Yearly income in the US in 1994

The dataset to be studied in this project belongs to the 1994 census data of the United States. It contains continuous and nominal attributes, describing some social information (age, race, sex, marital status, ...) about the citizens registered. The task is to predict whether the citizens income exceeds fifty thousand dollars a year, so its a binary classification task.

```
!pip install jovian scikit-learn --upgrade --quiet
```

```
     |████████████████████████████████| 22.3 MB 1.6 MB/s
```

```
!pip install numpy pandas matplotlib seaborn plotly --quiet
```

```
import jovian
```

```
# Execute this to save new versions of the notebook
jovian.commit(project="us-census-1994")
```

```
[jovian] Detected Colab notebook...
[jovian] Please enter your API key ( from https://jovian.ai/ ):
API KEY: ··········
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

```
'https://jovian.ai/viquiriglos/us-census-1994'
```

1. Find and Load the Data

First I will import some libraries that will be useful.

```
import pandas as pd
import numpy as np
import plotly.express as px
```

The dataset is found at (census dataset): https://sci2s.ugr.es/keel/category.php?cat=clas&order=name#sub2

KEEL-dataset citation paper: J. Alcalá-Fdez, A. Fernandez, J. Luengo, J. Derrac, S. García, L. Sánchez, F. Herrera. KEEL Data-Mining Software Tool: Data Set Repository, Integration of Algorithms and Experimental Analysis Framework. Journal of Multiple-Valued Logic and Soft Computing 17:2-3 (2011) 255-287.

I have downloaded the dataset manually to my PC and then I have converted to a csv file type. Originally, it presents 40 features but, as many of the columns are not very relevant or present redundant information, I have kept only 11 features that were the most interesting, in my opinion. As the dataset was too big I couldn't upload it directly to Google Colab, so I've Uploaded first to Google Drive and then load it into Google Colab. There are many other ways of doing this but I've found this was the easiest one.

```
from google.colab import drive
drive.mount('/gdrive')
%cd /gdrive
```

```
Mounted at /gdrive
/gdrive
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
data='/content/drive/MyDrive/Census11.csv'
```

```
df= pd.read_csv(data)
```

```
df.head()
```

	Age	Workclass	Education	Marital_Status	Working_Field	Rac
0	58	Self-employed-not_incorporated	Some_college_but_no_degree	Divorced	Construction	Whit
1	9	Not_in_universe	Children	Never_married	Not_in_universe_or_children	Whit
2	10	Not_in_universe	Children	Never_married	Not_in_universe_or_children	Whit
3	42	Private	Bachelors_degree(BA_AB_BS)	Married-civilian_spouse_present	Finance_insurance_and_real_estate	Whit
4	34	Private	Some_college_but_no_degree	Married-civilian_spouse_present	Construction	Whit

```
df.columns
```

```
Index(['Age', 'Workclass', 'Education', 'Marital_Status', 'Working_Field',
       'Race', 'Sex', 'Income_Responsible', 'Native_Country', 'US_Citicenship',
       'Working_Hours_per_Week', 'Class'],
      dtype='object')
```

```
#Saving our work at jovian
jovian.commit(project="us-census-1994")
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

```
'https://jovian.ai/viquiriglos/us-census-1994'
```

2. Explore and prepare the Data

Let's see first how big is our dataframe and the type of information that we have in each column:

```
df.shape
```

```
(142521, 12)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 142521 entries, 0 to 142520
Data columns (total 12 columns):
 #   Column                 Non-Null Count   Dtype 
---  ------                 --------------   ----- 
 0   Age                    142521 non-null  int64 
 1   Workclass              142521 non-null  object
 2   Education              142521 non-null  object
 3   Marital_Status         142521 non-null  object
 4   Working_Field          142521 non-null  object
 5   Race                   142521 non-null  object
 6   Sex                    142521 non-null  object
 7   Income_Responsible     142521 non-null  object
 8   Native_Country         142521 non-null  object
 9   US_Citicenship         142521 non-null  object
 10  Working_Hours_per_Week  142521 non-null  int64 
 11  Class                  142521 non-null  object
dtypes: int64(2), object(10)
memory usage: 13.0+ MB
```

We could take a look at some features. For instance, we could try to observe if there is an influence in the yeraly incomes due to the gender of the person:

```
px.histogram(df, x='Class', color='Sex', title='Incomes')
```

In the image we can observe that for the lower Class (incomes below 50000 dollars per year) it doesn't seem to be much difference between men and women. However, for the higher Class (incomes above 50000) the fringe of men is much wider than for women. We can observe, too, that the amount of people in the higher Class is much smaller than in the lower Class.

We could try to picture the influence of other features in the incomes. For example if it is easier or not to reach a high income coming from a foreign country (outside of US):

```
px.histogram(df, x='US_Citicenship', color='Class', title='Citicenship vs. Incomes')
```

```
Output hidden; open in https://colab.research.google.com to view.
```

It can be seen (in absolute numbers) that the people in the rich fringe is more for people born in the US. Nevertheless, the proportion between high and low incomes is almost the same for each column in the histogram.

From df.info() method we can see that we have 12 columns. The first eleven columns belong to the features (inputs) and the last one is the target (the value that we want to predict). Additionally, within the features, we can observe that mostly the columns are categorical (except age and working hours per week). The steps to follow in order to prepare our data are as follows:

a. Create two distinct variables for inputs and targets.

b. Identify numerical and Categorical columns.

c. Check if there are missing values and in case we find them take an action (drop those rows or impute an adequate value to fill in the missing information).

d. Scale the numeric data to values between 0 and 1.

e. After that, encode the categorical columns (maybe using a hot encoder).

f. Split the data into 2 subsets (train and validation).

a. Creating variables for Inputs and Targets

```
# create a list of input and target columns:
inputs_cols = df.columns[0:-1]
target_col = df.columns[-1]
```

```
inputs_cols
```

```
Index(['Age', 'Workclass', 'Education', 'Marital_Status', 'Working_Field',
       'Race', 'Sex', 'Income_Responsible', 'Native_Country', 'US_Citicenship',
       'Working_Hours_per_Week'],
      dtype='object')
```

```
target_col
```

```
'Class'
```

As the target has only two categories, I will assign '0' to the class earning less or equal to 50000 dollars per year and '1' to incomes higher than 50000/year.

```
df['Class'].unique()
```

```
array(['-_50000.', '50000+.'], dtype=object)
```

```
class_codes = {'-_50000.': 0, '50000+.': 1}
```

```
df['class_codes'] = df['Class'].map(class_codes)
df.head()
```

	Age	Workclass	Education	Marital_Status	Working_Field	Rac
0	58	Self-employed-not_incorporated	Some_college_but_no_degree	Divorced	Construction	Whit

	Age	Workclass	Education	Marital_Status	Working_Field	Rac
1	9	Not_in_universe	Children	Never_married	Not_in_universe_or_children	Whit
2	10	Not_in_universe	Children	Never_married	Not_in_universe_or_children	Whit
3	42	Private	Bachelors_degree(BA_AB_BS)	Married-civilian_spouse_present	Finance_insurance_and_real_estate	Whit
4	34	Private	Some_college_but_no_degree	Married-civilian_spouse_present	Construction	Whit

I will replace the target column ('Class') by the new column: 'class_codes'.

```
target_col = df.columns[-1]
target_col
```

```
'class_codes'
```

Now I'll make a copy of the columns in the dataframe that correspond to inputs and a copy of the target column and assign them to respective variables.

```
inputs_df = df[inputs_cols].copy()
inputs_df.head()
```

	Age	Workclass	Education	Marital_Status	Working_Field	Rac
0	58	Self-employed-not_incorporated	Some_college_but_no_degree	Divorced	Construction	Whit
1	9	Not_in_universe	Children	Never_married	Not_in_universe_or_children	Whit
2	10	Not_in_universe	Children	Never_married	Not_in_universe_or_children	Whit
3	42	Private	Bachelors_degree(BA_AB_BS)	Married-civilian_spouse_present	Finance_insurance_and_real_estate	Whit
4	34	Private	Some_college_but_no_degree	Married-civilian_spouse_present	Construction	Whit

```
target_df = df[target_col].copy()
target_df.head()
```

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

```
# Making a new commit to Jovian
jovian.commit(project="us-census-1994")
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

```
'https://jovian.ai/viquiriglos/us-census-1994'
```

b. Identifying Categorical and Numerical Inputs

For this step, we will first create a list for the input columns of the numeric and categorical types respectively:

```
numeric_cols = inputs_df.select_dtypes(include=['int64', 'float64']).columns.tolist()
numeric_cols
```

```
['Age', 'Working_Hours_per_Week']
```

```
categorical_cols = inputs_df.select_dtypes(include=['object']).columns.tolist()
categorical_cols
```

```
['Workclass',
 'Education',
 'Marital_Status',
 'Working_Field',
 'Race',
 'Sex',
 'Income_Responsible',
 'Native_Country',
 'US_Citicenship']
```

```
# Making a new commit to Jovian
jovian.commit(project="us-census-1994")
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

```
'https://jovian.ai/viquiriglos/us-census-1994'
```

c. Dealing with missing values

First I'll check if there are any missing values inside the numerical columns:

```
missing_counts = inputs_df[numeric_cols].isna().sum().sort_values(ascending=False)
missing_counts[missing_counts > 0]
```

```
Series([], dtype: int64)
```

We can observe there are no missing values in those columns. We'll perform the same operation in the categorical columns:

```
missing_cat = inputs_df[categorical_cols].isnull().sum()
missing_cat[missing_cat > 0]
```

```
Series([], dtype: int64)
```

Let's check the same in the target column:

```
missing_target = target_df.isna().sum()
missing_target[missing_target > 0]
```

```
array([], dtype=int64)
```

There are neither missing values inside any of the columns, so no action is required in this sense.

```
# Commit to Jovian
jovian.commit(project="us-census-1994")
```

```
[jovian] Detected Colab notebook...
[jovian] Please enter your API key ( from https://jovian.ai/ ):
API KEY: ··········
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

```
'https://jovian.ai/viquiriglos/us-census-1994'
```

d.Scaling Numeric Features

Before applying any more changes to the data it will be convenient to scale the numeric values to convert the values into values between 0 and 1. For that purpose we will import MinMaxScaler, we'll fit the scaler to the numeric data and then transform the values.

```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler().fit(inputs_df[numeric_cols])
```

```
inputs_df[numeric_cols] = scaler.transform(inputs_df[numeric_cols])
```

```
inputs_df[numeric_cols].head()
```

	Age	Working_Hours_per_Week
0	0.644444	1.0
1	0.100000	0.0
2	0.111111	0.0
3	0.466667	1.0
4	0.377778	1.0

e. Encode the categorical columns.

Let's see how many categories we have for each categorical column:

```
inputs_df[categorical_cols].nunique().sort_values(ascending=False)
```

```
Native_Country        41
Working_Field         24
Education             17
Workclass              9
Income_Responsible     8
Marital_Status         7
US_Citicenship         5
Race                   5
Sex                    2
dtype: int64
```

To apply a one-hot encoder we will need, first, to import the "OneHotEncoder" library from sklearn.preprocessing module. This encoder will create a new column for each category inside the categorical columns. We will have to create the encoder, fit it to the columns (this will create the new columns) and name each of the new columns with the name of the corresponding category. Finally we add these new columns to the inputs_df data frame.

As we have many rows, we will only hot-encode columns having less than 10 categories (low cardinality columns), we will discard the rest.

```
low_cardinality_cols = [col for col in categorical_cols if inputs_df[col].nunique() < 1

high_cardinality_cols = list(set(categorical_cols)-set(low_cardinality_cols))
print('Low cardinality columns: ',low_cardinality_cols)
print('High cardinality columns: ',high_cardinality_cols)
```

```
Low cardinality columns:  ['Workclass', 'Marital_Status', 'Race', 'Sex',
'Income_Responsible', 'US_Citicenship']
High cardinality columns:  ['Working_Field', 'Native_Country', 'Education']
```

```
from sklearn.preprocessing import OneHotEncoder
```

```
# 1. Create the encoder
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
```

```
# 2. Fit the encoder to the categorical colums with low cardinality (small number of ca
encoder.fit(inputs_df[low_cardinality_cols])
```

```
OneHotEncoder(handle_unknown='ignore', sparse=False)
```

```
# 3. Generate column names for each category
encoded_cols = list(encoder.get_feature_names(low_cardinality_cols))
len(encoded_cols)
```

```
# 4. Transform and add new one-hot category columns
one_hot = encoder.transform(inputs_df[low_cardinality_cols])
inputs_df[encoded_cols] = one_hot
```

```
inputs_df.head()
```

	Age	Workclass	Education	Marital_Status	Working_Field
0	0.644444	Self-employed-not_incorporated	Some_college_but_no_degree	Divorced	Construction
1	0.100000	Not_in_universe	Children	Never_married	Not_in_universe_or_children
2	0.111111	Not_in_universe	Children	Never_married	Not_in_universe_or_children
3	0.466667	Private	Bachelors_degree(BA_AB_BS)	Married-civilian_spouse_present	Finance_insurance_and_real_estate
4	0.377778	Private	Some_college_but_no_degree	Married-civilian_spouse_present	Construction

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Please enter your API key ( from https://jovian.ai/ ):
API KEY: ··········
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

```
'https://jovian.ai/viquiriglos/us-census-1994'
```

f. Split the data into train and validation data sets

Let's split the data into training and validation sets. For doing so, I will first import the train_test_split library.

```
from sklearn.model_selection import train_test_split
```

Now, I will split the inputs in train_inputs (75% of the rows) and val_inputs (the remaining 25%). The same is done for the targets, they are split into train_targets and val_targets.

```
train_inputs, val_inputs, train_targets, val_targets = train_test_split(inputs_df[numer
                                                                        target_df,
                                                                        test_size=0.25,
                                                                        random_state=40
```

```
train_inputs.head()
```

	Age	Working_Hours_per_Week	Workclass_Federal_government	Workclass_Local_government	Workclass_N
55772	0.233333	0.769231	0.0	0.0	
100947	0.033333	0.000000	0.0	0.0	
100193	0.022222	0.000000	0.0	0.0	
32572	0.455556	1.000000	0.0	1.0	
41188	0.666667	1.000000	0.0	0.0	

```
train_inputs.shape
```

```
(106890, 38)
```

```
train_targets.head()
```

```
55772     0
100947    0
100193    0
32572     0
41188     0
Name: class_codes, dtype: int64
```

```
len(train_targets)
```

```
106890
```

Everything looks fine in the train inputs and targets. At least, they have the same length and the train_inputs have 38 columns (2 numeric and 36 encoded). Let's take a look at the validation set:

```
val_inputs.head()
```

	Age	Working_Hours_per_Week	Workclass_Federal_government	Workclass_Local_government	Workclass_N
125555	0.733333	0.115385	0.0	0.0	
141952	0.344444	0.000000	0.0	0.0	
120283	0.511111	1.000000	0.0	0.0	
73176	0.644444	1.000000	0.0	0.0	
77068	0.555556	1.000000	0.0	0.0	

```
val_targets.head()
```

```
125555    0
141952    0
120283    1
73176     0
77068     0
Name: class_codes, dtype: int64
```

```
val_inputs.shape
```

```
(35631, 38)
```

```
len(val_targets)
```

```
35631
```

The validation set looks fine, as well.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

```
'https://jovian.ai/viquiriglos/us-census-1994'
```

3. Fit and train different models.

As it is a binary classification problem, I will first try a Logistic Regression model. Then, I will study a Decision Tree and a Random Forest model.

a. Logistic Regression

I will import the corresponding library from Scikit and define the model. Then, I will fit the model to the data and make predictions. I will train the model and check its accuracy in both training and validation sets.

```
from sklearn.linear_model import LogisticRegression
```

```
model = LogisticRegression(solver='liblinear')
```

```
model.fit(train_inputs, train_targets)
```

```
LogisticRegression(solver='liblinear')
```

Now that the model is defined, we will making predictions and evaluate the performance of the model.

```
train_preds = model.predict(train_inputs)
```

```
from sklearn.metrics import accuracy_score
```

```
accuracy_score(train_targets, train_preds)
```

```
0.9424455047244831
```

```
val_preds = model.predict(val_inputs)
```

```
accuracy_score(val_targets, val_preds)
```

```
0.9437007100558502
```

The model performs very good in both sets which means we are not having overfitting issues. It reaches an accuracy higher than 90%. We can take a look at the coefficients and the intercept of the model next:

```
print(model.coef_.tolist())
```

```
[[2.414996185106447, 2.655397059074427, 0.15957781224838907, -0.3716954451554031,
-0.3909628263409099, -0.5851244337330327, -0.35377880276159973, 0.6300853696407844,
-0.49654005704858384, -0.31287242587885383, -1.6648857685158092, -0.43210114299617564,
-1.0181176637442215, -0.22799723639068267, -0.16480569474158055, -0.631082607065348,
-0.6105231832606829, -0.30156904944129154, -1.2890268688913165, 0.1831200095516849,
-1.0451507222897654, -0.8501671096758419, -0.3849718860506851, -2.277440728106094,
-1.1087558493646958, -0.9190391330493313, -0.037858974727692764, -2.5879123977080414,
-0.6059518462365918, 0.8153402826460948, -0.020251794429744695, -0.5162906623326999,
0.48576794828239217, -1.2133159769839015, -0.4725544883197381, -0.20146497484100095,
-1.1919649147655385, -0.3068962224415578]]
```

```
print(model.intercept_)
```

```
[-3.38619658]
```

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

```
'https://jovian.ai/viquiriglos/us-census-1994'
```

b. Decision Tree Model

We will define a second model (Decision Tree Classifier), fit it to the data and make predictions.

```
from sklearn.tree import DecisionTreeClassifier
```

```
model2 = DecisionTreeClassifier(random_state=42)
```

```
model2.fit(train_inputs, train_targets)
```

```
DecisionTreeClassifier(random_state=42)
```

Now that the decision tree is created, I will evaluate its accuracy

```
train_preds2 = model2.predict(train_inputs)
```

```
accuracy_score(train_targets, train_preds2)
```

```
0.953578445130508
```

```
val_preds2 = model2.predict(val_inputs)
```

```
accuracy_score(val_targets, val_preds2)
```

```
0.9337374757935506
```

This second model have reached an even better performance than the previous one in the training data but not in the validation data, meaning that exist some overfitting issues.

```

```

c. Random Forest Model

```
from sklearn.ensemble import RandomForestClassifier
```

```
model3 = RandomForestClassifier(n_jobs=-1, random_state=42)
```

```
model3.fit(train_inputs, train_targets)
```

```
RandomForestClassifier(n_jobs=-1, random_state=42)
```

```
train_preds3 = model3.predict(train_inputs)
```

```
accuracy_score(train_targets, train_preds2)
```

```
0.953578445130508
```

```
val_preds3 = model3.predict(val_inputs)
```

```
accuracy_score(val_targets, val_preds3)
```

```
0.9386769947517611
```

Another way of displaying the accuracy is using the score method:

```
model3.score(train_inputs, train_targets)
```

```
0.9535597343062961
```

```
model3.score(val_inputs, val_targets)
```

```
0.9386769947517611
```

We can observe here some overfitting problems, as the accuracy in the training set is higher than that in the validation set and additionally the accuracy in the validation set is lower than the obtained for the first model.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

```
'https://jovian.ai/viquiriglos/us-census-1994'
```

Tuning the Hyperparameters

I will create some functions to easily observe changes while modifying the hyperparameters in the second and third models (the decission tree and the Random Forest models) and thus, decide which of those hyperparameters are better to avoid overfitting.

Improving the Decission Tree Model

```
def get_score_tree_depth(X, y, depths):
    """Return the score for a Decission Tree Classifier
    X=inputs
    y=targets
    depths -- a list with some values for the max_depth
    """
    scores={}

    for depth in depths:
        model = DecisionTreeClassifier(max_depth=depth, random_state=42)
        model.fit(train_inputs, train_targets)
        score = model.score(X, y)
        scores[depth] = score

    return scores
```

```
some_depths=[i for i in range(1,40,3)]
some_depths
```

```
[1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37]
```

```
get_score_tree_depth(train_inputs, train_targets, some_depths)
```

```
{1: 0.9424361493123772,
 4: 0.9424361493123772,
 7: 0.9431003835718963,
 10: 0.9444475629151464,
 13: 0.946711572644775,
 16: 0.9494246421554869,
```

```
 19: 0.9516325194124802,
 22: 0.9529142108709888,
 25: 0.9534474693610253,
 28: 0.9535597343062961,
 31: 0.953578445130508,
 34: 0.953578445130508,
 37: 0.953578445130508}
```

```
get_score_tree_depth(val_inputs, val_targets, some_depths)
```

```
{1: 0.9436165137099717,
 4: 0.9436165137099717,
 7: 0.9433919901209621,
 10: 0.9417361286520165,
 13: 0.9386208638545087,
 16: 0.9366843478993012,
 19: 0.935084617327608,
 22: 0.9344952429064579,
 25: 0.933933933933934,
 28: 0.9339619993825601,
 31: 0.9340181302798125,
 34: 0.9338778030366816,
 37: 0.9337374757935506}
```

The accuracy in the validation improved with respect to the previously encountered value (93.37%) when we had not specified the max_depth. However, the accuracy get worse when increasing max_depth further from 7. Then, the accuracy starts to slightly decay from 94.3% to 93.37%. It's not a big lost in accuracy but it's not convenient to modify this parameter outside the 1-6 range.

Let's try now to modify tha maximum number of leaf nodes:

```
def get_score_tree_leaf(X, y, leaves):
    """Return the score for a Decission Tree Classifier
    X=inputs
    y=targets
    leaves -- a list with some values for the max_leaf_nodes
    """
    scores={}

    for leaf in leaves:
        model = DecisionTreeClassifier(max_leaf_nodes=leaf, random_state=42)
        model.fit(train_inputs, train_targets)
        score = model.score(X, y)
        scores[leaf] = score

    return scores
```

```
some_leaves=[i for i in range(60,300,60)]
some_leaves
```

```
[60, 120, 180, 240]
```

```
get_score_tree_leaf(train_inputs, train_targets, some_leaves)
```

```
{60: 0.9432594255776967,
 120: 0.9437271961829918,
 180: 0.9441294789035457,
 240: 0.9445130507998878}
```

```
get_score_tree_leaf(val_inputs, val_targets, some_leaves)
```

```
{60: 0.9435884482613455,
 120: 0.9431674665319525,
 180: 0.9426903539053072,
 240: 0.9422693721759142}
```

Here again, setting a value for max_leaf_nodes showed an improved response for the accuracy of the model in the validation set, but the increasing much the max_leaf_nodes (more than 120) does not improve the accuracy. I will try the best values that I've found together:

```
model_tree_final = DecisionTreeClassifier(max_depth=4, max_leaf_nodes=60, random_state=
model_tree_final.fit(train_inputs, train_targets)
score_tree_train = model_tree_final.score(train_inputs, train_targets)
score_tree_train
```

```
0.9424361493123772
```

```
score_tree_val = model_tree_final.score(val_inputs, val_targets)
score_tree_val
```

```
0.9436165137099717
```

Here, setting the max depth to 4 and the max_leaf_nodes to 60, we could obtain a better score for the validation set in the Decission Tree model, even better than the obtained in the training set. Furthermore, we've reached almost the same accuracy than the Logistic Regression Model.

Improving the Random Forest Model

```
def get_score(X, y, estimators):
    """Return the score for a Random Forest Classifier
    X=inputs
    y=targets
    n_estimators -- the number of trees in the forest
    """
    scores={}

    for estimator in estimators:
        model = RandomForestClassifier(n_estimators=estimator, n_jobs=-1, random_state=
        model.fit(train_inputs, train_targets)
        score = model.score(X, y)
```

```
        scores[estimator] = score

    return scores
```

```
N_estimators=[i for i in range(50,300,50)]
N_estimators
```

```
[50, 100, 150, 200, 250]
```

```
get_score(train_inputs, train_targets, N_estimators)
```

```
{50: 0.9534381139489194,
 100: 0.9535597343062961,
 150: 0.953578445130508,
 200: 0.953578445130508,
 250: 0.953578445130508}
```

```
get_score(val_inputs, val_targets, N_estimators)
```

```
{50: 0.9381437512278634,
 100: 0.9386769947517611,
 150: 0.9388453874435183,
 200: 0.9387050602003874,
 250: 0.9387892565462659}
```

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

```
'https://jovian.ai/viquiriglos/us-census-1994'
```

in the Decission Tree, we can observe that the maximum accuracy achieved in the validation set is obtained when setting the n_estimators to a value of 150. The accuracy improved till 93.88% and then starts to decay. The best accuracy achieved for this model is even lower than in the first two cases. This model is more complex and time consuming than the Logistic Regression and the Decission Tree.

Final Comments

For this dataset I would recommend a Logistic Regression model because we are dealing with a binary classification problem and we have a great amount of data, so it shows a very good performance, i.e., it's fast and accurate and more simple. Decission Tree is an intermediate solution but the hyperparameters must be modified to achive the same accuracy as that of the Logistic regression model. The Random Forest model presented a slightly lower accuracy after hyperparameter tuning but, despite this, I wouldn't recommend it because it's slower and more difficult to optimize.

The data is not evenly distributed between high and low incomes, there is much more data in the low incomes region (less than 50000 dollars per year), but this haven't presented any problem for the modeling because the amount of data is quite big.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/viquiriglos/us-census-1994
```

'https://jovian.ai/viquiriglos/us-census-1994'

```
# me faltaria ver si la precision esta bien distribuida entre el "sí" y el "no".
```

```

```