

# MACHINE LEARNING (MATH 2319)

## Project Phase 2

### Predicting Credit Risk Using German Credit Risk Data

Name: Gaurav Diwan

Student IDs: s3799691

Date: 9<sup>th</sup> June 2020

# Table of Contents

---

- Abstract.....1
- Introduction.....1
- Methodology .....2
  - Feature Encoding .....2
  - Feature Scaling .....4
  - Train and Test split.....4
  - Model building and evaluation.....5
  - Model Building and hyper-parameter tuning .....6
    - K-Nearest Neighbors (k-NN).....6
    - Decision Tree .....10
    - Gradient Boosting.....12
  - Performance comparison .....15
- Results .....16
- Discussion.....19
- Conclusion .....20
- References.....21

## Abstract

---

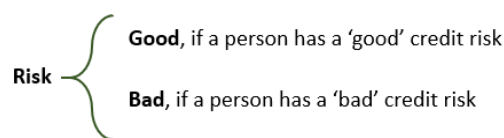
The main objective behind this report is to develop a recognition of pattern using supervised machine learning algorithms for a German bank to minimize the riskiness for a loan applicant while determining its creditworthiness. The dataset has 1000 no. of instances with 10 descriptive features, categorical and nominal both, including target feature 'Risk'. To develop an optimal machine learning model its important to begin with the data exploration analysis to find a pattern or important features contributing towards the target feature. And, to obtain more data-driven actionable insight, three different supervised machine learning techniques are performed and compared against each other for the best model selection. The reason behind the selection of a supervised machine learning technique is that we are dealing with the classification problem here. Once the best model is selected and compared against other models, the suggestions for the bank are made using the feature selection process.

## Introduction

---

The topic selected for the Machine Learning project<sup>[1]</sup> is the 'German Credit Risk'. The objective is to predict or classify whether a client has good or bad credit risk, and the data sourced for this project is sourced from [Kaggle](#) open data set repository. The project is branched into two phases where the first part focused on the data analysis part using data preparation, data exploration with the help of visualization and descriptive statistics. This report scrutinizes the second part of the project where different Machine Learning algorithms are demonstrated to meet the business objective.

The report demonstrates three different types of supervised machine learning algorithms, that are intuitively used in the world of data science, to predict or classify the credit riskiness of a customer. The processed dataset has 1000 no. of instances, 9 descriptive features and 1 target feature. The target feature is 'Risk' variable and is classified as binary:



Since the classification goal to 'predict the creditworthiness of a customer' where the outcomes are in the form of labeled data and are binary. Therefore, it is important to perform a supervised machine learning technique for the model building and selection process. Three different machine learning algorithms that are demonstrated during the process are:

- k-NN (k-nearest neighbor)
- Decision Tree
- Gradient Boosting

## Methodology

The results and the outputs are coded with the help of Python 3. In the first phase of the project, the implementation was done using libraries like numpy, pandas for data manipulation, data frame operations and data wrangling. Also, libraries like matplotlib, Altair, and seaborn are implemented for data visualization.

While for the second phase of the project different scikit-learn libraries are used for model building, feature selection, and model evaluation. Once the data set was cleaned and explored. The next step involved the process of model building.

Below is the overview of the dataset once all the data preparation steps were done and are ready for the model building process.

```
CreditData.head(10)
```

	Age	Sex	Job	Housing	Saving accounts	Checking account	Credit amount	Duration	Purpose	Risk	Age_Group
0	67	male	Skilled	own	NaN	little	1169	6	radio/TV	good	58+
1	22	female	Skilled	own	little	moderate	5951	48	radio/TV	bad	18-27
2	49	male	Unskilled and Resident	own	little	NaN	2096	12	education	good	48-57
3	45	male	Skilled	free	little	little	7882	42	furniture/equipment	good	38-47
4	53	male	Skilled	free	little	little	4870	24	car	bad	48-57
5	35	male	Unskilled and Resident	free	NaN	NaN	9055	36	education	good	28-37
6	53	male	Skilled	own	quite rich	NaN	2835	24	furniture/equipment	good	48-57
7	35	male	Highly Skilled	rent	little	moderate	6948	36	car	good	28-37
8	61	male	Unskilled and Resident	own	rich	NaN	3059	12	radio/TV	good	58+
9	28	male	Highly Skilled	own	little	moderate	5234	30	car	bad	NaN

## Feature Encoding

Since machine learning algorithms can read only numerical values. Therefore, before model building process it is important to transform all the categorical features into numerical values. For this step we have selected 'Get Dummies' method of encoding categorical features. 'Pandas get\_dummies' method is a straightforward procedure to encode categorical features to get dummy variables. In this method, each class of the categorical variable is converted into a feature with the value of 1 and 0, where 1 means the existence of that class feature and 0 means non-existence. The purpose to do so is to avoid multicollinearity in the dataset. However, the target feature Risk is encoded using 'replace' function manually, 'Good' and 'Bad' are encoded as 1 and 0 respectively.

For each of the categorical variable with two class, for example, 'Sex' the variables are converted into 1 and 0 using 'drop\_first=True'. While for a categorical variable with more than 2 levels of classes one-hot encoding is done using 'get\_dummies'. Once all the one-hot encoding was done, the preview of the data set is done using 'Data.columns'. Below is the chunk of codes.

```
CreditData.head(10)
```

	Age	Sex	Job	Housing	Saving accounts	Checking account	Credit amount	Duration	Purpose	Risk	Age_Group
0	67	male	Skilled	own	NaN	little	1169	6	radio/TV	good	58+
1	22	female	Skilled	own	little	moderate	5951	48	radio/TV	bad	18-27
2	49	male	Unskilled and Resident	own	little	NaN	2096	12	education	good	48-57
3	45	male	Skilled	free	little	little	7882	42	furniture/equipment	good	38-47
4	53	male	Skilled	free	little	little	4870	24	car	bad	48-57
5	35	male	Unskilled and Resident	free	NaN	NaN	9055	36	education	good	28-37
6	53	male	Skilled	own	quite rich	NaN	2835	24	furniture/equipment	good	48-57
7	35	male	Highly Skilled	rent	little	moderate	6948	36	car	good	28-37
8	61	male	Unskilled and Resident	own	rich	NaN	3059	12	radio/TV	good	58+
9	28	male	Highly Skilled	own	little	moderate	5234	30	car	bad	NaN

```
#Encoding Target feature
Data=CreditData.drop(columns='Risk')
Target=CreditData['Risk']
```

```
Target=Target.replace({'good':1,'bad':0})
Target.value_counts()
```

```
1    700
0    300
Name: Risk, dtype: int64
```

```
#Encoding Categorical Discreptive feature
```

```
categorical_cols = Data.columns[Data.dtypes==object].tolist()

for col in categorical_cols:
    n = len(Data[col].unique())
    if (n == 2):
        Data[col] = pd.get_dummies(Data[col],drop_first=True)

Data=pd.get_dummies(Data)
```

```
Data.columns
```

```
Index(['Age', 'Sex', 'Credit amount', 'Duration', 'Job_Highly Skilled',
       'Job_Skilled', 'Job_Unskilled and Non resident',
       'Job_Unskilled and Resident', 'Housing_free', 'Housing_own',
       'Housing_rent', 'Saving accounts_little', 'Saving accounts_moderate',
       'Saving accounts_quite rich', 'Saving accounts_rich',
       'Checking account_little', 'Checking account_moderate',
       'Checking account_rich', 'Purpose_business', 'Purpose_car',
       'Purpose_domestic appliances', 'Purpose_education',
       'Purpose_furniture/equipment', 'Purpose_radio/TV', 'Purpose_repairs',
       'Purpose_vacation/others', 'Age_Group_18-27', 'Age_Group_28-37',
       'Age_Group_38-47', 'Age_Group_48-57', 'Age_Group_58+'],
      dtype='object')
```

Overview of the data set, once feature encoding is done.

```
Data.sample(5,random_state=999)
```

	Age	Sex	Credit amount	Duration	Job_Highly Skilled	Job_Skilled	Job_Unskilled and Non resident	Job_Unskilled and Resident	Housing_free	Housing_own	...	Purpose_education	Purpose_ft
842	23	0	1943	18	0	1	0	0	0	1	...	0	
68	37	1	1819	36	0	1	0	0	1	0	...	1	
308	24	0	1237	8	0	1	0	0	0	1	...	0	
881	48	1	9277	24	0	1	0	0	1	0	...	0	
350	23	0	1236	9	0	1	0	0	0	0	...	0	

5 rows x 31 columns

## Feature Scaling

Post label encoding, the next step is to normalize the data. As the range of values for the data set varies differently, the objective functions may not work in a proper way for some of the machine learning algorithm. Hence, the data set is normalized using a min-max scalar operation using '*MinMaxScaler()*'. The output of the min-max scalar operation is an array, *NumPy array*, which led to the loss of columns names while reading, hence a replica of the data set is created to re-read the column names. Clearly, the binary feature, Sex, is the same as a binary feature with 1 and 0 values even after the min-max scaling. Below is the chunk of codes.

```
# Feature scaling
from sklearn import preprocessing

Data_df = Data.copy()

Data_scaler = preprocessing.MinMaxScaler()
Data_scaler.fit(Data)
Data = Data_scaler.fit_transform(Data)

pd.DataFrame(Data,columns=Data_df.columns).sample(5, random_state=999)
```

	Age	Sex	Credit amount	Duration	Job_Highly Skilled	Job_Skilled	Job_Unskilled and Non resident	Job_Unskilled and Resident	Housing_free	Housing_own	...	Purpose_education	Purp
842	0.071429	0.0	0.093155	0.205882	0.0	1.0	0.0	0.0	0.0	1.0	...	0.0	
68	0.321429	1.0	0.086332	0.470588	0.0	1.0	0.0	0.0	1.0	0.0	...	1.0	
308	0.089286	0.0	0.054308	0.058824	0.0	1.0	0.0	0.0	0.0	1.0	...	0.0	
881	0.517857	1.0	0.496699	0.294118	0.0	1.0	0.0	0.0	1.0	0.0	...	0.0	
350	0.071429	0.0	0.054253	0.073529	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	

## Train and Test split

One of the objectives of machine learning is to make good predictions for new data on unseen data. However, due to data constraint, building a model from the given data set is difficult due to the unavailability of previously unseen data<sup>[2]</sup>. Therefore, the data set is split into two different subsets:

**Train**, a subset to train a model.

**Test**, a subset to test the trained model.

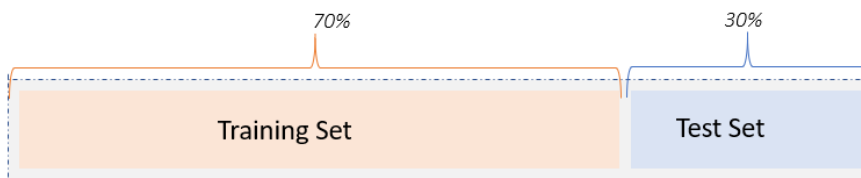


Figure 1, Slicing the data set into training and test set

Good performance on the new data set is generally determined by the good performance on the test data set. Based on the nature of the problem we have decided to split the data set into 70%-30% train and test split respectively using 'sklearn.model\_selection.train\_test\_split' function specified by 'test\_size' of 0.3 (30% test size), and using stratification 'stratify = y'. However, there is no thumb-rule of splitting the dataset into 70%-30% train and test split. It's largely dependent on a personal/business choice or based on the architect to be implemented.

```
# Splitting the data into train and test data set

from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn import model_selection
from sklearn.neighbors import KNeighborsClassifier

x=pd.DataFrame(Data).values
y=pd.DataFrame(Target).values

X_train, X_test,y_train,y_test = train_test_split(x,y,test_size=0.3,random_state=4,stratify = y)

print('X_Test dimension:',X_test.shape)
print('X_Train dimension:',X_train.shape)
print('y_Test dimension:',y_test.shape)
print('y_Train dimension:',y_train.shape)

X_Test dimension: (300, 31)
X_Train dimension: (700, 31)
y_Test dimension: (300, 1)
y_Train dimension: (700, 1)
```

Clearly, we can see that dataset is split into 70:30 train and test ratio. For train data set no. of instances are 700 and for test data set no. of instances are 300.

In the next step, we will train our model for 700 instances of training data set and test the trained model for 300 instances.

## Model building and evaluation

To estimate the skill of the supervised machine learning algorithms cross-validation is performed using 'RepeatedStratifiedKFold' technique. Repeated Stratified validation is the criteria ensuring that there is the same proportion of class label for each of the fold. For each of three models, 5 repeated stratified k-fold cross-validations is used, as we are dealing with the classification problem. With the help of model evaluation, the problem of underfitting and overfitting could be resolved, and the biases could be removed, if any.

```
# Model Evaluation
#Cross validation (5 fold Repeated stratified cross validation)

from sklearn.model_selection import RepeatedStratifiedKFold

cv_method = RepeatedStratifiedKFold(n_splits=5,random_state=999)
```

Cross-validation techniques are helpful in achieving more generalized relationships and could prevent models from losing stability.

## Model Building and hyper-parameter tuning

The basic intention behind splitting the dataset into two subsets is to find the possible parameters for each of the model and determining their accuracy to draw a possible conclusion towards the business objective.

The three supervise machine learning algorithms that are used are:

- K-Nearest Neighbors (k-NN),
- Decision Tree, and
- Gradient Boosting

### K-Nearest Neighbors (k-NN)

K-nearest neighbor is among the easiest and basic form of the supervised machine learning algorithm. It classifies the cases based on their proximity in terms of distance with other similar cases. In other words, similar cases are closer to each other while the non-similar cases are far from each other. Hence, distance is a measure for their similarity or dissimilarity and based on that specifies the number of nearest neighbors. There are two different distance metric to measure similar cases.

- Euclidean is the ordinary straight-line distance between two data points.
- Manhattan is the distance between two points is the sum of the absolute difference between two cartesian coordinates.

The algorithms assign a class to the new data based on the k-most similar no. of instances performed on the trained model. It's a non-parametric technique, as it doesn't take any underlying assumption about data distribution. Hence, the accuracy of KNN algorithms is dependent on these three parameters.

- The no. of neighbors, '*n\_neighbors*'
- Distance metric selection, '*p*' = 1: Manhattan and 2: Euclidean
- Weights, with each different contributing differently in terms of weight.

The k-NN is provided under '*sklearn.neighbors.KNeighborsClassifiers*' library.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix

KNN = KNeighborsClassifier()
KNN.fit(X_train, y_train)
KNNpred = KNN.predict(X_test)

print(confusion_matrix(y_test, KNNpred))
print(classification_report(y_test, KNNpred))
print(round(accuracy_score(y_test, KNNpred),3)*100)
```

```
[[ 19  71]
 [ 30 180]]
```

	precision	recall	f1-score	support
0	0.39	0.21	0.27	90
1	0.72	0.86	0.78	210
micro avg	0.66	0.66	0.66	300
macro avg	0.55	0.53	0.53	300
weighted avg	0.62	0.66	0.63	300

66.3



From the above results, we can see that the model with no hyper-parameter selections has an accuracy score of 66.3% on test split.

Therefore, in order to improve the model's accuracy, the best possible parameters are identified, below chunk of code was run and based on that AUC score was identified for the trained model.

```
#Hyper parameter tuning
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

parameter_KNN = {'p':[1,2],
                  'n_neighbors':[2,4,6,8,10,12,14,16,18,20,24,26,27,28,30,32,34,36],
                  'weights':['uniform','distance']}
GridSearchKNN = GridSearchCV(estimator = KNeighborsClassifier(),
                             param_grid = parameter_KNN,
                             cv = cv_method,
                             verbose = 1,
                             scoring = 'roc_auc',
                             n_jobs=-2,
                             return_train_score=True)

GridSearchKNN.fit(X_train,y_train)

print("AUC Score:" + str(GridSearchKNN.best_score_))
print("Best Parameters: " + str(GridSearchKNN.best_params_))
```

Fitting 50 folds for each of 72 candidates, totalling 3600 fits

```
[Parallel(n_jobs=-2)]: Using backend LokyBackend with 7 concurrent workers.
[Parallel(n_jobs=-2)]: Done 36 tasks | elapsed: 1.6s
[Parallel(n_jobs=-2)]: Done 1138 tasks | elapsed: 7.0s
[Parallel(n_jobs=-2)]: Done 3138 tasks | elapsed: 19.3s
```

AUC Score:0.6803109815354713

Best Parameters: {'n\_neighbors': 36, 'p': 2, 'weights': 'uniform'}

```
[Parallel(n_jobs=-2)]: Done 3600 out of 3600 | elapsed: 22.4s finished
```

It can be clearly inferred that the k-NN model has a mean AUC (Area under the curve) score of 0.68 and performs best with 'n\_neighbors'=36, 'p'=2, and with uniform 'weights'.

In order to check if these results are any significant or not to any approach, a custom function was created to format grid search outputs as Pandas data frame and based on that hyper-parameters with the highest mean AUC score are sorted and plotted using 'altair' library.

```
# Importing altair library for visualisation
```

```
import altair as alt
alt.renderers.enable('notebook')
```

```
RendererRegistry.enable('notebook')
```

```
def get_search_results(gs):
    def model_result(scores, params):
        scores = {'mean_score': np.mean(scores),
                  'std_score': np.std(scores),
                  'min_score': np.min(scores),
                  'max_score': np.max(scores)}
        return pd.Series(**params, **scores)

    models = []
    scores = []

    for i in range(gs.n_splits_):
        key = f"split{i}_test_score"
        r = gs.cv_results_[key]
        scores.append(r.reshape(-1,1))

    all_scores = np.hstack(scores)
    for p, s in zip(gs.cv_results_['params'], all_scores):
        models.append((model_result(s, p)))

    pipe_results = pd.concat(models, axis=1).T.sort_values(['mean_score'], ascending=False)

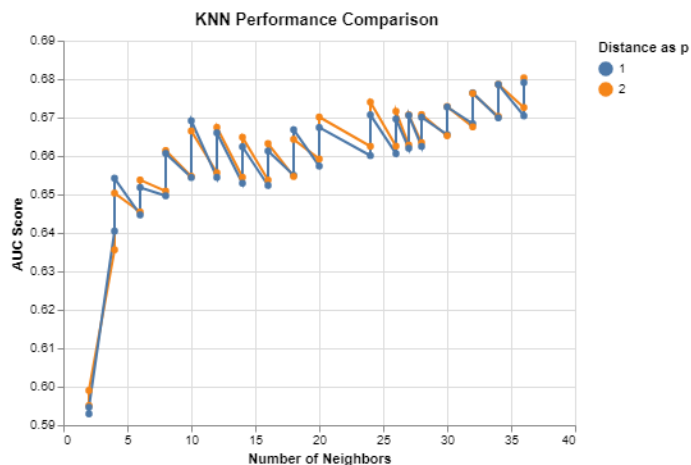
    columns_first = ['mean_score', 'std_score', 'max_score', 'min_score']
    columns = columns_first + [c for c in pipe_results.columns if c not in columns_first]

    return pipe_results[columns]
```

```
results_KNN = get_search_results(GridSearchKNN)
results_KNN.head()
```

	mean_score	std_score	max_score	min_score	n_neighbors	p	weights
70	0.680311	0.0436294	0.759232	0.553086	36	2	uniform
68	0.679118	0.0449714	0.76725	0.545068	36	1	uniform
66	0.678656	0.042742	0.763362	0.555879	34	2	uniform
64	0.678603	0.0429374	0.761054	0.555637	34	1	uniform
60	0.676443	0.0433901	0.750972	0.55758	32	1	uniform

```
import altair as alt
alt.Chart(results_KNN,
          title='KNN Performance Comparison '
          ).mark_line(point=True).encode(
    alt.X('n_neighbors', title='Number of Neighbors'),
    alt.Y('mean_score', title='AUC Score', scale=alt.Scale(zero=False)),
    alt.Color('p:N', title='Distance as p')
)
```



Also, for performance measurement, we use ROC (Receiver Operating Characteristics) curve to evaluate model performance. In other words, results from ROC and AUC can be interpreted as a model's capability of differentiation between classes.

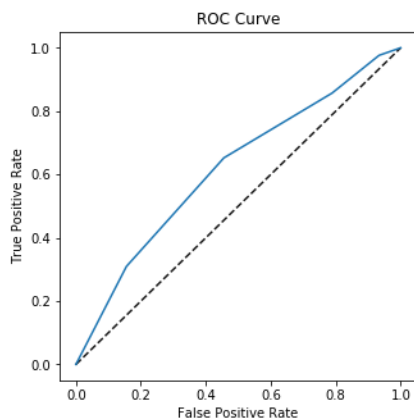
ROC curve is a probability curve plotted with 'True Positive Rate' (TPR) on its y-axis and 'False Positive Rate' (FPR) on its x-axis. However, for test evaluation, diagnostic ROC curve is a fundamental tool<sup>[3]</sup>.

```
from sklearn.utils import resample
from sklearn.metrics import roc_curve

y_pred_prob = KNN.predict_proba(X_test)[:,-1]

fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```



Each point on the ROC represents the relation between TPR and FPR with respect to a decision threshold. This means if the ROC curve passes through the upper left corner then it has a test with perfect discrimination. Hence, closer the ROC curve is to the upper left corner higher will be the accuracy of a specific model. In this case, the ROC curve is closer to the diagonal dashed line which means the accuracy for the k-NN model is poor.

## Decision Tree

Decision Tree algorithms are generally used for solving regression and classification problems. Decision tree algorithm follows a top-down approach, using tree representation. The leaves representing the class label and branch represents variable in conjunctions for those class labels. The decision tree classifier is build using 'sklearn.tree.DecisionTreeClassifier' library.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix

DT = DecisionTreeClassifier()
DT.fit(X_train, y_train)
DTpred = DT.predict(X_test)

print(confusion_matrix(y_test, DTpred))
print(classification_report(y_test, DTpred))
print(round(accuracy_score(y_test, DTpred),4)*100)
```

```
[[ 35  55]
 [ 48 162]]
```

	precision	recall	f1-score	support
0	0.42	0.39	0.40	90
1	0.75	0.77	0.76	210
micro avg	0.66	0.66	0.66	300
macro avg	0.58	0.58	0.58	300
weighted avg	0.65	0.66	0.65	300

65.67

Model accuracy on test split data is 65.6% with no hyper-parameter tuning. Hence, to increase the model accuracy, the best parameters are identified using 'GridSearchCV' function under 'sklearn.grid\_search.GridSearchCV' on the training data. However, one of the primary challenge during the algorithm implementation is feature selection to consider as the root node for each of the level. Hence, there two popular attribute selection measures.

- Information Gain measures the information contained by each feature and uses entropy to measure the homogeneity of a sample.
- Gini Index measures the probability for a variable that is falsely classified when selected randomly

With highest a mean AUC score of 0.68, the hyper-parameter selection for decision tree model are: 'max\_depth' =3, 'min\_samples\_split' = 3, and 'criterion' = Gini on the trained model.

#Hyper parameter tuning

```
parameter_DecisionTree = {'max_depth':[1,2,3,4,5,6,7,8,9,10,11,12,13,14],
                           'min_samples_split':[2,3,5],
                           'criterion':['entropy', 'gini']}
GridSearchDecisionTree = GridSearchCV(estimator=DecisionTreeClassifier(),
                                       param_grid=parameter_DecisionTree,
                                       cv=cv_method,
                                       verbose=1,
                                       n_jobs=-2,
                                       scoring='roc_auc')
```

```
GridSearchDecisionTree.fit(X_train,y_train)
```

```
print("Best Score:" + str(GridSearchDecisionTree.best_score_))
print("Best Parameters:" + str(GridSearchDecisionTree.best_params_))
```

Fitting 50 folds for each of 84 candidates, totalling 4200 fits

[Parallel(n\_jobs=-2)]: Using backend LokyBackend with 7 concurrent workers.  
 [Parallel(n\_jobs=-2)]: Done 546 tasks | elapsed: 0.6s

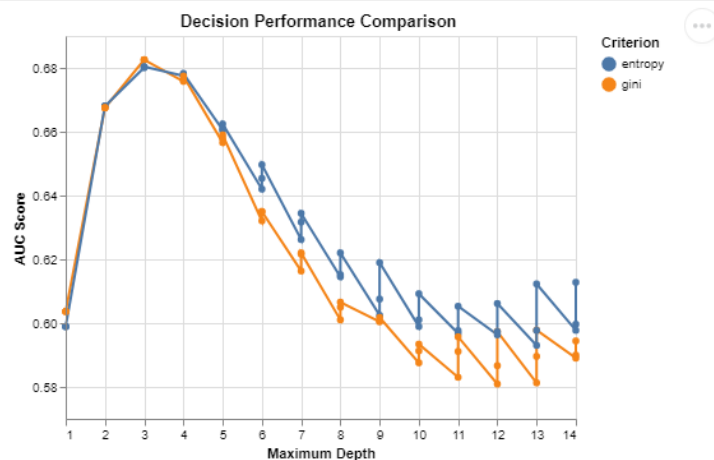
Best Score:0.6825631681243924  
 Best Parameters: {'criterion': 'gini', 'max\_depth': 3, 'min\_samples\_split': 3}

[Parallel(n\_jobs=-2)]: Done 4200 out of 4200 | elapsed: 4.6s finished

```
results_DT = get_search_results(GridSearchDecisionTree)
results_DT.head()
```

	mean_score	std_score	max_score	min_score	criterion	max_depth	min_samples_split
49	0.682563	0.0481436	0.807094	0.516642	gini	3	3
50	0.682529	0.0473553	0.807094	0.526361	gini	3	5
48	0.682519	0.0480889	0.807094	0.516642	gini	3	2
8	0.680374	0.050925	0.800777	0.516642	entropy	3	5
6	0.680262	0.0500365	0.800777	0.526361	entropy	3	2

```
import altair as alt
alt.Chart(results_DT,
          title='Decision Performance Comparison '
          ).mark_line(point=True).encode(
    alt.X('max_depth', title='Maximum Depth'),
    alt.Y('mean_score', title='AUC Score', scale=alt.Scale(zero=False)),
    alt.Color('criterion:N', title='Criterion')
)
```



Also, the results from the ROC curve are interpreted to evaluate model performance. Even the decision model algorithm on test data performs poorly, as ROC curve is closer to the diagonal line as shown in the figure below along with the chunk of codes.

```

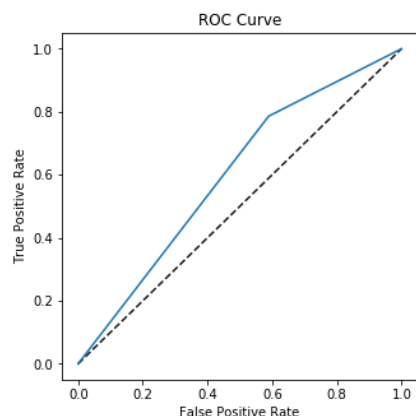
from sklearn.utils import resample
from sklearn.metrics import roc_curve

y_pred_prob = DT.predict_proba(X_test)[:,-1]

fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()

```



## Gradient Boosting

Clearly, k-NN and Decision Tree models are not the sophisticated models for our classification problem, as they have low AUC score for the trained data and poor ROC curve for test data, as a result, it is risky to implement these two algorithms. Hence to boost the predictive modeling approach. Gradient Boosting algorithm is built, as the third algorithm to increase the model capabilities for the prediction problem. Boosting is a sequential technique which follows the ensemble principle. The model prediction accuracy is increased by combining the set of weak learners. For any instant 't', model outcomes are weighed based on results for previous instant 't-1'<sup>[4]</sup>, where lower weights are generally assigned to the outcomes which are predicted correctly as compared to the outcomes which are misclassified. The Gradient Boost classifier algorithm builder is provided under the '*sklearn.ensemble.GradientBoostingClassifier*' library.

```
# GradientBoostClassification
```

```

from sklearn.ensemble import GradientBoostingClassifier

GBX = GradientBoostingClassifier()
GBX.fit(X_train, y_train)
GBXpred = GBX.predict(X_test)

print(confusion_matrix(y_test, GBXpred))
print(classification_report(y_test, GBXpred))
print(round(accuracy_score(y_test, GBXpred),3)*100)

```

```

[[ 35  55]
 [ 19 191]]

```

	precision	recall	f1-score	support
0	0.65	0.39	0.49	90
1	0.78	0.91	0.84	210
micro avg	0.75	0.75	0.75	300
macro avg	0.71	0.65	0.66	300
weighted avg	0.74	0.75	0.73	300

```
75.3
```

The model accuracy on test data is 75.3%, which is comparatively higher than the other two models. Therefore, it will be interesting to see the AUC score for Gradient Boost algorithm on the trained data and the ROC curve for test data.

The tree-specific parameters are used.

- *max\_depth* used to control over-fitting defining the maximum depth of a tree
- *max\_features*, number of features considered during the best split.
- *min\_samples\_split*, minimum no. of samples used in a node for splitting the data.

Using the below chunk of code, the best hyper-parameters are identified with the values.

```
#Hyper parameter tuning

parameter_GB = {'max_features': [5,10,15,20,25,30],
                'max_depth': [2,3,4,5,6,7],
                'min_samples_split': [2,3,4,5]}
GridSearchGB = GridSearchCV(estimator=GradientBoostingClassifier(),
                             param_grid=parameter_GB,
                             cv=cv_method,
                             verbose=1,
                             n_jobs=-2,
                             scoring='roc_auc')

GridSearchGB.fit(X_train,y_train)
print("Best Score:" + str(GridSearchGB.best_score_))
print("Best Parameters: " + str(GridSearchGB.best_params_))

Fitting 50 folds for each of 144 candidates, totalling 7200 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 7 concurrent workers.
[Parallel(n_jobs=-2)]: Done 102 tasks      | elapsed:    1.1s
[Parallel(n_jobs=-2)]: Done 702 tasks      | elapsed:    8.6s
[Parallel(n_jobs=-2)]: Done 1702 tasks     | elapsed:   23.9s
[Parallel(n_jobs=-2)]: Done 3102 tasks     | elapsed:   55.6s
[Parallel(n_jobs=-2)]: Done 4500 tasks     | elapsed:   1.7min
[Parallel(n_jobs=-2)]: Done 5600 tasks     | elapsed:   2.6min
[Parallel(n_jobs=-2)]: Done 6278 tasks     | elapsed:   3.4min
[Parallel(n_jobs=-2)]: Done 7028 tasks     | elapsed:   4.4min

Best Score:0.7292541302235179
Best Parameters: {'max_depth': 2, 'max_features': 10, 'min_samples_split': 5}

[Parallel(n_jobs=-2)]: Done 7200 out of 7200 | elapsed:  4.6min finished

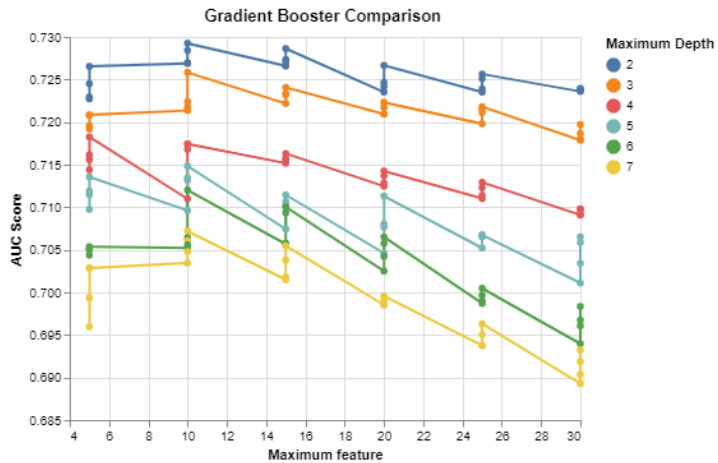
results_GB = get_search_results(GridSearchGB)
results_GB.head()
```

	mean_score	std_score	max_score	min_score	max_depth	max_features	min_samples_split
7	0.729254	0.041715	0.800777	0.596453	2.0	10.0	5.0
8	0.728632	0.041673	0.793732	0.597789	2.0	15.0	2.0
4	0.728416	0.044006	0.801020	0.591351	2.0	10.0	2.0
9	0.727347	0.042949	0.795190	0.596696	2.0	15.0	3.0
10	0.727046	0.041513	0.790816	0.594509	2.0	15.0	4.0

The best Gradient Boost classifier has *max\_features* of 10, *min\_samples\_split* of 5 and *max\_depth* of 2 with the mean AUC score of 0.72.

Visualization is illustrated using the following codes.

```
import altair as alt
alt.Chart(results_GB,
          title='Gradient Booster Comparison '
          ).mark_line(point=True).encode(
    alt.X('max_features', title='Maximum feature'),
    alt.Y('mean_score', title='AUC Score', scale=alt.Scale(zero=False)),
    alt.Color('max_depth:N', title='Maximum Depth')
)
```



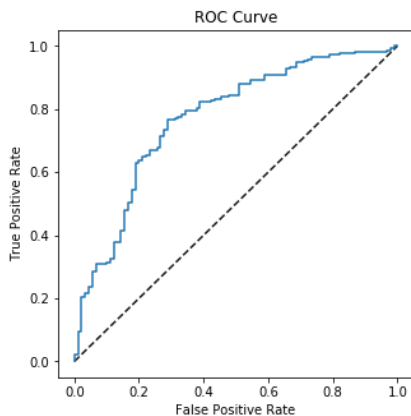
Interpretations of results using the ROC curve.

```
from sklearn.utils import resample
from sklearn.metrics import roc_curve

y_pred_prob = GB.predict_proba(X_test)[:,-1]

fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```



Based on the AUC score and the ROC curve, Gradient Boost classifier algorithms performs better than k-NN and Decision Tree algorithms.



## Performance comparison

However, it is too soon to conclude anything based on the model performance on trained data. Therefore, each of the optimized models with the help of optimal parameter selection is fitted on the test data using the cross-validation technique (5-fold repeated stratified cross-validation). However, cross-validation is a random process; therefore, a pairwise t-test is performed to determine and compare the model performances and to check if the results are statistically significant or not.

```
# Performance Comparison
from sklearn.model_selection import cross_val_score

cv_method_ttest = RepeatedStratifiedKFold(n_splits=5, random_state=999)

cv_results_KNN = cross_val_score(estimator=GridSearchKNN.best_estimator_,
                                X=X_test,
                                y=y_test,
                                cv=cv_method_ttest,
                                n_jobs=-2,
                                scoring='roc_auc')

print('Accuracy score for k-NN:', round(cv_results_KNN.mean(), 3) * 100)
```

Accuracy score for k-NN: 72.8

```
from sklearn.model_selection import cross_val_score

cv_method_ttest = RepeatedStratifiedKFold(n_splits=5, random_state=999)

cv_results_DT = cross_val_score(estimator=GridSearchDecisionTree.best_estimator_,
                                X=X_test,
                                y=y_test,
                                cv=cv_method_ttest,
                                n_jobs=-2,
                                scoring='roc_auc')

print('Accuracy score for Decision Tree:', round(cv_results_DT.mean(), 3) * 100)
```

Accuracy score for Decision Tree: 70.0

```
from sklearn.model_selection import cross_val_score

cv_method_ttest = RepeatedStratifiedKFold(n_splits=5, random_state=999)

cv_results_GB = cross_val_score(estimator=GridSearchGB.best_estimator_,
                                X=X_test,
                                y=y_test,
                                cv=cv_method_ttest,
                                n_jobs=-2,
                                scoring='roc_auc')

print('Accuracy score for Gradient Booster:', round(cv_results_GB.mean(), 3) * 100)
```

Accuracy score for Gradient Booster: 75.8

Clearly, comparing the accuracy score for all three models performed on test data 'Gradient Boost classifier' has the highest mean accuracy score 75.8% for test data.

We also conducted a paired t-test for the mean AUC score for the following models.

- k-NN vs DT,
- DT vs GB, and
- GB vs k-NN

'stats.ttest\_rel' function was used under 'SciPy' library to perform a pairwise t-test for the above models and results are compared based on their p-value. Below is the code

```
from scipy import stats

print(stats.ttest_rel(cv_results_KNN, cv_results_DT))
print(stats.ttest_rel(cv_results_DT, cv_results_GB))
print(stats.ttest_rel(cv_results_KNN, cv_results_GB))

Ttest_relResult(statistic=1.8897372286101224, pvalue=0.06471664943930171)
Ttest_relResult(statistic=-5.2185691347723635, pvalue=3.6406726982240483e-06)
Ttest_relResult(statistic=-3.4412017837608637, pvalue=0.0011932826098079142)
```

From the output, the p-value is less than  $\alpha=0.05$  for two instances where the Decision Tree model is compared with Gradient Boost, and k-NN is compared with Gradient Boost. In this case, results are statistically significant if the p-value is smaller than  $\alpha=0.05$  which is with the case for the Gradient Boost model. Hence it is safe to conclude that **'Gradient Boost classifier' model is the best model, in terms of mean AUC score, when performed for test data.**

## Results

---

As we have used the mean AUC score to optimize algorithm's hyperparameter, now we would evaluate the model's performance via 'classification\_report' using 'sklearn.metrics'. The classification report provides a summary of the following metrics: Accuracy, Precision, Recall, and F1 score. It also provides a brief explanation of 'confusion matrix'.

As we are dealing with the classification problem of 'predicting whether a loan applicant has a good or bad credit risk', this is where the classification report and confusion matrix are important. Below is the chunk of code for classification report for each of the three classifier models along with the output.

We also calculated the accuracy score for all three classifier models to evaluate their performances on test data split.

```
#Evaluating model performances on the test data
pred_KNN = GridSearchKNN.predict(X_test)
pred_DT = GridSearchDecisionTree.predict(X_test)
pred_GB = GridSearchGB.predict(X_test)

#Classification report and Accuracy score for each of the model.

from sklearn import metrics
print("\nClassification report and Accuracy score for K-Nearest Neighbor on test data split")
print(metrics.classification_report(y_test, pred_KNN))
print('Accuracy score',round(metrics.accuracy_score(y_test,pred_KNN),3)*100)

print("\nClassification report and Accuracy score for Decision Tree on test data split")
print(metrics.classification_report(y_test, pred_DT))
print('Accuracy score',round(metrics.accuracy_score(y_test,pred_DT),3)*100)

print("\nClassification report for Accuracy score for Gradient Boost on test data split")
print(metrics.classification_report(y_test, pred_GB))
print('Accuracy score',round(metrics.accuracy_score(y_test,pred_GB),3)*100)
```

```

Classification report K-Nearest Neighbor
precision    recall  f1-score   support

     0       0.57     0.04     0.08        90
     1       0.71     0.99     0.82       210

 micro avg       0.70     0.70     0.70       300
 macro avg       0.64     0.52     0.45       300
weighted avg       0.67     0.70     0.60       300

```

```

Classification report Decision Tree
precision    recall  f1-score   support

     0       0.58     0.46     0.51        90
     1       0.79     0.86     0.82       210

 micro avg       0.74     0.74     0.74       300
 macro avg       0.68     0.66     0.66       300
weighted avg       0.72     0.74     0.73       300

```

```

Classification report for Gradient Boost
precision    recall  f1-score   support

     0       0.64     0.33     0.44        90
     1       0.76     0.92     0.83       210

 micro avg       0.74     0.74     0.74       300
 macro avg       0.70     0.63     0.64       300
weighted avg       0.73     0.74     0.71       300

```

To summaries the performance of all the models and evaluating based on their mean AUC, hyper-parameter selection, ROC curve, t-test results and Model accuracy on test data. ***It can be clearly inferred that ‘Gradient Boost’ classifier is the best model.***


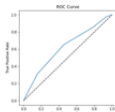
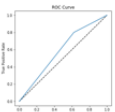
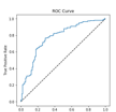
Model Evaluation	k-NN	Decision Tree	Gradient Boost 
Model accuracy on trained data	66.3%	65.6%	<b>75.3%</b>
AUC score for trained data, based on hyper-parameter selection	0.68	0.68	<b>0.72</b>
Hyper-parameters	n_neighbors = 36, p = 2, weights = uniform	criterion = Gini, max_depth = 3, min_samples_split = 3	max_depth = 2, max_features = 10, min_samples_split = 2
ROC Curve			
T-test results	k-NN vs DT, p-value > 0.05 <b>k-NN vs GB, p-value &lt; 0.05</b>	DT vs k-NN, p-value > 0.05 <b>DT vs GB, p-value &lt; 0.05</b>	<b>GB vs k-NN, p-value &lt; 0.05</b> <b>GB vs DT, p-value &lt; 0.05</b>
Model accuracy on test data	72.8%	70.0%	<b>75.8%</b>

Figure 2, Evaluation summary

To visualize the performance of all the classifiers on test data split the prediction results are presented using Confusion Matrix.

```
#Confusion Matrix
print("\nConfusion matrix for K-Nearest Neighbor")
print(metrics.confusion_matrix(y_test, pred_KNN))
print("\nConfusion matrix for Decision Tree")
print(metrics.confusion_matrix(y_test, pred_DT))
print("\nConfusion matrix for Gradient Boost")
print(metrics.confusion_matrix(y_test, pred_GB))
```

Confusion matrix for K-Nearest Neighbor

```
[[ 4 86]
 [ 3 207]]
```

Confusion matrix for Decision Tree

```
[[ 41 49]
 [ 30 180]]
```

Confusion matrix for Gradient Boost

```
[[ 30 60]
 [ 17 193]]
```

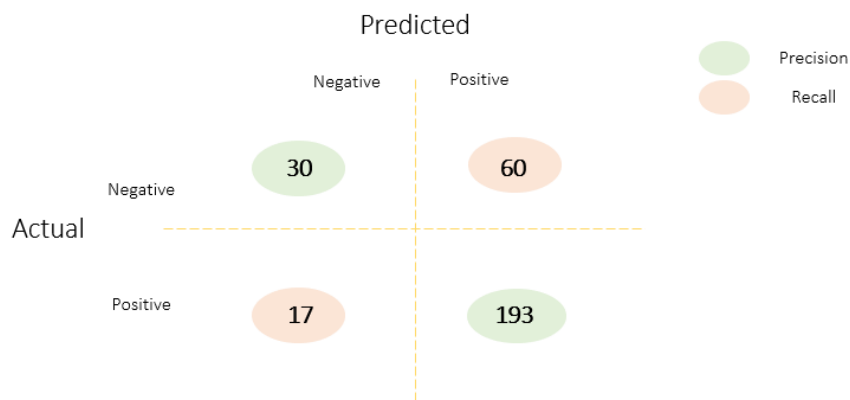


Figure 3, Confusion matrix for Gradient Boost classifier

For the gradient boost classifier, the precision score for negative (0) prediction is  $64\% = 30/(30+17)$ . In other words, the model predicted 47 (32+16) customers or loan applicants who have 'bad' credit risk, whereas the model prediction for positive (1), customers who have 'good' credit risk is  $76\% = 193/(60+194)$  and predicted 253 customers with 'good' credit risk.

The recall score for negative (0) is  $33\% = 30/(30+60)$ , which means there are 90 loan applicants classified as 'bad' credit risk, and model predicted 33% correctly. Also, the recall score for positive (1) is 92%, which means the model predicted 92% of the loan applicants correctly who are classified as 'good' credit risk.

Compared to the other confusion matrix for k-NN and Decision Tree, Gradient boost classifier has the highest precision and recall score.

From figure 3, there are two different wrong values.

**False positive**, where the model predicted that a loan applicant has a 'good' credit risk while in actual the loan applicant is a customer with a 'bad' credit risk. In this the bank would end up facilitation a loan to a potential defaulter.

**True negative**, where the model predicted that a loan applicant is a customer with 'bad' credit risk and in actual the customer has a 'good' credit risk. In this case the bank would lose the potential loan applicants resulting in lesser revenues or interest income.

As we can see that the lowest False positive and True negative values are for 'Gradient Boost classifier', it could be well concluded that so far it is the best model in terms of accuracy. However, the model performance and accurate prediction could be increased using more hyper-parameter tunings.

## Discussion

---

After experimenting three different algorithms for our classification problem using different parameter combinations, the best so far is the 'Gradient Boost classifier'. However, for all the algorithms the best accuracy scores are mostly on the low 70's, the accuracy would remain the same during the final testing stage which leads to the assumption that German Credit is relatively a harder dataset.

In the next step, the feature importance for the selected model is illustrated using '*sklearn.feature\_selection*'. The plot is helpful in providing data driven actionable insights for the German bank to minimize the degree of risk for different loan applicants.

```
#Feature selection and Ranking for top 8 most variables, post hyperparameter tuning

features = 8
model_gbx = GradientBoostingClassifier(max_features=10,min_samples_split=5,max_depth=2)
model_gbx.fit(x,y)
fs_indices_gbx = np.argsort(model_gbx.feature_importances_)[::-1][0:features]

best_features_gbx = Data_df.columns[fs_indices_gbx].values
feature_importances_gbx = model_gbx.feature_importances_[fs_indices_gbx]

import altair as alt

def plot_imp(best_features, scores, method_name, color):

    df = pd.DataFrame({'features': best_features,
                       'importances': scores})

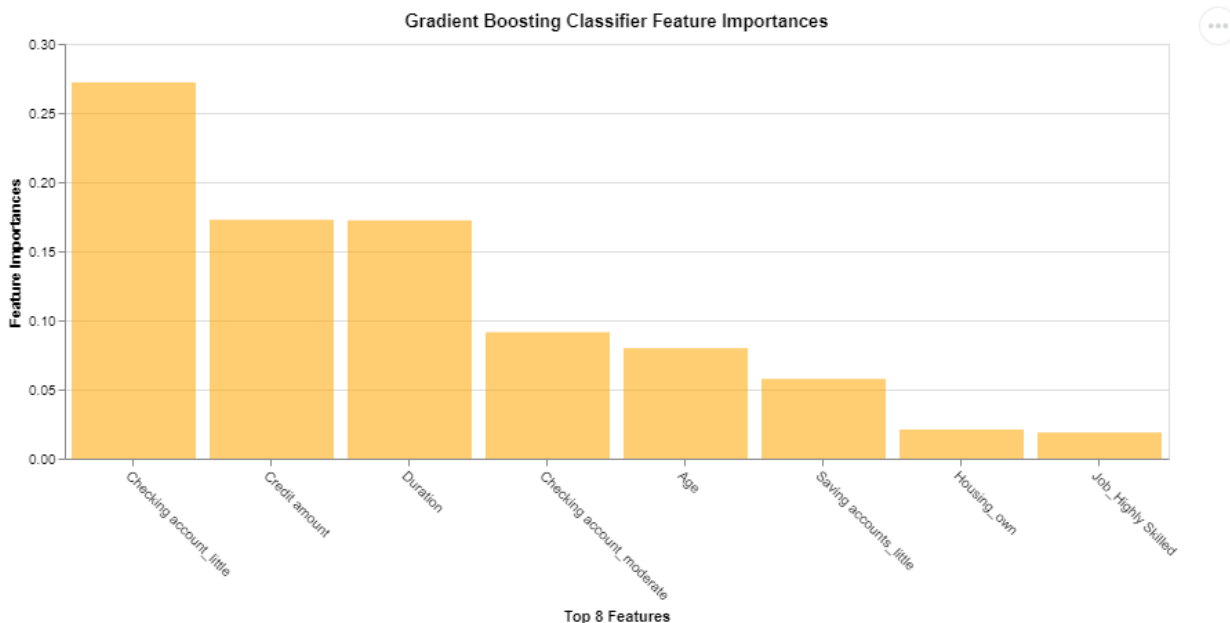
    chart = alt.Chart(df,
                       width=800,
                       title=method_name + ' Feature Importances'
    ).mark_bar(opacity=0.55,
               color=color).encode(
        alt.X('features', title='Top 8 Features', sort=None, axis=alt.AxisConfig(labelAngle=45)),
        alt.Y('importances', title='Feature Importances')
    )

    return chart

plot_imp(best_features_gbx, feature_importances_gbx, 'Gradient Boosting Classifier', 'orange')
```

Based on the output below, it is quite evident that variables 'checking account-little', 'credit amount', 'duration', 'checking-account-moderate', and 'Age' are some of the features with relatively higher importance in determining the creditworthiness of a loan applicant.

However, feature importance is one of the core concept which is usually performed after data-scaling and before model building step to remove the unwanted features that might result in under or overfitting of the model. In this case, since no. of variables are just 10, which were then converted to a total of 30 during feature encoding. Hence, we have not performed the feature importance earlier during the model building process and we have performed as a last step to understand how each variable is contributing towards the target feature.



Based on feature importance there are few recommendation to the bank representative that could help them to be more vigilant while lending.

- Checking account – little, a loan applicant with a little amount in their checking account should be thoroughly checked
- Credit amount – Proper valuation of the security, collateral or non-collateral, against which loan is going to be taken based on the credit amount.
- Duration – Repaying capacity of the borrower could be determined based on the tenure of the loan, shorter or longer.

## Conclusion

In this report we concluded that for all the three classifiers the best accuracy is in the low 70' but among those highest is for Gradient Boost, however the accuracy would remain the same even in final testing stage or during the cross-validation stage which leads us to the conclusion that German Credit Data is a harder dataset to work on. However, further improvements in the model performances is quite possible with more extensive fine-tuning. In this work, all the coding was done in python 3. This report demonstrates the applications that are followed in the world of data science for supervised machine learning techniques. Based on that

recommendations to the banks were made which would help them in future to predict or classify a loan applicant creditworthiness.

## References

---

[1] German Credit Risk, Kaggle.com. (2019). German Credit Risk.

Available at: <https://www.kaggle.com/uciml/german-credit>

[2] Generalization: Peril of Overfitting | Machine Learning Crash Course | Google Developers

Available at: <https://developers.google.com/machine-learning/crash-course/generalization/peril-of-overfitting>

[3] Schoonjans, F. (2019). ROC curve analysis with MedCalc.

Available at: <https://www.medcalc.org/manual/roc-curves.php>

[4] Learning, M. and Python, C. (2016). Complete Guide to Parameter Tuning in Gradient Boosting (GBM) in Python.

Available at: <https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/>