class, shape (n_feat

        Returns
        -------
        prob: numpy.ndarray
            Probability that the features belong to a spcefic class, shape (n_sampl
        """
        likelihood = self.gaussian(X, mean, std)
        likelihood = np.prod(likelihood, axis=1)
        return likelihood

    def fit(self, X, y):
        """
        Train the classifier by calculating the statistics of different features in

        Parameters
        ----------
        X: numpy.ndarray
            Input data, shape (n_samples, n_features)
        y: numpy.ndarray
            Labels (the actual classes), shape (n_samples)
        """
        ndata = y.shape[0]
        self.cls = np.unique(y)
        self.prior_prob = []
        self.cls_stats = []
        for i in range(len(self.cls)):
            cls_filter = (y == self.cls[i])
            mean, std = self.calculate_statistics(X[cls_filter])
            self.prior_prob.append(np.sum(cls_filter) / ndata)
            self.cls_stats.append((mean, std))
        self.trained = True

    def predict(self, X):
        # Do the prediction by outputing the class that has highest probability
        assert self.trained, f"The classfier has not been trained. Call {self.__cla
        probs = []
        for i in range(len(self.cls)):
            likelihood = self.calculate_prob(X, self.cls_stats[i][0], self.cls_stat
            posterior =  likelihood * self.prior_prob[i]
            #posterior /= np.sum(posterior, axis=0)
            probs.append(posterior)
        return self.cls[np.argmax(probs, axis=0)]
```

In [ ]:
```python
nbc = NaiveBayesClassifier()
wines = pd.read_csv('Datasets/wines.csv')
features = ['Alcohol %','Malic Acid','Ash','Alkalinity','Mg','Phenols','Flavanoids'

X = wines.loc[:, 'Alcohol %']
y = wines.iloc[:, -1]
nbc.fit(X, y)
cls_filter = (y == 1)
# Given a wine that belongs to cultivar 1, what is the chance of it having an Alcoh
mean, std = nbc.calculate_statistics(X[cls_filter])
res = nbc.calculate_prob([[13.0]], mean, std)
print("Chance that alcohol level is 13 percent, given a cultivar of 1:", res)
```

```
Chance that alcohol level is 13 percent, given a cultivar of 1: [0.23236758]
```

In [ ]:
```python
ndata = y.shape[0]
train_size = round(ndata * 2/3)
test_size = ndata - train_size
X_train = X[train_size]
y_train = y[train_size]

X_test = X[test_size]
y_test = y[test_size]

nbc2 = NaiveBayesClassifier()
```

## (b)

Preprocess data:

- Select relevant descriptors (columns other than "Start assignment" and "ranking"). Should get *178 datapoints and 13 features*.
- Normalize the data with `StandardScaler` in sklearn.
- The labels are in the "ranking" column.

*For debugging*: The accuracy could reach over 95%.

In [ ]:
```python
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
```

In [ ]:
```python
def calculate_accuracy(model, X, y):

    y_pred = model.predict(X)

    acc = np.sum(y_pred == y) / len(y)
    return acc

def KFoldNaiveBayes(k, X, y):
    """
    K-Fold Cross Validation for Naive Bayes Classifier

    Parameters
    ---------
    k: int
        Number of folds
    X: numpy.ndarray
        Input data, shape (n_samples, n_features)
    y: numpy.ndarray
        Class labels, shape (n_samples)
    """
    kf = KFold(n_splits=k, shuffle=True)
    train_acc_all = []
    test_acc_all = []
    for train_index, test_index in kf.split(X):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        model = NaiveBayesClassifier()
        model.fit(X_train, y_train)
        # Report prediction accuracy for this fold
        # use the calculate_accuracy() function
        train_acc = calculate_accuracy(model, X_train, y_train)
        train_acc_all.append(train_acc)
        test_acc = calculate_accuracy(model, X_test, y_test)
        test_acc_all.append(test_acc)
        print("Train accuracy:", train_acc)
        print("Test accuracy:", test_acc)

    # report mean & std for the training/testing accuracy
    print("Final results:")
    print("Training accuracy:", np.mean(train_acc_all))
    print("Training std:", np.std(train_acc_all))
    print("Testing  accuracy:", np.mean(test_acc_all))
    print("Testing std:", np.std(test_acc_all))
```

In [ ]:
```
wines = pd.read_csv('Datasets/wines.csv')
X = wines[features]
print(np.shape(X))
normalized_X = pd.DataFrame(StandardScaler().fit_transform(X))
#normalized_X = normalized_X.sample(frac=1).reset_index(drop=True)
KFoldNaiveBayes(3, normalized_X, y)
```

```
(178, 13)
Train accuracy: 0.9745762711864406
Test accuracy: 0.9833333333333333
Train accuracy: 0.9747899159663865
Test accuracy: 0.9830508474576272
Train accuracy: 0.9831932773109243
Test accuracy: 0.9491525423728814
Final results:
Training accuracy: 0.9775198214879172
Training std: 0.004012687109216623
Testing  accuracy: 0.9718455743879474
Testing std: 0.016046811232455657
```

# Question 3

## (a)

In [ ]:
```
import torch
import torch.nn as nn
from sklearn.model_selection import train_test_split
```

In [ ]:
```
# Define your model here
class NeuralNetwork(nn.Module):
    def __init__(self, input_size, output_size, doSoftmax=True):
        super().__init__()
        layers = [nn.Linear(input_size, output_size)]
        if doSoftmax:
            layers.append(nn.Softmax(dim=1))
        self.layers = nn.Sequential(*layers)
    def forward(self, X):
        o = self.layers(X)
        return o
```

In [ ]:
```
wines = pd.read_csv("Datasets/wines.csv")
X = torch.tensor(normalized_X.values, dtype=torch.float32)
y = torch.tensor(wines.iloc[:, -1].values-1)
model = NeuralNetwork(X.shape[1], len(torch.unique(y)))
model2 = NeuralNetwork(X.shape[1], len(torch.unique(y)), doSoftmax=False)
res = model(X)
print("With softmax:\n", res)
res2 = model2(X)
print("without softmax:\n", res2)
```

```
With softmax:
 tensor([[0.0815, 0.3422, 0.5763],
        [0.3964, 0.2073, 0.3963],
        [0.1077, 0.1989, 0.6934],
        [0.1377, 0.3219, 0.5404],
        [0.1551, 0.2875, 0.5575],
        [0.1566, 0.1840, 0.6594],
        [0.1013, 0.2672, 0.6316],
        [0.2219, 0.4127, 0.3655],
        [0.2260, 0.3443, 0.4297],
        [0.1382, 0.3876, 0.4742],
        [0.1061, 0.3659, 0.5281],
        [0.1702, 0.2036, 0.6262],
        [0.4424, 0.2044, 0.3532],
        [0.2968, 0.2727, 0.4305],
        [0.1876, 0.2303, 0.5820],
        [0.1165, 0.3437, 0.5398],
        [0.1343, 0.2257, 0.6400],
        [0.1390, 0.3265, 0.5345],
        [0.1131, 0.1410, 0.7460],
        [0.1255, 0.1720, 0.7025],
        [0.5404, 0.2123, 0.2474],
        [0.2614, 0.1547, 0.5839],
        [0.1108, 0.2661, 0.6231],
        [0.2740, 0.5310, 0.1950],
        [0.4037, 0.4409, 0.1554],
        [0.4216, 0.3036, 0.2748],
        [0.3662, 0.4111, 0.2227],
        [0.2905, 0.3627, 0.3468],
        [0.4199, 0.3905, 0.1896],
        [0.3147, 0.4022, 0.2831],
        [0.0679, 0.4461, 0.4860],
        [0.4626, 0.3893, 0.1481],
        [0.2820, 0.2740, 0.4440],
        [0.1357, 0.4958, 0.3685],
        [0.3127, 0.3608, 0.3265],
        [0.1133, 0.4883, 0.3984],
        [0.5509, 0.2065, 0.2427],
        [0.4897, 0.2503, 0.2600],
        [0.3307, 0.3273, 0.3420],
        [0.3240, 0.3916, 0.2845],
        [0.3231, 0.3780, 0.2989],
        [0.3432, 0.3237, 0.3330],
        [0.4884, 0.3474, 0.1641],
        [0.5631, 0.2738, 0.1631],
        [0.5893, 0.2300, 0.1807],
        [0.4896, 0.3055, 0.2049],
        [0.5909, 0.1964, 0.2127],
        [0.6702, 0.1892, 0.1406],
        [0.6463, 0.2503, 0.1034],
        [0.5417, 0.2788, 0.1795],
        [0.6265, 0.1317, 0.2418],
        [0.5659, 0.2330, 0.2011],
        [0.5585, 0.2196, 0.2219],
        [0.5261, 0.2368, 0.2372],
        [0.5130, 0.3245, 0.1625],
        [0.5143, 0.2330, 0.2527],
        [0.5599, 0.2803, 0.1599],
        [0.4950, 0.2720, 0.2330],
        [0.1622, 0.4041, 0.4337],
        [0.0657, 0.1530, 0.7813],
        [0.0958, 0.1759, 0.7284],
        [0.1118, 0.2687, 0.6195],
        [0.1091, 0.1464, 0.7445],
        [0.1825, 0.1561, 0.6614],
        [0.3152, 0.4059, 0.2790],
        [0.4205, 0.2142, 0.3653],
        [0.2550, 0.1960, 0.5490],
        [0.1398, 0.1676, 0.6926],
        [0.1700, 0.3000, 0.5299],
        [0.2498, 0.3015, 0.4487],
        [0.1751, 0.3386, 0.4863],
        [0.2178, 0.3072, 0.4750],
        [0.2930, 0.3454, 0.3616],
        [0.1958, 0.3048, 0.4994],
        [0.0915, 0.2518, 0.6567],
        [0.0869, 0.3601, 0.5530],
        [0.1731, 0.2242, 0.6027],
        [0.1215, 0.3419, 0.5366],
        [0.2647, 0.4314, 0.3039],
        [0.2356, 0.2804, 0.4840],
        [0.5912, 0.2001, 0.2087],
        [0.2100, 0.4641, 0.3259],
        [0.3197, 0.1226, 0.5576],
        [0.3270, 0.3100, 0.3630],
        [0.2113, 0.3594, 0.4294],
        [0.1066, 0.5459, 0.3475],
        [0.2906, 0.3300, 0.3794],
        [0.2202, 0.4021, 0.3778],
        [0.4159, 0.3649, 0.2192],
        [0.4589, 0.3579, 0.1832],
        [0.4132, 0.3629, 0.2239],
        [0.3421, 0.3751, 0.2828],
        [0.2033, 0.5526, 0.2441],
        [0.4033, 0.3428, 0.2539],
        [0.1534, 0.5207, 0.3259],
        [0.2199, 0.4417, 0.3384],
        [0.3458, 0.3454, 0.3088],
        [0.6700, 0.2137, 0.1163],
        [0.2025, 0.5648, 0.2326],
        [0.4837, 0.3586, 0.1576],
        [0.2214, 0.5758, 0.2028],
        [0.3667, 0.2508, 0.3826],
        [0.5541, 0.2276, 0.2183],
        [0.5982, 0.2540, 0.1478],
        [0.6925, 0.2271, 0.0804],
        [0.6154, 0.2649, 0.1197],
        [0.3965, 0.4254, 0.1782],
        [0.5471, 0.2479, 0.2050],
        [0.3269, 0.4585, 0.2146],
        [0.5222, 0.2517, 0.2262],
        [0.5243, 0.2763, 0.1994],
        [0.6539, 0.1892, 0.1569],
        [0.5649, 0.2794, 0.1556],
        [0.4917, 0.3588, 0.1495],
        [0.4938, 0.3234, 0.1828],
        [0.5583, 0.3439, 0.0978],
        [0.6054, 0.2617, 0.1329],
        [0.1654, 0.3239, 0.5107],
        [0.1162, 0.3741, 0.5097],
        [0.1499, 0.3483, 0.5018],
        [0.0559, 0.2396, 0.7045],
        [0.0580, 0.1826, 0.7594],
        [0.2093, 0.2104, 0.5803],
        [0.0694, 0.1377, 0.7929],
        [0.2661, 0.3161, 0.4178],
        [0.1218, 0.4007, 0.4775],
        [0.2489, 0.2842, 0.4669],
        [0.0841, 0.2571, 0.6587],
        [0.1578, 0.3089, 0.5333],
        [0.1298, 0.2578, 0.6124],
        [0.1754, 0.4049, 0.4198],
        [0.1112, 0.2427, 0.6460],
        [0.0803, 0.2301, 0.6896],
        [0.1129, 0.2639, 0.6232],
        [0.1048, 0.2187, 0.6766],
        [0.2397, 0.2204, 0.5399],
        [0.4781, 0.2306, 0.2914],
        [0.3983, 0.2330, 0.3687],
        [0.4420, 0.1774, 0.3805],
        [0.0504, 0.6588, 0.2908],
        [0.0943, 0.2004, 0.7053],
        [0.2053, 0.2881, 0.5066],
        [0.3612, 0.4278, 0.2110],
        [0.3271, 0.2765, 0.3964],
        [0.4628, 0.3177, 0.2194],
        [0.4002, 0.3164, 0.2834],
        [0.4116, 0.4166, 0.1718],
        [0.4423, 0.2452, 0.3126],
        [0.2794, 0.3106, 0.4100],
        [0.1898, 0.1683, 0.6419],
        [0.3336, 0.4805, 0.1859],
        [0.3560, 0.4151, 0.2289],
        [0.4069, 0.2740, 0.3191],
        [0.2737, 0.4081, 0.3182],
        [0.5027, 0.1579, 0.3394],
        [0.5095, 0.3020, 0.1885],
        [0.3032, 0.4361, 0.2607],
        [0.3373, 0.3522, 0.3104],
        [0.4813, 0.2865, 0.2321],
        [0.5112, 0.2612, 0.2276],
        [0.4762, 0.3325, 0.1913],
        [0.7010, 0.1952, 0.1038],
        [0.3195, 0.5398, 0.1408],
        [0.6263, 0.2561, 0.1177],
        [0.5859, 0.2120, 0.2021],
        [0.5718, 0.2575, 0.1707],
        [0.4330, 0.3260, 0.2409],
        [0.3487, 0.4196, 0.2316],
        [0.4931, 0.3205, 0.1864],
        [0.3201, 0.1430, 0.5369],
        [0.4685, 0.1663, 0.3652],
        [0.4695, 0.2965, 0.2341],
        [0.4802, 0.2748, 0.2451],
        [0.6594, 0.1797, 0.1609],
        [0.3652, 0.4104, 0.2244],
        [0.3637, 0.3827, 0.2536],
        [0.6490, 0.1426, 0.2084],
        [0.2527, 0.5272, 0.2200]], grad_fn=<SoftmaxBackward0>)
without softmax:
 tensor([[-3.0764e-02, -2.7117e-01,  9.4147e-01],
        [-2.6321e-02, -7.7035e-01,  1.1817e+00],
        [-8.0215e-01,  2.6736e-01,  3.1889e-01],
        [ 1.8316e-01, -1.4891e-02, -3.1790e-01],
        [ 9.1956e-02, -1.5874e-01,  1.9093e-02],
        [ 2.2648e-01, -7.2073e-01,  1.1573e+00],
        [-5.3583e-01, -3.0525e-01,  1.0383e+00],
        [ 2.1733e-01, -3.5595e-01, -4.3151e-02],
        [ 2.1816e-01, -2.0487e-01,  2.8422e-01],
        [-2.2424e-01,  6.9770e-02, -4.3430e-01],
        [-2.6522e-01,  2.9012e-01,  5.6323e-02],
        [ 2.6639e-01, -9.9333e-01,  9.7913e-01],
        [-1.6628e-01, -9.1990e-01,  5.9557e-01],
        [ 4.7437e-02, -5.2400e-01,  1.4183e-01],
        [-6.5146e-01, -4.9292e-01,  8.9355e-01],
        [-7.5565e-01, -1.5531e-01,  6.8887e-01],
        [-8.8245e-02, -1.5215e-01,  8.1013e-01],
        [ 6.7372e-01, -2.5280e-01,  3.4281e-01],
        [-9.5890e-01, -6.8643e-01,  1.3994e+00],
        [-5.2420e-01, -7.5960e-01,  1.3613e+00],
        [-5.6466e-01, -5.8068e-01, -7.6070e-01],
        [-1.0803e+00, -4.3300e-01,  1.0190e+00],
        [-1.2342e+00,  3.3867e-01,  2.5195e-01],
        [ 3.3639e-01, -1.1189e-01, -8.5725e-01],
        [-4.2305e-01, -2.1427e-01, -7.3404e-01],
        [ 5.7172e-01, -3.0672e-01, -3.1061e-01],
        [-7.6710e-02, -5.2523e-01,  6.5612e-01],
        [-2.7186e-01, -1.6291e-01, -1.2922e-01],
        [ 1.9961e-01, -3.3215e-01, -4.2794e-01],
        [ 1.0010e-01,  9.0640e-02, -4.6289e-01],
        [ 2.5760e-01, -3.7451e-01,  1.7064e+00],
        [ 5.0344e-01, -8.4581e-01,  2.9637e-01],
        [-4.7682e-01,  3.2281e-01,  6.3954e-01],
        [-4.8362e-01,  2.5065e-01, -7.6859e-01],
        [-1.1212e+00,  1.9761e-01, -2.4974e-01],
        [-1.0766e+00,  5.8007e-01,  1.2919e+00],
        [-2.6427e-01, -8.5105e-01,  4.1444e-01],
        [-4.6572e-01, -5.6632e-01,  1.9581e-01],
        [ 3.7678e-01, -2.5376e-01,  2.2111e-02],
        [-5.2469e-01, -5.0127e-01,  6.8259e-01],
        [-1.2359e+00,  9.4878e-01, -4.4240e-02],
        [-3.7319e-01, -9.8501e-02,  4.3920e-01],
        [-4.7408e-01,  2.4469e-01, -5.7665e-01],
        [-1.1567e-01, -4.9157e-01, -7.2761e-01],
        [-5.8867e-01, -3.2215e-01, -5.5062e-01],
        [-6.3884e-01,  3.7030e-01, -1.2559e+00],
        [-2.6120e-01, -4.0423e-01, -1.4898e-01],
        [-7.4336e-03, -2.1599e-01, -7.8804e-01],
        [-9.0954e-01,  7.8888e-01, -1.8492e+00],
        [-3.9644e-01, -6.0221e-01, -4.5884e-01],
        [-3.6429e-01, -1.0479e+00,  1.0487e+00],
        [-7.2890e-01,  1.1428e-01, -5.3678e-01],
        [-2.4893e-01, -2.0132e-01, -6.4898e-01],
        [ 3.1656e-01, -6.9800e-01, -1.8367e-01],
        [-1.0076e-01, -3.9116e-01, -1.1982e+00],
        [-3.1240e-01, -2.8760e-01, -5.3988e-01],
        [-4.5144e-01,  2.9507e-01, -1.0766e+00],
        [-1.5234e-01, -1.2521e-01, -3.5106e-01],
        [-7.4031e-01, -8.9709e-02, -3.4300e-01],
        [-5.0631e-01, -5.2546e-01,  1.5485e+00],
        [-3.4225e-01, -5.1336e-01,  9.9298e-01],
        [-3.7425e-01, -3.3321e-01,  4.9818e-01],
        [-5.8097e-01,  1.5231e-01,  9.2089e-01],
        [-1.8716e-02, -9.3853e-01,  1.1434e+00],
        [ 1.4688e-01, -1.1040e-01,  1.8235e-01],
        [ 8.6978e-01, -1.1718e+00,  1.5413e+00],
        [ 1.4249e-01, -5.7919e-01,  9.3540e-01],
        [ 5.2172e-01, -6.3413e-01,  1.3082e+00],
        [-1.9212e-02, -5.9719e-02,  2.5919e-01],
        [ 2.6680e-01, -5.3539e-01,  3.4712e-01],
        [-4.3362e-01, -1.0620e-01, -2.8569e-01],
        [-9.5821e-01, -1.3682e-02,  8.7881e-01],
        [-9.3458e-01,  1.5076e-01,  1.4152e-01],
        [-5.6701e-01,  5.1803e-02,  1.4916e-01],
        [-8.1386e-01, -5.7252e-02,  7.5290e-01],
        [-2.1843e-01, -1.5068e-02,  4.0822e-01],
        [ 1.6015e-01, -7.4711e-01,  7.8791e-01],
        [-4.0074e-01, -3.7875e-01,  4.0785e-01],
        [-1.8733e+00,  6.4454e-01, -1.7777e+00],
        [-7.5149e-01,  1.5554e-01, -8.3687e-01],
        [ 6.4042e-02, -7.3400e-01, -2.4262e-01],
        [-3.0925e-01,  1.2447e-02, -9.0970e-01],
        [-5.8949e-02, -2.6496e-01,  1.0210e+00],
        [-2.5974e-01,  3.5200e-02, -4.6381e-01],
        [-8.6202e-01,  2.5853e-01, -7.5583e-01],
        [-1.7228e-01, -1.3393e-01,  5.1956e-01],
        [-5.0010e-01, -9.5368e-02, -2.3388e-01],
        [-5.7284e-02,  3.1866e-03, -2.4974e-01],
        [ 9.9903e-01, -5.3892e-01, -2.6359e-01],
        [-7.9783e-02,  1.0025e-02, -6.5993e-01],
        [-6.3312e-02,  9.9151e-02, -6.8441e-01],
        [-5.9606e-01,  2.9051e-01, -2.4210e-02],
        [-2.3533e-01,  3.9898e-01, -1.1409e+00],
        [-6.1555e-01, -2.7053e-01,  2.5036e-01],
        [-2.1056e-01,  3.2477e-01, -6.0273e-01],
        [-8.8580e-01,  2.0897e-01,  1.7627e-01],
        [ 1.1171e-01, -6.9148e-01,  1.1620e+00],
        [ 6.0816e-01, -1.0514e+00, -2.2044e-01],
        [-2.6631e-01,  2.2565e-01, -5.8914e-01],
        [-1.3273e+00,  5.3975e-01, -1.4081e+00],
        [-4.9261e-01,  7.1011e-01, -7.0163e-01],
        [-1.0516e+00, -4.0657e-01,  8.0554e-01],
        [ 4.7099e-01, -7.7289e-01,  6.9894e-01],
        [-4.3653e-01, -3.9145e-01, -7.7952e-01],
        [-2.3245e-01,  2.7494e-02, -1.2552e+00],
        [ 2.6052e-02, -3.2265e-01, -8.0496e-01],
        [-1.0109e-01,  5.2838e-02, -1.3604e+00],
        [-7.4151e-01,  5.5174e-01, -9.3335e-01],
        [-4.8118e-01,  4.3776e-01, -8.9464e-01],
        [-5.5970e-01, -2.1392e-01, -4.1883e-01],
        [ 4.0064e-01, -6.7240e-01, -2.0762e-01],
        [-9.4588e-01, -3.5694e-01, -3.4384e-01],
        [ 4.1834e-02, -3.5698e-01, -5.9371e-01],
        [-4.2739e-01, -7.7389e-02, -9.7520e-01],
        [-2.4889e-01,  4.6589e-01, -1.2848e+00],
        [-3.7620e-01, -2.5001e-01, -1.2476e+00],
        [-4.7142e-01, -5.6643e-01, -1.1804e+00],
        [ 4.2643e-01, -4.1481e-01,  1.0979e+00],
        [ 8.6644e-02,  1.4167e-01, -4.0445e-04],
        [ 2.2267e-01, -4.6105e-01,  2.3837e-01],
        [ 2.9216e-01, -2.4321e-02,  6.3111e-01],
        [-8.6220e-02, -1.3538e-01,  1.0748e+00],
        [ 3.7380e-01, -9.6814e-01,  5.6064e-01],
        [ 1.6141e-01, -8.5712e-01,  9.3957e-01],
        [-5.0432e-01, -3.5743e-01,  5.0421e-01],
        [-3.8698e-02,  1.4717e-02,  7.8242e-02],
        [-2.5619e-01, -4.9277e-01,  3.4859e-01],
        [ 2.8969e-01, -5.2966e-01,  6.0884e-01],
        [-7.8393e-03, -2.8845e-01,  5.4739e-01],
        [-6.7990e-01, -4.1513e-01,  1.2595e+00],
        [-5.4829e-01,  4.2682e-01, -4.6170e-01],
        [-7.5644e-01,  4.1474e-01,  6.0494e-01],
        [ 4.7353e-02, -5.7331e-01,  6.8342e-01],
        [ 9.6442e-02, -4.6668e-01,  1.2434e+00],
        [-7.2147e-01, -2.9087e-01,  1.0687e+00],
        [-4.1684e-02, -8.0443e-01,  7.3385e-01],
        [-1.0819e+00, -4.7919e-01, -6.3783e-01],
        [ 2.4301e-04, -7.5823e-01,  6.8063e-01],
        [-1.0016e+00, -5.2223e-01, -1.0558e-01],
        [ 2.0148e-01,  7.3411e-02,  1.6667e-01],
        [ 6.8230e-01, -8.6852e-01,  1.8868e+00],
        [-5.0978e-01, -5.0132e-01,  8.9705e-01],
        [-4.3827e-01, -1.5741e-01, -1.0497e+00],
        [-3.7759e-01,  1.1756e-01,  5.6265e-01],
        [-2.7420e-01,  2.6707e-01, -5.9053e-01],
        [ 2.8451e-01, -1.8842e-01, -3.3714e-01],
        [ 4.6382e-01, -2.4786e-01, -5.5703e-01],
        [-8.9970e-01, -5.0665e-01,  3.9980e-01],
        [-8.7566e-01,  6.9765e-02, -7.7688e-02],
        [-1.1965e+00, -4.9575e-01,  1.1551e+00],
        [-1.9128e-01,  1.2539e-01, -5.5047e-01],
        [ 4.8545e-02,  4.2240e-02, -5.4811e-01],
        [-1.1119e+00,  8.1340e-02,  1.2484e-03],
        [-2.4097e-01, -1.3987e-01,  1.2275e-01],
        [ 4.7825e-01, -2.2257e+00,  2.5154e+00],
        [ 7.0679e-02, -8.6084e-02,  3.6654e-01],
        [-6.0250e-01,  3.6641e-01,  8.1368e-01],
        [-8.7746e-01, -1.0751e-01,  6.7349e-02],
        [-6.2644e-01, -6.3981e-01, -2.3315e-01],
        [-7.1203e-01, -2.5478e-01, -4.9937e-01],
        [-4.6506e-01, -2.8992e-01, -9.6900e-01],
        [-1.1832e-01,  1.6077e-04, -6.4003e-01],
        [-3.5065e-02, -3.2280e-01, -1.0850e+00],
        [-4.4566e-01, -8.7797e-02, -1.0156e+00],
        [-1.2831e-01, -9.1082e-01,  1.6750e-01],
        [-2.1741e-01, -1.0814e+00, -7.0757e-02],
        [-4.3465e-01, -2.5156e-01, -3.2314e-01],
        [-3.5445e-01, -4.1918e-01, -5.1622e-01],
        [-3.9679e-01,  2.6571e-01, -7.5474e-01],
        [-1.7058e-02, -7.9690e-01,  1.5253e+00],
        [-2.0850e-01, -8.1398e-01,  9.4595e-01],
        [-1.0814e-01, -2.0851e-01, -9.9072e-01],
        [ 5.0875e-01, -6.1449e-01, -7.5399e-02],
        [ 5.4461e-02, -6.6377e-01,  4.4719e-02],
        [-2.1860e-01, -3.1929e-01, -6.0614e-01],
        [-6.8488e-02, -6.1434e-01, -2.0689e-01],
        [-2.9661e-01, -3.0892e-01,  2.0547e-02],
        [-4.7056e-02,  1.3177e-01, -7.8308e-01]], grad_fn=<AddmmBackward0>)
```

## (a)

Softmax is a function that calculates multiple class probability. It is similar to sigmoid, but the problem with sigmoid (and in this case, linear regression), is that you only receive scalars as a raw probability of being in class 0 or class 1. You can't use raw output with multiple classes - we see that we have unnormalized confidence scores when we do not use Softmax (this doesn't make sense). Thus, softmax helps normalize the probabilities for each sample such that they add to 1. Therefore, we can see the probability of each sample of being in class 0, 1, or 2.

## (b)

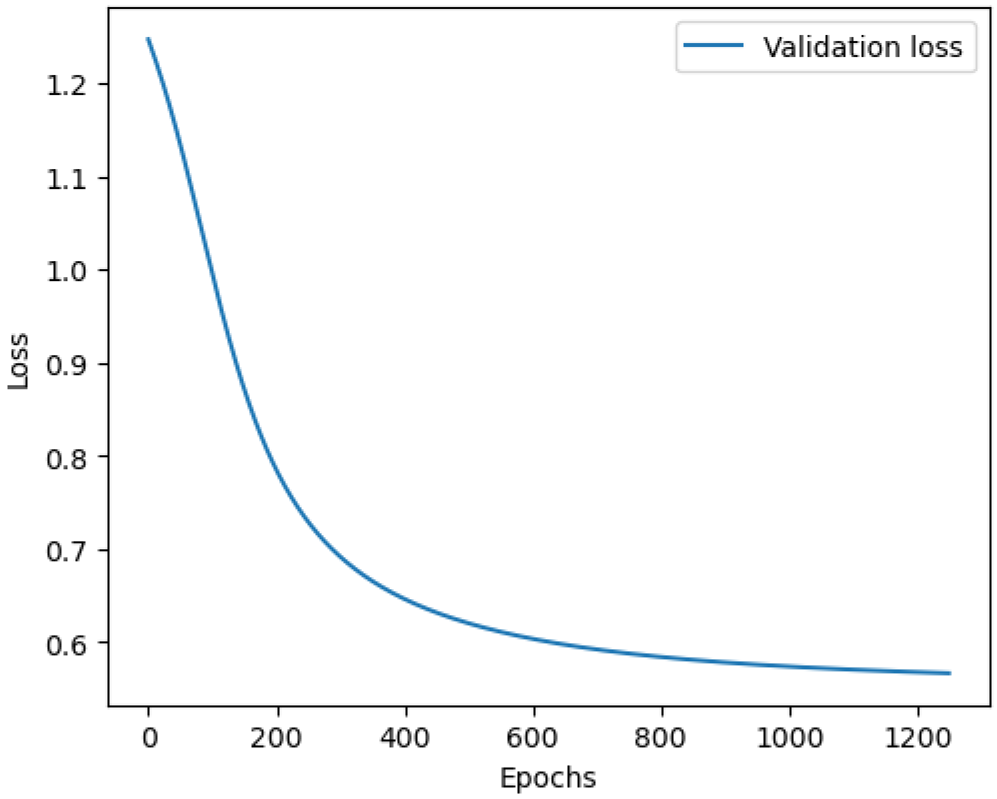
*For debugging*: The accuracy could reach over 95% if the hyperparamters are tuned properly.

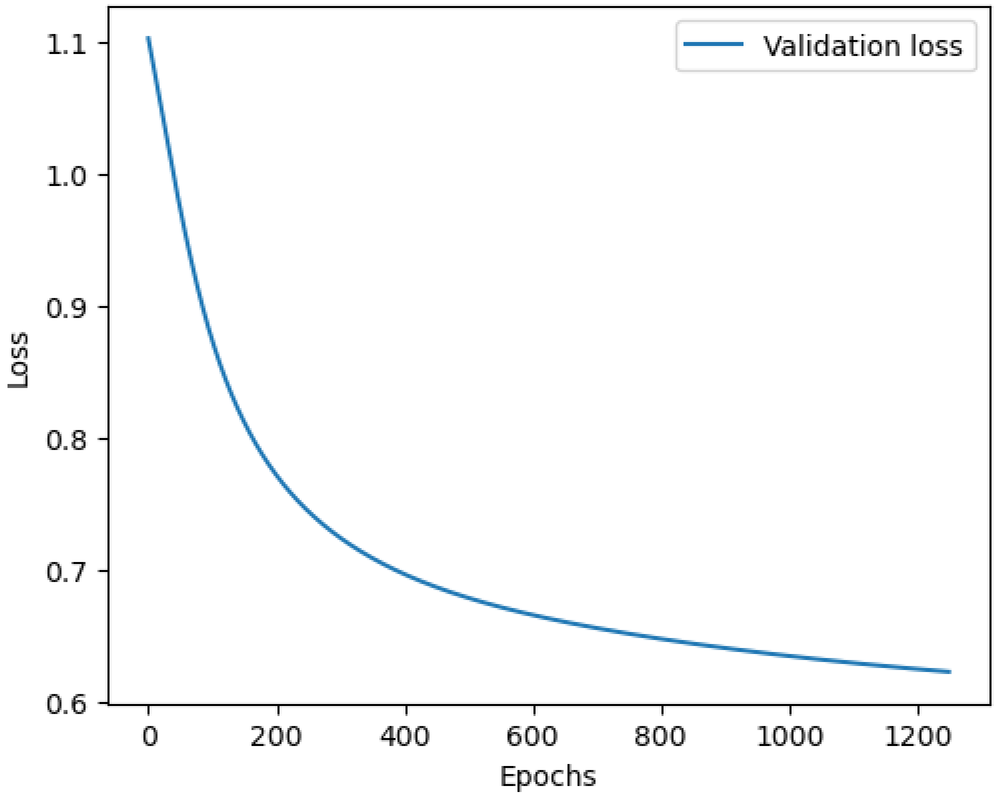
## I changed the number of epochs from 500 -> 1250 to achieve ~95% accuracy most of the time.

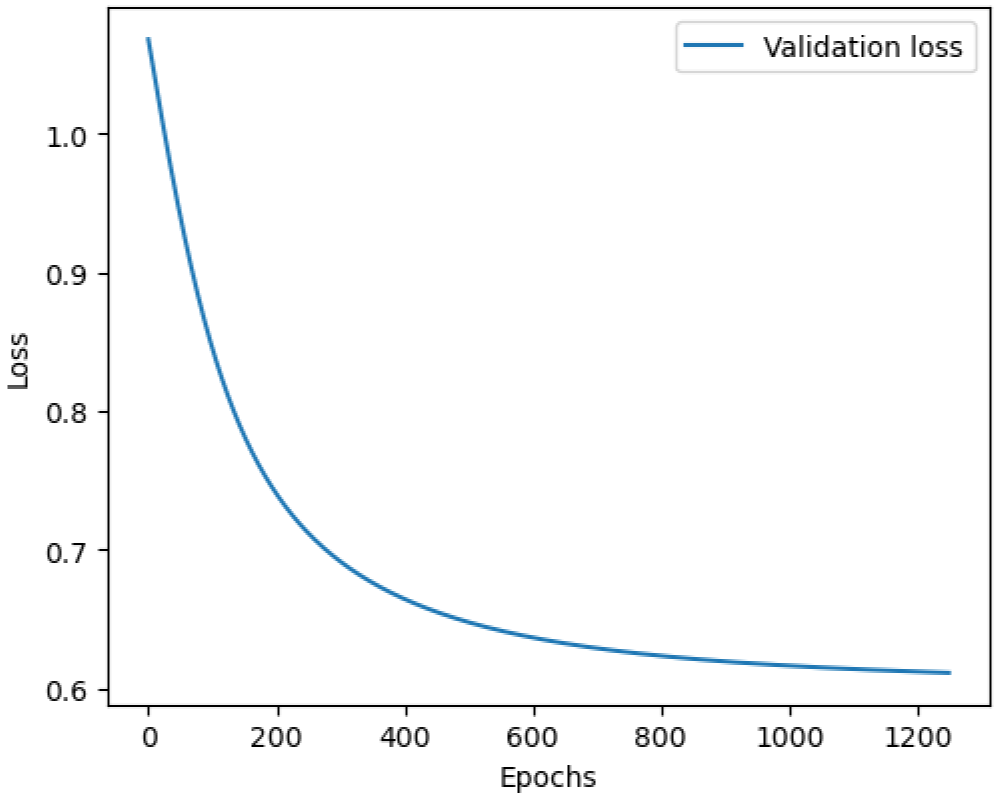
In [ ]:
```python
def train_and_val(model, X_train, y_train, epochs, draw_curve=True):
    """
    Further split the data into acutal train and validation subsets.
    """
    # Define your loss function, optimizer
    loss_func = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    X_train_in, X_val, y_train_in, y_val = train_test_split(X_train, y_train, test_

    val_losses = []
    lowest_val_loss = np.inf

    weights = {}
    for i in range(epochs):
        # Compute the loss and do back-propagation
        y_train_pred = model(X_train_in)
        loss = loss_func(y_train_pred, y_train_in)

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

        # Compute validation loss and keep track of the lowest val loss
        with torch.no_grad():
            val_loss = loss_func(model(X_val), y_val)

        if val_loss < lowest_val_loss:
            lowest_val_loss = val_loss
            weights = model.state_dict()
        val_losses.append(val_loss)

    # The final number of epochs is when the minimum error in validation set occurs
    final_epochs = np.argmin(val_losses)
    print("Number of epochs with lowest validation:", final_epochs)
    print(f"Validation loss: {np.min(val_losses)}")
    # Recover the model weights
    model.load_state_dict(weights)

    if draw_curve:
        fig, ax = plt.subplots(1, 1, figsize=(5, 4), constrained_layout=True)
        ax.plot(np.arange(epochs), val_losses, label='Validation loss')
        ax.set_xlabel('Epochs')
        ax.set_ylabel('Loss')
        ax.legend()

    return model


def calculate_accuracy_nn(model, X, y):
    with torch.no_grad():
        y_pred = torch.argmax(model(X), axis=1)
        acc = torch.sum(y_pred == y) / len(y)
    return acc.detach().numpy()


def KFoldNN(k, X, y, epochs=500):
    """
    K-Fold Validation for Neural Network

    Parameters
    ---------
    k: int
        Number of folds
    X: numpy.ndarray
        Input data, shape (n_samples, n_features)
    y: numpy.ndarray
        Class labels, shape (n_samples)
    epochs: int
        Number of epochs during training
    """
    # K-Fold
    kf = KFold(n_splits=k, shuffle=True)
    train_acc_all = []
    test_acc_all = []
    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        # further do a train/valid split on X_train
        model = NeuralNetwork(X_train.shape[1], len(torch.unique(y)))
        model = train_and_val(model, X_train, y_train, epochs)
        # Report prediction accuracy for this fold
        # use calculate_accuracy_nn() function
        train_acc = calculate_accuracy_nn(model, X_train, y_train)
        train_acc_all.append(train_acc)
        test_acc = calculate_accuracy_nn(model, X_test, y_test)
        test_acc_all.append(test_acc)
        print("Train accuracy:", train_acc)
        print("Test accuracy:", test_acc)

    # report mean & std for the training/testing accuracy
    print("Final results:")
    print("Training accuracy:", np.mean(train_acc_all))
    print("Training std:", np.std(train_acc_all))
    print("Testing  accuracy:", np.mean(test_acc_all))
    print("Testing std:", np.std(test_acc_all))
```

In [ ]:
```
KFoldNN(3, X, y, epochs=1250)
```

```
Number of epochs with lowest validation: 1249
Validation loss: 0.5666986107826233
Train accuracy: 1.0
Test accuracy: 0.98333335
Number of epochs with lowest validation: 1249
Validation loss: 0.6230376362800598
Train accuracy: 0.99159664
Test accuracy: 0.9322034
Number of epochs with lowest validation: 1249
Validation loss: 0.6113277077674866
Train accuracy: 0.9831933
Test accuracy: 0.9661017
Final results:
Training accuracy: 0.9915967
Training std: 0.006861315
Testing  accuracy: 0.9605462
Testing std: 0.021240147
```


Validation loss
Loss
Epochs
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0
200
400
600
800
1000
1200


Validation loss
Loss
Epochs
1.1
1.0
0.9
0.8
0.7
0.6
0
200
400
600
800
1000
1200



## (b)

The prediction is almost equivalent to the Gaussian Naive Bayes. I am using the test data ("unseen" data) as the metric of comparison here. I suppose that the data is relatively linear, meaning that having a linear activation function in the first layer of the NN is the same as generating a Gaussian PDF. The data is also quite low-dimensional, meaning that a Gaussian NB will still perform as well as an NN.