# hw4_DecTrees

March 4, 2024

**Name: Kefan Zheng**

**StudentId: 9086175008**

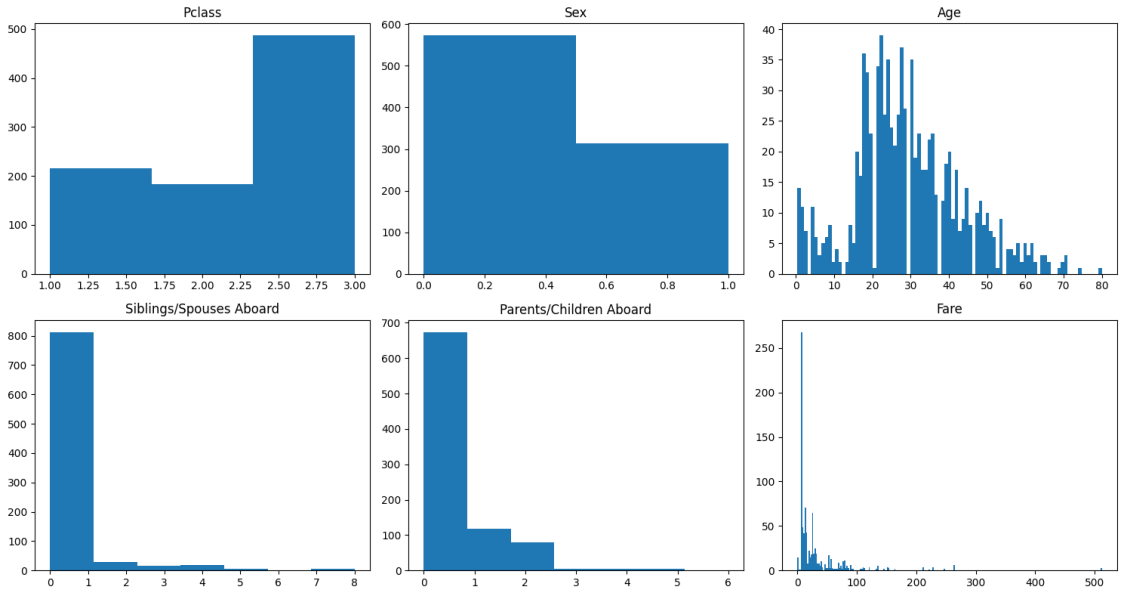
**Email: kzheng58@wisc.edu**

## 1 Problem 4.1

Display each feature's statistics first and transform the feature into a binary variable.
**Passenger Class**: It contains three classes (1,2,3). So One-hot encoding method is used to create a new binary variable for each class. For each sample, the variable value of the corresponding class is 1 and the variable value of all other classes is 0.
**Gender**: It is already a binary variable, so no extra transformation is required.
**Age**: It has many different values, so the median age is selected here as the critical point for binarization. Ages less than the median are assigned 0, and ages greater than the median are assigned 1. In addition, the median of all samples is 28 here, which also contains practical significance for distinguishing young people from older people.
**Siblings/Spouses**: The Sample with no siblings/spouse aboard is assigned 0, and the other samples regardless of how many siblings/spouse aboard are assigned 1.
**Parents/Children**: The Sample with no parents/children aboard is assigned 0, and the other samples regardless of how many parents/children aboard are assigned 1.
**Fare**: The median fare is selected as the critical point for binarization. Fares less than the median are assigned 0, and fares greater than the median are assigned 1.

[1]:
```
import numpy as np
import pandas as pd
from graphviz import Digraph
import matplotlib.pyplot as plt

def displayFeatureStatastics(X):
    _, ax = plt.subplots(2, 3, figsize=(15, 8))

    features = X.columns.tolist()
    for i, feature in enumerate(features):
        ax[i//3, i%3].hist(X[feature], bins=len(X[feature].value_counts()))
        ax[i//3, i%3].set_title(feature)
```

```
    plt.tight_layout()
    plt.show()

    for feature in features:
        print(X[feature].value_counts())

def loadDataSet():
    # read data from file
    df = pd.read_csv('titanic_data.csv')
    # split features and label
    X = df.drop(['Survived'], axis=1)
    y = df['Survived']

    # display data statistics
    displayFeatureStatastics(X)

    # transform each of your features into a binary variable
    # passenger class (one-hot)
    one_hot_column = pd.get_dummies(X['Pclass'], prefix='Pclass').
 ↪astype('int64')
    X = one_hot_column.join(X)
    X = X.drop(['Pclass'], axis=1)
    # age
    median_age = X['Age'].median()
    X['Age'] = X['Age'].apply(lambda x: 0 if x < median_age else 1)
    # siblings/spouses
    X['Siblings/Spouses Aboard'] = X['Siblings/Spouses Aboard'].apply(lambda x:␣
 ↪0 if x == 0 else 1)
    # parents/children
    X['Parents/Children Aboard'] = X['Parents/Children Aboard'].apply(lambda x:␣
 ↪0 if x == 0 else 1)
    # fare
    median_fare = X['Fare'].median()
    X['Fare'] = X['Fare'].apply(lambda x: 0 if x < median_fare else 1)

    return X, y

# load data
X, y = loadDataSet()
print("X:\n", X.head())
print("y:\n", y.head())
```




```
Pclass
3    487
1    216
2    184
Name: count, dtype: int64
Sex
0    573
1    314
Name: count, dtype: int64
Age
22.00    39
28.00    37
18.00    36
21.00    34
24.00    34
         ..
0.92      1
23.50     1
36.50     1
55.50     1
74.00     1
Name: count, Length: 89, dtype: int64
Siblings/Spouses Aboard
0    604
1    209
2     28
4     18
3     16
```

```
8      7
5      5
Name: count, dtype: int64
Parents/Children Aboard
0    674
1    118
2     80
5      5
3      5
4      4
6      1
Name: count, dtype: int64
Fare
8.0500     43
13.0000    42
7.8958     36
7.7500     33
26.0000    31
           ..
35.0000     1
28.5000     1
6.2375      1
14.0000     1
10.5167     1
Name: count, Length: 248, dtype: int64
X:
    Pclass_1  Pclass_2  Pclass_3  Sex  Age  Siblings/Spouses Aboard  \
0         0         0         1    0    0                         1
1         1         0         0    1    1                         1
2         0         0         1    1    0                         0
3         1         0         0    1    1                         1
4         0         0         1    0    1                         0

   Parents/Children Aboard  Fare
0                        0     0
1                        0     1
2                        0     0
3                        0     1
4                        0     0
y:
 0    0
1    1
2    1
3    1
4    0
Name: Survived, dtype: int64
```

## 2 Problem 4.2

[2]:
```python
# define data structure
class Node:
    def __init__(self):
        # split feature
        self.feature = None
        # children
        self.left = None # 0
        self.right = None # 1
        # leaf node
        self.is_leaf = False

class DecisionTree:
    def __init__(self):
        self.root = None

    def entropy(self, x):
        n = x.shape[0]
        entropy = 0
        for k in x.value_counts().keys():
            p_k = x.value_counts()[k] / n
            entropy += p_k * np.log2(1 / p_k)
        return entropy

    # calculate mutual information between x and y
    def mutualInfo(self, x, y):
        # calculate entropy of x
        entropy_x = self.entropy(x)

        # calculate entropy of x|y
        n = x.shape[0]
        entropy_x_y = 0
        for k in y.value_counts().keys():
            num_y_k = y.value_counts()[k]
            p_y_k = num_y_k / n
            x_y_k = x[y == k]
            entropy_x_y_k = 0
            for j in x_y_k.value_counts().keys():
                num_x_j_y_k = x_y_k.value_counts()[j]
                p_x_j_y_k = num_x_j_y_k / num_y_k
                entropy_x_y_k += p_x_j_y_k * np.log2(1 / p_x_j_y_k)
            entropy_x_y += p_y_k * entropy_x_y_k

        # calculate mutual information
        mutual_info = entropy_x - entropy_x_y
        return mutual_info
```

```
    # choose the feature with the highest mutual information
    def getBestFeature(self, X, y):
        max_mutual_info = 0
        best_feature = None
        # look each feature
        for feature in X:
            mutual_info = self.mutualInfo(X[feature], y)
            if mutual_info > max_mutual_info:
                max_mutual_info = mutual_info
                best_feature = feature

        return best_feature

    def splitData(self, X, y, best_feature):
        left_index = (X[best_feature] == 0)
        right_index = (X[best_feature] == 1)
        X = X.drop([best_feature], axis=1)
        # left subset
        left_X = X[left_index]
        left_y = y[left_index]
        # right subset
        right_X = X[right_index]
        right_y = y[right_index]
        return left_X, left_y, right_X, right_y

    def ifFinish(self, X, y):
        # check if it is a leaf node
        if y.shape[0] < 44 or (self.entropy(y) <= 0.01) or all(self.
↪mutualInfo(X[feature], y) <= 0.01 for feature in X):
            return True
        return False

    def construct(self, X, y):

        root = Node()

        # check if it is a leaf node
        if self.ifFinish(X, y):
            root.is_leaf = True
            root.feature = y.value_counts(sort=True).keys()[0]
            return root

        # choose the feature with the highest mutual information
        best_feature = self.getBestFeature(X, y)

        # split the data into two parts
```

```
        left_X, left_y, right_X, right_y = self.splitData(X, y, best_feature)
        root.feature = best_feature
        root.left = self.construct(left_X, left_y)
        root.right = self.construct(right_X, right_y)

        self.root = root
        return root

    def visualize(self, node, graph=None):
        if graph is None:
            graph = Digraph()
            graph.node(str(id(node)), str(node.feature))

        if node.left:
            graph.node(str(id(node.left)), str(node.left.feature))
            graph.edge(str(id(node)), str(id(node.left)), label='0')
            self.visualize(node.left, graph)

        if node.right:
            graph.node(str(id(node.right)), str(node.right.feature))
            graph.edge(str(id(node)), str(id(node.right)), label='1')
            self.visualize(node.right, graph)

        return graph

tree = DecisionTree()
mutual_info = []
for feature in X.columns:
    mutual_info.append(tree.mutualInfo(X[feature], y))
print("I(x,y) for each feature:")
print(mutual_info)
```

```
I(x,y) for each feature:
[0.057274865894062166, 0.005971550897804767, 0.07479007046514474,
0.2168495048312652, 1.4773436740500578e-05, 0.009236225402885823,
0.015040080377706211, 0.052022166946516735]
```

## 3 Problem 4.3

[3]:
```
# construct decision tree
tree.construct(X, y)
```

[3]:
```
<__main__.Node at 0x1057f4d60>
```

**Stopping Criteria**

```
def ifFinish(self, X, y):
    # check if it is a leaf node
```

```
    if y.shape[0] < 44 or (self.entropy(y) <= 0.01) or all(self.mutualInfo(X[feature], y) <= 0
        return True
    return False
```

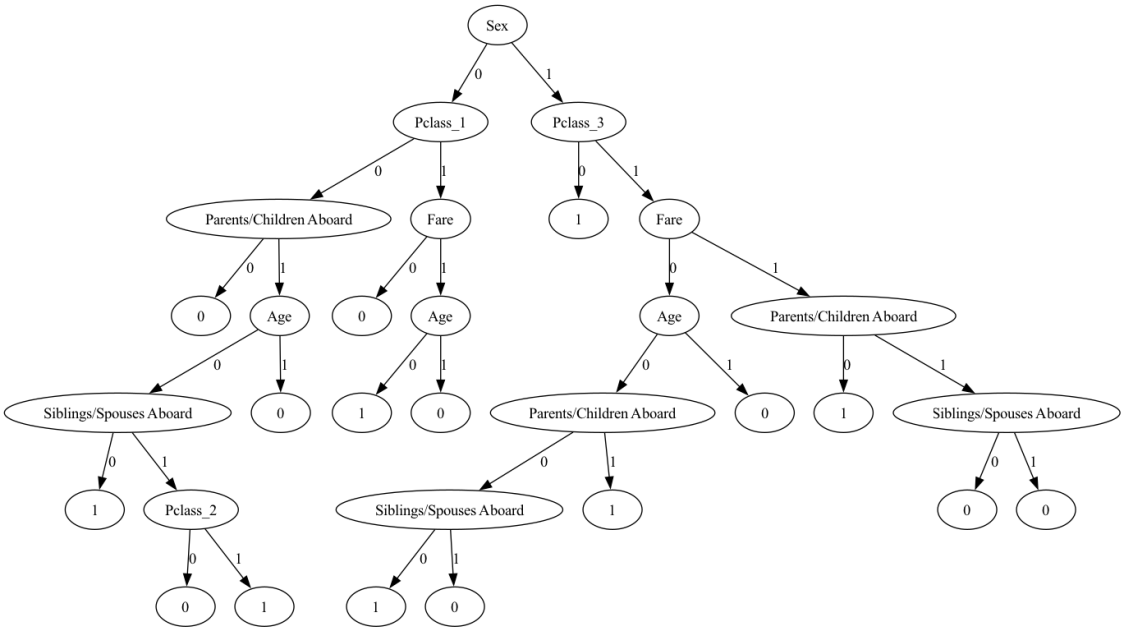
1. The current subset of data contains less than 5% of the total samples. In this problem, the number of 5% samples is 44.

2. The entropy of the response $y$ in the current subset is close to zero (less than 0.01), indicating $y$ provides little information given the existing features.
3. The mutual information of all features in the current subset $X$ and response $y$ is close to 0 (less than 0.01), also indicating $y$ provides little information given the existing features.

# 4 Problem 4.4

[4]:
```
# display decision tree
graph = tree.visualize(tree.root)
graph.render('decision_tree', format="png", view=False)
```

[4]: 'decision_tree.png'




# 5 Problem 4.5

[5]:
```
# split data into k folds
def splitKFold(X, y, k):
    fold_size = X.shape[0] // k
    kfolds = []
    for i in range(k):
        start = i * fold_size
```

```
        end = X.shape[0] if i == k-1 else (i + 1) * fold_size
        X_test = X[start:end]
        y_test = y[start:end]
        X_train = X.drop(X.index[start:end])
        y_train = y.drop(y.index[start:end])
        kfolds.append((X_train, y_train, X_test, y_test))

    return kfolds

def predict(self, node, x):
    # check if it is a leaf node
    if node.is_leaf:
        return node.feature

    feature = node.feature
    node = node.left if x[feature] == 0 else node.right
    return self.predict(node, x)

def crossValidate(self, X, y, k=10):
    # split data into k folds
    kfolds = splitKFold(X, y, k)

    # construct k decision tree
    print("\n10-Fold Cross Validation for decision tree:")
    cross_valid_accuracy = 0
    for i in range(k):
        X_train, y_train, X_test, y_test = kfolds[i]
        tree = DecisionTree()
        tree.construct(X_train, y_train)
        # predict
        y_pred = X_test.apply(lambda x: tree.predict(tree.root, x), axis=1)
        # calculate accuracy
        accuracy = np.sum(y_pred == y_test) / len(y_test)
        print("Fold ", i, " Accuracy :", accuracy)
        # accumulate accuracy
        cross_valid_accuracy += accuracy

    avg_accuracy = cross_valid_accuracy / k

    return avg_accuracy

DecisionTree.predict = predict
DecisionTree.crossValidate = crossValidate
# cross validation for decision tree
accuracy = tree.crossValidate(X, y, k=10)
print("Cross validation accuracy:", accuracy)
```

```
10-Fold Cross Validation for decision tree:
Fold  0  Accuracy : 0.7840909090909091
Fold  1  Accuracy : 0.8068181818181818
Fold  2  Accuracy : 0.7613636363636364
Fold  3  Accuracy : 0.7954545454545454
Fold  4  Accuracy : 0.8295454545454546
Fold  5  Accuracy : 0.7727272727272727
Fold  6  Accuracy : 0.8522727272727273
Fold  7  Accuracy : 0.7954545454545454
Fold  8  Accuracy : 0.8522727272727273
Fold  9  Accuracy : 0.8210526315789474
Cross validation accuracy: 0.8071052631578949
```

## 6 Problem 4.6

[6]:
```
# predict my feature vector
x = pd.Series({'Pclass_1': 0, 'Pclass_2': 0, 'Pclass_3': 1, 'Sex': 0, 'Age': 0,␣
 ↪'Siblings/Spouses Aboard': 0, 'Parents/Children Aboard': 0, 'Fare': 0})
y_pred = tree.predict(tree.root, x)
prediction = "survived" if y_pred == 1 else "deceased"
print(f"Prediction with decision tree: {prediction}, y_pred: {y_pred}")
```

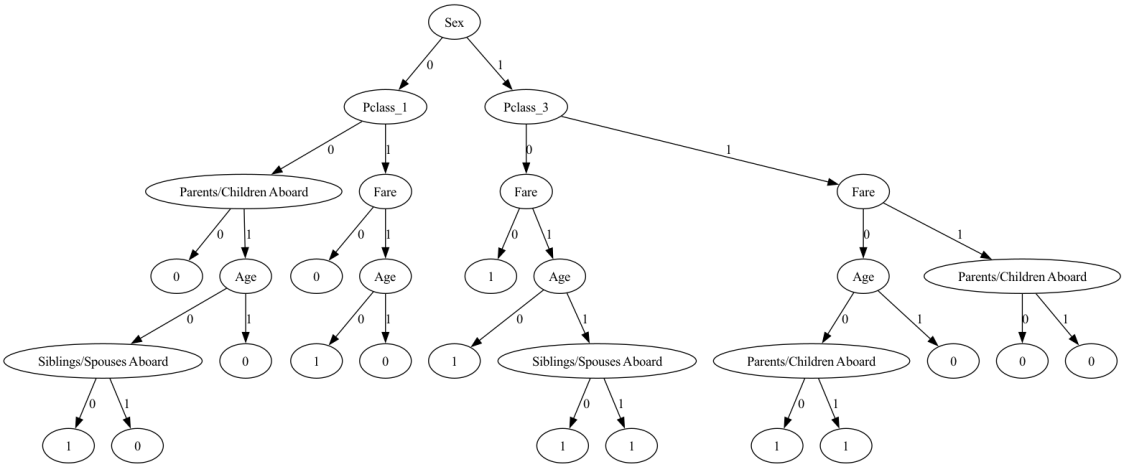
```
Prediction with decision tree: deceased, y_pred: 0
```

## 7 Problem 4.7 (a)

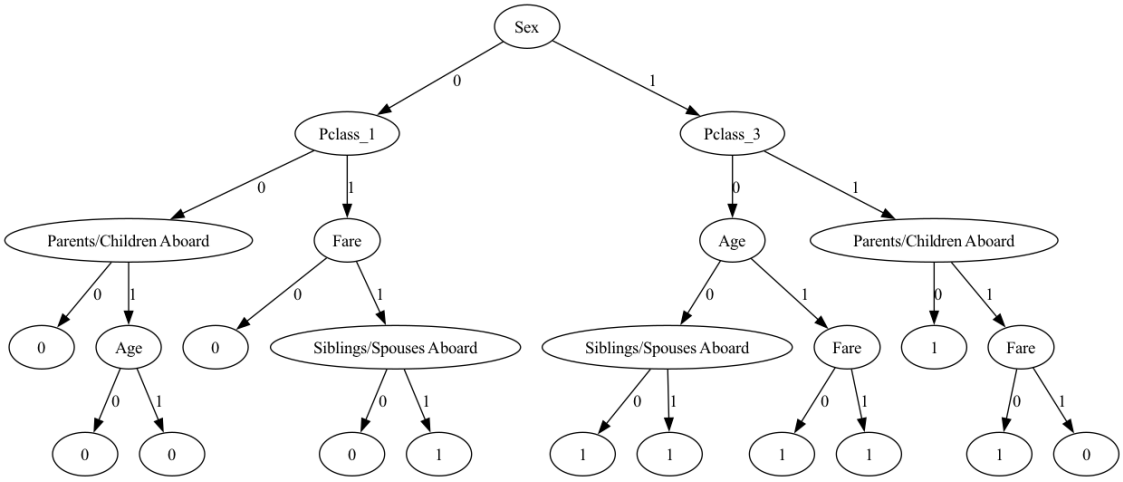
[7]:
```
# define data structure
class RandomForest():
    def __init__(self):
        self.trees = []
        self.n_trees = None

    def construct(self, X, y, n_trees=5):
        self.n_trees = n_trees
        trees = []
        for i in range(n_trees):
            # select 80% samples randomly
            samples_index = X.sample(frac=0.8, random_state = i).index
            X_train = X.loc[samples_index]
            y_train = y.loc[samples_index]
            # construct decision tree
            tree = DecisionTree()
            tree.construct(X_train, y_train)
            trees.append(tree)

        self.trees = trees
```

```
    def predict(self, x):
        y_pred = []
        for tree in self.trees:
            y_pred.append(tree.predict(tree.root, x))
        # majority vote
        random_forest_pred = 1 if sum(y_pred) > self.n_trees // 2 else 0
        return random_forest_pred

    def crossValidate(self, X, y, k=10):
        # split data into k folds
        kfolds = splitKFold(X, y, k)

        # construct k random forest
        print("\n10-Fold Cross Validation for random forest:")
        cross_valid_accuracy = 0
        for i in range(k):
            X_train, y_train, X_test, y_test = kfolds[i]
            randomForest = RandomForest()
            randomForest.construct(X_train, y_train, n_trees=5)
            # predict
            y_pred = X_test.apply(lambda x: randomForest.predict(x), axis=1)
            # calculate accuracy
            accuracy = np.sum(y_pred == y_test) / len(y_test)
            print("Fold ", i, " Accuracy :", accuracy)
            # accumulate accuracy
            cross_valid_accuracy += accuracy

        avg_accuracy = cross_valid_accuracy / k

        return avg_accuracy

    def visualize(self):
        graphs = []
        for _, tree in enumerate(self.trees):
            graph = tree.visualize(tree.root)
            graphs.append(graph)
        return graphs

# construct random forest with 5 decison trees
randomForest = RandomForest()
randomForest.construct(X, y, n_trees=5)

# display random forest
graphs = randomForest.visualize()
for i, graph in enumerate(graphs):
    graph.render('random_forest_subtree_'+str(i), format="png", view=False)
```

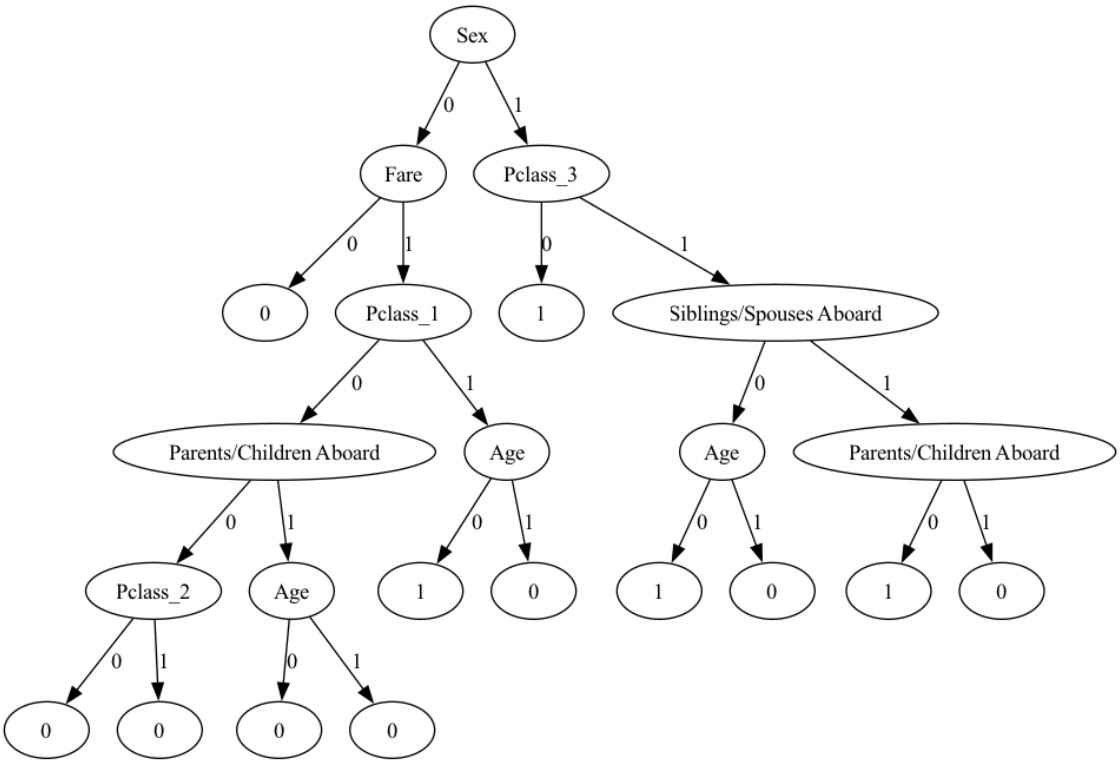
**random_forest_subtree_0**



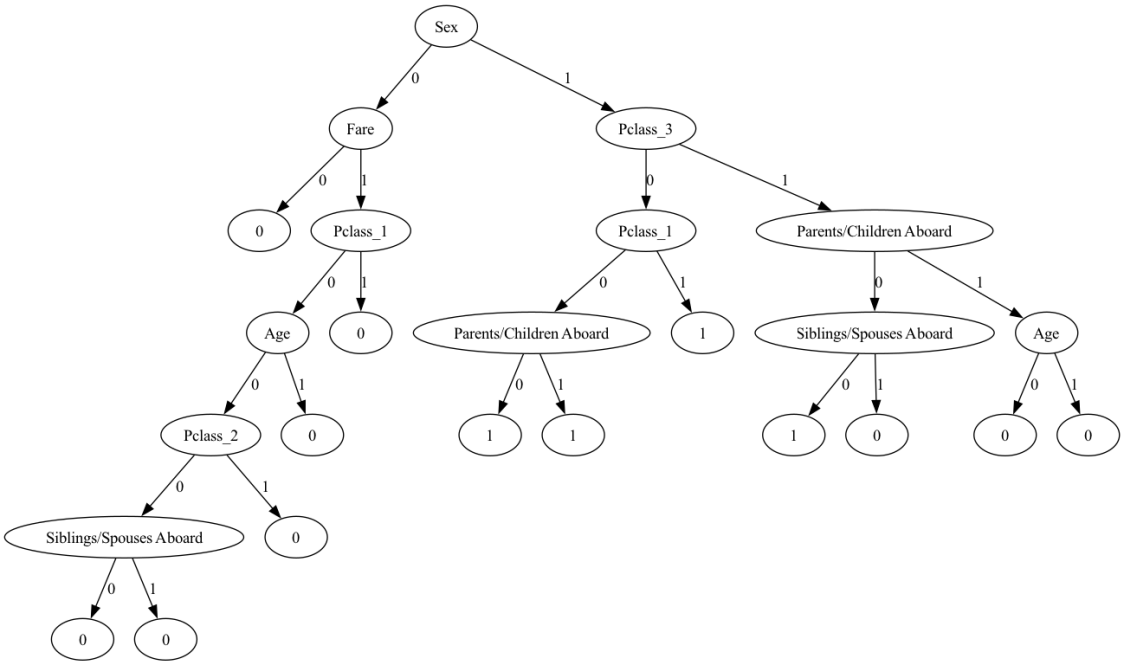

**random_forest_subtree_1**



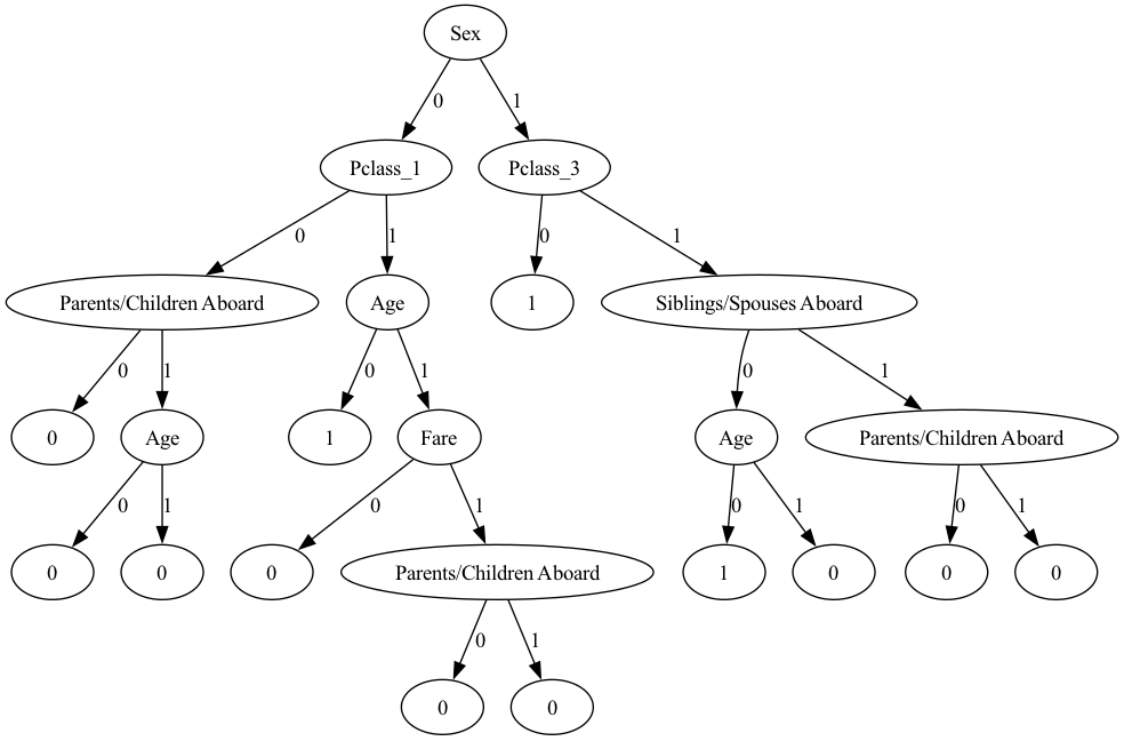

**random_forest_subtree_2**




**random_forest_subtree_3**




**random_forest_subtree_4**




# 8 Problem 4.7 (b)

[8]:
```
# cross validation for random forest
accuracy = randomForest.crossValidate(X, y, k=10)
print("Cross validation accuracy:", accuracy)
```

```
10-Fold Cross Validation for random forest:
Fold  0  Accuracy : 0.7613636363636364
Fold  1  Accuracy : 0.8409090909090909
Fold  2  Accuracy : 0.75
Fold  3  Accuracy : 0.7954545454545454
Fold  4  Accuracy : 0.8181818181818182
Fold  5  Accuracy : 0.7727272727272727
Fold  6  Accuracy : 0.8181818181818182
Fold  7  Accuracy : 0.7840909090909091
Fold  8  Accuracy : 0.8409090909090909
Fold  9  Accuracy : 0.7789473684210526
Cross validation accuracy: 0.7960765550239235
```

## 9 Problem 4.7 (c)

[9]:
```python
# predict my feature vector with random forest
y_pred = randomForest.predict(x)
prediction = "survived" if y_pred == 1 else "deceased"
print(f"Prediction with random forest: {prediction}, y_pred: {y_pred}")
```

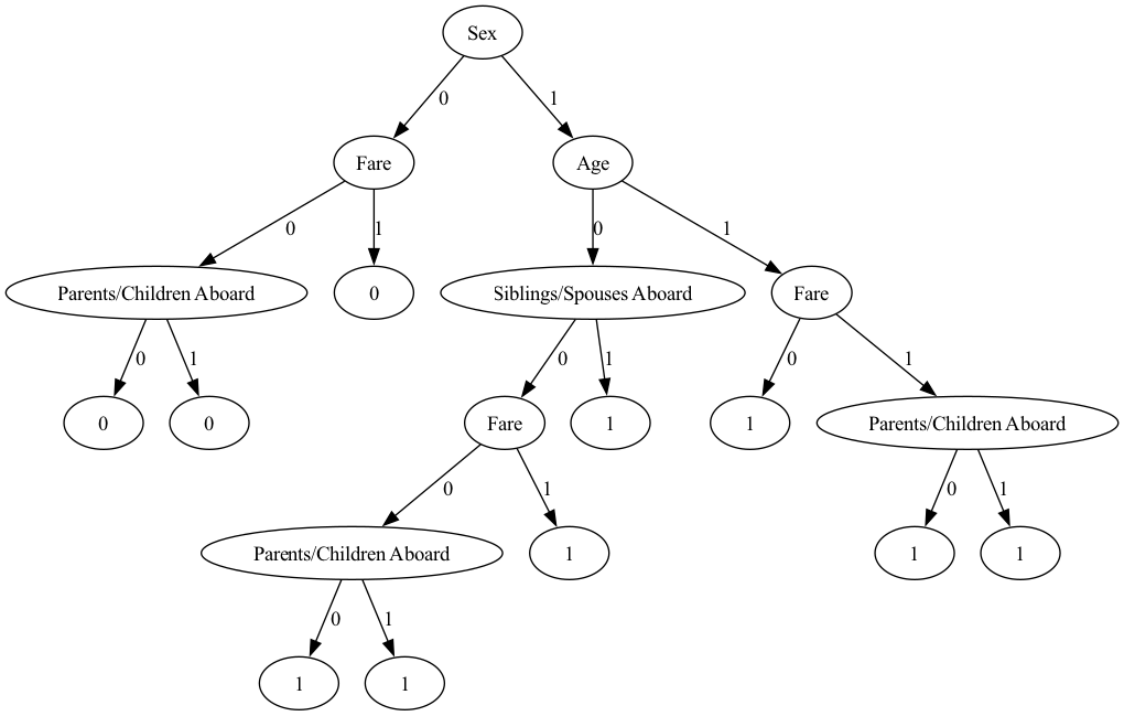
```
Prediction with random forest: deceased, y_pred: 0
```

## 10 Problem 4.8 (a)

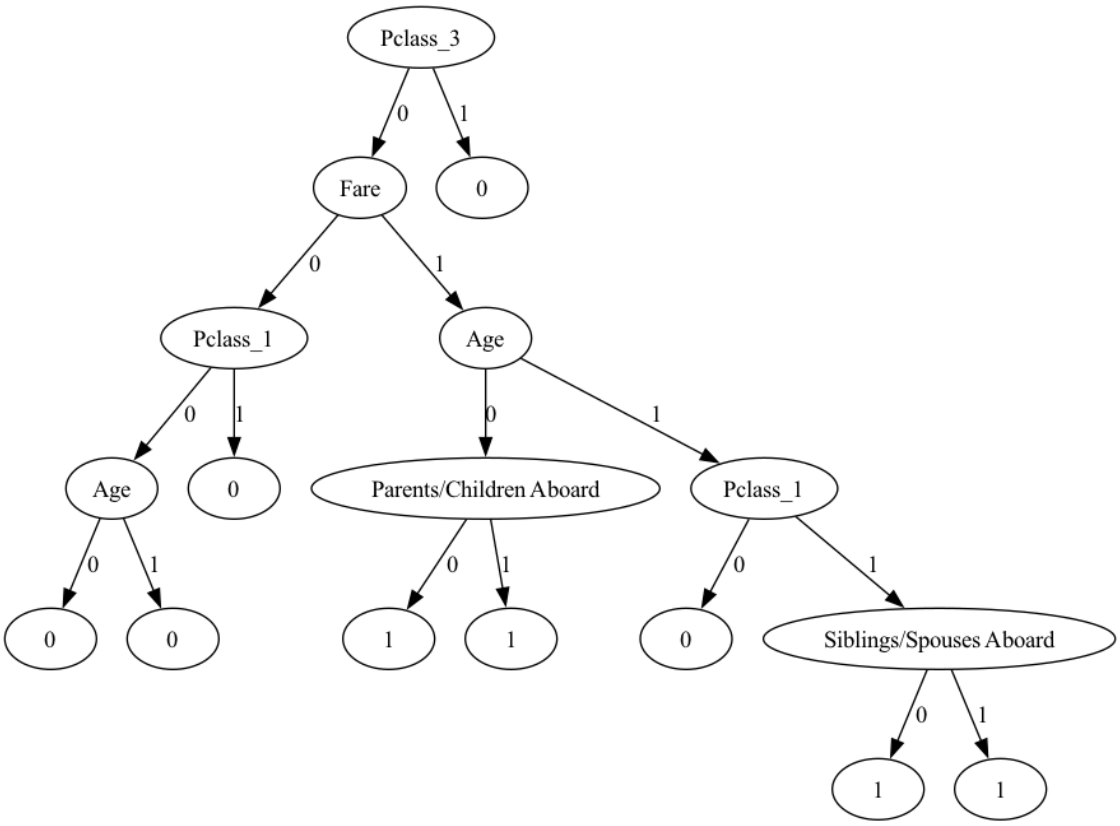
[10]:
```python
# new construct function
def constructX(self, X, y, n_trees=6):
    self.n_trees = n_trees
    trees = []
    for i in range(n_trees):
        # exclude one feature
        if i == 0:
            X_remain = X.drop(X.columns[:i+3], axis=1)
        else:
            X_remain = X.drop(X.columns[i+2], axis=1)
        # select 80% samples randomly
        samples_index = X_remain.sample(frac=0.8, random_state = i).index
        X_train = X_remain.loc[samples_index]
        y_train = y.loc[samples_index]
        # construct decision tree
        tree = DecisionTree()
        tree.construct(X_train, y_train)
        trees.append(tree)

    self.trees = trees

# new cross validate function
def crossValidateX(self, X, y, k=10):
    # split data into k folds
    kfolds = splitKFold(X, y, k)

    # construct k random forest
    print("\n10-Fold Cross Validation for random forest (excluding):")
    cross_valid_accuracy = 0
    for i in range(k):
        X_train, y_train, X_test, y_test = kfolds[i]
        randomForest = RandomForest()
        randomForest.constructX(X_train, y_train, n_trees=6)
        # predict
        y_pred = X_test.apply(lambda x: randomForest.predict(x), axis=1)
        # calculate accuracy
```

```
        accuracy = np.sum(y_pred == y_test) / len(y_test)
        print("Fold ", i, " Accuracy :", accuracy)
        # accumulate accuracy
        cross_valid_accuracy += accuracy

    avg_accuracy = cross_valid_accuracy / k

    return avg_accuracy

RandomForest.constructX = constructX
RandomForest.crossValidateX = crossValidateX
# construct random forest with 6 decison trees, each excluding one feature
randomForest = RandomForest()
randomForest.constructX(X, y, n_trees=6)

# display random forest with 6 decison trees, each excluding one feature
graphs = randomForest.visualize()
for i, graph in enumerate(graphs):
    graph.render('random_forest_subtree_excluding_feature_'+str(i),␣
 ↪format="png", view=False)
```

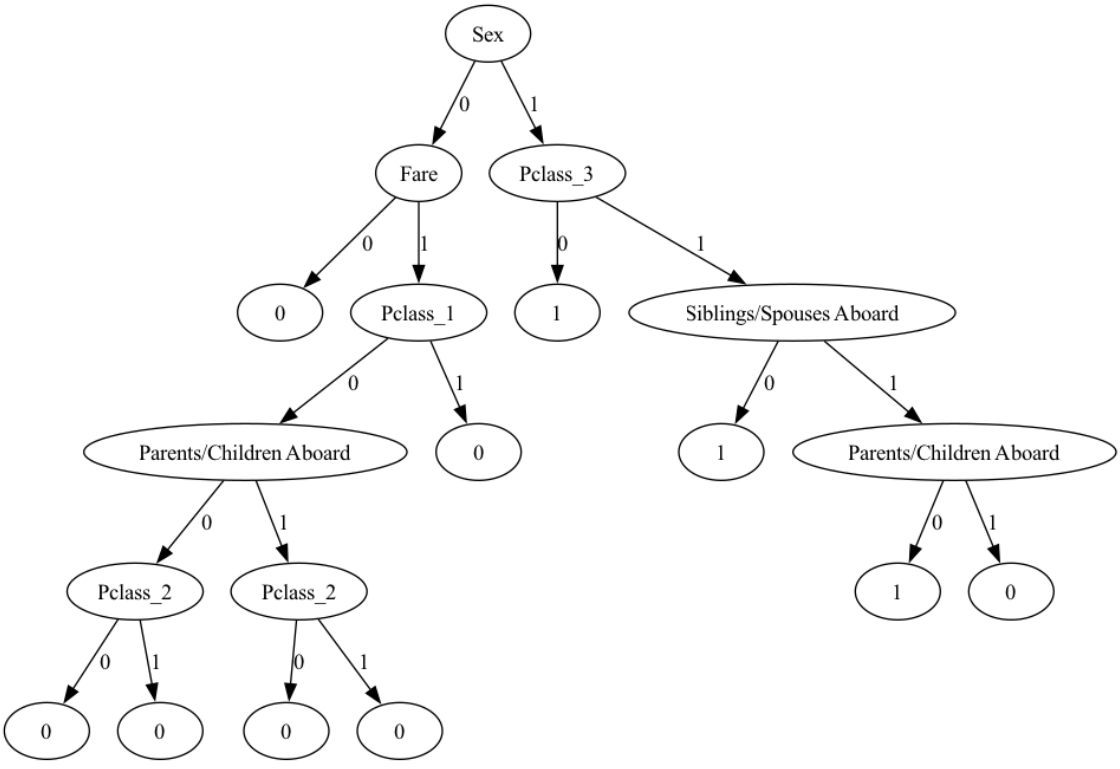
**random_forest_subtree_excluding_feature_PassengerClass**



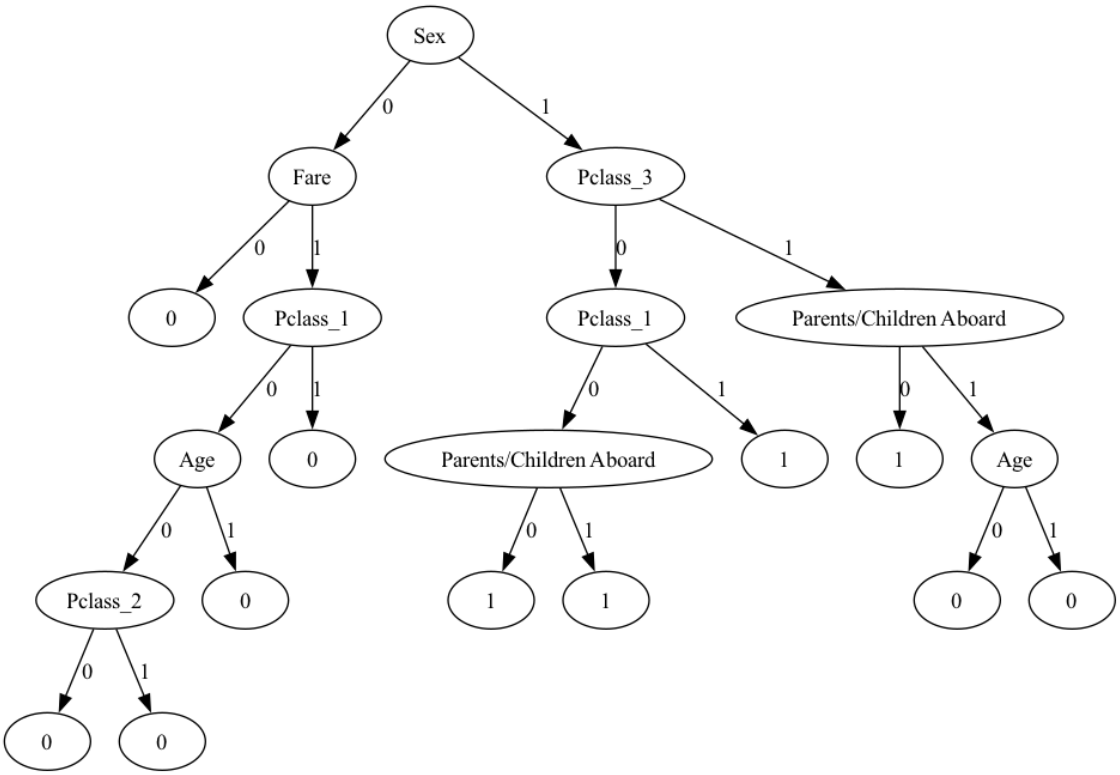

**random_forest_subtree_excluding_feature_Gender/Sex**



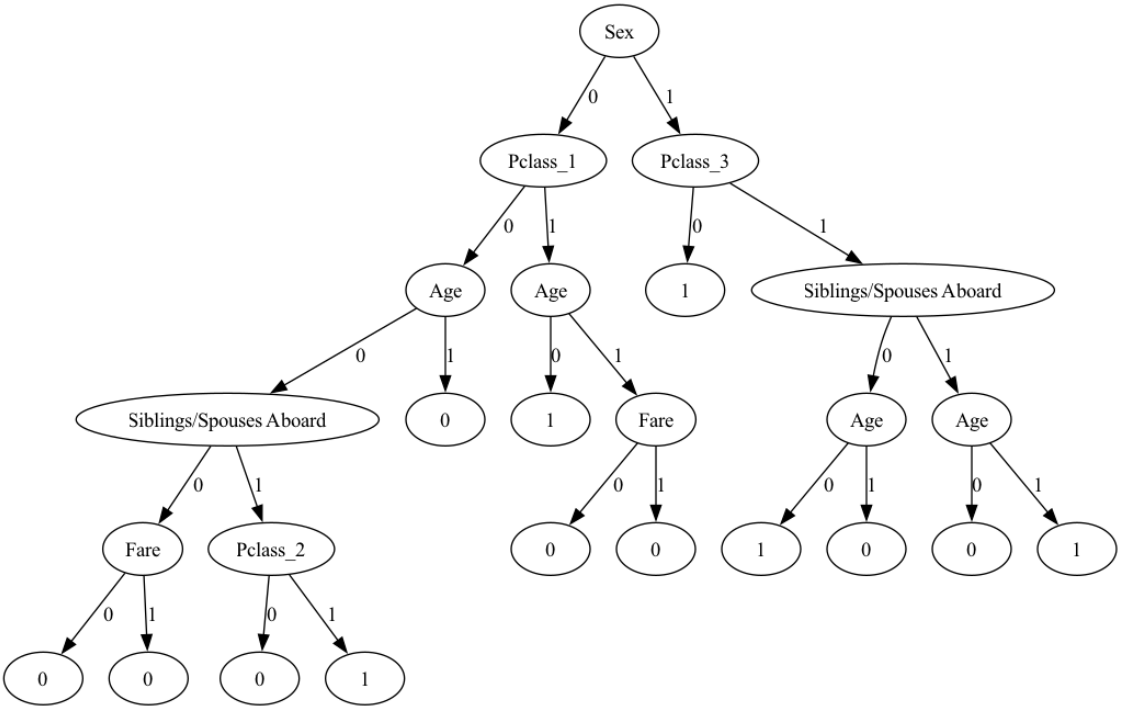

**random_forest_subtree_excluding_feature_Age**



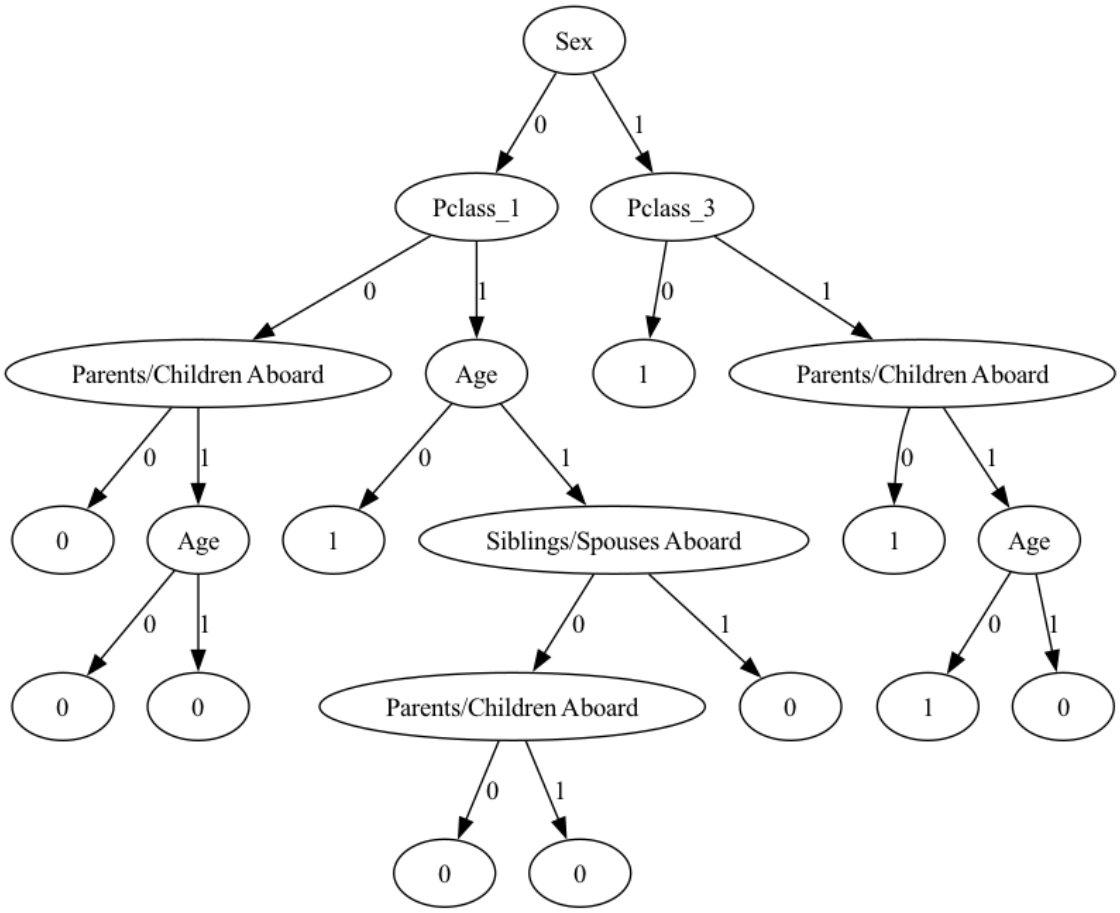

**random_forest_subtree_excluding_feature_Siblings/Spouses**




**random_forest_subtree_excluding_feature_Parents/Children**




**random_forest_subtree_excluding_feature_Fare**




## 11 Problem 4.8 (b)

[11]:
```
# cross validation for random forest with 6 decison trees, each excluding one␣
↪feature
accuracy = randomForest.crossValidateX(X, y, k=10)
print("Cross validation accuracy:", accuracy)
```

```
10-Fold Cross Validation for random forest (excluding):
Fold  0  Accuracy : 0.7727272727272727
Fold  1  Accuracy : 0.8522727272727273
Fold  2  Accuracy : 0.7386363636363636
Fold  3  Accuracy : 0.7954545454545454
Fold  4  Accuracy : 0.8295454545454546
Fold  5  Accuracy : 0.7613636363636364
Fold  6  Accuracy : 0.7954545454545454
Fold  7  Accuracy : 0.7840909090909091
Fold  8  Accuracy : 0.8409090909090909
Fold  9  Accuracy : 0.7894736842105263
Cross validation accuracy: 0.7959928229665072
```

## 12 Problem 4.8 (c)

[12]:
```
# predict my feature vector with random forest (excluding feature)
y_pred = randomForest.predict(x)
prediction = "survived" if y_pred == 1 else "deceased"
print(f"Prediction with random forest (excluding feature): {prediction}, y_pred:
  ↪ {y_pred}")
```

```
Prediction with random forest (excluding feature): deceased, y_pred: 0
```

## 13 Problem 4.9

Yes, the predictions of my logistic regression model, decision tree model, and random forest model (5 trees && 6 trees with each excluding one of the features) are consistent with each other, all predicting **deceased**.
I prefer the logistic regression model because:
1. The accuracy of my logistic regression model is around 0.82, which is higher than the cross-validation accuracy of both the decision tree (0.807) and random forest (0.796 && 0.795).
2. By calculating the confidence interval, the logistic regression model will provide a reliable probability of the predicted outcome. In this problem of predicting the survival of the Titanic, the result of the logistic regression model is reliable with a 95% probability. 3. The logistic regression model is more computationally efficient.

## 14 Problem 4.10

4.10

$$
\begin{aligned}
I(x;y) &:= H(x) - H(x|y) \\
&= E\left[\log_2\left(\frac{1}{P(x)}\right)\right] - E\left[\log_2\left(\frac{1}{P(x|y)}\right)\right] \\
&= E\left[\log_2\left(\frac{1}{P(x)}\right) - \log_2\left(\frac{1}{P(x|y)}\right)\right] \\
&= E\left[\log_2\left(\frac{P(x|y)}{P(x)}\right)\right] \\
&= E\left[\log_2\left(\frac{P(x|y)P(y)}{P(x)P(y)}\right)\right] \quad \text{according to Baye's theory} \\
&= E\left[\log_2\left(\frac{P(y|x)}{P(y)}\right)\right] \\
&= E\left[\log_2\left(\frac{1}{P(y)}\right) - \log_2\left(\frac{1}{P(y|x)}\right)\right] \\
&= E\left[\log_2\left(\frac{1}{P(y)}\right)\right] - E\left[\log_2\left(\frac{1}{P(y|x)}\right)\right] \\
&= H(y) - H(y|x) \\
&= I(y;x)
\end{aligned}
$$

## 15 Appendix

[13]:
```
import numpy as np
import pandas as pd
from graphviz import Digraph
import matplotlib.pyplot as plt
```

```
def displayFeatureStatastics(X):
    _, ax = plt.subplots(2, 3, figsize=(15, 8))

    features = X.columns.tolist()
    for i, feature in enumerate(features):
        ax[i//3, i%3].hist(X[feature], bins=len(X[feature].value_counts()))
        ax[i//3, i%3].set_title(feature)

    plt.tight_layout()
    plt.show()

    for feature in features:
        print(X[feature].value_counts())

def loadDataSet():
    # read data from file
    df = pd.read_csv('titanic_data.csv')
    # split features and label
    X = df.drop(['Survived'], axis=1)
    y = df['Survived']

    # display data statistics
    displayFeatureStatastics(X)

    # transform each of your features into a binary variable
    # passenger class (one-hot)
    one_hot_column = pd.get_dummies(X['Pclass'], prefix='Pclass').
 ↪astype('int64')
    X = one_hot_column.join(X)
    X = X.drop(['Pclass'], axis=1)
    # age
    median_age = X['Age'].median()
    X['Age'] = X['Age'].apply(lambda x: 0 if x < median_age else 1)
    # siblings/spouses
    X['Siblings/Spouses Aboard'] = X['Siblings/Spouses Aboard'].apply(lambda x:␣
 ↪0 if x == 0 else 1)
    # parents/children
    X['Parents/Children Aboard'] = X['Parents/Children Aboard'].apply(lambda x:␣
 ↪0 if x == 0 else 1)
    # fare
    median_fare = X['Fare'].median()
    X['Fare'] = X['Fare'].apply(lambda x: 0 if x < median_fare else 1)

    return X, y

# split data into k folds
```

```
def splitKFold(X, y, k):
    fold_size = X.shape[0] // k
    kfolds = []
    for i in range(k):
        start = i * fold_size
        end = X.shape[0] if i == k-1 else (i + 1) * fold_size
        X_test = X[start:end]
        y_test = y[start:end]
        X_train = X.drop(X.index[start:end])
        y_train = y.drop(y.index[start:end])
        kfolds.append((X_train, y_train, X_test, y_test))

    return kfolds

class Node:
    def __init__(self):
        # split feature
        self.feature = None
        # children
        self.left = None # 0
        self.right = None # 1
        # leaf node
        self.is_leaf = False

class DecisionTree:
    def __init__(self):
        self.root = None

    def entropy(self, x):
        n = x.shape[0]
        entropy = 0
        for k in x.value_counts().keys():
            p_k = x.value_counts()[k] / n
            entropy += p_k * np.log2(1 / p_k)
        return entropy

    # calculate mutual information between x and y
    def mutualInfo(self, x, y):
        # calculate entropy of x
        entropy_x = self.entropy(x)

        # calculate entropy of x|y
        n = x.shape[0]
        entropy_x_y = 0
        for k in y.value_counts().keys():
            num_y_k = y.value_counts()[k]
            p_y_k = num_y_k / n
```

```
            x_y_k = x[y == k]
            entropy_x_y_k = 0
            for j in x_y_k.value_counts().keys():
                num_x_j_y_k = x_y_k.value_counts()[j]
                p_x_j_y_k = num_x_j_y_k / num_y_k
                entropy_x_y_k += p_x_j_y_k * np.log2(1 / p_x_j_y_k)
            entropy_x_y += p_y_k * entropy_x_y_k

        # calculate mutual information
        mutual_info = entropy_x - entropy_x_y
        return mutual_info

    # choose the feature with the highest mutual information
    def getBestFeature(self, X, y):
        max_mutual_info = 0
        best_feature = None
        # look each feature
        for feature in X:
            mutual_info = self.mutualInfo(X[feature], y)
            if mutual_info > max_mutual_info:
                max_mutual_info = mutual_info
                best_feature = feature

        return best_feature

    def splitData(self, X, y, best_feature):
        left_index = (X[best_feature] == 0)
        right_index = (X[best_feature] == 1)
        X = X.drop([best_feature], axis=1)
        # left subset
        left_X = X[left_index]
        left_y = y[left_index]
        # right subset
        right_X = X[right_index]
        right_y = y[right_index]
        return left_X, left_y, right_X, right_y

    def ifFinish(self, X, y):
        # check if it is a leaf node
        if y.shape[0] < 44 or (self.entropy(y) <= 0.01) or all(self.
↪mutualInfo(X[feature], y) <= 0.01 for feature in X):
            return True
        return False

    def construct(self, X, y):

        root = Node()
```

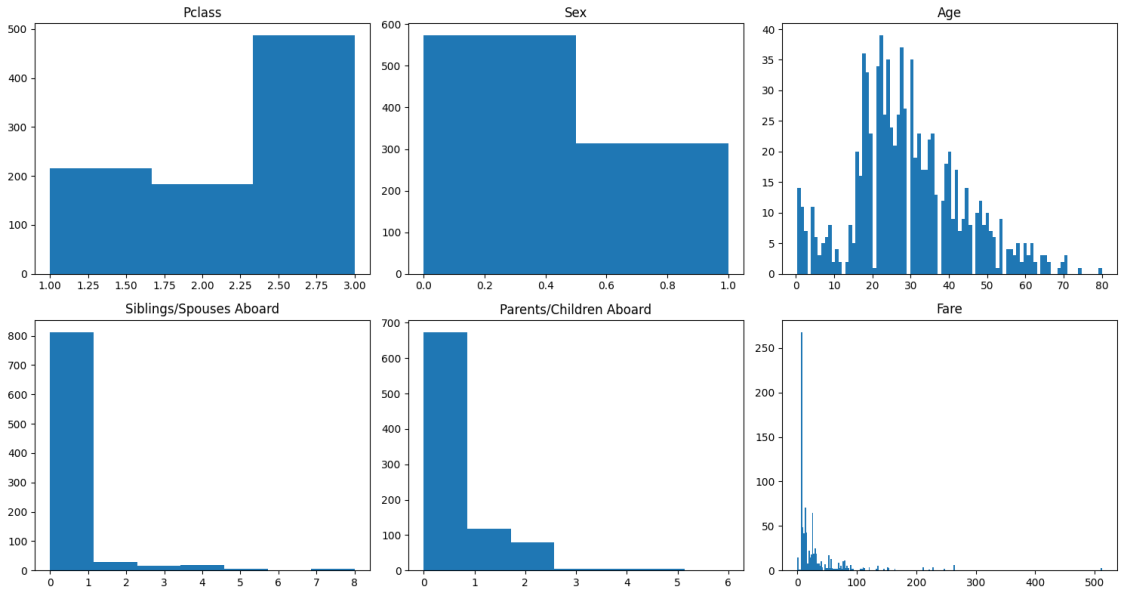
```
        # check if it is a leaf node
        if self.ifFinish(X, y):
            root.is_leaf = True
            root.feature = y.value_counts(sort=True).keys()[0]
            return root

        # choose the feature with the highest mutual information
        best_feature = self.getBestFeature(X, y)

        # split the data into two parts
        left_X, left_y, right_X, right_y = self.splitData(X, y, best_feature)
        root.feature = best_feature
        root.left = self.construct(left_X, left_y)
        root.right = self.construct(right_X, right_y)

        self.root = root
        return root

    def predict(self, node, x):
        # check if it is a leaf node
        if node.is_leaf:
            return node.feature

        feature = node.feature
        node = node.left if x[feature] == 0 else node.right
        return self.predict(node, x)

    def visualize(self, node, graph=None):
        if graph is None:
            graph = Digraph()
            graph.node(str(id(node)), str(node.feature))

        if node.left:
            graph.node(str(id(node.left)), str(node.left.feature))
            graph.edge(str(id(node)), str(id(node.left)), label='0')
            self.visualize(node.left, graph)

        if node.right:
            graph.node(str(id(node.right)), str(node.right.feature))
            graph.edge(str(id(node)), str(id(node.right)), label='1')
            self.visualize(node.right, graph)

        return graph

    def crossValidate(self, X, y, k=10):
        # split data into k folds
```

```
        kfolds = splitKFold(X, y, k)

        # construct k decision tree
        print("\n10-Fold Cross Validation for decision tree:")
        cross_valid_accuracy = 0
        for i in range(k):
            X_train, y_train, X_test, y_test = kfolds[i]
            tree = DecisionTree()
            tree.construct(X_train, y_train)
            # predict
            y_pred = X_test.apply(lambda x: tree.predict(tree.root, x), axis=1)
            # calculate accuracy
            accuracy = np.sum(y_pred == y_test) / len(y_test)
            print("Fold ", i, " Accuracy :", accuracy)
            # accumulate accuracy
            cross_valid_accuracy += accuracy

        avg_accuracy = cross_valid_accuracy / k

        return avg_accuracy

class RandomForest():
    def __init__(self):
        self.trees = []
        self.n_trees = None

    def construct(self, X, y, n_trees=5):
        self.n_trees = n_trees
        trees = []
        for i in range(n_trees):
            # select 80% samples randomly
            samples_index = X.sample(frac=0.8, random_state = i).index
            X_train = X.loc[samples_index]
            y_train = y.loc[samples_index]
            # construct decision tree
            tree = DecisionTree()
            tree.construct(X_train, y_train)
            trees.append(tree)

        self.trees = trees

    def constructX(self, X, y, n_trees=6):
        self.n_trees = n_trees
        trees = []
        for i in range(n_trees):
            # exclude one feature
            if i == 0:
```

```
            X_remain = X.drop(X.columns[:i+3], axis=1)
        else:
            X_remain = X.drop(X.columns[i+2], axis=1)
        # select 80% samples randomly
        samples_index = X_remain.sample(frac=0.8, random_state = i).index
        X_train = X_remain.loc[samples_index]
        y_train = y.loc[samples_index]
        # construct decision tree
        tree = DecisionTree()
        tree.construct(X_train, y_train)
        trees.append(tree)

    self.trees = trees

def predict(self, x):
    y_pred = []
    for tree in self.trees:
        y_pred.append(tree.predict(tree.root, x))
    # majority vote
    random_forest_pred = 1 if sum(y_pred) > self.n_trees // 2 else 0
    return random_forest_pred

def visualize(self):
    graphs = []
    for _, tree in enumerate(self.trees):
        graph = tree.visualize(tree.root)
        graphs.append(graph)
    return graphs

def crossValidate(self, X, y, k=10):
    # split data into k folds
    kfolds = splitKFold(X, y, k)

    # construct k random forest
    print("\n10-Fold Cross Validation for random forest:")
    cross_valid_accuracy = 0
    for i in range(k):
        X_train, y_train, X_test, y_test = kfolds[i]
        randomForest = RandomForest()
        randomForest.construct(X_train, y_train, n_trees=5)
        # predict
        y_pred = X_test.apply(lambda x: randomForest.predict(x), axis=1)
        # calculate accuracy
        accuracy = np.sum(y_pred == y_test) / len(y_test)
        print("Fold ", i, " Accuracy :", accuracy)
        # accumulate accuracy
        cross_valid_accuracy += accuracy
```

```
        avg_accuracy = cross_valid_accuracy / k

        return avg_accuracy

    def crossValidateX(self, X, y, k=10):
        # split data into k folds
        kfolds = splitKFold(X, y, k)

        # construct k random forest
        print("\n10-Fold Cross Validation for random forest:")
        cross_valid_accuracy = 0
        for i in range(k):
            X_train, y_train, X_test, y_test = kfolds[i]
            randomForest = RandomForest()
            randomForest.constructX(X_train, y_train, n_trees=6)
            # predict
            y_pred = X_test.apply(lambda x: randomForest.predict(x), axis=1)
            # calculate accuracy
            accuracy = np.sum(y_pred == y_test) / len(y_test)
            print("Fold ", i, " Accuracy :", accuracy)
            # accumulate accuracy
            cross_valid_accuracy += accuracy

        avg_accuracy = cross_valid_accuracy / k

        return avg_accuracy


if __name__ == '__main__':
    # load data
    X, y = loadDataSet()
    print("X:\n", X.head())
    print("y:\n", y.head())

    # construct decision tree
    tree = DecisionTree()
    tree.construct(X, y)

    # display decision tree
    graph = tree.visualize(tree.root)
    graph.render('decision_tree', format="png", view=False)

    # cross validation for decision tree
    accuracy = tree.crossValidate(X, y, k=10)
    print("Cross validation accuracy:", accuracy)
```

```
    # predict my feature vector
    x = pd.Series({'Pclass_1': 0, 'Pclass_2': 0, 'Pclass_3': 1, 'Sex': 0, 'Age':
↪ 0, 'Siblings/Spouses Aboard': 0, 'Parents/Children Aboard': 0, 'Fare': 0})
    y_pred = tree.predict(tree.root, x)
    prediction = "survived" if y_pred == 1 else "deceased"
    print(f"Prediction with decision tree: {prediction}, y_pred: {y_pred}")

    # construct random forest with 5 decison trees
    randomForest = RandomForest()
    randomForest.construct(X, y, n_trees=5)

    # display random forest
    graphs = randomForest.visualize()
    for i, graph in enumerate(graphs):
        graph.render('random_forest_subtree_'+str(i), format="png", view=False)

    # cross validation for random forest
    accuracy = randomForest.crossValidate(X, y, k=10)
    print("Cross validation accuracy:", accuracy)

    # predict my feature vector with random forest
    y_pred = randomForest.predict(x)
    prediction = "survived" if y_pred == 1 else "deceased"
    print(f"Prediction with random forest: {prediction}, y_pred: {y_pred}")

    # construct random forest with 6 decison trees, each excluding one feature
    randomForest = RandomForest()
    randomForest.constructX(X, y, n_trees=6)

    # display random forest with 6 decison trees, each excluding one feature
    graphs = randomForest.visualize()
    for i, graph in enumerate(graphs):
        graph.render('random_forest_subtree_excluding_feature_'+str(i),␣
↪format="png", view=False)

    # cross validation for random forest with 6 decison trees, each excluding␣
↪one feature
    accuracy = randomForest.crossValidateX(X, y, k=10)
    print("Cross validation accuracy:", accuracy)

    # predict my feature vector with random forest (excluding feature)
    y_pred = randomForest.predict(x)
    prediction = "survived" if y_pred == 1 else "deceased"
    print(f"Prediction with random forest (excluding feature): {prediction},␣
↪y_pred: {y_pred}")
```




```
Pclass
3    487
1    216
2    184
Name: count, dtype: int64
Sex
0    573
1    314
Name: count, dtype: int64
Age
22.00    39
28.00    37
18.00    36
21.00    34
24.00    34
         ..
0.92      1
23.50     1
36.50     1
55.50     1
74.00     1
Name: count, Length: 89, dtype: int64
Siblings/Spouses Aboard
0    604
1    209
2     28
4     18
3     16
```

```
8      7
5      5
Name: count, dtype: int64
Parents/Children Aboard
0    674
1    118
2     80
5      5
3      5
4      4
6      1
Name: count, dtype: int64
Fare
8.0500     43
13.0000    42
7.8958     36
7.7500     33
26.0000    31
           ..
35.0000     1
28.5000     1
6.2375      1
14.0000     1
10.5167     1
Name: count, Length: 248, dtype: int64
X:
    Pclass_1  Pclass_2  Pclass_3  Sex  Age  Siblings/Spouses Aboard  \
0         0         0         1    0    0                         1
1         1         0         0    1    1                         1
2         0         0         1    1    0                         0
3         1         0         0    1    1                         1
4         0         0         1    0    1                         0

   Parents/Children Aboard  Fare
0                        0     0
1                        0     1
2                        0     0
3                        0     1
4                        0     0
y:
 0    0
1    1
2    1
3    1
4    0
Name: Survived, dtype: int64

10-Fold Cross Validation for decision tree:
```

```
Fold  0  Accuracy : 0.7840909090909091
Fold  1  Accuracy : 0.8068181818181818
Fold  2  Accuracy : 0.7613636363636364
Fold  3  Accuracy : 0.7954545454545454
Fold  4  Accuracy : 0.8295454545454546
Fold  5  Accuracy : 0.7727272727272727
Fold  6  Accuracy : 0.8522727272727273
Fold  7  Accuracy : 0.7954545454545454
Fold  8  Accuracy : 0.8522727272727273
Fold  9  Accuracy : 0.8210526315789474
Cross validation accuracy: 0.8071052631578949
Prediction with decision tree: deceased, y_pred: 0

10-Fold Cross Validation for random forest:
Fold  0  Accuracy : 0.7613636363636364
Fold  1  Accuracy : 0.8409090909090909
Fold  2  Accuracy : 0.75
Fold  3  Accuracy : 0.7954545454545454
Fold  4  Accuracy : 0.8181818181818182
Fold  5  Accuracy : 0.7727272727272727
Fold  6  Accuracy : 0.8181818181818182
Fold  7  Accuracy : 0.7840909090909091
Fold  8  Accuracy : 0.8409090909090909
Fold  9  Accuracy : 0.7789473684210526
Cross validation accuracy: 0.7960765550239235
Prediction with random forest: deceased, y_pred: 0

10-Fold Cross Validation for random forest:
Fold  0  Accuracy : 0.7727272727272727
Fold  1  Accuracy : 0.8522727272727273
Fold  2  Accuracy : 0.7386363636363636
Fold  3  Accuracy : 0.7954545454545454
Fold  4  Accuracy : 0.8295454545454546
Fold  5  Accuracy : 0.7613636363636364
Fold  6  Accuracy : 0.7954545454545454
Fold  7  Accuracy : 0.7840909090909091
Fold  8  Accuracy : 0.8409090909090909
Fold  9  Accuracy : 0.7894736842105263
Cross validation accuracy: 0.7959928229665072
Prediction with random forest (excluding feature): deceased, y_pred: 0
```