ASSIGNMENT - 1

MACHINE LEARNING (BITS F464)

SUBMITTED BY

Harshvardhan Jouhary (2020B2A31623P)

Aryaman Chauhan (2020B5A72006P)

Devansh (2020B5A72001P)

Table of Contents

1. Description

The census-income dataset contains census information for 48,842 people. It has 14 attributes for each person (age, workclass, fnlwgt, education, education-num, marital-status, occupation, relationship, race, sex, capital-gain, capital-loss, hours-per-week, and native-country) and a Boolean attribute class classifying the input of the person as belonging to one of two categories >50K, <=50K. Given the attribute values, the prediction problem here is to classify whether a person's salary is >50K or <= 50K.

Properties of Data

- Number of Instances - 48842 instances, mix of continuous and discrete (train=32561, test=16281)
    - 45222 if instances with unknown values are removed (train=30162, test=15060)
    - Number of Attributes: 6 continuous, 8 nominal attributes

- Attribute Information:

1) age: continuous
2) workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked
3) fnlwgt: continuous
4) education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st4th, 10th, Doctorate, 5th-6th, Preschool
5) education-num: continuous
6) marital-status: Marriedciv-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AFspouse
7) occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Profspecialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces
8) relationship: Wife, Own-child, Husband, Not-infamily, Other-relative, Unmarried
9) race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black
10) sex: Female, Male
11) capital-gain: continuous
12) capital-loss: continuous
13) hours-perweek: continuous
14) native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, DominicanRepublic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands
15) class: >50K, <=50K To get started with the model, run environment.yml file to initialize the conda environment with relevant libraries.

2. Pre-Processing of Training Data

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn import tree
from sklearn.tree import plot_tree
import json
import pickle
import graphviz
%matplotlib inline
```

```python
train_df = pd.read_csv("data1/train.csv", na_values="?")
test_df = pd.read_csv("data1/test.csv", na_values="?")
```

The libraries of pandas, numpy, and matplotlib.pyplot are imported. The training and testing data are loaded into the data frames train_df and test_df using the Panda library. Initially, given data contains " ?" to be considered null. Just to make sure those are null values, all the question marks(" ?") were replaced with NULL.

2.1 Handling Missing Values

```python
mode_values = train_df.median(numeric_only=True)
train_df.fillna(mode_values, inplace=True)
mode_values
```

```
age                   37.0
fnlwgt            178356.0
education-num         10.0
capital-gain           0.0
capital-loss           0.0
hours-per-week        40.0
dtype: float64
```

All the NULL (or NaN) values of the floating point type in a column are replaced with the median value of that column. Here, there are 7 attribute columns with float type. The median values of these columns are listed above.

```python
mode_values = []
for column in train_df.columns:
    if train_df[column].dtype == 'object':
        mode_value = train_df[column].mode()[0]
        mode_values.append(mode_value)
        train_df[column].fillna(mode_value, inplace=True)
mode_values
```

```
['Private',
 'HS-grad',
 'Married-civ-spouse',
 'Prof-specialty',
 'Husband',
 'White',
 'Male',
 'United-States',
 '<=50K']
```

Now, all the NULL values of the object(or String) data type are replaced with the mode of respective columns. The mode or most occurring values of these columns are listed above. The 'Inplace = True' parameter lets us modify the original objects directly without creating a separate, new object.

This concludes the handling or replacement of missing data in the training data set.

2.2 Handling Continuous Values

```
train_df.rename(columns={'class':'Decision'}, inplace=True)
test_df.rename(columns={'class':'Decision'}, inplace=True)
```

This command changes or renames the 'class' column to the 'Decision' column. This is done because Chefboost libraries approach the *target variable or attribute* only when it is the last column of the data frame and is named 'Decision.'

The Chefboost package allows us to implement the C4.5 algorithm for Decision Tree and allows us to calculate the gain ratio as well. We will use this data to find discretization points and make continuous values into discrete data, which can be fed to our machine.

```
#!pip install chefboost
from chefboost.training import Training
config = {'algorithm':'C4.5'}
```

```
def gainratiocal(threshold:float, column:object) -> float:
    idx = train_df[train_df[f"{column}"] <= threshold].index
    tmp_df = train_df.copy()
    tmp_df["{column}"] = f">{threshold}"
    tmp_df.loc[idx, f"{column}"] = f"<={threshold}"
    grat = Training.findGains(tmp_df, config)['gains'][f"{column}"]
    return grat
```

```
for i in range(1,100,10):
    a = gainratiocal(float(i),'hours-per-week')
    print(i,":",a)
```

```
1 : 0.01753148492873723
11 : 0.01775184701678863 8
21 : 0.018828987795244542
31 : 0.020648930409024 23
41 : 0.023904627769250456
51 : 0.01766218719166458 6
61 : 0.008553549138154704
71 : 0.007151079969351616
81 : 0.00477984617092674
91 : 0.0023187326063295326
```

The above data calculates the gain ratios for different values using the gainratiocal (float, object) function. From the returned 'grat' float values, we can see that data around 41 may give desired decision boundary for the 'hours-per-week' column.

```
for i in range (35, 51,2):
    a = gainratiocal(float(i),'hours-per-week')
    print(i,":",a)
```

```
35 : 0.021594639037470862
37 : 0.02197617790402921
39 : 0.022415767382654897
41 : 0.023904627769250456
43 : 0.02476910237843594
45 : 0.024482106814724198
47 : 0.024970789695430925
49 : 0.026858965904353535
```

To get a more accurate threshold value, similar calculations are done, and it is found that 49 would be a much more accurate decision boundary for 'hours-per-week' column since it has the highest gain ratio.

Similar calculations are done for all other columns, and the decision boundaries are each column obtained are as follows:

hours-per-week	:	49
education-num	:	14
age	:	27
fnlwgt	:	14200

2.3 Label Mapping

The .json code below shows the Label Mapping for each column's attributes and their values. The string type values are integers for simplicity and better understanding.

{"workclass": {
"State-gov": 6, "Self-emp-not-inc": 5, "Private": 3, "Federal-gov": 0, "Local-gov": 1, "Self-emp-inc": 4, "Without-pay": 7, "Never-worked": 2},

"class": {
"<=50K": 0, ">50K": 1},

"native-country": {
"United-States": 38, "Cuba": 4, "Jamaica": 22, "India": 18, "Mexico": 25, "South": 34, "Puerto-Rico": 32, "Honduras": 15, "England": 8, "Canada": 1, "Germany": 10, "Iran": 19, "Philippines": 29, "Italy": 21, "Poland": 30, "Columbia": 3, "Cambodia": 0, "Thailand": 36, "Ecuador": 6, "Laos": 24, "Taiwan": 35, "Haiti": 13, "Portugal": 31, "Dominican-Republic": 5, "El-Salvador": 7, "France": 9, "Guatemala": 12, "China": 2, "Japan": 23, "Yugoslavia": 40, "Peru": 28, "Outlying-US(Guam-USVI-etc)": 27, "Scotland": 33, "Trinadad&Tobago": 37, "Greece": 11, "Nicaragua": 26, "Vietnam": 39, "Hong": 16, "Ireland": 20, "Hungary": 17, "Holand-Netherlands": 14},

"sex": {
"Male": 1, "Female": 0},

"race": {
"White": 4, "Black": 2, "Asian-Pac-Islander": 1, "Amer-Indian-Eskimo": 0, "Other": 3}, "relationship": {"Not-in-family": 1, "Husband": 0, "Wife": 5, "Own-child": 3, "Unmarried": 4, "Other-relative": 2},

"occupation": {
"Adm-clerical": 0, "Exec-managerial": 3, "Handlers-cleaners": 5, "Prof-specialty": 9, "Other-service": 7, "Sales": 11, "Craft-repair": 2, "Transport-moving": 13, "Farming-fishing": 4, "Machine-op-inspct": 6, "Tech-support": 12, "Protective-serv": 10, "Armed-Forces": 1, "Priv-house-serv": 8},

"marital-status": {
"Never-married": 4, "Married-civ-spouse": 2, "Divorced": 0, "Married-spouse-absent": 3, "Separated": 5, "Married-AF-spouse": 1, "Widowed": 6},

"education": {
"Bachelors": 9, "HS-grad": 11, "11th": 1, "Masters": 12, "9th": 6, "Some-college": 15, "Assoc-acdm": 7, "Assoc-voc": 8, "7th-8th": 5, "Doctorate": 10, "Prof-school": 14, "5th-6th": 4, "10th": 0, "1st-4th": 3, "Preschool": 13, "12th": 2}}

3. Training Data

3.1 Using ChefBoost Model

For the C4.5 decision tree, we will use the Chefboost library, which provides the model we require. We choose this lightweight library because - it supports categorical features, meaning we do not need one-hot encoding. In one-hot encoding, each unique category in the categorical variable is represented as a binary feature column. A new binary column is created for each category, and the presence or absence of that category is indicated by a 1 or 0, respectively.

The decision trees trained using Chefboost are stored as if-else statements in a dedicated Python file. This way, we can easily see what decisions the tree makes to arrive at a given prediction. It also provides the models for random forests, so it is considered a better library for our particular assignment.

```
config = {'algorithm':'C4.5'}
model = chef.fit(train_df, config)
```

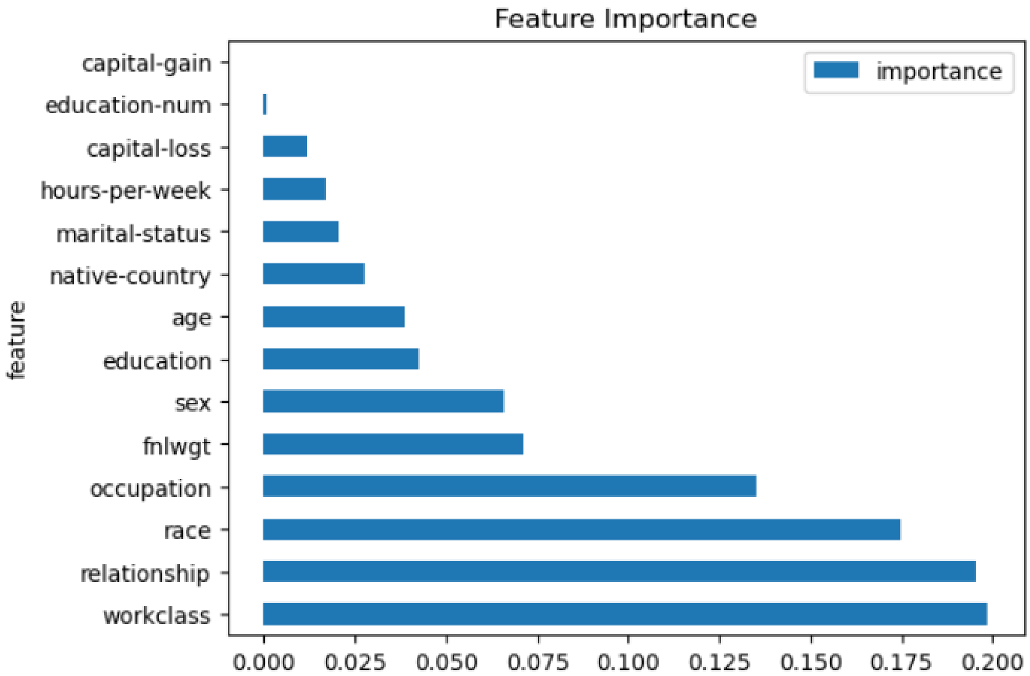
```
[INFO]:  4 CPU cores will be allocated in parallel running
C4.5  tree is going to be built…
-------------------------
```

```
finished in  2088.6931281089783  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  88.01019624704401 % on  32561  instances
Labels:  ['<=50K' '>50K']
Confusion matrix:  [[23115, 2299], [1605, 5542]]
Precision:  90.9538 %, Recall:  93.5073 %, F1:  92.2129 %
```

```
rules = "outputs/rules/rules.py"
fi = chef.feature_importance(rules).set_index("feature")
fi.plot(kind="barh", title="Feature Importance")
```

```
Decision rule:  outputs/rules/rules.py
```

```
<Axes: title={'center': 'Feature Importance'}, ylabel='feature'>
```




```
chfmdl = chef.load_model("model.pkl")
```

```
chef.evaluate(chfmdl, test_df)
```

```
-------------------------
Evaluate  test set
-------------------------
Accuracy:  82.07112585222038 % on  16281  instances
```



```
Labels:  ['<=50K' '>50K']
Confusion matrix:  [[11136, 1620], [1299, 2226]]
Precision:  87.3001 %, Recall:  89.5537 %, F1:  88.4125 %
```

Here, we have found the entire tree and the tree's dependence on different attributes. We will now begin plotting errors for training and testing data and finally obtain a tree of better accuracy. Now, due to the shortcomings of the chef-boost, we really can’t apply pruning, and hence will use the CART algorithm using Scikit Learn due to the lack of availability of the C4.5 algorithm.

3.2 Using Scikit Learn Model

The multiple columns of train_df DataFrame are converted to numerical representation using LabelEncoder() instances from sklearn.preprocessing library. The transformed values are stored in a new column with names similar to the original. The numerical representation is shown above under the Label Mapping sub-heading. The transformed data frame is stored in **X_train**. A similar transformation is done for the *target attribute* and is stored in the output data frame called **Y_train**. The target column must be removed from the input data frame X_train.
The testing dataset is also similarly transformed, and the input and output data are stored in the data frames **X_test** and **Y_test**, respectively.

3.2.1 Splitting of Testing Data

```
X_valid, X_test, Y_valid, Y_test = train_test_split(X_test, Y_test, test_size=0.
  ↪5, random_state=10)
X_valid.shape, X_test.shape
```

```
((8140, 14), (8141, 14))
```

X_test and Y_test are split equally and stored in X_valid and Y_valid correspondingly. The command ‘test_size=0.5’ specifies the proportion of the data that should be allocated for the test set. In this case, 0.5 (or 50%) of the data will be used for the test set. The command ‘random_state=10’ sets the random seed for reproducibility. It ensures that the same random splitting is applied every time the code is executed, resulting in consistent splits.

3.2.2 Training and Scoring

```
[67]: cartree = tree.DecisionTreeClassifier(criterion='entropy')
      cartree.fit(X_train, Y_train)
```

```
[67]: DecisionTreeClassifier(criterion='entropy')
```

```
[68]: cartree.tree_.node_count
```

```
[68]: 9431
```

```
[69]: cartree.get_depth()
```

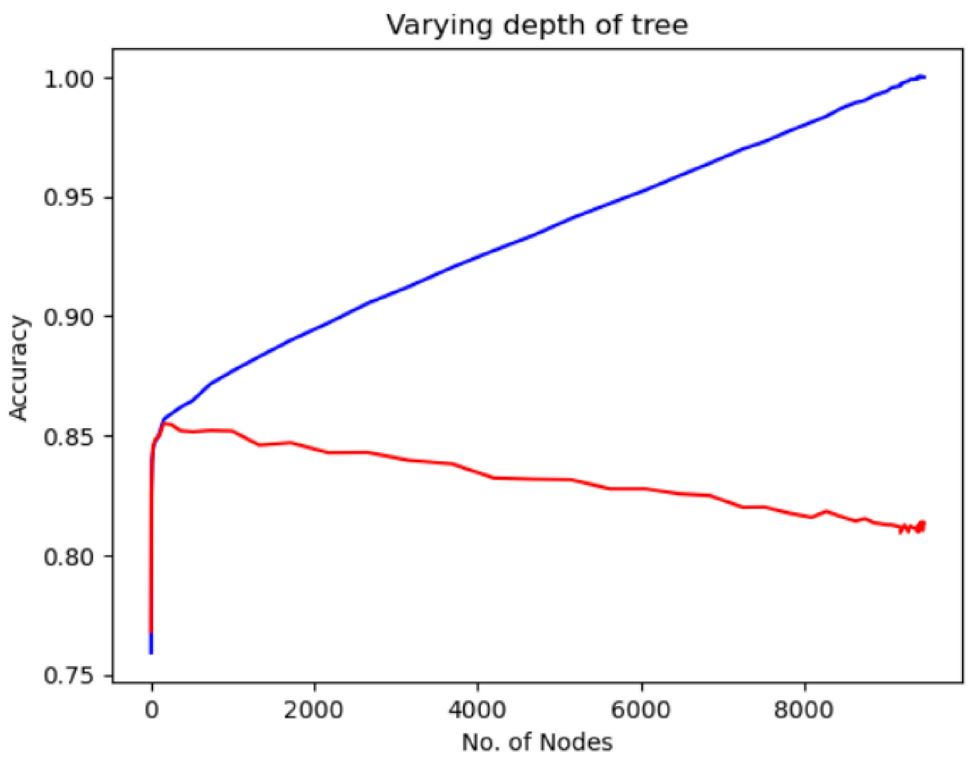
```
[69]: 49
```

The above code snippet shows the node count and depth of the Decision Tree using only the training data. This would be highly overfitted and have high accuracy for training data only. We must restrict the number of nodes and depth and obtain a graph against accuracy for the training and validation data set.

```
[73]: trainscore = []
      testscore = []
      nofnodes = []
      depthcou = []
      for i in range(1, 55):
          clf = tree.DecisionTreeClassifier(criterion = 'entropy', max_depth=i)
          clf.fit(X_train, Y_train)
          trainscore.append(clf.score(X_train, Y_train))
          testscore.append(clf.score(X_valid, Y_valid))
          nofnodes.append(clf.tree_.node_count)
          depthcou.append(i)
```

```
[74]: plt.plot(nofnodes,trainscore,'b')
      plt.plot(nofnodes,testscore, 'r')
      plt.xlabel("No. of Nodes")
      plt.ylabel("Accuracy")
      plt.title("Varying depth of tree")
```

```
[74]: Text(0.5, 1.0, 'Varying depth of tree')
```




```
[75]: maxacc = max(testscore)
      inx = testscore.index(maxacc)
      maxnodes = nofnodes[inx]
      maxnodes, depthcou[inx]
```

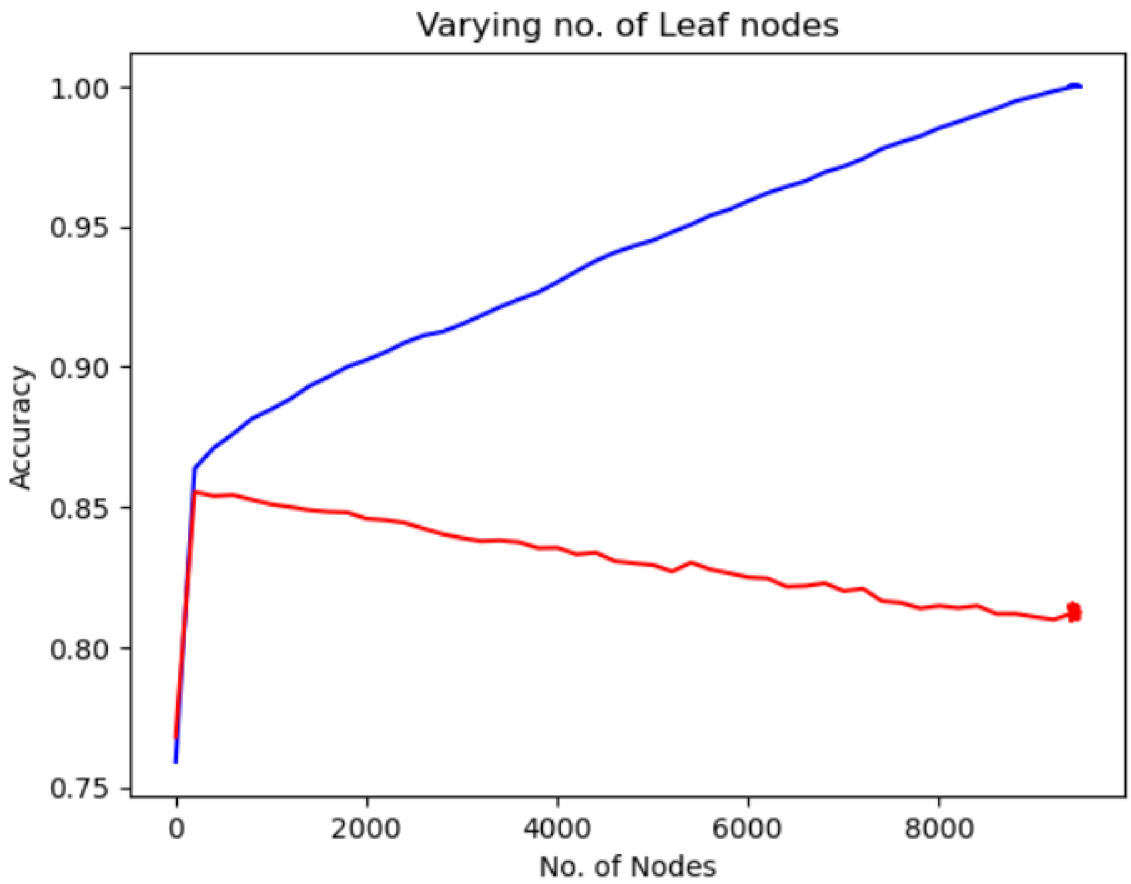
```
[75]: (157, 7)
```

The decision tree with varying depth i is generated, and the accuracy score is calculated for the training and validation set using the score( ) function. These accuracy scores are stored in trainscore and testscore lists, respectively. It is then plotted against the number of nodes, and the graph is obtained.

The list testscore has its maximum at 157 nodes and a depth of 7.

Similarly, the graph is obtained by varying the maximum number of leaf nodes i against accuracy.

```
trainscore1 = []
testscore1 = []
nofnodes1 = []
noofleaf = []
for i in range(2, 9503, 100):
    clf = tree.DecisionTreeClassifier(criterion = 'entropy', max_leaf_nodes=i)
    clf.fit(X_train, Y_train)
    trainscore1.append(clf.score(X_train, Y_train))
    testscore1.append(clf.score(X_valid, Y_valid))
    nofnodes1.append(clf.tree_.node_count)
    noofleaf.append(i)
```




```
[22]: maxacc = max(testscore1)
      inx = testscore1.index(maxacc)
      maxnodes = nofnodes1[inx]
      maxleaf = noofleaf[inx]
      maxnodes,maxleaf
```

```
[22]: (203, 102)
```

The list testscore1 has its maximum at 203 nodes and 102 leaf nodes. This shows that the validation data has its maximum accuracy score at these nodes.

```
[14]: cartree = tree.DecisionTreeClassifier(criterion='entropy', max_depth=8)
      cartree.fit(X_train, Y_train)
```

```
cartree.score(X_valid, Y_valid)
```

```
]: 0.8547911547911548
```

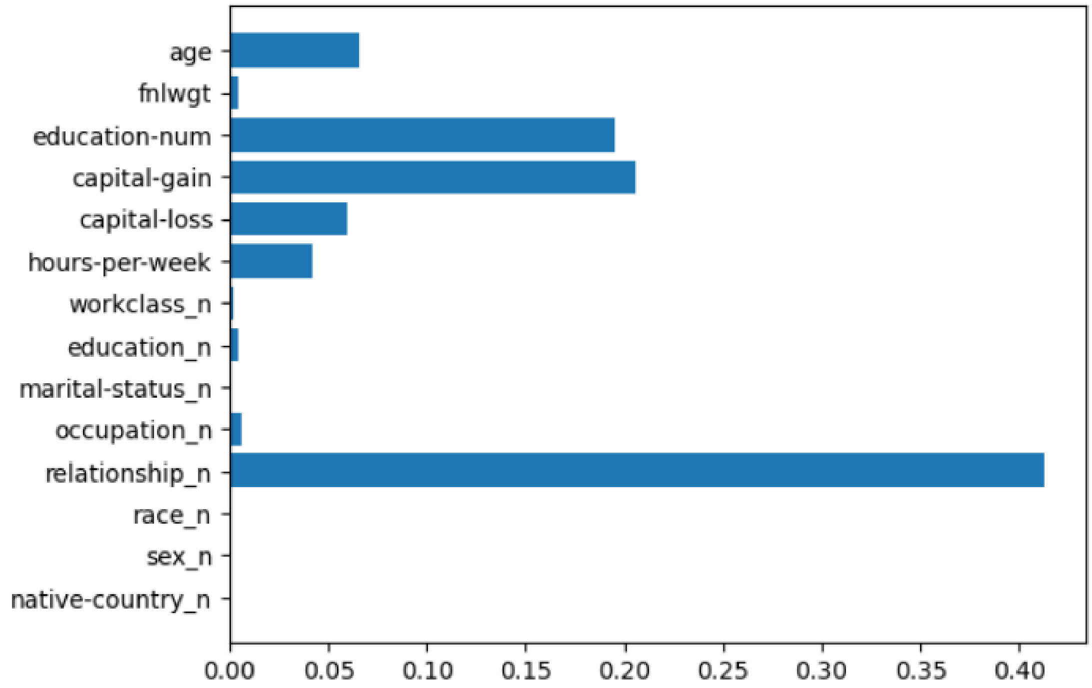
```
]: cartree.score(X_test, Y_test)
```

```
]: 0.8577570323056135
```

The accuracy score achieved by the validation and testing dataset is similar, and the percentage is **85.8%.**
Here, we are stopping the decision tree from overfitting the training data by stopping the number of nodes when the accuracy of the validation set starts to decrease. This is done by specifying the maximum depth of the decision tree.
This technique is called Pre-Pruning, where we limit the growth of the decision tree before it becomes overly complex or overfits the training data.

```
: with open("DecisionTreeFinal.pkl",'wb') as f:
      pickle.dump(cartree,f)
```

```
fi = cartree.feature_importances_
fig, ax = plt.subplots()
ax.barh(X_train.columns,fi)
ax.invert_yaxis()
```



In binary mode, a file is then opened to store the serialized representation of the trained decision tree classifier object. The 'pickle.dump( )' function is used from the pickle module of Python to serialize and write the trained decision tree classifier object 'cartree' into the opened file.

The 'feature_importances_' attribute provides a measure of importance of each feature in the decision tree based on the information gain during the splitting process. A horizontal bar chart plot is created, where the axes object 'ax' contains the columns of the training data and its feature importances.

The above-opened file is read, deserialized, and loaded in a variable, and the 'graphviz' Python package is used to generate the representation of the decision tree stored in this variable.

https://github.com/Aryaman-Chauhan/Machine_Learning_Assignments/blob/main/Assignment%201/OutputDecisionTree/DecisionFinal.png

3.2.3 Cost-Complexity Pruning

Post Pruning or Cost-Complexity Pruning (or ccp) is a technique used to reduce the complexity of a decision tree after it has fully grown. It removes specific nodes in the tree to improve its generalization ability and prevent overfitting. By iteratively evaluating the performance of different subtrees and removing those that do not improve or hurt the model's performance, the complexity of the tree is reduced while maintaining or improving its accuracy on unseen data. Reduced-Error Pruning is one of the techniques available for Post-Pruning. In scikit-learn, the 'DecisionTreeClassifier' class provides the ccp_alpha parameter that allows us to perform post-pruning.
Greater values of ccp_alpha imply more number of nodes pruned. It represents the tradeoff between the simplicity of the tree and its accuracy on the training data.

The decision tree criterion used is 'gini impurity.'

```python
path = cartree.cost_complexity_pruning_path(X_valid,Y_valid)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

```python
ccp_alphas
```

```
array([0.        , 0.00014781, 0.0001638 , 0.0001638 , 0.00016922,
       0.00016922, 0.00016922, 0.00016922, 0.00016922, 0.00016922,
       0.00016922, 0.00019205, 0.00019578, 0.00019656, 0.0001988 ,
       0.00019933, 0.00019933, 0.00019933, 0.00019933, 0.00019933,
       0.00020092, 0.00020335, 0.00021001, 0.00022099, 0.00022172,
       0.00022172, 0.00022172, 0.00022172, 0.00022172, 0.00022172,
       0.00022172, 0.00022172, 0.00022172, 0.00022172, 0.00022496,
       0.0002304 , 0.00023109, 0.00023343, 0.00023343, 0.00023343,
       0.00023343, 0.00023401, 0.00023769, 0.00023772, 0.00023957,
       0.00023957, 0.00023957, 0.00023957, 0.00023957, 0.00023957,
       0.00023957, 0.00023957, 0.00023957, 0.00023957, 0.00023957,
       0.0002457 , 0.0002457 , 0.0002457 , 0.0002457 , 0.0002457 ,
       0.0002457 , 0.0002457 , 0.0002457 , 0.0002457 , 0.0002457 ,
       0.0002457 , 0.0002457 , 0.0002457 , 0.0002457 , 0.0002457 ,
       0.0002457 , 0.0002457 , 0.0002457 , 0.0002457 , 0.0002457 ,
       0.0002457 , 0.0002457 , 0.0002457 , 0.0002457 , 0.0002457 ,
       0.0002457 , 0.0002457 , 0.0002457 , 0.00024694, 0.00024741,
```

The function 'cost_complexity_pruning_path( )' returns 2 values- alphas and impurities. We use this ccp_alphas as our ccp_alpha parameter in our decision tree. The list 'ccp_alphas' finds the weak point wrt the leaf nodes.

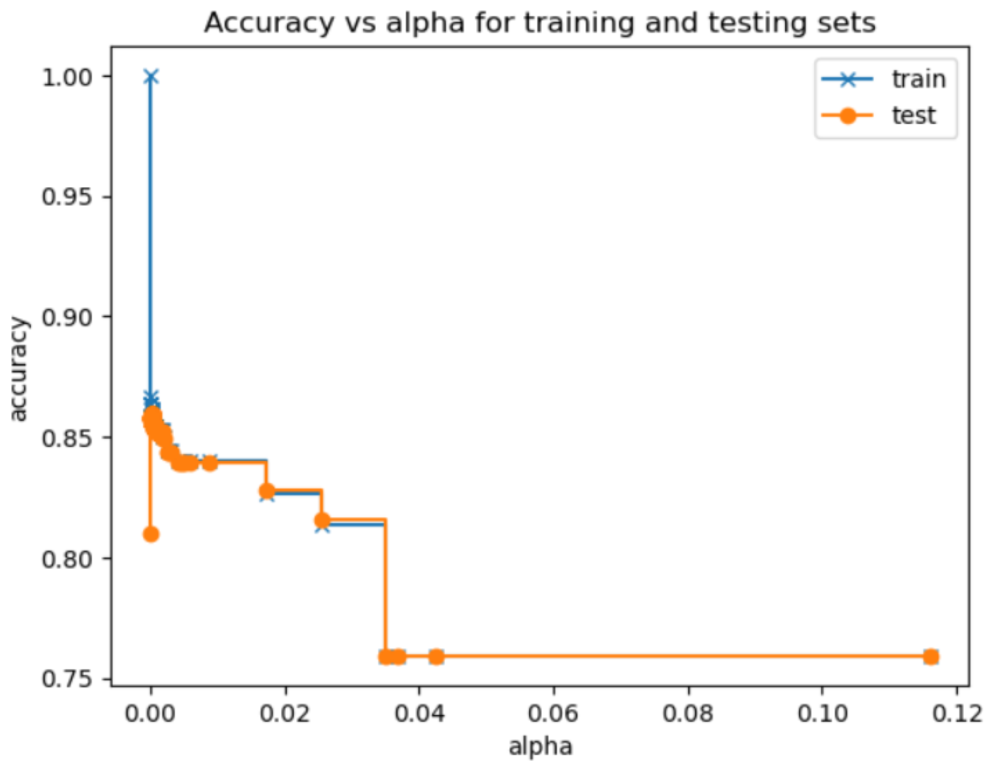
```python
clfs = []
for ccp_alpha in ccp_alphas:
    clf = tree.DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, Y_train)
    clfs.append(clf)
print("Number of nodes in the last tree is: {} with ccp_alpha: {}".format(
      clfs[-1].tree_.node_count, ccp_alphas[-1]))
```

```
Number of nodes in the last tree is: 1 with ccp_alpha: 0.11601805314672109
```

We run a loop where ccp_alpha takes up the values of the ccp_alphas list. A decision tree is classified for all these ccp_alphas values and stored in 'clfs[ ].'

```python
train_scores = [clf.score(X_train, Y_train) for clf in clfs]
test_scores = [clf.score(X_test, Y_test) for clf in clfs]

fig, ax = plt.subplots()
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas, train_scores, marker='x', label="train",
        drawstyle="steps-post")
ax.plot(ccp_alphas, test_scores, marker='o', label="test",
        drawstyle="steps-post")
ax.legend()
plt.show()
```

The accuracy scores of all classified decision trees stored in the clfs list is calculated on the training and testing datasets. These accuracy scores are then plotted.



Referring to the plot, we select the alpha value, which has reasonable accuracy for training and testing datasets.

```
max_score_test = max(test_scores)
inx = test_scores.index(max_score_test)
ccp_alphas[inx]
```

```
0.00022495982345852917
```

```
clf = tree.DecisionTreeClassifier(random_state=0, ccp_alpha=0.00022495982345852917)
clf.fit(X_train,Y_train)
```

The alpha value at the index with the maximum testing accuracy score is selected, and the final decision tree is classified. The accuracy of this decision tree is measured on testing data.

```
clf.score(X_test, Y_test)
```

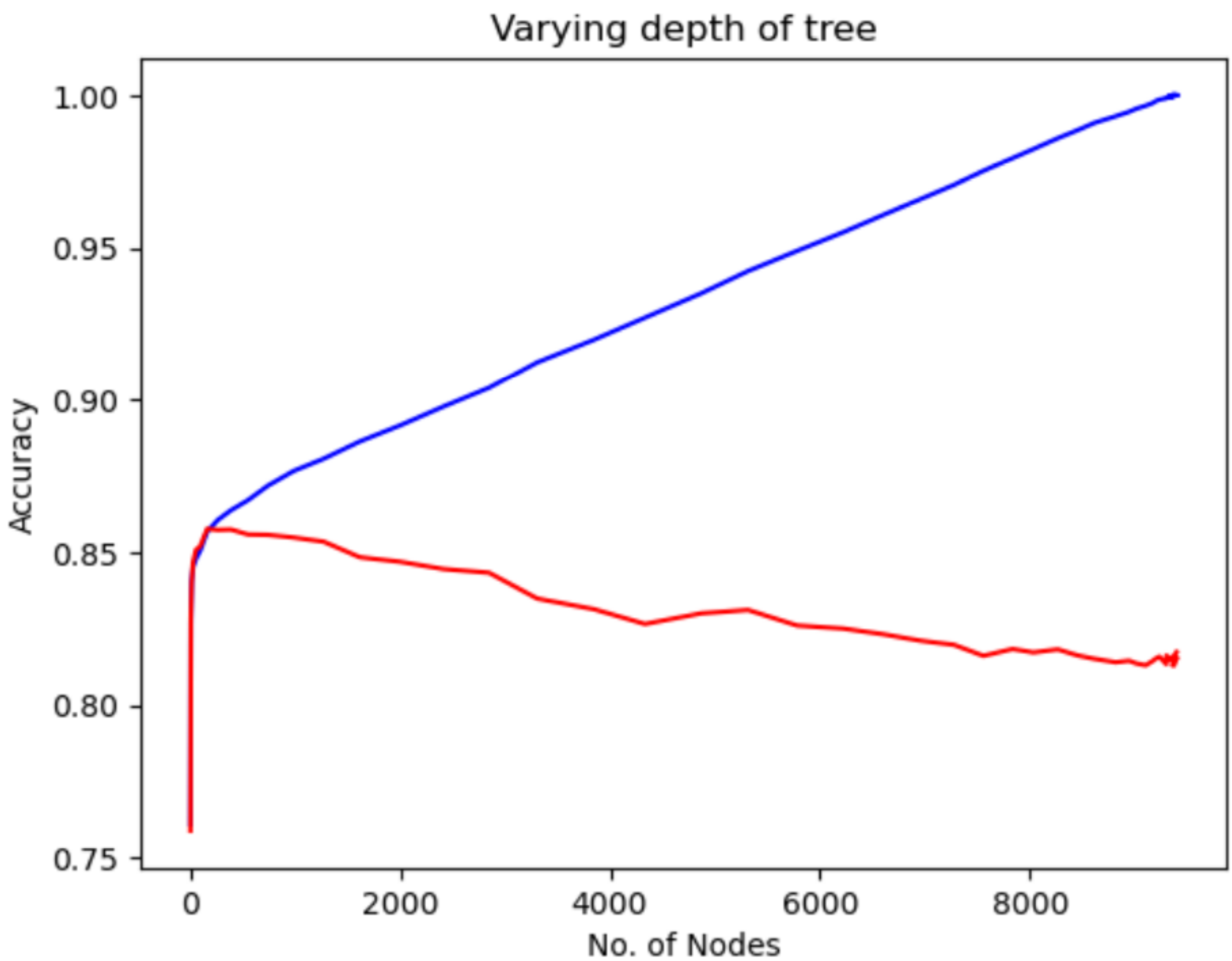
```
0.859231052696229
```

85.92% accuracy on the testing data set is obtained after using Cost - Complexity Pruning, a technique that falls under Post - Pruning.

3.3 Training Random Data

The training dataset is randomly (random_state = 0 or random_state = None) divided into training and testing data sets in a 0.33 proportion, i.e., the training data set is two times the testing data set. The training and testing data set is processed similarly as shown above. The missing values are handled, and labels are mapped. The testing data set is split equally and named validation and testing data.

The decision tree for the training data X_train and Y_train is plotted using 'entropy' as the criterion. The accuracy score for both the training and validation datasets are calculated. Below is the graph obtained from varying the depth of the tree with accuracy score and number of nodes at the axes.




```
maxacc = max(testscore)
inx = testscore.index(maxacc)
maxnodes = nofnodes[inx]
maxnodes,depthcon[inx]
```

```
(163, 7)
```

```
cartree = tree.DecisionTreeClassifier(criterion='entropy', max_depth=8)
cartree.fit(X_train, Y_train)
cartree.score(X_valid, Y_valid)
```

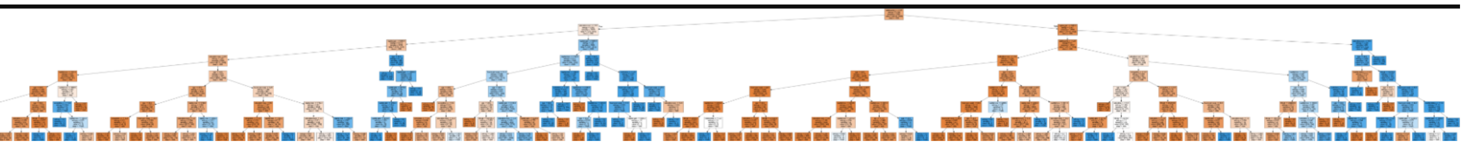
```
0.8554054054054054
```

```
cartree.score(X_test, Y_test)
```

```
0.8577570323056135
```

The depth for the maximum accuracy score of the validation data set is found, and a new decision tree is classified based on this depth. The accuracy of the latest decision tree is 85.5% on validation data and 85.8% on training data.

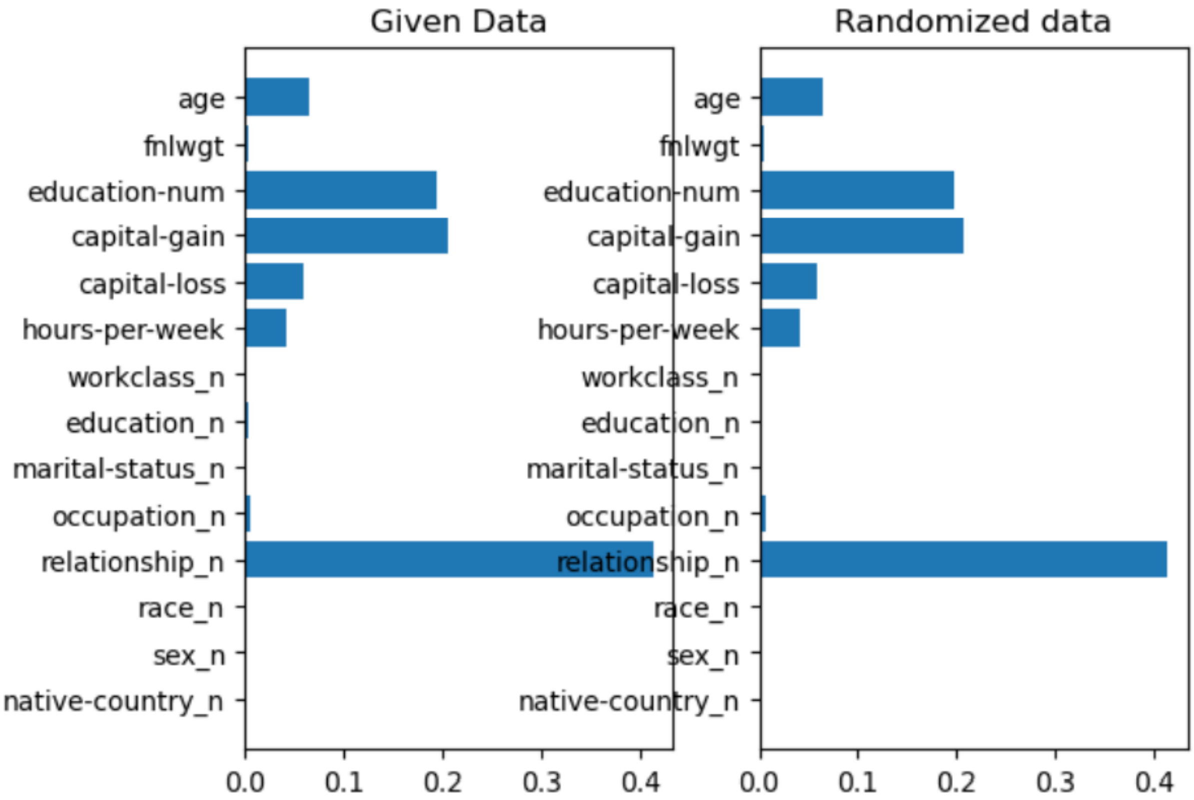
The decision tree plot using the 'graphviz' package would be like



https://github.com/Aryaman-Chauhan/Machine_Learning_Assignments/blob/main/Assignment%201/OutputDecisionTree/DecisionRandom.png

3.4 Comparing The Two Results

```
fig, ax = plt.subplots(1,2)
ax[0].barh(X_train.columns,fi)
ax[0].invert_yaxis()
ax[0].set_title("Given Data")
ax[1].barh(X_train.columns,fi1)
ax[1].invert_yaxis()
ax[1].set_title("Randomized data")
```



We can clearly see that even after splitting the dataset randomly into training and testing sets, the models and feature importances are very similar.

4. Random Forest

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(criterion='entropy', n_estimators=10)
rfc.fit(X_train, Y_train)
```

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

```
rfc.score(X_train,Y_train)
```

```
0.9877765554333211
```

```
rfc.score(X_valid, Y_valid)
```

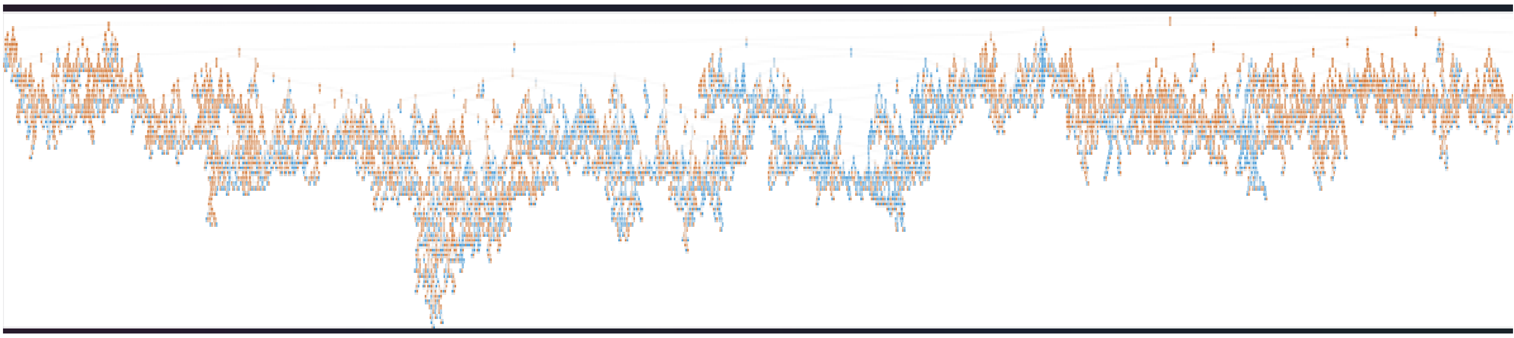
```
0.8491127931505149
```

The Python package 'sklearn.ensemble' is used to import RandomForestClassifier and is trained using the training datasets X_train and Y_train. The criterion for training would be 'entropy', and the parameter 'n_estimators' signifies the number of decision trees to be included in the random forest. Each decision tree is trained on a different random subset of the training data, and the final prediction is obtained by aggregating the predictions of all the trees (e.g., majority voting for classification). The training, validation, and testing dataset size would be the same as before, i.e.,

```
((32724, 14), (8059, 14), (8059, 14))
```

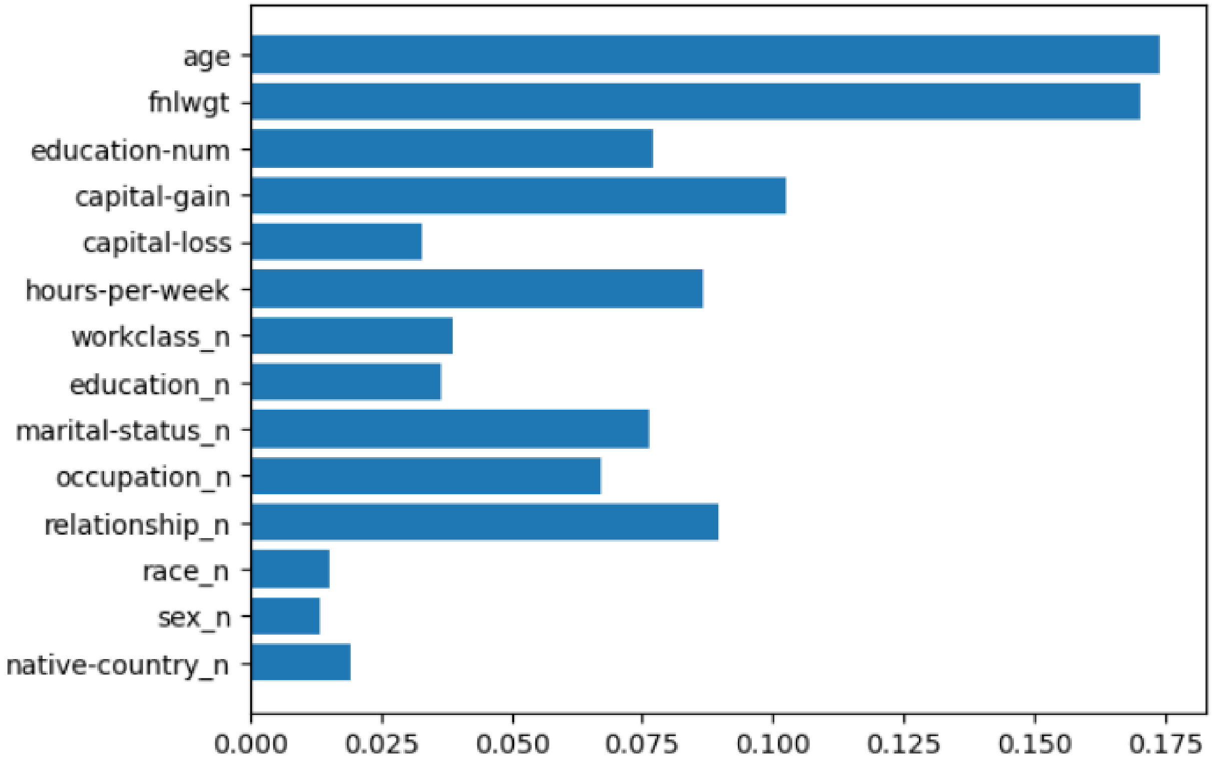
The value of n_estimators is slowly increased, and the accuracy score wrt the validation and training dataset is measured. It must be noted that increasing the 'n_estimators' value beyond a certain point may not always result in substantial improvement in performance, and it can also lead to longer training times.

The random forest plot would look like:

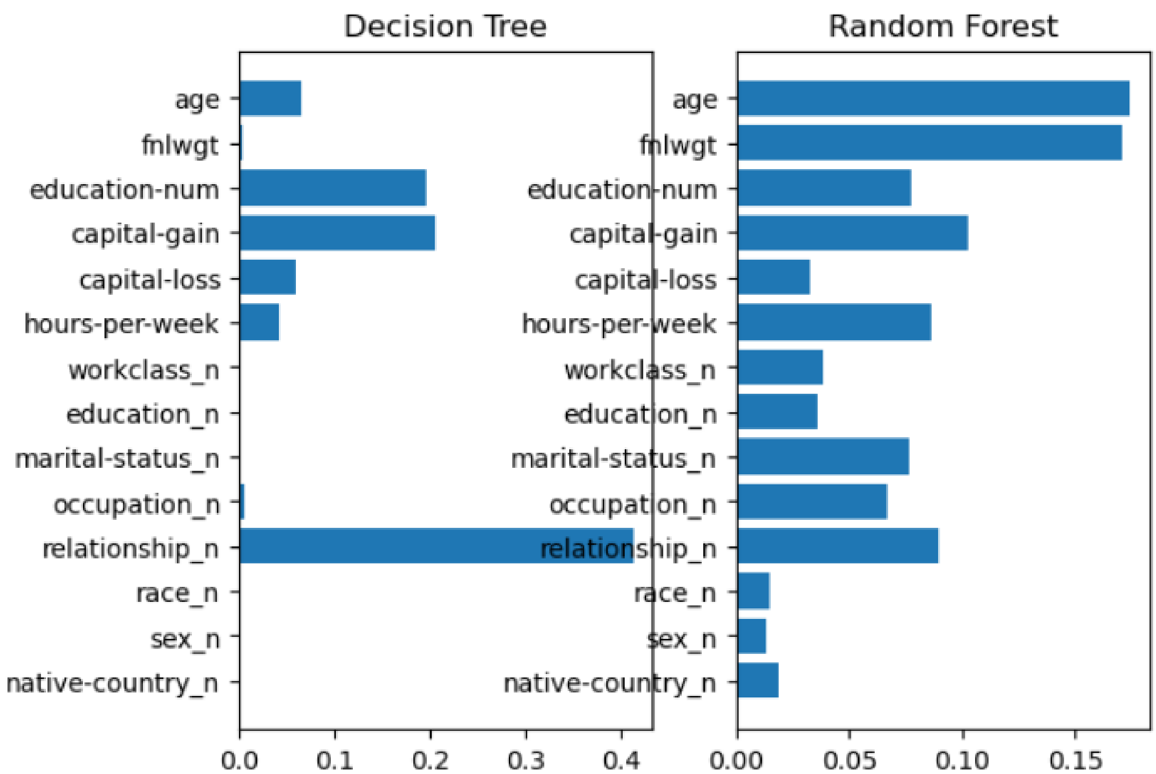


https://github.com/Aryaman-Chauhan/Machine_Learning_Assignments/blob/main/Assignment%201/outpurandomforest/tree4.png

The feature importance for the random forest 'rfc' is calculated and plotted in the figure below:



Now, we compare the 'feature_importances_' parameter for the random forest with that of the decision tree for a better understanding of all attributes.



It is evident that the random forest sees more value in all the attributes and is more meaningful from a data scientist's point of view. 'Age' can be easily understood as the factor affecting the data, while 'relationship' being the main attribute makes little sense.

We can also conclude that after increasing attributes beyond a certain point.

5. Results

The data was studied thoroughly, and decision tree classification was applied. The decision tree was built using training data. The testing data was divided equally into validation and testing data. The decision tree gave an accuracy of 99.14% on training data, 80.42% on validation data, and 81.01% on testing data before pruning. After pruning, the decision tree gave an accuracy of 87.73% on training data, 85.48% on validation data, and 85.77% on testing data. The accuracy of testing data after post-pruning is 85.92%.

Then the data were merged together and split as 67% training data and 33% testing data. The decision tree was again built using this new training data. The new decision tree has an accuracy of 99.43% on training data and 82.76% on validation data before pruning. After pruning, the accuracy improved to 85.54% on validation data and 85.77% on testing data.

Random forest classifier was run on n_estimators = 10, which gave an accuracy of 98.77% on training data, and 84.91% on validation data. On increasing the n_estimators to 30, it showed an accuracy of 99.81% on training data and 85.47% on validation data.

6. Conclusion

With this, we can finally report and complete our findings. For this assignment, we used Chefboost to understand the basics of the C4.5 algorithm, Owing to the lack of adaptability of the algorithm, we switched to sklearn's Decision Tree Classifier, which is a CART algorithm. We have used Graphviz, a tool that allowed us to plot the trees and save them inside the folder as .pngs. For One Hot Encoding, we have used LabelEncoder of sklearn, and for splitting data, we have used traintestsplit of sklearn. We have calculated the accuracy of the model by both pre-and post-pruning techniques.

We have also used the Random Tree Classifier of sklearn. ensemble, and .json and pickle libraries to store relevant data and models. We studied the difference between provided and random data and found the difference to be negligible, indicating that the data is sufficiently generalized. Then, we found the random tree classifier with 30 trees and had better accuracy, as well as an understanding of different attributes. We plotted the Feature importance graphs of all our models, thus understanding these models.

7. Rules Derived

For class <50k:

(relationship<=0.5)^(education <=12.5) ^ (capital gain <=5095.5) ^ (education <=8.5) ^ (capital loss < = 1791.5) ^ (age <=36.5) ^ (hour per week<=49) ^ (native country <= 34.5) And so on.

For class >=50k:

(Capital gain >7669.5) ^ (marital status <=1) ^ (hour per week > 35.5) ^ (flwgt > 33379) ^ (age >20) ^ (education <=10.5) ^ (capital gain > 7073.5) ^ (relationship > 0.5) And so on.

8. Appendix

The entire code would be as follows:

```
]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# import chefboost.Chefboost as chef
# import chefboost as chf
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn import tree
from sklearn.tree import plot_tree
import json
import pickle
import graphviz
%matplotlib inline
```

```
train_df = pd.read_csv("data1/train.csv", na_values="?")
test_df = pd.read_csv("data1/test.csv", na_values="?")
```

```
mode_values = train_df.median(numeric_only=True)
train_df.fillna(mode_values, inplace=True)
mode_values
```

```
age                  37.0
fnlwgt           178356.0
education-num        10.0
capital-gain          0.0
capital-loss          0.0
hours-per-week       40.0
dtype: float64
```

```
mode_values = []
for column in train_df.columns:
    if train_df[column].dtype == 'object':
        mode_value = train_df[column].mode()[0]
        mode_values.append(mode_value)
        train_df[column].fillna(mode_value, inplace=True)
mode_values
```

```
['Private',
 'HS-grad',
 'Married-civ-spouse',
 'Prof-specialty',
 'Husband',
 'White',
 'Male',
 'United-States',
 '<=50K']
```

```
train_df.rename(columns={'class':'Decision'}, inplace=True)
test_df.rename(columns={'class':'Decision'}, inplace=True)
```

```
train_df[25:30]
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	Decision
25	56	Local-gov	216851	Bachelors	13	Married-civ-spouse	Tech-support	Husband	White	Male	0	0	40	United-States	>50K
26	19	Private	168294	HS-grad	9	Never-married	Craft-repair	Own-child	White	Male	0	0	40	United-States	<=50K
27	54	Private	180211	Some-college	10	Married-civ-spouse	Prof-specialty	Husband	Asian-Pac-Islander	Male	0	0	60	South	>50K
28	39	Private	367260	HS-grad	9	Divorced	Exec-managerial	Not-in-family	White	Male	0	0	80	United-States	<=50K
29	49	Private	193366	HS-grad	9	Married-civ-spouse	Craft-repair	Husband	White	Male	0	0	40	United-States	<=50K

```python
#!pip install chefboost
from chefboost.training import Training
config = {'algorithm':'C4.5'}
```

```python
def gainratiocal(threshold:float, column:object) -> float:
    idx = train_df[train_df[f"{column}"] <= threshold].index
    tmp_df = train_df.copy()
    tmp_df["{column}"] = f">{threshold}"
    tmp_df.loc[idx, f"{column}"] = f"<={threshold}"
    grat = Training.findGains(tmp_df, config)['gains'][f"{column}"]
    return grat
```

```python
for i in range(1,100,10):
    a = gainratiocal(float(i),'hours-per-week')
    print(i,":",a)
```

```
1 : 0.01753148492873723
11 : 0.017751847016788638
21 : 0.018828987795244542
31 : 0.02064893040902423
41 : 0.023904627769250456
51 : 0.017662187191664586
61 : 0.008553549138154704
71 : 0.007151079969351616
81 : 0.00477984617092674
91 : 0.0023187326063295326
```

```
for i in range (35, 51,2):
    a = gainratiocal(float(i),'hours-per-week')
    print(i,":",a)
```

```
35 : 0.021594639037470862
37 : 0.02197617790402921
39 : 0.022415767382654897
41 : 0.023904627769250456
43 : 0.02476910237843594
45 : 0.024482106814724198
47 : 0.024970789695430925
49 : 0.026858965904353535
```

```
age = sorted(train_df['age'].unique())
for i in range(25,30):
    a = gainratiocal(float(i), 'age')
    print(i,":",a)
```

```
25 : 0.018878608763892753
26 : 0.018949020158395617
27 : 0.019008258539748947
28 : 0.018828616217681778
29 : 0.018589001236071493
```

For age, 27 is the decision boundary

```
fnlwg = sorted(train_df["fnlwgt"].unique())
for i in range(12200, 22200, 1000):
    a = gainratiocal(float(i), 'fnlwgt')
    print(i,":",a)
```

```
12200 : 0.04011843427056207
13200 : 0.04011843427056207
14200 : 0.04011860831741389
15200 : 0.04011287222370235
16200 : 0.04011287222370235
17200 : 0.04011287222370235
18200 : 0.04011287222370235
19200 : 0.04011209119723716
20200 : 0.04009173689623385
21200 : 0.04006503030840085
```

```
edunum = sorted(train_df['education-num'].unique())
```

```
for i in edunum:
    a = gainratiocal(float(i), 'education-num')
    print(i,":",a)
```

```
1 : 0.03192754615584799
2 : 0.03196053650702954
3 : 0.0321292294421387
4 : 0.032515666540499984
5 : 0.03304882274478729
6 : 0.03394432885606893
7 : 0.03527948364873703
8 : 0.03610098814658912
9 : 0.03933489231796834
10 : 0.05295739290033519
11 : 0.0604007911744962
12 : 0.06955873321174283
13 : 0.08772951926659586
14 : 0.1099725423325889
15 : 0.10501869293424891
16 : 0.0
```

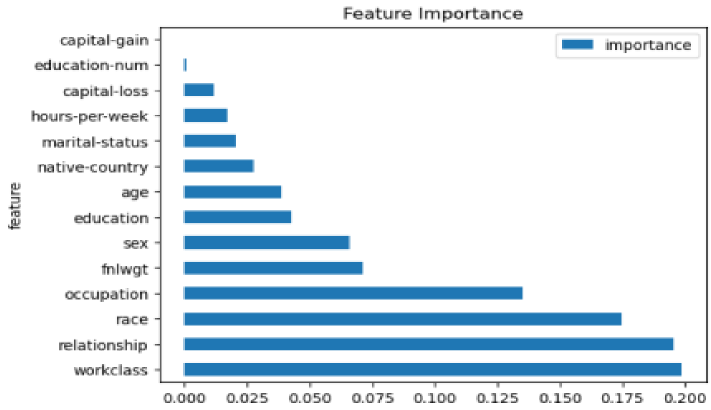
```
config = {'algorithm':'C4.5'}
model = chef.fit(train_df, config)
```

```
[INFO]:  4 CPU cores will be allocated in parallel running
C4.5  tree is going to be built…
-------------------------
```

```
finished in  2088.6931281089783  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  88.01019624704401 % on  32561  instances
Labels:  ['<=50K' '>50K']
Confusion matrix:  [[23115, 2299], [1605, 5542]]
Precision:  90.9538 %, Recall:  93.5073 %, F1:  92.2129 %
```

```
rules = "outputs/rules/rules.py"
fi = chef.feature_importance(rules).set_index("feature")
fi.plot(kind="barh", title="Feature Importance")
```

```
Decision rule:  outputs/rules/rules.py
<Axes: title={'center': 'Feature Importance'}, ylabel='feature'>
```




```
chfmdl = chef.load_model("model.pkl")
```

```
chef.evaluate(chfmdl, test_df)
```

```
-------------------------
Evaluate  test set
-------------------------
Accuracy:  82.07112585222038 % on  16281  instances
```

```
Labels:  ['<=50K' '>50K']
Confusion matrix:  [[11136, 1620], [1299, 2226]]
Precision:  87.3001 %, Recall:  89.5537 %, F1:  88.4125 %
```

```
train_df.columns
```

```
Index(['age', 'fnlwgt', 'education-num', 'capital-gain', 'capital-loss',
       'hours-per-week', 'workclass_n', 'education_n', 'marital-status_n',
       'occupation_n', 'relationship_n', 'race_n', 'sex_n', 'native-country_n',
       'class_n'],
      dtype='object')
```

```
train_df.select_dtypes(include=['object']).iloc[0]
```

```
Series([], Name: 0, dtype: float64)
```

These are the columns which need to be encoded, for which we will use label encoder, which allows us to perform one hot encoding.

```
le_workc = LabelEncoder()
le_educ = LabelEncoder()
le_mari = LabelEncoder()
le_occup = LabelEncoder()
le_relat = LabelEncoder()
le_race = LabelEncoder()
le_sex = LabelEncoder()
le_native = LabelEncoder()
le_class = LabelEncoder()
```

```
train_df["workclass_n"] = le_workc.fit_transform(train_df["workclass"])
train_df["education_n"] = le_educ.fit_transform(train_df["education"])
train_df["marital-status_n"] = le_mari.fit_transform(train_df["marital-status"])
train_df["occupation_n"] = le_occup.fit_transform(train_df["occupation"])
train_df["relationship_n"] = le_relat.fit_transform(train_df["relationship"])
train_df["race_n"] = le_race.fit_transform(train_df["race"])
train_df["sex_n"] = le_sex.fit_transform(train_df["sex"])
```

```
train_df["native-country_n"] = le_native.
  ↪fit_transform(train_df["native-country"])
train_df["class_n"] = le_class.fit_transform(train_df["class"])
train_df.iloc[0]
```

```
age                           39
workclass              State-gov
fnlwgt                     77516
education              Bachelors
education-num                 13
marital-status     Never-married
occupation          Adm-clerical
relationship       Not-in-family
race                       White
sex                         Male
capital-gain                2174
capital-loss                   0
hours-per-week                40
native-country     United-States
class                      <=50K
workclass_n                    6
education_n                    9
marital-status_n               4
occupation_n                   0
relationship_n                 1
race_n                         4
sex_n                          1
native-country_n              38
class_n                        0
Name: 0, dtype: object
```

Now that we have successfully encoded the classes, we can drop the unwanted columns

```
columns=['class','native-country','sex','race','relationship','occupation','marital-status','
X_train = train_df.drop(columns=columns,axis='columns')
X_train.head()
```

```
       age  fnlwgt  education-num  capital-gain  capital-loss  hours-per-week  \
39203   39  265685             10             0             0              65
16702   49  281647             14             0             0              45
43825   19  163885              9             0             0              40
48735   64   47298             16             0             0              45
34480   46  146786              6             0             0              50

       workclass_n  education_n  marital-status_n  occupation_n  \
39203          8.0           15                 0          14.0
16702          3.0           12                 2           3.0
43825          3.0           11                 4           6.0


48735          1.0           10                 2          10.0
34480          3.0            0                 0          13.0

       relationship_n  race_n  sex_n  native-country_n
39203               1       4      1              32.0
16702               0       4      1              38.0
43825               3       4      0              38.0
48735               0       4      1              38.0
34480               1       4      1              38.0
```

We have successfully encoded training data into X, and can similarly do that for Y, and train our model. Also, we'll retrive the mappings of each individual, and store them offline, to convert our test data when needed.

```
Y_train = train_df.class_n
Y_train.head()
```

```
0    0
1    0
2    0
3    0
4    0
Name: class_n, dtype: int64
```

```
label_mapping = {}
label_mapping['workclass'] = {label: encoded_label for label, encoded_label in␣
  ↪zip(train_df['workclass'], train_df['workclass_n'])}
label_mapping
```

```
columns.remove('class_n')
for column in columns:
    label_mapping[column] = {label: encoded_label for label, encoded_label in␣
  ↪zip(train_df[column], train_df[column+'_n'])}
label_mapping['sex']
```

```
{'Male': 1, 'Female': 0}
```

```
with open('label_mapping.json', 'w') as output_file:
    json.dump(label_mapping, output_file)
```

```
def encode_data_with_mapping(data_df, label_mapping_file='label_mapping.json'):
    # Load the label encoding mapping from the JSON file
    with open(label_mapping_file, 'r') as mapping_file:
        label_mapping = json.load(mapping_file)

    # Create a new DataFrame to store the encoded data
    encoded_data_df = pd.DataFrame()
```

```python
    for column in data_df.columns:
        if column not in label_mapping:
            encoded_data_df[column] = data_df[column]

    # Iterate over the columns of the input DataFrame
    for column in data_df.columns:
        if column in label_mapping:
            # Create a label encoder and set the classes using the label mapping
            label_encoder = LabelEncoder()
            label_encoder.classes_ = np.append(list(label_mapping[column].
↪values()), 'Unknown')

            # Apply the label encoding to the selected column
            encoded_column = data_df[column].astype(str).fillna('Unknown').
↪map(label_mapping[column]).fillna(len(label_mapping[column]))
            encoded_data_df[column+'_n'] = encoded_column

    return encoded_data_df
```

We have now defined a fuction which will automatically do all of this for us

```python
train_df = pd.read_csv("data1/train.csv")
train_df = encode_data_with_mapping(train_df)
train_df.head()
```

	age	fnlwgt	education-num	capital-gain	capital-loss	hours-per-week	\
0	39	77516	13	2174	0	40	
1	50	83311	13	0	0	13	
2	38	215646	9	0	0	40	
3	53	234721	7	0	0	40	
4	28	338409	13	0	0	40	

	workclass_n	education_n	marital-status_n	occupation_n	relationship_n	\
0	6.0	9	4	0.0	1	
1	5.0	9	2	3.0	0	
2	3.0	11	0	5.0	1	
3	3.0	1	2	5.0	0	
4	3.0	9	2	9.0	5	

	race_n	sex_n	native-country_n	class_n
0	4	1	38.0	0
1	4	1	38.0	0
2	4	1	38.0	0
3	2	1	38.0	0
4	2	0	4.0	0

```python
train_df = pd.read_csv("data1/train.csv", na_values="?")
train_df = encode_data_with_mapping(train_df)
X_train = train_df.drop(columns='class_n', axis='columns')
Y_train = train_df.class_n
```

```python
test_df = pd.read_csv("data1/test.csv", na_values="?")
test_df = encode_data_with_mapping(test_df)
test_df.head()
```

	age	fnlwgt	education-num	capital-gain	capital-loss	hours-per-week	\
0	25	226802	7	0	0	40	
1	38	89814	9	0	0	50	
2	28	336951	12	0	0	40	
3	44	160323	10	7688	0	40	
4	18	103497	10	0	0	30	

	workclass_n	education_n	marital-status_n	occupation_n	relationship_n	\
0	3.0	1	4	6.0	3	
1	3.0	11	2	4.0	0	
2	1.0	7	2	10.0	0	
3	3.0	15	2	6.0	0	
4	8.0	15	4	14.0	3	

	race_n	sex_n	native-country_n	class_n
0	2	1	38.0	0
1	4	1	38.0	0
2	4	1	38.0	1
3	2	1	38.0	1
4	4	0	38.0	0

```python
X_test = test_df.drop(columns='class_n', axis='columns')
Y_test = test_df.class_n
X_valid, X_test, Y_valid, Y_test = train_test_split(X_test, Y_test, test_size=0.
↪5, random_state=10)
X_valid.shape, X_test.shape
```

((8140, 14), (8141, 14))

We have successfully split the test set into testing and validating dataset

1.5.2 Training and scoring

```python
cartree = tree.DecisionTreeClassifier(criterion='entropy')
cartree.fit(X_train, Y_train)
```

DecisionTreeClassifier(criterion='entropy')

```python
cartree.tree_.node_count
```

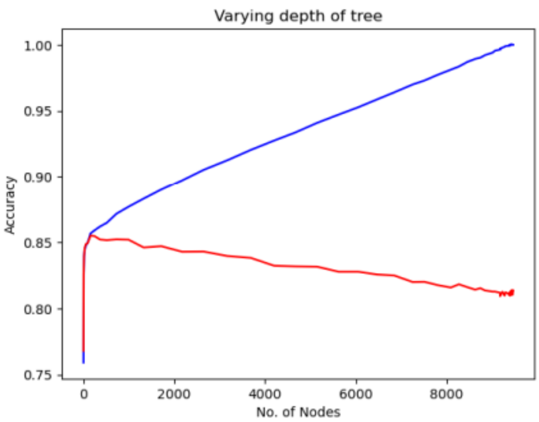
[68]: 9431

[69]:
```python
cartree.get_depth()
```

[69]: 49

```python
trainscore = []
testscore = []
nofnodes = []
depthcou = []
for i in range(1, 55):
    clf = tree.DecisionTreeClassifier(criterion = 'entropy', max_depth=i)
    clf.fit(X_train, Y_train)
    trainscore.append(clf.score(X_train, Y_train))
    testscore.append(clf.score(X_valid, Y_valid))
    nofnodes.append(clf.tree_.node_count)
    depthcou.append(i)
```

```python
plt.plot(nofnodes,trainscore,'b')
plt.plot(nofnodes,testscore, 'r')
plt.xlabel("No. of Nodes")
plt.ylabel("Accuracy")
plt.title("Varying depth of tree")
```




```python
maxacc = max(testscore)
inx = testscore.index(maxacc)
maxnodes = nofnodes[inx]
maxnodes, depthcou[inx]
```

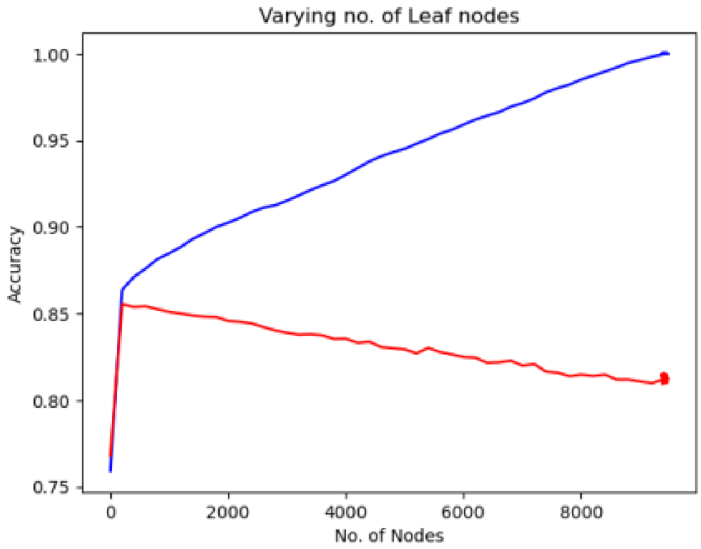
(157, 7)

Doing the same, but this time restricting no of nodes

9]:
```python
trainscore1 = []
testscore1 = []
nofnodes1 = []
noofleaf = []
for i in range(2, 9503, 100):
    clf = tree.DecisionTreeClassifier(criterion = 'entropy', max_leaf_nodes=i)
    clf.fit(X_train, Y_train)
    trainscore1.append(clf.score(X_train, Y_train))
    testscore1.append(clf.score(X_valid, Y_valid))
    nofnodes1.append(clf.tree_.node_count)
    noofleaf.append(i)
```

```python
plt.plot(nofnodes1,trainscore1,'b')
plt.plot(nofnodes1,testscore1, 'r')
plt.xlabel("No. of Nodes")
plt.ylabel("Accuracy")
plt.title("Varying no. of Leaf nodes")
```

Text(0.5, 1.0, 'Varying no. of Leaf nodes')



]:
```
maxacc = max(testscore1)
inx = testscore1.index(maxacc)
maxnodes = nofnodes1[inx]
maxleaf = noofleaf[inx]
maxnodes,maxleaf
```

]: (203, 102)

```
cartree = tree.DecisionTreeClassifier(criterion='entropy', max_depth=8)
cartree.fit(X_train, Y_train)
```

```
cartree.score(X_valid, Y_valid)
```

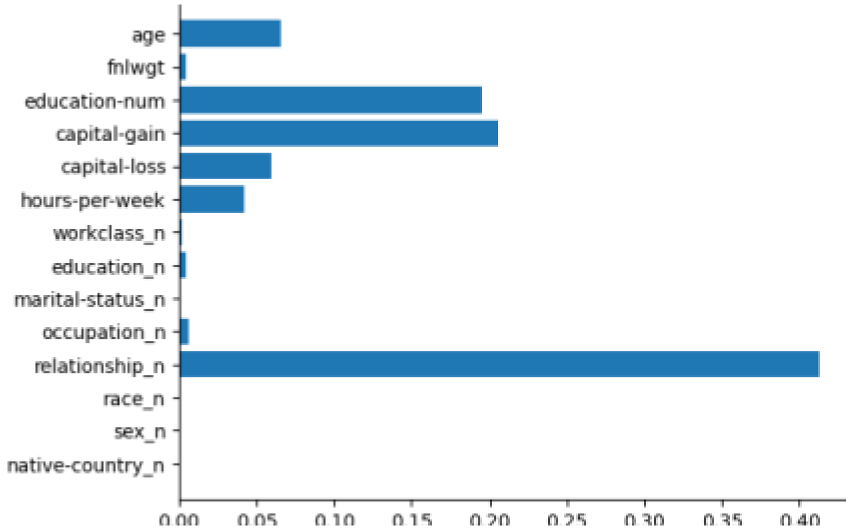
: 0.8547911547911548

:
```
cartree.score(X_test, Y_test)
```

: 0.8577570323056135

:
```
with open("DecisionTreeFinal.pkl",'wb') as f:
    pickle.dump(cartree,f)
```

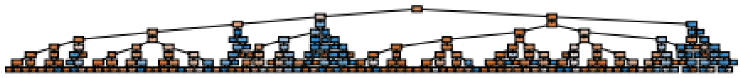
```
fi = cartree.feature_importances_
fig, ax = plt.subplots()
ax.barh(X_train.columns,fi)
ax.invert_yaxis()
```




```
fig.savefig('FeatureDF.png')
```

```
with open("DecisionTreeFinal.pkl",'rb') as f:
    cartree = pickle.load(f)
dot_data = tree.export_graphviz(cartree, out_file=None, feature_names=X_train.
 ↪columns, class_names=['<50K','>=50K'], filled=True)
```

```
# Create a graph from the Graphviz data
graph = graphviz.Source(dot_data)

# Render the graph
graph.render("decision_tree")

# Display the graph
graph
```



```
X_train,X_test,Y_train, Y_test = train_test_split(X,Y,test_size=0.
 ↪33,random_state=10)
X_train.shape,X_test.shape
```

((32724, 14), (16118, 14))

```
X_valid, X_test, Y_valid, Y_test = train_test_split(X_test, Y_test, test_size=0
 ↪5, random_state=10)
X_valid.shape, X_test.shape
```

((8059, 14), (8059, 14))

```python
cartree = tree.DecisionTreeClassifier(criterion='entropy')
cartree.fit(X_train, Y_train)
```

```
DecisionTreeClassifier(criterion='entropy')
```

```python
cartree.tree_.node_count
```

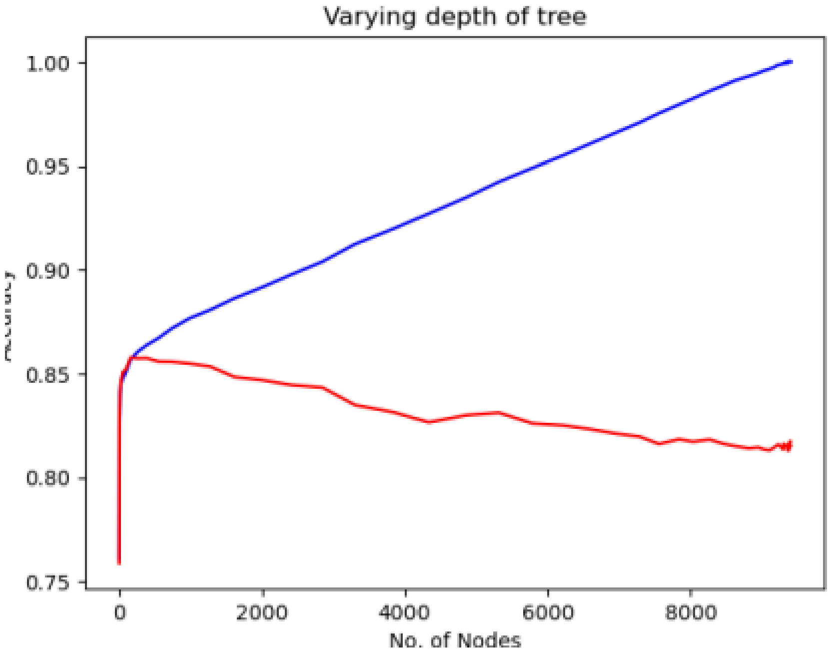
```
9365
```

```python
cartree.get_depth()
```

```
48
```

```python
trainscore = []
testscore = []
nofnodes = []
depthcon = []
for i in range(1, 55):
    clf = tree.DecisionTreeClassifier(criterion = 'entropy', max_depth=i)
    clf.fit(X_train, Y_train)
    trainscore.append(clf.score(X_train, Y_train))
    testscore.append(clf.score(X_valid, Y_valid))
    nofnodes.append(clf.tree_.node_count)
    depthcon.append(i)
```

```python
plt.plot(nofnodes,trainscore,'b')
plt.plot(nofnodes,testscore, 'r')
plt.xlabel("No. of Nodes")
plt.ylabel("Accuracy")
plt.title("Varying depth of tree")
```



```python
maxacc = max(testscore)
inx = testscore.index(maxacc)
maxnodes = nofnodes[inx]
maxnodes,depthcon[inx]
```

```
(163, 7)
```

```python
cartree = tree.DecisionTreeClassifier(criterion='entropy', max_depth=8)
cartree.fit(X_train, Y_train)
cartree.score(X_valid, Y_valid)
```

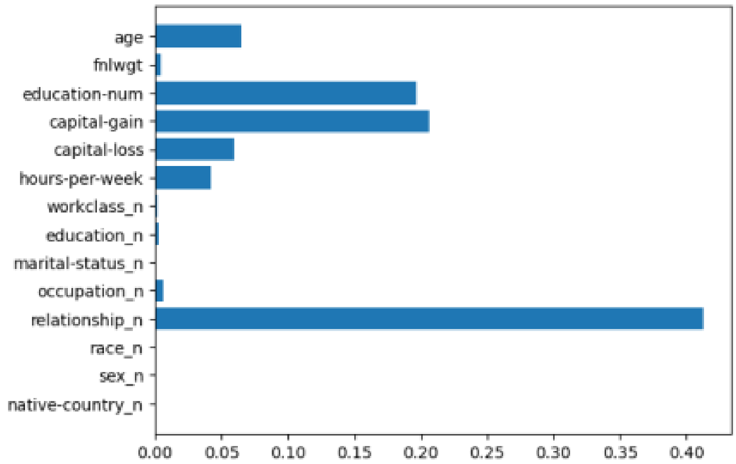
```
0.8554054054054054
```

```python
cartree.score(X_test, Y_test)
```

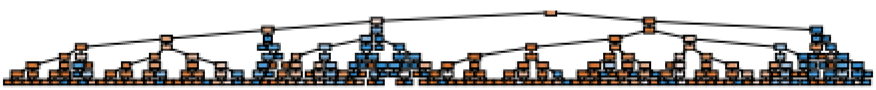
```
0.8577570323056135
```

```python
with open("DecisionTreeRandom.pkl",'wb') as f:
    pickle.dump(cartree,f)
```

```python
fi1 = cartree.feature_importances_
fig, ax = plt.subplots()
ax.barh(X_train.columns,fi1)
ax.invert_yaxis()
```

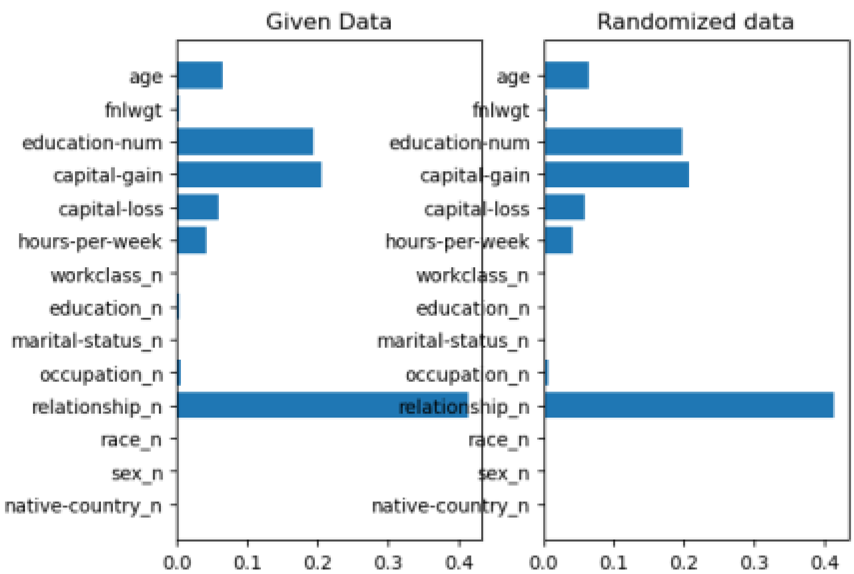



```python
with open("DecisionTreeRandom.pkl",'rb') as f:
    cartree = pickle.load(f)
dot_data = tree.export_graphviz(cartree, out_file=None, feature_names=X_train.
 ↪columns, class_names=['<50K','>=50K'], filled=True)

# Create a graph from the Graphviz data
graph = graphviz.Source(dot_data)

# Render the graph
graph.render("decision_tree")

# Display the graph
graph
```



```python
fig, ax = plt.subplots(1,2)
ax[0].barh(X_train.columns,fi)
ax[0].invert_yaxis()
ax[0].set_title("Given Data")
ax[1].barh(X_train.columns,fi1)
ax[1].invert_yaxis()
ax[1].set_title("Randomized data")
```

```
Text(0.5, 1.0, 'Randomized data')
```



```python
X_train.shape,X_test.shape, X_valid.shape
```

```
((32724, 14), (8059, 14), (8059, 14))
```

```python
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(criterion='entropy', n_estimators=10)
rfc.fit(X_train, Y_train)
```

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

```python
rfc.score(X_train,Y_train)
```

```
0.9877765554333211
```

```python
rfc.score(X_valid, Y_valid)
```

```
0.8491127931505149
```

```python
rfc = RandomForestClassifier(criterion='entropy', n_estimators=30)
rfc.fit(X_train, Y_train)
```

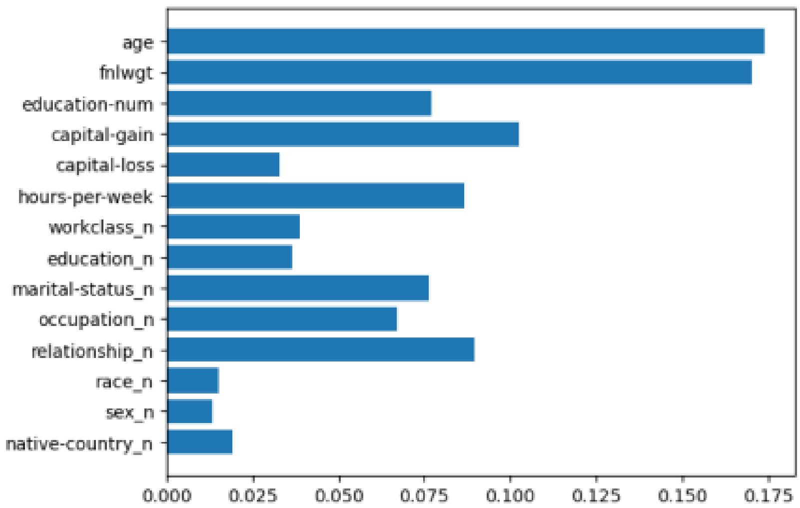
```
RandomForestClassifier(criterion='entropy', n_estimators=30)
```

```python
rfc.score(X_train,Y_train), rfc.score(X_valid, Y_valid)
```

```
(0.9981359247035815, 0.8546966124829384)
```

```python
with open("RandomForestModel.pkl",'wb') as f:
    pickle.dump(rfc,f)
```

```python
fi2 = rfc.feature_importances_
fig, ax = plt.subplots()
ax.barh(X_train.columns,fi2)
ax.invert_yaxis()
```




```python
fig.savefig('FeatureRF.png')
```

```python
fig, ax = plt.subplots(1,2)
ax[0].barh(X_train.columns,fi1)
ax[0].invert_yaxis()
ax[0].set_title("Decision Tree")
ax[1].barh(X_train.columns,fi2)
ax[1].invert_yaxis()
ax[1].set_title("Random Forest")
```