# US cencus income : Ensembles, Bagging and Shap Values

* toc: true
* badges: true
* comments: true
* categories: [fastpages, jupyter]
* image: images/shap\_values.png

# Problem Framework

Our task is to determine the income level for the person represented by the record. Incomes have been binned at the $50K level to present a **binary classification problem**.

The dataset used in this analysis was extracted from the census bureau database found at. The data was split into train/test in approximately 2/3, 1/3 proportions.

The following mappings of the data is as follow :

import pandas as pd  
from pandas.api.types import is\_string\_dtype, is\_numeric\_dtype, is\_categorical\_dtype  
import numpy as np

df\_labels = pd.read\_csv(f'{PATH}/census\_income\_metadata\_column.csv', sep=';')  
df\_labels.head(5)

In any sort of data science work, it's important to look at our data directly to make sure we understand the format, how it's stored, what types of values it holds, etc. Even if we've read a description of the data, the actual data may not be what we expect. We'll start by reading the training set into a Pandas DataFrame :

# Loading the train data   
df = pd.read\_csv(f'{PATH}/census\_income\_learn.csv', names = df\_labels['column\_name'])  
df.shape

Let's have a look at the columns, their types defined by Pandas and compared it to their actual mapping types :

# Chekcing the mapping of the data   
d1 = df.dtypes.apply(lambda x: x.name).to\_dict()  
d2 = {c: d for c,d in zip(df\_labels['column\_name'],df\_labels['dtype'])}  
mapping = [d1, d2]  
d = {}  
for k in d1.keys():  
 d[k] = tuple(d[k] for d in mapping)  
d

We can see that **detailed\_industry\_recode**, **detailed\_occupation\_recode**, **own\_business\_or\_self\_employed**, **veterans\_benefits** and **year** is set by default as a continuos category.

Let's redifined their types :

# Correcting data types  
d1['detailed\_industry\_recode']='object'  
d1['detailed\_occupation\_recode']='object'  
d1['own\_business\_or\_self\_employed']='object'  
d1['veterans\_benefits']='object'  
d1['year']='object'

Let's reload the data with its correspind feature's mapping :

# reload data with coorexted types  
df = pd.read\_csv(f'{PATH}/census\_income\_learn.csv', names =df\_labels['column\_name'],  
 dtype= d1)

The **info()** method is useful to get a quick description of the data, in particular the total number of rows, each attribute’s type, and the number of nonnull values :

df.info()

# dispplay first rows   
with pd.option\_context('display.max\_rows', None, 'display.max\_columns', None):   
 display(df.head(3))

# drop 'ignore' column  
df.drop('ignore', axis=1,inplace=True)

# list columns  
df.columns

We load the test set with the same training data types :

# loading the test set   
test = pd.read\_csv(f'{PATH}/census\_income\_test.csv', names =df\_labels['column\_name'], dtype= d1 )

test.info()

We verify if we got the same columns both on the train and the test set :

# checking columns on test set which not in train  
set(test.columns).difference(set(df.columns))

# dropping 'ignore' columns  
test.drop('ignore', inplace=True, axis=1)

test.columns

# display first rows of the test set  
with pd.option\_context('display.max\_rows', None, 'display.max\_columns', None):  
 display(test.head(3))

df.shape, test.shape

# Looking at the data

The most important data column is the **dependent variable**—that is, the one we want to predict which is **income\_level** :

dep\_var = 'income\_level'

Let's see its distribution :

print(df[dep\_var].value\_counts(normalize = True))  
df[dep\_var].value\_counts(normalize = True).plot(kind='bar',  
 edgecolor='black',  
 color='blue',  
 title='income\_level')

We have an **imbalanced dataset** where the income level of -50k is representing more than 93% of the total records.

Next, we automatically handle which columns are **continuous** and which are **categorical** :

# get categorical and numerical variables  
def cont\_cat\_split(df, dep\_var=None):  
 "Helper function that returns column names of cont and cat variables from given `df`."  
 cont\_names, cat\_names = [], []  
 for label in df:  
 if label in [dep\_var]: continue  
 if (pd.api.types.is\_integer\_dtype(df[label].dtype) or  
 pd.api.types.is\_float\_dtype(df[label].dtype)):  
 cont\_names.append(label)  
 else: cat\_names.append(label)  
 return cont\_names, cat\_names  
  
cont, cat = cont\_cat\_split(df, dep\_var= dep\_var)  
cont , cat

Let's start by checking the modalties of our categorical variables :

Some categorical features are **purely nominal**-having multiple modalities (with modality **?** for nan values) and others are **ordinal columns** like **education** and **year**:

# Ediucation modalities  
df['education'].unique(), df['education'].nunique()

# Year modalities  
df['year'].unique(), df['year'].nunique()

We can tell Pandas about a suitable ordering of these levels like so:

# Setting the order of education variable  
education = ' Children',' Less than 1st grade',' 1st 2nd 3rd or 4th grade',' 5th or 6th grade',\  
' 7th and 8th grade',' 9th grade',' 10th grade',' 11th grade', ' 12th grade no diploma',\  
' High school graduate', ' Associates degree-academic program',' Associates degree-occup /vocational',\  
' Prof school degree (MD DDS DVM LLB JD)',' Some college but no degree',' Bachelors degree(BA AB BS)',\  
' Masters degree(MA MS MEng MEd MSW MBA)',' Doctorate degree(PhD EdD)'  
len(education)

# Setting the order of year variaable  
year = '94', '95'

# apply the defined ordering fot our data :  
df['education'] = df['education'].astype('category')  
df['education'].cat.set\_categories(education, ordered=True, inplace=True)  
  
df['year'] = df['year'].astype('category')  
df['year'].cat.set\_categories(year, ordered=True, inplace=True)  
  
#Same for test set :  
test['education'] = test['education'].astype('category')  
test['education'].cat.set\_categories(education, ordered=True, inplace=True)  
  
test['year'] = test['year'].astype('category')  
test['year'].cat.set\_categories(year, ordered=True, inplace=True)

Lets check our continous features: The **describe()** method shows a summary of the numerical attributes

df[cont].describe()

The **count**, **mean**, **min**, and **max** rows are self-explanatory.The **std** row shows the standard deviation, which measures how dispersed the values are. The 25%, 50%, and 75% rows show the corresponding percentiles.

We plot a histogram for each numerical attribute :

%matplotlib inline   
import matplotlib.pyplot as plt  
df[cont].hist(bins=50, figsize=(20,15))  
plt.show()

* We can see that these attributes have very **different scales**.
* Some numerical varaibles are countinous like **age** and others are discrete and finite like **weeks\_worked\_in\_year** or infinete **num\_persons\_worked\_for\_employer**.
* Some features as **wage\_per\_hour**,**capital\_gains**,**capital\_losses**,**dividends\_from\_stocks** are tail-heavy: they extend much farther to the median right with high coefficient of variation :

df[cont].boxplot(column=['wage\_per\_hour','capital\_gains','capital\_losses','dividends\_from\_stocks'],  
 figsize=(10,5))

We can see the presence of **extreme values** for those features.

Using the **skewness value**, which explains the extent to which the data is normally distributed, in order to confirm that. Ideally, the skewness value should be between -1 and +1, and any major deviation from this range indicates the presence of extreme values.

We can calculate the skwenss value :

# skewness value  
df[['wage\_per\_hour','capital\_gains','capital\_losses','dividends\_from\_stocks']].skew()

Using the **IQR score**, let's see the number of obseravtions that are not in the (Q1 - 1.5 IQR) and (Q3 + 1.5 IQR) range :

# IQR score  
Q1 = df[['wage\_per\_hour','capital\_gains','capital\_losses','dividends\_from\_stocks']].quantile(0.25)  
Q3 = df[['wage\_per\_hour','capital\_gains','capital\_losses','dividends\_from\_stocks']].quantile(0.75)  
IQR = Q3 - Q1  
print(IQR)

# number of observation out of the definied range  
out = df[['wage\_per\_hour','capital\_gains','capital\_losses','dividends\_from\_stocks']]  
df\_out = out[((out < (Q1 - 1.5 \* IQR)) |(out > (Q3 + 1.5 \* IQR))).any(axis=1)]  
  
out.shape, df\_out.shape, df\_out.shape[0]/out.shape[0]

From 199.523 observation of the selcted features, 38.859 records (19%) represent extrem values.

For **weeks\_worked\_in\_year** :

df['weeks\_worked\_in\_year'].plot( kind='hist',  
 bins=53,  
 edgecolor='black',  
 color='blue',  
 title='weeks worked in a year',  
 figsize=(10,5))

For **num\_persons\_worked\_for\_employer** :

df['num\_persons\_worked\_for\_employer'].plot( kind='hist',  
   
 edgecolor='black',  
 color='blue',  
 title='num\_persons\_worked\_for\_employer',  
 figsize=(7,5))

We notice an increase in the 7th bins **num\_persons\_worked\_for\_employer=6**. Check if this variable is capped ?

# Exploratory data analysis

* Starting with **numerical variables** :

import seaborn as sns

data\_dia = df[dep\_var]  
data = df[['age']]  
data = pd.concat([data\_dia,data],axis=1)  
data = pd.melt(data,id\_vars="income\_level",  
 var\_name="features",  
 value\_name='value')  
plt.figure(figsize=(6,5))  
sns.violinplot(x="features", y="value", hue="income\_level", data=data,split=True, inner="quartile")  
plt.xticks(rotation=90)

For the **age** feature, we can see that the medians of the income levels +/- 50k look separated. The income level of +50k with a median of 50 years old has a lower interquntile range (IQR) with value spread of 10 years. Whereas The income level of -50k has a median of 30 years old has and interquantile range (IQR) of 40 years. So, **age** can be good for classification.

Let's look at the **weeks\_worked\_in\_year** feature :

# get the number of income class in each week  
weeks\_worked\_in\_year = df.groupby(["weeks\_worked\_in\_year", "income\_level"])\  
 .size()\  
 .groupby(level=0).apply(lambda x: 100\*x/x.sum()).unstack()  
  
# print the percentage class for the first and last weeks   
weeks\_worked\_in\_year.iloc[[0,1,2, -3,-2,-1]]

weeks\_worked\_in\_year.plot(kind='bar',   
 stacked=True,  
 edgecolor='black',   
 figsize=(12,5))

We can see that the propotion of people making more than 50k a year is increasing with the number of working weeks in a given year where it can reach more than 14% for those working 52 weeks . However, the -50k level of income is representing the higher propotion regardless of the number of working weeks. We notice that among those how don't work at all, 0.6% still make more than 50k a year.

Let's look at **num\_persons\_worked\_for\_employer** :

num\_persons\_worked\_for\_employer = df.groupby(["num\_persons\_worked\_for\_employer", "income\_level"])\  
 .size()\  
 .groupby(level=0).apply(lambda x: 100\*x/x.sum()).unstack()  
  
num\_persons\_worked\_for\_employer

num\_persons\_worked\_for\_employer.plot(kind='bar',   
 stacked=True,  
 edgecolor='black',   
 figsize=(12,5))

The proportion of +50k income level increases with the number of the num\_preson\_worked\_for\_employer where it reaches **16% for num\_preson\_worked\_for\_employer= 6**.

Let's see the average of **wage\_per\_hour**,**capital\_gains**,**capital\_losses**,**dividends\_from\_stocks** across the income levels :

avg = df[['wage\_per\_hour','capital\_gains','capital\_losses','dividends\_from\_stocks','income\_level']]\  
.groupby('income\_level')\  
.mean()  
  
avg.plot(kind='bar', title = 'average across income levels', figsize=(10,5))  
  
avg

We can see that people making more than 50k a year, have on average, **higher wage per hour**,**higher return on capital asset** and **dividends from stock options**.

* Next, let's analyse some **categorical variables** :

# Education variable  
pd.crosstab(df['income\_level'],   
 df['education'],  
 margins = True,  
 normalize = 'columns').style.format('{:.2%}')

pd.crosstab(df['income\_level'],   
 df['education'],  
 margins = True,  
 normalize = 'columns').plot(kind='bar',stacked=True, edgecolor='black',   
 figsize=(12,10))  
  
plt.legend(bbox\_to\_anchor=(1.5, 1.0))

We can see the effect of education on income level where more than 50% of **Prof school degree** and **Doctorate degree** earn more than 50k a year. On the other hand, the majority of people (more than 90%) with **no degree** earn less than 50k a year.

Let's further this analysis and see the effect of **education** and **the number of working weeks** :

pd.crosstab(df['income\_level'],   
 df['education'],  
 values = df['weeks\_worked\_in\_year'],  
 aggfunc = 'mean').round(2)

import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
plt.figure(figsize=(20, 4))  
sns.heatmap(  
 pd.crosstab(df['income\_level'],   
 df['education'],  
 values = df['weeks\_worked\_in\_year'],  
 aggfunc = 'mean').round(1)   
 ,annot = True  
 ,linewidths=.5  
 ,cmap="YlGnBu"  
   
)  
plt.show()

We can see that earning more than 50k a year demands high level of education but also lot of hard work !

Let's analyse the effect of **sex** and **marital\_stat** on income level :

pd.crosstab(df['income\_level'],   
 [df['sex'],df['marital\_stat']],  
 margins = True,  
 normalize = 'columns').style.format('{:.2%}')

pd.crosstab(df['income\_level'],   
 [df['sex'],df['marital\_stat']]).plot(kind='bar',stacked=True, edgecolor='black',  
   
 figsize=(12,10))

We can see that the highest proportion of people earning less than 50k a year are mostly **female Married-A F spouse present** or **never married** and **seperated male**. On the other hand, **male Married-civilian spouse present** represent the highest propotion on the +50k income level.

We can further the analysis more as we have got many interesting features with several modalities but for now let's see how machine learning models can help us understanding more our data.

# Data preparation

We set **the feature vector** and **the target variable** :

# setting feature vector and target variable for the train set  
  
X = df.drop(['income\_level'], axis = 1)  
y = df['income\_level']  
  
# setting feature vector and target variable for the test set  
test\_x = test.drop(['income\_level'], axis = 1)  
test\_y = test['income\_level']

# Cheching the result  
df.shape, X.shape, y.shape

We will keep the provided **test set** hidden and will use it as a realtime dataset when we make our model on production in order to avoid the risk of **data snooping**.

For that, we will be using a validation set derived from our training set (30%). Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is **train\_test\_split()**.

Since we have an imbalanced dataset, we can't considered purely random sampling methods. For that, we do stratified sampling based on the **income level**.

from sklearn.model\_selection import train\_test\_split  
  
X\_train, X\_val, y\_train, y\_val = train\_test\_split(X,   
 y,   
 test\_size=0.3,   
 random\_state=12,  
 stratify=y)

# Checking the train and validation set  
X\_train.shape, X\_val.shape

# Checking the income level proportion  
y\_train.value\_counts(normalize = True), y\_val.value\_counts(normalize = True)

What if we didn't stratify with respect to income level ?

We can compare the income level proportions in the **overall dataset**, in the **test set** generated with stratified sampling, and in **a test set** generated using purely random sampling.

def income\_cat\_proportions(data):  
 return data["income\_level"].value\_counts() / len(data)  
  
train\_set, test\_set = train\_test\_split(df, test\_size=0.3, random\_state=12)  
  
train\_set, test\_set\_strat = train\_test\_split(df, test\_size=0.3, random\_state=12,stratify=df['income\_level'])  
  
compare\_props = pd.DataFrame({  
 "Overall": income\_cat\_proportions(df),  
 "Stratified": income\_cat\_proportions(test\_set\_strat),  
 "Random": income\_cat\_proportions(test\_set),  
}).sort\_index()  
compare\_props["Rand. %error"] = 100 \* compare\_props["Random"] / compare\_props["Overall"] - 100  
compare\_props["Strat. %error"] = 100 \* compare\_props["Stratified"] / compare\_props["Overall"] - 100

As we can see, the test set generated using stratified sampling has income level proportions almost identical to those in the full dataset, whereas the test set generated using purely random sampling is skewed.

compare\_props

Now that we defined our training set, It’s time to prepare the data for our machine Learning algorithms.

* Data cleaning :

We have seen previously that we don't have any **missing values**. For some cataegorical features, we assumed that the **?** modality is encoded for NaN values.

* Handling Text and Categorical Attributes :

Strating with the target variable **income\_level**, we use **LabelEncoder()** to encode target labels with value between 0 and n\_classes-1 = 1.

We have seen also that we have some ordinal variable as **education** and **year**, so we use **OrdinalEncoder** to encode the categorical features as an integer array. The results in a single column of integers (0 to n\_categories - 1) per feature.

Since the remaining categorical features have several modalities per feature, we use also **OrdinalEncoder** instead of **OneHotEncoder**.

Working with **OneHotEncoder** leads, in our case, to high memory consumption. We can combine **OneHotEncoder** and **PCA** : The benefit in PCA is that combination of N attributes is better than any individual attribute. And the disadvantage is in harder explanation what exactly that PCA component means. Therefore, for this work, we will sacrifice a bit of predictive power to get more understandable model.

# categorical variables encoding  
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder  
  
# For the traget varaible  
le = LabelEncoder()  
y\_train = le.fit\_transform(y\_train) #fit on training set  
y\_val = le.transform(y\_val)   
test\_y = le.transform(test\_y)   
  
  
# For categorical features :  
Or = OrdinalEncoder(handle\_unknown='use\_encoded\_value', unknown\_value = -1)  
for c in cat :   
 X\_train[c] = Or.fit\_transform(np.array(X\_train[c]).reshape(-1,1).astype(str)) #fit on training set  
 X\_val[c] = Or.transform(np.array(X\_val[c]).reshape(-1,1).astype(str))  
 test\_x[c] = Or.transform(np.array(test\_x[c]).reshape(-1,1).astype(str))

# Cheking categorical features encoding  
X\_train[cat].head(3)

# Cheking target feature encoding  
set(y\_train)

* Feature Scaling :

We saw previously that out numerical inputs have different scales like the **weeks\_worked\_in\_year** and **capital\_gains**. We will be using **StandardScaler** since standardization is much less affected by outliers.

from sklearn.preprocessing import StandardScaler  
  
scaler = StandardScaler()  
  
for c in cont:  
 X\_train[c] = scaler.fit\_transform(np.array(X\_train[c]).reshape(-1,1)) # fir on the train set  
 X\_val[c] = scaler.transform(np.array(X\_val[c]).reshape(-1,1))  
 test\_x[c] = scaler.transform(np.array(test\_x[c]).reshape(-1,1))

#checking the standardization  
X\_train[cont].head(3)

So far, we have handled the **categorical columns** and the **numerical columns** :

# Checking the training set  
with pd.option\_context('display.max\_rows', None, 'display.max\_columns', None):   
 display(X\_train.head(3))  
  
X\_train.shape, X\_val.shape, test\_x.shape

# Data modeling

* Selecting a Performance Measure :

**Accuracy** is the simplest way to measure the effectiveness of a classification task, and it's the percentage of correct predictions over all predictions. In other words, in a binary classification task, you can calculate this by adding the number of True Positives (TPs) and True Negatives (TNs) and dividing them by a tally of all predictions made. As with regression metrics, you can measure accuracy for both train and test to gauge **overfitting**.

But, we can get an accuracy of 94%, which sounds pretty good, but it turns out we are always predicting **-50k**! In other words, even if we get high accuracy, it is meaningless unless we are predicting accurately for the least represented class, **+50k**.

For this reasing, we will be using **F1-score**. The **F1-score** is also called the harmonic average of precision and recall because it's calculated like this: 2TP / 2TP + FP + FN. Since it includes both precision and recall metrics, which pertain to the proportion of true positives, it's a good metric choice to use when the dataset is **imbalanced**, and we don't prefer either precision or recall.

* Base model :

Let's start with **Decision tree ensembles**.

A decision tree asks a series of binary (that is, yes or no) questions about the data. After each question the data at that part of the tree is split between a "yes" and a "no" branch. After one or more questions, either a prediction can be made on the basis of all previous answers or another question is required.

We illustarte a tree classification using **4 leaf nodes**.

from sklearn.tree import DecisionTreeClassifier, plot\_tree  
  
m = DecisionTreeClassifier(max\_leaf\_nodes=4, random\_state=14) # to plot the tree classification  
m.fit(X\_train, y\_train)

# to get the class output  
m.classes\_

!pip install pydotplus

!pip install graphviz

The top node represents the **initial model** before any splits have been done, when all the data is in the initial income levels. This is the simplest possible model. It is the result of asking zero questions and will always predict the more represented class which is -50k. We use the **Gini method** to create split points. The strategy is to select each pair of adjacent values as a possible split-point and the point with smaller gini index chosen as the splitting point. In our case, the **capital gains** at 1.47 was choosen first.

Moving down and to the left, this node shows us that there were 130,999 records for income level of -50k where **capital gains** was less than 1.47. The class predicted is -50k in this case. Moving down and to the right from the initial model takes us to the records where **capital gains** was greater than 1.47. The class predicted is +50k in this case where 1370 records have an income of +50k and **capital gains** >0.4

The bottom row contains our leaf nodes: the nodes with no answers coming out of them, because there are no more questions to be answered.

Returning back to the top node after the first decision point, we can see that a second binary decision split has been made, based on asking whether **weeks\_worked\_per\_year** is less than or equal to 0.9. For the group where this is true, the class predicted is -50k with a gini of 0.019 and there are 85,411 records. For the records where this decision is false, the class predicted is -50k with a gini of 0.019, and there are 52,361 records. So again, we can see that the decision tree algorithm has successfully split out more records into two more groups which differ in gini value significantly.

Now, let's run our base model :

m = DecisionTreeClassifier(random\_state=14)  
m.fit(X\_train, y\_train)

We evaluate the model on our validation set using **accuracy**, **recall** and **f1 score** :

from sklearn.metrics import accuracy\_score, f1\_score, recall\_score  
  
# on the train set   
accuracy\_score(y\_train,m.predict(X\_train)) , \  
recall\_score(y\_train,m.predict(X\_train)), \  
f1\_score(y\_train,m.predict(X\_train), average='binary', pos\_label=1)

# on the valid set   
accuracy\_score(y\_val,m.predict(X\_val)) , \  
recall\_score(y\_val,m.predict(X\_val)), \  
f1\_score(y\_val,m.predict(X\_val), average='binary', pos\_label=1)

It's seems that we are doing badly on the validation set. Let's see houw many leaf nodes we got :

m.get\_n\_leaves(), len(X\_train)

Sklearn's default settings allow it to continue splitting nodes until there is only one item in each leaf node. Let's change the stopping rule to tell sklearn to ensure every leaf node contains at least 25 records:

m = DecisionTreeClassifier(min\_samples\_leaf=25,random\_state=14)  
m.fit(X\_train, y\_train)  
  
# on the train set   
accuracy\_score(y\_train,m.predict(X\_train)) , \  
recall\_score(y\_train,m.predict(X\_train)), \  
f1\_score(y\_train,m.predict(X\_train), average='binary', pos\_label=1)

# on the valid set   
accuracy\_score(y\_val,m.predict(X\_val)) , \  
recall\_score(y\_val,m.predict(X\_val)), \  
f1\_score(y\_val,m.predict(X\_val), average='binary', pos\_label=1)

That looks much better. Let's check the number of leaves again:

m.get\_n\_leaves(), len(X\_train)

We got less leaf nodes than before. So, the more we increase the number of leaf nodes, the more is the possibility of **overfitting**.

Building a **decision tree** is a good way to create a model of our data. It is very flexible, since it can clearly handle nonlinear relationships and interactions between variables. But we can see there is a fundamental compromise between how well it generalizes (which we can achieve by creating small trees) and how accurate it is on the training set (which we can achieve by using large trees).

So how do we get the best of both worlds?

* Ensembling :

An an example of an Ensemble method is **Random Forest** : we can train a group of Decision Tree classifiers, each on a different random subset of the training set. The process of subseting the data is called **bagging** done with **max\_samples** hyperparameter ( we set it at 100.00 samples) and the ramdom selection process this called **bootsraping** done by setting **bootstrap = True**.

With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. The remaining sampled are called **out-of-bag (oob) instances** used as **validation set** in the training process and done by setting **oob\_score=True**.

We train a Random Forest classifier **with 50 trees** (each limited to **minimum 5 samples per leaf**). and instead of searching for the very best feature when splitting a node, we searches for the best feature among a random subset of 50% of our initial features.

from sklearn.ensemble import RandomForestClassifier  
  
rf = RandomForestClassifier(n\_estimators = 50, max\_samples=100\_000, max\_features=0.5, min\_samples\_leaf= 5,   
 bootstrap= True,oob\_score = True,random\_state=14)

%%time  
rf.fit(X\_train,y\_train)

# on the train set   
accuracy\_score(y\_train,rf.predict(X\_train)) , \  
recall\_score(y\_train,rf.predict(X\_train)), \  
f1\_score(y\_train,rf.predict(X\_train), average='binary', pos\_label=1)

# on the valid set   
accuracy\_score(y\_val,rf.predict(X\_val)) , \  
recall\_score(y\_val,rf.predict(X\_val)), \  
f1\_score(y\_val,rf.predict(X\_val), average='binary', pos\_label=1)

Looking at what happens to the **oob error rate** as we add more and more trees, we you can see that the improvement levels off quite a bit after around 40 trees:

scores =[]  
for k in range(1, 50):  
 rfc = RandomForestClassifier(n\_estimators = k, max\_samples=100\_000, max\_features=0.5, min\_samples\_leaf= 5,   
 bootstrap= True,oob\_score = True,random\_state=14)  
 rfc.fit(X\_train, y\_train)  
 #y\_pred = rfc.predict(X\_val)  
 #scores.append(accuracy\_score(y\_test, y\_pred)) oob\_score\_  
 oob\_error = 1 - rfc.oob\_score\_  
 scores.append(oob\_error)  
  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
# plot the relationship between K and testing accuracy  
# plt.plot(x\_axis, y\_axis)  
plt.plot(range(1, 50), scores)  
plt.xlabel('Value of n\_estimators for Random Forest Classifier')  
#plt.ylabel('Testing Accuracy')  
plt.ylabel('OOB error rate')

Let's try to improve our model :

We may ask **which columns are the strongest predictors, which can we ignore?**

It's not normally enough just to know that a model can make accurate predictions—we also want to know how it's making predictions. Feature importance gives us insight into this. We can get these directly from sklearn's random forest by looking in the feature\_importances\_ attribute. Here's a simple function we can use to pop them into a DataFrame and sort them:

def rf\_feat\_importance(m, df):  
 return pd.DataFrame({'cols':df.columns, 'imp':m.feature\_importances\_}  
 ).sort\_values('imp', ascending=False)

fi = rf\_feat\_importance(rf, X\_train)  
fi[:14]

The feature importances for our model show that the first few most important columns have much higher importance scores than the rest, with (not surprisingly) **capital\_gains** and **dividends\_from\_stocks** being at the top of the list.

A plot of the feature importances shows the relative importances more clearly:

def plot\_fi(fi):  
 return fi.plot('cols', 'imp', 'barh', figsize=(12,7), legend=False)  
  
plot\_fi(fi[:30]);

The way these importances are calculated is quite simple yet elegant. The feature importance algorithm loops through each tree, and then recursively explores each branch. At each branch, it looks to see what feature was used for that split, and how much the model improves as a result of that split. The improvement (weighted by the number of rows in that group) is added to the importance score for that feature. This is summed across all branches of all trees, and finally the scores are normalized such that they add to 1.

It seems likely that we could use just a subset of the columns by removing the variables of low importance and still get good results. Let's try just keeping those with a feature importance greater than **0.005**:

to\_keep = fi[fi.imp>0.005].cols  
len(to\_keep)

We can retrain our model using just this subset of the columns:

X\_train\_imp = X\_train[to\_keep]  
X\_val\_imp = X\_val[to\_keep]

m = RandomForestClassifier(n\_estimators = 50, max\_samples=100\_000, max\_features=0.5, min\_samples\_leaf= 5,   
 bootstrap= True,oob\_score = True,random\_state=14)  
  
m.fit(X\_train\_imp,y\_train)

# on the train set   
accuracy\_score(y\_train,m.predict(X\_train\_imp)) , \  
recall\_score(y\_train,m.predict(X\_train\_imp)), \  
f1\_score(y\_train,m.predict(X\_train\_imp), average='binary', pos\_label=1)

# on the valid set   
accuracy\_score(y\_val,m.predict(X\_val\_imp)) , \  
recall\_score(y\_val,m.predict(X\_val\_imp)), \  
f1\_score(y\_val,m.predict(X\_val\_imp), average='binary', pos\_label=1)

Our **accuracy is about the same**, but we have **far fewer columns** to study:

len(X\_train.columns), len(X\_train\_imp.columns)

We've found that generally the first step to improving a model is **simplifying it**—48 columns was too many for us to study them all in depth! Furthermore, in practice often a simpler, more interpretable model is easier to roll out and maintain.

This also makes our feature importance plot easier to interpret. Let's look at it again:

plot\_fi(rf\_feat\_importance(m, X\_train\_imp));

Let's see if we have redundent feature in our model by determining their similarities :

Determining Similarity: The most similar pairs are found by calculating the rank correlation, which means that all the values are replaced with their rank (i.e., first, second, third, etc. within the column), and then the correlation is calculated.

import scipy  
from scipy.cluster import hierarchy as hc  
  
def cluster\_columns(df, figsize=(10,6), font\_size=12):  
 corr = np.round(scipy.stats.spearmanr(df).correlation, 4)  
 corr\_condensed = hc.distance.squareform(1-corr)  
 z = hc.linkage(corr\_condensed, method='average')  
 fig = plt.figure(figsize=figsize)  
 hc.dendrogram(z, labels=df.columns, orientation='left', leaf\_font\_size=font\_size)  
 plt.show()  
  
cluster\_columns(X\_train\_imp)

Looking good! This is really not much worse than the model with all the fields. Let's create DataFrames without these columns, and save them:

X\_train\_final = X\_train\_imp # train  
X\_val\_final = X\_val\_imp # valid   
  
test\_x\_final = test\_x[to\_keep] # test set

X\_train\_final.shape , X\_val\_final.shape, test\_x\_final.shape

# Model Assesment :

We have seen the **DecisionTreeClassifier** as our basemodel, then we tried **RandomForestClassifier** and finaly we tried to optimize so we can have less features for better interpretation.

Here is the model metrics on our **validation set** :

models\_metrics = {'DTC': [0.95, 0.42, 0.51],   
 'RF': [0.96, 0.42, 0.53],  
 'RF\_less\_feat': [0.95, 0.41, 0.53]  
   
 }  
df = pd.DataFrame(data = models\_metrics)  
df.rename(index={0:'Accuracy',1:'Recall', 2: 'F1 score'},   
 inplace=True)  
ax = df.plot(kind='bar', figsize = (10,5),   
 color = ['gold', 'lightgreen','lightcoral'],  
 rot = 0, title ='Models performance',  
 edgecolor = 'grey', alpha = 0.5)  
for p in ax.patches:  
 ax.annotate(str(p.get\_height()), (p.get\_x() \* 1.01, p.get\_height() \* 1.0005))  
plt.show()

Based on **F1 score**, we select the **RandomForestClassifier** with 25 features as our best model.

Let's see the **experiment results**\* of this model :

The **precision\_recall\_curve** and **roc\_curve** are useful tools to visualize the **sensitivity-specificty** tradeoff in the classifier. They help inform a data scientist where to set the decision threshold of the model to maximize either sensitivity or specificity. This is called the **operating point** of the model.

from sklearn.metrics import roc\_curve, precision\_recall\_curve, \  
auc, make\_scorer, recall\_score, accuracy\_score, precision\_score, confusion\_matrix

# We create an array of the class probabilites called y\_scores  
y\_scores = m.predict\_proba(X\_val\_imp)[:, 1]  
  
# we enerate the precision-recall curve for the classifier:  
p, r, thresholds = precision\_recall\_curve(y\_val, y\_scores)  
  
# We calculate the F1 scores  
f1\_scores = 2\*r\*p/(r+p)

Let's plot the **decision chart** of our model :

def plot\_precision\_recall\_vs\_threshold(precisions, recalls, thresholds):  
   
 plt.figure(figsize=(8, 8))  
 plt.title("Precision, Recall Scores and F1 scores as a function of the decision threshold")  
 plt.plot(thresholds, precisions[:-1], "b--", label="Precision")  
 plt.plot(thresholds, recalls[:-1], "g-", label="Recall")  
 plt.plot(thresholds, f1\_scores[:-1], "r-", label="F1-score")  
 plt.ylabel("Score")  
 plt.xlabel("Decision Threshold")  
 plt.legend(loc='best')  
   
plot\_precision\_recall\_vs\_threshold(p, r, thresholds)

We can see that the the optimal threshold to achieve the highest F1 score is set at **0.30** with **59% F1-score**.

print('Best threshold: ', thresholds[np.argmax(f1\_scores)])  
print('Best F1-Score: ', np.max(f1\_scores))

Let's creat **an animated confusion matrix** where the users get to choose the **threesholds** and we **dislpay the confusion matrix and recall vs precision curve** :

pip install ipywidgets

import ipywidgets as widgets

import itertools  
def plot\_confusion\_matrix(cm, classes,  
 normalize = False,  
 title = 'Confusion matrix"',  
 cmap = plt.cm.Blues) :  
 plt.imshow(cm, interpolation = 'nearest', cmap = cmap)  
 plt.title(title)  
 plt.colorbar()  
 tick\_marks = np.arange(len(classes))  
 plt.xticks(tick\_marks, classes, rotation = 0)  
 plt.yticks(tick\_marks, classes)  
  
 thresh = cm.max() / 2.  
 for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])) :  
 plt.text(j, i, cm[i, j],  
 horizontalalignment = 'center',  
 color = 'white' if cm[i, j] > thresh else 'black')  
  
 plt.tight\_layout()  
 plt.ylabel('True label')  
 plt.xlabel('Predicted label')  
   
def adjusted\_classes(y\_scores, t):  
 """  
 This function adjusts class predictions based on the prediction threshold (t).  
 Will only work for binary classification problems.  
 """  
 return [1 if y >= t else 0 for y in y\_scores]  
  
def precision\_recall\_threshold(p, r, thresholds, t=0.5):  
 """  
 plots the precision recall curve and shows the current value for each  
 by identifying the classifier's threshold (t).  
 """  
   
 # generate new class predictions based on the adjusted\_classes  
 # function above and view the resulting confusion matrix.  
 y\_pred\_adj = adjusted\_classes(y\_scores, t)  
   
 cm = confusion\_matrix(y\_val, y\_pred\_adj)  
 class\_names = [0,1]  
 plt.figure()  
 plot\_confusion\_matrix(cm,   
 classes=class\_names,   
 title='RF Confusion matrix')  
  
 plt.figure(figsize=(8,8))  
 plt.title("Precision and Recall curve ^ = current threshold")  
 plt.step(r, p, color='b', alpha=0.2,  
 where='post')  
 plt.fill\_between(r, p, step='post', alpha=0.2,  
 color='b')  
   
 plt.xlabel('Recall');  
 plt.ylabel('Precision');  
   
 # plot the current threshold on the line  
 close\_default\_clf = np.argmin(np.abs(thresholds - t))  
 plt.plot(r[close\_default\_clf], p[close\_default\_clf], '^', c='k',  
 markersize=15)

slider = widgets.IntSlider(  
 min=0,  
 max=10,  
 step=1,  
 description='Slider:',  
 value=3 # The best threshhold for our model  
)  
display(slider)

print(f'For this threshold : {slider.value/10}, the confusion matrix is as follow :')  
precision\_recall\_threshold(p, r, thresholds, slider.value/10)

def plot\_roc\_curve(fpr, tpr, label=None):  
 plt.figure(figsize=(8,8))  
 plt.title('ROC Curve')  
 plt.plot(fpr, tpr, linewidth=2, label=label)  
 plt.plot([0, 1], [0, 1], 'k--')  
 plt.axis([-0.005, 1, 0, 1.005])  
 plt.xticks(np.arange(0,1, 0.05), rotation=90)  
 plt.xlabel("False Positive Rate")  
 plt.ylabel("True Positive Rate (Recall)")  
 plt.legend(loc='best')

fpr, tpr, auc\_thresholds = roc\_curve(y\_val, y\_scores)  
print(f'AUC : {auc(fpr, tpr)}') # AUC of ROC  
plot\_roc\_curve(fpr, tpr, 'recall\_optimized')

Now, let's test this model on our **test set** :

accuracy\_score(test\_y,m.predict(test\_x\_final)) , \  
recall\_score(test\_y,m.predict(test\_x\_final)) , \  
f1\_score(test\_y,m.predict(test\_x\_final), average='binary', pos\_label=1)

# Results : Partial dependency and SHAP values

Let's look at **partial dependence plots**.

**Partial dependence** plots try to answer the question: if a row varied on nothing other than the feature in question, how would it impact the dependent variable?

For instance, how does **capital\_gains** and **dividends\_from\_stocks** impact probability of belonging to the +50k income levl, all other things being equal?

from sklearn.inspection import plot\_partial\_dependence  
  
fig,ax = plt.subplots(figsize=(20, 8))  
plot\_partial\_dependence(m, X\_val\_final, ['capital\_gains','dividends\_from\_stocks'], percentiles=(0,1),  
 grid\_resolution=15, ax=ax);

Looking first at the **dividends\_from\_stocks** plot, we can see a nearly linear relationship between capital dividends\_from\_stocks and the probabillity of income level. Same for **capital\_gains** at 5 standad deviation from the mean after reaching a steady state above that.

!pip install shap

import shap

row\_to\_show = 20  
data\_for\_prediction = test\_x\_final.iloc[row\_to\_show] # use 1 row of data here. Could use multiple rows if desired

# Create object that can calculate shap values  
explainer = shap.TreeExplainer(m)  
  
# Calculate Shap values  
shap\_values = explainer.shap\_values(data\_for\_prediction)

The above explanation shows features each contributing to push the model output from the base value (the average model output over the training dataset we passed) to the model output. Features pushing the prediction higher are shown in red, those pushing the prediction lower are in blue

* The base\_value here is 0.062 while our predicted value is 0.0.
* sex = 1 has the biggest impact on increasing the prediction, while
* Weeks\_worked\_im\_year (below the average) and Age (below the average) feature has the biggest effect in decreasing the prediction.

explainer = shap.TreeExplainer(m)  
  
# calculate shap values. This is what we will plot.  
# Calculate shap\_values for all of val\_X rather than a single row, to have more data for plot.  
shap\_values = explainer.shap\_values(test\_x\_final.iloc[:1000,])  
  
# Make plot. Index of [1] is explained in text below.  
shap.summary\_plot(shap\_values[1],test\_x\_final.iloc[:1000,])

For every dot:

* Vertical location shows what feature it is depicting
* Color shows whether that feature was high or low for that row of the dataset
* Horizontal location shows whether the effect of that value caused a higher or lower prediction.

For the **age** variable, the point in the upper left was depicts a person whose age level is less thereby reducing the prediction of income level +50k class by 0.2.

# Conclusion :

In this work, we presented some techniques for dealing with a machine learning project :

* We used Decision Tree ensembles : Random Forest are the easiest to train, because they are extremely resilient to hyperparameter choices and require very little preprocessing. They are very fast to train, and should not overfit if we have enough trees.
* we used the model for feature selection and partial dependence analysis and Shap values, to get a better understanding of our data.

For futur improvements :

* We can try Gradient Boosting machines as in theory are just as fast to train as random forests, but in practice we will have to try lots of different hyperparameters. They can overfit, but they are often a little more accurate than random forests.
* We can try OneHotEncoder with PCA to deal with the multiple modalities on our categorical variables.
* We can creat new features to challenge the model performance.