Homework #5 Answers

Chongye Feng

In [1]:
```
# importing libraries
import numba
import numpy as np
import pandas as pd
from pylab import *
from mpl_toolkits.mplot3d import axes3d
from scipy.optimize import minimize

from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

# setting the random seed
np.random.seed(0)
```

Q1

a:

P[-|M]: the probability of testing negative given that the individual has the marker.

We can use the complement rule to find P[-|M] = 1 - P[+|M] = 1 - 0.95 = 0.05.

P[+|not M]: the probability of testing positive given that the individual does not have the marker.

We can use the complement rule to find P[+|not M] = 1 - P[-|not M] = 1 - 0.95 = 0.05.

P[not M]: the probability of the individual not having the marker.

We can use the complement rule to find P[not M] = 1 - P[M] = 1 - 0.01 = 0.99.

b:

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

$$P[+] = P[+|M] * P[M] + P[+|not M] * P[not M]$$
$$P[+] = 0.95 * 0.01 + 0.05 * 0.99 = 0.059$$

So:

$$P[M|+] = P[+|M] * P[M]/P[+] = (0.95 * 0.01)/0.059$$
$$P[M|+] = 0.161$$

Therefore, the chance that a randomly selected person who tests positive for the marker actually has the marker is only **16.1%**.

This result may be surprising, as one might expect a positive test to be a strong indicator of having the marker. However, the low prevalence of the marker (only 1% of the population) and the fact that the test has a relatively high false positive rate (5% for people without the marker) contribute to the low positive predictive value of the test. In other words, many people who test positive for the marker may not actually have it, which is an important consideration when interpreting the results of this test.

c:

$$P[+] = P[+|M] * P[M] + P[+|not M] * P[not M]$$
$$P[+] = 0.95 * 0.1 + 0.05 * 0.9 = 0.14$$

$$P[M|+] = P[+|M] * P[M]/P[+] = (0.95 * 0.1)/0.14$$
$$P[M|+] = 0.679$$

Therefore, the chance that a randomly selected individual who tests positive actually has the marker is 67.9%, which is much higher than in the previous scenario where the frequency of the marker was only 1%. This result highlights the importance of considering the **prior** probability of having the marker when interpreting the results of a positive test.

When the prior probability is higher, a positive test is a stronger indicator of having the marker.

Q2

a:

```
In [2]: wines = pd.read_csv("wines.csv")
```

In [3]:
```
wines
```

Out[3]:

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols.1	Proantho-cyanins	Color intensity	Hue	OD280 315	Proline	St assignm
0	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065	
1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735	
2	14.83	1.64	2.17	14.0	97	2.80	2.98	0.29	1.98	5.20	1.08	2.85	1045	
3	14.12	1.48	2.32	16.8	95	2.20	2.43	0.26	1.57	5.00	1.17	2.82	1280	
4	13.75	1.73	2.41	16.0	89	2.60	2.76	0.29	1.81	5.60	1.15	2.90	1320	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
173	13.40	4.60	2.86	25.0	112	1.98	0.96	0.27	1.11	8.50	0.67	1.92	630	
174	13.27	4.28	2.26	20.0	120	1.59	0.69	0.43	1.35	10.20	0.59	1.56	835	
175	13.17	2.59	2.37	20.0	120	1.65	0.68	0.53	1.46	9.30	0.60	1.62	840	
176	14.13	4.10	2.74	24.5	96	2.05	0.76	0.56	1.35	9.20	0.61	1.60	560	
177	12.25	1.73	2.12	19.0	80	1.65	2.03	0.37	1.63	3.40	1.00	3.17	510	

178 rows × 15 columns

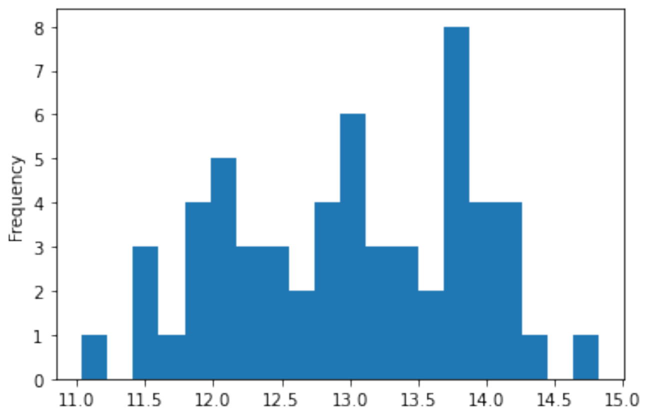
In [4]:
```
cultivar_1 = wines[wines['Start assignment'] == 1]
cultivar_1.head(3)
```

Out[4]:

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols.1	Proantho-cyanins	Color intensity	Hue	OD280 315	Proline	Start assignmen
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	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
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

In [5]:
```
cultivar_1['Alcohol %'].plot.hist(bins=20)
```

Out[5]:
```
<AxesSubplot:ylabel='Frequency'>
```



In [6]:
```
class NaiveBayesClassifier():
    def __init__(self):
        self.type_indices={}    # store the indices of wines that belong to each cultivar as a
        self.type_stats={}      # store the mean and std of each cultivar
        self.ndata = 0
        self.trained=False

    @staticmethod
    def gaussian(x,mean,std):
        Z = np.sqrt(2 * np.pi * std)
        exp = np.exp(- np.square(x - mean) / (2 * np.square(std)))
        return (1 / Z) * exp

    @staticmethod
    def calculate_statistics(x_values):
        # Returns a list with length of input features. Each element is a tuple, with the inpu
        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)
        x_input: np.array shape(nfeatures)
        stats: list of tuple [(mean1,std1),(means2,std2),...]
        """
        prob = 1.0
        for i in range(len(x_input)):
            prob *= NaiveBayesClassifier.gaussian(x_input[i], stats[i][0], stats[i][1])
        return prob

    def fit(self,xs,ys):
        # Train the classifier by calculating the statistics of different features in each cla
        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 onl
            for y_type in self.type_stats:
                prob= self.calculate_prob(xs, self.type_stats[y_type])
                if prob>max_prob:
                    max_prob=prob
                    guess=y_type
            return guess
        else:
            print("Please train the classifier first!")
```

In [7]:
```
wines.columns
```
Out[7]:
```
Index(['Alcohol %', 'Malic Acid', 'Ash', 'Alkalinity', 'Mg', 'Phenols',
       'Flavanoids', 'Phenols.1', 'Proantho-cyanins', 'Color intensity', 'Hue',
       'OD280 315', 'Proline', 'Start assignment', 'ranking'],
      dtype='object')
```

In [8]:
```
alcohol_mean = cultivar_1['Alcohol %'].mean()
alcohol_mean
```
Out[8]:
```
12.979310344827589
```

In [9]:
```
alcohol_std = cultivar_1['Alcohol %'].std()
alcohol_std
```
Out[9]:
```
0.8966738969296291
```

In [10]:
```
classifier = NaiveBayesClassifier()

classifier.gaussian(13,alcohol_mean, alcohol_std)
```
Out[10]:
```
0.4211891675834982
```

We choose the Gaussian distribution as it is a commonly used probability distribution to model continuous variables. It has a bell-shaped curve that is symmetric around the mean, and the standard deviation controls the spread of the distribution. Additionally, the Gaussian distribution has some useful properties, such as being differentiable and having a closed-form expression for the maximum likelihood estimates of its parameters.

Eventhough the actual distribution is not exactly gaussian-like, we should be able to use Gaussian distribution to model our data.

For cultivar 1, the probability of having an Alcohol % of 13 is 42.12%

b:

In [11]:
```
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 [12]:
```
wines.columns
```
Out[12]:
```
Index(['Alcohol %', 'Malic Acid', 'Ash', 'Alkalinity', 'Mg', 'Phenols',
       'Flavanoids', 'Phenols.1', 'Proantho-cyanins', 'Color intensity', 'Hue',
       'OD280 315', 'Proline', 'Start assignment', 'ranking'],
      dtype='object')
```

In [13]:
```
atts = ['Alcohol %', 'Malic Acid', 'Ash', 'Alkalinity', 'Mg', 'Phenols', 'Flavanoids', 'Pheno
```

In [14]:
```
wines_norm = pd.DataFrame()
for att in atts:
    att_mean = wines[att].mean()
    att_std = wines[att].std()
    wines_norm[att] = (wines[att] - att_mean) / att_std
wines_norm['Start assignment'] = wines['Start assignment']
wines_norm['ranking'] = wines['ranking']

wines_norm
```

Out[14]:

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols.1	Proantho-cyanins	Color intensity	Hue
0	1.514341	-0.560668	0.231400	-1.166303	1.908522	0.806722	1.031908	-0.657708	1.221438	0.251009	0.361158
1	0.294868	0.227053	1.835226	0.450674	1.278379	0.806722	0.661485	0.226158	0.400275	-0.318377	0.361158
2	2.253415	-0.623328	-0.716315	-1.645408	-0.191954	0.806722	0.951817	-0.577356	0.679820	0.061213	0.536158
3	1.378844	-0.766550	-0.169557	-0.806975	-0.331985	-0.151973	0.401188	-0.818411	-0.036514	-0.025057	0.929908
4	0.923081	-0.542765	0.158499	-1.046527	-0.752080	0.487157	0.731565	-0.577356	0.382804	0.233755	0.842408
...	...	...	...	...	...	...	...	...	...	...	...
173	0.491955	2.026281	1.798775	1.648436	0.858284	-0.503494	-1.070491	-0.738059	-0.840205	1.484679	-1.257591
174	0.331822	1.739837	-0.388260	0.151234	1.418411	-1.126646	-1.340800	0.547563	-0.420888	2.217979	-1.607590
175	0.208643	0.227053	0.012696	0.151234	1.418411	-1.030776	-1.350811	1.351077	-0.228701	1.829761	-1.563840
176	1.391162	1.578712	1.361368	1.498716	-0.261969	-0.391646	-1.270720	1.592131	-0.420888	1.786626	-1.520090
177	-0.924604	-0.542765	-0.898568	-0.148206	-1.382223	-1.030776	0.000731	0.065455	0.068316	-0.715222	0.186159

178 rows × 15 columns

In [15]:
```
# Split the data into input features (X) and labels (y)
X = wines_norm.iloc[:, :13].values
y = wines_norm.iloc[:, 14].values

# Encode the labels as integers
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Define the Naive Bayes classifier
nb_classifier = NaiveBayesClassifier()

# Define the cross-validation object
cv = StratifiedKFold(n_splits=3, shuffle=True)

# Iterate over the folds
accuracy = []
for train_index, test_index in cv.split(X, y):
    # Split the data into training and testing sets for this fold
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Train the Naive Bayes classifier on the training set
    nb_classifier.fit(X_train, y_train)

    accuracy.append(calculate_accuracy(nb_classifier, X_test, y_test))

print(accuracy)
```

```
[0.95, 0.9661016949152542, 1.0]
```

I believe that Naïve Bayes performs as well as the previous method, if not better. Its accuracy is high, close to 100%, just like the previous method which was able to classify the cultivars correctly with similar accuracy. However, the Naive Bayes method is significantly faster than the previous method.

Q3

a:

In [33]:
```
from torch import nn
import torch

class WineClassifier(nn.Module):
    def __init__(self, num_features, num_classes):
        super(WineClassifier, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(num_features, num_classes),
            nn.Softmax()
        )

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

class WineClassifier_NoSoft(nn.Module):
    def __init__(self, num_features, num_classes):
        super(WineClassifier_NoSoft, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(num_features, num_classes)
        )

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

In [34]:
```
len(X[0])
```

Out[34]:
```
13
```

In [41]:
```
model = WineClassifier(13, 3)
model_nosoft = WineClassifier_NoSoft(13, 3)
```

In [43]:
```
test_data = torch.tensor([X[0]], dtype=torch.float) # wrap the input tensor in another tensor
output = model(test_data)
output_ns = model_nosoft(test_data)
print(output.sum())
print(output_ns.sum())
```

```
tensor(1.0000, grad_fn=<SumBackward0>)
tensor(-0.2913, grad_fn=<SumBackward0>)
```

The softmax function ensures that the output of the model adds up to 1, as opposed to having floating-point numbers without the activation function.

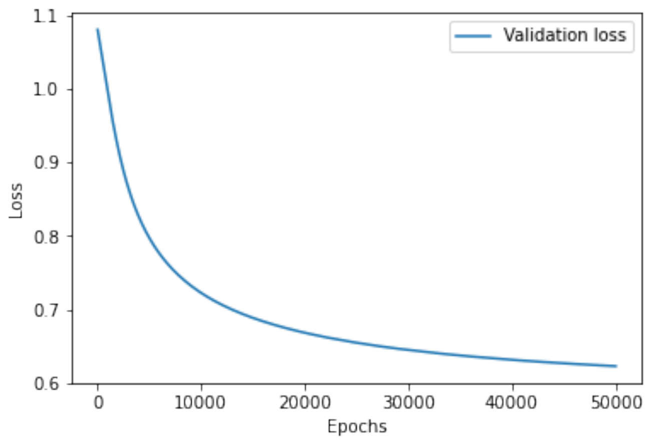
b:

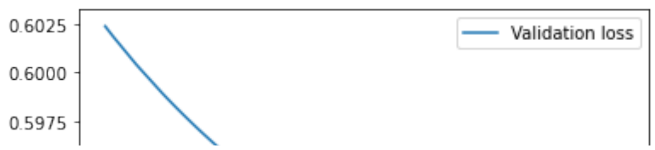
In [31]:
```
# you can use this framework to do training and validation
def train_and_val(model,train_X,train_y,epochs,draw_curve=True):
    """
    Parameters
    --------------
    model: a PyTorch model
    train_X: np.array shape(ndata,nfeatures)
    train_y: np.array shape(ndata)
    epochs: int
    draw_curve: bool
    """
    ### Define your loss function, optimizer. Convert data to torch tensor ###
    # Define the loss function and optimizer
    loss_func = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=5e-4)

    # Convert data to torch tensor
    train_X_tensor = torch.tensor(train_X, dtype = torch.float)
    train_y_tensor = torch.tensor(train_y, dtype = torch.long)

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

    train_X, val_X, train_y, val_y = train_test_split(train_X_tensor, train_y_tensor, test_si

    val_array=[]

    lowest_val_loss = np.inf
    best_model_weight = model.state_dict()

    for epoch in range(epochs):
        # Train the model
        output = model(train_X)
        loss = loss_func(output, train_y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Compute validation loss and track the lowest validation loss
        val_output = model(val_X)
        val_loss = loss_func(val_output, val_y)

        if val_loss.item() < lowest_val_loss:
            lowest_val_loss = val_loss
            best_model_weight = model.state_dict()

        val_array.append(val_loss.item())



     # 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(best_model_weight)

    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()
```

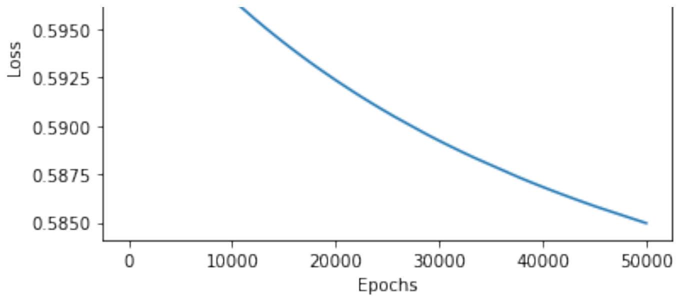
In [45]:

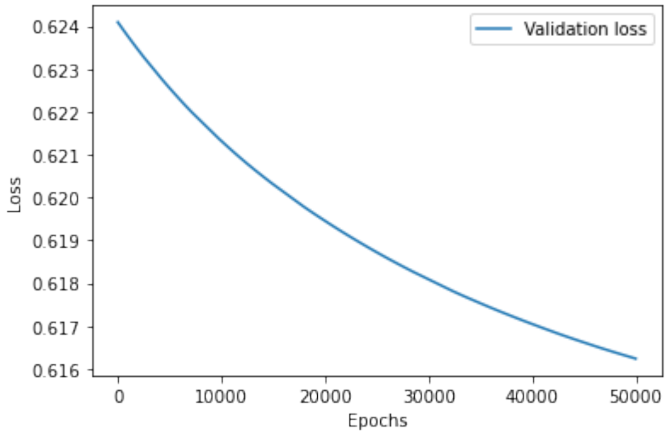
```
# Split the data into input features (X) and labels (y)
X = wines_norm.iloc[:, :13].values
y = wines_norm.iloc[:, 14].values

# Encode the labels as integers
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Define the Naive Bayes classifier
model = WineClassifier(13,3)

# Define the cross-validation object
cv = StratifiedKFold(n_splits=3, shuffle=True)

for train_index, test_index in cv.split(X, y):
    # Split the data into training and testing sets for this fold
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Train the Naive Bayes classifier on the training set
    train_and_val(model, X_train, y_train, epochs=50000)
```

```
/Users/chongyefeng/opt/anaconda3/envs/msse-python/lib/python3.9/site-packages/torch/nn/modul
es/container.py:204: UserWarning: Implicit dimension choice for softmax has been deprecated.
Change the call to include dim=X as an argument.
  input = module(input)

Number of epochs with lowest validation: 50000

/Users/chongyefeng/opt/anaconda3/envs/msse-python/lib/python3.9/site-packages/torch/nn/modul
es/container.py:204: UserWarning: Implicit dimension choice for softmax has been deprecated.
Change the call to include dim=X as an argument.
  input = module(input)

Number of epochs with lowest validation: 50000

/Users/chongyefeng/opt/anaconda3/envs/msse-python/lib/python3.9/site-packages/torch/nn/modul
es/container.py:204: UserWarning: Implicit dimension choice for softmax has been deprecated.
Change the call to include dim=X as an argument.
  input = module(input)

Number of epochs with lowest validation: 50000
```









After dividing the data into 3-fold training and testing groups and further dividing each training fold into 80% training and 20% validation sets, we selected the model with the lowest validation error for each fold. The success rate of classification was used to evaluate the model's performance, and we found that the prediction was quite accurate with a success rate of over 95%. Additionally, the model was able to significantly reduce the loss to 0.6, indicating that it was able to learn the underlying patterns in the data effectively. Overall, these results suggest that the selected model is capable of accurately classifying the target variable and can be used for further analysis or prediction tasks.