Fall 2019 CX4641/CS7641 Homework 4

Grade: 105/110

Note:

- Did not implement part 3
- Random Forest cannot reach over accuracy 83% (-5 pts). The highest accuracy for Decision Tree I reached in part 2 is ~81.9% and I didn't notice accuracy increase applying RF. In fact, it is even hard to reach 80%. I suggest reaching accuracy to at least 84% for DT in order to get 83% for RF. After all, random forest is just many DTs with fewer features.
- If I use DTClassifier from scikit.learn, DT accuracy can reach over 85% and RF is able to reach over 83% easily. Therefore, you should try improving the DT method in part 2. There aren't necessarily bugs in the existing code but try different ways to handle special cases can increase accuracy.
- Also try using the original 'find_best_feature' function in part 1.2(3) for DT and see if accuracy increses. (I instead wrote and used 'find_best_feature_mean' so it's faster). Beware it can take HOURS. In comparison, using scikit learn or using 'find_best_feature_mean' only takes seconds.

Environment Setup

In [1]:
```
import numpy as np
from collections import Counter
from scipy import stats
from math import log2, sqrt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
```

Part 1: Utility Functions [25pts]

Here, we ask you to develop a few functions that will be the main building blocks of your decision tree and random forest algorithms.

Entropy and information gain [10pts]

First, we computes entropy and then use this entropy for information gain.

In [2]:

```
def entropy(class_y):
    """
    Input:
        - class_y: list of class labels (0's and 1's)

    TODO: Compute the entropy for a list of classes
    Example: entropy([0,0,0,1,1,1,1,1]) = 0.9544
    """

    total_count = len(class_y)

    if total_count != 0:
        a = Counter(class_y)

        zero_count = a[0]
        one_count = a[1]

        P_zero = float(zero_count) / total_count
        P_one = float(one_count) / total_count

        if P_zero == 0:
            H = -P_one * log2(P_one)
        elif P_one == 0:
            H = -P_zero * log2(P_zero)
        else:
            H = -P_zero * log2(P_zero) - P_one * log2(P_one)

        return H

    else:
        return 0
```

In [3]:
```
def information_gain(previous_y, current_y):
    """
    Inputs:
        - previous_y : the distribution of original labels (0's and 1's)
        - current_y  : the distribution of labels after splitting based
 on a particular
                    split attribute and split value

    TODO: Compute and return the information gain from partitioning the
 previous_y labels into the current_y labels.

    Reference: http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15381
-s06/www/DTs.pdf

    Example: previous_y = [0,0,0,1,1,1], current_y = [[0,0], [1,1,1,0]],
info_gain = 0.4591
    """

    if current_y[0] and current_y[1]:

        H = entropy(previous_y)
        total_count = len(previous_y)

        H_left = entropy(current_y[0])
        P_left = float(len(current_y[0])) / total_count

        H_right = entropy(current_y[1])
        P_right = float(len(current_y[1])) / total_count

        informationGain = H - (H_left * P_left + H_right * P_right)

        return informationGain

    else:
        return 0.0
```

In [4]:
```
# TEST CASE
test_class_y = [0,0,0,1,1,1,1,1]
print(entropy(test_class_y))

previous_y = [0,0,0,1,1,1]
current_y = [[0,0], [1,1,1,0]]
print(information_gain(previous_y, current_y))
```

```
0.954434002924965
0.4591479170272448
```

Build a simple desicion tree step by step [15pts]

Now we will implement three functions to build a decision tree from the scratch.

(1) partition_classes: [5pts]

One of the basic operations is to split a tree on one attribute (features) with a specific value for that attribute.

In partition_classes(), we split the data (X) and labels (y) based on the split feature and value - BINARY SPLIT.

You will have to first check if the split attribute is numerical or categorical. If the split attribute is numeric, split_val should be a numerical value. For example, your split_val should go over all the values of attributes. If the split attribute is categorical, split_val should include all the categories one by one.

You can perform the partition in the following way:

- Numeric Split Attribute:

  Split the data X into two lists(X_left and X_right) where the first list has all the rows where the split attribute is less than or equal to the split value, and the second list has all the rows where the split attribute is greater than the split value. Also create two lists(y_left and y_right) with the corresponding y labels.
- Categorical Split Attribute:

  Split the data X into two lists(X_left and X_right) where the first list has all the rows where the split attribute is equal to the split value, and the second list has all the rows where the split attribute is not equal to the split value. Also create two lists(y_left and y_right) with the corresponding y labels.

In [5]:
```
def partition_classes(X, y, split_attribute, split_val):
    """
    Inputs:
    - X               : (N,D) list containing all data attributes
    - y               : a list of labels
    - split_attribute : column index of the attribute to split on
    - split_val       : either a numerical or categorical value to divid
e the split_attribute

    TODO: Partition the data(X) and labels(y) based on the split value -
BINARY SPLIT.

    Example:

    X = [[3, 'aa', 10],                 y = [1,
         [1, 'bb', 22],                      1,
         [2, 'cc', 28],                      0,
         [5, 'bb', 32],                      0,
         [4, 'cc', 32]]                      1]

    Here, columns 0 and 2 represent numeric attributes, while column 1 i
s a categorical attribute.

    Consider the case where we call the function with split_attribute =
 0 (the index of attribute) and split_val = 3
    (the value of attribute).
    Then we divide X into two lists - X_left, where column 0 is <= 3 and
X_right, where column 0 is > 3.

    X_left = [[3, 'aa', 10],                y_left = [1,
              [1, 'bb', 22],                          1,
              [2, 'cc', 28]]                          0]

    X_right = [[5, 'bb', 32],               y_right = [0,
               [4, 'cc', 32]]                         1]

    Consider another case where we call the function with split_attribut
e = 1 and split_val = 'bb'
    Then we divide X into two lists, one where column 1 is 'bb', and the
other where it is not 'bb'.

    X_left = [[1, 'bb', 22],                y_left = [1,
              [5, 'bb', 32]]                          0]

    X_right = [[3, 'aa', 10],               y_right = [1,
               [2, 'cc', 28],                         0,
               [4, 'cc', 32]]                         1]

    Return in this order: (X_left, X_right, y_left, y_right)
    """

    X_left = []
    X_right = []
    y_left = []
    y_right = []
```

```
    for i in range(0, len(X)):
        if isinstance(split_val, str):
            if X[i][split_attribute] == split_val:
                X_left.append(X[i])
                y_left.append(y[i])

            else:
                X_right.append(X[i])
                y_right.append(y[i])

        else:
            if X[i][split_attribute] <= split_val:
                X_left.append(X[i])
                y_left.append(y[i])

            else:
                X_right.append(X[i])
                y_right.append(y[i])

    return (X_left, X_right, y_left, y_right)
```

(2) find_best_split [5pts]

Given the data and labels, we need to find the order of splitting features, which is also the importance of the feature. For each attribute (feature), we need to calculate its optimal split value along with the corresponding information gain and then compare with all the features to find the optimal attribute to split.

First, we specify an attribute. After computing the corresponding information gain of each value at this attribute list, we can get the optimal split value, which has the maximum information gain.

In [6]:
```
def find_best_split(X, y, split_attribute):
    """Inputs:
        - X               : (N,D) list containing all data attributes
        - y               : a list array of labels
        - split_attribute : Column of X on which to split

    TODO: Compute and return the optimal split value for a given attribu
te, along with the corresponding information gain

    Note: You will need the functions information_gain and partition_cla
sses to write this function

    Example:

        X = [[3, 'aa', 10],                 y = [1,
             [1, 'bb', 22],                      1,
             [2, 'cc', 28],                      0,
             [5, 'bb', 32],                      0,
             [4, 'cc', 32]]                      1]

        split_attribute = 0

        Starting entropy: 0.971

        Calculate information gain at splits:
           split_val = 1  -->  info_gain = 0.17
           split_val = 2  -->  info_gain = 0.02
           split_val = 3  -->  info_gain = 0.02
           split_val = 4  -->  info_gain = 0.32
           split_val = 5  -->  info_gain = 0.

       best_split_val = 4; info_gain = .32;
    """

    # get unique value in the selected feature
    unique_value = np.unique([k[split_attribute] for k in X])

    max_info_gain = 0.0
    best_split_val = None
    for i in unique_value:
        (X_left, X_right, y_left, y_right) = partition_classes(X, y, spl
it_attribute, i)
        current_y = [y_left, y_right]
        informationGain = information_gain(y, current_y)

        if informationGain > max_info_gain:
            max_info_gain = informationGain
            best_split_val = i

    return (best_split_val, max_info_gain)
```

(3) find_best_feature [5pts]

Based on the above functions, we can find the most important feature that we will split first.

In [7]:
```
def find_best_feature(X, y):
    """
    Inputs:
        - X: (N,D) list containing all data attributes
        - y : a list of labels

    TODO: Compute and return the optimal attribute to split on and optim
al splitting value

    Note: If two features tie, choose one of them at random

    Example:

        X = [[3, 'aa', 10],                 y = [1,
             [1, 'bb', 22],                      1,
             [2, 'cc', 28],                      0,
             [5, 'bb', 32],                      0,
             [4, 'cc', 32]]                      1]

        split_attribute = 0

        Starting entropy: 0.971

        Calculate information gain at splits:
           feature 0:  -->  info_gain = 0.32
           feature 1:  -->  info_gain = 0.17
           feature 2:  -->  info_gain = 0.42

        best_split_feature: 2 best_split_val: 22
    """

    max_info_gain = 0.0
    best_sv = None
    best_sf = -1

    for i in range(0, len(X[0])):

        (feature_split_val, feature_info_gain) = find_best_split(X, y, i
)

        if feature_info_gain > max_info_gain:
            best_split_feature = i
            best_split_value = feature_split_val

    return (best_split_feature, best_split_value)
```

In [8]:
```
# TEST CASE
test_X = [[3, 'aa', 10],[1, 'bb', 22],[2, 'cc', 28],[5, 'bb', 32],[4, 'c
c', 32]]
test_y = [1,1,0,0,1]
print(partition_classes(test_X, test_y, 0, 3))
print(partition_classes(test_X, test_y, 1, 'bb'))

split_attribute = 0
best_split_val, info_gain = find_best_split(test_X, test_y, split_attrib
ute)
print("best_split_val:", best_split_val, "info_gain:", info_gain)

best_feature, best_split_val = find_best_feature(test_X, test_y)
print("best_split_feature:", best_feature, "best_split_val:", best_split
_val)
```

```
([[3, 'aa', 10], [1, 'bb', 22], [2, 'cc', 28]], [[5, 'bb', 32], [4, 'c
c', 32]], [1, 1, 0], [0, 1])
([[1, 'bb', 22], [5, 'bb', 32]], [[3, 'aa', 10], [2, 'cc', 28], [4, 'c
c', 32]], [1, 0], [1, 0, 1])
best_split_val: 4 info_gain: 0.3219280948873623
best_split_feature: 2 best_split_val: 22
```

Part 2: Decision Tree [20 pts]

Please read the following instructions carefully before you dive into coding

In this part, you will implement your own ID3 decision tree class and make it work on training and test set.

You may use a recursive way to construct the tree and make use of helper functions in Part1.

Please keep in mind that we use information gain to find the best feature and value to split the data for ID3 tree.

To save your training time, we have added a `max_depth` parameter to control the maximum depth of the tree. You may adjust its value to pre-pruned the tree. If set to None, it has no control of depth.

You need to have a stop condition for splitting. This can be like, all labels in the current node are the same or reaching the pre-defined max_depth.

The MyDecisionTree class should have some member variables. We highly encourage you to use a list in Python to store the tree information. For leaves nodes, this list may have just one element representing the class label. For non-leaves node, the list should at least store the feature and value to split, and references to its left and right child.

If you use different ways to represent and store the information, please include clear comments or documentations with your code. If your result if not correct, partial credits can only be awarded if we are able to understand your code

In [9]:

```
"""
IMPORTANT:

This is an additional function I wrote for finding the best feature. Th
is function calculates mean value for each
feature and finds out the feature with maximum information gain.

Returns (best_split_feature, best_split_value) like (3).

I could have added this in my class.fit below, but just for convenience
and better readability.
"""

def find_best_feature_mean(X, y):

    max_info_gain = 0.0
    best_split_value = None
    best_split_feature = -1

    for feature in range(0, len(X[0])):

        if isinstance(feature, str):
            split_value = stats.mode([k[feature] for k in X])[0][0]

        else:
            split_value = np.mean([k[feature] for k in X])

        (_, _, y_left, y_right) = partition_classes(X, y, feature, split
_value)
        current_y = [y_left, y_right]
        feature_info_gain = information_gain(y, current_y)

        if feature_info_gain > max_info_gain:
            best_split_feature = feature
            best_split_value = split_value

    return (best_split_feature, best_split_value)
```

In [10]:
```
class MyDecisionTree(object):
    def __init__(self, max_depth=None):
        """
        TODO: Initializing the tree as an empty dictionary or list, as p
referred.

        For example: self.tree = [] or self.tree = {}
        """
        self.tree = {}
        self.max_depth = max_depth

    def fit(self, X, y, depth):
        """
        TODO: Train the decision tree (self.tree) using the the sample X
and labels y.

        NOTE: You will have to make use of the utility functions to trai
n the tree.
        One possible way of implementing the tree: Each node in self.tre
e could be in the form of a dictionary:
        https://docs.python.org/2/library/stdtypes.html#mapping-types-di
ct

        For example, a non-leaf node with two children can have a 'left'
key and  a  'right' key.
        You can add more keys which might help in classification (eg. sp
lit attribute and split value)
        """

        self.tree['Labels'] = None

        if self.max_depth != None:
            # if tree reaches max_depth
            if depth >= self.max_depth:
                temp = stats.mode(y)[0][0] # using mode is better than r
andomly assign a label, same as below
                self.tree['Labels'] = temp
                return

        # all data are in the same categroy
        if len(np.unique(y)) == 1:
            self.tree['Labels'] = y[0]
            return

        # leaf feature node
        if len(X[0]) <= 1:
            temp = stats.mode(y)[0][0]
            self.tree['Labels'] = temp
            return

        bf, bsv = find_best_feature_mean(X, y)

        # if did not find best split feature or value
        if bf == -1 or bsv == None:
            temp = stats.mode(y)[0][0]
            self.tree['Labels'] = temp
```

```
            return

        x_left, x_right, y_left, y_right = partition_classes(X, y, bf, b
sv)

        # if reaches the end
        if len(x_left) == 0 or len(x_right) == 0:
            temp = stats.mode(y)[0][0]
            self.tree['Labels'] = temp
            return

        else:
            self.tree['Left'] = MyDecisionTree(self.max_depth)
            self.tree['Right'] = MyDecisionTree(self.max_depth)
            self.tree['split_feature'] = bf
            self.tree['split_value'] = bsv
            self.tree['Left_Labels'] = y_left
            self.tree['Right_Labels'] = y_right
            self.tree['Left'].fit(x_left, y_left, depth + 1)
            self.tree['Right'].fit(x_right, y_right, depth + 1)

    def predict(self, record):
        """
        TODO: classify a sample in test data set using self.tree and ret
urn the predicted label
        """

        node = self.tree
        while self.tree['Labels'] == None:

            split_val = node['split_value']
            split_feature = node['split_feature']

            if isinstance(record[split_feature], str):
                if record[split_feature] == split_val:
                    label = node['Left'].predict(record)
                else:
                    label = node['Right'].predict(record)

            else:
                if record[split_feature] <= split_val:
                    label = node['Left'].predict(record)
                else:
                    label = node['Right'].predict(record)

            return label

        label = self.tree['Labels']

        return label
```

In [21]:
```
# helper function. You don't have to modify it
def DecisionTreeEvalution(dt, X, y, verbose=True):

    # Make predictions
    # For each test sample X, use our fitting dt classifer to predict
    y_predicted = []
    for record in X:
        # add .reshape(1, -1) after "record" if you want to use dt class
ifier from scikit.learn
        y_predicted.append(dt.predict(record)) # .reshape(1, -1)


    # Comparing predicted and true labels
    results = [prediction == truth for prediction, truth in zip(y_predic
ted, y)]

    # Accuracy
    accuracy = float(results.count(True)) / float(len(results))
    if verbose:
        print("accuracy: %.4f" % accuracy)
    return accuracy
```

Now, let us use the Decision Tree to build a classifier and then to make predictions. First load training and test dataset. Please do not modify the code in the below cell

In [12]:
```
# helper function. You don't have to modify it
data_test = pd.read_csv("hw4_data_test.csv")
data_valid = pd.read_csv("hw4_data_valid.csv")
data_train = pd.read_csv("hw4_data_train.csv")

categorical = ['workclass', 'education', 'marital-status', 'occupation',
                   'relationship', 'race', 'sex', 'native-country']
numerical = ['age', 'fnlwgt', 'education-num','capital-gain', 'capital-l
oss',
                'hours-per-week']

for feature in categorical:
        le = LabelEncoder()
        data_train[feature] = le.fit_transform(data_train[feature])
        data_test[feature] = le.fit_transform(data_test[feature])

X_train = pd.concat([data_train[categorical], data_train[numerical]], ax
is=1)
y_train = data_train['high-income']
X_test = pd.concat([data_test[categorical], data_test[numerical]], axis=
1)
y_test = data_test['high-income']
X_train, y_train, X_test, y_test = np.array(X_train), np.array(y_train),
np.array(X_test), np.array(y_test)

for feature in categorical:
        le = LabelEncoder()
        data_valid[feature] = le.fit_transform(data_valid[feature])

X_valid = pd.concat([data_valid[categorical], data_valid[numerical]], ax
is=1)
y_valid = data_valid['high-income']
X_valid, y_valid = np.array(X_valid), np.array(y_valid)
```

Now, use the training data to fit your decision tree. It may take 3 - 10 minutes for fully fitting the tree. You may adjust the max_depth parameter to save some of your time(This may affect accuracy) . We will not take running time into account when grading this part. You should reach at least 80% accuracy on test set to receive full credits

In [42]:
```
# Initializing a decision tree.

max_depth = 11
dt = MyDecisionTree(max_depth)

# Building a tree
print("fitting the decision tree")

dt.fit(X_train, y_train, 0)

# Evaluating the decision tree
DecisionTreeEvalution(dt, X_test, y_test, True)
```

```
fitting the decision tree
accuracy: 0.8187
```

Out[42]:
```
0.8186539949165653
```

In [26]:
```
from sklearn.tree import DecisionTreeClassifier

for i in range(1, 20, 1):
    DTS = DecisionTreeClassifier(random_state=0, max_depth=i)
    DTS.fit(X_train, y_train)
    DecisionTreeEvalution(DTS, X_test, y_test, True)
```

```
accuracy: 0.7453
accuracy: 0.8227
accuracy: 0.8379
accuracy: 0.8468
accuracy: 0.8511
accuracy: 0.8474
accuracy: 0.8509
accuracy: 0.8503
accuracy: 0.8461
accuracy: 0.8459
accuracy: 0.8422
accuracy: 0.8404
accuracy: 0.8366
accuracy: 0.8328
accuracy: 0.8313
accuracy: 0.8257
accuracy: 0.8216
accuracy: 0.8191
accuracy: 0.8175
```

Part 3

This part is challenging so bonus for both undergrads and grads : Pruning (10 Pts)

In order to avoid overfitting, you can: 1. Acquire more training data; 2. Remove irrelevant attributes; 3. Grow full tree, then post-prune; 4. Ensemble learning.

In this bonus part, you are going to apply reduced error post-pruning to prune the fully grown tree. The idea is basically about, starting at the leaves, each node is replaced with its most popular class. If the prediction accuracy is not affected then the change is kept. You may also try recursive function to apply the post-pruning. Please notice we use validation set to get the accuracy for each node during the pruning

In [82]:
```
#  Define the post-pruning function
def pruning(dt, X, y):
    """
    TODO:
    1. Prune the full grown decision tress recursively.
    2. Classify examples in validation set.
    3. For each node:
    3.1 Sum errors over the entire subtree. You may want to use the help
er function "DecisionTreeEvalution".
    3.2 Calculate the error on same example if converted to a leaf with
 majority class label.
    You may want to use the helper function "DecisionTreeError".
    4. If error rate in the subtree is greater than in the single leaf,
 replace the whole subtree by a leaf node.
    5. Return the pruned decision tree.
    """
    #  Delete this line when you implement the function
    raise NotImplementedError

def DecisionTreeError(y):
    # helper function for calculating the error of the entire subtree if
converted to a leaf with majority class label.
    # You don't have to modify it
    num_ones = np.sum(y)
    num_zeros = len(y) - num_ones
    return 1.0 - max(num_ones, num_zeros) / float(len(y))
```

Now, you should make use of the decision tree you trained in part1. Make sure to let it have 20 or greater depths. Due the unbalance of our dataset, the post-pruning does not necessarily have better accuracy on test set. We will award full credits as long as your implementation is correct the

In [83]:
```
# helper function. You don't have to modify it.
# pruning the full grown decision tree using validation set
# dt should be a decision tree object that has been fully trained
dt_pruned=pruning(dt, X_test, y_test)

# Evaluate the decision tree using test set
DecisionTreeEvalution(dt_pruned, X_valid, y_valid, False)
```

```
---------------------------------------------------------------------
----
NotImplementedError                       Traceback (most recent call l
ast)
<ipython-input-83-f6f30320cb7d> in <module>
      2 # pruning the full grown decision tree using validation set
      3 # dt should be a decision tree object that has been fully train
ed
----> 4 dt_pruned=pruning(dt, X_test, y_test)
      5
      6 # Evaluate the decision tree using test set

<ipython-input-82-155a232d567c> in pruning(dt, X, y)
     13     """
     14     #  Delete this line when you implement the function
---> 15     raise NotImplementedError
     16
     17 def DecisionTreeError(y):

NotImplementedError:
```

Part 4: Random Forests [35pts]

The decision boundaries drawn by decision trees are very sharp, and fitting a decision tree of unbounded depth to a list of examples almost inevitably leads to **overfitting**. In an attempt to decrease the variance of our classifier we're going to use a technique called 'Bootstrap Aggregating' (often abbreviated 'bagging').

A Random Forest is a collection of decision trees, built as follows:

1) For every tree we're going to build:

```
a) Subsample the examples with replacement. Note that in this question, the
 size of the subsample data is equal to the original dataset.

b) From the subsamples in a), choose attributes at random to learn on in acc
ordance with a provided attribute subsampling rate. Based on what it was men
tioned in the class, we randomly pick features in each split. We use a more
 general approach here to make the programming part easier. Let's randomly p
ick some features (70% percent of features) and grow the tree based on the p
re-determined randomly selected features. Therefore, there is no need to fin
d random features in each split.

c) Fit a decision tree to the subsample of data we've chosen to a certain de
pth.
```

Classification for a random forest is then done by taking a majority vote of the classifications yielded by each tree in the forest after it classifies an example.

In RandomForests Class,

1. X is assumed to be a matrix with num_training rows and num_features columns where num_training is the number of total records and num_features is the number of features of each record.
2. y is assumed to be a vector of labels of length num_training.

NOTE: Lookout for TODOs for the parts that needs to be implemented.

In [43]:
```
"""
NOTE: For graduate student, you are required to use your own decision tr
ee MyDecisionTree() to finish random forest.
"""

class RandomForest(object):
    def __init__(self, n_estimators=50, max_depth=None, max_features=0.7
):
        # helper function. You don't have to modify it
        # Initialization done here
        self.n_estimators = n_estimators  # num_trees
        self.max_depth = max_depth
        self.max_features = max_features
        self.bootstraps_row_indices = []
        self.feature_indices = []
        self.out_of_bag = []
        # TODO my own classifier
        self.decision_trees = [MyDecisionTree(max_depth=max_depth) for i
in range(n_estimators)]
#         self.decision_trees = [DecisionTreeClassifier(max_depth=max_de
pth) for i in range(n_estimators)]

    def _bootstrapping(self, num_training, num_features):
        """
        TODO:
        - Randomly select a sample dataset of size num_training with rep
lacement from the original dataset.
        - Randomly select certain number of features (num_features denot
es the total number of features in X,
          max_features denotes the percentage of features that are used
 to fit each decision tree) without replacement from the total number of
features.

        Return:
        - row_idx: the row indices corresponding to the row locations of
the selected samples in the original dataset.
        - col_idx: the column indices corresponding to the column locati
ons of the selected features in the original feature list.

        Reference: https://en.wikipedia.org/wiki/Bootstrapping_(statisti
cs)
        """

        # randomly select indices of original dataset
        row_idx = np.random.choice(range(0, num_training), size=num_trai
ning, replace=True)

        # randomly select features of original dataset
        sample_size = int(num_features * self.max_features)
        col_idx = np.random.choice(range(0, num_features), size=sample_s
ize, replace=False)

        return (row_idx, col_idx)

    def bootstrapping(self, num_training, num_features):
        # helper function. You don't have to modify it
```

```
        # Initializing the bootstap datasets for each tree
        for i in range(self.n_estimators):
            total = set(list(range(num_training))) # index of 1 to num_t
raining
            row_idx, col_idx = self._bootstrapping(num_training, num_fea
tures)
            total = total - set(row_idx) # index of total minus index of
row_idx
            self.out_of_bag.append(total) # rest of training sample inde
x (total - bootstrap), array of arrays
            self.bootstraps_row_indices.append(row_idx) # bootstrap samp
le index, array of arrays
            self.feature_indices.append(col_idx) # bootstrap feature ind
ex, array of arrays


    def fit(self, X, y):
        """
        TODO:
        Train decision trees using the bootstrapped datasets.
        Note that you need to use the row indices and column indices.
        """

        num_training = len(X)
        num_features = len(X[0])

        self.bootstrapping(num_training, num_features)

        for tree_no, tree in enumerate(self.decision_trees):
            feature_no = self.feature_indices[tree_no]
            row_no = self.bootstraps_row_indices[tree_no]

            sub_dataset = np.take(X, row_no, axis=0)
            sub_dataset = np.take(sub_dataset, feature_no, axis=1)
            sub_label = np.take(y, row_no, axis=0)

            tree.fit(sub_dataset, sub_label, 0)


    def OOB_score(self, X, y):
        # helper function. You don't have to modify it
        #1. Find the set of trees that consider the record as an out-of-
bag sample.
        #2. Predict the label using each of the above found trees.
        #3. Calculate accuracy rate for the record ( num predicted corre
ctly / total num of found trees)
        # 4. Do this for all samples in X, calculate the mean accuracy r
ate.
        accuracy = []
        for i in range(len(X)):
            predictions = []
            for t in range(self.n_estimators):
                if i in self.out_of_bag[t]:
                    # TODO remove reshape(1,-1)
                    predictions.append(self.decision_trees[t].predict(X[
i][self.feature_indices[t]]))
            if len(predictions) > 0:
```

```
                accuracy.append(np.sum(predictions == y[i]) / float(len(
predictions)))
        return np.mean(accuracy)
```

In [ ]:

```
"""
TODO:
n_estimators defines how many decision trees are fitted for the random f
orest (at least 10).
max_depth defines a stop condition when the tree reaches to a certain de
pth.
max_features controls the percentage of features that are used to fit ea
ch decision tree.
Tune these three parameters to achieve a better accuracy (Required min.
 accuracy is 0.83.)
The random forest fitting may take 5 - 15 minutes. We will not take runn
ing time into account when grading this part.
"""

n_estimators = 15
max_depth = 11
max_features = 0.95

random_forest = RandomForest(n_estimators, max_depth,  max_features)

random_forest.fit(X_train, y_train)
accuracy=random_forest.OOB_score(X_train, y_train)

print("accuracy: %.4f" % accuracy)
```

Part 5: SVM (30 Pts)

5.1 Fitting an SVM classifier by hand (20 Pts)

Consider a dataset with 2 points in 1-dimensional space: $(x_1 = 0, y_1 = -1)$ and $(x_2 = \sqrt{2}, y_2 = 1)$.

Consider mapping each point to 3-dimensional space using the feature vector $\phi(x) = [1, \sqrt{2}x, x^2]$. (This is equivalent to using a second order polynomial kernel.) The max margin classifier has the form

$$min||\theta||^2 \, s.t.$$
$$y_1(\phi(x_1)\theta + b) \geq 1$$
$$y_2(\phi(x_2)\theta + b) \geq 1$$

Hint: $\phi(x_1)$ and $\phi(x_2)$ are the suppport vectors. We have already given you the solution for the suppport vectors and you need to calculate back the parameters. Margin is equal to $\frac{1}{||\theta||}$ and full margin is equal to $\frac{2}{||\theta||}$.

(1) Find a vector parallel to the optimal vector θ. (4pts)

(2) Calculate the value of the margin achieved by this θ? (4pts)

(3) Solve for θ, given that the margin is equal to $1/||\theta||$. (4pts)

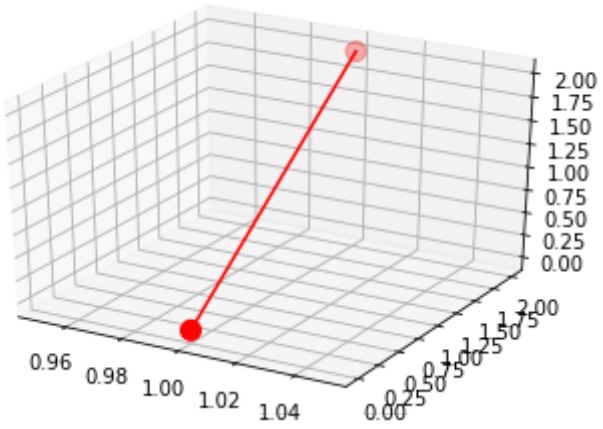
(4) Solve for b using your value for θ. (4pts)

(5) Write down the form of the discriminant function $f(x) = \phi(x)\theta + b$ as an explicit function of x. (4pts)

****Answer****

(1). Given $(x_1, y_1) = (0, -1)$, $(x_2, y_2) = (\sqrt{2}, 1)$, and $\phi(x) = [1, \sqrt{2}x, x^2]$, we can calculate: $\phi(x_1) = (1, 0, 0)$ and $\phi(x_2) = (1, 2, 2)$. From the (not very good) plot below we can see that the decision boundary lies in the middle of the line connecting two points, with θ perpendicular to the plane. Therefore, from the plot we can easily construct an optimal vector $\vec{w} = \phi(x_1) - \phi(x_2) = [0, 2, 2]$.

In [87]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

ax = plt.gca(projection="3d")
x,y,z = [1,1],[0,2],[0,2]
ax.scatter(x,y,z, c='r',s=100)
ax.plot(x,y,z, color='r')

plt.show()
```



(2). First calculate the distance between two data points: $d = \sqrt{(1-1)^2 + (2-0)^2 + (2-0)^2} = 2\sqrt{2}$. We know this distance is $\frac{2}{\theta}$ by definition. Therefore, $\frac{2}{||\theta||} = 2\sqrt{2} \Rightarrow margin = \frac{1}{||\theta||} = \sqrt{2}$.

(3). Let $\theta = [\theta_1, \theta_2, \theta_3] \Rightarrow ||\theta|| = \sqrt{\theta_1^2 + \theta_2^2 + \theta_3^2} = \frac{1}{\sqrt{2}}$ from (2). Also, since we know θ is parallel to $\vec{w}$, we know that $\theta_1 = 0$, $\theta_2 = \theta_3$, because vector $\vec{A}$ is parallel to vector $\vec{B}$ if only if $k\vec{A} = \vec{B}$). Plug in what we know, we have $2 * \theta_2^2 = \frac{1}{2}, \Rightarrow \theta_2 = \theta_3 = \frac{1}{2} or -\frac{1}{2}$. These are the same because they are just different directions. $\theta = (0, \frac{1}{2}, \frac{1}{2})$

(4). Given $y_1(\phi(x_1)\theta + b) \geq 1$, plug in $y_1 = -1$ and $\phi(x_1)$,

$$\Rightarrow (-1) * (0 * 1 + \frac{1}{2} * 0 + \frac{1}{2} * 0 + b) \geq 1,$$
$$\Rightarrow b \leq -1.$$

Doing the same for $y_2(\phi(x_2)\theta + b) \geq 1$,

$$\Rightarrow 1 * (0 * 1 + \frac{1}{2} * 2 + \frac{1}{2} * 2 + b) \geq 1,$$
$$\Rightarrow b \geq -1.$$

Combining the answers, we get $b = -1$.

(5). Given $\phi(x) = [1, \sqrt{2}x, x^2]$ and $\theta = (0, \frac{1}{2}, \frac{1}{2})$, we can write:

$$f(x) = \frac{\sqrt{2}x}{2} + \frac{x^2}{2} - 1.$$

5.2 SVM Kernel (10 Pts)

Suppose we have a dataset in 2-dimensional space which consists of 1 data point $(x_1 = 2, x_2 = 2)$ with the positive label and 4 data points $(x_1 = 1, x_2 = 1), (x_1 = 3, x_2 = 1), (x_1 = 3, x_2 = 3), (x_1 = 1, x_2 = 3)$ with the negative labels.

(1) Find a feature map, which will map the original 2-dimensional data points to the feature space so that the positive samples and the negative samples are linearly separable with each other. Draw the dataset after mapping in the feature space. (5pts)

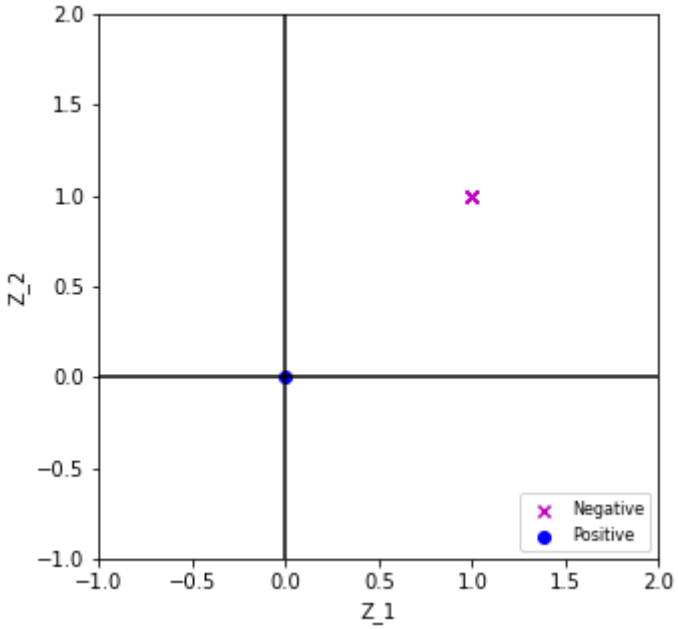
(2) In your plot above, draw the decision boundary given by hard-margin linear SVM. Mark the corresponding support vectors. (5pts)

****Answer****

(1). Let's map x-space to z-space: $z_1 = (x_1 - 2)^2, z_2 = (x_2 - 2)^2$. Hence the dataset can be transformed as follows: positive: $(x_1, x_2) = (0, 0)$, negative: All points are $(1, 1)$ after mapping. See below for the plot. (4 datapoints have the same coordinates after mapping so they got overlapped.)

In [88]:

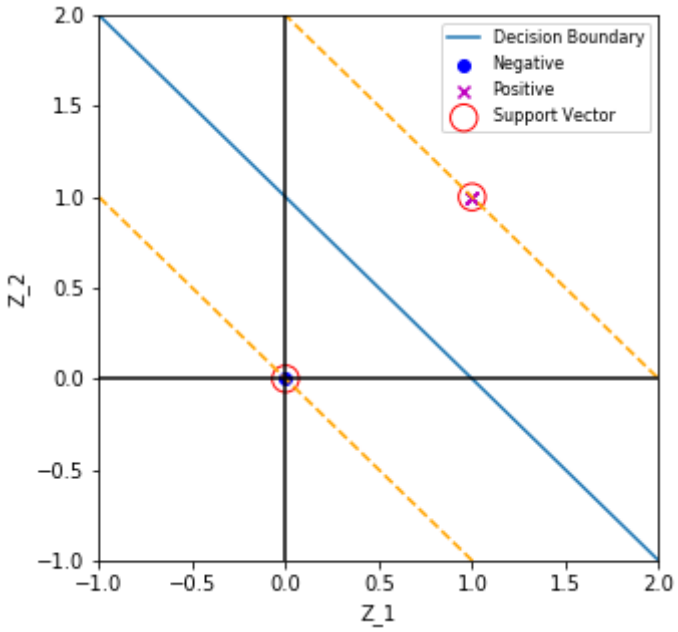
```
import matplotlib.pyplot as plt
from numpy.random import random

plt.figure(figsize=(5, 5))
neg = plt.scatter(x=0, y=0, marker='o', color='b')
pos = plt.scatter(x=[1,1,1,1], y=[1,1,1,1], marker='x', color='m')

plt.legend((pos, neg),
           ('Negative', 'Positive'),
           scatterpoints=1,
           loc='lower right',
           fontsize=8)
plt.xlabel('Z_1')
plt.ylabel('Z_2')
plt.xlim([-1,2])
plt.ylim([-1,2])
plt.axhline(0, color='black')
plt.axvline(0, color='black')

plt.show()
```



(2) See plot below

Desicion boundary lines: $y = -x$, $y = -x + 2$, as shown in orange dash lines

In [89]:
```
import numpy as np

plt.figure(figsize=(5, 5))
plt.scatter(x=0, y=0, marker='o', color='b', label='Negative')
plt.scatter(x=[1,1,1,1], y=[1,1,1,1], marker='x', color='m', label='Posi
tive')

boundary = np.linspace(-1, 2, 500)
plt.plot(boundary, -boundary + 1, label='Decision Boundary')
plt.plot(boundary, -boundary + 2, '--', color='orange')
plt.plot(boundary, -boundary, '--', color='orange')

plt.scatter(x=0, y=0, s=180, facecolors='none', edgecolors='r', label='S
upport Vector')
plt.scatter(x=[1], y=[1],s=180, facecolors='none', edgecolors='r')

plt.legend(loc='upper right', fontsize=8)
plt.xlabel('Z_1')
plt.ylabel('Z_2')
plt.xlim([-1,2])
plt.ylim([-1,2])
plt.axhline(0, color='black')
plt.axvline(0, color='black')

plt.show()
```



End