In [ ]:
```
import numpy as np
import pandas as pd
import math
import random as rand

data = pd.read_csv("titanic_data.csv")
df = pd.DataFrame(data)
```

In [ ]:
```
# Q1
df.describe()

# Passengers in Class 1 and 2 have value 1, and Passengers in Class 3 have valu
# Passengers in first and second class have higher status, and are more likely
df['Pclass'] = np.where(df['Pclass'] < 3, 1, 0)

# Passengers no more than 28 years old have value 1, and older Passengers value
# Younger Passengers tend to be more likely to survive because of physical stre
# Choose 28 as the median of all ages
df['Age'] = np.where(df['Age'] <= 28, 1, 0)

# Single passengers have value 0, while passengers with any other relatives hav
# Being single or being with family is critical to survival, but increase in fa
df['Siblings/Spouses Aboard'] = np.where(df['Siblings/Spouses Aboard'] == 0, 1,
df['Parents/Children Aboard'] = np.where(df['Parents/Children Aboard'] == 0, 1,

# Passengers with fare greater than $14.45 have value 1, and others have value
# The ability to purchase a higher fare is directly related to higher status ar
df['Fare'] = np.where(df['Fare'] >= 14.45, 1, 0)

df.describe()
```

In [ ]:
```
X = np.array(df.iloc[:,1:])
y = np.array(df.iloc[:,0])
np.count_nonzero(X[1])
```

In [ ]:
```
# Q2 (Compute Mutual Information)
def entropy(X, index):
    """
    In:
    X: feature vector
    index: index of a feature

    Out:
    H(x): entropy of a feature x
    """

    # Variable to return entropy
    entropy = 0
    values = [0, 1]
    for value in values:
        px = np.size(np.where(X[:, index] == value)) / len(X)
        entropy -= px * math.log2(px)
    return entropy

def entropy_y(y):
    """
    In:
```

```
    y: response vector
    index: index of a feature

    Out:
    H(x): entropy of a feature x
    """

    # Variable to return entropy
    entropy = 0

    # Get uniques values of feature x, which are 0 and 1
    values = [0, 1]

    for value in values:
        py = np.size(np.where(y == value)) / len(y)
        entropy -= py * math.log2(py)
    return entropy

def conditional_entropy(X, index, y):
    """
    In:
    X: feature vector
    index: index of a feature
    y: response vector

    Out:
    H(x, y): conditional entropy of sample x with respect to y
    """

    conditional_entropy = 0
    # Get uniques values of feature x and response y, which are both 0 and 1
    values_x = set(X[:, index])
    values_y = set(Y)

    for value_x in values_x:
        for value_y in values_y:
            # Compute the cross entropy between x and y
            pxy = len(np.where(np.in1d(np.where(X[:, index]==value_x),
                                np.where(y==value_y))==True)) / len(X)

            # Calculate the conditional entropy from single and cross entropy
            conditional_entropy -= pxy * math.log2(pxy/(entropy(X, index)*entro
    return conditional_entropy


def mutual_information(X, index, y):
    """
    In:
    X: feature vector
    index: index of a feature
    y: response vector

    Out:
    I(x, y): mutual information of sample x
    """

    # Calculate entropy and conditional entropy
    H_x = entropy(X, index)
    H_xy = conditional_entropy(X, index, y)
```

```python
    # Calculate Mutual Information
    I_xy = H_x - H_xy
    return  I_xy
```

In [ ]:

```python
# Q3 (Build a Decision Tree)
def split(X,index):
    """
    In:
    X: feature vector
    index: index of the feature to split data on

    Out:
    set 0, set 1: two data sets based on the value of X[index]
    """
    split_function = lambda X:X[index] == 0
    set0 = [x for x in X if split_function(x)]
    set1 = [x for x in X if not split_function(x)]
    return(set0, set1)

class node:
    def __init__(self, feature = None, leftnode = None, rightnode = None, end =
        self.feature = feature
        self.leftnode = leftnode
        self.rightnode = rightnode
        self.end = False
        self.result = None

class tree:
    def __init__(self, X=None, y=None):
        self.root = None
        self.decisiontree(X,y)

    def decisiontree(self, X, y):
        self.root = self.build_node(X,y,np.arange(len(X[0])))
        self.graph = None

    def build_node(self, X, y, indices):
        # Create a node
        n = node()

        # Stopping Conditions:
        # 1: entropy of y is close to zero
        # 2: the sample size is smaller than 5% of the total data
        # 3: there are no more features left

        if entropy_y(y)<1e-5 or len(X)/887 < 0.05 or len(indices)==0:
            node.end = True # It is a leaf
            node.result = int(np.mean(y) > 0.5) # Check if there are more 0s or
        else:
            node.end=False # It is not a leaf

            list_info = [mutual_information(X, index, y) for index in indices]
            feature_index = list_info.index(np.amax(list_info)) # Find the inde
            node.feature = feature_index

            X0, X1 = split(X, feature_index) # Split the data
            y0 = y[np.where(X[:,indices[feature_index]] == 0)[0]]
            y1 = y[np.where(X[:,indices[feature_index]] == 1)[0]]
            new_indices = np.delete(indices, feature_index) # Delete the curren
```

```
            node.leftnode = self.build_node(X0, y0, new_indices) # Recursively
            node.rightnode = self.build_node(X1, y1, new_indices) # Recursively
        return node

    def check(self,node,x):
        if node.isend:
              return node.result # Return value of the leaf: 0 or 1
        if x[node.feature] == 0 :
            return self.check(node.leftnode,x) # Recursively check leftnodes
        else:
            return self.check(node.rightnode,x) # Recursivley check rightnodes

    def classify(self,x):
        if np.ndim(x)== 1:
            return self.check(self.root,x)
        if np.ndim(x)== 2:
            y = np.zeros(len(X[0]))
            for i in range(len(X[0])):
                y[i]=self.check(self.root, x[i])
        return y

    def graph(tree,indent = ''):
        if tree.results!=None:
            print str(tree.results)
        print indent + "0: ",
        graph(tree.leftnode,indent+" ") # Recursion
        print indent + "1: ",
        graph(tree.rightnode,indent+" ") # Recursion
```

In [ ]:

```
# Q4
t = tree(X, y)
```

In [ ]:

```
# Q5 (10-fold cross validation)

def cv(X, folds):
    X_split = []
    fold_size = len(X[0]) // folds # Calculate the fold size

    for i in range(folds):
        fold = []
        while len(fold) < fold_size:
            j = rand.randrange(len(X[0])) # Choose a random element
            index = X.index[j] # Find the index
            fold.append(X[index])
            X = X.drop(index) # Drop the data with the index
        X_split.append(fold) # Concatenate subgroups of data
    return X_split

def kfold(X, y, k=10):
    Xy = np.concatenate((X,y), axis = 0)
    total = cv(Xy,k)
    accuracy = []

    for i in range(k):
        k_list = np.arange(k)
        k_list = np.delete(k_list, i)
        for j in k_list :
            if j == k_list[0]:
                cv = total[j]
```

```
            cv = np.concatenate((cv,total[j]), axis=0)

        t = tree(X,y)
        test = t.classify(cv)
        a = np.sum(test == Xy[-1])

        acc = a/len(test) # Calculate accuracy of each prediction
        result.append(a/len(test))
    return accuracy
```

Q5

The accuracy is about 80%.

In [ ]:
```
# Q6
x = np.array([3, 1, 30, 0, 0, 100])
t = tree(X, y)
t.classify(x)
```

In [ ]:
```
# Q7 (Random Forests)

# Choose a subsample of 20% of total data
def subset(X, ratio=0.2):
    subset = list()
    n = int(len(X) * ratio)
    while len(subset) < n:
        index = rand.randrange(len(X))
        subset.append(X[index])
    return subset

# Use consensus to decide the prediction
def predict(trees, x):
    for tree in trees:
        predictions = [tree.classify(x)]
    return max(set(predictions))

def random_forest(training, testing, y, n_trees = 5):
    trees = list()
    for i in range(n_trees):
        sample = subset(training)
        t = tree(training, y)
        trees.append(t)
    predictions = [predict(trees, x) for x in testing]
    return(predictions)
```

Q9

All the predictions have shown that, with the choice of x, I would have survived the Titanic. I would prefer to use Logistic Regression because the impaired robustness of our implementation of Random Forest, since we only have all binary variables instead of categorical variables.

Q10

$H(x) = \sum_{\mathtt{x} \in X} P(x = \mathtt{x}) \log_2(\frac{1}{P(x=\mathtt{x})}) = \sum_{\mathtt{x} \in X} \sum_{\mathtt{y} \in Y} P(x = \mathtt{x}, y = \mathtt{y}) \log_2(\frac{1}{P(x=\mathtt{x})})$

$H(x|y) = \sum_{\mathtt{x} \in X} \sum_{\mathtt{y} \in Y} P(x = \mathtt{x}, y = \mathtt{y}) \log_2(\frac{1}{P(x=\mathtt{x}|y=\mathtt{y})})$

$I(x; y) = H(x) - H(x|y) = \sum_{\mathtt{x} \in X} \sum_{\mathtt{y} \in Y} P(x = \mathtt{x}, y = \mathtt{y}) \log_2(\frac{P(x=\mathtt{x}|y=\mathtt{y})}{P(x=\mathtt{x})}) = \sum_{\mathtt{x} \in X} \sum_{\mathtt{y} \in}$

$H(y) = \sum_{\mathtt{y} \in Y} P(y = \mathtt{y}) \log_2(\frac{1}{P(y=\mathtt{y})}) = \sum_{\mathtt{y} \in Y} \sum_{\mathtt{x} \in X} P(y = \mathtt{y}, x = \mathtt{x}) \log_2(\frac{1}{P(y=\mathtt{y})})$

$H(y|x) = \sum_{\mathtt{y} \in Y} \sum_{\mathtt{x} \in X} P(y = \mathtt{y}, x = \mathtt{x}) \log_2(\frac{1}{P(y=\mathtt{y}|x=\mathtt{x})})$

$I(y; x) = H(y) - H(y|x) = \sum_{\mathtt{y} \in Y} \sum_{\mathtt{x} \in X} P(y = \mathtt{y}, x = \mathtt{x}) \log_2(\frac{P(y=\mathtt{y}|x=\mathtt{x})}{P(y=\mathtt{y})}) = \sum_{\mathtt{y} \in Y} \sum_{\mathtt{x} \in}$

Therefore, $I(x; y) = I(y; x)$