SUPERVISED MACHINE LEARNING: CLASSIFICATION

COURSE PROJECT:

Finding Donors for Charity using. Census Income Data Set

IBM / COURSERA

IDDI ABDUL AZIZ

INTRODUCTION

Supervised classification is probably the most commonly used machine learning technique. As the name suggests it is “supervised”: It leverages a labelled training set to build up a logic, and then eventually uses that logic on unlabelled data.

The problem it solves is classification. **Classification** is about assigning each training example to a different class. A class is essentially a trait of an individual example (like the species of an animal), with a few special conditions. Specifically, the number of classes are finite and we know and have examples of all the classes we are expected to classify. If we had a class with *zero* training examples we wouldn’t know what to look for when classifying it. Each example belongs to exactly one class. We can’t say that an example belongs to *no* class, similarly, we can’t classify it as *more* than one class either.

1. BRIEF DESCRIPTION OF THE DATA SET AND SUMMARY OF ITS ATTRIBUTES

In this project, I will use a number of different supervised machine learning algorithms to precisely predict individual’s *‘income’* using data collected from the 1994 U.S. Census. I will then choose the best candidate algorithm from preliminary results and further optimize this algorithm to best model the data. My goal with this implementation is to build a model that accurately predicts whether an individual makes more than $50,000.

This sort of task can arise in a non-profit setting, where organizations survive on donations. Understanding an individual’s income can help a non-profit better understand how large of a donation to request, or whether or not they should reach out to begin with.

The dataset for this project originates from the UCI Machine Learning Repository. The dataset was donated by Ron Kohavi and Barry Becker, after being published in the article *“Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid”*. You can find the article by Ron Kohavi online. The data we investigate here consists of small changes to the original dataset, such as removing the *'fnlwgt'* feature and records with missing or ill-formatted entries.

1.1 Data

The modified census dataset consists of approximately 32,000 data points, with each datapoint having 13 features.

1.2 Features

- age: Age
- workclass: Working Class (Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked)
- education_level: Level of Education (Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool)
- education-num: Number of educational years completed
- marital-status: Marital status (Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse)
- Occupation: Work Occupation (Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces)
- relationship: Relationship Status (Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried)
- race: Race (White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black)

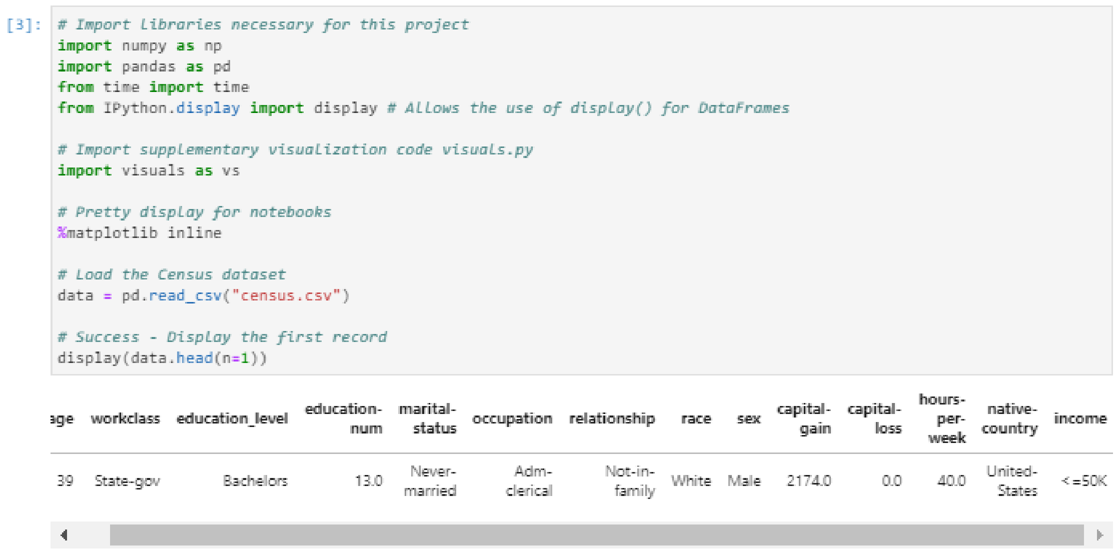
- sex: Sex (Female, Male)
- capital-gain: Monetary Capital Gains
- capital-loss: Monetary Capital Losses
- hours-per-week: Average Hours Per Week Worked.
- Native-country: Native Country (United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holland-Netherlands)

1.3 Target Variable

- Income: Income (<=50K, >50K)

1.4 Importing Libraries and Load Data

I will first load the Python libraries that I am going to use, as well as the census data. The last column is my target variable, 'income', and the rest will be the features.

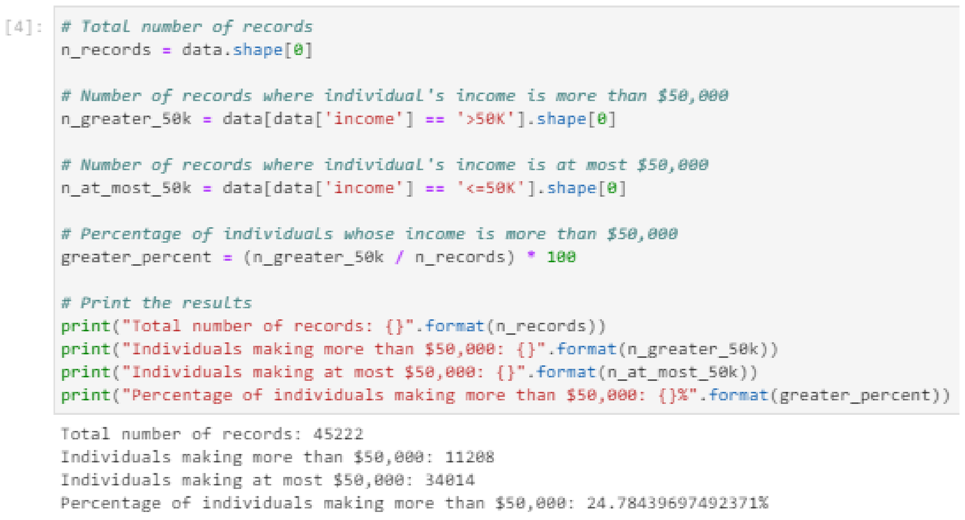


```
[3]: # Import libraries necessary for this project
import numpy as np
import pandas as pd
from time import time
from IPython.display import display # Allows the use of display() for DataFrames

# Import supplementary visualization code visuals.py
import visuals as vs

# Pretty display for notebooks
%matplotlib inline

# Load the Census dataset
data = pd.read_csv("census.csv")

# Success - Display the first record
display(data.head(n=1))
```

age	workclass	education_level	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
39	State-gov	Bachelors	13.0	Never-married	Adm-clerical	Not-in-family	White	Male	2174.0	0.0	40.0	United-States	<=50K

Implementation: Data Exploration

A cursory investigation of the dataset will determine how many individuals fit into either group, and will tell us



```
[4]: # Total number of records
n_records = data.shape[0]

# Number of records where individual's income is more than $50,000
n_greater_50k = data[data['income'] == '>50K'].shape[0]

# Number of records where individual's income is at most $50,000
n_at_most_50k = data[data['income'] == '<=50K'].shape[0]

# Percentage of individuals whose income is more than $50,000
greater_percent = (n_greater_50k / n_records) * 100

# Print the results
print("Total number of records: {}".format(n_records))
print("Individuals making more than $50,000: {}".format(n_greater_50k))
print("Individuals making at most $50,000: {}".format(n_at_most_50k))
print("Percentage of individuals making more than $50,000: {}%".format(greater_percent))
```

```
Total number of records: 45222
Individuals making more than $50,000: 11208
Individuals making at most $50,000: 34014
Percentage of individuals making more than $50,000: 24.78439697492371%
```

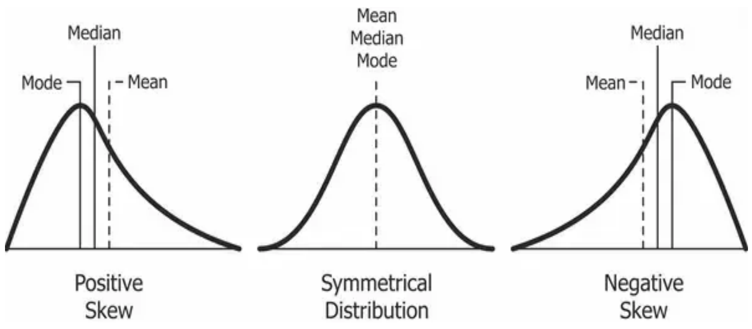
about the percentage of these individuals making more than $50,000.

Preparing the Data

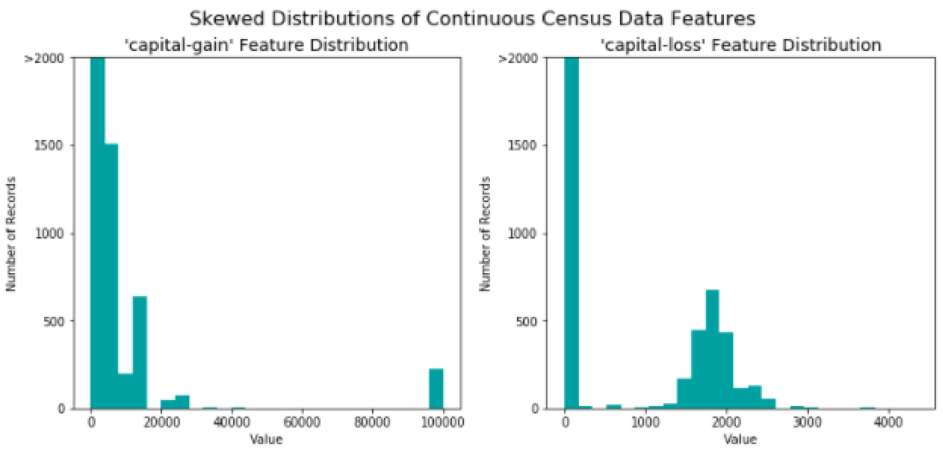
Before data can be used as input for machine learning algorithms, it often must be cleaned, formatted, and restructured — this is typically known as **pre-processing**. Fortunately, for this dataset, there are no invalid or missing entries I will be dealing with, however, there are some qualities about certain features that must be adjusted. This pre-processing can help tremendously with the outcome and predictive power of nearly all learning algorithms.

Transforming Skewed Continuous Features

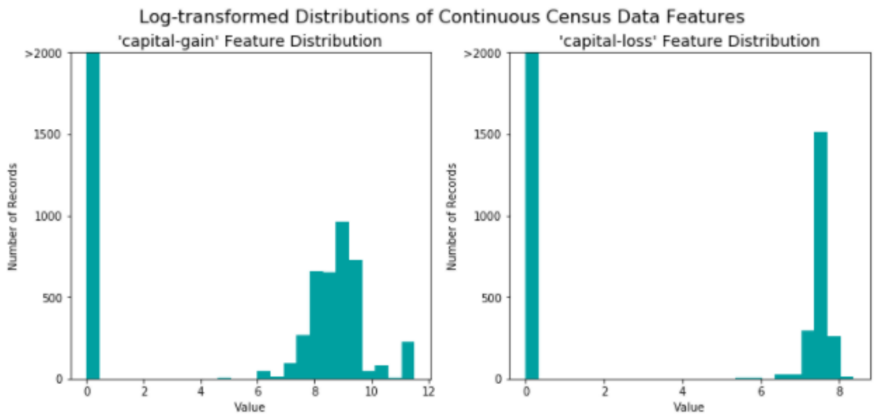
Skewed distributions on the feature's value can make an algorithm to underperform if the range is not normalised.



A dataset may sometimes contain at least one feature whose values tend to lie near a single number, but will also have a non-trivial number of vastly larger or smaller values than that single number. Algorithms can be sensitive to such distributions of values and can underperform if the range is not properly normalized.
With the census dataset two features fit this description: 'capital-gain' and 'capital-loss'.

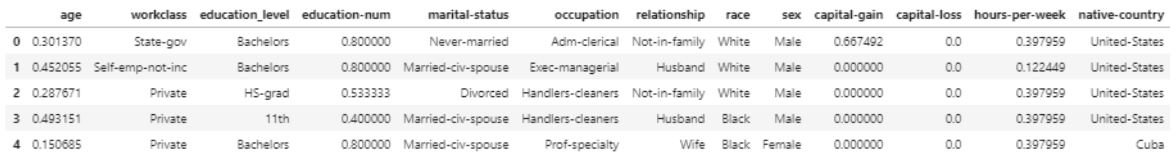


From this visualisation; for highly-skewed feature distributions such as 'capital-gain' and 'capital-loss', it is common practice to apply a [logarithmic transformation](https://en.wikipedia.org/wiki/Data_transformation_(statistics)) on the data, so that the very large and very small values do not negatively affect the performance of a learning algorithm. Using a logarithmic transformation significantly reduces the range of values caused by outliers. However, the logarithm of 0 ‘*(log (0))’* is undefined, so I will translate the values by a small amount above 0 to apply the logarithm successfully.



Normalising Numerical Features

It is recommended to perform some type of scaling on numerical features. This scaling will not change the shape of the features’ distribution, but it ensures the equal treatment of each feature when supervised models are applied.

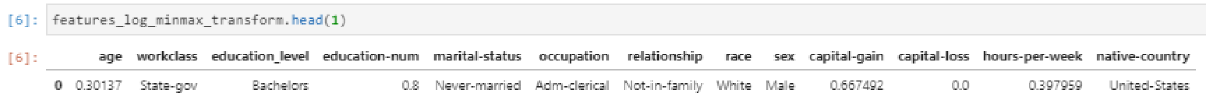


	age	workclass	education_level	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country
0	0.301370	State-gov	Bachelors	0.800000	Never-married	Adm-clerical	Not-in-family	White	Male	0.667492	0.0	0.397959	United-States
1	0.452055	Self-emp-not-inc	Bachelors	0.800000	Married-civ-spouse	Exec-managerial	Husband	White	Male	0.000000	0.0	0.122449	United-States
2	0.287671	Private	HS-grad	0.533333	Divorced	Handlers-cleaners	Not-in-family	White	Male	0.000000	0.0	0.397959	United-States
3	0.493151	Private	11th	0.400000	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0.000000	0.0	0.397959	United-States
4	0.150685	Private	Bachelors	0.800000	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0.000000	0.0	0.397959	Cuba

Pre-processing Categorical Features

Looking at the previous table above, we can see that there some features like *‘occupation’* or *‘race’* that are not numerical, they are categorical. Machine learning algorithms expect to work with numerical values, so it is necessary for these categorical features to be transformed.

One of the most popular categorical transformations schemes is called **‘one-hot encoding’**. One-hot encoding creates a *‘dummy’* variable for each possible category of each non-numeric feature. With that, I will transform the target variable ‘income’ to numerical values for the learning algorithm to work. Since there are only two possible categories for this label *(“<=50K” and “>50K”)*, I can avoid using one-hot encoding and simply encode these two categories as *0 and 1*, respectively.

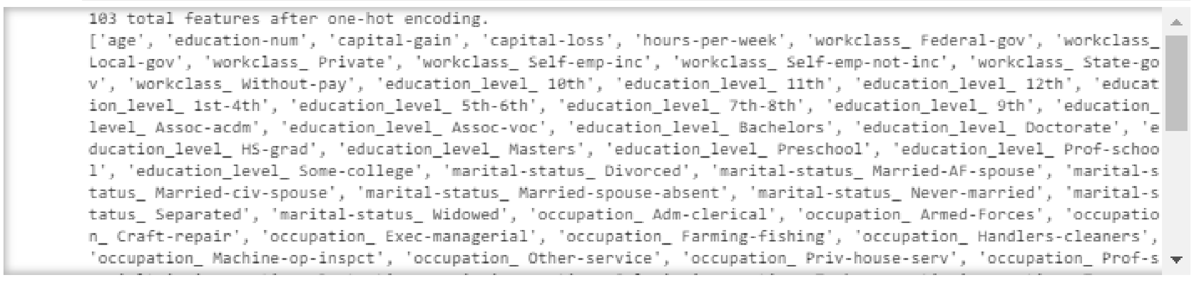


```
[6]: features_log_minmax_transform.head(1)
```

[6]:

	age	workclass	education_level	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country
0	0.30137	State-gov	Bachelors	0.8	Never-married	Adm-clerical	Not-in-family	White	Male	0.667492	0.0	0.397959	United-States

After one-hot encoding I had 103 total features.



```
103 total features after one-hot encoding.
['age', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week', 'workclass_ Federal-gov', 'workclass_
Local-gov', 'workclass_ Private', 'workclass_ Self-emp-inc', 'workclass_ Self-emp-not-inc', 'workclass_ State-go
v', 'workclass_ Without-pay', 'education_level_ 10th', 'education_level_ 11th', 'education_level_ 12th', 'educat
ion_level_ 1st-4th', 'education_level_ 5th-6th', 'education_level_ 7th-8th', 'education_level_ 9th', 'education_
level_ Assoc-acdm', 'education_level_ Assoc-voc', 'education_level_ Bachelors', 'education_level_ Doctorate', 'e
ducation_level_ HS-grad', 'education_level_ Masters', 'education_level_ Preschool', 'education_level_ Prof-schoo
l', 'education_level_ Some-college', 'marital-status_ Divorced', 'marital-status_ Married-AF-spouse', 'marital-s
tatus_ Married-civ-spouse', 'marital-status_ Married-spouse-absent', 'marital-status_ Never-married', 'marital-s
tatus_ Separated', 'marital-status_ Widowed', 'occupation_ Adm-clerical', 'occupation_ Armed-Forces', 'occupatio
n_ Craft-repair', 'occupation_ Exec-managerial', 'occupation_ Farming-fishing', 'occupation_ Handlers-cleaners',
'occupation_ Machine-op-inspct', 'occupation_ Other-service', 'occupation_ Priv-house-serv', 'occupation_ Prof-s
```

Shuffle and Split Data

Now all *categorical variables* have been converted into numerical features, and all numerical features have been normalized. As always, we will now split the data (both features and their labels) into training and test sets. 80% of the data will be used for training and 20% for testing.

```
[12]: # Import train_test_split
      from sklearn.model_selection import train_test_split

      # Split the 'features' and 'income' data into training and testing sets
      X_train, X_test, y_train, y_test = train_test_split(features_final,
                                                          income,
                                                          test_size = 0.2,
                                                          random_state = 0)

      # Show the results of the split
      print("Training set has {} samples.".format(X_train.shape[0]))
      print("Testing set has {} samples.".format(X_test.shape[0]))
```

```
Training set has 36177 samples.
Testing set has 9045 samples.
```

Evaluating Model Performance

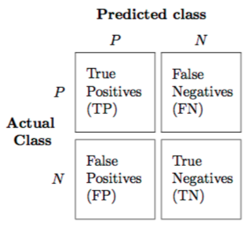
In this section, I will investigate four different algorithms, and determine which is best at modelling the data. Three of these algorithms will be supervised learners, and the fourth algorithm is known as a *naive predictor* which serves as baseline of performance.

Metrics and The Naïve Predictor

The objective of the project is to correctly identify what individuals make more than $50k per year, as they are the group most likely to donate money to charity. Therefore, I wisely choose evaluation metric.

Note: *Recap of accuracy, precision, recall*

When making predictions on events we can get 4 types of results: True Positives, True Negatives, False Positives, and False Negatives. All of these are represented in the following classification matrix:



		Predicted class	
		P	*N*
Actual Class	*P*	True Positives (TP)	False Negatives (FN)
	N	False Positives (FP)	True Negatives (TN)

Accuracy: *Measures how often the classifier makes the correct prediction. It's the ratio of the number of correct predictions to the total number of predictions (the number of test data points)*

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

Precision: *Tells what proportion of events we classified as a certain class, actually were that class. It is a ratio of true positives to all positives.*

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

Recall (sensitivity): *Tells what proportion of events that actually were the of a certain class were classified by us as that class. It is a ratio of true positives to all the positives.*

$$\text{Recall} = \frac{TP}{TP + FN}$$

For classification problems that are skewed like in our case, accuracy by itself is not an appropriate metric. Instead, precision and recall are much more representative. These two metrics can be combined to get the **F1 score**, which is weighted average harmonic mean) of the precision and recall scores. This score can range from 0 to 1, with 1 being the best possible F1 score (we take the harmonic mean as we are dealing with ratios).

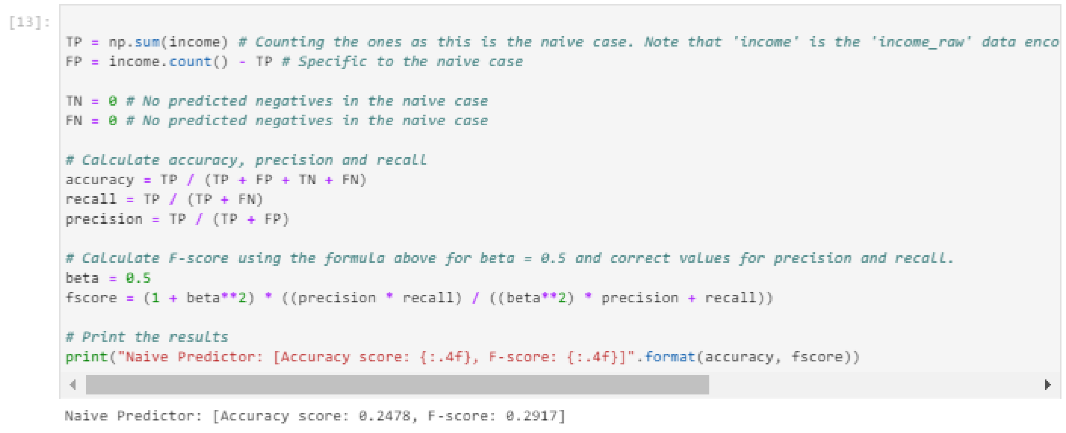
$$F_\beta = \frac{(1+\beta^2)\cdot(\text{precision}\cdot\text{recall})}{(\beta^2\cdot\text{precision}+\text{recall})}$$

In addition, since we are searching individuals willing to donate, the model's ability to precisely predict those that make more than $50,000 is more important than the model's ability to recall those individuals. We can use F-beta score as a metric that considers both precision and recall. Concretely, for β = 0.5, it is given more importance on precision.

If we take a look at the distribution of classes of the ‘income’ variable, it is evident that there are a lot more people that make at most $50K per year than those who make more. So, we could make the naive prediction of taking a random individual, and predicting that he/she will not make more than $50K per year. It is called naive because we haven’t considered any relevant information to substantiate the claim.
This naive prediction will serve us as a benchmark to determine if our model is performing well or not. It is important to notice that it is pointless to use this type of prediction by itself, as all individuals would be classified as not donors.

The Naïve Predictor Performance

If we chose a model that always predicted an individual made more than $50,000, what would that model's accuracy and F-score be on this dataset? The below code determines our naïve predictor performance.



```
[13]:
TP = np.sum(income) # Counting the ones as this is the naive case. Note that 'income' is the 'income_raw' data enco
FP = income.count() - TP # Specific to the naive case

TN = 0 # No predicted negatives in the naive case
FN = 0 # No predicted negatives in the naive case

# Calculate accuracy, precision and recall
accuracy = TP / (TP + FP + TN + FN)
recall = TP / (TP + FN)
precision = TP / (TP + FP)

# Calculate F-score using the formula above for beta = 0.5 and correct values for precision and recall.
beta = 0.5
fscore = (1 + beta**2) * ((precision * recall) / ((beta**2) * precision + recall))

# Print the results
print("Naive Predictor: [Accuracy score: {:.4f}, F-score: {:.4f}]".format(accuracy, fscore))
```

```
Naive Predictor: [Accuracy score: 0.2478, F-score: 0.2917]
```

Supervised Learning Models

Considering the shape of our data:

- ✓ 45222 data points
- ✓ 103 Features

The number of datapoints is not very large but there is a significant number of features and not all supervised algorithms are suitable to handle quantity appropriately.

Some of the available classification algorithms in scikit-learn:

- ✓ Gaussian Naïve Bayes (GassianNB)
- ✓ Decision Trees
- ✓ Ensemble Methods (Bagging, AdaBoost, Random Forest, Gradient Boosting)
- ✓ K-Nearest Neighbors (KNeighbors)
- ✓ Stochastic Gradient Descent Classifier (SGDC)
- ✓ Support Vector Machines (SVM)
- ✓ Logistic Regression

Considering their characteristics and our dataset, I'll choose the following three:

1. **Gaussian Naïve Bayes:**
   - **The strengths** of the model are: It is simple and fast classifier that provides good results with little tuning of the model's hyperparameters. In addition, it does not require a large amount of data to be properly trainied.

   - T**he weaknesses** of the model are: It has a strong feature independence assumption. If we do not have occurrences of a class label and a certain attribute value together (e.g. class = 'nice', shape = 'sphere') then the frequency-based probability estimate will be zero, so given the conditional independence assumption, when all the probabilities are multiplied we will get zero, which will affect the posterior probability estimate.

   - One possible real-world application where this model can be applied is for text learning.
   - It is a good candidate because it is an efficient model and can deal with many features (the data set contains 98 features).

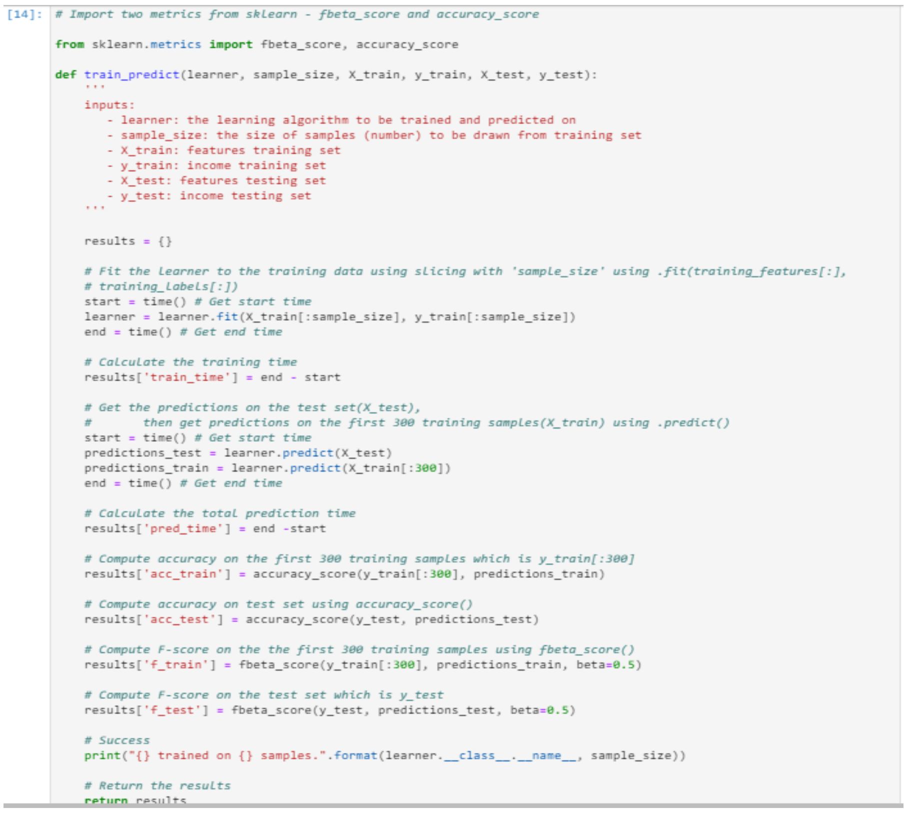
2. **Random Forest:**
   - The **strengths** of the model are: It works well with binary features, as it is an ensembling of decision trees. It does not expect linear features. It works well with high dimensional spaces and large number of training examples.
   - The main **weakness** is that it may overfit when dealing with noisy data.
   - One possible real-world application where this model can be applied is for predicting stock market prices.
   - It is a good candidate because it is often a quite accurate classifier and works well with binary features and high dimensional datasets.

3. **Support Vector Machines Classifier:**
   - The **strengths** of the model are: It works well with no linearly separable data and high dimensional spaces.
   - The main **weakness** is that it may be quite inefficient to train, so it is no suitable for "industrial scale" applications.
   - One possible real-world application where this model can be applied is for classifying people with and without common diseases.

- It is a good candidate because it is often a quite accurate classifier and works well with binary features and high dimensional datasets.

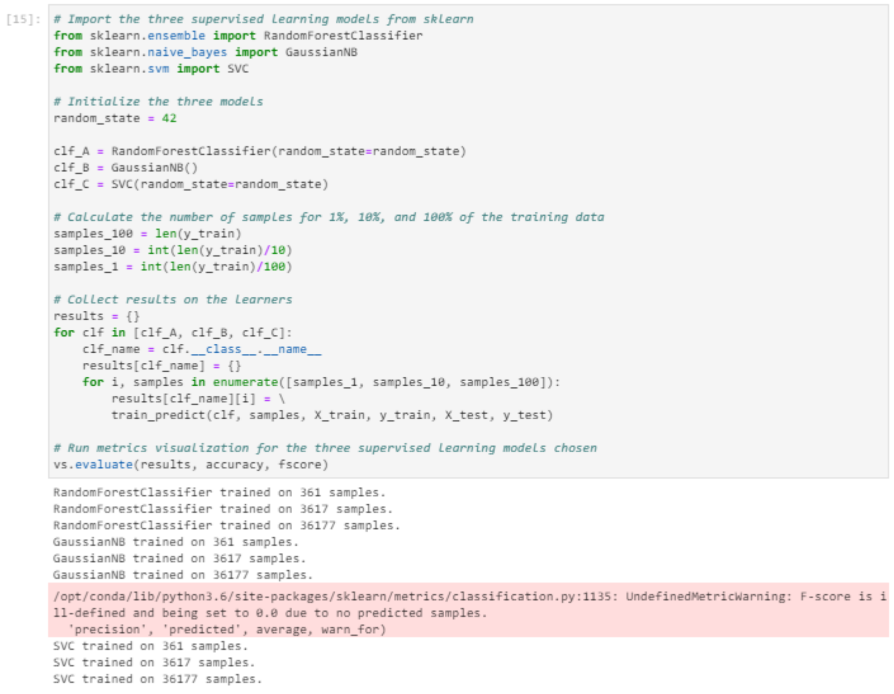
Implementation - Creating a Training and Predicting Pipeline

To properly evaluate the performance of each model, we will create a training and predicting pipeline that will allow us to quickly and effectively train models using various sizes of training data and perform predictions on the testing data.



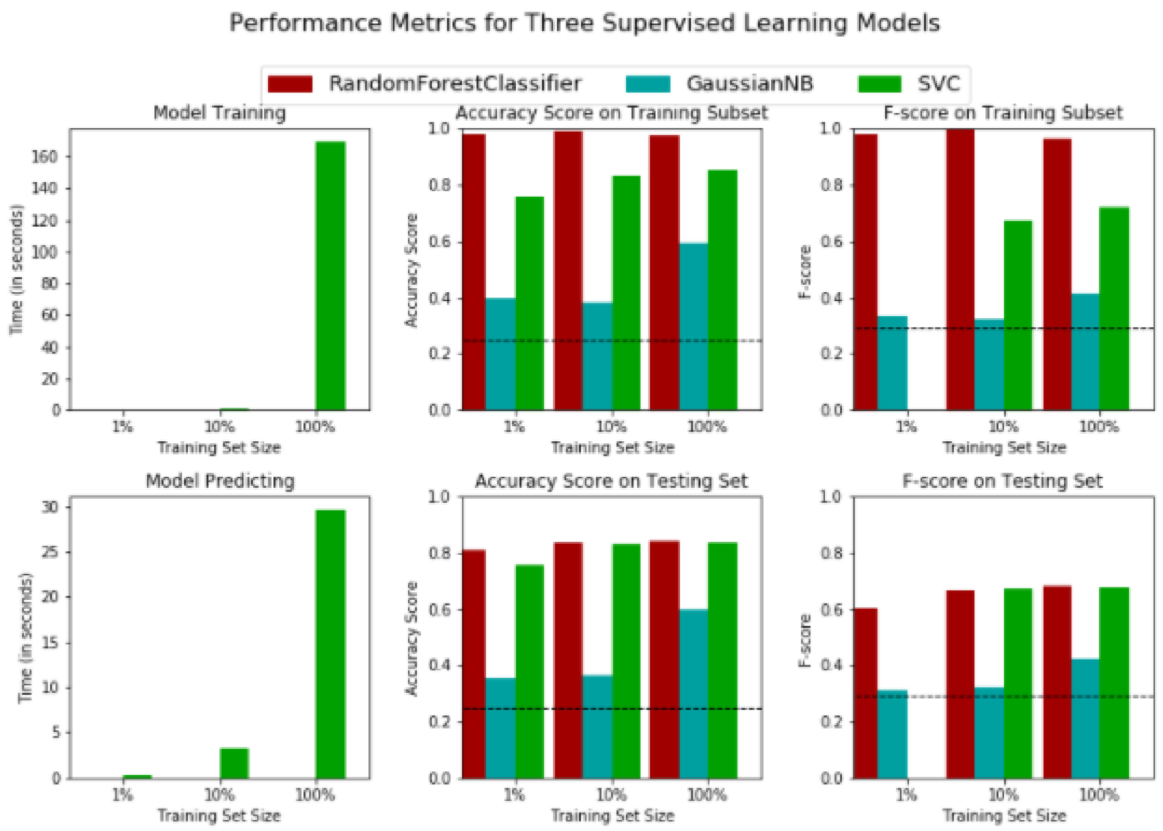
```
[14]: # Import two metrics from sklearn - fbeta_score and accuracy_score

      from sklearn.metrics import fbeta_score, accuracy_score

      def train_predict(learner, sample_size, X_train, y_train, X_test, y_test):
          '''
          inputs:
             - learner: the learning algorithm to be trained and predicted on
             - sample_size: the size of samples (number) to be drawn from training set
             - X_train: features training set
             - y_train: income training set
             - X_test: features testing set
             - y_test: income testing set
          '''

          results = {}

          # Fit the learner to the training data using slicing with 'sample_size' using .fit(training_features[:],
          # training_labels[:])
          start = time() # Get start time
          learner = learner.fit(X_train[:sample_size], y_train[:sample_size])
          end = time() # Get end time

          # Calculate the training time
          results['train_time'] = end - start

          # Get the predictions on the test set(X_test),
          #       then get predictions on the first 300 training samples(X_train) using .predict()
          start = time() # Get start time
          predictions_test = learner.predict(X_test)
          predictions_train = learner.predict(X_train[:300])
          end = time() # Get end time

          # Calculate the total prediction time
          results['pred_time'] = end -start

          # Compute accuracy on the first 300 training samples which is y_train[:300]
          results['acc_train'] = accuracy_score(y_train[:300], predictions_train)

          # Compute accuracy on test set using accuracy_score()
          results['acc_test'] = accuracy_score(y_test, predictions_test)

          # Compute F-score on the the first 300 training samples using fbeta_score()
          results['f_train'] = fbeta_score(y_train[:300], predictions_train, beta=0.5)

          # Compute F-score on the test set which is y_test
          results['f_test'] = fbeta_score(y_test, predictions_test, beta=0.5)

          # Success
          print("{} trained on {} samples.".format(learner.__class__.__name__, sample_size))

          # Return the results
          return results
```

Implementation: Initial Model Evaluation



```
[15]: # Import the three supervised learning models from sklearn
      from sklearn.ensemble import RandomForestClassifier
      from sklearn.naive_bayes import GaussianNB
      from sklearn.svm import SVC

      # Initialize the three models
      random_state = 42

      clf_A = RandomForestClassifier(random_state=random_state)
      clf_B = GaussianNB()
      clf_C = SVC(random_state=random_state)

      # Calculate the number of samples for 1%, 10%, and 100% of the training data
      samples_100 = len(y_train)
      samples_10 = int(len(y_train)/10)
      samples_1 = int(len(y_train)/100)

      # Collect results on the learners
      results = {}
      for clf in [clf_A, clf_B, clf_C]:
          clf_name = clf.__class__.__name__
          results[clf_name] = {}
          for i, samples in enumerate([samples_1, samples_10, samples_100]):
              results[clf_name][i] = \
              train_predict(clf, samples, X_train, y_train, X_test, y_test)

      # Run metrics visualization for the three supervised learning models chosen
      vs.evaluate(results, accuracy, fscore)
```

```
RandomForestClassifier trained on 361 samples.
RandomForestClassifier trained on 3617 samples.
RandomForestClassifier trained on 36177 samples.
GaussianNB trained on 361 samples.
GaussianNB trained on 3617 samples.
GaussianNB trained on 36177 samples.
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWarning: F-score is i
ll-defined and being set to 0.0 due to no predicted samples.
  'precision', 'predicted', average, warn_for)
SVC trained on 361 samples.
SVC trained on 3617 samples.
SVC trained on 36177 samples.
```



Improving Results

Finally, in this section we will choose the best model to use on our data, and then, perform a grid search optimization for the model over the entire training set (X_train and y_train) by tuning parameters to improve the model's F-score.

Choosing the Best Model

Based on the results of the evaluation, the most appropriate model to identify potential donors is the Random Forest Classifier as it yields the same F-score of the Support Vector Classifier but in much less time.
This is coherent with our knowledge of the algorithm as it is a very good choice when dealing with high-dimensional datasets, in other words, datasets with a large number of features.
So, among the evaluated models, this is the more efficient one and best the suited to work with our dataset.

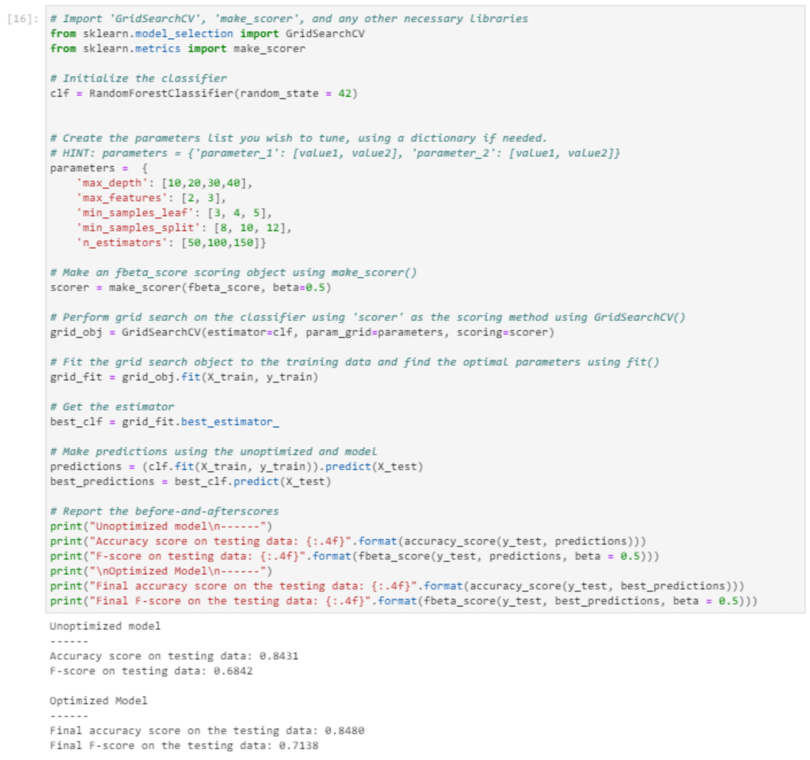
Describing Random Forest in Layman's Terms

To understand Random Forests Classificators we need first to introduce the concept of Decision Trees. A Decision Tree is a flowchart-like structure, in which each internal node represents a test on an attribute of the dataset, each brand represents the outcome of the test and each leaf represents a class label. So, the algorithm will make the tests on the data, finding out which are the most relevant features of the dataset to predict a certain outcome, and separating accordingly the dataset.
A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy of the model, and control overfitting by preventing it to become too complex and unable to generalize on unseen data. It randomly selects a number of features and trains each decision tree classifier in every sub-set of the features. Then, it each decision tree to make predictions by making them vote for the correct label.

Random Forests classifiers are quite popular in classification problems due to its simplicity to use, efficiency and its good predicting accuracy.

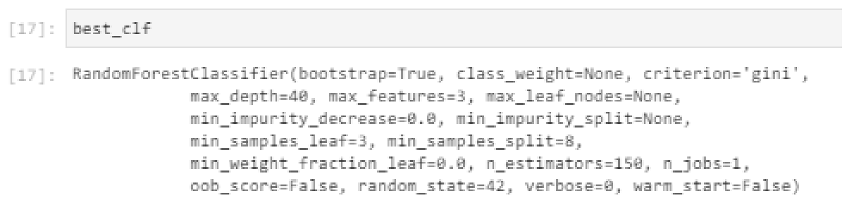
Implementation: Model Tuning

Fine tuning the chosen model, using GridSearchCV



```
[16]: # Import 'GridSearchCV', 'make_scorer', and any other necessary libraries
      from sklearn.model_selection import GridSearchCV
      from sklearn.metrics import make_scorer

      # Initialize the classifier
      clf = RandomForestClassifier(random_state = 42)


      # Create the parameters list you wish to tune, using a dictionary if needed.
      # HINT: parameters = {'parameter_1': [value1, value2], 'parameter_2': [value1, value2]}
      parameters =  {
          'max_depth': [10,20,30,40],
          'max_features': [2, 3],
          'min_samples_leaf': [3, 4, 5],
          'min_samples_split': [8, 10, 12],
          'n_estimators': [50,100,150]}

      # Make an fbeta_score scoring object using make_scorer()
      scorer = make_scorer(fbeta_score, beta=0.5)

      # Perform grid search on the classifier using 'scorer' as the scoring method using GridSearchCV()
      grid_obj = GridSearchCV(estimator=clf, param_grid=parameters, scoring=scorer)

      # Fit the grid search object to the training data and find the optimal parameters using fit()
      grid_fit = grid_obj.fit(X_train, y_train)

      # Get the estimator
      best_clf = grid_fit.best_estimator_

      # Make predictions using the unoptimized and model
      predictions = (clf.fit(X_train, y_train)).predict(X_test)
      best_predictions = best_clf.predict(X_test)

      # Report the before-and-afterscores
      print("Unoptimized model\n------")
      print("Accuracy score on testing data: {:.4f}".format(accuracy_score(y_test, predictions)))
      print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, predictions, beta = 0.5)))
      print("\nOptimized Model\n------")
      print("Final accuracy score on the testing data: {:.4f}".format(accuracy_score(y_test, best_predictions)))
      print("Final F-score on the testing data: {:.4f}".format(fbeta_score(y_test, best_predictions, beta = 0.5)))
```

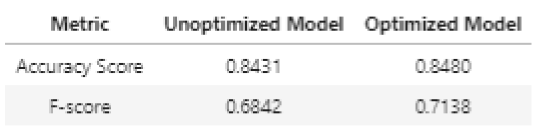
```
Unoptimized model
------
Accuracy score on testing data: 0.8431
F-score on testing data: 0.6842

Optimized Model
------
Final accuracy score on the testing data: 0.8480
Final F-score on the testing data: 0.7138
```



```
[17]: best_clf

[17]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                 max_depth=40, max_features=3, max_leaf_nodes=None,
                 min_impurity_decrease=0.0, min_impurity_split=None,
                 min_samples_leaf=3, min_samples_split=8,
                 min_weight_fraction_leaf=0.0, n_estimators=150, n_jobs=1,
                 oob_score=False, random_state=42, verbose=0, warm_start=False)
```

Final Model Evaluation

Results:



Metric	Unoptimized Model	Optimized Model
Accuracy Score	0.8431	0.8480
F-score	0.6842	0.7138

Observations

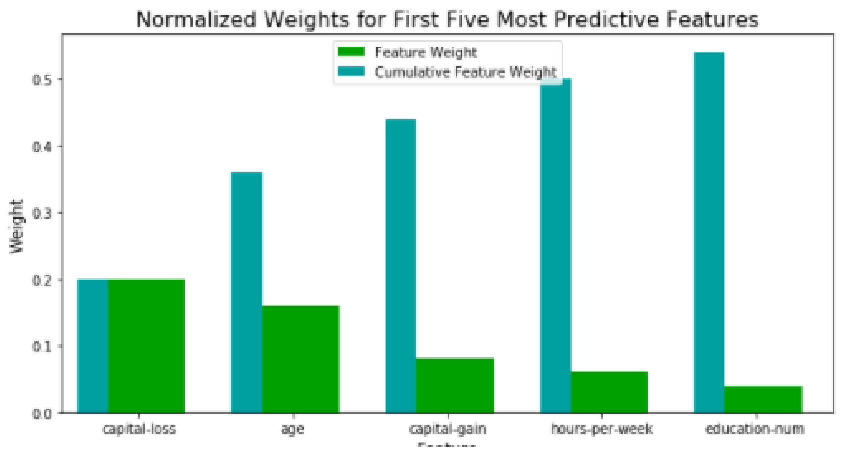
- The optimized model's accuracy and F-score on testing data are: 84.8% and 71.38% respectively.
- These scores are slightly better than the ones of the unoptimized model but the computing time is far larger.
- The naive predictor benchmarks found on Question 1for the accuracy and F-score are 24.78% and 29.27% respectively, which are much worse than the ones obtained with the trained model.

Feature Importance

An important task when performing supervised learning on a dataset like the census data we study here is determining which features provide the most predictive power. By focusing on the relationship between only a few crucial features and the target label we simplify our understanding of the phenomenon, which is most always a useful thing to do. In the case of this project, that means

we wish to identify a small number of features that most strongly predict whether an individual makes at most or more than $50,000.
Intuitively, of the 13 features available on the original dataset I believe that the most important ones for predicting I believe that the 5 ranked most important for predicting the income are:
1) age
2) education
3) native-country
4) occupation
5) hours per week
I would rank them in this order because it is likely that a person's income would increase over the years and older people tend to earn more money than younger ones. In addition, people who have higher education tend to get higher paid jobs and this is a factor that is also heavily related with the native country as, usually, people that are native of economically stronger countries tend to have access to higher education. Occupation is an important feature to take into consideration as yearly income vary a lot depending on the industry and sector that someone works in. And finally, hours per week as normally, people who work more hours tend to earn more.

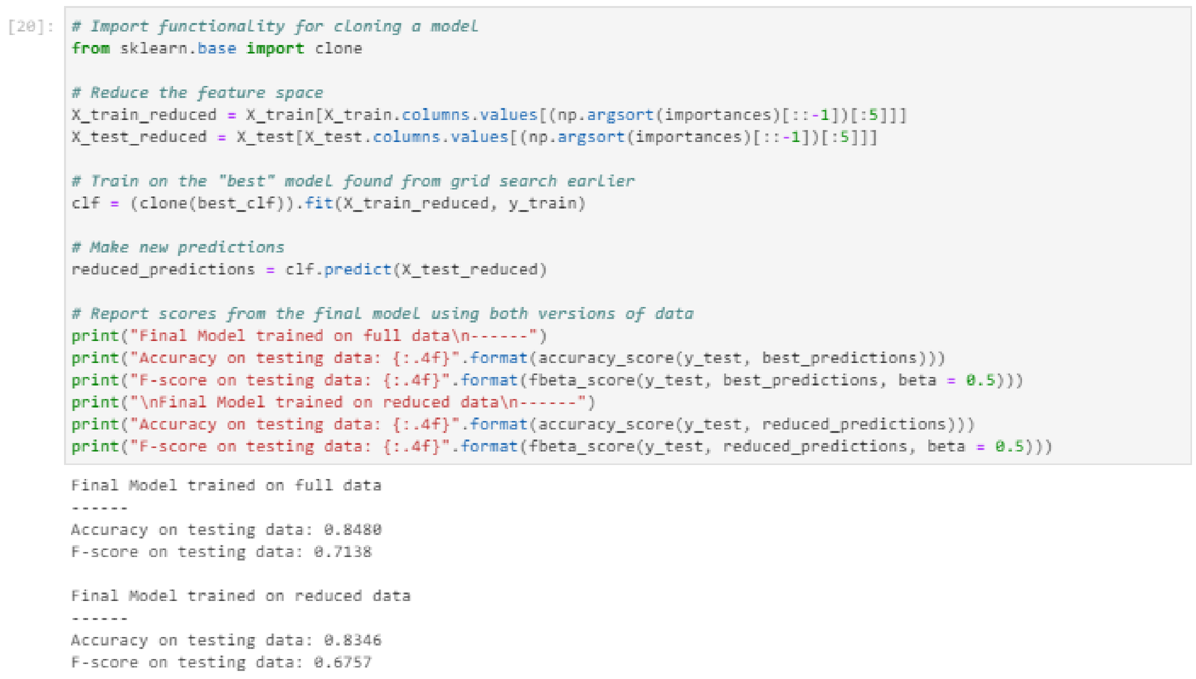
I will now check the accuracy of our logic by passing our data through a model that has the *'feature_importance_method'*



The intuition followed in the previous question was partially right as the AdaBoost tests shows that features like age, hours per week and education are quite relevant to predict the income. However, I didn't identify capital-loss and capital-gain mainly because of ignorance of what those features meant.

Feature Selection

How does a model perform if we only use a subset of all the available features in the data? With less features required to train, the expectation is that training and prediction time is much lower — at the cost of performance metrics. From the visualization above, we see that the top five most important features contribute more than half of the importance of **all** features present in the data. This hints that we can attempt to *reduce the feature space* and simplify the information required for the model to learn.



```
[20]:
# Import functionality for cloning a model
from sklearn.base import clone

# Reduce the feature space
X_train_reduced = X_train[X_train.columns.values[(np.argsort(importances)[::-1])[:5]]]
X_test_reduced = X_test[X_test.columns.values[(np.argsort(importances)[::-1])[:5]]]

# Train on the "best" model found from grid search earlier
clf = (clone(best_clf)).fit(X_train_reduced, y_train)

# Make new predictions
reduced_predictions = clf.predict(X_test_reduced)

# Report scores from the final model using both versions of data
print("Final Model trained on full data\n------")
print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, best_predictions)))
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, best_predictions, beta = 0.5)))
print("\nFinal Model trained on reduced data\n------")
print("Accuracy on testing data: {:.4f}".format(accuracy_score(y_test, reduced_predictions)))
print("F-score on testing data: {:.4f}".format(fbeta_score(y_test, reduced_predictions, beta = 0.5)))
```

```
Final Model trained on full data
------
Accuracy on testing data: 0.8480
F-score on testing data: 0.7138

Final Model trained on reduced data
------
Accuracy on testing data: 0.8346
F-score on testing data: 0.6757
```

Effects of Feature Selection

- Both accuracy and f-score are lower on the reduced data than on the original dataset. Specially f-score 7138% vs 6757%
- Considering the metrics obtained in the evaluations made with: the default model, the optimized model and the reduced dataset, I believe that the best option give the current problem would be to use the default model version with the complete dataset as it yields a good combination of accuracy and f-score in a good training time.

Conclusion

Throughout this article I made a machine learning classification project from end-to-end and we learned and obtained several insights about classification models and the keys to develop one with a good performance.