Data and Machine Learning

German Loan Prediction

I. Introduction and Objectives

We have initially a csv data file with historical records of defaulted and non-defaulted loans. The aim of this coursework is to find the best possible prediction model (i.e. to predict loan default given characteristics) based on 1000 samples across 20 features.

We first load the data and try to extract some basic information. We follow the below steps:

II.A Data-processing: Preparing the data for Machine Learning

1 - Load the data into a Pandas dataframe

```
your_path = "Relevant path"
df = pd.read_csv(your_path)
```

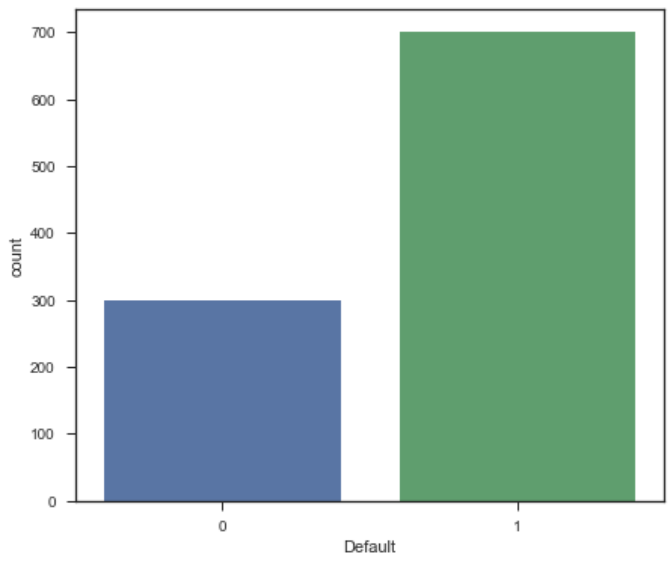
2 - Examine the dataframe for missing data and decide how to treat these

```
print(df.isnull().any().any())
print(df.info())
```

There are no data missing nor purely categorical (literal/string) data so we continue. We tried one hot encoding though creating dummies variables for some of the features like Sex & Marital Status but it did not improve the accuracy of our model anyhow, so we decided to use the rough data set.

3 – Visualize the target distribution

```
plt.figure(figsize=(7,6))
sns.set(style="ticks", color_codes=True)
sns.countplot(x="Default", data=df);
```



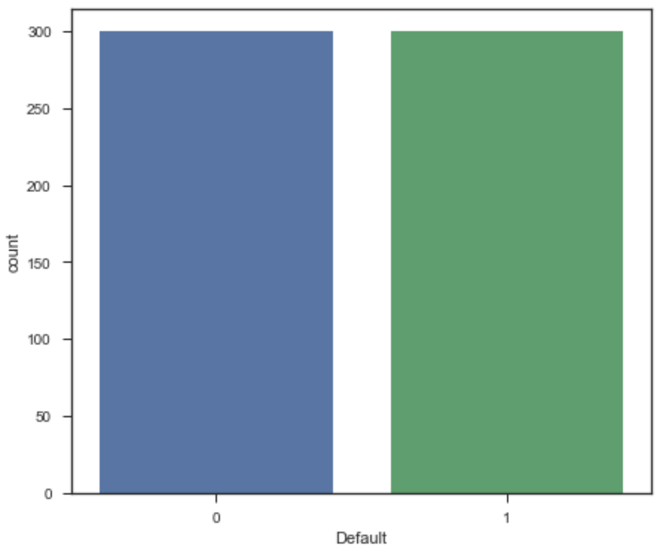
4 – Let's stop and think about the unbalanced data set:

Our loan data are very unbalanced: 700 defaults for 300 non-defaults. Then, we may want to transform our dataset by selecting only a portion of the 700 non-defaults to not "overtrain" the models we are about to use. Indeed, the model (and the selected model) would then focus more on the negative class rather than the positive class.

Note: In the first place, we were expecting a data containing more non-defaults than defaults, as non-defaults are much more likely to happen. In this case, rebalancing the data is even more essential, as the false positives have less impact than the false negatives in loan detections.

Let's look at the process to transform our data set:

```
#Let's shuffle the non_default dataframe
shuffle(default)

#Drop a certain amount of rows in the non_default dataframe
default = default.drop(default.index[0:400])

#Merge the two dataframes to create our main dataframe
df = default.append(non_default)
df = df.reset_index(drop = True)

# Look balanced data set
plt.figure(figsize=(7,6))
sns.set(style="ticks", color_codes=True)
sns.countplot(x="Default", data=df);
```



2 – Let's finally standardize our dataset:

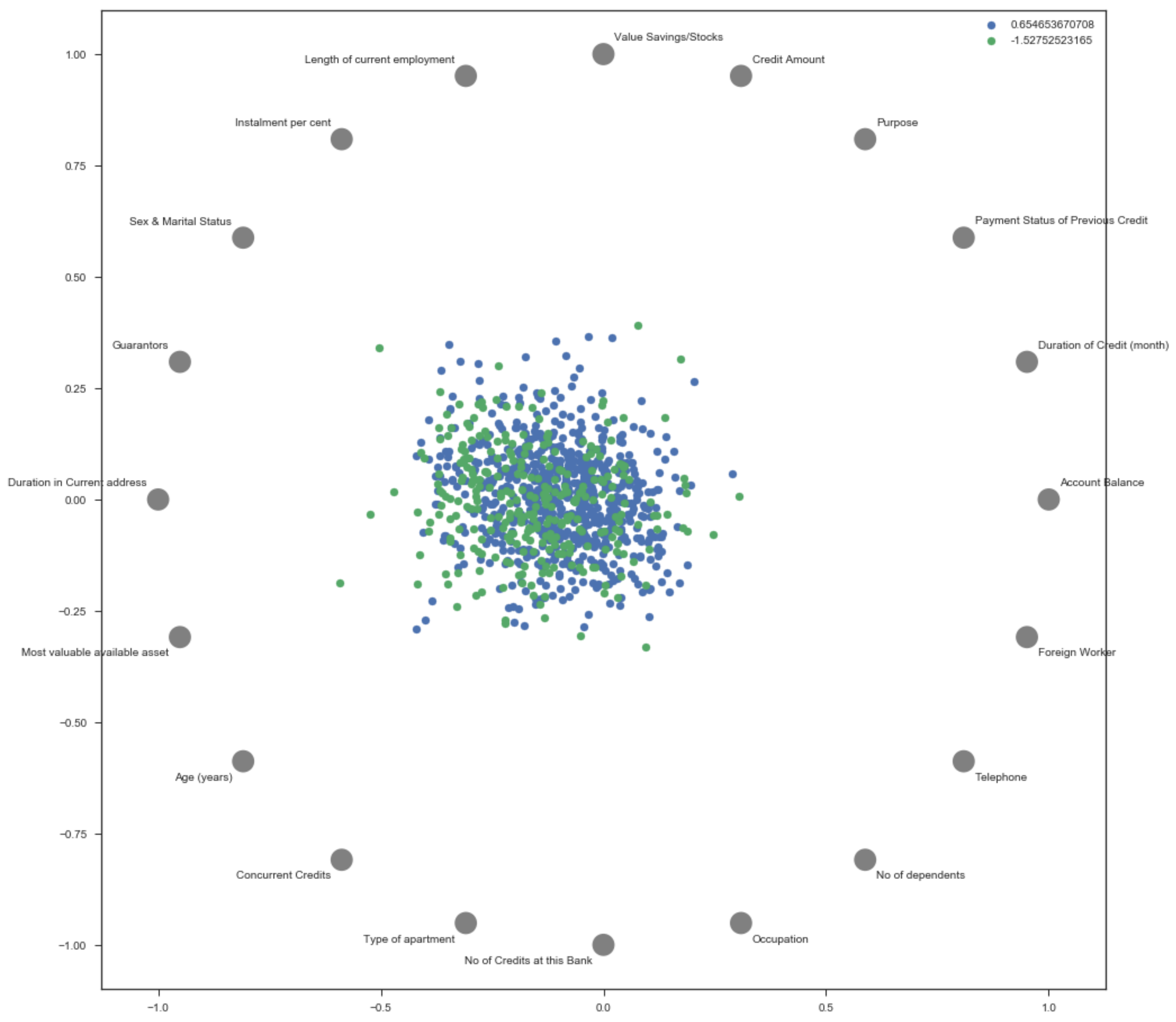
Eventually, we do not forget to apply standardization techniques. As the range of values in our initial dataset can be very heterogeneous, we can use feature scaling. We want each feature to contribute approximatively proportionally. We choose to use rather standardization than MinMax scaling as the method is usually less affected by outlier values.

```
# Standardize the data in a new DF
from sklearn.preprocessing import StandardScaler
Scaler = StandardScaler()
scaled_data = Scaler.fit_transform(df)
df_scaled = pd.DataFrame(scaled_data , columns=df.columns)
```

```
6.  X=df_scaled.values
```

II.1: Which features are important?

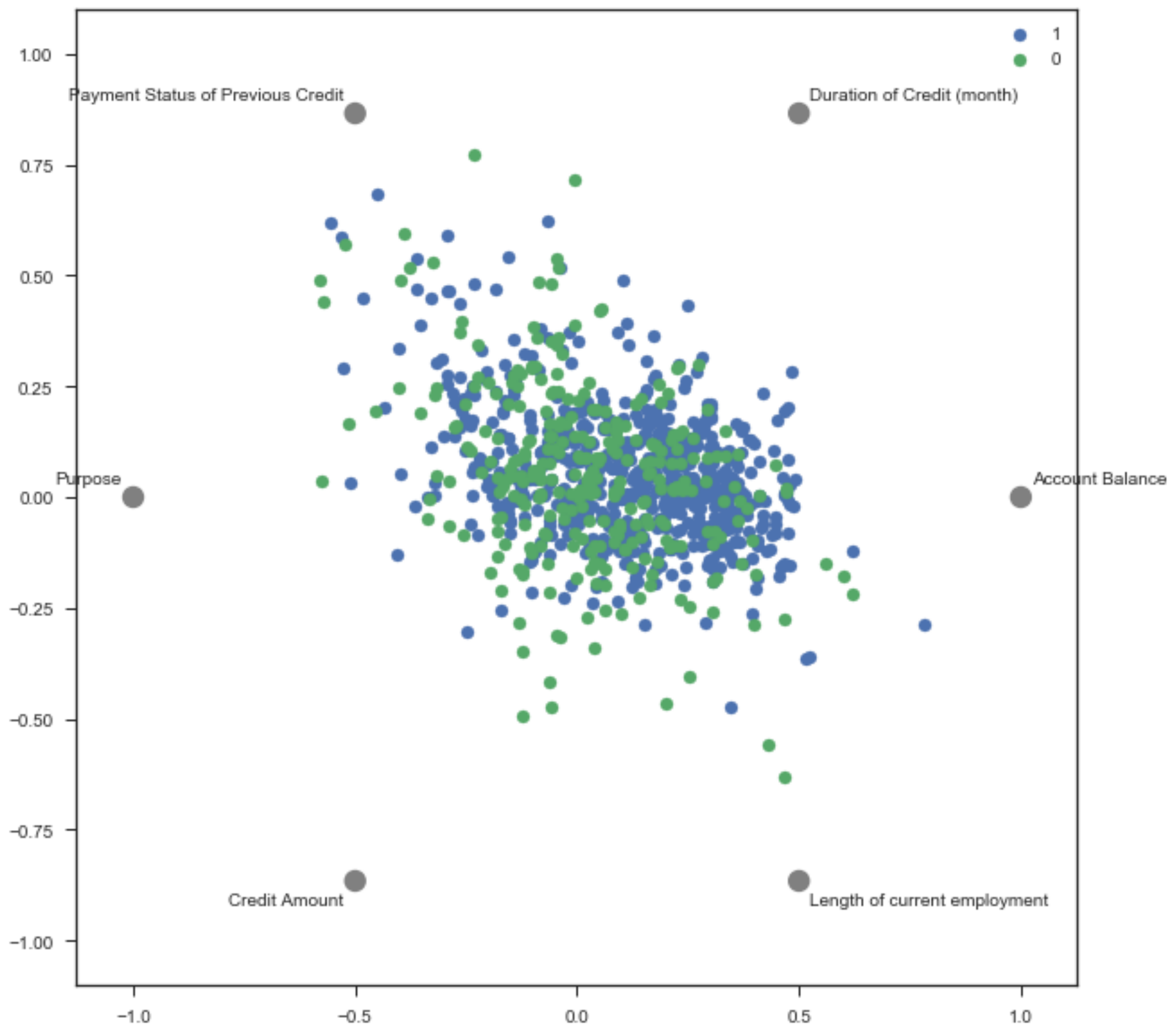
II.1.1: Defaults vs Non-Defaults Profiles



This setting is a simple spring tension minimization visualization of equally spaced point on a unit circle, each point representing a single attribute. Each sample in the data set is attached to each of these points by a spring, the stiffness of which is proportional to the numerical value of that attribute (they are normalized to unit interval). More information to be found on https://cran.r-project.org/web/packages/Radviz/vignettes/single_cell_projections.html

The green points are the defaulted loans and the blue one the non-defaulted. In relation to our dataset it is difficult to deduct towards which features the defaulted or the non-defaulted class tends even if the defaulted class seems a little bit more clustered than the other.

Reducing the number of features to deal with we find something clearer:



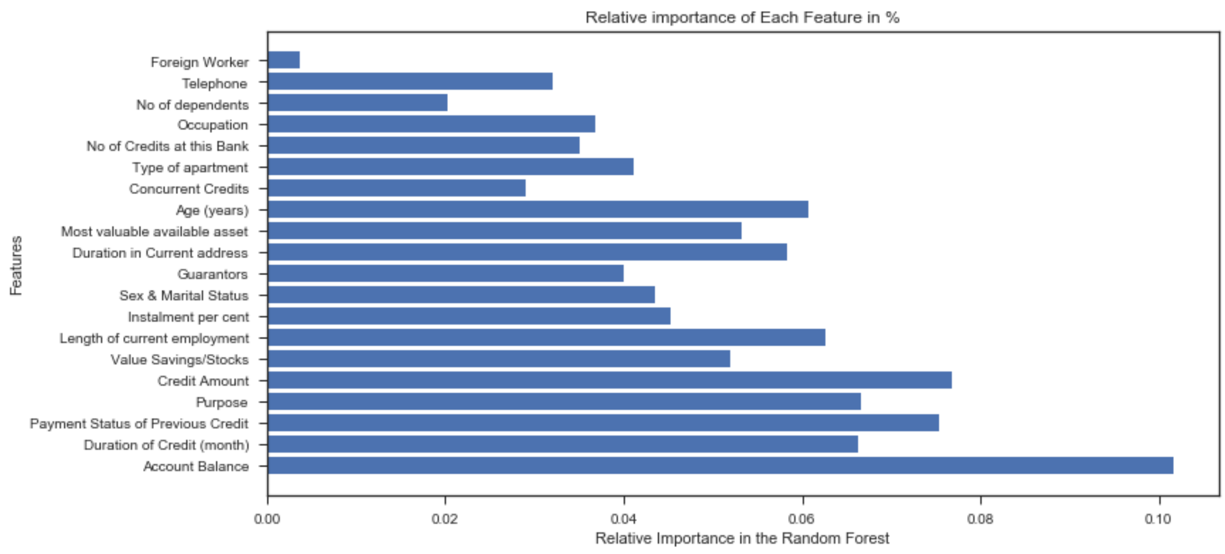
This time, with less features (we chose knowing that some features had more explanative power) we observe that – if we do not take into account to many "outliers" that the defaulted points are more profiled and clustered towards the length of current employment, the account balance and the duration of credit in month while the class 0 (the non-defaulted is more clustered toward the payment status of previous credit, the purpose and the credit amount).

II.1.2: Extremely randomized trees: which features has the best explanatory variable?

Now that we have transformed our data set, we can try to see which features have the best explanatory power. One way to do it is to use an Extra Trees Classifier algorithm. Let's have a look at the principles:

The approach comes from the meta-algorithm family (ensemble learning). The basic principle is to rely on a majority vote (better to ask a group of people than rely on only one person). One way to implement that idea is to run multiple models (same type of models) on different subset of the initial training data sample. That is known as we have seen as the bagging "concept".

Extra Trees Classifier averages predictions obtained on a number of randomized decision trees on the sub-samples. It is computationally easy to run, can control over-fitting. The algorithm is known for its accuracy on other datasets.



We can therefore give a ranking of the five most important features to determine the default: 1: The Account Balance, 2: The Credit Amount, 3: The Payment Status of Previous Credit, 4: Purpose, 5: Duration of Credit (month).

Based on that approach, we could have decided which explanatory variable we want to keep, in order to get the best possible AUC. **We built a for loop to determine the number and the names of the features it is relevant to keep. As it there is no clear pattern to drop some features, we have decided to keep all the columns of our dataframe. (We tried to drop some of them but without really improving the accuracy of our model).**

Another method to reduce the features dataset would have been to run a Principle Component Analysis - reduce dimensionality by building linear combinations of the initial features that are uncorrelated and dropping those that have the less explanatory power (lowest eigenvalue). We have not found satisfying results using that method.

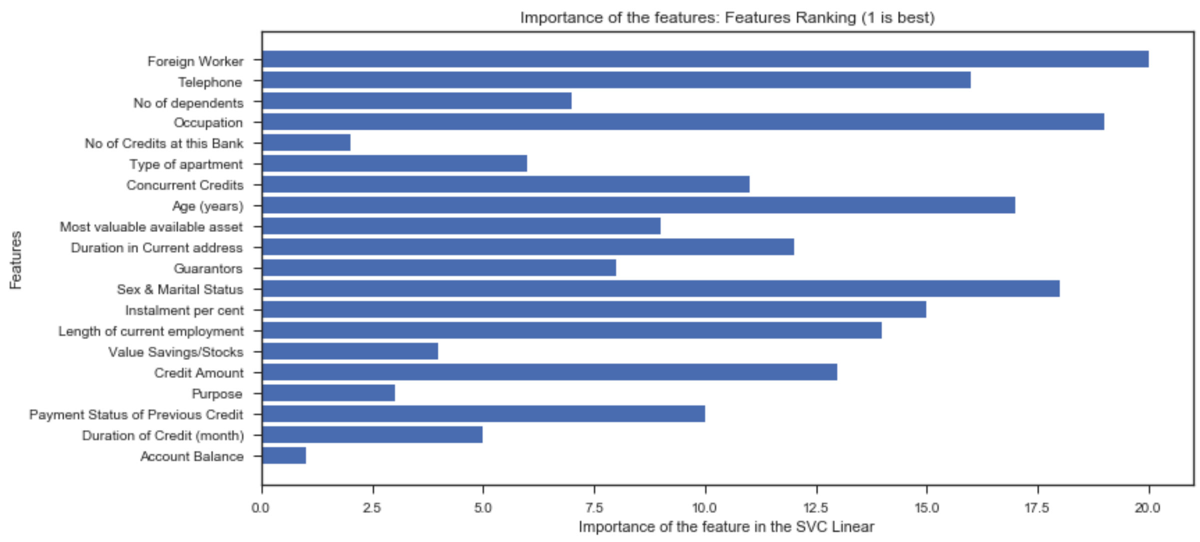
Let's now try to see which features "really matter" using the Recursive Features Elimination method on some interesting models.

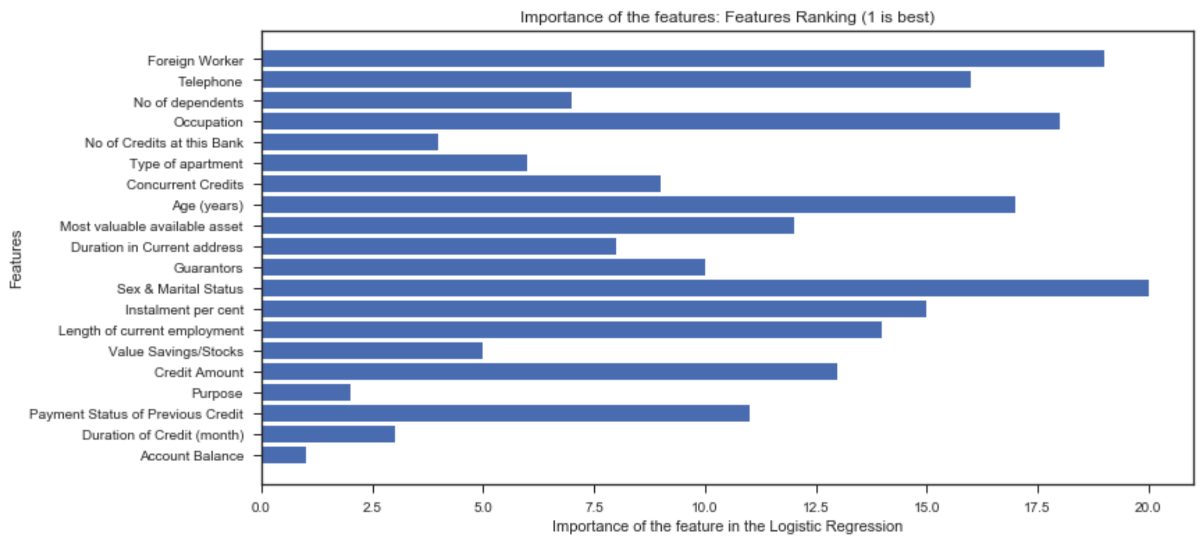
Recursive Features Elimination (RFE)

Let's recall that the RFE is method that aims to identify the best or worst performing feature. The algorithm sets the feature aside and then repeat the process with the rest of the features. Features are then ranked. The stability of RFE depends heavily on the model used. We have decided to apply that method to the SVC linear model and the logistic regression. We implement the following codes:

```python
from sklearn.feature_selection import RFE

# Create the RFE object for SVC linear
clf = SVC(kernel="linear",C=1, gamma=0.001)
```

```
rfe = RFE(estimator=clf, n_features_to_select=1, step=1)
rfe.fit(X, y)
ranking = rfe.ranking_
ranking

ranking = rfe.ranking_.reshape((4,4))
plt.matshow(ranking, cmap=plt.cm.Blues)
plt.colorbar()
plt.title("Ranking with RFE on SVC Linear Fine-Tuned")
plt.show()

##### Logistic regression #####
# Create the RFE object
clf_2 = LogisticRegression(C=1)
rfe_2 = RFE(estimator=clf_2, n_features_to_select=1, step=1)
rfe_2.fit(X, y)
ranking_2 = rfe_2.ranking_
ranking_2

ranking_2 = rfe_2.ranking_.reshape((4,4))
plt.matshow(ranking_2, cmap=plt.cm.Blues)
plt.colorbar()
plt.title("Ranking with Logistic Regression Fine-Tune")
plt.show()
```





As we can see above, we get a very similar result using both models. The results are clearly similar to the one found in the Extra Trees Classification. But in addition to: “The Account Balance”, “The Credit Amount”, “The Payment Status of Previous Credit”, “Purpose”, “Duration of Credit (month)” – the “Value Savings/Stocks” and the “number of credit at this bank” are added to be important features.

Now that we have completed all the data processing steps, let’s choose some models and assess them using ROC curve and AUC values.

III. Model Specification and parameter estimation:

Selection of models and assessment of fitting performance

Support Vector Classification (SVC):

Support Vector classification is a particular powerful and flexible class of supervised machine learning techniques. First, let’s re-call that we have to deal with a simple classification task. We can use a “line” (in fact an hyperplane) to separate the two sets of data. But we have to choose the best possible line that can discriminate between the two classes (non-defaulted and defaulted loans). One powerful way to do it is to use a support vector machine approach. Rather than drawing one line, we draw a margin. The “line” that maximize the margin is said to be optimal.

However, we must consider the case where we can’t draw a linear separation margin. If we want to fit nonlinear relationship, we can change the kernel. This is what we have done.
When it is clear that a linear discrimination is not relevant, we can think about other kernels such a radial or sigmoid. In our case we can be pretty sure to use the radial one or the sigmoid one given the data we have in possession.

The most powerful characteristic of that approach is that there it is not sensitive to the exact behavior of the distant points. Any point further from the margin which are correctly classify does not have any impact on the shape of the margin.

Hyper-parameter tuning on support vector classification (SVC):

Eventually, we can use a grid search cross-validation to explore various different parameters. We can adjust C (tuning parameter that controls the hardness of the margin). If the parameter is very large, the margin is hard and the algorithm does not tolerate points to lie in the margin. On the contrary, if the parameter is low, some points can lie in the margin area. We can also search for the optimal gamma (tuning parameter that controls the size of the radial basis function).

We test for C=1,10,100,1000 and gamma = 0.001, 0.0001 on all the SVC models using the following code:

```
print("Fine Tuning on SVC models")
from sklearn.model_selection import GridSearchCV
parameters = {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel':['linear',
'rbf','sigmoid','poly']}
grid = GridSearchCV(SVC(probability = True), parameters, verbose=3)
grid.fit(X_train,y_train)
print(grid.best_params_)
```

GridSearchCV is a powerful tuning method that helps us to find the best possible parameters. We find the best parameters and we implement them:

```
model9=SVC(kernel='rbf',C=100, gamma=0.001)
```

Fine tuning our parameters, we obtain the following metrics and classification report:

Note: for the metrics obtained, we have chosen not to display the coefficient of determination (R-squared) since it is not a linear classification problem and in also because in most of the case the chosen model will not follow the trend of the data (we have some data similar to categorical one), fitting worse than a horizontal line.

Average Precision on training set: 0.719061285228
Average Recall on training set: 0.716666666667
Average F1 Score on training set: 0.716391880625

Root Mean Squared Error (RMSE): 0.532290647422
Mean Absolute Error (MAE): 0.283333333333

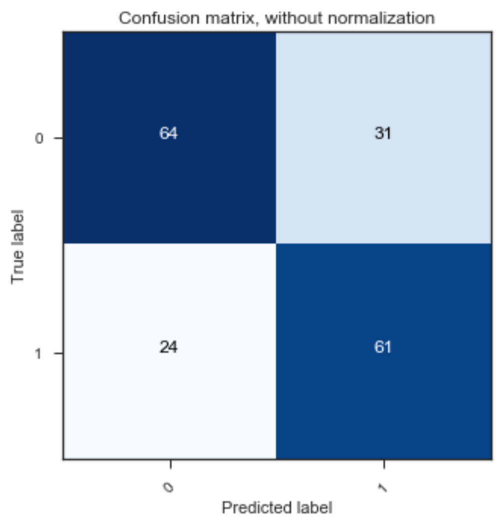
Average Precision on testing set: 0.696942248573
Average Recall on testing set: 0.694444444444
Average F1 Score on testing set: 0.69464254474

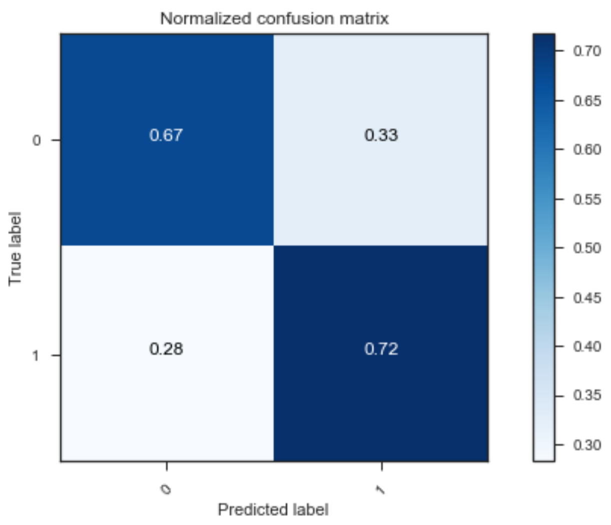
Root Mean Squared Error (RMSE): 0.552770798393
Mean Absolute Error (MAE): 0.305555555556

Classification Report:

	Precision	Recall	F1-Score	Support
0	0.72	0.67	0.7	95
1	0.66	0.72	0.69	85
Avg / Total	**0.7**	**0.69**	**0.69**	**180**

We obtain the following confusion matrices:





As said the having a False Negative more inaccuracy more damageable than having a false Positive one. In this case the SVM model using a radial (rbf) kernel with the fined tuned parameter does pretty well since the FP Rate of 33% vs 28% for the False Negative one and the F1-score on the testing set is of 69%.

Logistic Regression:

We are deeply interested in the logistic regression model. Why so? It is a useful model hen the AUC of the best model is below 0.80.

Moreover, as we have a lot of features and we struggle to determine which ones are relevant and which ones are not, there is the risk that the signal to noise ratio might be too small. In that case, we know that logistic regression tends to do well. Logistic regression can also be more robust: you don't have to make assumptions about the normality of the independent variables. It may handle nonlinear patterns

Hyper-parameter tuning on Logistic Regression:

We use GridSearchCV algorithm to get the best possible parameters for the logistic regression:

```python
print("Fine Tuning on Logistic Regression")
grid=GridSearchCV(cv=None,
       estimator=LogisticRegression(C=1.0, intercept_scaling=1, dual=False, fit_intercept=True,
          penalty='l2', tol=0.0001),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]},
       pre_dispatch='2*n_jobs', refit=True, verbose=3)
grid.fit(X_train, y_train)
print(grid.best_params_)
display_model_evaluation(grid)
y_pred = grid.predict(X_test)
display_confusion_matrices(y_test,y_pred)
```

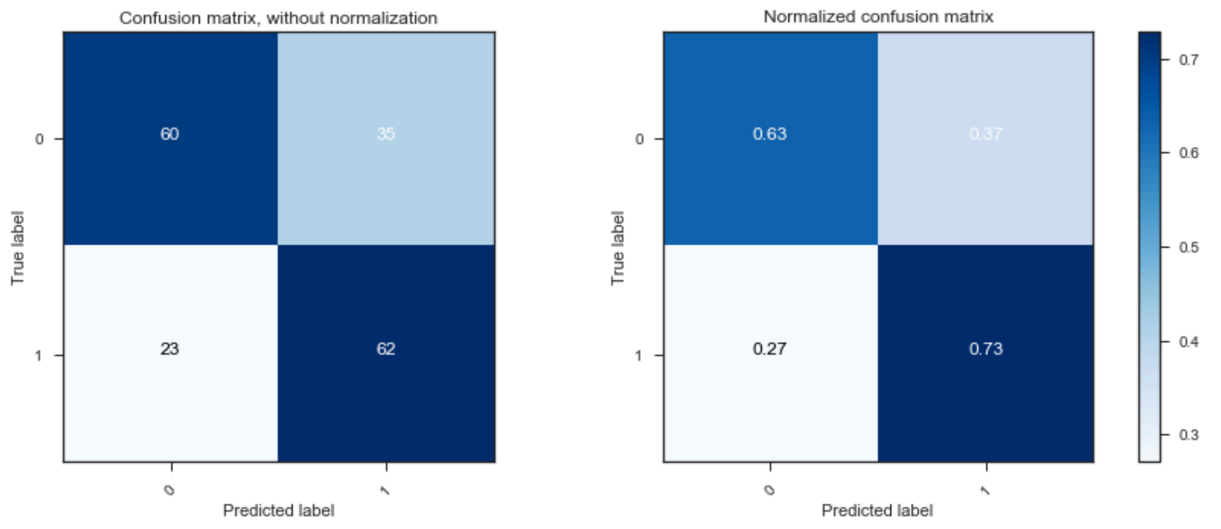
Metrics & Accuracy Evaluation:

```
Average Precision on training set: 0.7499773057371097
Average Recall on training set: 0.75
Average F1 Score on training set: 0.7499843997299735

Root Mean Squared Error (RMSE): 0.5
Mean Absolute Error (MAE): 0.25
```

```
Average Precision on testing set: 0.6833588650133179
Average Recall on testing set: 0.6777777777777778
Average F1 Score on testing set: 0.6775390651795146

Root Mean Squared Error (RMSE): 0.5676462121975467
Mean Absolute Error (MAE): 0.32222222222222224
```

Classification Report:

	Precision	Recall	F1-Score	Support
0	0.72	0.63	0.67	95
1	0.64	0.73	0.68	85
Avg / Total	**0.68**	**0.68**	**0.68**	**180**



In this case, we have FP Rate of 37% vs 27% for the False Negative one and the F1-score on the testing set is of 68%.

Naïve Bayes:

Naive Bayes is a powerful algorithm for classification problems.

Naïve Bayes relies on the following formula:

$$P(y|x) = \frac{P(y)P(x|y)}{P(x)}$$

Let's look at the same idea when we have to deal with multiple features:

$$P(y|x_1, x_2, \dots, x_n) = \frac{P(x_1, x_2, \dots, x_n|y)}{P(x_1, x_2, \dots, x_n)} P(y)$$

The approach is based on the assumption of class conditional independence of the features

$$P(y|x_1, x_2, \dots, x_n) = \frac{\prod_{i=1}^{n} P(x_i|y)}{P(x_1, x_2, \dots, x_n)} P(y)$$

That is a very strong assumption that is most likely false. Attributes interact surely. However, we know that even if we rely on bad assumptions, the model performs surprisingly well!

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i|y)$$

Gaussian Naïve Bayes (GNB):

Naives Bayes can be extended assuming a Gaussian distribution. We only need to estimate the mean μ_y and the standard deviation σ_y^2 using Maximum Likelihood estimation.

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp\left(-\frac{\left(x_i - \mu_y\right)^2}{2\sigma_y^2}\right)$$

Hyper-parameter tuning on Gaussian Naïve Bayes (GNB)?

Since there is not a hyper-parameter to tune, we have nothing to search with GridSearch algorithm.

This is the metrics and the classification report we obtained:

Metrics & Accuracy Evaluation: GaussianNB(priors=None)

Average Precision on training set: 0.699503074503
Average Recall on training set: 0.697619047619
Average F1 Score on training set: 0.69741500379

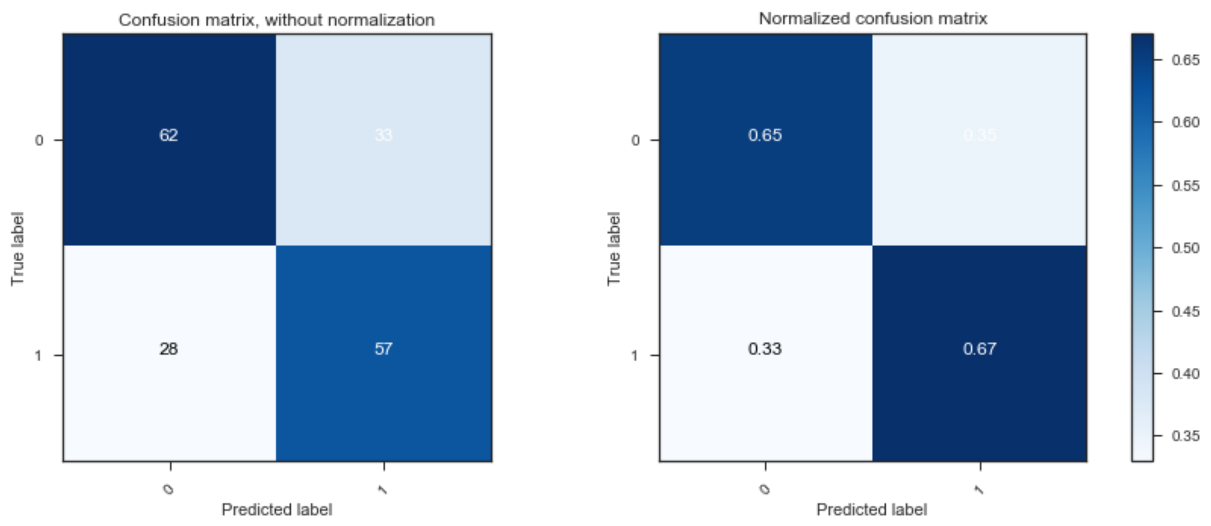
Root Mean Squared Error (RMSE): 0.549891764242
Mean Absolute Error (MAE): 0.302380952381

Average Precision on testing set: 0.662654320988
Average Recall on testing set: 0.661111111111
Average F1 Score on testing set: 0.661372801373

Root Mean Squared Error (RMSE): 0.582141639886
Mean Absolute Error (MAE): 0.338888888889

<u>Classification Report:</u>

	Precision	Recall	F1-Score	Support
0	0.69	0.65	0.67	95
1	0.63	0.67	0.65	85
Avg / Total	**0.66**	**0.66**	**0.66**	**180**



In this case the Naïve Bayes model is correct since there is a FP Rate of 35% vs 33% for the False Negative one and the F1-score on the testing set is of 66%.

K-Nearest Neighbor (KNN):

Let's re-call the basic principles of KNN. It is a non-parametric approach that we use to classify our data set. We do not have to make any assumption. We store all the data set and for each new point, we determine its Euclidian distance from each of the known label data. We select K data points that are closest to that point. We classify the point using a majority vote. It is an interesting method as the decision boundaries are no longer linear and can be very complex.

Hyper-parameter tuning on K-Nearest Neighbor (KNN):

We use GridSearch algorithm to try to find the best possible parameter k (number of nearest neighbor K). If K is too small the decision boundaries are going to be too sensitive to outliers. On the contrary, if K is too high, we can lose some important information. The choice of K is therefore predominant and we use GridSearch to find the better parameter.

```
k = np.arange(200)+1
parameters = {'n_neighbors': k}
knn = KNeighborsClassifier()
grid = GridSearchCV(knn,parameters,cv=10,verbose=3)
grid.fit(X_train,y_train)
print(grid.best_params_)
```

This is the metrics and the classification report we obtained with the best K in this case K=24.

Average Precision on training set: 0.712145346328
Average Recall on training set: 0.704761904762
Average F1 Score on training set: 0.703189308775

Root Mean Squared Error (RMSE): 0.543358164785
Mean Absolute Error (MAE): 0.295238095238

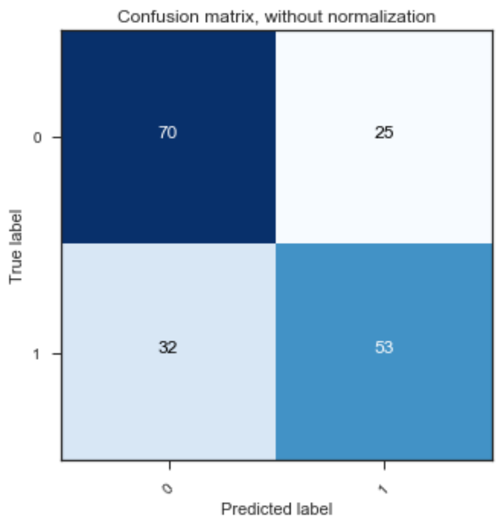
Average Precision on testing set: 0.683069381599
Average Recall on testing set: 0.683333333333

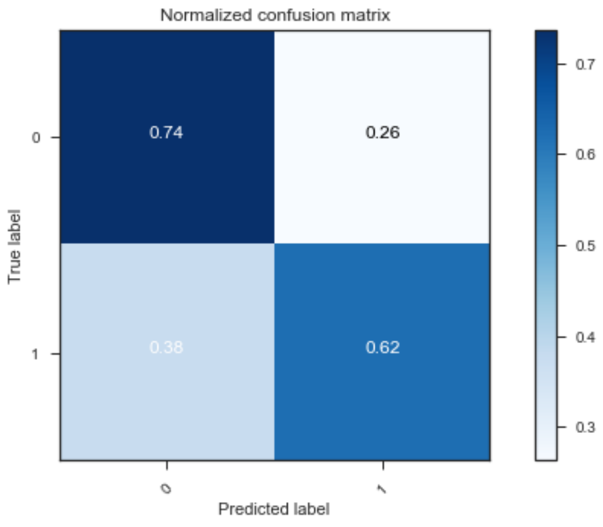
Average F1 Score on testing set: 0.682159799861

Root Mean Squared Error (RMSE): 0.562731433871
Mean Absolute Error (MAE): 0.316666666667

Classification Report:

	Precision	Recall	F1-Score	Support
0	0.69	0.74	0.71	95
1	0.68	0.62	0.65	85
Avg / Total	**0.68**	**0.68**	**0.68**	**180**





In this case the KNN model is acceptable since there is a FP Rate of 26% vs 38% for the False Negative one and the F1-score on the testing set is of 68%.

Multi-Layer Perceptron (MLPC):

We have also decided to implement MLPC for its capability to learn non-linear models. However, as the model uses a number of hyperparameters such as the number of hidden neurons, layers, and iterations, using tuning on MLPC can be computationally heavy. To do so we also used a grid search with a RELU activation function as seen in the lecture notes

```
param_grid = {
        'hidden_layer_sizes': [(7, 7), (128,), (128, 7)],
        'tol': [1e-2, 1e-3, 1e-4, 1e-5, 1e-6],
        'epsilon': [1e-3, 1e-7, 1e-8, 1e-9, 1e-8]
    }
grid= GridSearchCV(MLPClassifier(activation='relu',learning_rate='adaptive', learning_rate_init=1., early_stopping=True, shuffle=True),param_grid=param_grid, n_jobs=-1)
grid.fit(X_train,y_train)

This returns us the best parameters we have used to continue the study:
MLPClassifier {'epsilon': 0.001, 'hidden_layer_sizes': (128,), 'tol': 0.01}
```

Providing us with the following metrics:

Average Precision on training set: 0.718639093639
Average Recall on training set: 0.716666666667
Average F1 Score on training set: 0.716475475992

Root Mean Squared Error (RMSE): 0.532290647422
Mean Absolute Error (MAE): 0.283333333333

Average Precision on testing set: 0.688888888889
Average Recall on testing set: 0.688888888889
Average F1 Score on testing set: 0.688888888889

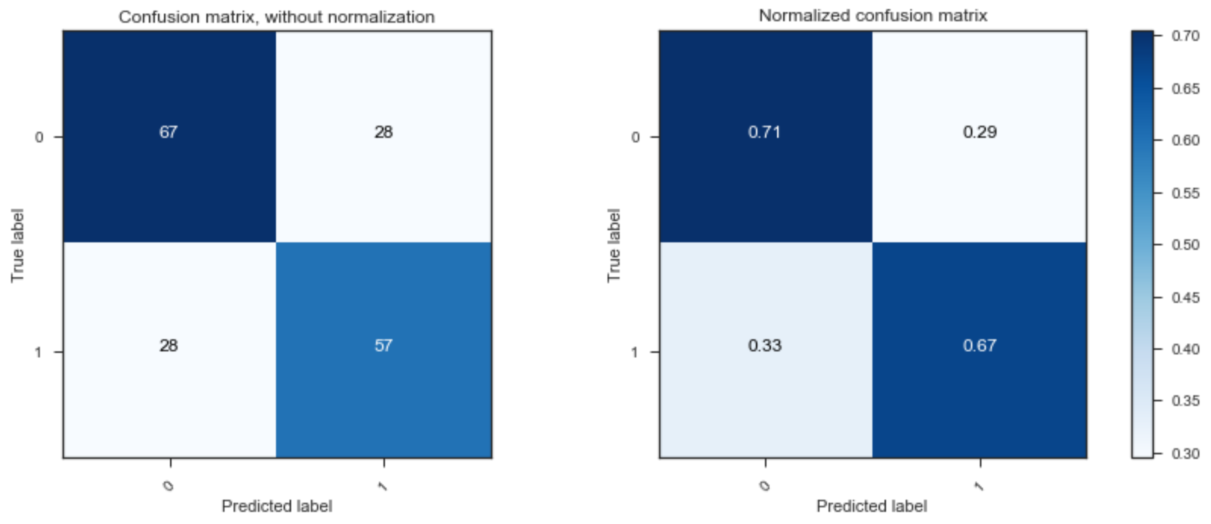
Root Mean Squared Error (RMSE): 0.557773351023
Mean Absolute Error (MAE): 0.311111111111

Classification Report:

	Precision	Recall	F1-Score	Support
0	0.71	0.71	0.71	95
1	0.67	0.67	0.67	85
Avg / Total	**0.69**	**0.69**	**0.69**	**180**

And with the following confusion matrices.

In this case the MLPC model is also acceptable since there is a FP Rate of 29% vs 33% for the False Negative one and the F1-score on the testing set is of 69%.



Decision Tree Classifier:

At first glance, we believe that Decision Tree Classifier can be a good model. Why so? Because nonlinear relationships between parameters do not affect tree performance. Moreover, decision trees implicitly perform feature selection. Decision Tree are known to be transparent: there is no ambiguity in decision-making with Decision Tree Classifier.

Hyper-tuning on Decision Tree Classifier:

Fine tuning our parameters, we obtain the following metrics and classification report:
The best {'min_samples_split' is 106}.

Average Precision on training set: 0.726329782639
Average Recall on training set: 0.72619047619
Average F1 Score on training set: 0.726223081619

Root Mean Squared Error (RMSE): 0.523268118472
Mean Absolute Error (MAE): 0.27380952381

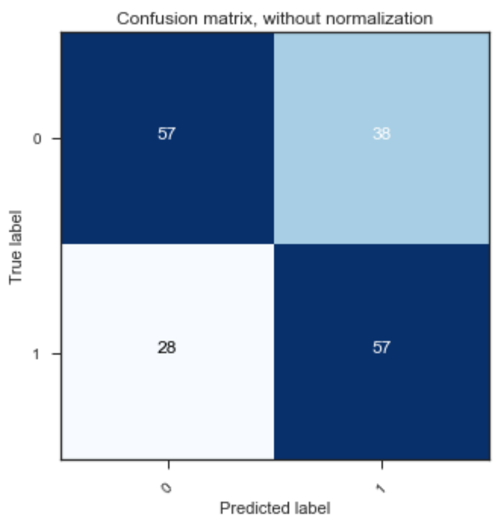
Average Precision on testing set: 0.637254901961
Average Recall on testing set: 0.633333333333
Average F1 Score on testing set: 0.633333333333

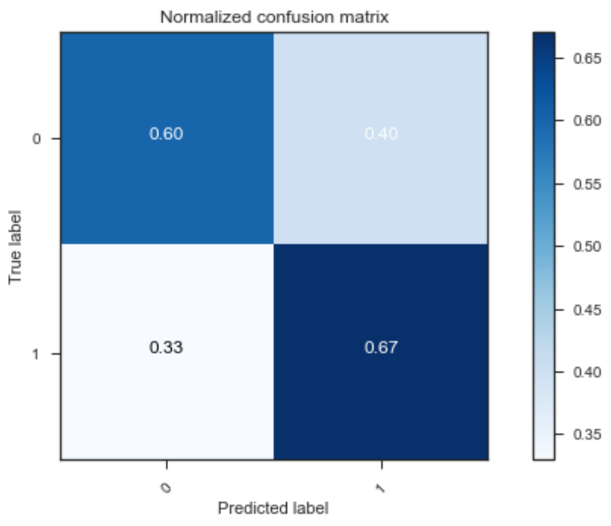
Root Mean Squared Error (RMSE): 0.605530070819
Mean Absolute Error (MAE): 0.366666666667

Classification Report :

	Precision	Recall	F1-Score	Support
0	0.67	0.60	0.63	95
1	0.60	0.67	0.63	85
Avg / Total	**0.64**	**0.63**	**0.63**	**180**

And the following confusion matrices:





The Decision Tree model is less acceptable since there is a FP Rate of 40% vs 33% for the False Negative one and the F1-score on the testing set is of 63%. This is for now the worst of our models.

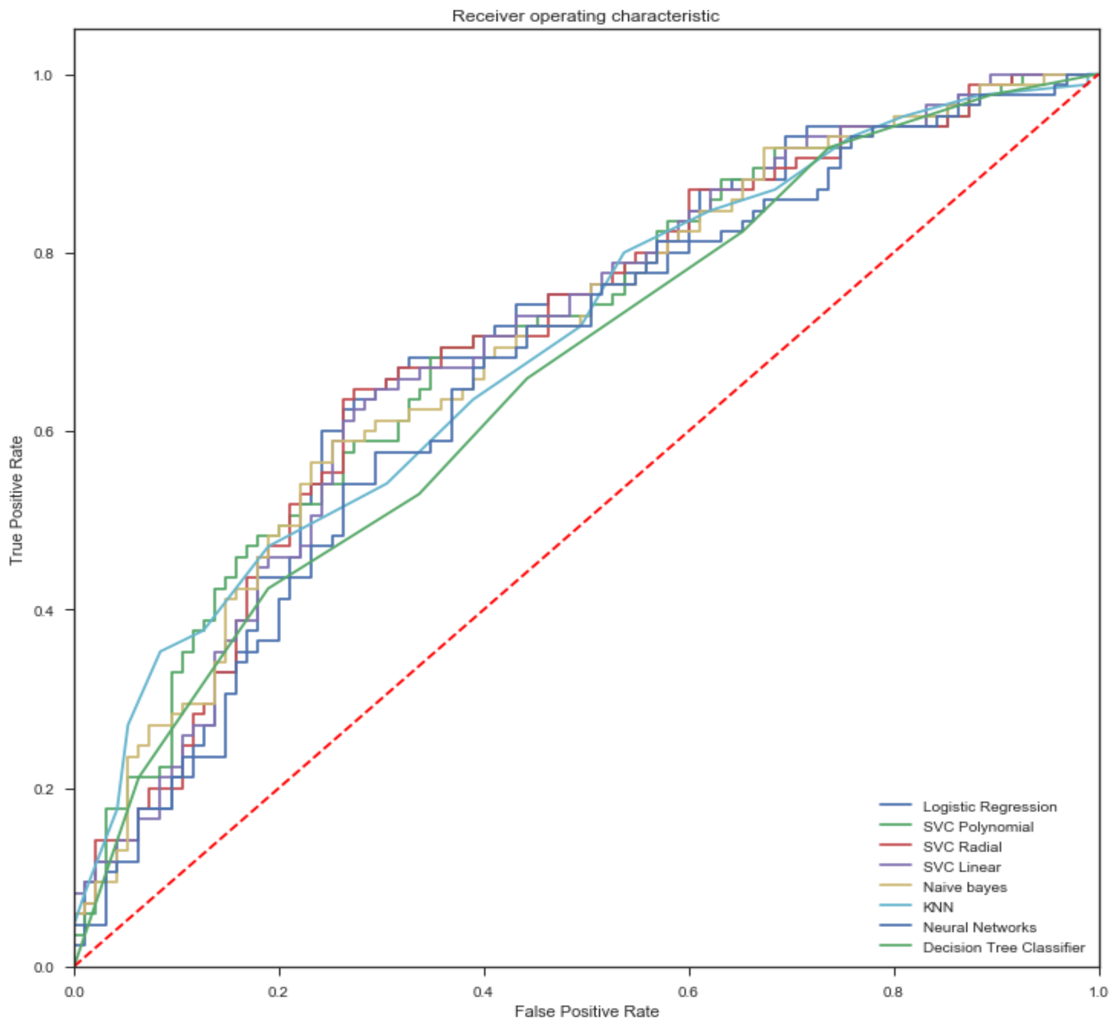
Now that we have re-call the basic characteristics of all the model used, we want to implement a method to compare and assess the efficiency of these methods. In a classification machine learning problem, a good way to proceed is to build the Receiver Operating Characteristic Curve (ROC)

IV. Model assessment:

The Receiver Operating Characteristic Curve (ROC):

Building the Receiver Operating Characteristic Curve is not an easy thing. In some case, it can be difficult to compute the model score and we then switch from [model].decision_function to [model].predict_proba.

You will find below the curves for all the models implemented. Let's re-call that the coordinates (x=0,y=1) refer to the ideal model. The red line represents the random guessing. And we want to our models to stay in the upper area of that red line. If it is not case, it means the model is worse than random guessing.



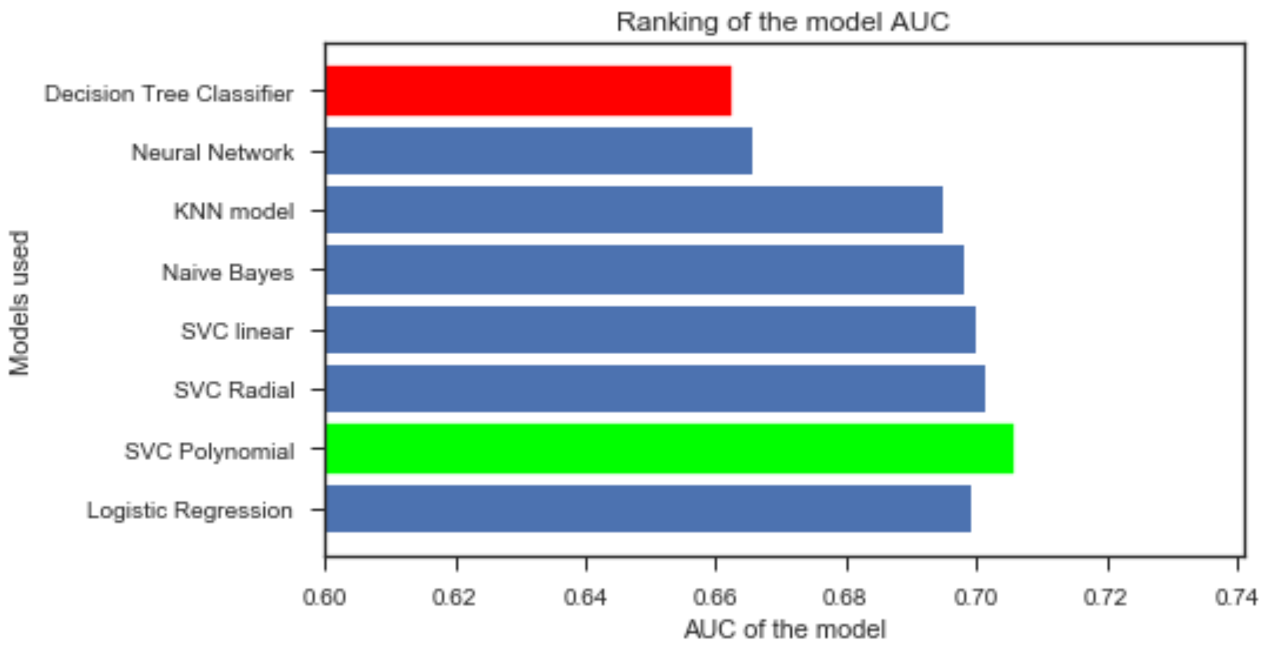
Fortunately, all the curves lie in the upper area. Moreover, we can't say with certainty that there is one better model above all by just looking at the graph. That is why we need to calculate the Area Under the Curve (AUC) to classify without any doubt the models.
The Logistic Regression tends to lie below the Neural Network one and the SVC Polynomial slightly lie above in average of the Decision Tree Classifier one.

Area Under the Curve (AUC):

That area is easy to calculate. For SVC models, we use the following code:

```python
models = [model1,model8,model9,model10]
models_names= ['Logistic Regression','SVC Polynomial','SVC Radial','SVC linear','Naive Bayes','KNN model','Neural Network']

auc_list=[]
fpr_list=[]
tpr_list=[]

for model in models:
    model=model.fit(X_train,y_train)
    y_score = model.decision_function(X_test)
    fpr, tpr, thresholds = roc_curve(y_test, y_score)
    fpr_list.append(fpr)
    tpr_list.append(tpr)
    auc_list.append(auc(fpr,tpr))
```

You will find below a graph with the best and worst models:



V. Stratified K-fold Cross Validation

Cross-validation process is a good procedure to reduce overfitting, by splitting our data into both a training and testing set. This is what we have done so far.

Nevertheless, it does not completely erase the hazard, as our predictive model is still likely to get overtrained on our training set. One improvement of this method would be to split our dataset into many more subsets and train our model on each one of them. If we refer to K as the number of subsets we create, this method is called the K-Fold Cross Validation.

In our case, we chose to use an improved version of the K-Fold method, called the Stratified K-Fold. The mechanism is the same but each split contains an equal proportion of the different classes we want to predict. For instance, in our case, each subset will contain 50% of non-default training example, and 50% of default training example. This aims at coping with the imbalance of our data, composed of 700 defaults and 300 defaults training examples.

Before applying the Stratified K-Fold method to all our models, we have to take the original data without any previous rebalancing. In fact, by applying a Stratified K-Fold method to some already balanced data, we will throw away too much data and give rise to a model with a very poor predictive power, as the initial purpose of the Stratified K-Fold is to tackle with the problem of imbalance.

Now, let’s see how to implement this.

```
#We define the number of splits we want to make
kf = StratifiedKFold(n_splits=150)
kf.get_n_splits(X)

#We go through our models and calculate the Area Under Curve for each of one them
for model in models:
    auc_average=[]
    for train, test in kf.split(X, y):
        X_train, X_test = X[train], X[test]
        y_train, y_test = y[train], y[test]
        model.fit(X_train, y_train)
        y_score = model.decision_function(X_test)
        fpr, tpr, thresholds = roc_curve(y_test, y_score)
        auc_average.append(auc(fpr,tpr))
    #For each model, we define the AUC as being the average of each one of the K spl
it
    auc_list_stratified.append(sum(auc_average) / len(auc_average))
```

Then, we display the AUC of all our different models in a bar chart and compare them.

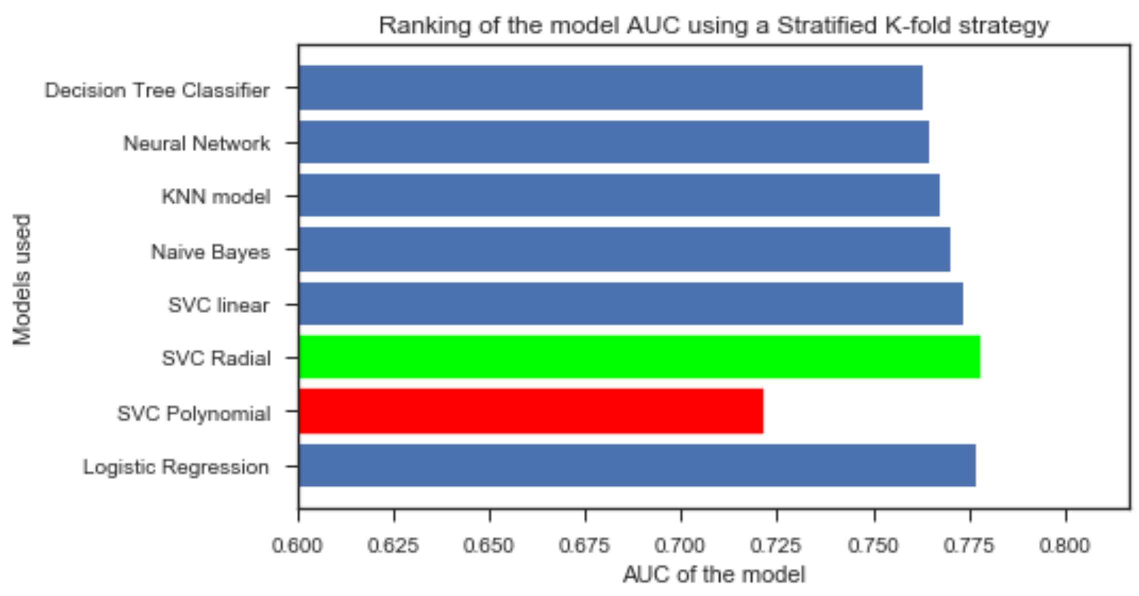


Chart x – AUC Ranking with Stratified K-Fold strategy

Without K-Stratified K Fold		*With K-Stratified K Fold*		
MODEL	AUC	MODEL	AUC	AUC gain
Logistic Regression	**71.084%**	Logistic Regression	**77.700%**	**6.616%**
SVC Polynomial	**70.576%**	SVC Polynomial	**75.400%**	**4.824%**
SVC Radial	**71.269%**	SVC Radial	**77.783%**	**6.514%**
SVC Linear	**70.960%**	SVC Linear	**77.350%**	**6.390%**
Naïve Bayes	**72.272%**	Naïve Bayes	**77.003%**	**4.731%**
KNN	**71.957%**	KNN	**76.743%**	**4.786%**
Neural Networks	**69.349%**	Neural Networks	**76.242%**	**6.893%**
Decision Tree	**66.247%**	Decision Tree	**76.215%**	**9.968%**

Table – AUC values with Stratified K-Fold strategy

One direct observation we can make is that every model performance has substantially increased, which shows the efficiency of the Stratified K-fold strategy. For some model, such as the Decision Tree Classifier, it resulted in a gain of more than 9.96% in the AUC!

Furthermore, we can remark that the best model is no longer the Naive Bayes model but the SVC Radial one, with an AUC of **77.783%.**

VI. Conclusion: Our preferred model

To sum up, in terms of predictive power, the Support Vector Classification seems to get the best results, hand in hand with the logistic regression. In fact, whatever the type of Kernel we choose for the SVC (radial or polynomial), our model has very high score. **But above all, this is the SVC Radial which has the strongest predictive power with an AUC of 77.783%.**

Furthermore, we call also note that the Stratified K-Fold Cross Validation allows us to get way higher scores than before, and also changes the ranking (the SVC Radial model comes first, whereas without K-Fold Strategy the SVC Polynomial comes first).