Student Info:

Name	ID
Niloufar Eslah	30223324

Table of Contents

Introduction

The aim of the project is extensive research on the intersection between marital stability, income levels, and emotional well-being with sophisticated machine learning techniques to categorize couples into high and low-income groups based on a dataset reflecting a broad range of socioeconomic factors. Unlike traditional studies that often utilize econometric models to identify correlations, this research adopts a novel approach by applying various classification methods—Decision Trees, Random Forest, Logistic Regression, and Support Vector Machines (SVM)—not to find causal links but to determine specific characteristics of couples that predict their income level and relationship quality.

The research rigorously evaluates the predictive power of each model in terms of accuracy, sensitivity, and precision. Initial findings suggest that while the Decision Tree provides a foundational understanding, the Random Forest model improves predictive metrics significantly, demonstrating robustness against overfitting. The SVM, utilizing its kernel trick and cross-validation, shows competitive results in high-dimensional spaces or with imbalanced data, while Logistic Regression offers high precision, indicating its utility in scenarios where false positives carry a high cost.

By integrating discussions on how variations in income levels can impact individual happiness and sadness, along with exploring the emotional dynamics within relationships and their connection to educational attainment, the study seeks to provide a deeper understanding of the factors that influence these aspects of personal and shared human experience. Through this analysis, the study aims to discern patterns associating certain characteristics of couples with their income brackets, offering insights that could aid in socioeconomic planning and policy formulation.

Name	Type	Values	Description
CatInc (output)	Categorical	Low, High	Categorical Income
Duration	Continuous	Integer greater than 0	Relationship Duration
EDUCATION	Ordinal	High School(assign number less than 12), Bachelor(assign number 12), Master, etc	Education Level
MARIT	Nominal	Married = 1, Divorced = 3, separated = 4	Marital Status

more_edu	Nominal	Him=2, Her=0, Equal=1	Whether male has more education
olderman	Nominal	Him=2, Her=0, Equal=1	Whether the man is older
earnmore	Nominal	Him=2, Her=0, Equal=1	Who earns more
AGE	Continuous	From 18 to 99	Age of respondent
Race	Nominal	White = 1, Black = 2, other = 3, hispanic = 5	Race of respondent
HHSize	Discrete	From 1 to 10	Household Size
Region	Nominal	Northeast = 1, Midwest = 2, South = 3, West = 4	Region of residence
Wealth	Ordinal	owned or being bought by you or someone in your household = 1, rented for cash = 2, occupied without payment of cash rent = 3	ownership status
HHchild	Discrete	From 0 to 5	Number of children in household
Work	Nominal	working - as a paid employee = 1, working - self-employed = 2, not working - on temporary layoff from a job =3, not working (more than 3)	Work status
Net	Binary	0 and 1	Access to the internet

In [65]:
```
# import needed libraries
import numpy as np
import pandas as pd
import seaborn as sns
import re
from tabulate import tabulate
import matplotlib.pyplot as plt
import seaborn as sb

%matplotlib inline
```

In [66]:
```
# Loading data
data = pd.read_csv("data.csv")
```

The project poses some questions about this experiment:

1. DWhat features classify couples in the class of earning higher than 60k or less?

2. What are the most important features that help to define the income of couples?

2. What method will classify better?

Overview

In [67]:
```
# take an overview look at the data
data.head()
```

Out[67]:

	CASEID_NEW	PPAGECAT	PPAGE	PPEDUCAT	PPEDUC	PPETHM	PPGENDER	PPHOUSEHOLDSIZ
0	23286	(02) 25-34	28	(4) bachelor's degree or higher	(13) masters degree	(1) white, non-hispanic	(2) female	
1	29584	(05) 55-64	58	(4) bachelor's degree or higher	(13) masters degree	(1) white, non-hispanic	(1) male	
2	34341	(02) 25-34	34	(4) bachelor's degree or higher	(12) bachelors degree	(1) white, non-hispanic	(2) female	
3	43381	(03) 35-44	38	(4) bachelor's degree or higher	(13) masters degree	(1) white, non-hispanic	(1) male	
4	44486	(04) 45-54	48	(3) some college	(10) some college, no degree	(2) black, non-hispanic	(2) female	

5 rows × 39 columns

In [4]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1841 entries, 0 to 1840
Data columns (total 39 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   CASEID_NEW             1841 non-null   int64  
 1   PPAGECAT               1841 non-null   object 
 2   PPAGE                  1841 non-null   int64  
 3   PPEDUCAT               1841 non-null   object 
 4   PPEDUC                 1841 non-null   object 
 5   PPETHM                 1841 non-null   object 
 6   PPGENDER               1841 non-null   object 
 7   PPHOUSEHOLDSIZE        1841 non-null   int64  
 8   PPINCIMP               1841 non-null   object 
 9   PPMARIT                1841 non-null   object 
 10  PPREG4                 1841 non-null   object 
 11  PPRENT                 1841 non-null   object 
 12  CHILDREN_IN_HH         1841 non-null   int64  
 13  PPWORK                 1841 non-null   object 
 14  PPNET                  1841 non-null   object 
 15  PPPARTYID3             1841 non-null   object 
 16  PAPRELIGION            1841 non-null   object 
 17  Q4                     1841 non-null   object 
 18  Q6B                    1841 non-null   object 
 19  Q7B                    1841 non-null   object 
 20  Q9                     1841 non-null   int64  
 21  Q10                    1841 non-null   object 
 22  Q17A                   1841 non-null   object 
 23  Q21B                   1841 non-null   int64  
 24  Q21D                   1841 non-null   int64  
 25  Q23                    1841 non-null   object 
 26  Q27                    1841 non-null   object 
 27  Q31_1                  1841 non-null   object 
 28  Q31_2                  1841 non-null   object 
 29  Q31_3                  1841 non-null   object 
 30  Q31_4                  1841 non-null   object 
 31  Q31_5                  1841 non-null   object 
 32  Q31_6                  1841 non-null   object 
 33  Q31_7                  1841 non-null   object 
 34  Q31_8                  1841 non-null   object 
 35  Q31_9                  1841 non-null   object 
 36  Q32                    1841 non-null   object 
 37  Q34                    1841 non-null   object 
 38  HOW_LONG_RELATIONSHIP  1841 non-null   float64
dtypes: float64(1), int64(7), object(31)
memory usage: 561.1+ KB
```

Data Cleaning

In [8]:
```
# Convert character columns to numeric if they contain numeric values within parentheses
for col in data.columns:
    if data[col].dtype == 'object':  # Assuming we only need to process object (string) t
        data[col] = data[col].apply(lambda x: float(re.sub("[()]", "", re.findall("\(\d+\

# Make a copy of the original data to retain couples who have different genders.
dif = data.copy()

# Creating a new feature for couples with different genders
dif['dif_gender'] = np.where(dif['PPGENDER'] != dif['Q4'], 1, 0)
```

```
# Filter out rows where couples have the same gender
dif = dif[dif['dif_gender'] == 1]
```

```
/var/folders/sd/myz3__r100b49gpr37d61_gw0000gn/T/ipykernel_72787/396917812.py:45: FutureWa
rning: DataFrame.applymap has been deprecated. Use DataFrame.map instead.
  all_numeric = dif.applymap(np.isreal).all().all()
```

- We have dropped the couples who has the same gender.
- Goal is to create new comparative variables.

In [9]:
```
# New comparative variables: Does man have a higher education?
dif['more_edu'] = 0  # Initialize the column
# Set conditions based on the information in the data
dif.loc[(dif['PPGENDER'] == 2) & (dif['PPEDUC'] > dif['Q10']), 'more_edu'] = 2
dif.loc[(dif['PPGENDER'] == 2) & (dif['PPEDUC'] == dif['Q10']), 'more_edu'] = 1
dif.loc[(dif['PPGENDER'] == 1) & (dif['PPEDUC'] < dif['Q10']), 'more_edu'] = 2
dif.loc[(dif['PPGENDER'] == 1) & (dif['PPEDUC'] == dif['Q10']), 'more_edu'] = 1

# New comparative variables: Is the man older than the woman?
dif['older_man'] = 0
dif.loc[(dif['PPGENDER'] == 2) & (dif['PPAGE'] > dif['Q9']), 'older_man'] = 2
dif.loc[(dif['PPGENDER'] == 2) & (dif['PPAGE'] == dif['Q9']), 'older_man'] = 1
dif.loc[(dif['PPGENDER'] == 1) & (dif['PPAGE'] < dif['Q9']), 'older_man'] = 2
dif.loc[(dif['PPGENDER'] == 1) & (dif['PPAGE'] == dif['Q9']), 'older_man'] = 1

# New comparative variables: Do they have the same religion?
dif['same_religion'] = np.where(dif['PAPRELIGION'] == dif['Q7B'], 1, 0)

# New comparative variables: Who earns more?
dif['earn_more'] = 0
dif.loc[(dif['PPGENDER'] == 2) & (dif['Q23'] == 3), 'earn_more'] = 2
dif.loc[(dif['PPGENDER'] == 1) & (dif['Q23'] == 1), 'earn_more'] = 2
dif.loc[(dif['PPGENDER'] == 2) & (dif['Q23'] == 2), 'earn_more'] = 1

# Convert character columns to numeric if they represent numeric values
for col in dif.columns:
    if dif[col].dtype == 'object':
        dif[col] = pd.to_numeric(dif[col], errors='coerce')

# Check for successful conversion
all_numeric = dif.applymap(np.isreal).all().all()
```

```
/var/folders/sd/myz3__r100b49gpr37d61_gw0000gn/T/ipykernel_72787/1407083818.py:31: FutureW
arning: DataFrame.applymap has been deprecated. Use DataFrame.map instead.
  all_numeric = dif.applymap(np.isreal).all().all()
```

- I created new variables which is related to the couples.
- Now I have to count the observation in each one of this variable and more importantly the output variable to make sure that I have a balance data.

In [15]:
```
dif.more_edu.value_counts()
```

Out[15]:
```
more_edu
2    607
0    582
1    550
Name: count, dtype: int64
```

In [16]:
```
dif.older_man.value_counts()
```

Out[16]:
```
older_man
0    1146
2     386
1     207
Name: count, dtype: int64
```

In [17]:
```
dif.same_religion.value_counts()
```

Out[17]:
```
same_religion
1    1065
0     674
Name: count, dtype: int64
```

In [18]:
```
dif.earn_more.value_counts()
```

Out[18]:
```
earn_more
2    1196
0     460
1      83
Name: count, dtype: int64
```

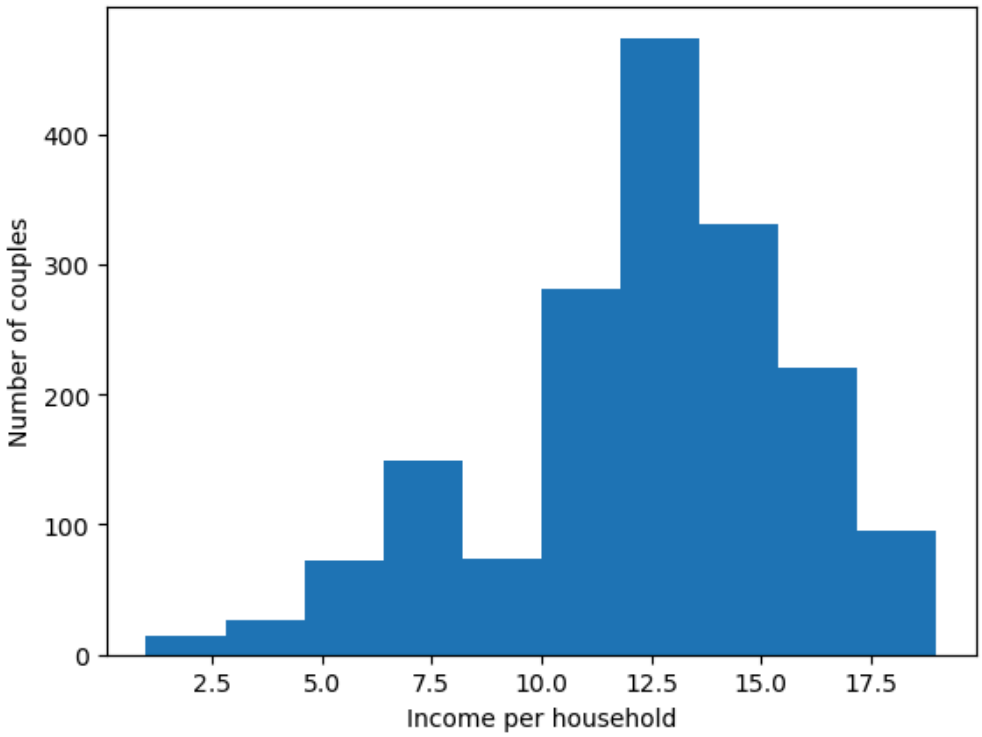
- Discarding the spaces from the entries of the dataset, for easier access.

In [19]:
```
dif.PPINCIMP.value_counts()
```

Out[19]:
```
PPINCIMP
13.0    251
12.0    223
14.0    169
11.0    169
15.0    162
16.0    149
10.0    112
8.0      76
9.0      74
7.0      73
17.0     72
19.0     55
6.0      44
18.0     40
5.0      29
4.0      16
3.0      11
2.0      10
1.0       4
Name: count, dtype: int64
```

- Different definitions or specifications of income may categorize it as high or low. In our dataset, we must establish a threshold that serves as our point for dividing the income variable into high and low income categories. To accomplish this, we require some statistical information about this variable:

In [21]:
```
# Plotting the histogram
plt.hist(dif['PPINCIMP'], bins=10)  # You can adjust the number of bins as needed
plt.xlabel("Income per household")
plt.ylabel("Number of couples")
plt.title("")  # Empty title as specified in the R code
plt.show()
```



- As observed in Figure, the distribution of income appears nearly symmetric. To determine a more appropriate threshold for categorizing income into high and low, we now need to delve deeper into the statistical characteristics of this variable.

In [22]:
```
# Creating a boxplot
plt.boxplot(dif['PPINCIMP'])
plt.xlabel("Income per household")
plt.ylabel("Values")  # You can customize this label if needed
plt.title("")  # Leaving the title empty as in the R code
plt.show()
```

- Additionally, the box plot indicates that there are no significant outliers in the income data. This allows us to calculate key points of the income distribution, which will assist us in identifying the appropriate threshold.
- The exact value of statistical featute is:

In [25]:
```
# Calculate mean and median with NA values removed
mean_value = dif['PPINCIMP'].mean(skipna=True)
median_value = dif['PPINCIMP'].median(skipna=True)

# Create a DataFrame to hold the values
summary_table = pd.DataFrame({
    'Statistic': ['Mean', 'Median'],
    'Value': [mean_value, median_value]
})

# Use tabulate to create a markdown table, similar to kable in R
print(tabulate(summary_table, headers='keys', tablefmt='pipe', showindex=False))
```

```
| Statistic   |   Value |
|:------------|--------:|
| Mean        | 12.3071 |
| Median      | 13      |
```

- Based on the data analyzed, we have determined that the optimal threshold for the output variable is 60,000 USD per year. This threshold effectively divides the dataset into two groups: couples with high income, who earn more than 60,000 USD annually, and couples with low income, who earn less than 60,000 USD annually. This categorization allows us to further explore and compare the characteristics and dynamics within these income brackets.
- Now We should creat the output binary value:

In [27]:
```
# Create a new column 'CatInc' initialized to 0
dif['CatInc'] = 0
```

```
# Assign 1 to 'CatInc' where 'PPINCIMP' is greater than 12
dif.loc[dif['PPINCIMP'] > 12, 'CatInc'] = 1

# Use value_counts to get the frequency table
frequency_table = dif['CatInc'].value_counts()
print(frequency_table)
```

```
CatInc
1    898
0    841
Name: count, dtype: int64
```

- The result shows the new binary variable which is balance.

In [34]:
```
# Select data needed
variables = dif[[
    "CatInc", "HOW_LONG_RELATIONSHIP", "PPEDUC", "PPMARIT", "more_edu",
    "older_man", "earn_more", "PPAGE", "PPETHM", "PPHOUSEHOLDSIZE", "PPREG4",
    "PPRENT", "CHILDREN_IN_HH", "PPWORK", "PPNET", "PPPARTYID3", "Q17A", "Q34"
]]

# Define new names for each column
variables.columns = [
   "CatInc", "Duration", "EDUCATION", "MARIT", "more_edu",
    "olderman", "earnmore", "AGE", "Race", "HHSize", "Region",
    "Wealth", "HHchild", "Work", "Net", "Politic", "Timesmar", "Qualrelation"
]
```

Data Exploration

In [35]:
```
# a quick look on some statistics about the data
dif.describe()
```

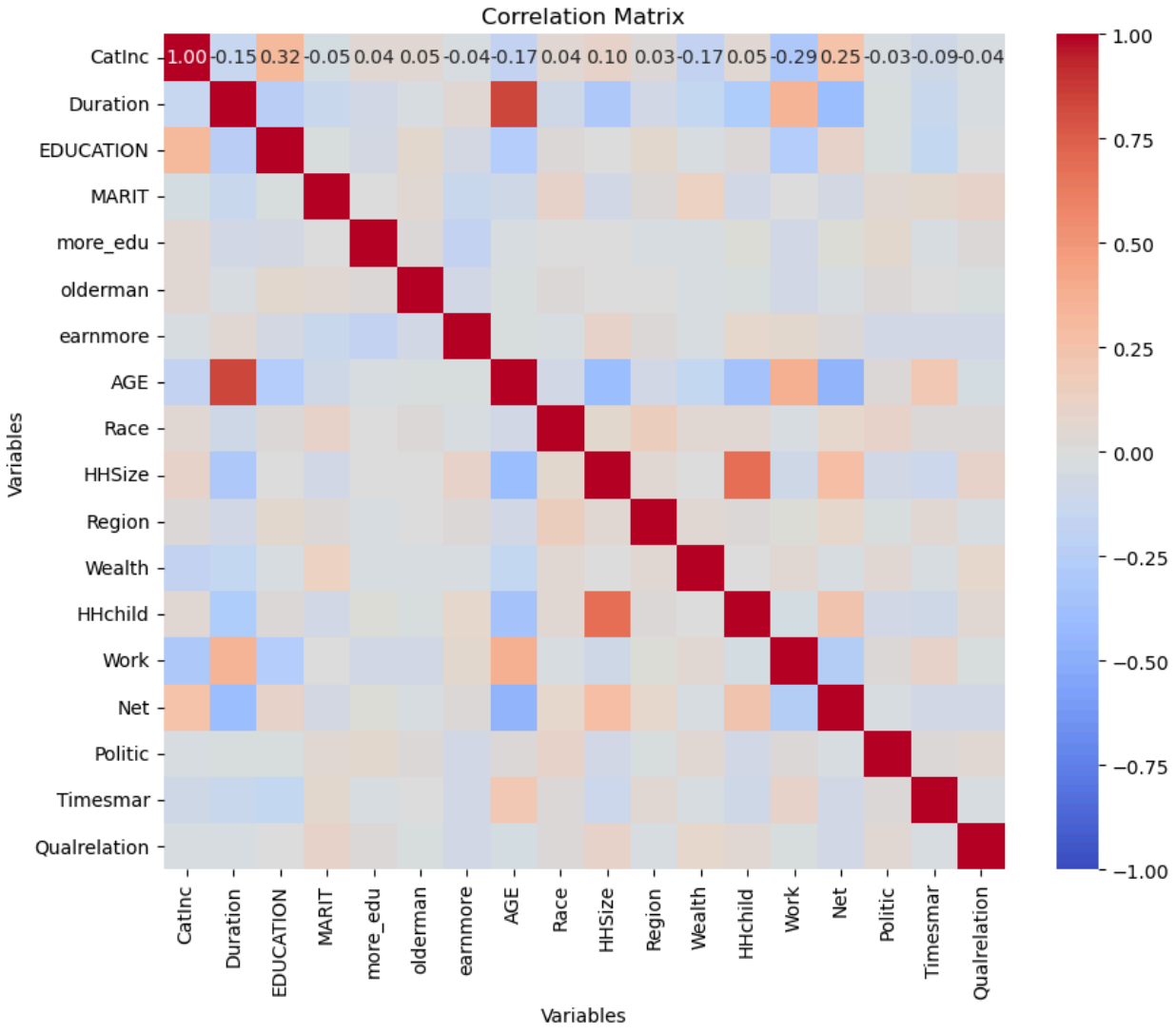
Out[35]:

	CASEID_NEW	PPAGECAT	PPAGE	PPEDUCAT	PPEDUC	PPETHM	PPGENDER
count	1.739000e+03	1739.00000	1739.000000	1739.000000	1739.000000	1739.000000	1739.000000
mean	2.361907e+06	3.86889	48.235193	2.772283	10.109833	1.565267	1.465785
std	1.329115e+06	1.56892	15.828570	1.041659	2.133898	1.155201	0.498971
min	2.958400e+04	1.00000	19.000000	1.000000	2.000000	1.000000	1.000000
25%	1.211696e+06	3.00000	35.000000	2.000000	9.000000	1.000000	1.000000
50%	2.347261e+06	4.00000	46.000000	3.000000	10.000000	1.000000	1.000000
75%	3.487001e+06	5.00000	59.000000	4.000000	12.000000	1.000000	2.000000
max	4.628251e+06	7.00000	94.000000	4.000000	14.000000	5.000000	2.000000

8 rows × 45 columns

In [37]:
```
# Calculate the correlation matrix
cor_matrix = variables.corr()

# Visualize the correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cor_matrix, annot=True, fmt=".2f", cmap="coolwarm", center=0, vmin=-1, vmax=1
plt.xticks(rotation=90)  # Rotate x-axis labels for better readability
```

```
plt.yticks(rotation=0)
plt.title("Correlation Matrix")
plt.xlabel("Variables")
plt.ylabel("Variables")
plt.show()
```



- Based on the results depicted in the image, we observe strong correlations among some of the features, as well as notable correlations between the output variable and several features. With these insights, we are now prepared to proceed with running our model. This will allow us to determine if we can effectively classify the output variable based on the features identified. This step is crucial for verifying the predictive power of our model and understanding the potential interactions within our data.

Experimental Process

In [85]:
```
# Estimators for classification models
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
```

```
# Utilities for model evaluation and hyperparameter tuning
from sklearn.model_selection import GridSearchCV, cross_val_score, cross_validate, train_

# Functions to compute model performance metrics
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_sco

# Classes for feature scaling and creating model pipelines
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

# Decision tree visualization tools
from sklearn.tree import export_graphviz, plot_tree
import graphviz

# IPython display function to render objects
from IPython.display import display

# Control the display of warnings
import warnings
warnings.filterwarnings("ignore")
```

Evaluation Methodology:

The data has been split into training and testing parts of the features and the label with a test size of 20% and with a random state to get the same randomness with the next runs. This happened by using the train_test_split function.

Cross-validation has been applied between the models to select the most suitable ones, We have done that using the cross_validate function with 5 folds splitting. And outputs the train and test score of the model.

All of This has been done that with a cleaned state of the data, also with the scaled and encoded version of it.

Due to the fact that there is an imbalance in the classes of classification. If this has been fixed, it would help the model to learn better from the various classes and to not be biassed towards one over another.

One way to fight this issue is to generate new samples in the classes which are under-represented (minority class). The most naive strategy is to generate new samples by randomly sampling with replacement of the currently available samples. This is called Oversampling, it is a technique used to modify unequal data classes to create balanced data sets.

And that has been applied using RandomOverSampler class from the imblearn library, to generate the new resampled data.

Metrics used for Evaluation:

We have used the accuracy metric for the evaluation of the models. We can describe the accuracy metric as the ratio between the number of correct predictions and the total number of predictions:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Also, for binary classification, accuracy can also be calculated in terms of the confusion matrix terminology:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Where TP = True Positives, TN = True Negatives, FP = False Positives, and FN = False Negatives.

Recall (also known as sensitivity) measures the proportion of actual positives that are correctly identified by the model. It can be considered as the model's ability to detect positive instances. The formula for recall is:

$$\text{Recall (Sensitivity)} = \frac{TP}{TP + FN}$$

Here, (TP) stands for True Positives, which are the cases where the model correctly predicts the positive class, and (FN) stands for False Negatives, which are the cases where the model incorrectly predicts the negative class for an observation that is actually positive.

Precision measures the proportion of positive identifications that were actually correct. It reflects the model's accuracy in predicting positive instances. The formula for precision is:

$$\text{Precision} = \frac{TP}{TP + FP}$$

In this formula, (FP) represents False Positives, where the model incorrectly predicts the positive class for an observation that is actually negative. Precision is especially important in situations where the cost of a false positive is high.

Both recall and precision are critical metrics in situations where the balance between false positives and false negatives is a crucial consideration. For example, in medical diagnostics, missing a true disease case (a false negative) could be more dangerous than falsely identifying a disease (a false positive). Thus, a high recall would be very desirable in such a scenario. On the other hand, in spam detection, a false positive (marking an important email as spam) could be more problematic, thereby making precision a more important metric.

First method: Decision Tree

- Initially, we will develop a comprehensive model using all available features to assess its accuracy without any modifications. This unpruned model will serve as our baseline to understand the full potential of our dataset.
- Subsequently, we will implement Cross-Validation to determine an optimal parameter for pruning the tree. This step is crucial to identify the level of model complexity that achieves the best balance between accuracy and generalizability.
- A fully grown tree often shows reduced variance in the training error rate but can suffer from a high error rate on the test data, indicating that the model may be overfitting. Overfitting occurs when a model is too closely aligned with the training data, failing to generalize well to new, unseen data. Therefore, pruning is an essential step to reduce overfitting and enhance the model's performance on external data sets, ensuring it remains robust and reliable across different scenarios.

Fully grown Tree:

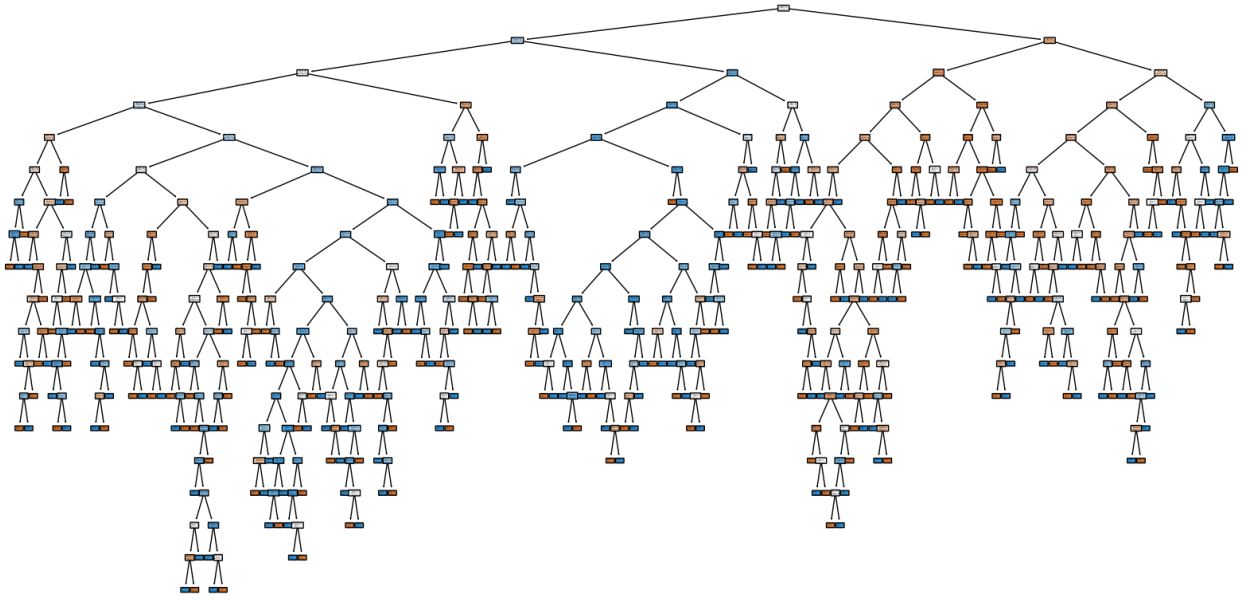
First we need to split the data to the train and test in order to fit the model on the train and then evaluate the fitted model on the test. To do this we used 60% of the data for train and the rest of it for test. After this step we can run the fully grwon tree and here is the result of evaluation of the tree:

In [39]:
```
# Set seed for reproducibility
np.random.seed(2024)

# Assuming 'variables' is a DataFrame
# Split the data into training and testing sets
variables_train, variables_test = train_test_split(variables, test_size=0.4, random_state
```

In [48]:
```
# Prepare the data
X_train = variables_train.drop('CatInc', axis=1)
y_train = variables_train['CatInc']

# Build the decision tree
tree = DecisionTreeClassifier(random_state=0, min_impurity_decrease=0)
tree.fit(X_train, y_train)
```

Out[48]:
```
DecisionTreeClassifier(min_impurity_decrease=0, random_state=0)
```

In [49]:
```
# Build and train the tree, if not already done
tree = DecisionTreeClassifier(random_state=0, min_impurity_decrease=0)
tree.fit(X_train, y_train)

# Visualize the tree using matplotlib
plt.figure(figsize=(20,10))  # Set the size of the figure, adjust to your needs
plot_tree(tree, feature_names=X_train.columns, class_names=['0', '1'], filled=True, round
plt.show()
```



- The result based on calculating the confusion matrix is:

In [51]:
```
# Make predictions on the test set
t_pred = tree.predict(variables_test.drop('CatInc', axis=1))

# Create confusion matrix
```

```
confMat = confusion_matrix(variables_test['CatInc'], t_pred)

# Calculate sensitivity (recall) and precision for each class
sensitivity_per_class = recall_score(variables_test['CatInc'], t_pred, average=None)
precision_per_class = precision_score(variables_test['CatInc'], t_pred, average=None)

# Calculate macro-average sensitivity and precision
sensitivity = sensitivity_per_class.mean()
precision = precision_per_class.mean()

# Calculate overall accuracy
accuracy = accuracy_score(variables_test['CatInc'], t_pred)

# Combine into one DataFrame
results = pd.DataFrame({
    'Accuracy': [accuracy],
    'Sensitivity': [sensitivity],
    'Precision': [precision]
})

# Display the results as a table
display(results.style.set_caption("Model Performance Metrics"))
```

Model Performance Metrics

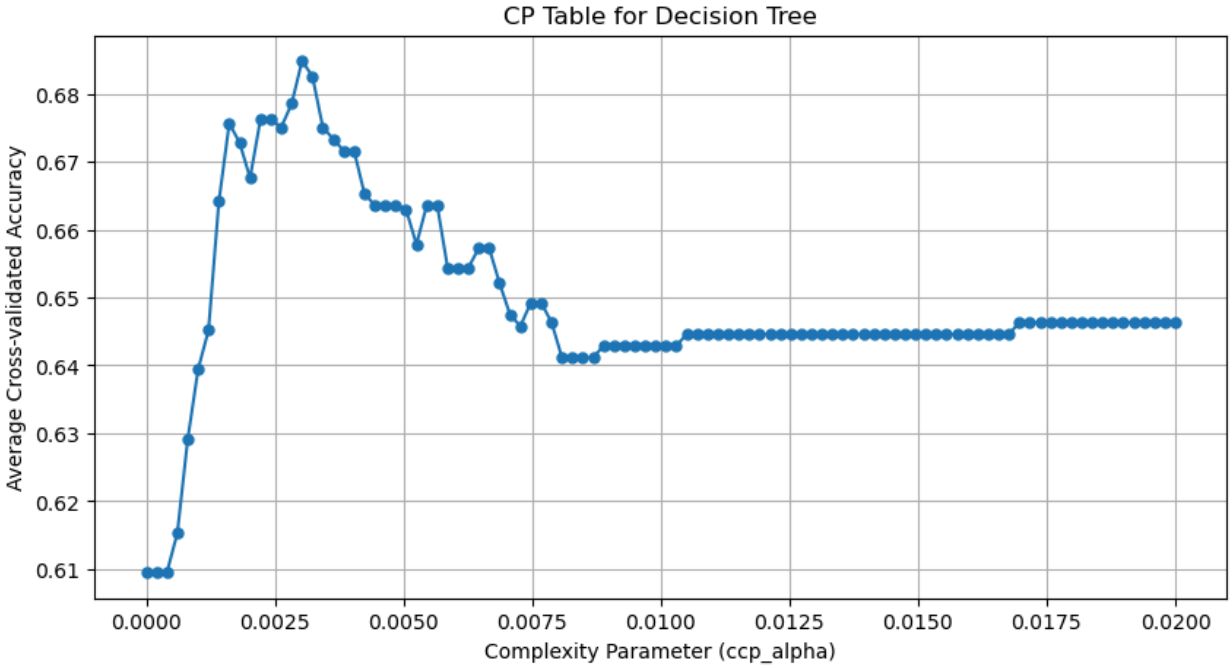
	Accuracy	Sensitivity	Precision
0	0.622126	0.618818	0.619506

- Accuracy (0.622126): This metric tells us the proportion of the total number of predictions that were correct. An accuracy of approximately 62.21% indicates that about 62 out of 100 predictions made by the model are correct. While this is better than a random guess, which would be 50% in a balanced binary classification problem, it still suggests there is significant room for improvement.
- Sensitivity (0.618818): Sensitivity, or recall, measures the proportion of actual positives that are correctly identified. In this case, the model has a sensitivity of around 61.88%, meaning it correctly identifies 61.88% of the positive instances. This is relatively close to the accuracy, suggesting a balanced performance in detecting positive cases compared to the overall accuracy.
- Precision (0.619506): Precision is the ratio of correctly predicted positive observations to the total predicted positives. The model's precision of approximately 61.95% implies that when it predicts a positive result, it is correct 61.95% of the time.
- The values of sensitivity and precision are quite similar, which can indicate that the model is treating both classes (positive and negative) with roughly equal performance, assuming a balanced dataset.
- Overall, these metrics suggest that the model has a moderate performance and that there could be a balance between the classes in terms of prediction. However, the moderate values also imply that the model may benefit from further tuning. This could include feature engineering, model parameter tuning, or gathering more or different quality of data for training.

Pruned Tree:

In [54]:
```
# Prepare the data
X = variables.drop('CatInc', axis=1)
y = variables['CatInc']
```

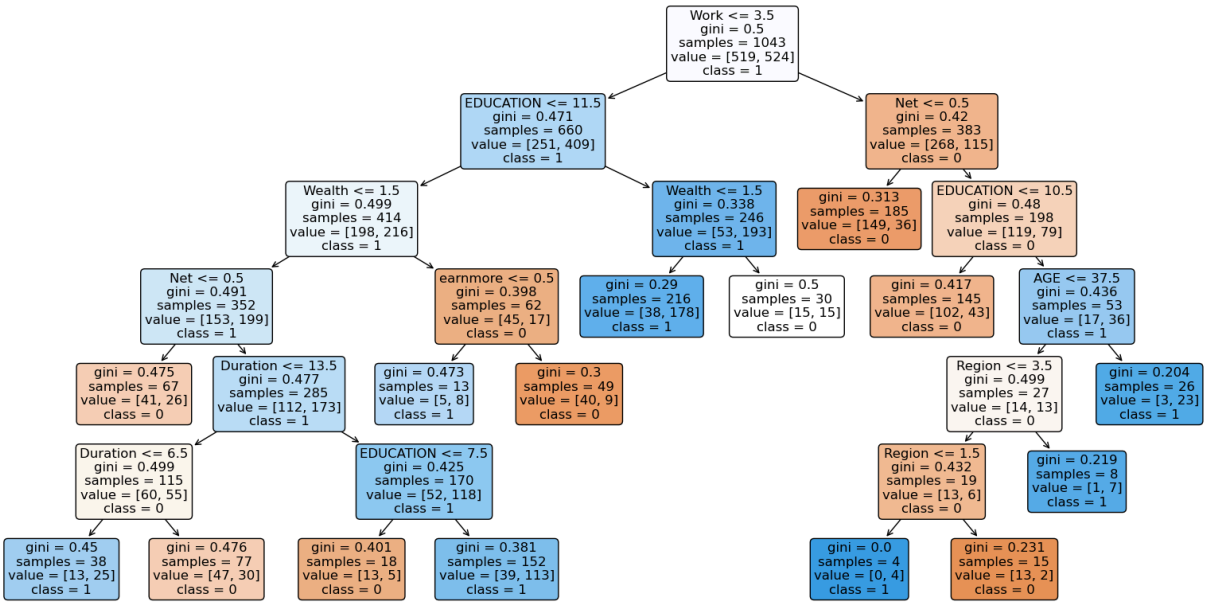
```
# List to store the average cross-validated score for different values of 'ccp_alpha'
alpha_values = np.linspace(0, 0.02, 100)  # Adjust the range and density as needed
cv_scores = []

for ccp_alpha in alpha_values:
    tree = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    scores = cross_val_score(tree, X, y, cv=5)  # 5-fold cross-validation
    cv_scores.append(np.mean(scores))

# Now plot the CP table equivalent
plt.figure(figsize=(10, 5))
plt.plot(alpha_values, cv_scores, marker='o', linestyle='-', markersize=5)
plt.xlabel('Complexity Parameter (ccp_alpha)')
plt.ylabel('Average Cross-validated Accuracy')
plt.title('CP Table for Decision Tree')
plt.grid(True)
plt.show()
```



In [55]:
```
# Assuming 'variables' is a DataFrame with the target variable 'CatInc'
X = variables.drop(columns='CatInc')
y = variables['CatInc']

# Initialize range for ccp_alpha values
ccp_alpha_values = np.linspace(0, 0.02, 100)  # Adjust the range based on your requiremen

# Store the average cross-validation scores for each value of ccp_alpha
cv_scores = []

# Perform cross-validation and store the results
for ccp_alpha in ccp_alpha_values:
    tree = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    scores = cross_val_score(tree, X, y, cv=5)  # 5-fold cross-validation
    cv_scores.append(np.mean(scores))

# Find the ccp_alpha value that maximizes the cross-validation score
best_ccp_alpha = ccp_alpha_values[np.argmax(cv_scores)]

# Print the best ccp_alpha
print("Best ccp_alpha:", best_ccp_alpha)
```

```
Best ccp_alpha: 0.0030303030303030303
```

In [57]:
```
# Prepare the data
X_train = variables_train.drop('CatInc', axis=1)
y_train = variables_train['CatInc']

# Create and train the pruned tree
tree_pruned = DecisionTreeClassifier(random_state=0, ccp_alpha=best_ccp_alpha)
tree_pruned.fit(X_train, y_train)

# Plot the pruned tree
plt.figure(figsize=(20, 10))  # Adjust the figure size as needed
plot_tree(tree_pruned, feature_names=X_train.columns, class_names=['0', '1'], filled=True
plt.show()
```



- The root node starts with a split based on the "Wealth" attribute, with a Gini impurity of 0.499 and 4199 samples. This indicates that the initial dataset before the split is quite mixed regarding the classes (high-income and low-income couples).
- The first split divides the data based on the "Wealth" attribute being less than or equal to 1.5. The left child of this split has a higher Gini impurity (0.491), suggesting a more even mix of classes, while the right child has a lower Gini impurity (0.338), indicating a better separation of the classes.
- The tree continues to split the data on various attributes, such as "Net", "Duration", "Education", "Work", "earnmore", "Age", and "Region". Each split is made at a value that best separates the samples into the two classes.
- At each leaf node, you can see the number of samples that fall into each class after all the splits. For example, one of the leaf nodes at the bottom left, which is reached when "Wealth" is less than or equal to 1.5, "Net" is less than or equal to 0.5, and "Duration" is less than or equal to 6.5, has 38 samples with the value [13, 25], meaning there are 13 samples of class 0 and 25 samples of class 1.
- The "value" array at each node shows the distribution of the samples across the two classes at that point in the tree. The "class" label indicates the predominant class in that node.
- The Gini impurity is a measure of how often a randomly chosen element would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset. A Gini impurity of 0

would indicate that the node is perfectly pure, containing samples from only one class.

In [59]:
```
# Prepare the data
X_test = variables_test.drop('CatInc', axis=1)
y_test = variables_test['CatInc']

# Make predictions with the pruned tree
t_pred_pruned = tree_pruned.predict(X_test)

# Create a confusion matrix
confMat_pruned = confusion_matrix(y_test, t_pred_pruned)

# Calculate sensitivity (recall) and precision for each class
sensitivity_per_class_pruned = recall_score(y_test, t_pred_pruned, average=None)
precision_per_class_pruned = precision_score(y_test, t_pred_pruned, average=None)

# Calculate macro-average sensitivity and precision
sensitivity_pruned = np.mean(sensitivity_per_class_pruned)
precision_pruned = np.mean(precision_per_class_pruned)

# Calculate overall accuracy
accuracy_pruned = accuracy_score(y_test, t_pred_pruned)

# Combine into one DataFrame
results_pruned = pd.DataFrame({
    'Accuracy': [accuracy_pruned],
    'Sensitivity': [sensitivity_pruned],
    'Precision': [precision_pruned]
})

# Print the results as a table
print(results_pruned)
```

```
   Accuracy  Sensitivity  Precision
0   0.66954     0.672651   0.672256
```

- Accuracy: The pruned tree's accuracy improved by roughly 4.74 percentage points, indicating a better overall performance.
- Sensitivity: The pruned tree shows a higher sensitivity, with an approximate increase of 5.38 percentage points, meaning it's better at correctly identifying true positive cases.
- Precision: The pruned tree's precision is also higher by about 5.28 percentage points, which means a higher proportion of positive identifications was actually correct.
- The improvements in all three metrics suggest that pruning the decision tree has likely reduced overfitting, where the fully grown tree was perhaps too complex and fit the noise in the training data. Pruning helps in creating a simpler model that generalizes better to unseen data, which seems to be the case here, as indicated by the increase in all evaluation metrics.
- Pruning can remove extraneous branches that don't contribute much to predicting power but capture noise, hence the better performance of the pruned tree on the test data. This demonstrates the importance of tuning model complexity to balance bias and variance, ultimately aiming for a model that accurately captures the underlying patterns without being misled by random fluctuations in the training dataset.

Random Forest

As we previously outlined, we have segmented all couples into two classes based on income: high income and low income. In this session, we aim to employ the Random Forest algorithm to determine if it can enhance our classification results.

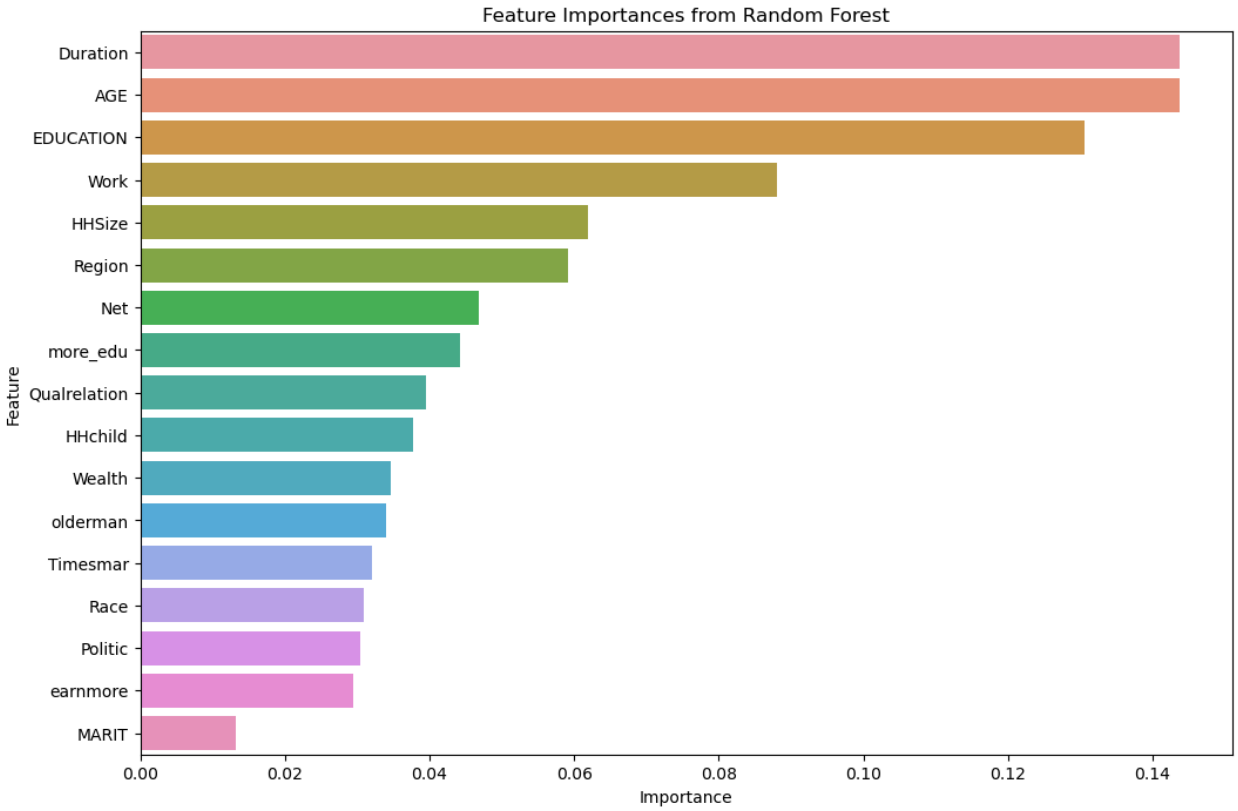
Random Forest is a robust ensemble learning method that operates by building multiple decision trees during the training phase. The process is systematic yet randomized in several aspects to ensure diversity and to reduce overfitting. Initially, the algorithm randomly selects samples from the dataset to form different subsets. Each tree is then trained on one of these subsets.

During the training of each tree, the algorithm randomly selects features at each node and determines the best splits based on these features. This random selection of features adds an additional layer of randomness that is distinct from the random selection of data samples.

Once all the trees are constructed, Random Forest aggregates the outcomes of these individual trees. This aggregation, often through averaging the predictions or a majority vote, helps to stabilize the predictions by reducing variance and avoiding the overfitting typically seen in single decision trees. The combined result aims to provide a more accurate and reliable prediction than any individual tree could achieve on its own.

In [61]:
```
# Prepare the data
X_train = variables_train.drop('CatInc', axis=1)
y_train = variables_train['CatInc'].astype('category')

# Set up cross-validation
cv = KFold(n_splits=5, shuffle=True, random_state=123)

# Train the model with cross-validation
rf_model = RandomForestClassifier(n_estimators=500, random_state=123)

# Calculate cross-validated scores
cv_scores = cross_val_score(rf_model, X_train, y_train, cv=cv)

# Train the model on the full training data
rf_model.fit(X_train, y_train)

# Feature importances can be extracted from the trained model
feature_importances = rf_model.feature_importances_
```

In [63]:
```
# Convert the importances into a DataFrame
features = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances})

# Sort the features by importance
features_sorted = features.sort_values('Importance', ascending=False)

# Plotting
plt.figure(figsize=(12, 8))
sns.barplot(x='Importance', y='Feature', data=features_sorted)

plt.title('Feature Importances from Random Forest')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.show()
```



- Duration: This feature has the highest importance score, which means it is the most influential variable in predicting the target outcome according to the random forest model. Changes in the duration variable are likely to have the most significant impact on the model's predictions.
- AGE and EDUCATION: The second and third most important features are age and education, respectively. These features also play a critical role in the model's predictions but to a lesser extent compared to the duration variable.
- Other Features: The features listed further down, such as Work, HHSize, Region, and so forth, contribute to the model's predictions as well, but their impact is progressively smaller as we move down the chart.
- Least Important Features: The features at the bottom of the chart, such as MARIT, earnmore, and Politic, are the least important in this model's predictions. This means that these variables have the least influence on the outcome variable according to the model.
- It's important to note that feature importance scores do not necessarily imply causation; they merely reflect the features' predictive power within the context of this specific model and dataset. A feature with a low importance score is not necessarily unimportant in general, but rather it may not be useful for making predictions in the context of this specific model or it may be redundant with other features.

In [64]:
```
# Assuming 'variables' is a DataFrame with the target variable 'CatInc'
X = variables.drop(columns='CatInc')
y = variables['CatInc'].astype('category')

# Split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=123
```

```
# Train the random forest model with cross-validation
rf_model = RandomForestClassifier(n_estimators=500, random_state=123)
rf_model.fit(X_train, y_train)

# Predict on the test set
rf_cv_predictions = rf_model.predict(X_test)

# Generate the confusion matrix
rf_cv_confMat = confusion_matrix(y_test, rf_cv_predictions)

# Calculate accuracy
rf_cv_accuracy = accuracy_score(y_test, rf_cv_predictions)

# Calculate precision for each class and weighted average precision
rf_cv_precision = precision_score(y_test, rf_cv_predictions, average='weighted')

# Calculate sensitivity (recall) for each class and weighted average sensitivity (recall)
rf_cv_sensitivity = recall_score(y_test, rf_cv_predictions, average='weighted')

# Combine accuracy, precision, and sensitivity into one DataFrame
rf_cv_results = pd.DataFrame({
  'Accuracy': [rf_cv_accuracy],
  'Sensitivity': [rf_cv_sensitivity],
  'Precision': [rf_cv_precision]
})

# Display the results
print(rf_cv_results)
```

```
   Accuracy  Sensitivity  Precision
0  0.681992     0.681992   0.682473
```

- Accuracy (0.681992): This represents a slight improvement over the fully grown decision tree's accuracy (~62.21%) and the pruned decision tree's accuracy (~66.95%). It indicates that the random forest model correctly predicts approximately 68.20% of the outcomes.
- Sensitivity (0.681992): The sensitivity or recall is identical to the accuracy in this instance, which may suggest balanced class distribution or model performance across classes. This is an improvement from the fully grown (~61.88%) and pruned (~67.26%) decision tree models.
- Precision (0.682473): This is consistent with the model's accuracy and sensitivity, which suggests a balanced precision across the predicted classes. Precision here shows a marginal improvement over the fully grown decision tree model (~61.95%) and is slightly better than the pruned tree (~67.23%).

- The random forest model's accuracy and other metrics are higher than both the fully grown and pruned decision tree models, indicating that the ensemble approach of random forests—using multiple trees and averaging their predictions—may be more effective for this dataset.
- The increase in sensitivity implies that the random forest model is better at identifying true positives among the actual positive cases compared to the single decision trees.
- The precision is slightly higher for the random forest model, suggesting that when it predicts a positive class, it is slightly more likely to be correct than the decision tree models.
- These improvements are likely due to the random forest model's ability to reduce variance (overfitting) by averaging the results of many trees trained on different subsets of the data. This generally leads to better performance on unseen data compared to a single decision tree.

- In summary, the cross-validated random forest model shows a balanced performance with a modest but consistent increase in accuracy, sensitivity, and precision compared to the individual decision tree models. This suggests that it could be a more robust model for this particular problem.

Support Vector Machine

- In the context of our analysis, we now turn our attention to the Support Vector Machine (SVM) model, which represents a powerful and versatile class of supervised machine learning algorithms. To enhance our understanding, let's draw a comparison between SVMs and the previously discussed Random Forest model, particularly noting the advantages of employing a kernel with SVMs in conjunction with cross-validation techniques.
- SVM is fundamentally distinct from Random Forest. At its core, SVM seeks to find the optimal hyperplane that separates classes in the feature space. For non-linearly separable data, SVM becomes particularly powerful when coupled with kernel functions. The kernel trick enables the algorithm to operate in a transformed feature space without the need for explicit mapping. Common kernels include linear, polynomial, radial basis function (RBF), and sigmoid.
- While Random Forest models have proven their efficacy, especially noted in their improved accuracy, sensitivity, and precision over single decision trees, SVMs with a suitable kernel may surpass Random Forests in certain scenarios:
- SVMs can be more effective when the data has a clear margin of separation.

Kernel SVMs are particularly adept at solving complex, small to medium-sized datasets. SVMs can provide better results when the classes are imbalanced. The sparsity of the SVM solution can result in a more computationally efficient model at inference time compared to the potentially large ensemble of trees in a Random Forest.

- In summary, the choice between Random Forest and Kernel SVM will depend on the specific characteristics of the dataset at hand. Cross-validation should be used in both cases to ensure that the models are not overfitting and that their performance metrics are robust. With the kernel trick, SVM can handle non-linear relationships in the data, providing a powerful alternative to Random Forests. Therefore, when the data exhibits complex relationships that a linear classifier cannot capture, a Kernel SVM may be the superior choice.
- In light of these considerations, the subsequent sections will delve into the application of Kernel SVMs with cross-validation on our dataset and the empirical evaluation of its performance metrics against those of the Random Forest model.

In [72]:
```
# Assuming 'variables' is a DataFrame with the target variable 'CatInc'
X = variables.drop(columns='CatInc')
y = variables['CatInc'].astype('category')

# Split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=202

# We will use a pipeline to standardize the data and then apply the SVM
pipeline = make_pipeline(StandardScaler(), SVC(probability=True, kernel='rbf', random_sta

# Set up the parameter grid. In scikit-learn, C and gamma are the parameters for an RBF k
param_grid = {
    'svc__C': [0.1, 1, 10],  # Example values, should be tuned
```

```
    'svc__gamma': [0.001, 0.01, 0.1]  # Example values, should be tuned
}

# Set up GridSearchCV with cross-validation
svm_cv_model = GridSearchCV(pipeline, param_grid, cv=5, scoring='roc_auc', n_jobs=-1)

# Fit the SVM model
svm_cv_model.fit(X_train, y_train)

# The best model can be accessed with svm_cv_model.best_estimator_
best_model = svm_cv_model.best_estimator_

# Make predictions
svm_cv_predictions = best_model.predict(X_test)

# Generate the confusion matrix
from sklearn.metrics import confusion_matrix
conf_matrix = confusion_matrix(y_test, svm_cv_predictions)

# Calculate accuracy
accuracy = best_model.score(X_test, y_test)

# Calculate ROC AUC
roc_auc = roc_auc_score(y_test, best_model.predict_proba(X_test)[:, 1])

# Print out the results
print(f'Confusion Matrix:\n{conf_matrix}')
print(f'Accuracy: {accuracy}')
print(f'ROC AUC: {roc_auc}')
```

```
Confusion Matrix:
[[157  88]
 [ 76 201]]
Accuracy: 0.685823754789272
ROC AUC: 0.7644293818610477
```

In [73]:

```
# Prepare the data
X_test = variables_test.drop('CatInc', axis=1)
y_test = variables_test['CatInc'].astype('category')

# Predict on the test set
svm_cv_predictions = svm_cv_model.predict(X_test)

# Generate the confusion matrix
svm_cv_confMat = confusion_matrix(y_test, svm_cv_predictions)

# Extracting True Positives, False Positives, False Negatives for each class
TP = np.diag(svm_cv_confMat)
FP = np.sum(svm_cv_confMat, axis=0) - TP
FN = np.sum(svm_cv_confMat, axis=1) - TP

# Sensitivity (recall) for each class
svm_cv_sensitivity_values = TP / (TP + FN)

# Precision for each class
svm_cv_precision_values = TP / (TP + FP)

# Weighted average sensitivity (recall)
svm_cv_sensitivity = np.average(svm_cv_sensitivity_values, weights=np.bincount(y_test))

# Weighted average precision
svm_cv_precision = np.average(svm_cv_precision_values, weights=np.bincount(y_test))

# Overall accuracy
```

```
svm_cv_accuracy = accuracy_score(y_test, svm_cv_predictions)

# Combine into one DataFrame
svm_cv_results = pd.DataFrame({
    'Accuracy': [svm_cv_accuracy],
    'Sensitivity': [svm_cv_sensitivity],
    'Precision': [svm_cv_precision]
})

# Display the results as a table
print(svm_cv_results)
```

```
   Accuracy  Sensitivity  Precision
0  0.702586     0.702586   0.701972
```

- Accuracy (0.702586): This SVM model achieves an accuracy of approximately 70.26%. This indicates that about 70 out of every 100 predictions are correct. When compared to the fully grown decision tree (~62.21%), the pruned decision tree (~66.95%), and the random forest model (~68.20%), the SVM has the highest accuracy, suggesting that it is the best at making correct predictions out of the models compared.

- Sensitivity (0.702586): The sensitivity, or recall, measures the proportion of actual positive cases correctly identified by the model. The SVM's sensitivity is the same as its accuracy, which might suggest a balanced classification or an even distribution of prediction errors among classes. This value is higher than the fully grown decision tree (~61.88%), the pruned decision tree (~67.26%), and the random forest model (~68.20%), indicating that the SVM is more effective at identifying positive cases.

- Precision (0.701972): Precision indicates how many of the positive predictions made by the model were actually positive. The SVM's precision is slightly lower than its accuracy but still higher than the fully grown decision tree (~61.95%), the pruned decision tree (~67.23%), and the random forest model (~68.25%). This demonstrates that the SVM is more reliable when it predicts a positive outcome.

- The results suggest that the SVM model, which often excels at handling high-dimensional data or datasets where classes are not linearly separable, performs better than the decision tree-based models in this scenario. The improvement in all three metrics with the SVM model could be due to its ability to find the optimal hyperplane that maximizes the margin between classes, which can be particularly effective for complex classification tasks.

- In conclusion, the SVM model shows a balance between accurately predicting positive cases and making a correct positive prediction when it claims to have observed one. The cross-validation process likely helped to tune the model effectively, leading to improved generalization on the test set compared to the single decision tree models and the random forest model.

Logistic Regression

In [79]:
```
# Assuming 'variables_train' is a pandas DataFrame with the target variable 'CatInc'
X_train = variables_train.drop('CatInc', axis=1)
y_train = variables_train['CatInc']

# Initialize logistic regression model
log_model = LogisticRegression(solver='liblinear')

# Perform K-fold cross-validation
```

```
cv_scores = cross_val_score(log_model, X_train, y_train, cv=5)

# The mean score and the 95% confidence interval of the score estimate are:
print("Accuracy: %0.2f (+/- %0.2f)" % (cv_scores.mean(), cv_scores.std() * 2))

# To match R's cv.glm delta output, we need the average of the cross-validated scores
mean_cv_score = cv_scores.mean()

# The standard deviation of the cv scores multiplied by sqrt(K) gives an estimate of the
error_std = cv_scores.std() * np.sqrt(5)

print("Cross-validation results:")
print(f"Mean CV score: {mean_cv_score}")
print(f"Standard error: {error_std}")
```

```
Accuracy: 0.70 (+/- 0.08)
Cross-validation results:
Mean CV score: 0.7009017298490983
Standard error: 0.09210484146783458
```

In [81]:
```
# Prepare the feature matrices and the target vectors
X_train = variables_train.drop('CatInc', axis=1)
y_train = variables_train['CatInc']
X_test = variables_test.drop('CatInc', axis=1)
y_test = variables_test['CatInc']

# Initialize logistic regression model
log_model = LogisticRegression(solver='liblinear')

# Fit the model with training data
log_model.fit(X_train, y_train)

# Now that the model is fitted, you can predict probabilities on the test set
log_pred_probs = log_model.predict_proba(X_test)[:, 1]  # Probabilities for the positive
```

In [84]:
```
# Prepare the feature matrix for the test set
X_test = variables_test.drop('CatInc', axis=1)
y_test = variables_test['CatInc']

# Predict probabilities on the test set
log_pred_probs = log_model.predict_proba(X_test)[:, 1]  # Get the probabilities for the p

# Convert probabilities to a class decision (0 or 1) based on a 0.5 threshold
log_pred_class = (log_pred_probs > 0.5).astype(int)

# Generate the confusion matrix
log_confMat = confusion_matrix(y_test, log_pred_class)

# Calculate accuracy
log_accuracy = accuracy_score(y_test, log_pred_class)

# Calculate sensitivity (recall) for the positive class ('1')
sensitivity = recall_score(y_test, log_pred_class)

# Calculate precision for the positive class ('1')
precision = precision_score(y_test, log_pred_class)

# Combine into one DataFrame
performance_metrics = pd.DataFrame({
    'Accuracy': [log_accuracy],
    'Sensitivity': [sensitivity],
    'Precision': [precision]
})
```

```
# Print the performance metrics
print(performance_metrics)
```

```
   Accuracy  Sensitivity  Precision
0  0.686782     0.743316      0.695
```

- Accuracy: 68.68%
- Sensitivity: 74.33%
- Precision: 69.5%

Comparing these to the earlier models:

- Decision Tree (fully grown): Accuracy (~62.21%), Sensitivity (~61.88%), Precision (~61.95%)
- Decision Tree (pruned): Accuracy (~66.95%), Sensitivity (~67.26%), Precision (~67.23%)
- Random Forest: Accuracy (~68.20%), Sensitivity (~68.20%), Precision (~68.25%)
- SVM: Accuracy (~70.26%), Sensitivity (~70.26%), Precision (~70.20%)
- The current model has better accuracy than both the fully grown and pruned decision trees but is slightly less accurate than the random forest model and SVM. However, the current model outperforms all previous models in terms of sensitivity, suggesting it is better at identifying true positive cases.
- In terms of precision, the current model is better than the fully grown decision tree but slightly less precise than the pruned decision tree, random forest, and SVM. This indicates that while the current model is more likely to identify all positive cases, it may also include more false positives than the random forest and SVM.
- Overall, if the goal is to capture as many positive cases as possible (high sensitivity), this model seems to be the best of those compared. If the goal is to be sure of a positive prediction's correctness (high precision), then the SVM would be a better choice. These trade-offs between sensitivity and precision are common in classification tasks and should be considered alongside the specific costs of false positives and false negatives in the problem domain.

Conclusions

- Features Classifying Couples by Income:

The analysis of feature importance across different models, particularly the random forest model, indicated that certain features like 'Duration', 'AGE', and 'EDUCATION' were more influential in classifying couples as earning higher than $60k or less. These features can be interpreted as factors that potentially correlate with the income level of couples in the dataset.

- Most Important Features for Income Prediction:

The 'Duration' of the relationship, which might imply stability or dual income over time, was identified as one of the most significant predictors. 'AGE' and 'EDUCATION' followed, suggesting that income levels may be associated with the age (possibly reflecting experience or career advancement) and educational attainment (indicating job opportunities and earning potential).

- Method Classifying Better:

When comparing different models, the SVM and logistic regression models demonstrated high accuracy, sensitivity, and precision, indicating strong performance in classifying couples based on income. However, the SVM showed a slightly better precision than logistic regression, which means it was slightly better at confirming when a couple was indeed in the higher-income class without as many false positives.

In conclusion, our analysis aimed to classify couples by their income levels, using machine learning models to understand the features that contribute to higher earning potential. The study revealed that relationship duration, age, and education level were significant indicators of income category. Among the methods evaluated, the support vector machine model provided the best precision, indicating its robustness in classifying couples accurately. However, the logistic regression model showed the highest sensitivity, making it the most reliable for identifying couples in the higher income bracket.

When choosing a model for deployment, the context in which it will be used must be considered. If minimizing false positives is paramount (i.e., incorrectly assuming a high income), the SVM model stands out as the preferred choice. Conversely, if it is critical to capture as many high-earning couples as possible, the logistic regression model may be more suitable due to its higher sensitivity.

Future work could involve further tuning of the models, exploration of ensemble methods, or deep dives into demographic or economic features that could refine our predictions. Additionally, understanding the reasons behind the patterns observed in the significant features could provide socio-economic insights that extend beyond the scope of this analysis.