In [5]:
```
#A Star Algo:
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('C', 3), ('D', 2)],
    'C': [('D', 1), ('E', 5)],
    'D': [('C', 1), ('E', 8)],
    'E': [('I', 5), ('J', 5)],
    'F': [('G', 1), ('H', 7)],
    'G': [('I', 3)],
    'H': [('I', 2)],
    'I': [('E', 5), ('J', 3)]
}

def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
def h(n):
    H_dist = {
        'A': 10,
        'B': 8,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]
def aStar(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node
    while len(open_set) > 0:
        n =  None
        for v in open_set:
            if n == None or g[v] + h(v) < g[n] + h(n):
                n = v
        if n == stop_node or Graph_nodes[v] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                if m not in open_set and m not in closed_set:
                    g[m] = g[n] + weight
                    parents[m] = n
                    open_set.add(m)
                else:
                    if g[m] > g[n] + weight:
                        parents[m] = n
                        g[m] = g[n] + weight
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print("Path does not exist")
```

```
            return None
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print("Path exists: {}".format(path))
            return path
        open_set.remove(n)
        closed_set.add(n)
aStar('A', 'J')
```

```
Path exists: ['A', 'F', 'G', 'I', 'J']
```

Out[5]:

```
['A', 'F', 'G', 'I', 'J']
```

In [11]:
```
#AO Star algo:
class Graph:
    def __init__(self, graph, heuristicNodeList, start):
        self.graph = graph
        self.H = heuristicNodeList
        self.start = start
        self.status = {}
        self.parents = {}
        self.solutionGraph = {}
    def applyAOStar(self):
        self.aoStar(self.start, False)
    def getNeighbors(self, v):
        return self.graph.get(v, '')
    def getStatus(self, v):
        return self.status.get(v, 0)
    def setStatus(self, v, val):
        self.status[v] = val
    def getHeuristicNodeValue(self, v):
        return self.H.get(v, 0)
    def setHeuristicNodeValue(self, v, val):
        self.H[v] = val
    def printSolution(self):
        print("Traverse from: ", self.start)
        print(self.solutionGraph)
    def computeMinimumCostChildNodes(self, v):
        minimumCost = 0
        minimumCostChildNodeListDict = {}
        minimumCostChildNodeListDict[minimumCost] = []
        flag = True
        for nodeListEntryTuple in self.getNeighbors(v):
            cost = 0
            nodeList = []
            for c, weight in nodeListEntryTuple:
                cost = cost + self.getHeuristicNodeValue(c) + weight
                nodeList.append(c)
            if flag == True:
                minimumCost = cost
                minimumCostChildNodeListDict[minimumCost] = nodeList
                flag = False
            else:
                if minimumCost > cost:
                    minimumCost = cost
                    minimumCostChildNodeListDict[minimumCost] = nodeList
        return minimumCost, minimumCostChildNodeListDict[minimumCost]
    def aoStar(self, v, backTracking):
        print("Heuristic Values: ", self.H)
        print("Solution Graph: ", self.solutionGraph)
        print("Processing Node: ", v)
        childNodeList = []
        if self.getStatus(v) >= 0:
            minimumCost, childNodeList = self.computeMinimumCostChildNodes(
            self.setHeuristicNodeValue(v, minimumCost)
        self.setStatus(v, len(childNodeList))
        solved = True
        for childNode in childNodeList:
            self.parents[childNode] = v
            if self.getStatus(childNode) != -1:
                solved = solved & False
        if solved == True:
            self.setStatus(v, -1)
            self.solutionGraph[v] = childNodeList
```

```
        if v != self.start:
            self.aoStar(self.parents[v], True)
        if backTracking == False:
            for childNode in childNodeList:
                self.setStatus(childNode, 0)
                self.aoStar(childNode, False)
h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
graph2 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'D': [[('E', 1)], [('F', 1)]]
}
G2 = Graph(graph2, h2, 'A')
G2.applyAOStar()
G2.printSolution()
```

```
Heuristic Values:  {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4,
'G': 5, 'H': 7}
Solution Graph:  {}
Processing Node:  A
Heuristic Values:  {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4,
'G': 5, 'H': 7}
Solution Graph:  {}
Processing Node:  D
Heuristic Values:  {'A': 11, 'B': 6, 'C': 12, 'D': 5, 'E': 4, 'F': 4,
'G': 5, 'H': 7}
Solution Graph:  {}
Processing Node:  A
Heuristic Values:  {'A': 6, 'B': 6, 'C': 12, 'D': 5, 'E': 4, 'F': 4, 'G':
5, 'H': 7}
Solution Graph:  {}
Processing Node:  E
Heuristic Values:  {'A': 6, 'B': 6, 'C': 12, 'D': 5, 'E': 0, 'F': 4, 'G':
5, 'H': 7}
Solution Graph:  {'E': []}
Processing Node:  D
Heuristic Values:  {'A': 6, 'B': 6, 'C': 12, 'D': 1, 'E': 0, 'F': 4, 'G':
5, 'H': 7}
Solution Graph:  {'E': [], 'D': ['E']}
Processing Node:  A
Traverse from:  A
{'E': [], 'D': ['E'], 'A': ['D']}
```

In [7]:
```python
import numpy as np
import pandas as pd

data = pd.read_csv('enjoysport.csv')
print(data)

concept = np.array(data.iloc[:, 0:-1])
target = np.array(data.iloc[:, -1])

def learn(concept, target):
    for i, val in enumerate(target):
        if val == "yes":
            break
    specific_h = concept[i].copy()
    generic_h = [["?" for i in range(len(specific_h))]for i in range(len(sp

    for i, h in enumerate(concept):
        if target[i] == "yes":
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    specific_h[x] = "?"
                    generic_h[x][x] = "?"
        if target[i] == "no":
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    generic_h[x][x] = specific_h[x]
                else:
                    generic_h[x][x] = "?"
    indices = [i for i, val in enumerate(generic_h) if val == ["?", "?", "

    for i in indices:
        generic_h.remove(["?", "?", "?", "?", "?", "?"])
    return specific_h, generic_h

s_final, g_final = learn(concept, target)

print("\nFinal S: ", s_final)
print("\nFinal G: ", g_final)
```

```
     sky air_temp humidity   wind water forecast enjoy_sport
0  sunny     warm   normal strong  warm     same         yes
1  sunny     warm     high strong  warm     same         yes
2  rainy     cold     high strong  warm   change          no
3  sunny     warm     high strong  cool   change         yes

Final S:  ['sunny' 'warm' '?' 'strong' '?' '?']

Final G:  [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?',
'?', '?']]
```

In [16]:
```
#ID3 algo:
import numpy as np
import pandas as pd
import math

data = pd.read_csv('Book1.csv')
features = [feat for feat in data]
features.remove('Classification')

class Node:
    def __init__(self):
        self.children = []
        self.value = ""
        self.isLeaf = False
        self.pred = ""

def entropy(examples):
    pos = 0.0
    neg = 0.0
    for _, row in examples.iterrows():
        if row["Classification"] == "Yes":
            pos += 1
        else:
            neg += 1
    if pos == 0.0 or neg == 0.0:
        return 0.0
    else:
        p = pos / (pos+neg)
        n = neg / (pos+neg)
        return -((p*math.log(p, 2))+(n*math.log(n, 2)))

def info_gain(examples, attr):
    uniq = np.unique(examples[attr])
    gain = entropy(examples)
    for u in uniq:
        subdata = examples[examples[attr] == u]
        sub_e = entropy(subdata)
        gain -= ((float(len(subdata)))/(float(len(examples))))*sub_e
    return gain

def ID3(examples, attrs):
    root = Node()
    max_feat = ""
    max_gain = 0

    for feature in attrs:
        gain = info_gain(examples, feature)
        if gain > max_gain:
            max_gain = gain
            max_feat = feature
    root.value = max_feat
    uniq = np.unique(examples[max_feat])
    for u in uniq:
        subdata = examples[examples[max_feat] == u]
        if entropy(subdata) == 0.0:
            newNode = Node()
            newNode.value = u
            newNode.isLeaf = True
            newNode.pred = np.unique(subdata["Classification"])
            root.children.append(newNode)
        else:
```

```
            dummyNode = Node()
            dummyNode.value = u
            new_attrs = attrs.copy()
            new_attrs.remove(max_feat)
            child = ID3(subdata, new_attrs)
            dummyNode.children.append(child)
            root.children.append(dummyNode)
    return root
def printTree(root: Node, depth = 0):
    for i in range(depth):
        print("\t", end="")
    print(root.value, end="")
    if root.isLeaf:
        print("->", root.pred)
    print()
    for child in root.children:
        printTree(child, depth+1)
root = ID3(data, features)
printTree(root)
```

```
A3
    High
        A1
            False
                A2
                    Cool-> ['No']

                    Hot-> ['Yes']

            True-> ['No']

    Normal-> ['Yes']
```

In [11]:
```
#Backpropagation
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype = float)
y = np.array(([92], [86], [89]), dtype = float)
X = X/np.amax(X, axis = 0)
y = y/100

def sigmoid(x):
    return (1/1+np.exp(-x))
def derivative_sigmoid(x):
    return x*(1-x)

epoch = 7000
lr = 0.1
inputlayer_neurons = 2
hiddenlayer_neurons = 3
outputlayer_neurons = 1
wh = np.random.uniform(size=(inputlayer_neurons, hiddenlayer_neurons))
bh = np.random.uniform(size=(1, hiddenlayer_neurons))
wout = np.random.uniform(size=(hiddenlayer_neurons, outputlayer_neurons))
bout = np.random.uniform(size=(1, outputlayer_neurons))

for i in range(epoch):
    hinp1 = np.dot(X, wh)
    hinp = hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1 = np.dot(hlayer_act, wout)
    outinp = outinp1 + bout
    output = sigmoid(outinp)
    EO = y - output
    outputgrad = derivative_sigmoid(output)
    d_output = EO*outputgrad
    EH = d_output.dot(wout.T)
    hiddengrad = derivative_sigmoid(hlayer_act)
    d_hidden = EH * hiddengrad
    wout += hlayer_act.T.dot(d_output)*lr
    bout += np.sum(d_output, axis = 0, keepdims = 0)*lr
    wh += X.T.dot(d_hidden)*lr
print("Input: ", str(X))
print("Actual Output: ", str(y))
print("Predicted Output: ", output)
```

```
Input:  [[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:  [[0.92]
 [0.86]
 [0.89]]
Predicted Output:  [[1.00047642]
 [1.0003971 ]
 [1.00067038]]
```

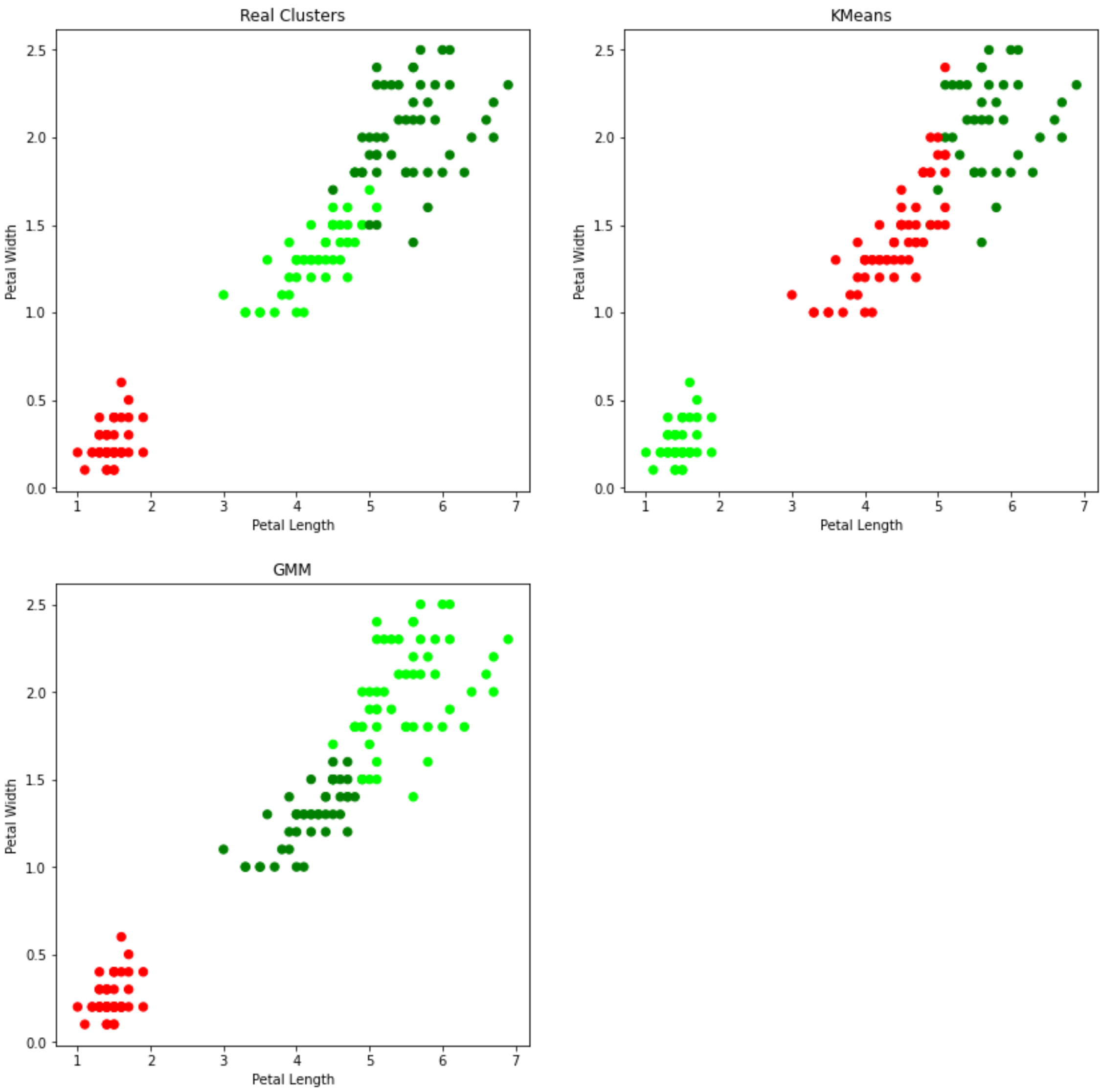
In [12]:
```
#Naive Bayes Classifier
import csv
import random
import math
def loadCsv(filename):
    lines = csv.reader(open(filename, "r"));
    dataset = list(lines)
    for i in range(len(dataset)):
        dataset[i] = [float(x) for x in dataset[i]]
    return dataset
def splitDataset(dataset, splitRatio):
    trainSize = int(len(dataset) * splitRatio);
    trainSet = [ ]
    copy = list(dataset);
    while len(trainSet) < trainSize:
        index = random.randrange(len(copy));
        trainSet.append(copy.pop(index))
    return [trainSet, copy]
def separateByClass(dataset):
    separated = {}
    for i in range(len(dataset)):
        vector = dataset[i]
        if (vector[-1] not in separated):
            separated[vector[-1]] = []
        separated[vector[-1]].append(vector)
    return separated
def mean(numbers):
    return sum(numbers)/float(len(numbers))
def stdev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)
    return math.sqrt(variance)
def summarize(dataset):
    summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*
    del summaries[-1]
    return summaries
def summarizeByClass(dataset):
    separated = separateByClass(dataset);
    summaries = {}
    for classValue, instances in separated.items():
        summaries[classValue] = summarize(instances)
    return summaries
def calculateProbability(x, mean, stdev):
    exponent = math.exp(-(math.pow(x-mean,2)/(2*math.pow(stdev,2))))
    return (1 / (math.sqrt(2*math.pi) * stdev)) * exponent
def calculateClassProbabilities(summaries, inputVector):
    probabilities = {}
    for classValue, classSummaries in summaries.items():
        probabilities[classValue] = 1
        for i in range(len(classSummaries)):
            mean, stdev = classSummaries[i]
            x = inputVector[i]
            probabilities[classValue] *= calculateProbability(x, mean, stde
    return probabilities
def predict(summaries, inputVector):
    probabilities = calculateClassProbabilities(summaries, inputVector)
    bestLabel, bestProb = None, -1
    for classValue, probability in probabilities.items():
        if bestLabel is None or probability > bestProb:
            bestProb = probability
            bestLabel = classValue
```

```
    return bestLabel
def getPredictions(summaries, testSet):
    predictions = []
    for i in range(len(testSet)):
        result = predict(summaries, testSet[i])
        predictions.append(result)
    return predictions
def getAccuracy(testSet, predictions):
    correct = 0
    for i in range(len(testSet)):
        if testSet[i][-1] == predictions[i]:
            correct += 1
        return (correct/float(len(testSet))) * 100.0
def main():
    filename = 'nb.csv'
    splitRatio = 0.67
    dataset = loadCsv(filename);
    trainingSet, testSet = splitDataset(dataset, splitRatio)
    print('Split {0} rows into train={1} and test={2} rows'.format(len(data
    summaries = summarizeByClass(trainingSet)
    predictions = getPredictions(summaries, testSet)
    accuracy = getAccuracy(testSet, predictions)
    print('Accuracy of the classifier is : {0}%'.format(accuracy))
main()
```

```
Split 767 rows into train=513 and test=254 rows
Accuracy of the classifier is : 0.39370078740157477%
```

In [13]:
```
#EM based GMM and KMeans
from sklearn import datasets, preprocessing
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

iris = datasets.load_iris()
X = pd.DataFrame(iris.data)
X.columns = ["Sepal_Length", "Sepal_Width", "Petal_Length", "Petal_Width"]
y = pd.DataFrame(iris.target)
y.columns = ["Targets"]

model = KMeans(n_clusters = 3)
model.fit(X)
plt.figure(figsize = (14, 14))
colormap = np.array(["red", "lime", "green"])
plt.subplot(2, 2, 1)
plt.title("Real Clusters")
plt.scatter(X.Petal_Length, X.Petal_Width, c = colormap[y.Targets], s = 40)
plt.xlabel("Petal Length")
plt.ylabel("Petal Width")
plt.subplot(2, 2, 2)
plt.title("KMeans")
plt.scatter(X.Petal_Length, X.Petal_Width, c = colormap[model.labels_], s =
plt.xlabel("Petal Length")
plt.ylabel("Petal Width")

scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)

gmm = GaussianMixture(n_components = 3)
gmm.fit(xs)
gmm_y = gmm.predict(xs)
plt.subplot(2, 2, 3)
plt.title("GMM")
plt.scatter(X.Petal_Length, X.Petal_Width, c = colormap[gmm_y], s = 40)
plt.xlabel("Petal Length")
plt.ylabel("Petal Width")
```

Out[13]:
```
Text(0, 0.5, 'Petal Width')
```

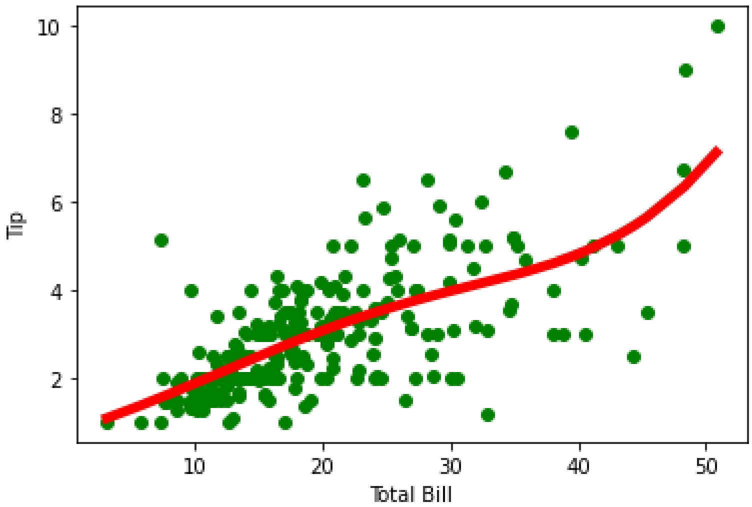

Real Clusters
KMeans
GMM
Petal Width
Petal Length

In [14]:
```
#KNearest Neighbors
from sklearn.datasets import load_iris
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

iris_dataset = load_iris()
target = iris_dataset.target_names
print("Class: number\n")
for i in range(len(target)):
    print(target[i], ": ", i)

X_train, X_test, Y_train, Y_test = train_test_split(iris_dataset["data"], i

kn = KNeighborsClassifier(1)
kn.fit(X_train, Y_train)
for i in range(len(X_test)):
    x_new = np.array([X_test[i]])
    prediction = kn.predict(x_new)
    print("Actual: [{0}][{1}], Predicted: {2}{3}".format(Y_test[i], target[
print("\nAccuracy: ", kn.score(X_test, Y_test))
```

```
Class: number

setosa :  0
versicolor :  1
virginica :  2
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [1][versicolor], Predicted: [2]['virginica']
Actual: [0][setosa], Predicted: [0]['setosa']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [0][setosa], Predicted: [0]['setosa']
Actual: [0][setosa], Predicted: [0]['setosa']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [0][setosa], Predicted: [0]['setosa']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [0][setosa], Predicted: [0]['setosa']
Actual: [0][setosa], Predicted: [0]['setosa']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [0][setosa], Predicted: [0]['setosa']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [0][setosa], Predicted: [0]['setosa']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [2][virginica], Predicted: [1]['versicolor']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [0][setosa], Predicted: [0]['setosa']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [2][virginica], Predicted: [2]['virginica']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [0][setosa], Predicted: [0]['setosa']
Actual: [1][versicolor], Predicted: [1]['versicolor']
Actual: [1][versicolor], Predicted: [2]['virginica']

Accuracy:  0.9210526315789473
```

In [15]:

```
#Locally Weighted Regression
import matplotlib.pyplot as plt
import numpy as np

def kernel(point, xmat, k):
    m, n = np.shape(xmat)
    weights = np.mat(np.eye(m))
    for j in range(m):
        diff = point - X[j]
        weights[j, j] = np.exp(diff*diff.T/(-2.0*k**2))
    return weights

def localWeight(point, xmat, ymat, k):
    wei = kernel(point, xmat, k)
    W = (X.T*(wei*X)).I*(X.T*(wei*ymat.T))
    return W

def localWeightRegression(xmat, ymat, k):
    m, n = np.shape(xmat)
    ypred = np.zeros(m)
    for i in range(m):
        ypred[i] = xmat[i] * localWeight(xmat[i], xmat, ymat, k)
    return ypred

def graphPlot(X, ypred):
    sortindex = X[:, 1].argsort(0)
    xsort = X[sortindex][:, 0]
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.scatter(bill, tip, color = "green")
    ax.plot(xsort[:, 1], ypred[sortindex], color = "red", linewidth = 5)
    plt.xlabel("Total Bill")
    plt.ylabel("Tip")
    plt.show()

data = pd.read_csv('data10_tips.csv')
bill = np.array(data.total_bill)
tip = np.array(data.tip)
mbill = np.mat(bill)
mtip = np.mat(tip)
m = np.shape(mbill)[1]
one = np.mat(np.ones(m))
X = np.hstack((one.T, mbill.T))
ypred = localWeightRegression(X, mtip, 8)
graphPlot(X, ypred)
```



In [ ]: