Question 1

1.a

P[-|M] = 0.05, or the probability that a person has the marker but will test negative equals 5%. This is because this is the complement/opposite of someone having the marker but testing positive. This information was given to us, P[+|M] = 0.95. Together, P[+|M] + P[-|M] = 1, due to both of them being probabilities. Thus, **P[-|M] = 0.05**

Similarly, we are given the information P[-|Not M] = 0.95. P[+|Not M] + P[-|Not M] = 1, due to both of them being probabilities and defining the entire population. Therefore, P[+|Not M] = 1 - P[-|Not M] = 0.05. **P[+|Not M] = 0.05**

Finally, P[M] = 0.01 is given. Since P[M] + P[Not M] = 1, P[Not M] = 1 - P[M] = .99. **P[Not M] = 0.99**

1.b

Bayes Theorem states that $P(A|B) = \frac{P(B|A)*P(A)}{P(B)}$. In other words, the probability of A being true given B being true is equal to the probability of B being true given A being true, times the probability of A being true, divided by the probability of B being true.

We are asked to find the probability that a person who tested positive (+) actually has the marker (M). Therefore, we are trying to find $P[M|+]$. Following Bayes' Theorem:

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

We were given the values for P[+|M] and P[M], however we still need to solve for P[+]. We can do this by summing up all the ways to get a positive reading (P[+|M] and P[+|Not M]), multiplyling by their respective probabilities of assumed conditions (P[M] and P[Not M]). Thus,

$$P[+] = P[+|M]*P[M] + P[+|NotM]*P[NotM]$$

$$P[+] = .95*.01 + .05*.99$$

$$P[+] = .059$$

Finally, we have all our values to solve the original probability, of whether a person who tested positive (+) actually has the marker (M).

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

$$P[M|+] = \frac{.95 * .01}{.059}$$

$$P[M|+] = 0.16101694915$$

So there is only a 16.1% chance that, after one positive test result, the randomly selected person actually has the marker. Due to the test being relatively inaccurate (having a 5% chance of giving a false positive), along with the probability of actually having the marker being very low (only a 1% chance of actually randomly having the marker), these two factors lead to the low chance that after one positive test, the random person actually has the marker. Since the probability of the marker's prevalence rate is factored into Bayes' Theorem more times than the false positive rate (it is multiplied as P(A) and is also utilized in calculating P(+), the marker's low prevalence is the more dominating factor in terms of why the chance is so low.

1.c

If the frequency of the marker increases to P[M] = .1, then surely the chance that P[M|+] would increase. We can recalculate using the updated marker frequency and Bayes' Theorem. We would also need to recalculate P[Not M] and P[+], since those are dependent on the P[M] as well.

Solving for P[Not M]:

$$P[NotM] = 1 - P[M]$$

$$P[NotM] = 1 - .1$$

$$P[NotM] = .9$$

Solving for P[+]:

$$P[+] = P[+|M] * P[M] + P[+|NotM] * P[NotM]$$

$$P[+] = .95 * .1 + .05 * .9$$

$$P[+] = .14$$

Plugging in the new values into Bayes' Theorem:

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

$$P[M|+] = \frac{.95 * .1}{.14}$$

$$P[M|+] = 0.67857142857$$

Now the percentage has increased to 67.86%

Question 2

2.a

The code for the gaussian function is below. I implemented it using the numpy functions for root, pi, exponent, and square because x is likely to be a vector or array due to the large number of features and data points in our wine dataset. We use a Gaussian function to model the probability because we can presume that most variables in the world are normally distributed.

In [1]:
```
import numpy as np, pandas as pd, matplotlib.pyplot as plt

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):
        """
        f(x) = 1 / (sigma*root(2pi))*e^(-1/2*((x-mu)/sigma)^2
        """
        return 1 / (std * np.sqrt(2 * np.pi)) * np.exp((-1/2) * ((x - mean)

    @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, (mean, std) in enumerate(stats):
            this_prob = NaiveBayesClassifier.gaussian(x_input[i], mean, std)
            init_prob *= this_prob
            # print(f"This prob is {this_prob}, new init_prob is {init_prob}

        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 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
            for y_type in self.type_stats:
                prob = self.calculate_prob(xs, self.type_stats[y_type]) * (t
                # print(f"Probability for rank {y_type} is {prob}")
                if prob>max_prob:
                    max_prob=prob
                    guess=y_type
            return guess
        else:
            print("Please train the classifier first!")
```

```
/var/folders/k8/mg372j_55z30k1z4y_8mb0w00000gn/T/ipykernel_21276/3183466931.
py:1: DeprecationWarning:
Pyarrow will become a required dependency of pandas in the next major releas
e of pandas (pandas 3.0),
(to allow more performant data types, such as the Arrow string type, and bet
ter interoperability with other libraries)
but was not found to be installed on your system.
If this would cause problems for you,
please provide us feedback at https://github.com/pandas-dev/pandas/issues/54
466

  import numpy as np, pandas as pd, matplotlib.pyplot as plt
```

In [2]:
```
# NaiveBayesClassifier.gaussian(20, 20, 1)
```

In [3]:
```
wines = pd.read_csv("wines.csv")
wines.drop(labels='Start assignment', axis=1, inplace=True)
wines.head()
```

Out[3]:

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols.1	Proantho-cyanins	in
0	14.23	1.71	2.43	15.6	127	2.8	3.06	0.28	2.29	
1	13.24	2.59	2.87	21.0	118	2.8	2.69	0.39	1.82	
2	14.83	1.64	2.17	14.0	97	2.8	2.98	0.29	1.98	
3	14.12	1.48	2.32	16.8	95	2.2	2.43	0.26	1.57	
4	13.75	1.73	2.41	16.0	89	2.6	2.76	0.29	1.81	

In [4]:

```
"""
.fit(self, xs, ys) function debugging
"""

# typefilter for ys
ys = wines['ranking']
type_indices = {} # is a dictionary
type_stats = {}
for y in set(ys):
    type_filter = (ys==y)
    type_indices[y] = type_filter
    # type_stats[y] = NaiveBayesClassifier.calculate_statistics(xs[type_filt
# type_filter is a boolean array corresponding to Trues if the type is the c
# Stores the filter in the type_indices dictionary, then calculates statisti
# each of the three types, and stores in type.stats. Filters the xs to only
# passing in a boolean array to xs to filter it

print(type_indices[1].sum())
print(type_indices[2].sum())
print(type_indices[3].sum())
assert(type_indices[1].sum() + type_indices[2].sum() + type_indices[3].sum()
```

```
59
71
48
```

In [39]:

```
wine_features = wines.iloc[:, :-1]
wine_features_np = wine_features.to_numpy()
display(wine_features.shape)
display(wine_features.head())

wine_rankings = wines.iloc[:, -1]
wine_rankings_np = wine_rankings.to_numpy()
display(wine_rankings.shape)
wine_rankings.head()
```

```
(178, 13)
```

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols.1	Proantho-cyanins	inte
0	14.23	1.71	2.43	15.6	127	2.8	3.06	0.28	2.29	
1	13.24	2.59	2.87	21.0	118	2.8	2.69	0.39	1.82	
2	14.83	1.64	2.17	14.0	97	2.8	2.98	0.29	1.98	
3	14.12	1.48	2.32	16.8	95	2.2	2.43	0.26	1.57	
4	13.75	1.73	2.41	16.0	89	2.6	2.76	0.29	1.81	

```
(178,)
```

Out[39]:
```
0    1
1    1
2    1
3    1
4    1
Name: ranking, dtype: int64
```

In [6]:
```python
# Create the WineClassifier, fit to the features and rankings numpy array
WineClassifier = NaiveBayesClassifier()
WineClassifier.fit(xs=wine_features_np, ys=wine_rankings_np)
WineClassifier.type_stats
display(len(WineClassifier.type_stats[3]))
```

```
13
```

Test the WineClassifier for different data points, see if we can get all the different rankings

In [7]:
```python
display(WineClassifier.predict(wine_features.iloc[0, :].to_numpy()))
display(WineClassifier.predict(wine_features.iloc[-1, :].to_numpy()))
display(WineClassifier.predict(wine_features.iloc[len(wine_features) // 2, :
```

```
1
2
2
```

In [8]:
```python
# Test for rankings == 3
display(wine_rankings[wine_rankings == 3].head(1))
display(WineClassifier.predict(wine_features.iloc[43, :].to_numpy()))
```

```
43    3
Name: ranking, dtype: int64
3
```

To calculate the probability that, given a wine is in Cultivar 1, it has an alcohol content of 13% would be equivalent to asking $P[13\%|\text{Cultivar 1}]$. To do this, we would use our PDF, the Gaussian, with our x-value as 13 and our mean and standard deviation as the statistics determined for the mean/std deviation for cultivar 1, across all confirmed cultivar 1 values. The code for it is below.

In [9]:
```python
c1_abv_mean, c1_abv_sd = WineClassifier.type_stats[1][0] # Find the mu and s
```

```
WineClassifier.gaussian(13, c1_abv_mean, c1_abv_sd)
```

Out[9]:
```
0.23236757865410357
```

So $P[13\%|\text{Cultivar 1}] = 0.23236757865410357$

2.b

In [30]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold

scaler = StandardScaler()
wine_features_norm = scaler.fit_transform(wine_features_np)
display(wine_features_norm)
display(wine_features_np)
```

```
array([[ 1.51861254, -0.5622498 ,  0.23205254, ...,  0.36217728,
         1.84791957,  1.01300893],
       [ 0.29570023,  0.22769377,  1.84040254, ...,  0.36217728,
         0.44960118, -0.03787401],
       [ 2.25977152, -0.62508622, -0.7183361 , ...,  0.53767082,
         0.33660575,  0.94931905],
       ...,
       [ 0.20923168,  0.22769377,  0.01273209, ..., -1.56825176,
        -1.40069891,  0.29649784],
       [ 1.39508604,  1.58316512,  1.36520822, ..., -1.52437837,
        -1.42894777, -0.59516041],
       [-0.92721209, -0.54429654, -0.90110314, ...,  0.18668373,
         0.78858745, -0.7543851 ]])
array([[1.423e+01, 1.710e+00, 2.430e+00, ..., 1.040e+00, 3.920e+00,
        1.065e+03],
       [1.324e+01, 2.590e+00, 2.870e+00, ..., 1.040e+00, 2.930e+00,
        7.350e+02],
       [1.483e+01, 1.640e+00, 2.170e+00, ..., 1.080e+00, 2.850e+00,
        1.045e+03],
       ...,
       [1.317e+01, 2.590e+00, 2.370e+00, ..., 6.000e-01, 1.620e+00,
        8.400e+02],
       [1.413e+01, 4.100e+00, 2.740e+00, ..., 6.100e-01, 1.600e+00,
        5.600e+02],
       [1.225e+01, 1.730e+00, 2.120e+00, ..., 1.000e+00, 3.170e+00,
        5.100e+02]])
```

In [33]:
```
# A performance tester:
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)

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=3)
    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]

        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(f"Training accuracy mean: {np.mean(train_acc_all)}, std: {np.std(t
    print(f"Testing  accuracy mean: {np.mean(test_acc_all)}, std: {np.std(te

KFoldNaiveBayes(5, wine_features_norm, wine_rankings_np)
```

```
Train accuracy: 0.9830508474576272
Test accuracy: 1.0
Train accuracy: 0.9915966386554622
Test accuracy: 0.9491525423728814
Train accuracy: 0.9915966386554622
Test accuracy: 0.9661016949152542
Final results:
Training accuracy mean: 0.9887480415895172, std: 0.0040285246043956485
Testing  accuracy mean: 0.9717514124293786, std: 0.021139307269909268
```

Question 3

Hint: cross entropy loss of pytorch (https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html) works by taking in y_hat(your model prediction) as a tensor of size (Batchsize, number of class) and y(reference values) as a tensor of size(Batchsize,). You can think of each element of the y_hat tensor(X_ij) as the probability that the element i belong to class j. And each element of y should be a int/torch.long that is a number between [0,number_of_class-1]

You can save a model and reload it by calling:

In [178…
```
import torch
import torch.nn as nn

class WineNet(nn.Module):
    def __init__(self):
        super().__init__()
        # create a NN with no hidden layer, just a linear layer plus a Softm
        self.layers = nn.Linear(13, 3, dtype=torch.float32)
        # self.layers = nn.Linear(13, 3, dtype=torch.float64)

    def forward(self, X):
        return self.layers(X)

    def forward_softmax(self, X):
        Softmax = nn.Softmax(dim=1)
        return Softmax(self.layers(X))


model = WineNet()
```

In [179…
```
wine_features_tensor = torch.from_numpy(wine_features_norm).float()
# display(wine_features_tensor)
model.forward(wine_features_tensor)
```

```
Out[179…  tensor([[-1.1029e-01,  1.0260e-01,  7.2270e-01],
                   [-7.2586e-01,  5.8687e-01,  2.3956e-01],
                   [ 2.8051e-01, -4.5739e-01, -1.0251e-01],
                   [ 1.0230e-01, -2.2169e-01, -2.8597e-01],
                   [-2.8478e-01, -6.1923e-02, -1.4116e-01],
                   [-2.4039e-01,  1.7564e-01,  1.0009e-01],
                   [-1.0671e-01,  3.1025e-01,  7.2162e-01],
                   [-5.8297e-01,  6.0141e-01,  2.6119e-01],
                   [-7.6119e-01,  5.8831e-01,  3.2545e-01],
                   [ 2.3678e-01,  1.3614e-02,  1.4725e-01],
                   [-2.6121e-02, -1.7209e-01,  3.8673e-01],
                   [-1.0433e-01,  9.2290e-01,  2.5054e-01],
                   [-4.0119e-01,  4.9359e-01, -4.7397e-01],
                   [-3.3989e-01,  3.4648e-01, -2.0866e-01],
                   [-4.9832e-01,  4.0147e-01,  1.8943e-01],
                   [-3.2379e-02,  3.3601e-01,  1.0996e+00],
                   [-2.2997e-01, -2.2081e-01,  3.3548e-01],
                   [-8.2327e-01,  1.5549e-02,  7.9489e-03],
                   [-2.9901e-01,  6.1062e-01,  5.0214e-01],
                   [-3.3536e-01,  3.9782e-01,  4.6831e-01],
                   [ 7.4922e-01,  4.8873e-01, -6.4152e-01],
                   [-9.8844e-01,  1.1183e+00,  6.8815e-01],
                   [-1.4163e-01,  1.5072e-01,  9.1534e-01],
                   [ 6.9334e-01, -4.3413e-01,  8.7065e-02],
                   [ 3.8436e-01,  1.1642e-01,  1.5216e-01],
                   [-8.5170e-01,  8.8828e-01,  1.7368e-01],
                   [-6.9258e-01,  3.9379e-01,  7.4078e-01],
                   [-2.5956e-01,  6.0804e-01,  2.6791e-01],
                   [-4.1228e-01,  5.6952e-01,  4.3729e-01],
                   [-5.7914e-01,  8.1333e-01,  7.1076e-01],
                   [-1.7039e-01,  2.9496e-01,  1.6918e+00],
                   [-2.8849e-01,  2.3801e-01,  2.8016e-02],
                   [-1.3656e+00,  3.2952e-01,  7.1246e-01],
                   [-4.9335e-03,  3.7557e-01,  6.8430e-01],
                   [ 1.2856e-01,  4.3657e-01,  8.0690e-01],
                   [-7.6131e-01, -1.8992e-01,  2.2217e+00],
                   [-6.3341e-01,  1.1392e+00,  4.0259e-01],
                   [-5.0615e-01,  1.1879e+00,  6.2764e-01],
                   [-1.4546e+00,  1.3863e+00,  7.8758e-01],
                   [-8.5105e-01,  9.6534e-01,  1.2403e+00],
                   [-5.4621e-01, -6.0678e-01,  7.3121e-01],
                   [-4.1815e-01,  6.1502e-01,  1.0589e+00],
                   [-3.6264e-01, -1.3797e-01,  2.7741e-01],
                   [ 9.1606e-01, -6.6005e-01, -4.7068e-01],
                   [ 5.4183e-01, -6.7001e-01, -6.9870e-01],
                   [ 1.2263e+00, -1.1869e+00, -6.0793e-01],
                   [ 3.1955e-01,  6.1971e-02, -1.9098e-01],
                   [ 5.5759e-01, -4.5494e-01, -8.8608e-01],
                   [ 1.3861e+00, -2.0848e+00, -1.3016e+00],
                   [ 1.2132e+00, -1.3158e+00, -4.5646e-01],
                   [ 4.1735e-01, -2.8074e-01, -1.0114e-01],
                   [ 1.1142e+00, -1.8151e+00, -6.3640e-01],
                   [ 8.4946e-01, -7.6997e-01, -9.3110e-01],
                   [ 7.3110e-01, -1.2075e+00, -6.2326e-01],
                   [ 1.0803e+00, -1.3142e+00, -7.0158e-01],
                   [ 1.3321e+00, -1.4881e+00, -8.0904e-01],
                   [ 1.0081e+00, -1.7877e+00, -8.4823e-01],
                   [ 7.7735e-01, -1.1993e+00, -3.9888e-01],
                   [ 1.4941e-01,  1.6848e-01,  4.3125e-02],
                   [-2.5904e-01,  2.5968e-01,  7.2894e-01],
                   [-4.1312e-02,  1.3303e-01,  2.2286e-01],
                   [-1.2511e-01,  1.8506e-01,  4.4836e-01],
                   [-3.8283e-01, -4.5892e-01, -1.4280e-01],
                   [-2.4409e-01,  5.8365e-01,  2.1942e-01],
                   [-9.2628e-01, -6.8407e-02,  1.6257e-01],
                   [-1.1905e+00,  1.1516e+00,  4.7334e-01],
                   [-8.6777e-01,  7.1774e-01,  5.3820e-02],
                   [-9.5811e-01,  5.2456e-01,  4.3034e-01],
                   [-1.9038e-01,  9.6591e-02,  1.8196e-01],
                   [-3.1136e-01,  3.1945e-01, -7.2219e-02],
                   [ 6.4072e-02,  2.6817e-01,  6.1893e-02],
                   [-2.2807e-01, -4.5846e-01,  8.7153e-02],
                   [-7.8373e-02, -2.8691e-01,  3.2971e-01],
                   [ 4.4861e-02, -1.8535e-01,  1.0306e-01],
                   [-2.9134e-02,  3.4874e-02,  7.5474e-01],
                   [-4.2163e-01, -4.7434e-01,  5.3091e-01],
                   [-2.5444e-01,  3.7136e-01,  2.3880e-02],
                   [ 3.0186e-01,  8.4869e-02,  4.4675e-01],
                   [ 1.5710e+00, -3.6120e-01, -3.8663e-01],
                   [ 6.8944e-01,  9.1385e-02, -3.0221e-01],
                   [-4.0394e-01,  7.7290e-01, -5.0199e-01],
                   [-6.6528e-03,  5.4986e-01,  5.5081e-01],
                   [-1.4038e+00,  9.2253e-01,  1.0344e-01],
                   [ 1.7982e-01,  1.5168e-01,  1.0677e-01],
                   [ 5.1749e-01, -1.8229e-01,  7.5268e-02],
                   [ 5.3170e-01, -2.7711e-01,  1.0510e+00],
                   [-5.9807e-01,  1.0163e+00,  5.4582e-01],
                   [-4.9466e-01,  2.6715e-01,  3.7690e-01],
                   [-1.1698e+00,  1.1979e+00,  4.4765e-01],
                   [-6.3298e-02, -4.4111e-02,  1.4144e-02],
                   [ 4.3487e-01, -3.1084e-01,  1.0628e-01],
                   [-8.9807e-01,  2.5984e-01,  4.6279e-01],
                   [ 4.7454e-01, -1.7858e-01,  2.8679e-01],
                   [-3.6589e-01,  6.6492e-01,  7.0977e-01],
                   [-1.1694e-01,  4.1882e-01,  9.2041e-01],
                   [ 1.0076e-01,  2.6747e-01,  1.2533e+00],
                   [-1.3920e+00,  1.0179e+00,  1.0549e+00],
                   [-6.5022e-01,  7.1965e-01, -3.8963e-01],
                   [-1.2016e-01,  4.8644e-01,  1.0358e+00],
                   [ 1.0020e+00, -9.0582e-01, -3.6298e-01],
                   [-2.1541e-01, -1.9197e-01,  8.0752e-01],
                   [-2.2736e-01,  6.8573e-01,  1.1015e+00],
                   [-8.6730e-01,  9.9389e-01,  8.7300e-01],
                   [ 8.2740e-01, -8.3689e-01, -8.1704e-01],
                   [ 5.4921e-01, -9.8874e-01, -8.5394e-01],
                   [ 6.1404e-01, -4.3378e-01, -5.7791e-01],
                   [ 1.0624e+00, -8.4201e-01, -5.3640e-01],
                   [ 4.8720e-01, -1.0961e+00, -7.5322e-01],
                   [ 1.2299e+00, -1.2379e+00, -3.4368e-02],
                   [ 1.1018e+00, -1.1935e+00, -3.2684e-01],
                   [ 6.2963e-01, -3.2690e-01,  1.9106e-01],
                   [ 7.3678e-01, -7.2806e-01, -1.8483e-01],
                   [ 6.6924e-01, -4.6423e-01, -3.2953e-01],
                   [ 1.1158e+00, -1.0862e+00, -5.8445e-01],
                   [ 1.0813e+00, -1.5976e+00, -6.3979e-01],
                   [ 1.0230e+00, -8.2184e-01, -4.9496e-01],
                   [ 1.5468e+00, -1.1142e+00, -6.3348e-01],
                   [-1.0192e+00,  8.5938e-02,  6.5849e-01],
                   [-4.0644e-02, -3.9052e-01,  1.0968e-03],
                   [-8.3805e-04,  3.2370e-01,  6.1145e-02],
                   [-5.6894e-01,  3.1189e-02,  4.7308e-01],
                   [-4.8858e-01, -4.1574e-01,  2.1523e-01],
                   [-4.6555e-01,  4.7193e-01, -3.2839e-01],
                   [-3.0009e-01,  3.7930e-01,  1.1297e-01],
                   [-3.7930e-01, -1.4020e-02, -5.1493e-02],
                   [-3.9442e-01,  3.8296e-01,  4.6170e-01],
                   [-3.1658e-01,  4.1297e-01,  6.8454e-02],
                   [-2.8402e-01,  2.5826e-01,  4.7736e-01],
                   [-4.9242e-01,  4.5613e-01,  6.4780e-01],
                   [-1.4484e-01,  2.8619e-01,  9.4960e-01],
                   [-4.9816e-02, -3.4141e-01,  2.8755e-01],
                   [-4.1926e-01, -3.0528e-01,  3.5336e-01],
                   [-4.0468e-02,  6.3454e-02,  5.1914e-01],
                   [-4.2829e-01,  1.7915e-01,  8.1664e-01],
                   [ 1.1685e-01,  8.6790e-02,  6.0048e-01],
                   [-5.7379e-01,  4.8567e-01, -1.9810e-02],
                   [ 1.2315e+00, -4.2665e-02, -3.0934e-01],
                   [-6.6447e-01,  4.8947e-01,  1.2438e-01],
                   [ 9.7847e-01,  4.7502e-01, -4.4506e-01],
                   [ 3.2557e-01,  2.2371e-02,  1.2955e+00],
                   [-1.2565e+00,  1.8059e+00,  1.5880e+00],
                   [-7.7867e-01,  1.2263e+00,  1.0423e+00],
                   [ 3.0013e-01,  4.7328e-02, -8.1232e-02],
                   [-1.0091e+00,  5.3453e-01,  7.4255e-01],
                   [ 3.9826e-01, -8.0865e-01,  5.1993e-02],
                   [-4.5619e-01,  4.9499e-01,  1.6280e-01],
                   [-1.2204e-01,  1.5153e-01,  3.2428e-01],
                   [-8.8467e-01,  1.0501e+00,  3.8301e-01],
                   [-4.0186e-01,  4.5333e-01,  3.8614e-01],
                   [-7.2666e-01,  1.1072e+00,  1.0299e+00],
                   [ 2.1720e-01,  2.1395e-01,  8.8879e-01],
                   [ 2.8347e-01, -1.9673e-01,  3.7612e-01],
                   [-2.5863e-01,  5.0066e-01,  4.6757e-01],
                   [-1.4618e-01,  6.1717e-01,  8.6551e-01],
                   [-1.9406e+00,  2.4725e+00,  1.6705e+00],
                   [-9.8742e-01,  3.8218e-01,  6.4504e-01],
                   [-1.5450e+00,  6.5301e-02,  1.3827e+00],
                   [-3.5991e-01,  8.1729e-01,  9.9629e-01],
                   [ 1.1247e+00, -4.1395e-01, -5.2273e-01],
                   [ 1.1028e+00, -5.0212e-01, -1.7787e-01],
                   [ 1.4010e+00, -8.1900e-01, -2.6039e-01],
                   [ 1.8567e-01, -7.8566e-01, -3.9485e-01],
                   [ 1.0550e+00, -1.1905e+00, -1.7709e-01],
                   [ 8.4066e-01, -1.2324e+00, -5.7564e-01],
                   [ 9.2758e-01, -1.0143e+00, -5.3258e-01],
                   [ 1.1009e+00, -1.0509e+00, -3.0520e-01],
                   [ 1.4156e+00, -1.6791e+00, -1.1323e-01],
                   [ 1.6575e+00, -8.6186e-01,  2.9070e-01],
                   [ 1.0220e+00, -1.7001e+00, -2.3801e-01],
                   [ 6.3857e-01, -1.0620e+00,  4.3835e-01],
                   [ 6.2901e-01, -9.2963e-01,  1.7054e-01],
                   [ 1.3519e+00, -1.4083e+00, -6.7754e-01],
                   [ 6.7330e-01, -9.6653e-01, -3.8663e-01],
                   [ 2.3415e-01, -8.9601e-01, -6.8454e-01],
                   [ 1.5111e+00, -1.6660e+00, -2.8023e-01],
                   [ 1.5448e+00, -1.1622e+00, -2.6715e-02],
                   [ 8.3098e-01, -1.3198e+00, -6.4212e-01],
                   [-8.5998e-02,  7.3360e-02,  5.3868e-01]], grad_fn=<AddmmBackward0>)
```

In [180…

```
weights = model.state_dict()
model.load_state_dict(weights)
```

Out[180…

```
<All keys matched successfully>
```

In [181…

```
model = WineNet()
model.forward_softmax(wine_features_tensor)
```

```
Out[181…  tensor([[0.1736, 0.5839, 0.2425],
                 [0.2892, 0.5328, 0.1780],
                 [0.1139, 0.5276, 0.3585],
                 [0.1563, 0.6207, 0.2230],
                 [0.1577, 0.5862, 0.2561],
                 [0.0789, 0.7753, 0.1458],
                 [0.2125, 0.5031, 0.2845],
                 [0.4070, 0.4257, 0.1673],
                 [0.3347, 0.5262, 0.1391],
                 [0.2418, 0.4429, 0.3153],
                 [0.2472, 0.4544, 0.2984],
                 [0.1504, 0.6446, 0.2050],
                 [0.3059, 0.5203, 0.1739],
                 [0.2795, 0.4983, 0.2222],
                 [0.1594, 0.6956, 0.1451],
                 [0.2615, 0.3316, 0.4069],
                 [0.1264, 0.5685, 0.3051],
                 [0.1652, 0.6633, 0.1715],
                 [0.1164, 0.5758, 0.3078],
                 [0.1063, 0.6887, 0.2049],
                 [0.4487, 0.1325, 0.4188],
                 [0.4412, 0.2230, 0.3358],
                 [0.2479, 0.1588, 0.5934],
                 [0.4569, 0.2056, 0.3375],
                 [0.5865, 0.1304, 0.2832],
                 [0.6166, 0.2045, 0.1789],
                 [0.6649, 0.1494, 0.1857],
                 [0.5859, 0.1768, 0.2372],
                 [0.5854, 0.1722, 0.2424],
                 [0.6242, 0.1671, 0.2087],
                 [0.3013, 0.1359, 0.5628],
                 [0.6999, 0.1981, 0.1020],
                 [0.3327, 0.1069, 0.5603],
                 [0.4182, 0.1416, 0.4403],
                 [0.5166, 0.0799, 0.4035],
                 [0.2925, 0.0561, 0.6514],
                 [0.6393, 0.1239, 0.2368],
                 [0.6030, 0.2208, 0.1763],
                 [0.5885, 0.0589, 0.3526],
                 [0.5610, 0.1898, 0.2491],
                 [0.3115, 0.3195, 0.3689],
                 [0.4947, 0.2107, 0.2946],
                 [0.4869, 0.2352, 0.2778],
                 [0.3619, 0.4560, 0.1820],
                 [0.3733, 0.4243, 0.2024],
                 [0.2968, 0.2731, 0.4301],
                 [0.4535, 0.3273, 0.2192],
                 [0.3977, 0.4552, 0.1471],
                 [0.2956, 0.4095, 0.2949],
                 [0.1996, 0.5330, 0.2674],
                 [0.3269, 0.5091, 0.1640],
                 [0.2045, 0.4130, 0.3825],
                 [0.3070, 0.4342, 0.2588],
                 [0.1734, 0.6000, 0.2266],
                 [0.2844, 0.4156, 0.3001],
                 [0.1881, 0.5616, 0.2503],
                 [0.1877, 0.5083, 0.3040],
                 [0.2322, 0.4834, 0.2844],
                 [0.3034, 0.3544, 0.3421],
                 [0.0661, 0.7574, 0.1764],
                 [0.0781, 0.6526, 0.2693],
                 [0.1655, 0.5955, 0.2390],
                 [0.0797, 0.4124, 0.5079],
                 [0.1005, 0.6940, 0.2056],
                 [0.3718, 0.4539, 0.1743],
                 [0.2987, 0.5974, 0.1038],
                 [0.2120, 0.6346, 0.1533],
                 [0.1122, 0.7171, 0.1707],
                 [0.2164, 0.4059, 0.3777],
                 [0.2299, 0.6063, 0.1638],
                 [0.2806, 0.3455, 0.3738],
                 [0.1955, 0.5707, 0.2338],
                 [0.2717, 0.3995, 0.3288],
                 [0.1216, 0.7250, 0.1534],
                 [0.1542, 0.5057, 0.3402],
                 [0.1667, 0.2954, 0.5379],
                 [0.1299, 0.6652, 0.2049],
                 [0.1826, 0.5764, 0.2410],
                 [0.4532, 0.0432, 0.5036],
                 [0.3288, 0.3244, 0.3468],
                 [0.6026, 0.1696, 0.2279],
                 [0.5387, 0.2111, 0.2502],
                 [0.3122, 0.5692, 0.1186],
                 [0.3788, 0.4477, 0.1735],
                 [0.3777, 0.1625, 0.4598],
                 [0.3100, 0.0767, 0.6133],
                 [0.6143, 0.1068, 0.2789],
                 [0.4061, 0.3052, 0.2887],
                 [0.6823, 0.1979, 0.1198],
                 [0.5272, 0.1122, 0.3605],
                 [0.4089, 0.1638, 0.4273],
                 [0.5987, 0.1382, 0.2632],
                 [0.5347, 0.1497, 0.3156],
                 [0.5807, 0.2385, 0.1808],
                 [0.5056, 0.2022, 0.2922],
                 [0.4847, 0.0871, 0.4283],
                 [0.5901, 0.1984, 0.2115],
                 [0.5408, 0.2648, 0.1945],
                 [0.6523, 0.1055, 0.2422],
                 [0.4234, 0.1586, 0.4180],
                 [0.5199, 0.1044, 0.3757],
                 [0.3932, 0.2626, 0.3443],
                 [0.5029, 0.2749, 0.2222],
                 [0.3791, 0.4134, 0.2075],
                 [0.4312, 0.2639, 0.3049],
                 [0.4982, 0.3217, 0.1801],
                 [0.3924, 0.4193, 0.1882],
                 [0.3152, 0.3588, 0.3260],
                 [0.3098, 0.2819, 0.4084],
                 [0.2500, 0.4227, 0.3273],
                 [0.3048, 0.4200, 0.2752],
                 [0.3497, 0.2716, 0.3787],
                 [0.4198, 0.3061, 0.2741],
                 [0.3811, 0.3186, 0.3003],
                 [0.2543, 0.3964, 0.3493],
                 [0.5374, 0.2085, 0.2541],
                 [0.3526, 0.3309, 0.3164],
                 [0.1880, 0.5453, 0.2667],
                 [0.1607, 0.6171, 0.2222],
                 [0.1552, 0.7203, 0.1245],
                 [0.0873, 0.6681, 0.2446],
                 [0.0654, 0.5427, 0.3919],
                 [0.1264, 0.7282, 0.1454],
                 [0.0389, 0.7878, 0.1734],
                 [0.2305, 0.6028, 0.1668],
                 [0.2674, 0.5247, 0.2079],
                 [0.2152, 0.5870, 0.1978],
                 [0.0966, 0.6998, 0.2036],
                 [0.2287, 0.5725, 0.1989],
                 [0.2049, 0.3979, 0.3972],
                 [0.1895, 0.5156, 0.2949],
                 [0.1242, 0.5729, 0.3029],
                 [0.0764, 0.6532, 0.2704],
                 [0.1773, 0.5900, 0.2327],
                 [0.1529, 0.5590, 0.2881],
                 [0.1565, 0.6494, 0.1942],
                 [0.3381, 0.1613, 0.5005],
                 [0.3951, 0.2651, 0.3398],
                 [0.3743, 0.2664, 0.3593],
                 [0.5078, 0.0828, 0.4093],
                 [0.2513, 0.6346, 0.1140],
                 [0.5584, 0.2376, 0.2040],
                 [0.5249, 0.0917, 0.3835],
                 [0.3957, 0.2804, 0.3239],
                 [0.2663, 0.2820, 0.4517],
                 [0.5307, 0.1723, 0.2970],
                 [0.6051, 0.1441, 0.2508],
                 [0.6716, 0.1337, 0.1947],
                 [0.4617, 0.1212, 0.4171],
                 [0.2983, 0.2688, 0.4329],
                 [0.4979, 0.1573, 0.3448],
                 [0.4288, 0.2398, 0.3314],
                 [0.5306, 0.2211, 0.2484],
                 [0.5535, 0.1841, 0.2624],
                 [0.2492, 0.6636, 0.0872],
                 [0.4975, 0.3399, 0.1627],
                 [0.5215, 0.1959, 0.2826],
                 [0.5261, 0.1926, 0.2812],
                 [0.4326, 0.3114, 0.2560],
                 [0.3799, 0.1242, 0.4959],
                 [0.3031, 0.1951, 0.5017],
                 [0.3786, 0.3186, 0.3027],
                 [0.3546, 0.3378, 0.3076],
                 [0.3161, 0.4276, 0.2562],
                 [0.1775, 0.6919, 0.1306],
                 [0.2545, 0.5523, 0.1932],
                 [0.1659, 0.2570, 0.5771],
                 [0.3224, 0.1763, 0.5012],
                 [0.1871, 0.3231, 0.4898],
                 [0.1072, 0.5952, 0.2976],
                 [0.1981, 0.4349, 0.3671],
                 [0.2074, 0.5388, 0.2538],
                 [0.2169, 0.6150, 0.1681],
                 [0.2039, 0.7079, 0.0881],
                 [0.1734, 0.4506, 0.3760],
                 [0.2194, 0.3658, 0.4148],
                 [0.1565, 0.6306, 0.2128],
                 [0.5269, 0.1713, 0.3018]], grad_fn=<SoftmaxBackward0>)
```

The SoftMax function just converts the values into probabilities. So, we are given three probabilities in the output tensor, with each probability corresponding to a certain category (in this case, the probability that this data point belongs in which cultivar, 1 2 or 3)

3.b

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

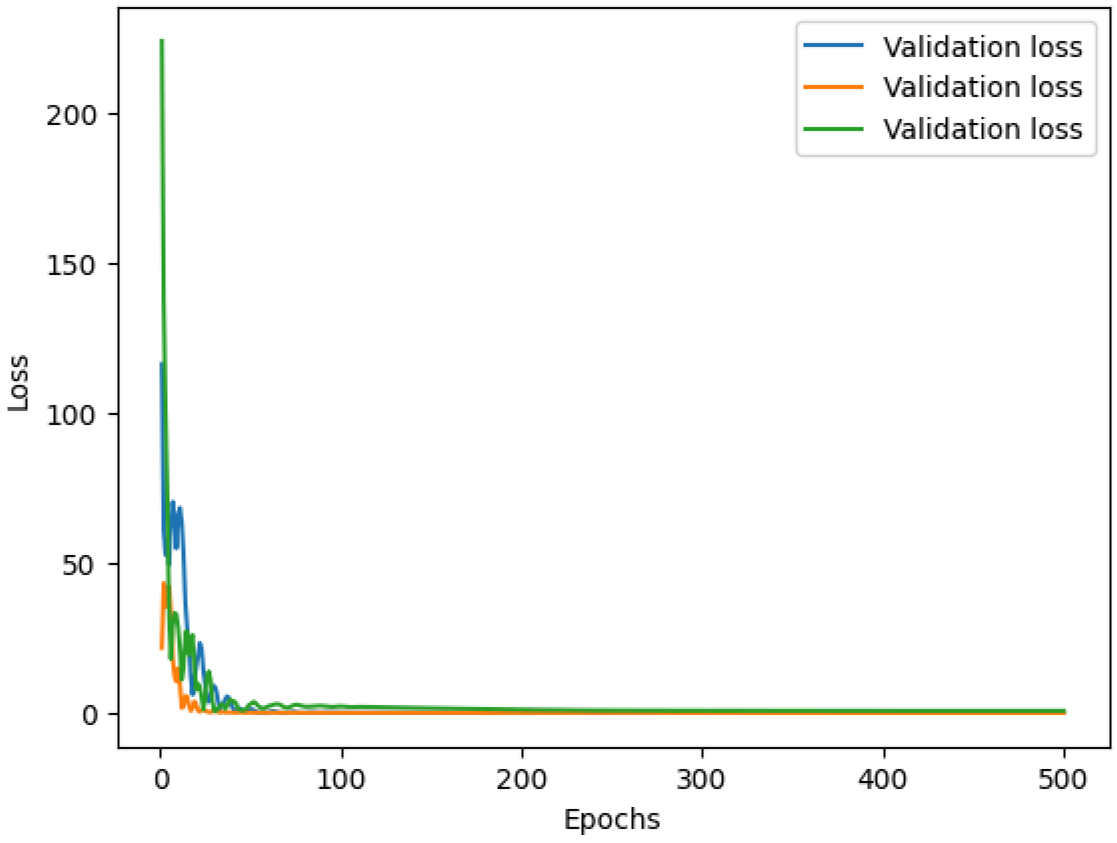
In [215…

```python
import torch
import matplotlib.pyplot as plt
from torch.optim import Adam
from torch import nn
from sklearn.model_selection import train_test_split

# you can use this framework to do training and validation
def train_and_val(model,train_X,train_y,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 #
    optimizer = Adam(model.parameters(), lr=5E-2)
    loss_func = nn.CrossEntropyLoss()
    train_X = torch.tensor(train_X, dtype=torch.float)
    # train_y = train_y - 1
    train_y = torch.tensor(train_y, dtype=torch.long)

    ### Split training examples further into training and validation ###
    X_train, X_val, y_train, y_val = train_test_split(train_X, train_y, test

    val_array=[]
    lowest_val_loss = np.inf
    model_param = model.state_dict()

    for i in range(epochs):
        ### Compute the loss and do backpropagation ###
        # display(X_train.dtype)
        y_pred = model(X_train)
        loss = loss_func(y_pred, y_train-1)

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

        ### compute validation loss and keep track of the lowest val loss ##
        with torch.no_grad():
            y_hat = model(X_val)
            # print(y_hat)
            # print(y_val)
            val_loss = loss_func(model(X_val), y_val-1)
            val_array.append(val_loss)

            if val_loss < lowest_val_loss:
                lowest_val_loss = val_loss
                model_param = model.state_dict()

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

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

In [219…

```
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=3)
    train_acc_all = []
    test_acc_all = []
    for train_index, test_index in kf.split(X):
        # X_train, X_test = torch.from_numpy(X[train_index]), torch.from_num
        # y_train, y_test = torch.from_numpy(y[train_index]), torch.from_num
        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 = WineNet()
        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, torch.from_numpy(X_train).f
        train_acc_all.append(train_acc)
        test_acc = calculate_accuracy_nn(model, torch.from_numpy(X_test).flo
        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(f"Training accuracy mean: {np.mean(train_acc_all)}, std: {np.std(t
    print(f"Testing  accuracy mean: {np.mean(test_acc_all)}, std: {np.std(te

KFoldNN(3, wine_features_np, wine_rankings_np)
# KFoldNN(3, wine_features_tensor, wine_rankings_np)
```

```
Number of epochs with lowest validation: 69
Train accuracy: 0.9830508
Test accuracy: 0.95
Number of epochs with lowest validation: 500
Train accuracy: 0.99159664
Test accuracy: 0.9830508
Number of epochs with lowest validation: 30
Train accuracy: 0.96638656
Test accuracy: 0.9322034
Final results:
Training accuracy mean: 0.9803447127342224, std: 0.010468349792063236
Testing  accuracy mean: 0.9550848007202148, std: 0.02106744423508644
```


Validation loss
Validation loss
Validation loss
Loss
Epochs
0
50
100
150
200
0
100
200
300
400
500

```
In [ ]:
```