problem statement2:

In [21]:
```
#Import dataset
import numpy as np
import pandas as pd
dataset = pd.read_csv('Social_Network_Ads.csv')
dataset
```

Out[21]:

	User ID	Gender	Age	EstimatedSalary	Purchased
0	15624510	Male	19	19000	0
1	15810944	Male	35	20000	0
2	15668575	Female	26	43000	0
3	15603246	Female	27	57000	0
4	15804002	Male	19	76000	0
...	...	...	...	...	...
395	15691863	Female	46	41000	1
396	15706071	Male	51	23000	1
397	15654296	Female	50	20000	1
398	15755018	Male	36	33000	0
399	15594041	Female	49	36000	1

400 rows × 5 columns

In [22]:
```
#Create a matrix of independent variables
X = dataset.iloc[:,[2,3]].values

#Create an array of dependent variable
y = dataset.iloc[:,4].values
```

In [23]:
```
#Splitting the dataset into test set and training set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
```

In [24]:
```
#feature the scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
```

In [25]:
```
from pprint import pprint
import scipy.stats as sps

dataset = pd.read_csv('Social_Network_Ads.csv',header=None)
dataset = dataset.sample(frac=1)
dataset.columns = ['User ID','Gender','Age','EstimatedSalary','Purchased']
```

In [26]:
```
def entropy(target_col):
    elements,counts = np.unique(target_col,return_counts = True)
    entropy = np.sum([(-counts[i]/np.sum(counts))*np.log2(counts[i]/np.sum(counts)) for i in range(len(elements))])
    return entropy
```

In [27]:
```
def InfoGain(data,split_attribute_name,target_name="Purchased"):

    #Calculate the entropy of the total dataset
    total_entropy = entropy(data[target_name])


    #Calculate the values and the corresponding counts for the split attribute
    vals,counts= np.unique(data[split_attribute_name],return_counts=True)

    #Calculate the weighted entropy
    Weighted_Entropy = np.sum([(counts[i]/np.sum(counts))*entropy(data.where(data[split_attribute_name]==vals[i]).dropna()[ta

    #Calculate the information gain
    Information_Gain = total_entropy - Weighted_Entropy
    return Information_Gain
```

In [28]:
```python
def ID3(data,originaldata,features,target_attribute_name="Purchased",parent_node_class = None):
    #Define the stopping criteria --> If one of this is satisfied, we want to return a leaf node#

    #If all target_values have the same value, return this value
    if len(np.unique(data[target_attribute_name])) <= 1:
        return np.unique(data[target_attribute_name])[0]

    #If the dataset is empty, return the mode target feature value in the original dataset
    elif len(data)==0:
        return np.unique(originaldata[target_attribute_name])[np.argmax(np.unique(originaldata[target_attribute_name],return_


    elif len(features) ==0:
        return parent_node_class

    #If none of the above holds true, grow the tree!

    else:
        #Set the default value for this node --> The mode target feature value of the current node
        parent_node_class = np.unique(data[target_attribute_name])[np.argmax(np.unique(data[target_attribute_name],return_cou



        features = np.random.choice(features,size=np.int(np.sqrt(len(features))),replace=False)

        #Select the feature which best splits the dataset
        item_values = [InfoGain(data,feature,target_attribute_name) for feature in features] #Return the information gain val
        best_feature_index = np.argmax(item_values)
        best_feature = features[best_feature_index]

        #Create the tree structure. The root gets the name of the feature (best_feature) with the maximum information
        #gain in the first run
        tree = {best_feature:{}}

        #Remove the feature with the best inforamtion gain from the feature space
        features = [i for i in features if i != best_feature]


        #Grow a branch under the root node for each possible value of the root node feature

        for value in np.unique(data[best_feature]):
            value = value
            #Split the dataset along the value of the feature with the largest information gain and therwith create sub_datas
            sub_data = data.where(data[best_feature] == value).dropna()

            #Call the ID3 algorithm for each of those sub_datasets with the new parameters --> Here the recursion comes in!
            subtree = ID3(sub_data,dataset,features,target_attribute_name,parent_node_class)

            #Add the sub tree, grown from the sub_dataset to the tree under the root node
            tree[best_feature][value] = subtree

        return(tree)
```

In [29]:
```python
def predict(query,tree,default = 'p'):

    for key in list(query.keys()):
        if key in list(tree.keys()):
            try:
                result = tree[key][query[key]]
            except:
                return default
            result = tree[key][query[key]]
            if isinstance(result,dict):
                return predict(query,result)

            else:
                return result
```

In [30]:
```python
def train_test_split(dataset):
    training_data = dataset.iloc[:round(0.75*len(dataset))].reset_index(drop=True)
    #We drop the index respectively relabel the index
    #starting 0, because we do not want to run into errors regarding the row labels / indexes
    testing_data = dataset.iloc[round(0.75*len(dataset)):].reset_index(drop=True)
    return training_data,testing_data


training_data = train_test_split(dataset)[0]
testing_data = train_test_split(dataset)[1]
```

In [31]:

```python
#Train the Random Forest model

def RandomForest_Train(dataset,number_of_Trees):
    #Create a list in which the single forests are stored
    random_forest_sub_tree = []

    #Create a number of n models
    for i in range(number_of_Trees):
        #Create a number of bootstrap sampled datasets from the original dataset
        bootstrap_sample = dataset.sample(frac=1,replace=True)

        #Create a training and a testing datset by calling the train_test_split function
        bootstrap_training_data = train_test_split(bootstrap_sample)[0]
        bootstrap_testing_data = train_test_split(bootstrap_sample)[1]


        #Grow a tree model for each of the training data
        #We implement the subspace sampling in the ID3 algorithm itself. Hence take a look at the ID3 algorithm above!
        random_forest_sub_tree.append(ID3(bootstrap_training_data,bootstrap_training_data,bootstrap_training_data.drop(labels

    return random_forest_sub_tree


random_forest = RandomForest_Train(dataset,50)
```

In [12]:

```python
#Predict a new query instance
def RandomForest_Predict(query,random_forest,default='p'):
    predictions = []
    for tree in random_forest:
        predictions.append(predict(query,tree,default))
    return sps.mode(predictions)[0][0]


query = testing_data.iloc[0,:].drop('Purchased').to_dict()
query_target = testing_data.iloc[0,0]
#print('target: ',query_target)
prediction = RandomForest_Predict(query,random_forest)
#print('prediction: ',prediction)
```

In [13]:

```python
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
classifier.fit(X_train, y_train)
```

Out[13]:

```
RandomForestClassifier(criterion='entropy', n_estimators=10, random_state=0)
```

In [14]:

```python
y_pred = classifier.predict(X_test)
```

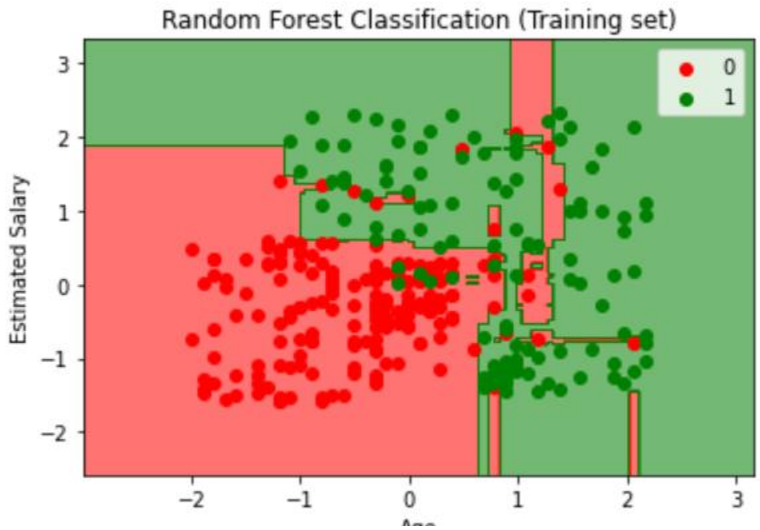
In [19]:

```python
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
```

In [18]:

```python
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.55, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Random Forest Classification (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence
in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with a single row
if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence
in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with a single row
if you intend to specify the same RGB or RGBA value for all points.
```



In [20]:

```
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.55, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Random Forest Classification (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.
```