GOVT. AKHTAR NAWAZ KHAN (SHAHEED)
DEGREE COLLEGE KTS, HARIPUR
DEPARTMENT OF COMPUTER SCIENCE
Near Chandni Chowk, Sector No. 1, Khalabat Township, Tehsil & District Haripur
Phone: **0995-621918**, Email: **gdckts@gmail.com**

Pre-Course Lab Activities

Pre-Lab **Date: 04-02-2025**

Course: Deep Learning and Neural Networks

Subject Teacher: Dr. Abid Ali

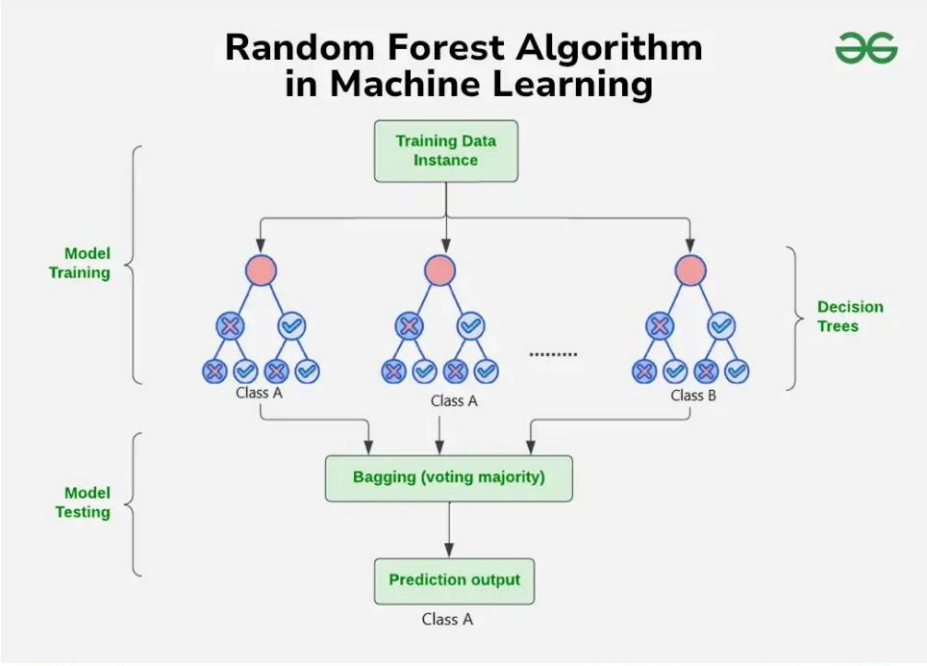
Implementation of Random Forest Classifier in Machine Learning

Introduction: Random Forest Algorithm

Random Forest algorithm is a powerful tree learning technique in Machine Learning. It works by creating a number of Decision Trees during the training phase. Each tree is constructed using a random subset of the data set to measure a random subset of features in each partition. This randomness introduces variability among individual trees, reducing the risk of overfitting and improving overall prediction performance.

In prediction, the algorithm aggregates the results of all trees, either by voting (for classification tasks) or by averaging (for regression tasks) This collaborative decision-making process, supported by multiple trees with their insights, provides an example stable and precise results. Random forests are widely used for classification and regression functions, which are known for their ability to handle complex data, reduce overfitting, and provide reliable forecasts in different environments.

GOVT. AKHTAR NAWAZ KHAN (SHAHEED)
DEGREE COLLEGE KTS, HARIPUR
DEPARTMENT OF COMPUTER SCIENCE
Near Chandni Chowk, Sector No. 1, Khalabat Township, Tehsil & District Haripur
Phone: **0995-621918**, Email: **gdckts@gmail.com**



Working

- Step 1: Select random K data points from the training set.
- Step 2:Build the decision trees associated with the selected data points(Subsets).
- Step 3:Choose the number N for decision trees that you want to build.
- Step 4:Repeat Step 1 and 2.
- Step 5: For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes.

Objectives

- To understand the fundamentals of the Random Forest Classifier and its role in machine learning for classification tasks.
- To develop the ability to implement and evaluate the Random Forest Classifier using Python and Scikit-Learn.
- To analyze the performance of the Random Forest model using key metrics like precision, recall, and ROC-AUC and compare it to Decision Tree results.

Learning Outcomes

GOVT. AKHTAR NAWAZ KHAN (SHAHEED)
DEGREE COLLEGE KTS, HARIPUR
DEPARTMENT OF COMPUTER SCIENCE
Near Chandni Chowk, Sector No. 1, Khalabat Township, Tehsil & District Haripur
Phone: **0995-621918**, Email: gdckts@gmail.com

- Students will be able to implement a Random Forest Classifier in Python for a given dataset and visualize its decision-making process.
- Students will critically evaluate the strengths and limitations of Random Forests by comparing their results with a single Decision Tree model on training and testing datasets.

Once we understand how a single decision tree thinks, we can transfer this knowledge to an entire forest of trees.

```
import numpy as np
import pandas as pd

# Set random seed to ensure reproducible runs
RSEED = 50
```

Basic Problem

To begin, we'll use a very simple problems with only two features and two classes. This is a binary classification problem.

First, we create the features X and the labels y. There are only two features, which will allow us to visualize the data and which makes this a very easy problem.

```
X = np.array([[2, 2],
              [2, 1],
              [2, 3],
              [1, 2],
              [1, 1],
              [3, 3]])

y = np.array([0, 1, 1, 1, 0, 1])
```

Data Visualization

To get a sense of the data, we can graph the data points with the number showing the label.

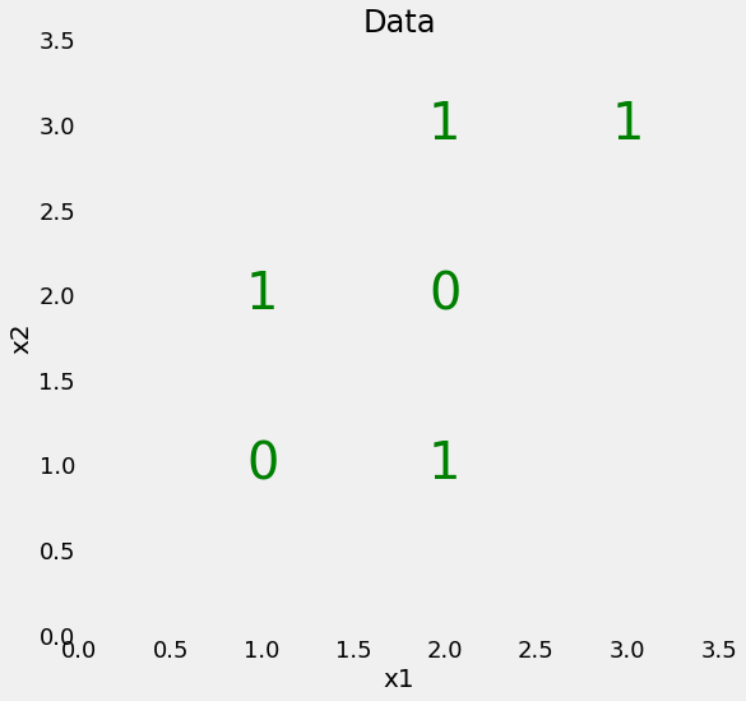
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

```
# Plot formatting
plt.style.use('fivethirtyeight')
plt.rcParams['font.size'] = 18
plt.figure(figsize = (8, 8))

# Plot each point as the label
for x1, x2, label in zip(X[:, 0], X[:, 1], y):
   plt.text(x1, x2, str(label), fontsize = 40, color = 'g',
         ha='center', va='center')

# Plot formatting
plt.grid(None);
plt.xlim((0, 3.5));
plt.ylim((0, 3.5));
plt.xlabel('x1', size = 20); plt.ylabel('x2', size = 20); plt.title('Data', size = 24)
```

Output:



Single Decision Tree

GOVT. AKHTAR NAWAZ KHAN (SHAHEED)
DEGREE COLLEGE KTS, HARIPUR
DEPARTMENT OF COMPUTER SCIENCE
Near Chandni Chowk, Sector No. 1, Khalabat Township, Tehsil & District Haripur
Phone: **0995-621918**, Email: gdckts@gmail.com

Here we quickly build and train a single decision tree on the data using Scikit-Learn. The tree will learn how to separate the points, building a flowchart of questions based on the feature values and the labels. At each stage, the decision tree makes splits by maximizing the reduction in Gini impurity. We'll use the default hyperparameters for the decision tree which means it can grow as deep as necessary in order to completely separate the classes. This will lead to overfitting because the model memorizes the training data, and in practice, we usually want to limit the depth of the tree so it can generalize to testing data.

```
from sklearn.tree import DecisionTreeClassifier

# Make a decision tree and train
tree = DecisionTreeClassifier(random_state=RSEED)
tree.fit(X, y)
```

```
print(f'Decision tree has {tree.tree_.node_count} nodes with maximum depth
{tree.tree_.max_depth}.')
```

Our decision tree formed 9 nodes and reached a maximum depth of 3. It will have achieved 100% accuracy on the training data because we did not limit the depth and it therefore can classify every training point perfectly.

```
print(f'Model Accuracy: {tree.score(X, y)}')
```

Visualize Decision Tree

To get a sense of how the decision tree "thinks", it's helpful to visualize the entire structure. This will show each node in the tree which we can use to make new predictions. Because the tree is relatively small, we can understand the entire image. First we export the tree as a dot file making sure to label the features and the classes.

```
from sklearn.tree import export_graphviz

# Export as dot
export_graphviz(tree, 'tree.dot', rounded = True,
        feature_names = ['x1', 'x2'],
        class_names = ['0', '1'], filled = True)
```

Next we use a system command and the Graphziv dot function to convert to a png (image). This requires Graphviz to be installed on your computer.

```
from subprocess import call
# Convert to png
call(['dot', '-Tpng', 'tree.dot', '-o', 'tree.png', '-Gdpi=400']);
```

Finally, we display the entire tree.

```
from IPython.display import Image
Image('tree.png')
```

Model Accuracy

```
# Limit maximum depth and train
short_tree = DecisionTreeClassifier(max_depth = 2, random_state=RSEED)
short_tree.fit(X, y)

print(f'Model Accuracy: {short_tree.score(X, y)}')
```

Export

```
# Export as dot
export_graphviz(short_tree, 'shorttree.dot', rounded = True,
        feature_names = ['x1', 'x2'],
        class_names = ['0', '1'], filled = True)

call(['dot', '-Tpng', 'shorttree.dot', '-o', 'shorttree.png', '-Gdpi=400']);
Image('shorttree.png')
```

On Real Dataset

The following data set is from the Centers for Disease Control and Prevention (CDC) and includes socioeconomic and lifestyle indicators for hundreds of thousands of individuals. The objective is to predict the overall health of an individual: either 0 for poor health or 1 for good health. We'll limit the data to 100,000 individuals to speed up training.

The problem is imbalanced (far more of one label than another) so for assessing performance, we'll use recall, precision, receiver operating characteristic area under the curve (ROC AUC), and also plot the ROC curve. Accuracy is not a useful metric when dealing with an imbalanced problem.

Data Cleaning

```
df = pd.read_csv('data/2015_health.csv').sample(100000, random_state = RSEED)
df.head()
```

```
df = df.select_dtypes('number')
```

Label Distribution

```
df['_RFHLTH'] = df['_RFHLTH'].replace({2: 0})
df = df.loc[df['_RFHLTH'].isin([0, 1])].copy()
df = df.rename(columns = {'_RFHLTH': 'label'})
df['label'].value_counts()
```

The label imbalanced means that accuracy is not the best metric. We won't do any data exploration in this notebook, but in general, exploring the data is a best practice. This can help you for feature engineering (which we also won't do here) or by identifying and correcting anomalies / mistakes in the data. Below, we drop a number of columns that we should not use for modeling (they are different versions of the labels).

```
# Remove columns with missing values
df = df.drop(columns = ['POORHLTH', 'PHYSHLTH', 'GENHLTH', 'PAINACT2',
                        'QLMENTL2', 'QLSTRES2', 'QLHLTH2', 'HLTHPLN1', 'MENTHLTH'])
```

Split Data into Training and Testing Set

To assess our predictions, we'll need to use a training and a testing set. The model learns from the training data and then makes predictions on the testing data. Since we have the

correct answers for the testing data, we can tell how well the model is able to generalize to new data. It's important to only use the testing set once, because this is meant to be an estimate of how well the model will perform on new data.

We'll save 30% of the examples for testing.

```
from sklearn.model_selection import train_test_split

# Extract the labels
labels = np.array(df.pop('label'))

# 30% examples in test data
train, test, train_labels, test_labels = train_test_split(df, labels,
                                        stratify = labels,
                                        test_size = 0.3,
                                        random_state = RSEED)
```

Imputation of Missing values

We'll fill in the missing values with the mean of the column. It's important to note that we fill in missing values in the test set with the mean of columns in the training data. This is necessary because if we get new data, we'll have to use the training data to fill in any missing values.

```
train = train.fillna(train.mean())
test = test.fillna(test.mean())

# Features for feature importances
features = list(train.columns)
```

```
train.shape
```

```
test.shape
```

Decision Tree On Real Data

```
# Train tree
tree.fit(train, train_labels)
```

```
print(f'Decision tree has {tree.tree_.node_count} nodes with maximum depth
{tree.tree_.max_depth}.')
```

Access Decision Tree Performance

Given the number of nodes in our decision tree and the maximum depth, we expect it has overfit to the training data. This means it will do much better on the training data than on the testing data.

```
# Make probability predictions
train_probs = tree.predict_proba(train)[:, 1]
probs = tree.predict_proba(test)[:, 1]

train_predictions = tree.predict(train)
predictions = tree.predict(test)
```

```
from sklearn.metrics import precision_score, recall_score, roc_auc_score,
roc_curve

print(f'Train ROC AUC Score: {roc_auc_score(train_labels, train_probs)}')
print(f'Test ROC AUC  Score: {roc_auc_score(test_labels, probs)}')
```

```
print(f'Baseline ROC AUC: {roc_auc_score(test_labels, [1 for _ in
range(len(test_labels))])}')
```

Evaluate the Decision Tree

We'll write a short function that calculates a number of metrics for the baseline (guessing the most common label in the training data), the testing predictions, and the training predictions. The function also plots the ROC curve where a better model is to the left and towards the top.

```
def evaluate_model(predictions, probs, train_predictions, train_probs):
    """Compare machine learning model to baseline performance.
    Computes statistics and shows ROC curve."""

    baseline = {}
```

```
  baseline['recall'] = recall_score(test_labels, [1 for _ in range(len(test_labels))])
  baseline['precision'] = precision_score(test_labels, [1 for _ in
range(len(test_labels))])
  baseline['roc'] = 0.5

  results = {}

  results['recall'] = recall_score(test_labels, predictions)
  results['precision'] = precision_score(test_labels, predictions)
  results['roc'] = roc_auc_score(test_labels, probs)

  train_results = {}
  train_results['recall'] = recall_score(train_labels, train_predictions)
  train_results['precision'] = precision_score(train_labels, train_predictions)
  train_results['roc'] = roc_auc_score(train_labels, train_probs)

  for metric in ['recall', 'precision', 'roc']:
    print(f'{metric.capitalize()} Baseline: {round(baseline[metric], 2)} Test:
{round(results[metric], 2)} Train: {round(train_results[metric], 2)}')

  # Calculate false positive rates and true positive rates
  base_fpr, base_tpr, _ = roc_curve(test_labels, [1 for _ in range(len(test_labels))])
  model_fpr, model_tpr, _ = roc_curve(test_labels, probs)

  plt.figure(figsize = (8, 6))
  plt.rcParams['font.size'] = 16

  # Plot both curves
  plt.plot(base_fpr, base_tpr, 'b', label = 'baseline')
  plt.plot(model_fpr, model_tpr, 'r', label = 'model')
  plt.legend();
  plt.xlabel('False Positive Rate'); plt.ylabel('True Positive Rate'); plt.title('ROC
Curves');
```

```
from collections import Counter
print(Counter(probs))
print(Counter(predictions))
```

```
evaluate_model(predictions, probs, train_predictions, train_probs)
```

Confusion Matrix

```
from sklearn.metrics import confusion_matrix
import itertools

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Oranges):
  """
  This function prints and plots the confusion matrix.
  Normalization can be applied by setting `normalize=True`.
  Source: http://scikit-
learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
  """
  if normalize:
    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    print("Normalized confusion matrix")
  else:
    print('Confusion matrix, without normalization')

  print(cm)

  plt.figure(figsize = (10, 10))
  plt.imshow(cm, interpolation='nearest', cmap=cmap)
  plt.title(title, size = 24)
  plt.colorbar(aspect=4)
  tick_marks = np.arange(len(classes))
  plt.xticks(tick_marks, classes, rotation=45, size = 14)
  plt.yticks(tick_marks, classes, size = 14)

  fmt = '.2f' if normalize else 'd'
  thresh = cm.max() / 2.

  # Labeling the plot
  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt), fontsize = 20,
```

```
            horizontalalignment="center",
            color="white" if cm[i, j] > thresh else "black")

    plt.grid(None)
    plt.tight_layout()
    plt.ylabel('True label', size = 18)
    plt.xlabel('Predicted label', size = 18)
```

```
cm = confusion_matrix(test_labels, predictions)
plot_confusion_matrix(cm, classes = ['Poor Health', 'Good Health'],
                      title = 'Health Confusion Matrix')
```

Feature Importances

```
fi = pd.DataFrame({'feature': features,
                   'importance': tree.feature_importances_}).\
                    sort_values('importance', ascending = False)
fi.head()
```

Visualize Full Tree

As before, we can look at the decision tree on the data. This time, we have to limit the maximum depth otherwise the tree will be too large and cannot be converted and displayed as an image.

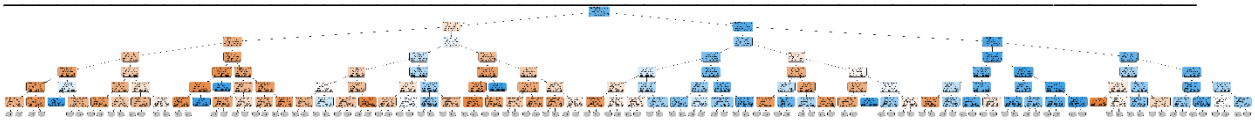
```
# Save tree as dot file
export_graphviz(tree, 'tree_real_data.dot', rounded = True,
                feature_names = features, max_depth = 6,
                class_names = ['poor health', 'good health'], filled = True)

# Convert to png
call(['dot', '-Tpng', 'tree_real_data.dot', '-o', 'tree_real_data.png', '-Gdpi=200'])

# Visualize
Image(filename='tree_real_data.png')
```

Output

GOVT. AKHTAR NAWAZ KHAN (SHAHEED)
DEGREE COLLEGE KTS, HARIPUR
DEPARTMENT OF COMPUTER SCIENCE
Near Chandni Chowk, Sector No. 1, Khalabat Township, Tehsil & District Haripur
Phone: **0995-621918**, Email: **gdckts@gmail.com**



Random Forest

Now we can move on to a more powerful model, the random forest. This takes the idea of a single decision tree, and creates an ensemble model out of hundreds or thousands of trees to reduce the variance. Each tree is trained on a random set of the observations, and for each split of a node, only a subset of the features are used for making a split. When making predictions, the random forest averages the predictions for each of the individual decision trees for each data point in order to arrive at a final classification.

Creating and training a random forest in extremely easy in Scikit-Learn. The cell below is all you need.

```
from sklearn.ensemble import RandomForestClassifier

# Create the model with 100 trees
model = RandomForestClassifier(n_estimators=100,
                               random_state=RSEED,
                               max_features = 'sqrt',
                               n_jobs=-1, verbose = 1)

# Fit on training data
model.fit(train, train_labels)
```

We can see how many nodes there are for each tree on average and the maximum depth of each tree. There were 100 trees in the forest.

```
n_nodes = []
max_depths = []

for ind_tree in model.estimators_:
    n_nodes.append(ind_tree.tree_.node_count)
    max_depths.append(ind_tree.tree_.max_depth)
```

```
print(f'Average number of nodes {int(np.mean(n_nodes))}')
print(f'Average maximum depth {int(np.mean(max_depths))}')
```

We see that each decision tree in the forest has many nodes and is extremely deep. However, even though each individual decision tree may overfit to a particular subset of the training data, the idea is that the overall random forest should have a reduced variance.

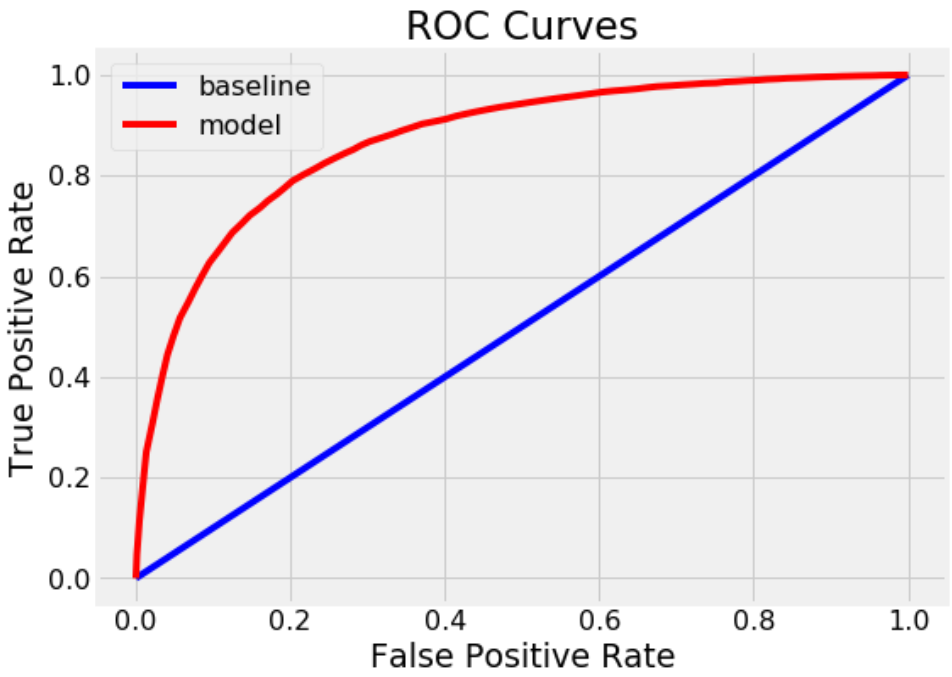
Random Forest Results

```
train_rf_predictions = model.predict(train)
train_rf_probs = model.predict_proba(train)[:, 1]

rf_predictions = model.predict(test)
rf_probs = model.predict_proba(test)[:, 1]
```

```
evaluate_model(rf_predictions, rf_probs, train_rf_predictions, train_rf_probs)
```

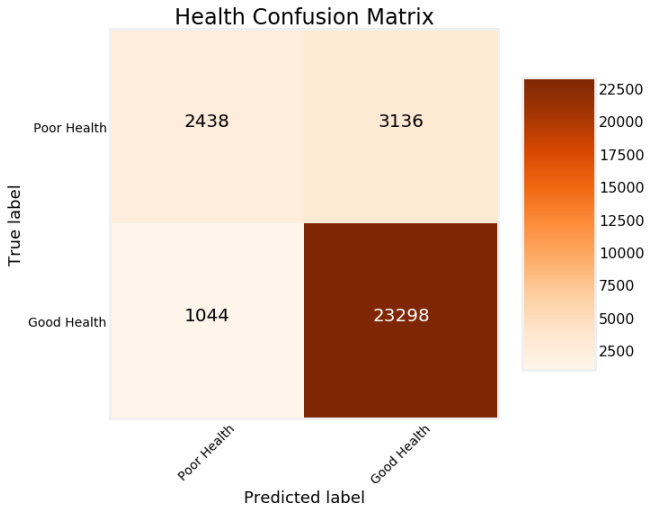
Results



The model still achieves perfect measures on the training data, but this time, the testing scores are much better. If we compare the ROC AUC, we see that the random forest does significantly better than a single decision tree.

```
cm = confusion_matrix(test_labels, rf_predictions)
plot_confusion_matrix(cm, classes = ['Poor Health', 'Good Health'],
                      title = 'Health Confusion Matrix')
```

Results



Compared to the single decision tree, the model has fewer false postives although more false negatives. Overall, the random forest does significantly better than a single decision tree. This is what we expected!

```
fi_model = pd.DataFrame({'feature': features,
                         'importance': model.feature_importances_}).\
                          sort_values('importance', ascending = False)
fi_model.head(10)
```

Random Forest Optimization through Random Search

```
from sklearn.model_selection import RandomizedSearchCV

# Hyperparameter grid
param_grid = {
```

```
  'n_estimators': np.linspace(10, 200).astype(int),
  'max_depth': [None] + list(np.linspace(3, 20).astype(int)),
  'max_features': ['auto', 'sqrt', None] + list(np.arange(0.5, 1, 0.1)),
  'max_leaf_nodes': [None] + list(np.linspace(10, 50, 500).astype(int)),
  'min_samples_split': [2, 5, 10],
  'bootstrap': [True, False]
}

# Estimator for use in random search
estimator = RandomForestClassifier(random_state = RSEED)

# Create the random search model
rs = RandomizedSearchCV(estimator, param_grid, n_jobs = -1,
                        scoring = 'roc_auc', cv = 3,
                        n_iter = 10, verbose = 1, random_state=RSEED)

# Fit
rs.fit(train, train_labels)
```

```
rs.best_params_
```

Use Best Model

Now we can take the best model (it has already been trained) and evaluate it. Hopefully it does better than the stock Random Forest.

```
best_model = rs.best_estimator_
```

```
train_rf_predictions = best_model.predict(train)
train_rf_probs = best_model.predict_proba(train)[:, 1]

rf_predictions = best_model.predict(test)
rf_probs = best_model.predict_proba(test)[:, 1]
```

```
n_nodes = []
max_depths = []
```

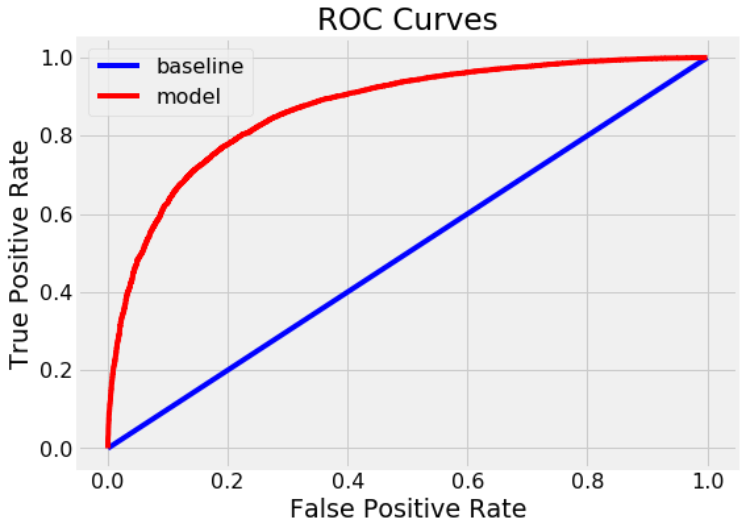
```
for ind_tree in best_model.estimators_:
  n_nodes.append(ind_tree.tree_.node_count)
  max_depths.append(ind_tree.tree_.max_depth)

print(f'Average number of nodes {int(np.mean(n_nodes))}')
print(f'Average maximum depth {int(np.mean(max_depths))}')
```

The best maximum depth is not unlimited as we see above! This indicates that restricting the maximum depth of the individual decision trees can improve the cross validation performance of the random forest.

```
evaluate_model(rf_predictions, rf_probs, train_rf_predictions, train_rf_probs)
```

Results



The optimized model achieves around the same performance as the default model. More random search iterations could improve performance, or it's possible that we are close the limit of what the random forest can achieve for this problem.

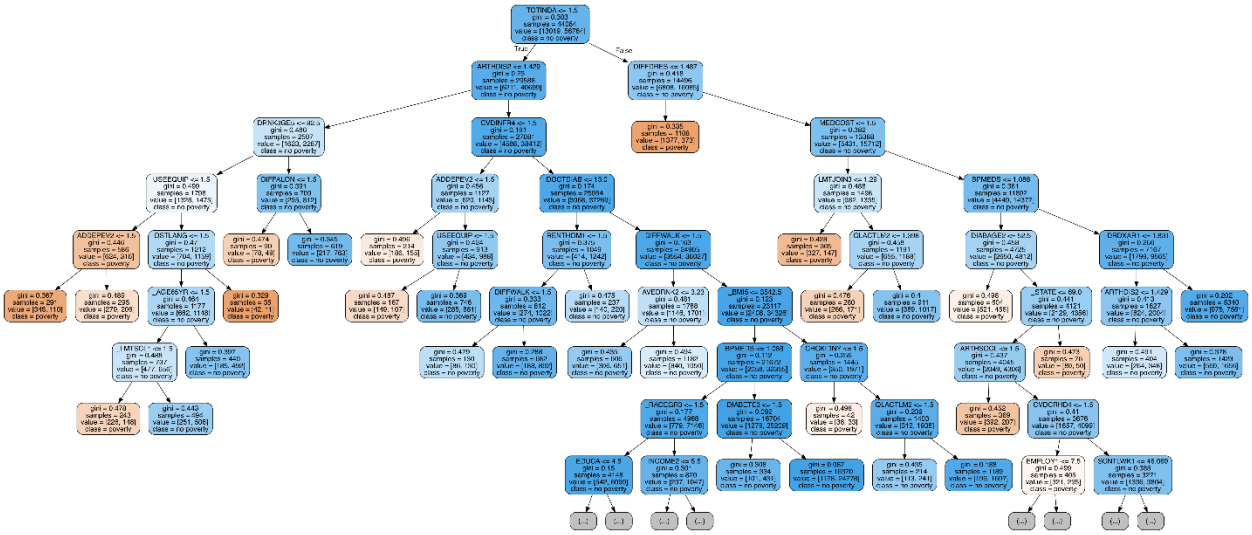
```
estimator = best_model.estimators_[1]
```

GOVT. AKHTAR NAWAZ KHAN (SHAHEED)
DEGREE COLLEGE KTS, HARIPUR
DEPARTMENT OF COMPUTER SCIENCE
Near Chandni Chowk, Sector No. 1, Khalabat Township, Tehsil & District Haripur
Phone: **0995-621918**, Email: **gdckts@gmail.com**

```
# Export a tree from the forest
export_graphviz(estimator, 'tree_from_optimized_forest.dot', rounded = True,
                feature_names=train.columns, max_depth = 8,
                class_names = ['poverty', 'no poverty'], filled = True)
```

```
call(['dot', '-Tpng', 'tree_from_optimized_forest.dot', '-o',
'tree_from_optimized_forest.png', '-Gdpi=200'])
Image('tree_from_optimized_forest.png')
```

Output



GOVT. AKHTAR NAWAZ KHAN (SHAHEED)
DEGREE COLLEGE KTS, HARIPUR
DEPARTMENT OF COMPUTER SCIENCE
Near Chandni Chowk, Sector No. 1, Khalabat Township, Tehsil & District Haripur
Phone: **0995-621918**, Email: gdckts@gmail.com

Lab Home Tasks:

Implementation Task:

Use a different dataset (e.g., from UCI Machine Learning Repository) to implement the ANN social media and fake jobs posting. Document your findings with metrics like accuracy, precision, recall, and ROC-AUC.

Research Task:

Write a short report (1000 words) discussing at least three real-world applications of Random Forest Classifiers and why they are well-suited for these applications.

Marking Rubric:

Criteria	Excellent (4 Marks)	Good (3 Marks)	Satisfactory (2 Marks)	Needs Improvement (1 Mark)	Marks
Implementation of Random Forest	Fully functional implementation with clear code structure and proper parameter tuning.	Functional implementation with minimal parameter tuning.	Partially functional implementation with minor errors.	Implementation incomplete or contains major errors.	/4
Visualization and Insights	Clear and detailed visualizations with insightful interpretations.	Good visualizations with some interpretations provided.	Basic visualizations with limited or unclear insights.	Poor or missing visualizations and interpretations.	/4
Performance Evaluation	Comprehensive evaluation using multiple metrics with well-documented comparisons.	Good evaluation using basic metrics with partial comparison.	Limited evaluation with few metrics and vague comparisons.	Poor or missing evaluation of model performance.	/4
Home Task Submission	Complete, well-	Complete report with	Partially complete	Report incomplete or	/4

	documented report with insightful analysis.	basic analysis and documentation .	report with minimal analysis.	lacks analysis and documentation .	
Code Quality and Documentation	Code is well-structured, efficient, and thoroughly documented.	Code is structured and partially documented.	Code is functional but poorly structured or documented.	Code is disorganized and lacks documentation .	/4

Good Luck!