# Tree Ensembles and Random Forests - Lab

## Introduction

In this lab, we'll create some popular tree ensemble models such as a bag of trees and random forest to predict a person's salary based on information about them.

## Objectives

In this lab you will:

- Train a random forest model using `scikit-learn`
- Access, visualize, and interpret feature importances from an ensemble model

## Import data

In this lab, you'll use personal attributes to predict whether people make more than 50k/year. The dataset was extracted from the census bureau database. The goal is to use this dataset to try and draw conclusions regarding what drives salaries. More specifically, the target variable is categorical (> 50k and <= 50 k). Let's create a classification tree!

To get started, run the cell below to import everything we'll need for this lab.

```
import pandas as pd
import numpy as np
np.random.seed(0)
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
```

Our dataset is stored in the file `'salaries_final.csv'` .

In the cell below, import the dataset from this file and store it in a DataFrame. Be sure to set the `index_col` parameter to `0` . Then, display the `.head()` of the DataFrame to ensure that everything loaded correctly.

```
# Import the data
salaries = pd.read_csv('salaries_final.csv', index_col=0)
salaries.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

| | Age | Education | Occupation | Relationship | Race | Sex | Target |
|---|---|---|---|---|---|---|---|
| 0 | 39 | Bachelors | Adm-clerical | Not-in-family | White | Male | <=50K |
| 1 | 50 | Bachelors | Exec-managerial | Husband | White | Male | <=50K |
| 2 | 38 | HS-grad | Handlers-cleaners | Not-in-family | White | Male | <=50K |
| 3 | 53 | 11th | Handlers-cleaners | Husband | Black | Male | <=50K |
| 4 | 28 | Bachelors | Prof-specialty | Wife | Black | Female | <=50K |

In total, there are 6 predictors, and one outcome variable, the salary, `Target - <= 50k` and `>50k` .

The 6 predictors are:

- `Age` : continuous
- `Education` : Categorical. Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool
- `Occupation` : Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces
- `Relationship` : Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried
- `Race` : White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black
- `Sex` : Female, Male

First, we'll need to store our `'Target'` column in a separate variable and drop it from the dataset.

Do this in the cell below.

```
# Split the outcome and predictor variables
target = salaries['Target']
```

```
salaries = salaries.drop("Target", axis=1)
```

In the cell below, examine the data type of each column:

```
salaries.dtypes
```

```
Age              int64
Education       object
Occupation      object
Relationship    object
Race            object
Sex             object
dtype: object
```

Great. `'Age'` is numeric, as it should be. Now we're ready to create some dummy columns and deal with our categorical variables.

In the cell below, use Pandas to create dummy columns for each of categorical variables. If you're unsure of how to do this, check out the documentation.

```
# Create dummy variables
data = pd.get_dummies(salaries)
data.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

| | Age | Education_10th | Education_11th | Education_12th | Education_1st-4th |
|---|---|---|---|---|---|
| **0** | 39 | 0 | 0 | 0 | 0 |
| **1** | 50 | 0 | 0 | 0 | 0 |

| | Age | Education_10th | Education_11th | Education_12th | Education_1st-4th |
|---|---|---|---|---|---|
| **2** | 38 | 0 | 0 | 0 | 0 |
| **3** | 53 | 0 | 1 | 0 | 0 |
| **4** | 28 | 0 | 0 | 0 | 0 |

5 rows × 45 columns

Now, split `data` and `target` into 75/25 training and test sets. Set the `random_state` to 123.

```
data_train, data_test, target_train, target_test = train_test_split(data, target,
                                                                   test_size = 0.25
```

## Build a "regular" tree as a baseline

We'll begin by fitting a regular decision tree classifier, so that we have something to compare our ensemble methods to.

### Build the tree

In the cell below, instantiate and fit a decision tree classifier. Set the `criterion` to `'gini'`, and a `max_depth` of `5`. Then, fit the tree to the training data and labels.

```
# Instantiate and fit a DecisionTreeClassifier
tree_clf = DecisionTreeClassifier(criterion='gini', max_depth=5)
tree_clf.fit(data_train, target_train)
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

## Feature importance

Let's quickly examine how important each feature ended up being in our decision tree model. Check the `feature_importances_` attribute of the trained model to see what it displays.

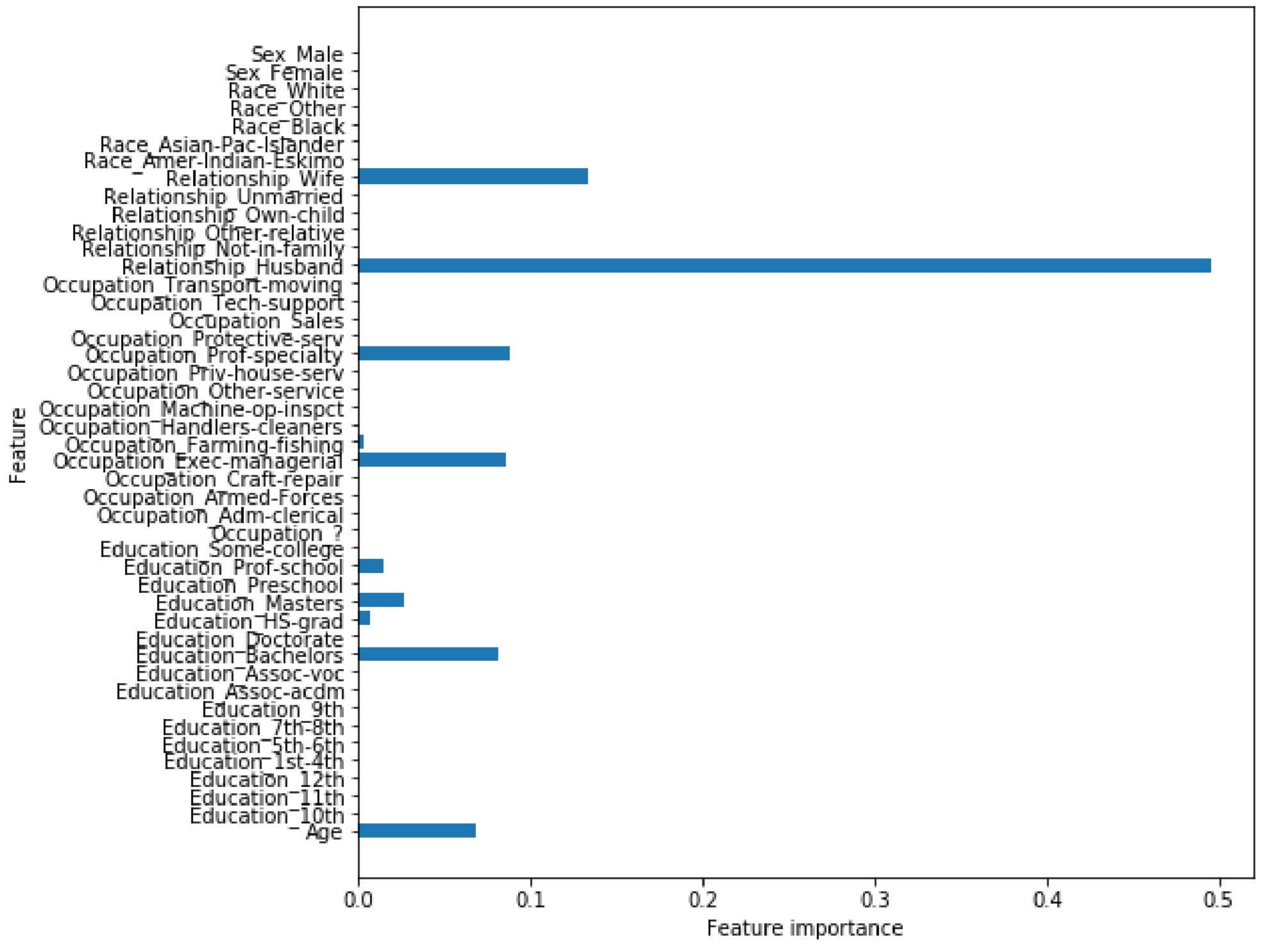
```
# Feature importance
tree_clf.feature_importances_
```

```
array([0.06761352, 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.08071446, 0.        , 0.006495  , 0.02596604, 0.        ,
       0.01482269, 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.0853097 , 0.00311049, 0.        , 0.        ,
       0.        , 0.        , 0.0879446 , 0.        , 0.        ,
       0.        , 0.        , 0.4950878 , 0.        , 0.        ,
       0.        , 0.        , 0.1329357 , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ])
```

That matrix isn't very helpful, but a visualization of the data it contains could be. Run the cell below to plot a visualization of the feature importances for this model.

```
def plot_feature_importances(model):
    n_features = data_train.shape[1]
    plt.figure(figsize=(8,8))
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), data_train.columns.values)
    plt.xlabel('Feature importance')
    plt.ylabel('Feature')

plot_feature_importances(tree_clf)
```



## Model performance

Next, let's see how well our model performed on the test data.

In the cell below:

- Use the model to generate predictions on the test set
- Print out a `confusion_matrix` of the test set predictions
- Print out a `classification_report` of the test set predictions

```
# Test set predictions
pred = tree_clf.predict(data_test)

# Confusion matrix and classification report
print(confusion_matrix(target_test, pred))
print(classification_report(target_test, pred))
```

```
[[5762  403]
 [1059  917]]
              precision    recall  f1-score   support
```

```
       <=50K       0.84      0.93      0.89      6165
        >50K       0.69      0.46      0.56      1976

    accuracy                           0.82      8141
   macro avg       0.77      0.70      0.72      8141
weighted avg       0.81      0.82      0.81      8141
```

Now, let's check the model's accuracy. Run the cell below to display the test set accuracy of the model.

```
print("Testing Accuracy for Decision Tree Classifier: {:.4}%".format(accuracy_score(
```

```
Testing Accuracy for Decision Tree Classifier: 82.04%
```

## Bagged trees

The first ensemble approach we'll try is a bag of trees. This will make use of ***Bagging***, along with a number of decision tree classifier models.

Now, let's instantiate a `BaggingClassifier` . First, initialize a `DecisionTreeClassifier` and set the same parameters that we did above for `criterion` and `max_depth` . Also set the `n_estimators` parameter for our `BaggingClassifier` to `20` .

```
# Instantiate a BaggingClassifier
bagged_tree =  BaggingClassifier(DecisionTreeClassifier(criterion='gini', max_depth=
                                 n_estimators=20)
```

Great! Now, fit it to our training data.

```
# Fit to the training data
bagged_tree.fit(data_train, target_train)
```

```
BaggingClassifier(base_estimator=DecisionTreeClassifier(class_weight=None,
                                                        criterion='gini',
                                                        max_depth=5,
                                                        max_features=None,
                                                        max_leaf_nodes=None,

min_impurity_decrease=0.0,
                                                        min_impurity_split=None,
                                                        min_samples_leaf=1,
                                                        min_samples_split=2,

min_weight_fraction_leaf=0.0,
                                                        presort=False,
                                                        random_state=None,
                                                        splitter='best'),
                  bootstrap=True, bootstrap_features=False, max_features=1.0,
                  max_samples=1.0, n_estimators=20, n_jobs=None,
                  oob_score=False, random_state=None, verbose=0,
                  warm_start=False)
```

Checking the accuracy of a model is such a common task that all (supervised learning) models have a `.score()` method that wraps the `accuracy_score()` helper function we've been using. All we have to do is pass it a dataset and the corresponding labels and it will return the accuracy score for those data/labels.

Let's use it to get the training accuracy of our model. In the cell below, call the `.score()` method on our bagging model and pass in our training data and training labels as parameters.

```
# Training accuracy score
bagged_tree.score(data_train, target_train)
```

```
0.8277231777231777
```

Now, let's check the accuracy score that really matters -- our testing accuracy. This time, pass in our testing data and labels to see how the model did.

```
# Test accuracy score
bagged_tree.score(data_test, target_test)
```

```
0.8221348728657413
```

## Random forests

Another popular ensemble method is the ***Random Forest***. Let's fit a random forest classifier next and see how it measures up compared to all the others.

### Fit a random forests model

In the cell below, instantiate and fit a `RandomForestClassifier`, and set the number estimators to `100` and the max depth to `5`. Then, fit the model to our training data.

```
# Instantiate and fit a RandomForestClassifier
forest = RandomForestClassifier(n_estimators=100, max_depth= 5)
forest.fit(data_train, target_train)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=5, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

Now, let's check the training and testing accuracy of the model using its `.score()` method:

```
# Training accuracy score
forest.score(data_train, target_train)
```

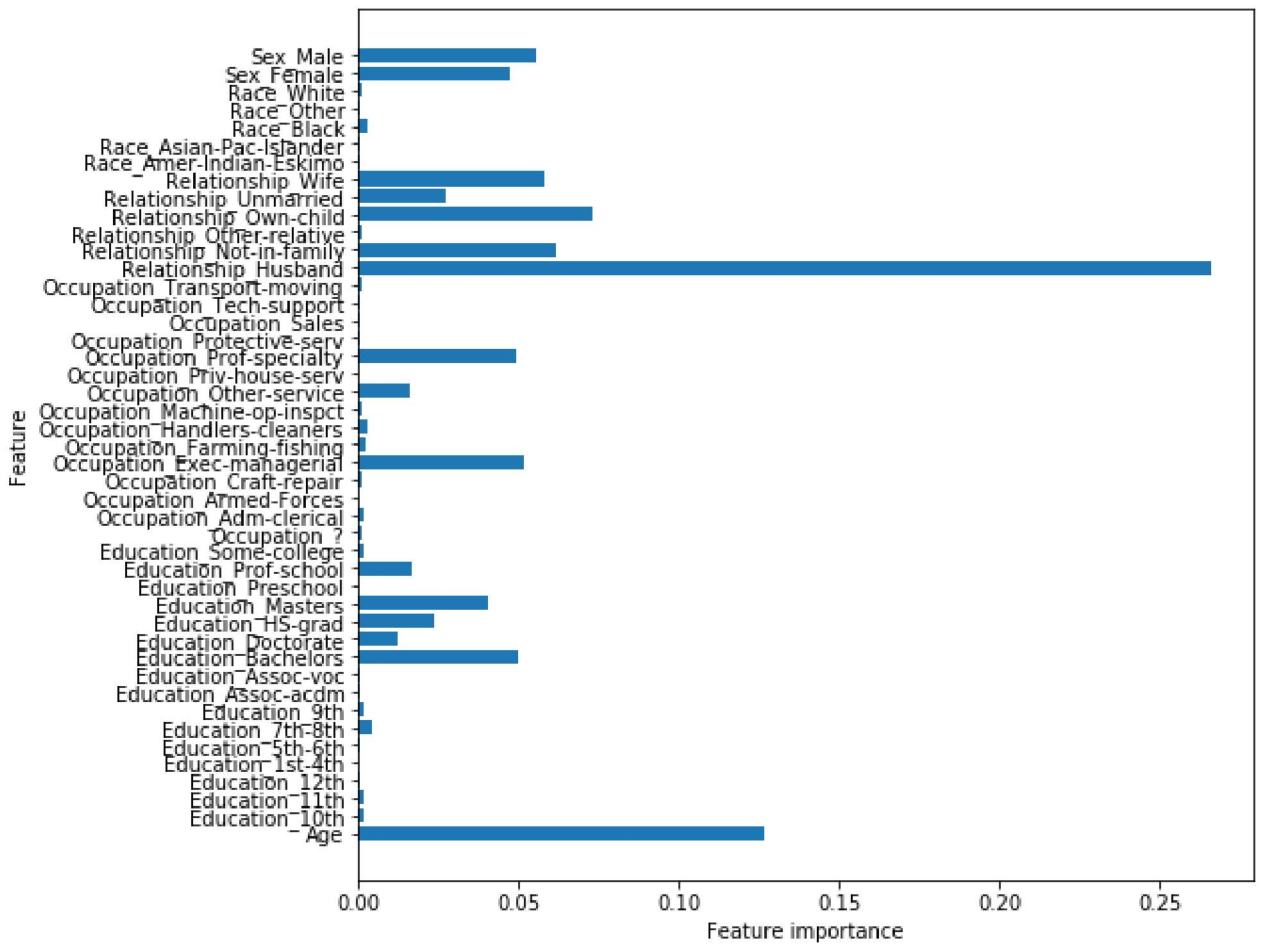
```
0.8054054054054054
```

```
# Test accuracy score
forest.score(data_test, target_test)
```

```
0.8042009581132539
```

### Feature importance

```
plot_feature_importances(forest)
```



Note: "relationship" represents what this individual is relative to others. For example an individual could be a Husband. Each entry only has one relationship, so it is a bit of a weird attribute.

Also note that more features show up. This is a pretty typical result.

## Look at the trees in your forest

Let's create a forest with some small trees. You'll learn how to access trees in your forest!

In the cell below, create another `RandomForestClassifier` . Set the number of estimators to 5, the `max_features` to 10, and the `max_depth` to 2.

```
# Instantiate and fit a RandomForestClassifier
forest_2 = RandomForestClassifier(n_estimators = 5, max_features= 10, max_depth= 2)
forest_2.fit(data_train, target_train)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=2, max_features=10, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=5,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

Making `max_features` smaller will lead to very different trees in your forest! The trees in your forest are stored in the `.estimators_` attribute.

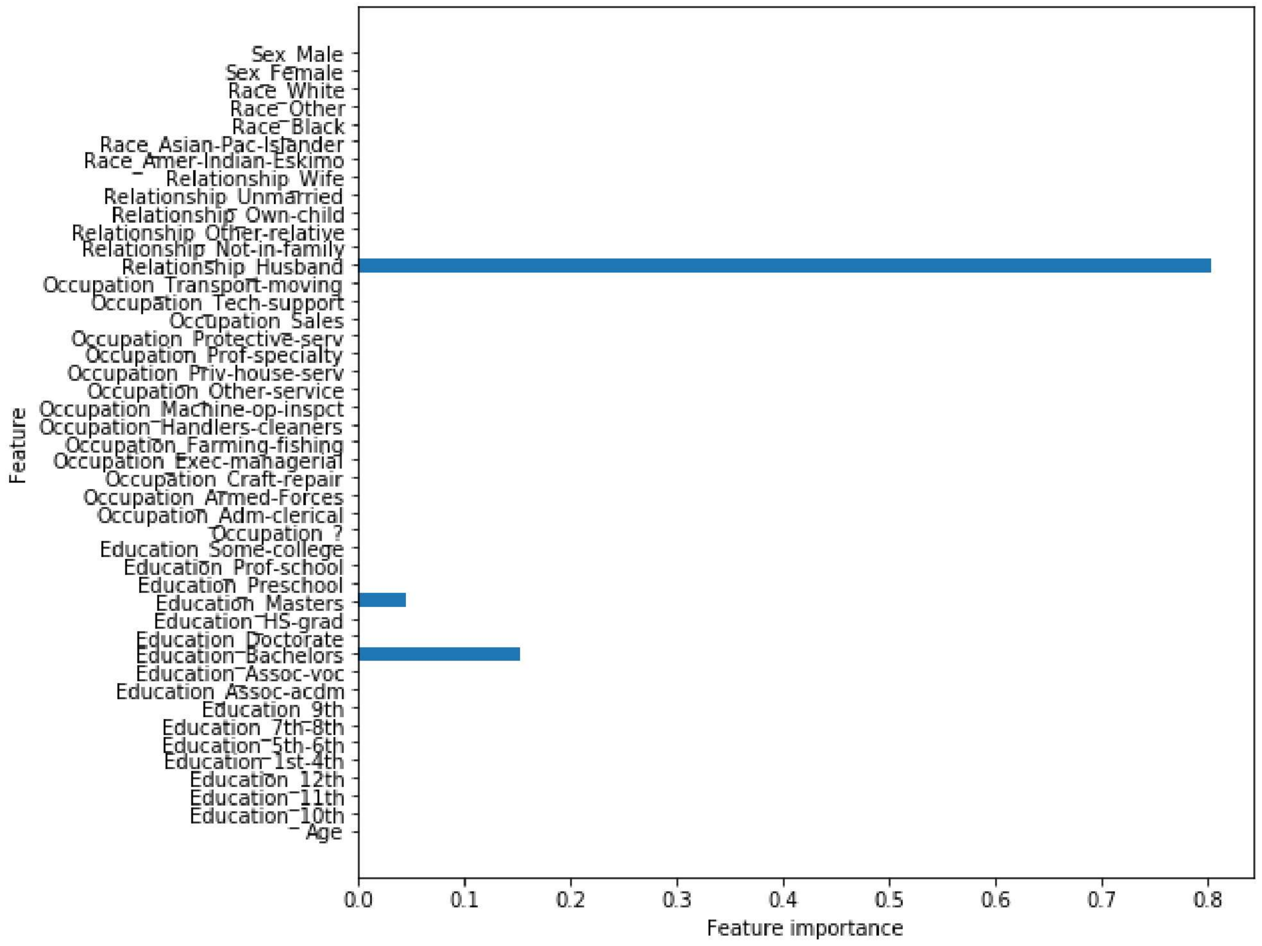
In the cell below, get the first tree from `forest_2.estimators_` and store it in `rf_tree_1`

```
# First tree from forest_2
rf_tree_1 = forest_2.estimators_[0]
```

Now, we can reuse our `plot_feature_importances()` function to visualize which features this tree was given to use duing subspace sampling.

In the cell below, call `plot_feature_importances()` on `rf_tree_1` .

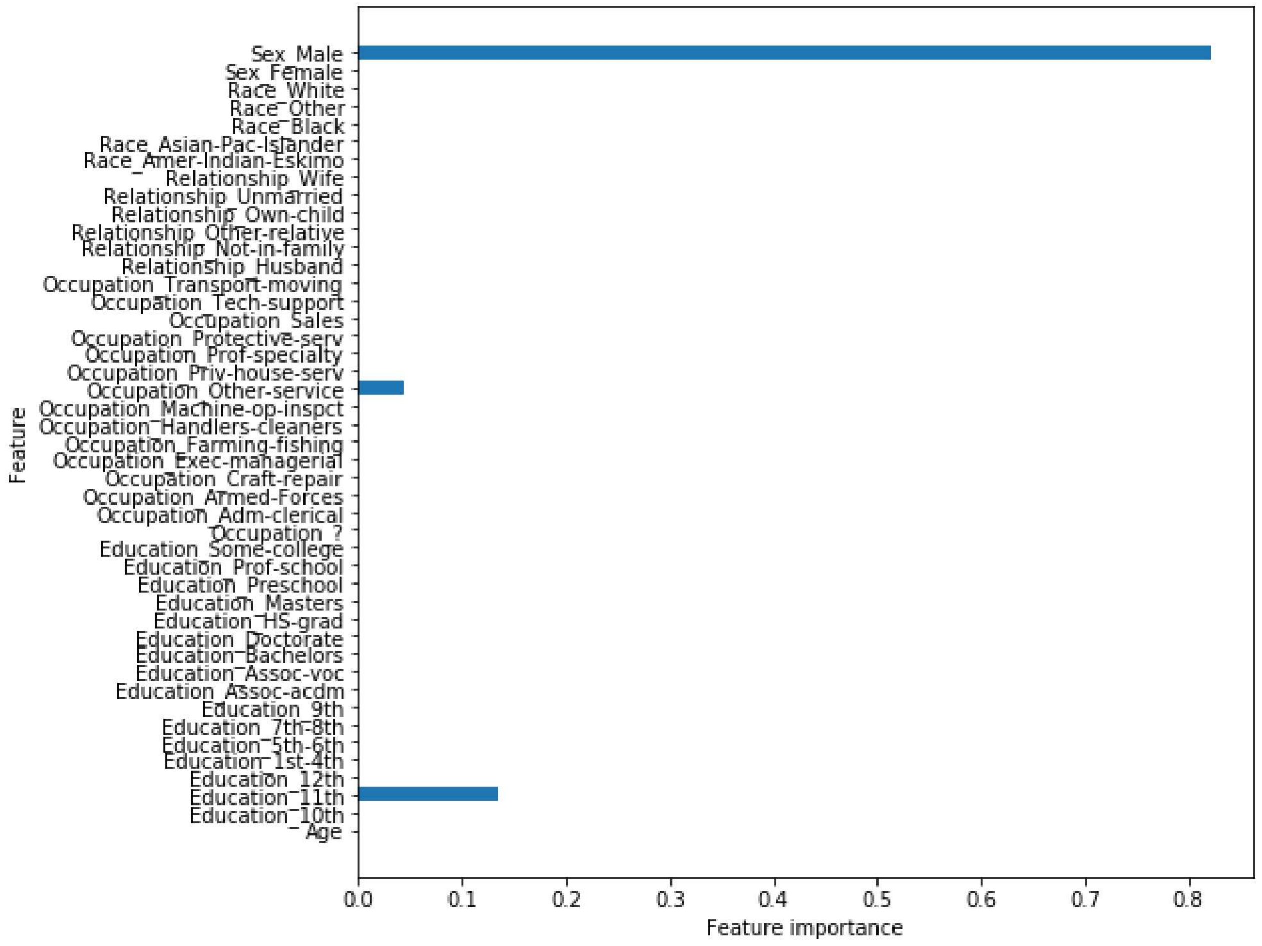
```
# Feature importance
plot_feature_importances(rf_tree_1)
```



Now, grab the second tree and store it in `rf_tree_2`, and then pass it to `plot_feature_importances()` in the following cell so we can compare which features were most useful to each.

```
# Second tree from forest_2
rf_tree_2 = forest_2.estimators_[1]
```

```
# Feature importance
plot_feature_importances(rf_tree_2)
```



We can see by comparing the two plots that the two trees we examined from our random forest look at different attributes, and have wildly different feature importances!

## Summary

In this lab, we got some practice creating a few different tree ensemble methods. We also learned how to visualize feature importances, and compared individual trees from a random forest to see if we could notice the differences in the features they were trained on.

## Contributors 3

**mike-kane** Mike Kane

**LoreDirick** Lore Dirick

**sumedh10** Sumedh Panchadhar

## Languages

● **Jupyter Notebook** 100.0%