Census Income Project

In this blog, we will analyze the [Census Dataset](https://archive.ics.uci.edu/ml/datasets/census+income) for Machine Learning Repository.

We use all three sets of controls in our weighting program and "rake" through them 6 times so that by the end we come back to all the controls we used. The term estimate refers to population totals derived from CPS by creating "weighted tallies" of any specified socio-economic characteristics of the population. People with similar demographic characteristics should have similar weights. There is one important caveat to remember about this statement. That is that since the CPS sample is actually a collection of 51 state samples, each with its own probability of selection, the statement only applies within state.

We will use **Logistic Regression** to build the classifier because it is kind of classification dataset.

We will see how to build a practical machine learning project. In general, any machine learning project requires the following steps:

* Defining the problem statement
* Exploratory Data Analysis
* Training the model
* Fine tuning the model
* Save the model

So let’s get started.

**Defining the problem statement**

The data contains anonymous information such as age, occupation, education, working class, etc. *The goal is to train a binary classifier to predict the income which has two possible values ‘>50K’ and ‘<50K’.* There are 48842 instances and 14 attributes in the dataset. The data contains a good blend of categorical, numerical and missing values.

## **Description of fnlwgt (final weight)**

The weights on the Current Population Survey (CPS) files are controlled to independent estimates of the civilian non-institutional population of the US. These are prepared monthly for us by Population Division here at the Census Bureau. We use 3 sets of controls. These are:

1. A single cell estimate of the population 16+ for each state.
2. Controls for Hispanic Origin by age and sex.
3. Controls by Race, age and sex.

First, we will import the required libraries

import numpy as np  
import pandas as pd  
import seaborn as sns  
from matplotlib import pyplot as plt  
import pickle  
from sklearn.pipeline import Pipeline  
from sklearn.metrics import accuracy\_score  
from sklearn.linear\_model import LogisticRegression  
from sklearn.model\_selection import GridSearchCV  
from sklearn.metrics import classification\_report  
from sklearn.metrics import confusion\_matrix  
from sklearn.preprocessing import StandardScaler  
from sklearn.pipeline import FeatureUnion  
from sklearn.model\_selection import cross\_val\_score%matplotlib inline

**Downloading the data**

df=pd.read\_csv('https://raw.githubusercontent.com/dsrscientist/dataset1/master/census\_income.csv')

df.head()

Next, we will explore the data. This is an important step before going building the model.

Exploratory Data Analysis

Let’s get more information about the training data using df.info()

RangeIndex: 32561 entries, 0 to 32560   
Data columns (total 15 columns):   
age 32561 non-null int64   
workClass 30725 non-null object   
fnlwgt 32561 non-null int64   
education 32561 non-null object   
education-num 32561 non-null int64   
marital-status 32561 non-null object   
occupation 30718 non-null object   
relationship 32561 non-null object   
race 32561 non-null object   
sex 32561 non-null object   
capital-gain 32561 non-null int64   
capital-loss 32561 non-null int64   
hours-per-week 32561 non-null int64   
native-country 31978 non-null object   
income 32561 non-null object

**Observations**

* There are **32561** samples in the training dataset
* There are both categorical and numerical columns in the dataset
* The columns **workClass**, **occupation**, **native-country** have missing values

Similarly, for the test dataset

* There are **16281**samples
* There are **no**missing values

Let’s look the numerical and the categorical data with the help of some visualizations.

**Handling Numerical Columns**

We select the numerical columns using the select\_dtypes function.

num\_attributes = df.select\_dtypes(include=['int'])  
print(num\_attributes.columns)['age', 'fnlwgt', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week']

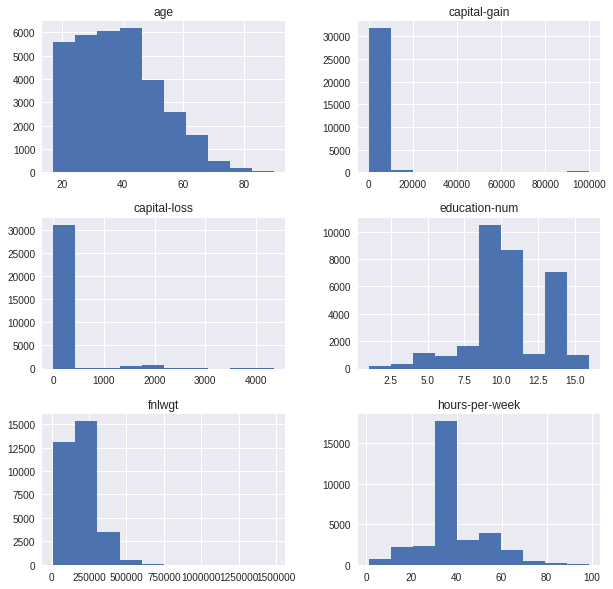
The variables **age**, **hours-per-week** are self-explanatory.

* **fnlwgt**: sampling weight
* **education-num**: number of years of education in total
* **capital-gain/capital-loss**: income from investment sources other than salary/wages

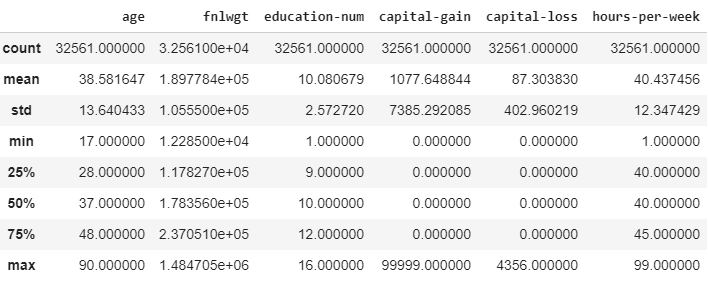
**fnlwgt** is not related to the target variable **income** and will be removed before building the model

**Data Visualizations**

num\_attributes.hist(figsize=(10,10))



More information about the data can be gathered by using df.describe()



**Observations**

* None of the numerical attributes have missing values
* The values are on different scales. Many machine learning models require the values to be on the same scale. We will use [StandardScaler](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html) from the sklearn library to scale the features.

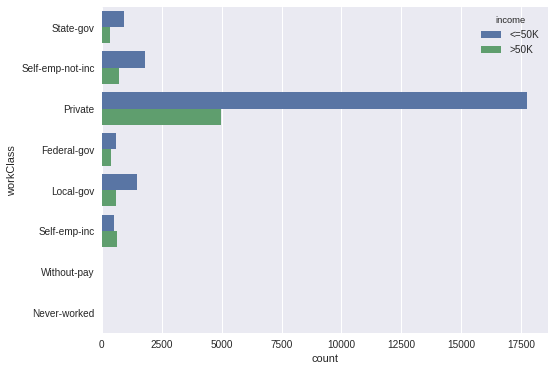
Handling Categorical Columns

cat\_attributes = df.select\_dtypes(include=['object'])  
print(cat\_attributes.columns)['workClass', 'education', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'native-country', 'income']

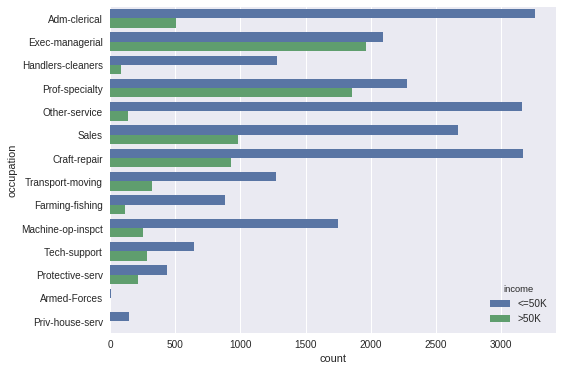
**Data Visualization**

We will use countplot from the seaborn package.

sns.countplot(y='workClass', hue='income', data = cat\_attributes)



sns.countplot(y='occupation', hue='income', data = cat\_attributes)



**Observations**

* The column **education** is just a string representation of the column **education-num**. We will drop the **education** column.
* The ariables **workClass**, **occupation**, **native-country** have missing values. We will replace the missing values in each column with the **most\_frequent**occurring value of that column.

We need to handle the numerical and categorical attributes differently. Numerical attributes need to be scaled, whereas for categorical columns we need to fill the missing values and then encode the categorical values into numerical values. To apply these sequence of transformations we will use the sklearn’s [Pipeline](http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html) class. We will also build custom transformers that can be directly used with Pipeline.

Creating Pipelines

sklearn has many in-built transformers. However, if the in-built ones don’t get the job done for you, you can build a custom transformer. All you need to do is to inherit [BaseEstimator](http://scikit-learn.org/stable/modules/generated/sklearn.base.BaseEstimator.html) and [TransformerMixin](http://scikit-learn.org/stable/modules/generated/sklearn.base.TransformerMixin.html) classes. You also need to implement the **fit** and **transform** methods.

* **fit:**should return an instance of self
* **transform:**the transformation logic can be added here

**ColumnsSelector Pipeline**

sklearn doesn’t provide libraries to directly manipulate with pandas dataframe. We will write it our own custom transformer which will select the corresponding attributes (either numerical or categorical).

class ColumnsSelector(BaseEstimator, TransformerMixin):  
   
 def \_\_init\_\_(self, type):  
 self.type = type  
   
 def fit(self, X, y=None):  
 return self  
   
 def transform(self,X):  
 return X.select\_dtypes(include=[self.type])

**Numerical Data Pipeline**

We select the numerical attributes using the **ColumnsSelector** transformer defined above and then scale the values using the StandardScaler.

num\_pipeline = Pipeline(steps=[  
 ("num\_attr\_selector", ColumnsSelector(type='int')),  
 ("scaler", StandardScaler())  
])

If we call the fit and transform methods for the num\_pipeline it internally calls the fit and transform methods for all the transformers defined in the pipeline. In this case, the ColumnsSelector and StandardScaler transformers.

**Categorical Data Pipeline**

We need to replace the missing values in the categorical columns. We will replace the missing values with the most frequently occurring value in each column. sklearn comes with [Imputer](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Imputer.html) to handle missing values. However, **Imputer** works only with numerical values. We will write a custom transformer which will accept a list of columns for which we need to replace the missing values and the strategy used to fill the missing values

class CategoricalImputer(BaseEstimator, TransformerMixin):  
   
 def \_\_init\_\_(self, columns = None, strategy='most\_frequent'):  
 self.columns = columns  
 self.strategy = strategy  
   
   
 def fit(self,X, y=None):  
 if self.columns is None:  
 self.columns = X.columns  
   
 if self.strategy is 'most\_frequent':  
 self.fill = {column: X[column].value\_counts().index[0] for   
 column in self.columns}  
 else:  
 self.fill ={column: '0' for column in self.columns}  
   
 return self  
   
 def transform(self,X):  
 X\_copy = X.copy()  
 for column in self.columns:  
 X\_copy[column] = X\_copy[column].fillna(self.fill[column])  
 return X\_copy

All the machine learning models expect numerical values. We will use [pd.get\_dummies](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html) to convert the categorical values to numerical values. This is similar to using [OneHotEncoder](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html) except that OneHotEncoder requires numerical columns.

We need to merge the train and test dataset before using pd.get\_dummies as there might be classes in the test dataset that might not be a present in the training dataset. For this, in the fit method, we will a concatenate the train and test dataset and find out all the possible values for a column. In the transform method, we will convert each columns to [Categorical](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.api.types.CategoricalDtype.html) Type and specify the list of categories that the column can take. pd.get\_dummies will create a column of all zeros for the category not present in the list of the categories for that column.

The transformer also takes an argument dropFirst which indicates whether we should drop the first column after creating dummy columns using pd.get\_dummies. We should drop the first column to avoid multicollinearity. By default, the value is set to True

class CategoricalEncoder(BaseEstimator, TransformerMixin):  
   
 def \_\_init\_\_(self, dropFirst=True):  
 self.categories=dict()  
 self.dropFirst=dropFirst  
   
 def fit(self, X, y=None):  
 join\_df = pd.concat([df, test\_data])  
 join\_df = join\_df.select\_dtypes(include=['object'])  
 for column in join\_df.columns:  
 self.categories[column] =   
 join\_df[column].value\_counts().index.tolist()  
 return self  
   
 def transform(self, X):  
 X\_copy = X.copy()  
 X\_copy = X\_copy.select\_dtypes(include=['object'])  
 for column in X\_copy.columns:  
 X\_copy[column] = X\_copy[column].astype({column:  
 CategoricalDtype(self.categories[column])})  
 return pd.get\_dummies(X\_copy, drop\_first=self.dropFirst)

**Complete Categorical Pipeline(fill the data)**

cat\_pipeline = Pipeline(steps=[  
 ("cat\_attr\_selector", ColumnsSelector(type='object')),  
 ("cat\_imputer", CategoricalImputer(columns=  
 ['workClass','occupation', 'native-country'])),  
 ("encoder", CategoricalEncoder(dropFirst=True))  
])

**Complete Pipeline**

We have two transformer pipeline i.e, **num\_pipeline** and **cat\_pipeline**. We can merge them using [FeatureUnion](http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.FeatureUnion.html)

full\_pipeline = FeatureUnion([("num\_pipe", num\_pipeline),   
 ("cat\_pipeline", cat\_pipeline)])

Now we have all the pipelines for building the model, let’s prepare the data for the model and build it.

We will drop the columns that aren’t required

df.drop(['fnlwgt', 'education'], axis=1, inplace=True)  
test\_data.drop(['fnlwgt', 'education'], axis=1, inplace=True)

Training the model

We will create a copy of the training dataset and separate the feature vectors and the target values because it will help us to make a decision based on the dataset.

train\_copy = df.copy()  
train\_copy["income"] = train\_copy["income"].apply(lambda x:0 if   
 x=='<=50K' else 1)X\_train = train\_copy.drop('income', axis =1)  
Y\_train = train\_copy['income']

Next, we pass the X\_train to the full\_pipeline we built.

X\_train\_processed=full\_pipeline.fit\_transform(X\_train)model = LogisticRegression(random\_state=0)  
model.fit(X\_train\_processed, Y\_train)

Testing the model

test\_copy = test\_data.copy()  
test\_copy["income"] = test\_copy["income"].apply(lambda x:0 if   
 x=='<=50K.' else 1)X\_test = test\_copy.drop('income', axis =1)  
Y\_test = test\_copy['income']

We apply the same transformations to the test dataset that we applied to the training dataset.

X\_test\_processed = full\_pipeline.fit\_transform(X\_test)predicted\_classes = model.predict(X\_test\_processed)

Model Evaluation

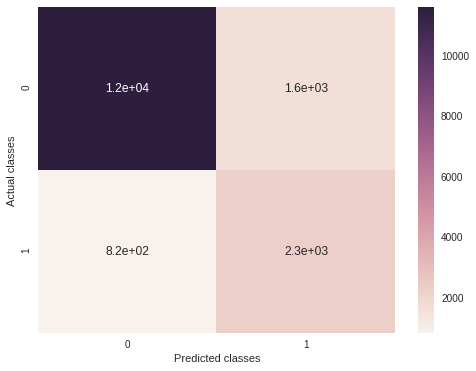
We will use [accuracy\_score](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html) from sklearn to find the accuracy of the model

accuracy\_score(predicted\_classes, Y\_test.values)

The accuracy is 85.2%

Let’s plot the [confusion matrix](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html).

cfm = confusion\_matrix(predicted\_classes, Y\_test.values)  
sns.heatmap(cfm, annot=True)  
plt.xlabel('Predicted classes')  
plt.ylabel('Actual classes')



The X-axis represents the Predicted classes and the Y-axis represents the Actual classes. How do we interpret the confusion matrix? 1.2e+04 times the model correctly predicted the class 0 when the actual class was 0. Similarly, conclusions can be drawn for the remaining cases.

Cross-Validation

We will use [StratifiedKFold](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html) to divide our dataset into k folds. In each iteration, k-1 folds are used as the training set and the remaining fold is used as the validation. We use StratifiedKFold because it preserves the percentage of samples from each class.

If we use [KFold](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html), we might run the risk of introducing sampling bias i.e, the training set might contain a large number of samples where income is greater than 50K and test set contains more samples where income is less than 50K. In this case, the model build from training data will not generalize well for test dataset. Whereas StratifiedKFold will ensure that there are enough samples of each class in both the train and test dataset.

We will use [cross\_val\_score](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html) for cross-validation. The parameter cv determines the cross-validation strategy. StatifiedKFold is used if an integer value is passed to cv

cross\_val\_model = LogisticRegression(random\_state=0)  
scores = cross\_val\_score(cross\_val\_model, X\_train\_processed,   
 Y\_train, cv=5)  
print(np.mean(scores))

We take the mean of all the score obtained in each iteration as the final score of our model.

The accuracy with cross-validation is 85.0%.

Fine Tuning the model

We can fine-tune our model by playing around with the parameters. sklearn comes with [GridSearchCV](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) to do an exhaustive search over specified parameter values for an estimator.

# penalty specifies the norm in the penalization  
penalty = ['l1', 'l2']# C is the inverese of regularization parameter  
C = np.logspace(0, 4, 10)random\_state=[0]# creating a dictionary of hyperparameters  
hyperparameters = dict(C=C, penalty=penalty,   
 random\_state=random\_state)

Using GridSearchCV to find the optimal parameters

clf = GridSearchCV(estimator = model, param\_grid = hyperparameters,   
 cv=5)  
best\_model = clf.fit(X\_train\_processed, Y\_train)  
print('Best Penalty:', best\_model.best\_estimator\_.get\_params() ['penalty'])  
print('Best C:', best\_model.best\_estimator\_.get\_params()['C'])

The best parameters are

Best Penalty: l1   
Best C: 1.0

Predicting using the best model

best\_predicted\_values = best\_model.predict(X\_test\_processed)  
accuracy\_score(best\_predicted\_values, Y\_test.values)

The accuracy of the model with the best parameters is 85.2%

Saving the model

We have done all the hard work of creating and testing the model. It would be good if we could save the model for future use rather than retrain it. We will save our model in the [pickle](https://docs.python.org/2/library/pickle.html).

filename = 'final\_model.sav'  
pickle.dump(model, open(filename, 'wb'))

Loading the model from the pickle

saved\_model = pickle.load(open(filename, 'rb'))

Now we can use the model for prediction purposes.

That’s all for this blog. The complete Jupyter Notebook can be found [here](https://github.com/animesh-agarwal/Machine-Learning-Datasets/blob/master/census-data/census%20income%20logistic%20regression.ipynb).

Final Remarks

We have learned to build a complete machine learning project. In the process, we built custom transformers that can be used with sklearn’s Pipeline class. We also learned to fine-tune our model and save it for further use.