Introduction

In this project, we will use a number of different supervised algorithms to precisely predict individuals' income using *Adult data Set* collected from the UCI machine learning repository. We will then choose the best candidate algorithm from preliminary results and further optimize this algorithm to best model the data. Our goal with this implementation is to build a model that accurately predicts whether an individual makes more than $50,000.

Data

The modified dataset consists of approximately 48,841 data points, with each data point having 15 features.

Target Variable : income (<=50K, >50K)

Import Libraries and Load Data

We will first load the Python libraries that we are going to use, as well as the adult data. The last column will be our target variable, 'income', and the rest will be the features.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import pointbiserialr, spearmanr
from sklearn.feature_selection import SelectKBest
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

data = pd.read_csv("adult.csv")
data
```

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	gain	loss	hours	native-country	income
0	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K
1	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K

Data Analysis

An initial exploration of the dataset like finding the number of records, the number of individuals making more or less than 50k etc, will show us how many individuals fit in each group.

```
n_records = data.shape[0]
n_greater_50k = data[data['income'] == '>50K'].shape[0]
n_at_most_50k = data[data['income'] == '<=50K'].shape[0]
greater_percent = (n_greater_50k / n_records) * 100
```

```
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: 48841
Individuals making more than $50,000: 11687
Individuals making at most $50,000: 37154
Percentage of individuals making more than $50,000: 23.92866648922012%
```

Data Preprocessing

Data must be preprocessed in order to be used in Machine Learning algorithms. This preprocessing phase includes the cleaning and preparing the data.

```
col_names = data.columns
num_data = data.shape[0]
for c in col_names:
    num_non = data[c].isin(["?"]).sum()
    if num_non > 0:
        print (c)
        print (num_non)
        print ("{0:.2f}%".format(float(num_non) / num_data * 100))
```

```
workclass
2799
5.73%
occupation
2809
5.75%
native-country
857
1.75%
```

```
data = data[data["workclass"] != "?"]
data = data[data["occupation"] != "?"]
data = data[data["native-country"] != "?"]
data.shape
```

```
(45221, 15)
```

Missing Value Imputation

Normalization

It is recommended to perform some type of scaling on numerical features. It is used to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
numerical = ['age', 'educational-num', 'gain', 'loss', 'hours']
features_log_minmax_transform = pd.DataFrame(data = features_log_transformed)
features_log_minmax_transform[numerical] = scaler.fit_transform(features_log_transformed[numerical])
display(features_log_minmax_transform.head(n = 5))
```

age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	gain	loss	hours	native-country

	age	workclass	fnlwgt	education	num	marital-status	occupation	relationship	race	gender	gain	loss	hours	country
0	0.287671	Private	89814	HS-grad	0.533333	Married-civ-spouse	Farming-fishing	Husband	White	Male	0.000000	0.0	0.500000	United-States
1	0.150685	Local-gov	336951	Assoc-acdm	0.733333	Married-civ-spouse	Protective-serv	Husband	White	Male	0.000000	0.0	0.397959	United-States
2	0.369863	Private	160323	Some-college	0.600000	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	0.777174	0.0	0.397959	United-States
4	0.232877	Private	198693	10th	0.333333	Never-married	Other-service	Not-in-family	White	Male	0.000000	0.0	0.295918	United-States
6	0.630137	Self-emp-not-inc	104626	Prof-school	0.933333	Married-civ-spouse	Prof-specialty	Husband	White	Male	0.698384	0.0	0.316327	United-States

Min-Max scaling

Preprocess Categorical Features

If we take a look at the output above, we can see that there are some features like ‘occupation’ or ‘race’ that are not numerical, they are categorical. Machine learning algorithms expect to work with numerical values, so these categorical features should be transformed. One of the most popular categorical transformations is called ‘***One-hot encoding***’ as below.

```
features_final = pd.get_dummies(features_log_minmax_transform)
income = income_raw.map({'<=50K':0,'>50K':1})
encoded = list(features_final.columns)
print("{} total features after one-hot encoding.".format(len(encoded)))
encoded
```

```
104 total features after one-hot encoding.

['age',
 'fnlwgt',
 'educational-num',
 'gain',
 'loss',
 'hours',
 'workclass_Federal-gov',
 'workclass_Local-gov',
 'workclass_Private',
 'workclass_Self-emp-inc',
 'workclass_Self-emp-not-inc',
 'workclass_State-gov',
 'workclass_Without-pay',
 'education_10th',
 'education_11th',
 'education_12th',
 'education_1st-4th',
 'education_5th-6th',
```

One Hot Encoding

Feature Selection

Having irrelevant features in your data can decrease the accuracy of the models and make your model learn based on irrelevant features.

```
col_names = data.columns
param=[]
correlation=[]
abs_corr=[]
for c in col_names:
    if c != "income":
        if len(data[c].unique()) <= 2:
            corr = spearmanr(data['income'],data[c])[0]
        else:
            corr = pointbiserialr(data['income'],data[c])[0]
        param.append(c)
```

```
        correlation.append(corr)
        abs_corr.append(abs(corr))
param_df=pd.DataFrame({'correlation':correlation,'parameter':param, 'abs_corr':abs_corr})
param_df=param_df.sort_values(by=['abs_corr'], ascending=False)
param_df=param_df.set_index('parameter')
param_df
```

parameter	correlation	abs_corr
marital-status	-0.437672	0.437672
educational-num	0.332791	0.332791
relationship	-0.253393	0.253393
age	0.237031	0.237031
hours	0.227199	0.227199
gain	0.221033	0.221033
gender	0.215771	0.215771
loss	0.148685	0.148685
education	0.081171	0.081171
race	0.070822	0.070822
occupation	0.049788	0.049788
native-country	0.020106	0.020106
workclass	0.015657	0.015657
fnlwgt	-0.007259	0.007259

Finding Correlation for each feature

```
best_features=param_df.index[0:4].values
print('Best features:\t',best_features)
```

```
Best features:   ['marital-status' 'educational-num' 'relationship' 'age']
```

4 Best Features selected

Performance Evaluation

When making predictions on events we can get 4 types of results: True Positives, True Negatives, False Positives, and False Negatives. The objective of the project is to correctly identify what individuals make more than 50k$ per year, Therefore we should choose wisely our evaluation metric.

```
TP = np.sum(income)
FP = income.count() - TP
TN = 0
FN = 0
accuracy = TP / (TP + FP + TN + FN)
recall = TP / (TP + FN)
precision = TP / (TP + FP)
```

```
precision = TP / (TP + FP)
beta = 0.5
fscore = (1 + beta**2) * ((precision * recall) / ((beta**2) * precision + recall))
print("Naive Predictor: [Accuracy score: {:.4f}, F-score: {:.4f}]".format(accuracy, fscore))
```

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

We can use the F-beta score as a metric that considers both precision and recall.

Splitting the Data

Now when all categorical variables are transformed and all numerical features are normalized, we need to split our data into training and test sets. We split 80% to training and 20% for testing.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(features_final,income,test_size = 0.2, random_state = 0)
print("Training set has {} samples.".format(X_train.shape[0]))
print("Testing set has {} samples.".format(X_test.shape[0]))
```

```
Training set has 36176 samples.
Testing set has 9045 samples.
```

Prediction

Some of the available Classification algorithms in scikit-learn:

- Gaussian Naive Bayes (GaussianNB)
- Decision Trees
- Ensemble Methods (AdaBoost, Random Forest,..)
- Stochastic Gradient Descent Classifier (SGDC)
- Logistic Regression

Gaussian NB and Random Forest Classifier

The strengths of Naive Bayes Prediction are its simple and fast classifier that provides good results with little tunning of the model's hyperparameters whereas a random forest classifier works well with a large number of training examples.

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import fbeta_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB

def train_predict(learner, sample_size, X_train, y_train, X_test, y_test):
    results = {}
    learner = learner.fit(X_train[:sample_size], y_train[:sample_size])
    predictions_test = learner.predict(X_test)
    predictions_train = learner.predict(X_train[:300])

    results['acc_train'] = accuracy_score(y_train[:300], predictions_train)
    results['acc_test'] = accuracy_score(y_test, predictions_test)
    results['f_train'] = fbeta_score(y_train[:300], predictions_train, beta=0.5)
```

```
    results['f_train'] = fbeta_score(y_train[:300], predictions_train, beta=0.5)
    results['f_test'] = fbeta_score(y_test, predictions_test, beta=0.5)
    print("{} trained on {} samples.".format(learner.__class__.__name__, sample_size))
    return results


random_state = 42
clf_A = RandomForestClassifier(random_state=random_state)
clf_B = GaussianNB()

samples_100 = len(y_train)
samples_10 = int(len(y_train)/10)
samples_1 = int(len(y_train)/100)


results = {}
for clf in [clf_A, clf_B]:
    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)
```

Calculate the number of samples for 1%, 10%, and 100% of the training data

```
RandomForestClassifier trained on 361 samples.
RandomForestClassifier trained on 3617 samples.
RandomForestClassifier trained on 36176 samples.
GaussianNB trained on 361 samples.
GaussianNB trained on 3617 samples.
GaussianNB trained on 36176 samples.
```

Model tuning and evaluating the performance:

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
clf = RandomForestClassifier(random_state = 42)
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]}

scorer = make_scorer(fbeta_score, beta=0.5)
grid_obj = GridSearchCV(estimator=clf, param_grid=parameters, scoring=scorer)
grid_fit = grid_obj.fit(X_train, y_train)
best_clf = grid_fit.best_estimator_
predictions = (clf.fit(X_train, y_train)).predict(X_test)
best_predictions = best_clf.predict(X_test)

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)))
```

```
/home/sushmitha/Desktop/AI/lib/python3.6/site-packages/sklearn/model_selection/_split.py:1978: FutureWarning: The d
efault value of cv will change from 3 to 5 in version 0.22. Specify it explicitly to silence this warning.
  warnings.warn(CV_WARNING, FutureWarning)
/home/sushmitha/Desktop/AI/lib/python3.6/site-packages/sklearn/ensemble/forest.py:245: FutureWarning: The default v
alue of n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
Unoptimized model
------
Accuracy score on testing data 0.8401
F-score on testing data: 0.6860

Optimized Model
------
Final accuracy score on the testing data: 0.8449
Final F-score on the testing data: 0.7120
```

Logistic Regression

```
from sklearn.linear_model import LogisticRegression
logistic = LogisticRegression(solver = 'newton-cg')
logistic.fit(X_train, y_train)

y_pred_logistic=logistic.predict(X_test)
y_train_score_logistic=logistic.predict(X_train)

print("accuracy of the model is:\nTest ", accuracy_score(y_test, y_pred_logistic, normalize=False, sample_weight=Non
print('Train',accuracy_score(y_train, y_train_score_logistic, normalize=False, sample_weight=None))
```

```
accuracy of the model is:
Test  7602
Train 30538
```

Decision Tree Classifier

```
from sklearn import tree
clf = tree.DecisionTreeClassifier(max_depth=10)
clf = clf.fit(X_train, y_train)
```

```
y_pred_dt=clf.predict(X_test)
y_train_score_dt=clf.predict(X_train)
```

```
print("accuracy of the model is:\nTest ", accuracy_score(y_test, y_pred_dt, normalize=True, sample_weight=None))
print('Train',accuracy_score(y_train, y_train_score_dt, normalize=True, sample_weight=None))
```

```
accuracy of the model is:
Test  0.8531785516860144
Train 0.8659608580274215
```

Max depth of the tree is 10

Gradient Descent Classifier

```
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import GradientBoostingClassifier
gdc_model= GradientBoostingClassifier(learning_rate=0.01,random_state=41)
gdc_model.fit(X_train, y_train)
y_pred_gdc=gdc_model.predict(X_test)
y_train_score_gdc=gdc_model.predict(X_train)
print("accuracy of the model is:\nTest ", accuracy_score(y_test, y_pred_gdc, normalize=True, sample_weight=None))
print('Train',accuracy_score(y_train, y_train_score_gdc, normalize=True, sample_weight=None))
```

```
accuracy of the model is:
Test  0.8405749032614704
Train 0.8417459088898718
```

Observations

- GaussianNB: 84.01%
- Decision Trees: 85.31%
- Random Forest: 84.49%
- GDC: 84.05%
- Logistic Regression : 76.02%

The optimized model's accuracy and F-score on testing data on random forest classifier are 84.49% and 71.20% respectively. These scores are slightly better than the ones of the unoptimized model but the computing time is far larger. The naive predictor benchmarks for the accuracy and F-score are 24.78% and 29.17% respectively, which are much worse than the ones obtained with the trained model.

Conclusion

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