# Homework 7 Written Part

**Q1: Feed Forward and Backpropagation**

**1a)** In the neural network structure, $f_1(\cdot)$ is a sigmoid function and $f_2(\cdot)$ is a softmax function. The sigmoid function is defined as below:

$$f_1(z) = \frac{1}{1+e^{-z}}$$

The sigmoid function's derivative can be defined as below:

Through power rule and chain rule:

$$f_1 = (1+e^{-z})^{-1} \;\rightarrow\; f_1' = -(1+e^{-z})^{-2} \cdot -e^{-z} = \frac{e^{-z}}{(1+e^{-z})^2}$$

We can split the denominator and manipulate the numerator as below:

$$f_1' = \frac{1}{(1+e^{-z})} \cdot \frac{e^{-z}+1-1}{(1+e^{-z})} = \frac{1}{(1+e^{-z})}\left(1 - \frac{1}{(1+e^{-z})}\right)$$

Replacing $\frac{1}{1+e^{-z}}$ with $f_1$, which is the sigmoid function definition:

$$\boxed{f_1'(z) = f(z)(1-f(z))}$$

Next, the softmax function is defined as below:

$$f_2(z_i) = \frac{e^{z_i}}{\sum_j^K e^{z_j}}$$

In the above function, the $z_i$ can be broken down into $K$ sub-components ($z_j$). We want to differentiate each and every one of these sub-components because the total derivative make up the entire derivative in respect to $z_i$, given us the expression below:

Through quotient rule:

$$\frac{\partial f_2}{\partial z_j} = \frac{\sum_j^K e^{z_j} \frac{\partial}{\partial z_j} e^{z_i} - e^{z_i} \frac{\partial}{\partial z_j} \sum_j^K e^{z_j}}{\left(\sum_j^K e^{z_j}\right)^2}$$

There are two underlying scenarios we want to observe: when $i = j$, and when $i \neq j$, the derivatives can be defined depending on the scenario:

$$\frac{\partial}{\partial z_j} e^{z_i} = \begin{cases} e^{z_j} & i = j \\ 0 & i \neq j \end{cases}$$

$$\frac{\partial}{\partial z_j} \sum_j^K e^{z_j} = \frac{\partial}{\partial z_j}(e^{z_j} + \cdots + e^{z_K}) = e^{z_j}$$

Substituting these definition into $\frac{\partial f_2}{\partial z_j}$ starting with scenario where $i = j$:

$$\frac{\partial f_2}{\partial z_j} = \frac{e^{z_j}\sum_j^K e^{z_j} - e^{z_j}e^{z_i}}{\left(\sum_j^K e^{z_j}\right)^2} = \frac{e^{z_j}\left(\sum_j^K e^{z_j} - e^{z_i}\right)}{\left(\sum_j^K e^{z_j}\right)^2}$$

Similarly to the sigmoid function, we want to split the denominator and manipulate the numerator a little bit:

$$= \frac{e^{z_j}}{\sum_j^K e^{z_j}}\left(\frac{\sum_j^K e^{z_j}}{\sum_j^K e^{z_j}} - \frac{e^{z_i}}{\sum_j^K e^{z_j}}\right) = \frac{e^{z_j}}{\sum_j^K e^{z_j}}\left(1 - \frac{e^{z_i}}{\sum_j^K e^{z_j}}\right)$$

Replace with softmax function definition:

$$\frac{\partial f_2}{\partial z_j} = \boxed{f_2(z_j)(1-f_2(z_i)),\ i = j}$$

When $i \neq j$:

$$\frac{\partial f_2}{\partial z_j} = \frac{e^{z_j} \cdot 0 - e^{z_j} e^{z_i}}{\left(\sum_j^K e^{z_j}\right)^2} = -\left(\frac{e^{z_j}}{\sum_j^K e^{z_j}}\right)\left(\frac{e^{z_i}}{\sum_j^K e^{z_j}}\right) = \boxed{-f_2(z_j) f_2(z_i),\ i \neq j}$$

**1b)** The error cost for output is defined as the cross-entropy function: $L = -t^T \ln(x_2)$. To define the partial derivative of $L$ w.r.t. $x_2$:

$$\frac{\partial L}{\partial x_2} = -t^T \frac{\partial}{\partial x_2} \ln(x_2) = \boxed{-t^T \frac{1}{x_2}}$$

**1c)** Because $a_1, x_1$ and $\delta_1$ are not at the output layer, they have the following definition:

$$a_i = W_i \cdot x_{i-1} + b_i \quad x_i = f_i(a_i) \quad \delta_i = \delta_{i+1} W_{i+1} f_i'(a_i)$$

Using the provided information such as:

$$W_1 = \begin{bmatrix} 1 & 2 & -3 & 0 & 1 & -3 \\ 3 & 1 & 2 & 1 & 0 & 2 \\ 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 0 & 2 & 1 & -2 & 2 \end{bmatrix} \quad W_2 = \begin{bmatrix} 1 & 2 & -2 & 1 \\ 1 & -1 & 1 & 2 \\ 3 & 1 & -1 & 1 \end{bmatrix} \quad x_0 = [1,1,0,0,1,1]^T \quad t = [0,1,0]^T \quad b_i = 1, \forall i$$

$$a_1 = W_1 x_0 + b_1 = \begin{bmatrix} 1 & 2 & -3 & 0 & 1 & -3 \\ 3 & 1 & 2 & 1 & 0 & 2 \\ 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 0 & 2 & 1 & -2 & 2 \end{bmatrix}_{4x6} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}_{6x1} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}_{4x1} = \boxed{\begin{bmatrix} 2 \\ 7 \\ 8 \\ 2 \end{bmatrix}_{4x1}}$$

$$x_1 = f_1(a_1) = \begin{bmatrix} \frac{1}{1+e^{-2}} \\ \frac{1}{1+e^{-7}} \\ \frac{1}{1+e^{-8}} \\ \frac{1}{1+e^{-2}} \end{bmatrix} = \boxed{\begin{bmatrix} 0.8808 \\ 0.9991 \\ 0.9997 \\ 0.8808 \end{bmatrix}}$$

Since $\delta_1 = \delta_2 W_2 f_1'(a_1)$ and it contains components ($\delta_2$) at the output layer, we need a different set of definition to describe those components. Note that variables need to cross over to Exercise 1d in order to solve for $\delta_1$ in Exercise 1c, Exercise 1d starts here to prevent repeated work and I will put answer for $\delta_1$ at the end of Exercise 1d.

**1d)**

$$\delta_2 = \frac{\partial L}{\partial x_2} \frac{\partial x_2}{\partial a_2} = \frac{\partial L}{\partial x_2} \circ f_2'(a_2)$$

$$a_2 = W_2 \cdot x_1 + b_2 = \begin{bmatrix} 1 & 2 & -2 & 1 \\ 1 & -1 & 1 & 2 \\ 3 & 1 & -1 & 1 \end{bmatrix}_{3x4} \begin{bmatrix} 0.8808 \\ 0.9991 \\ 0.9997 \\ 0.8808 \end{bmatrix}_{4x1} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}_{3x1} = \boxed{\begin{bmatrix} 2.7604 \\ 3.6430 \\ 4.5226 \end{bmatrix}_{3x1}}$$

$$x_2 = f_2(a_2) = f_2(W_2 x_1) = \frac{e^{W_2 x_1}}{\sum_{j=1}^{K=3} e^{W_{2,j} x_1}} = \frac{e^{W_2 x_1}}{e^{W_{2,1} x_1} + e^{W_{2,2} x_1} + e^{W_{2,3} x_1}}$$

where $W_{2,j}$ refers to the $j$th row in $W_2$. Continuing to evaluate:

$$x_2 = \frac{e^{a_2}}{e^{2.8763} + e^{3.6430} + e^{4.5226}} = \boxed{\begin{bmatrix} 0.1082 \\ 0.2615 \\ 0.6303 \end{bmatrix}}$$

To match the matrix multiplication, the order of multiplication is switched, such that:

$$\delta_2 = \frac{\partial x_2}{\partial a_2}\frac{\partial L}{\partial x_2} = -f_2'(a_2) \circ t^T \frac{1}{x_2} = -f_2' \left( \begin{bmatrix} 2.8763 \\ 3.6430 \\ 4.5226 \end{bmatrix} \right)_{3x3} \left( \begin{bmatrix} 0/0.1082 \\ 1/0.2615 \\ 0/0.6303 \end{bmatrix} \right)_{3x1}$$

$f_2'$ is a 3x3 matrix from due to the nature of the derivative of the softmax function, the 3x3 matrix can be represented as below, where $a_{2,j}$ represents to the $j$th row in $a_2$:

$$f_2'(a_2) = \begin{bmatrix} f_2(a_{2,1})(1 - f_2(a_{2,1})) & -f_2(a_{2,1})f_2(a_{2,2}) & -f_2(a_{2,1})f_2(a_{2,3}) \\ -f_2(a_{2,2})f_2(a_{2,1}) & f_2(a_{2,2})(1 - f_2(a_{2,2})) & -f_2(a_{2,2})f_2(a_{2,3}) \\ -f_2(a_{2,3})f_2(a_{2,1}) & -f_2(a_{2,3})f_2(a_{2,2}) & f_2(a_{2,3})(1 - f_2(a_{2,3})) \end{bmatrix}$$

$$f_2'(a_2) = \begin{bmatrix} x_{2,1}(1 - x_{2,1}) & -x_{2,1}x_{2,2} & -x_{2,1}x_{2,3} \\ -x_{2,2}x_{2,1} & x_{2,2}(1 - x_{2,2}) & -x_{2,2}x_{2,3} \\ -x_{2,3}x_{2,1} & -x_{2,3}x_{2,2} & x_{2,3}(1 - x_{2,3}) \end{bmatrix}$$

$$= \begin{bmatrix} 0.1082(1 - 0.1082) & -(0.1082)(0.2615) & -(0.1082)(0.6303) \\ -(0.2615)(0.1082) & 0.2615(1 - 0.2615) & -(0.2615)(0.6303) \\ -(0.6303)(0.1082) & -(0.6303)(0.2615) & 0.6303(1 - 0.6303) \end{bmatrix} = \boxed{\begin{bmatrix} 0.0965 & -0.0283 & -0.0682 \\ -0.0283 & 0.1931 & -0.1648 \\ -0.0682 & -0.1648 & 0.2330 \end{bmatrix}}$$

Tying the terms altogether:

$$\delta_2 = -\begin{bmatrix} 0.0965 & -0.0283 & -0.0682 \\ -0.0283 & 0.1931 & -0.1648 \\ -0.0682 & -0.1648 & 0.2330 \end{bmatrix} \begin{bmatrix} 0 \\ 3.8236 \\ 0 \end{bmatrix} = \boxed{\begin{bmatrix} 0.1082 \\ -0.7385 \\ 0.6303 \end{bmatrix}}$$

Going back to part 1c to $\delta_1$, where $\delta_1 = \delta_2 W_2 f_1'(a_1)$ We are still missing the value $f_1'(a_1)$, therefore solving as follows:

$$f_1'(a_1) = f_1(a_1)(1 - f_1(a_1)) = [x_{1,1}(1 - x_{1,1})\ x_{1,2}(1 - x_{1,2})\ x_{1,3}(1 - x_{1,3})\ x_{1,4}(1 - x_{1,4})]^T$$

$$= \begin{bmatrix} 0.1050 \\ 0.0009 \\ 0.0003 \\ 0.1050 \end{bmatrix}$$

We need to modify the multiplication sequence a little bit in order to utilize the Hadamard Product, $\delta_1$ becomes $W^T\delta_2 \circ f_1'(a_1)$:

$$\delta_1 = \begin{bmatrix} 1 & 1 & 3 \\ 2 & -1 & 1 \\ -2 & 1 & -1 \\ 1 & 2 & 1 \end{bmatrix}_{4x3} \begin{bmatrix} 0.1082 \\ -0.7385 \\ 0.6303 \end{bmatrix}_{3x1} \circ \begin{bmatrix} 0.1050 \\ 0.0009 \\ 0.0003 \\ 0.1050 \end{bmatrix}_{4x1} = \boxed{\begin{bmatrix} 0.1323 \\ 0.0014 \\ -0.0005 \\ -0.0775 \end{bmatrix}}$$

**1e)** Based on the result from 1d, the output of $x_2$ from the input $x_0$ equals to $[0.1082,\ 0.2615,\ 0.6303]^T$, therefore, the index with maximum probability will be chosen: $\boxed{[0,\ 0,\ 1]^T}$, which corresponds to class 3.

**1f)** The loss function is defined by the cross entropy function, in which the loss can be found as below:

$$L = -t^T \ln(x_2) = -[0\ 1\ 0] \begin{bmatrix} \ln(0.1082) \\ \ln(0.2615) \\ \ln(0.6303) \end{bmatrix} = \boxed{1.3413}$$

**2a)** The gradient descent of $W_2$ is defined by:

$$W_{2,new} \leftarrow W_{2,old} - \alpha \frac{\partial L}{\partial w_2}$$

This can be further defined as:

$$W_{2,new} \leftarrow W_{2,old} - \alpha \delta_2 x_1$$

For a learning rate of 0.5, the updated $W_2$ is calculated as below, with $x_1$ transposed to match the matrix dimension for subtraction:

$$W_{2,new} = \begin{bmatrix} 1 & 2 & -2 & 1 \\ 1 & -1 & 1 & 2 \\ 3 & 1 & -1 & 1 \end{bmatrix} - 0.5 \begin{bmatrix} 0.1082 \\ -0.7385 \\ 0.6303 \end{bmatrix} \begin{bmatrix} 0.8808 & 0.9991 & 0.9997 & 0.8808 \end{bmatrix} =$$

$$\boxed{\begin{bmatrix} 0.9524 & 1.9460 & -2.0541 & 0.9524 \\ 1.3252 & -0.6311 & 1.3691 & 2.3252 \\ 2.7224 & 0.6852 & -1.3150 & 0.7224 \end{bmatrix}}$$

**2b)** Define gradient descent update for $b_2$ as:

$$b_{2,new} \leftarrow b_{2,old} - \alpha \delta_2$$

$$b_{2,new} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - 0.5 \begin{bmatrix} 0.1082 \\ -0.7385 \\ 0.6303 \end{bmatrix} = \boxed{\begin{bmatrix} 0.9459 \\ 1.3692 \\ 0.6849 \end{bmatrix}}$$

**2c)**

$$W_{1,new} \leftarrow W_{1,old} - \alpha \delta_1 x_0$$

$$W_{1,new} = \begin{bmatrix} 1 & 2 & -3 & 0 & 1 & -3 \\ 3 & 1 & 2 & 1 & 0 & 2 \\ 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 0 & 2 & 1 & -2 & 2 \end{bmatrix} - 0.5 \begin{bmatrix} 0.1323 \\ 0.0014 \\ -0.0005 \\ -0.0775 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} =$$

$$\boxed{\begin{bmatrix} 0.9338 & 1.9338 & -3.0000 & 0 & 0.9338 & -3.0662 \\ 2.9993 & 0.9993 & 2.0000 & 1.0000 & -0.0007 & 1.9993 \\ 2.0003 & 2.0003 & 2.0000 & 2.0000 & 2.0003 & 1.0003 \\ 1.0388 & 0.0388 & 2.0000 & 1.0000 & -1.9612 & 2.0388 \end{bmatrix}}$$

**2d)**

$$b_{1,new} \leftarrow b_{1,old} - \alpha \delta_1$$

$$b_{1,new} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} - 0.5 \begin{bmatrix} 0.1323 \\ 0.0014 \\ -0.0005 \\ -0.0775 \end{bmatrix} = \boxed{\begin{bmatrix} 0.9338 \\ 0.9993 \\ 1.0003 \\ 1.0388 \end{bmatrix}}$$

**2e)** The updated $x_2$ using forward propagation of updated components is equal to $\boxed{[0.0517,\ 0.8566,\ 0.0918]}$, which corresponds to a classification of $[0,\ 1,\ 0]$, same as the ground truth.

# Q2



In [11]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import tree, neighbors, linear_model, svm, neural_network, metrics
```

# Problem 2a

## Generating Dataset

In [2]:
```
X = np.genfromtxt('winequality-red.csv', delimiter=';')
```

In [3]:
```
train_X = X[1:1001, 0:-1]
train_Y = np.array([X[1:1001, -1]>=6]).astype(int).squeeze()
test_X = X[1001::, 0:-1]
test_Y = np.array([X[1001::, -1]>=6]).astype(int).squeeze()
```

## Decision Tree

In [4]:
```
metrics = ["gini", "entropy"]
max_depth = [3, 5, 8]
for i in metrics:
    for j in max_depth:
        clf = tree.DecisionTreeClassifier(criterion=i, max_depth=int(j)).fit(train_X, t
        train_preds = clf.predict(train_X)
        test_preds = clf.predict(test_X)
        train_acc = clf.score(train_X, train_Y)*100
        test_acc = clf.score(test_X, test_Y)*100
        print("Criterion: {}, Max Depth: {}, Training Accuracy: {}, Test Accuracy: {}.\
```

```
Criterion: gini, Max Depth: 3, Training Accuracy: 73.8, Test Accuracy: 71.6193656093489
2.

Criterion: gini, Max Depth: 5, Training Accuracy: 79.4, Test Accuracy: 72.9549248747913
2.

Criterion: gini, Max Depth: 8, Training Accuracy: 89.1, Test Accuracy: 67.9465776293823.

Criterion: entropy, Max Depth: 3, Training Accuracy: 73.4, Test Accuracy: 71.78631051752
922.

Criterion: entropy, Max Depth: 5, Training Accuracy: 77.7, Test Accuracy: 73.78964941569
282.

Criterion: entropy, Max Depth: 8, Training Accuracy: 86.9, Test Accuracy: 69.44908180300
501.
```

## K-Nearest Neighbors

In [5]:
```
n = [8, 12, 25]
for i in n:
    clf = neighbors.KNeighborsClassifier(n_neighbors=i).fit(train_X, train_Y)
```

# Q2



```
    train_preds = clf.predict(train_X)
    test_preds = clf.predict(test_X)
    train_acc = clf.score(train_X, train_Y)*100
    test_acc = clf.score(test_X, test_Y)*100
    print("N-neighbors: {}, Training Accuracy: {}, Test Accuracy: {}.\n".format(i, trai
```

```
N-neighbors: 8, Training Accuracy: 75.5, Test Accuracy: 58.931552587646074.

N-neighbors: 12, Training Accuracy: 74.1, Test Accuracy: 59.59933222036727.

N-neighbors: 25, Training Accuracy: 71.8, Test Accuracy: 62.604340567612695.
```

## Logistic Regression

In [6]:

```
train_X_wbias = np.hstack((train_X, np.ones_like(train_Y)[:, np.newaxis]))
test_X_wbias = np.hstack((test_X, np.ones_like(test_Y)[:, np.newaxis]))
reg_method = ["l1", "l2", "elasticnet"]
for i in reg_method:
    if i == "l1":
        clf = linear_model.LogisticRegression(penalty=i, solver="saga", max_iter=100000
    elif i == "elasticnet":
        clf = linear_model.LogisticRegression(penalty=i, solver="saga", l1_ratio=1, max
    else:
        clf = linear_model.LogisticRegression(penalty=i, max_iter=100000).fit(train_X_w
    train_preds = clf.predict(train_X_wbias)
    test_preds = clf.predict(test_X_wbias)
    train_acc = clf.score(train_X_wbias, train_Y)*100
    test_acc = clf.score(test_X_wbias, test_Y)*100
    print("Penalty: {}, Training Accuracy: {}, Test Accuracy: {}.\n".format(i, train_ac
```

```
Penalty: l1, Training Accuracy: 72.8, Test Accuracy: 75.45909849749583.

Penalty: l2, Training Accuracy: 73.7, Test Accuracy: 76.62771285475793.

Penalty: elasticnet, Training Accuracy: 72.8, Test Accuracy: 75.45909849749583.
```

## Support Vector Machine

In [7]:

```
kernels = ["linear", "rbf"]
Cs = [0.01, 10, 1000]
for i in kernels:
    for j in Cs:
        clf = svm.SVC(C=j, kernel=i).fit(train_X, train_Y)
        train_preds = clf.predict(train_X)
        test_preds = clf.predict(test_X)
        train_acc = clf.score(train_X, train_Y)*100
        test_acc = clf.score(test_X, test_Y)*100
        print("Kernel: {}, C: {}, Training Accuracy: {}, Test Accuracy: {}.\n".format(i
clf = svm.SVC(kernel="poly").fit(train_X, train_Y)
print("Kernel: poly, Training Accuracy: {}, Test Accuracy: {}.\n".format(train_acc, tes
```

```
Kernel: linear, C: 0.01, Training Accuracy: 72.0, Test Accuracy: 72.45409015025042.

Kernel: linear, C: 10, Training Accuracy: 73.3, Test Accuracy: 76.62771285475793.

Kernel: linear, C: 1000, Training Accuracy: 73.2, Test Accuracy: 75.79298831385643.
```

# Q2



```
Kernel: rbf, C: 0.01, Training Accuracy: 62.3, Test Accuracy: 60.60100166944908.

Kernel: rbf, C: 10, Training Accuracy: 71.5, Test Accuracy: 71.28547579298832.

Kernel: rbf, C: 1000, Training Accuracy: 77.5, Test Accuracy: 74.95826377295492.

Kernel: poly, Training Accuracy: 77.5, Test Accuracy: 74.95826377295492.
```

## Neural Network

In [8]:

```
act_fun = ["logistic", "tanh", "relu"]
for i in act_fun:
    clf = neural_network.MLPClassifier(hidden_layer_sizes=(10, 20, 10), activation=i, m
    train_preds = clf.predict(train_X_wbias)
    test_preds = clf.predict(test_X_wbias)
    train_acc = clf.score(train_X_wbias, train_Y)*100
    test_acc = clf.score(test_X_wbias, test_Y)*100
    print("Set 1: Activation Function: {}, Training Accuracy: {}, Test Accuracy: {}.\n"
for i in act_fun:
    clf = neural_network.MLPClassifier(hidden_layer_sizes=(20, 50, 10), activation=i, m
    train_preds = clf.predict(train_X_wbias)
    test_preds = clf.predict(test_X_wbias)
    train_acc = clf.score(train_X_wbias, train_Y)*100
    test_acc = clf.score(test_X_wbias, test_Y)*100
    print("Set 2: Activation Function: {}, Training Accuracy: {}, Test Accuracy: {}.\n"
for i in act_fun:
    clf = neural_network.MLPClassifier(hidden_layer_sizes=(30, 70, 10), activation=i, m
    train_preds = clf.predict(train_X_wbias)
    test_preds = clf.predict(test_X_wbias)
    train_acc = clf.score(train_X_wbias, train_Y)*100
    test_acc = clf.score(test_X_wbias, test_Y)*100
    print("Set 3: Activation Function: {}, Training Accuracy: {}, Test Accuracy: {}.\n"
```

```
Set 1: Activation Function: logistic, Training Accuracy: 75.0, Test Accuracy: 75.7929883
1385643.

Set 1: Activation Function: tanh, Training Accuracy: 75.8, Test Accuracy: 73.12186978297
161.

Set 1: Activation Function: relu, Training Accuracy: 74.4, Test Accuracy: 73.62270450751
252.

Set 2: Activation Function: logistic, Training Accuracy: 74.7, Test Accuracy: 76.1268781
3021702.

Set 2: Activation Function: tanh, Training Accuracy: 76.6, Test Accuracy: 71.28547579298
832.

Set 2: Activation Function: relu, Training Accuracy: 76.4, Test Accuracy: 72.78797996661
102.

Set 3: Activation Function: logistic, Training Accuracy: 74.7, Test Accuracy: 75.7929883
1385643.

Set 3: Activation Function: tanh, Training Accuracy: 76.4, Test Accuracy: 72.45409015025
042.

Set 3: Activation Function: relu, Training Accuracy: 75.4, Test Accuracy: 77.46243739565
```

# Q2



```
944.
```

# Problem 2b

A confusion matrix is a matrix that takes a prediction $i$ and try to categorize it with to a ground truth $j$, if $i$ and $j$ does not match, it is a false positive or negative, otherwise, it is a true positive and negative.

A precision takes components from confusion matrix (true positives and false positives) and put it in a ratio such that it represents a performance/efficacy value for the prediction algorithm.

A recall matrix components acts as a precision, but on the true positive and false negative from the confusion matrix. It is a measurement of the ability of the prediction algorithm to output positive values.

A F1 matrix is the weighted average between the recall and precision.

## Best parameters from each category

### Decision Tree: Criterion = entropy, Max Depth = 5, Training Accuracy: 77.7, Test Accuracy: 73.45575959933221.

In [12]:
```
clf = tree.DecisionTreeClassifier(criterion="entropy", max_depth=5).fit(train_X, train_
test_preds = clf.predict(test_X)
confusion = metrics.confusion_matrix(test_Y, test_preds)
precision = metrics.precision_score(test_Y, test_preds)
recall = metrics.recall_score(test_Y, test_preds)
f1 = metrics.f1_score(test_Y, test_preds)
print("Confusion Matrix =\n", confusion)
print("Precision = ", precision)
print("Recall = ", recall)
print("F1 = ", f1)
```

```
Confusion Matrix =
 [[145  94]
 [ 65 295]]
Precision =  0.7583547557840618
Recall =  0.8194444444444444
F1 =  0.787716955941255
```

### KNN: N-neighbors = 25, Training Accuracy: 71.8, Test Accuracy: 62.604340567612695.

In [13]:
```
clf = neighbors.KNeighborsClassifier(n_neighbors=25).fit(train_X, train_Y)
test_preds = clf.predict(test_X)
confusion = metrics.confusion_matrix(test_Y, test_preds)
precision = metrics.precision_score(test_Y, test_preds)
recall = metrics.recall_score(test_Y, test_preds)
f1 = metrics.f1_score(test_Y, test_preds)
print("Confusion Matrix =\n", confusion)
print("Precision = ", precision)
print("Recall = ", recall)
print("F1 = ", f1)
```

# Q2



```
Confusion Matrix =
 [[133 106]
 [118 242]]
Precision =  0.6954022988505747
Recall =  0.6722222222222223
F1 =  0.6836158192090396
```

## Logistic Regression: Penalty = l2, Training Accuracy: 73.7, Test Accuracy: 76.62771285475793.

In [14]:

```
clf = linear_model.LogisticRegression(max_iter=100000).fit(train_X_wbias, train_Y)
test_preds = clf.predict(test_X_wbias)
confusion = metrics.confusion_matrix(test_Y, test_preds)
precision = metrics.precision_score(test_Y, test_preds)
recall = metrics.recall_score(test_Y, test_preds)
f1 = metrics.f1_score(test_Y, test_preds)
print("Confusion Matrix =\n", confusion)
print("Precision = ", precision)
print("Recall = ", recall)
print("F1 = ", f1)
```

```
Confusion Matrix =
 [[156  83]
 [ 57 303]]
Precision =  0.7849740932642487
Recall =  0.8416666666666667
F1 =  0.8123324396782843
```

## Support Vector Machine: Kernel = linear, C = 10, Training Accuracy: 73.3, Test Accuracy: 76.62771285475793.

In [15]:

```
clf = svm.SVC(C=10, kernel="linear").fit(train_X, train_Y)
test_preds = clf.predict(test_X)
confusion = metrics.confusion_matrix(test_Y, test_preds)
precision = metrics.precision_score(test_Y, test_preds)
recall = metrics.recall_score(test_Y, test_preds)
f1 = metrics.f1_score(test_Y, test_preds)
print("Confusion Matrix =\n", confusion)
print("Precision = ", precision)
print("Recall = ", recall)
print("F1 = ", f1)
```

```
Confusion Matrix =
 [[170  69]
 [ 71 289]]
Precision =  0.8072625698324022
Recall =  0.8027777777777778
F1 =  0.8050139275766017
```

## Neural Network: Hidden Layer Sizes = 20, 50, 10, Activation Function = Logistic

In [16]:

```
clf = neural_network.MLPClassifier(hidden_layer_sizes=(20, 50, 10), activation=i, max_i
train_preds = clf.predict(train_X_wbias)
test_preds = clf.predict(test_X_wbias)
confusion = metrics.confusion_matrix(test_Y, test_preds)
precision = metrics.precision_score(test_Y, test_preds)
recall = metrics.recall_score(test_Y, test_preds)
```

# Q2



```
f1 = metrics.f1_score(test_Y, test_preds)
print("Confusion Matrix =\n", confusion)
print("Precision = ", precision)
print("Recall = ", recall)
print("F1 = ", f1)
```

```
Confusion Matrix =
 [[159  80]
 [ 77 283]]
Precision =  0.7796143250688705
Recall =  0.7861111111111111
F1 =  0.7828492392807745
```

## Case study

Case A: I would sell the Logistic Regression model in this case because I want to maximize the number of positive value (good wine), therefore, I should look for the model with the highest Recall Score.

Case B: I would sell the Logisitic Regression model again in this case. The customer does not want to miss too many good wines, this implies that she wants a good recall score on the good wines. However, she also does not desire when classified good wines to actually taste bad (false positives), which implies a precision scoring. Together, I conclude that the best F1 score will satisfy her the most, which the Logistic Regression model is able to offer.