MATH2319 Machine Learning Project Phase 2
Churn Modelling : Detailed performance algorithm of algorithms

Names: Vishwas Krishna Reddy& Taruni Gadala
Student ID: s3712298 & s3730405

June 10, 2019

Contents

Chapter 1

Project Phase 2

1.1 Binary Classification:

1.1.1 Objective:

The objective of this case study is to fit and compare 3 different binary classifiers to predict whether customer exits the bank or not . Data sourced from the Kaggle. The descriptive features include 7 numeric and 2 nominal categorical features. The target feature has two classes defined as "0" means not exited and "1" means exited respectively. The full dataset contains about 10K observations.

This report is organized as follows: 1 Overview of Methodology. 2 Data preparation process and model evaluation strategy. 3 Hyperparameter tuning process for each classification algorithm. 4 Model Performance Comparison. 5 Limitations of our approach and possible solutions. 6 Summary.

1.1.2 Overview

Methodology

The following binary classifiers are used to predict the target feature: K-Nearest Neighbors (KNN), Decision trees (DT), and Naive Bayes (NB).

Modeling strategy begins by transforming the full dataset cleaned from project Phase I. This includes encoding categorical descriptive features and scaling of the descriptive features. We first randomly sample 5K rows from the full dataset of 10k rows and then split this sample into training and testing sets with aratio of 70:30 . After splitting the training data has 3500 rows and test data has 1500 rows.

1.1.3 Reading Dataset

Dataset is read directly from github account.

```
In [2]: import warnings
        warnings.filterwarnings("ignore")
        import pandas as pd #importing pandas library
        import seaborn as sns
        import matplotlib.pyplot as plt
        import numpy as np

        # set seed for reproducibility of results
        np.random.seed(999)
        churn_data = pd.read_csv("Churn_Modelling.csv",sep=',',decimal='.')
        churn_data.head()
        churn_data.shape
        churn_data.columns
```

```
Out[2]: Index(['RowNumber', 'CustomerId', 'Surname', 'CreditScore', 'Geography',
               'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCard',
               'IsActiveMember', 'EstimatedSalary', 'Exited'],
              dtype='object')
```

The Churn dataset consists of 10k observations. It has 13 descriptive features and the "Exited" target feature.

```
In [3]: churn_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
RowNumber          10000 non-null int64
CustomerId         10000 non-null int64
Surname            10000 non-null object
CreditScore        10000 non-null int64
Geography          10000 non-null object
Gender             10000 non-null object
Age                10000 non-null int64
Tenure             10000 non-null int64
Balance            10000 non-null float64
NumOfProducts      10000 non-null int64
HasCrCard          10000 non-null int64
IsActiveMember     10000 non-null int64
EstimatedSalary    10000 non-null float64
Exited             10000 non-null int64
dtypes: float64(2), int64(9), object(3)
memory usage: 1.1+ MB
```

```
In [4]: churn_data_categorical = churn_data.select_dtypes(include=['object']).copy() # Getting only cat
        for column in churn_data_categorical:
            print("The Column '{columnName}' has'{numOfCategories}' categories".format(columnName = col
```

```
The Column 'Surname' has'2932' categories
The Column 'Geography' has'3' categories
The Column 'Gender' has'2' categories
```

The dataset consists of 3 nominal categorical features ,'Surname','Geography'and 'Gender'

```
In [5]: churn_data=churn_data.drop(["Surname","RowNumber","CustomerId"], axis=1)
```

Here we have dropped the columns "Surname","RowNumber","CustomerId" as they are not much of use.

1.1.4 Checking for missing values:

```
In [6]: churn_data.isna().sum()
```

```
Out[6]: CreditScore        0
        Geography          0
        Gender             0
        Age                0
```

```
Tenure             0
Balance            0
NumOfProducts      0
HasCrCard          0
IsActiveMember     0
EstimatedSalary    0
Exited             0
dtype: int64
```

Dataset does not have any missing values as shown above.
5 randomly selected rows from the raw dataset are displayed below.

```
In [7]: churn_data.sample(n=5, random_state=999)
```

Out[7]:

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts
9031	541	France	Male	39	7	0.00	2
3462	428	France	Female	62	1	107735.93	1
3863	674	France	Female	28	3	0.00	1
1144	765	Germany	Male	43	4	148962.76	1
2692	751	France	Male	31	8	0.00	2

	HasCrCard	IsActiveMember	EstimatedSalary	Exited
9031	1	0	19823.02	0
3462	0	1	58381.77	0
3863	1	0	51536.99	0
1144	0	1	173878.87	1
2692	0	0	17550.49	0

1.1.5 Summary Statistics

Summary statistics ofthe full data are shown below:

```
In [8]: churn_data.describe(include='all')
```

Out[8]:

	CreditScore	Geography	Gender	Age	Tenure
count	10000.000000	10000	10000	10000.000000	10000.000000
unique	NaN	3	2	NaN	NaN
top	NaN	France	Male	NaN	NaN
freq	NaN	5014	5457	NaN	NaN
mean	650.528800	NaN	NaN	38.921800	5.012800
std	96.653299	NaN	NaN	10.487806	2.892174
min	350.000000	NaN	NaN	18.000000	0.000000
25%	584.000000	NaN	NaN	32.000000	3.000000
50%	652.000000	NaN	NaN	37.000000	5.000000
75%	718.000000	NaN	NaN	44.000000	7.000000
max	850.000000	NaN	NaN	92.000000	10.000000

	Balance	NumOfProducts	HasCrCard	IsActiveMember
count	10000.000000	10000.000000	10000.00000	10000.000000
unique	NaN	NaN	NaN	NaN
top	NaN	NaN	NaN	NaN
freq	NaN	NaN	NaN	NaN
mean	76485.889288	1.530200	0.70550	0.515100
std	62397.405202	0.581654	0.45584	0.499797
min	0.000000	1.000000	0.00000	0.000000

```
25%          0.000000       1.000000      0.00000        0.000000
50%      97198.540000       1.000000      1.00000        1.000000
75%     127644.240000       2.000000      1.00000        1.000000
max     250898.090000       4.000000      1.00000        1.000000

        EstimatedSalary        Exited
count      10000.000000  10000.000000
unique              NaN           NaN
top                 NaN           NaN
freq                NaN           NaN
mean      100090.239881      0.203700
std        57510.492818      0.402769
min           11.580000      0.000000
25%        51002.110000      0.000000
50%       100193.915000      0.000000
75%       149388.247500      0.000000
max       199992.480000      1.000000
```

1.1.6 Encoding Categorical Features

Before modeling all the categorical features are encoded.

Encoding the Target Feature

"Exited" feature is been removed from the full dataset and called as "target". The remaining of the features are the descriptive features which are called "Data".

In [9]:
```python
import numpy as np
data = churn_data.drop(columns='Exited')
target = churn_data['Exited']
target.value_counts()
```

Out[9]:
```
0    7963
1    2037
Name: Exited, dtype: int64
```

The classes in the target feature are not balanced .

Encoding the Descriptive Features

In [10]:
```python
categorical_cols = data.columns[data.dtypes==object].tolist()
categorical_cols
```

Out[10]:
```
['Geography', 'Gender']
```

Two of the descriptive features 'Geography' and 'Gender' are nominal,so label encoding is performed here.

In [11]:
```python
from sklearn.preprocessing import LabelEncoder,OneHotEncoder
label = LabelEncoder()
churn_data['Geography'] = label.fit_transform(churn_data['Geography'])
churn_data['Gender'] = label.fit_transform(churn_data['Gender'])
print(churn_data['Gender'].head(7))
print(churn_data['Geography'].head(7))
```

```
0    0
1    0
2    0
3    0
4    0
5    1
6    1
Name: Gender, dtype: int32
0    0
1    2
2    0
3    0
4    2
5    2
6    0
Name: Geography, dtype: int32
```

Label encoder can be used only when there are 2 levels , here Gender has 2 levels 0 or 1 but Geography has 3 levels 0,1,2 .

For 2 level categorical variable, we set the "drop_first" option to""True" and then encode the categorical variable into a single column of 0 or 1. Then regular one-hot encoding is done for categorical features with more than 2 levels.

```
In [12]: for col in categorical_cols:
             n = len(data[col].unique())
             if (n == 2):
                 data[col] = pd.get_dummies(data[col], drop_first=True)

         # use one-hot-encoding for categorical features with >2 levels
         data = pd.get_dummies(data)

In [13]: data.columns

Out[13]: Index(['CreditScore', 'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts',
                'HasCrCard', 'IsActiveMember', 'EstimatedSalary', 'Geography_France',
                'Geography_Germany', 'Geography_Spain'],
               dtype='object')

In [14]: data.sample(5, random_state=999)
```

Out[14]:

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	\
9031	541	1	39	7	0.00	2	1	
3462	428	0	62	1	107735.93	1	0	
3863	674	0	28	3	0.00	1	1	
1144	765	1	43	4	148962.76	1	0	
2692	751	1	31	8	0.00	2	0	

	IsActiveMember	EstimatedSalary	Geography_France	Geography_Germany	\
9031	0	19823.02	1	0	
3462	1	58381.77	1	0	
3863	0	51536.99	1	0	
1144	1	173878.87	0	1	
2692	0	17550.49	1	0	

```
      Geography_Spain
9031                0
3462                0
3863                0
1144                0
2692                0
```

1.1.7 Scaling of Features

Here Minmax scaling of the descriptive features is performed. To keep track of the column names a copy of Data is made first.

```
In [15]: import warnings
         warnings.filterwarnings("ignore")

         from sklearn import preprocessing
         Data_df = data.copy()
         Data = preprocessing.MinMaxScaler().fit_transform(data)
```

We can observe below that binary features are still kept as binary after the scaling.

```
In [16]: pd.DataFrame(Data, columns=Data_df.columns).sample(5, random_state=999)
```

```
Out[16]:      CreditScore  Gender       Age  Tenure   Balance  NumOfProducts  \
         9031        0.382     1.0  0.283784     0.7  0.000000       0.333333
         3462        0.156     0.0  0.594595     0.1  0.429401       0.000000
         3863        0.648     0.0  0.135135     0.3  0.000000       0.000000
         1144        0.830     1.0  0.337838     0.4  0.593718       0.000000
         2692        0.802     1.0  0.175676     0.8  0.000000       0.333333

               HasCrCard  IsActiveMember  EstimatedSalary  Geography_France  \
         9031        1.0             0.0         0.099067               1.0
         3462        0.0             1.0         0.291879               1.0
         3863        1.0             0.0         0.257652               1.0
         1144        0.0             1.0         0.869419               0.0
         2692        0.0             0.0         0.087703               1.0

               Geography_Germany  Geography_Spain
         9031                0.0              0.0
         3462                0.0              0.0
         3863                0.0              0.0
         1144                1.0              0.0
         2692                0.0              0.0
```

1.2 Feature Selection & Ranking

Using Random Forest Importance (RFI) the most important features are selected from the dataset. RFI is also included as a part of the pipeline to determine which number of features works best with each classifier used.

```
In [17]: from sklearn.ensemble import RandomForestClassifier

         numofFeatures = 10
         RFM = RandomForestClassifier(n_estimators=100)
```

```
RFM.fit(Data, target)
RFM_indices = np.argsort(RFM.feature_importances_)[::-1][0:numofFeatures]

RFM_Features = Data_df.columns[RFM_indices].values
RFM_Features
```

Out[17]:
```
array(['Age', 'EstimatedSalary', 'CreditScore', 'Balance',
       'NumOfProducts', 'Tenure', 'IsActiveMember', 'Geography_Germany',
       'Gender', 'HasCrCard'], dtype=object)
```

In [18]:
```
feature_importances_RFM = RFM.feature_importances_[RFM_indices]
feature_importances_RFM
```

Out[18]:
```
array([0.23617501, 0.14730065, 0.1444014 , 0.1427712 , 0.12932589,
       0.0817808 , 0.04146588, 0.02110595, 0.0186113 , 0.01786835])
```

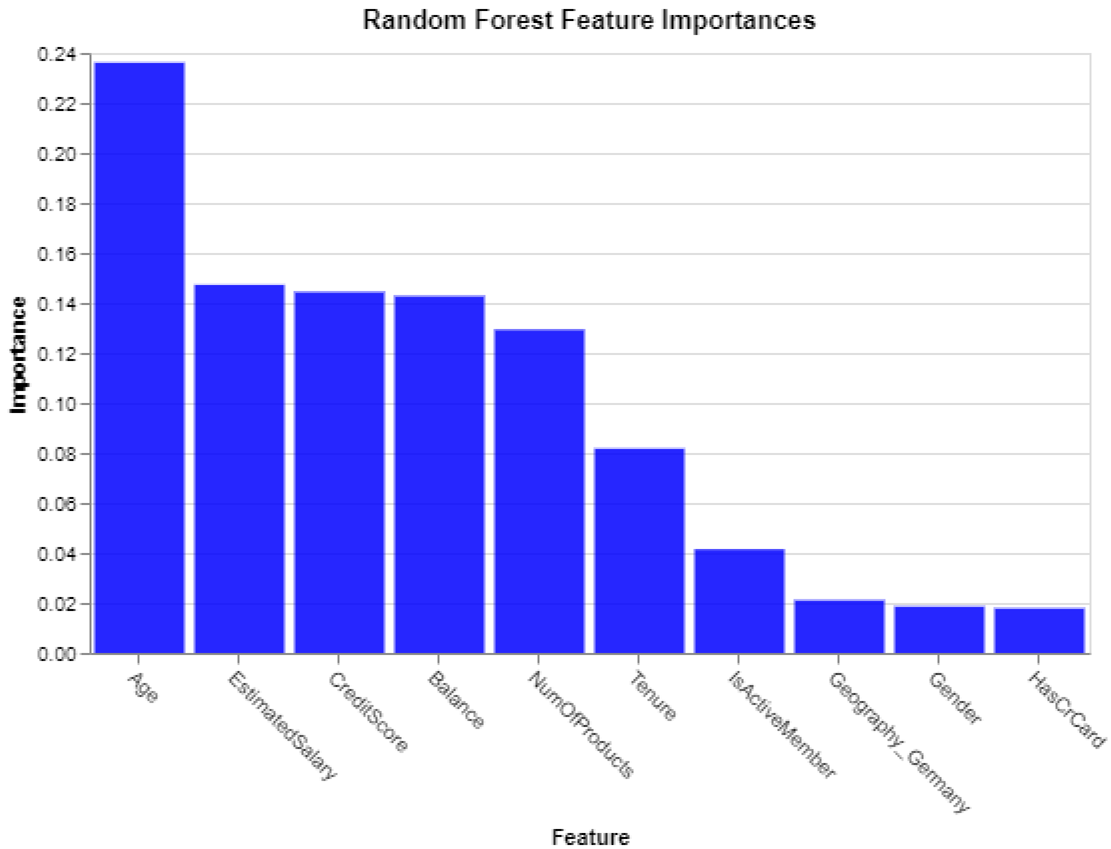
Visualising these importances:

In [19]:
```
import altair as alt
alt.renderers.enable('notebook')
def plot_graphs(best_features, scores, method_name, color):

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

    chart = alt.Chart(df,
                      width=500,
                      title=method_name + ' Feature Importances'
                     ).mark_bar(opacity=0.85,
                                color=color).encode(
        alt.X('features', title='Feature', sort=None, axis=alt.AxisConfig(labelAngle=45)),
        alt.Y('importances', title='Importance')
    )

    return chart
```

In [22]:
```
plot_graphs(RFM_Features, feature_importances_RFM, 'Random Forest', 'blue')
```

```
<vega.vegalite.VegaLite at 0x24ad4635390>
```

Out[22]:



The most important feature in the dataset is EstimatedSalary followed by CreditScore, Age and Balance.

1.3 Data Sampling & Train-Test Splitting

The original dataset has 10K rows, which is a lot. So, we would like to work with a small sample here with 5K rows. Thus, we will do the following: - Randomly select 5K rows from the full dataset. - Split this sample into train and test partitions with a 70:30 ratio using stratification.

```
In [23]: n_samples = 5000

         Data = pd.DataFrame(Data).sample(n=n_samples, random_state=8).values
         target = pd.DataFrame(target).sample(n=n_samples, random_state=8).values

         print(Data.shape)
         print(target.shape)
```

```
(5000, 12)
(5000, 1)
```

```
In [24]: from sklearn.model_selection import train_test_split

         train_data, test_data, train_target, test_target = train_test_split(Data,target,test_size = 0.3
                                                                             stratify = target)
```

```
print(train_data.shape)
print(test_data.shape)
```

```
(3500, 12)
(1500, 12)
```

Model Evaluation Strategy

Here first train and tune the selected models on 3500 rows of training data and then testing them on 1500 rows of test data.

For hyperparameter tuning ,5-fold stratified cross-validation evaluation method is used on the models.

```
In [25]: from sklearn.model_selection import StratifiedKFold, GridSearchCV

         SKF = StratifiedKFold(n_splits=5, random_state=999)
```

1.4 Hyperparameter Tuning

1.4.1 K-Nearest Neighbors (KNN)

Using Pipeline, we stack feature selection and grid search for KNN hyperparameter tuning via cross-validation. We will use the same Pipeline methodology for NB and DT. The KNN hyperparameters are as follows: 1.number of neighbors (n_neighbors) and 2.the distance metric p. For feature selection, we use the powerful Random Forest Importance (RFI) method with 100 estimators. A trick here is that we need a bit of coding so that we can make RFI feature selection as part of the pipeline. For this reason, we define the custom RFIFeatureSelector() class below to pass in RFI as a "step" to the pipeline.

```
In [26]: SKF.get_n_splits(Data, target)

Out[26]: 5

In [27]: from sklearn.base import BaseEstimator, TransformerMixin

         # custom function for RFI feature selection inside a pipeline
         # here we use n_estimators=100
         class RFIFeatureSelector(BaseEstimator, TransformerMixin):

             # class constructor
             # make sure class attributes end with a "_"
             # per scikit-learn convention to avoid errors
             def __init__(self, n_features_=10):
                 self.n_features_ = n_features_
                 self.fs_indices_ = None

             # override the fit function
             def fit(self, X, y):
                 from sklearn.ensemble import RandomForestClassifier
                 from numpy import argsort
                 rfi_model = RandomForestClassifier(n_estimators=100)
                 rfi_model.fit(X, y)
                 self.fs_indices_ = argsort(rfi_model.feature_importances_)[::-1][0:self.n_features_]
                 return self

             # override the transform function
             def transform(self, X, y=None):
                 return X[:, self.fs_indices_]
```

In [28]:
```python
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier

p_KNN = Pipeline(steps=[('rfi_fs', RFIFeatureSelector()),
                        ('knn', KNeighborsClassifier())])

params_p_KNN = {'rfi_fs__n_features_': [5, 10, Data.shape[1]],
                'knn__n_neighbors': [1, 10, 20, 40, 60, 100],
                'knn__p': [1, 2]}

gs_p_KNN = GridSearchCV(estimator=p_KNN,
                        param_grid=params_p_KNN,
                        cv=SKF,
                        refit=True,
                        n_jobs=-2,
                        scoring='roc_auc',
                        verbose=1)
```

In [29]:
```python
gs_p_KNN.fit(train_data, train_target);
```

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits


[Parallel(n_jobs=-2)]: Using backend LokyBackend with 7 concurrent workers.
[Parallel(n_jobs=-2)]: Done  36 tasks      | elapsed:    4.1s
[Parallel(n_jobs=-2)]: Done 180 out of 180 | elapsed:   16.4s finished
```

In [30]:
```python
gs_p_KNN.best_params_
```

Out[30]:
```
{'knn__n_neighbors': 40, 'knn__p': 1, 'rfi_fs__n_features_': 5}
```

In [31]:
```python
gs_p_KNN.best_score_
```

Out[31]:
```
0.7971710332268944
```

Here we can see that KNN model has a mean AUC score of 0.797. The best performing KNN selected 5 features with 40 nearest neighbors and =1.

In [32]:
```python
# custom function to format the search results as a Pandas data frame
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]
```

In [33]:
```
results_KNN = get_search_results(gs_p_KNN)
results_KNN.head()
```

Out[33]:

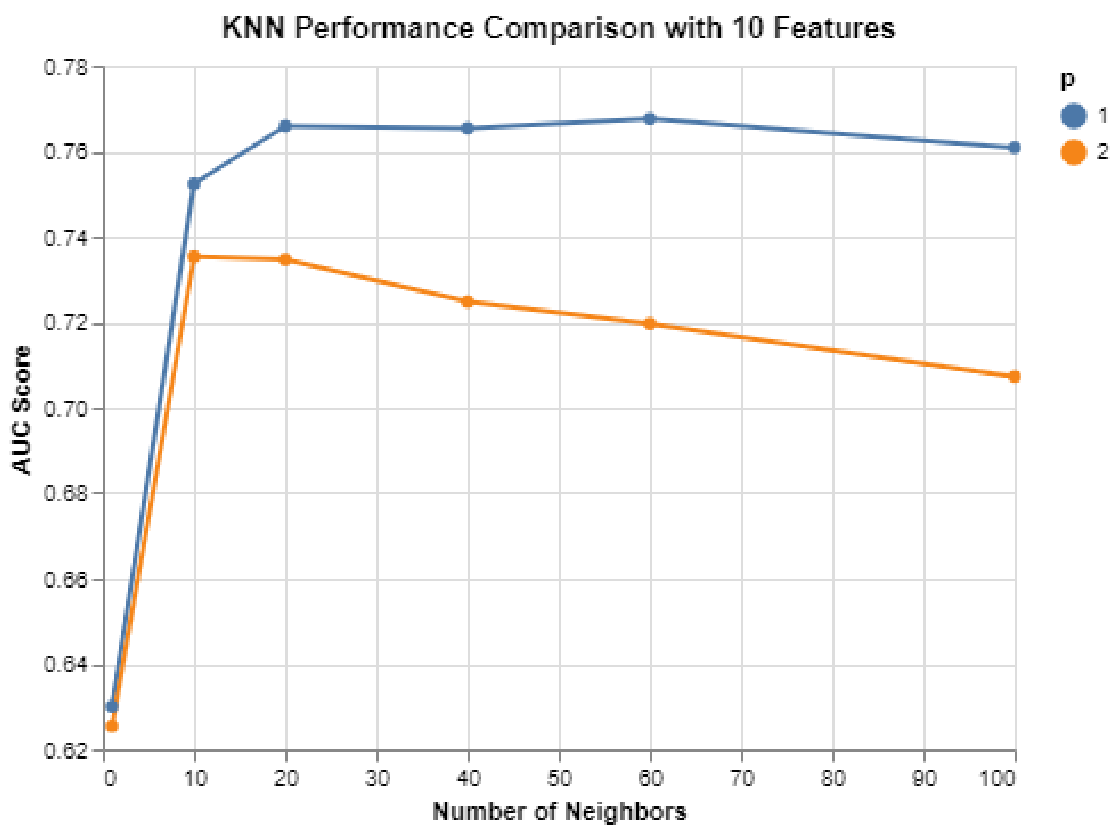
```
    mean_score  std_score  max_score  min_score  knn__n_neighbors  knn__p  \
18    0.797171   0.009873   0.814149   0.783493              40.0     1.0
24    0.795256   0.006173   0.806243   0.789424              60.0     1.0
30    0.793665   0.007772   0.808589   0.787276             100.0     1.0
12    0.793664   0.011144   0.814020   0.782371              20.0     1.0
15    0.793270   0.007361   0.804762   0.785686              20.0     2.0

    rfi_fs__n_features_
18                  5.0
24                  5.0
30                  5.0
12                  5.0
15                  5.0
```

Visualizing the results of grid search corresponding to 10 selected features:

In [34]:
```
import altair as alt

results_KNN_10_features = results_KNN[results_KNN['rfi_fs__n_features_'] == 10.0]

alt.Chart(results_KNN_10_features,
          title='KNN Performance Comparison with 10 Features'
         ).mark_line(point=True).encode(
    alt.X('knn__n_neighbors', title='Number of Neighbors'),
    alt.Y('mean_score', title='AUC Score', scale=alt.Scale(zero=False)),
    alt.Color('knn__p:N', title='p')
)
```

```
<vega.vegalite.VegaLite at 0x24ad5a55e10>
```

Out[34]:



1.5 Naive Bayes

Here we are implementing a Gaussian Naive Bayes model. -first performing a power transformation on the input data before model fitting. -Optimizing "var_smoothing" -Conducting the grid search in the "logspace" .

In [35]:
```
from sklearn.preprocessing import PowerTransformer
Data_sample_train_transformed = PowerTransformer().fit_transform(train_data)
```

In [36]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import RandomizedSearchCV

p_NB = Pipeline([('rfi_fs', RFIFeatureSelector()),
                 ('nb', GaussianNB())])

params_p_NB = {'rfi_fs__n_features_': [10, 20, Data.shape[1]],
               'nb__var_smoothing': np.logspace(1,-3, num=200)}

n_iter_search = 20
gs_p_NB = RandomizedSearchCV(estimator=p_NB,
                             param_distributions=params_p_NB,
                             cv=SKF,
                             refit=True,
                             n_jobs=-2,
```

```
                                      scoring='roc_auc',
                                      n_iter=n_iter_search,
                                      verbose=1)

        gs_p_NB.fit(Data_sample_train_transformed, train_target);
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-2)]: Using backend LokyBackend with 7 concurrent workers.
[Parallel(n_jobs=-2)]: Done  36 tasks      | elapsed:    2.3s
[Parallel(n_jobs=-2)]: Done 100 out of 100 | elapsed:    5.7s finished
```

In [37]:
```
gs_p_NB.best_params_
```

Out[37]:
```
{'rfi_fs__n_features_': 10, 'nb__var_smoothing': 0.007316807143427192}
```

In [38]:
```
gs_p_NB.best_score_
```

Out[38]:
```
0.7896100232113775
```

NB shows an AUC score of 0.789 with 10 features which is less than that of KNN. Paired TTest is performed to conclude which is better model.

In [39]:
```
results_NB = get_search_results(gs_p_NB)
results_NB.head()
```

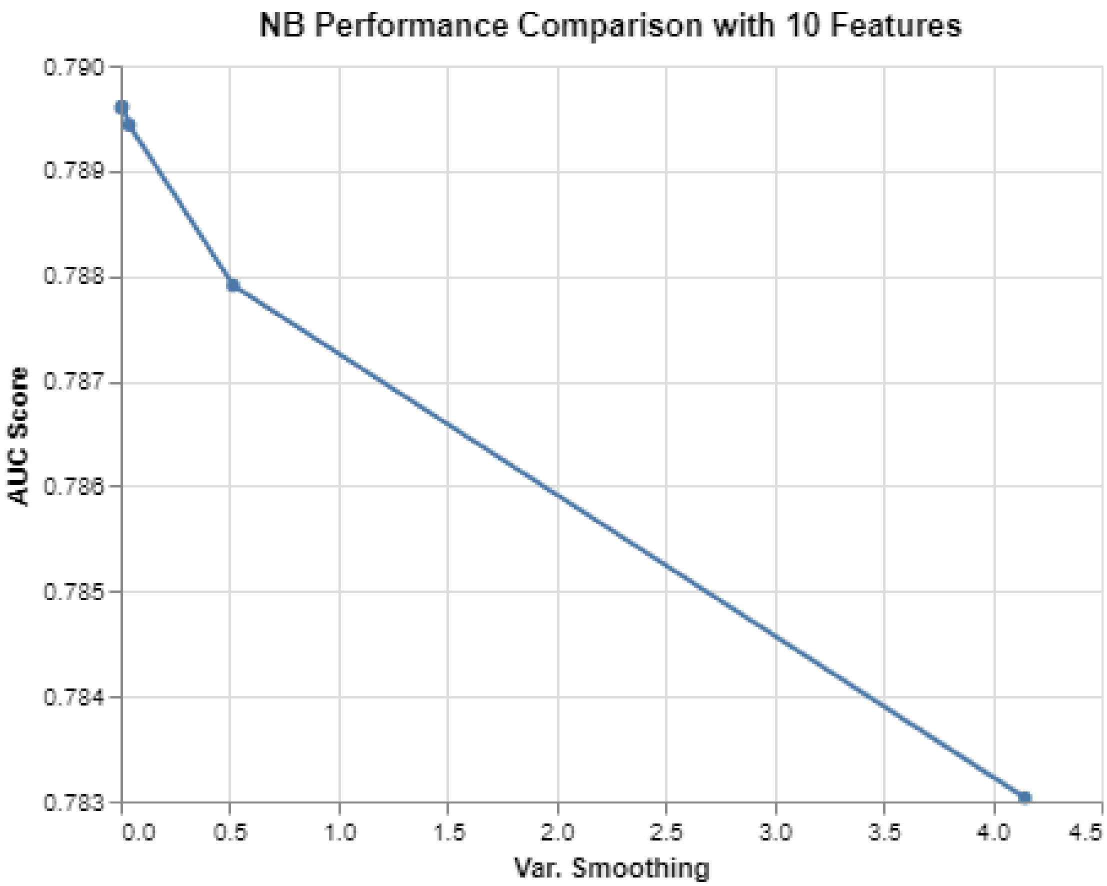
Out[39]:
```
    mean_score  std_score  max_score  min_score  rfi_fs__n_features_  \
14    0.789610   0.018165   0.814373   0.765873                 10.0
9     0.789600   0.018189   0.814386   0.765822                 10.0
13    0.789436   0.018071   0.814194   0.765693                 10.0
2     0.787910   0.017207   0.811475   0.763924                 10.0
10    0.783031   0.015264   0.803024   0.759935                 10.0

    nb__var_smoothing
14           0.007317
9            0.004009
13           0.040555
2            0.517092
10           4.150405
```

Visualising these results:

In [40]:
```
results_NB_10_features = results_NB[results_NB['rfi_fs__n_features_'] == 10.0]

alt.Chart(results_NB_10_features,
          title='NB Performance Comparison with 10 Features'
         ).mark_line(point=True).encode(
    alt.X('nb__var_smoothing', title='Var. Smoothing'),
    alt.Y('mean_score', title='AUC Score', scale=alt.Scale(zero=False))
)
```

```
<vega.vegalite.VegaLite at 0x24ad6c46f60>
```

Out[40]:



Decision Trees (DT)

Here a DT is built using gini index to maximize information gain. The aim to determine the combinations of maximum depthand minimum sample split.

```
In [41]: from sklearn.tree import DecisionTreeClassifier

         p_DT = Pipeline([('rfi_fs', RFIFeatureSelector()),
                          ('dt', DecisionTreeClassifier(criterion='gini'))])

         params_p_DT = {'rfi_fs__n_features_': [5, 7, 10, Data.shape[1]],
                        'dt__max_depth': [3, 4, 5],
                        'dt__min_samples_split': [2, 5]}

         gs_p_DT = GridSearchCV(estimator=p_DT,
                                param_grid=params_p_DT,
                                cv=SKF,
                                refit=True,
                                n_jobs=-2,
                                scoring='roc_auc',
                                verbose=1)

         gs_p_DT.fit(train_data, train_target);
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
```

```
[Parallel(n_jobs=-2)]: Using backend LokyBackend with 7 concurrent workers.
[Parallel(n_jobs=-2)]: Done  36 tasks      | elapsed:    2.4s
[Parallel(n_jobs=-2)]: Done 120 out of 120 | elapsed:    7.1s finished
```

In [42]:
```
gs_p_DT.best_params_
```

Out[42]:
```
{'dt__max_depth': 5, 'dt__min_samples_split': 5, 'rfi_fs__n_features_': 12}
```

In [43]:
```
gs_p_DT.best_score_
```

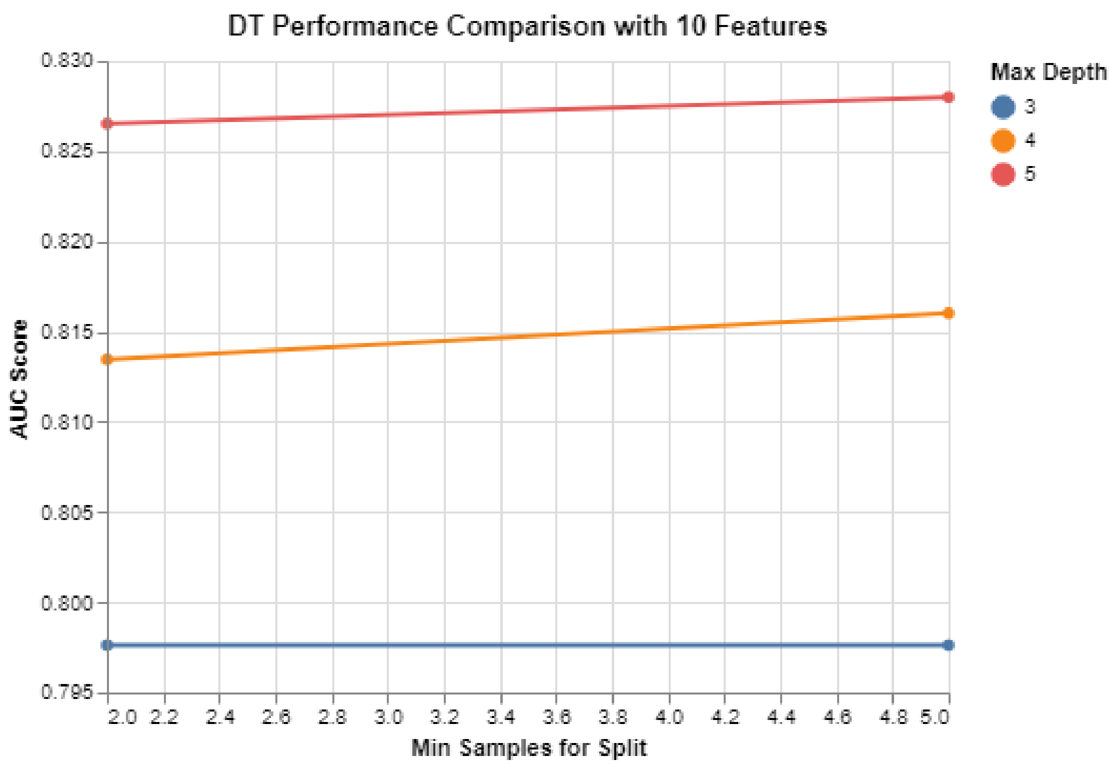
Out[43]:
```
0.8296156657561651
```

DT has a maximum depth of 5 and minimum split value of 5 samples and the best features are 12 ,with an AUC score of 0.829. Visualization of the search results is below:

In [44]:
```
results_DT = get_search_results(gs_p_DT)

results_DT_10_features = results_DT[results_DT['rfi_fs__n_features_'] == 10.0]

alt.Chart(results_DT_10_features,
          title='DT Performance Comparison with 10 Features'
         ).mark_line(point=True).encode(
    alt.X('dt__min_samples_split', title='Min Samples for Split'),
    alt.Y('mean_score', title='AUC Score', scale=alt.Scale(zero=False)),
    alt.Color('dt__max_depth:N', title='Max Depth')
)
```

```
<vega.vegalite.VegaLite at 0x24ad6c62be0>
```

Out[44]:



```
In [45]: params_p_DT2 = {'rfi_fs__n_features_': [10],
                         'dt__max_depth': [5, 10, 15],
                         'dt__min_samples_split': [5, 50, 100, 150]}

         gs_p_DT2 = GridSearchCV(estimator=p_DT,
                                 param_grid=params_p_DT2,
                                 cv=SKF,
                                 refit=True,
                                 n_jobs=-2,
                                 scoring='roc_auc',
                                 verbose=1)

         gs_p_DT2.fit(train_data, train_target);
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits


[Parallel(n_jobs=-2)]: Using backend LokyBackend with 7 concurrent workers.
[Parallel(n_jobs=-2)]: Done  36 tasks      | elapsed:    2.4s
[Parallel(n_jobs=-2)]: Done  60 out of  60 | elapsed:    3.5s finished
```

```
In [46]: gs_p_DT2.best_params_
```

```
Out[46]: {'dt__max_depth': 15, 'dt__min_samples_split': 150, 'rfi_fs__n_features_': 10}
```

```
In [47]: gs_p_DT2.best_score_
```

```
Out[47]: 0.8378666051116327
```

There is not much difference with AUC score of DT and DT2 . DT2 has an AUC score of 0.838,maximum depth is 15 ,samples split is 150, and best features are 10.

Compared to the other 2 models DT has best AUC score .

```
In [48]: results_DT = get_search_results(gs_p_DT2)
         results_DT.head()
```

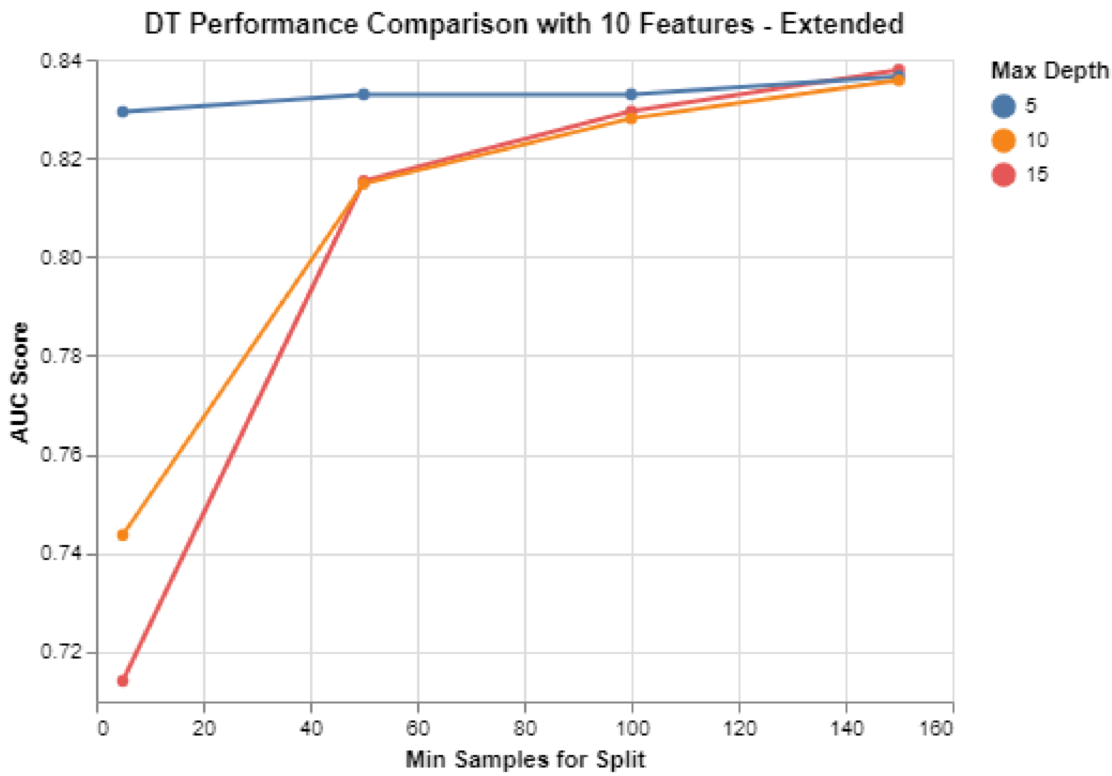
```
Out[48]:     mean_score  std_score  max_score  min_score  dt__max_depth  \
         11    0.837867   0.004995   0.842034   0.828178           15.0
         3     0.836532   0.012905   0.855224   0.820984            5.0
         7     0.835740   0.006273   0.844176   0.825915           10.0
         2     0.832886   0.016476   0.854788   0.814380            5.0
         1     0.832886   0.016229   0.854788   0.814380            5.0

             dt__min_samples_split  rfi_fs__n_features_
         11                  150.0                 10.0
         3                   150.0                 10.0
         7                   150.0                 10.0
         2                   100.0                 10.0
         1                    50.0                 10.0
```

Visualization the new search results:

In [49]:
```
results_DT_10_features = results_DT[results_DT['rfi_fs__n_features_'] == 10.0]

alt.Chart(results_DT_10_features,
          title='DT Performance Comparison with 10 Features - Extended'
         ).mark_line(point=True).encode(
    alt.X('dt__min_samples_split', title='Min Samples for Split'),
    alt.Y('mean_score', title='AUC Score', scale=alt.Scale(zero=False)),
    alt.Color('dt__max_depth:N', title='Max Depth')
)
```

<vega.vegalite.VegaLite at 0x24ad6c65f98>

Out[49]:



Chapter 2

Performance Comparison

< First using the train data each one of the 3 classifiers are been optimized. Now fitting those optimized models on the test data . Here we are performing a pairwise t-tests to check if there is any difference between the performance of any two classifiers which are optimized is statistically significant . First, perform StratifiedKFold cross-validation on each best model. Next conduct a paired t-test for the AUC score between the following model combinations:

- KNN vs. NB,
- KNN vs. DT, and
- DT vs. NB.

In [50]:
```
from sklearn.model_selection import cross_val_score

cv_method_ttest = StratifiedKFold(n_splits=10, random_state=111)

cv_results_KNN = cross_val_score(estimator=gs_p_KNN.best_estimator_,
                                 X=test_data,
                                 y=test_target,
                                 cv=cv_method_ttest,
                                 n_jobs=-2,
                                 scoring='roc_auc')
cv_results_KNN.mean()
```

Out[50]: 0.7647910294164213

In [51]:
```
Data_sample_test_transformed = PowerTransformer().fit_transform(test_data)

cv_results_NB = cross_val_score(estimator=gs_p_NB.best_estimator_,
                                X=Data_sample_test_transformed,
                                y=test_target,
                                cv=cv_method_ttest,
                                n_jobs=-2,
                                scoring='roc_auc')
cv_results_NB.mean()
```

Out[51]: 0.780480407523511

In [52]:
```
cv_results_DT = cross_val_score(estimator=gs_p_DT2.best_estimator_,
                                X=test_data,
                                y=test_target,
                                cv=cv_method_ttest,
                                n_jobs=-2,
```

```
                                     scoring='roc_auc')
cv_results_DT.mean()
```

Out[52]: 0.7767327348722333

Performing following t-test on the test data:

In [53]:
```
from scipy import stats

print(stats.ttest_rel(cv_results_KNN, cv_results_NB))
print(stats.ttest_rel(cv_results_DT, cv_results_KNN))
print(stats.ttest_rel(cv_results_DT, cv_results_NB))
```

```
Ttest_relResult(statistic=-0.6212106932020908, pvalue=0.5498688990624216)
Ttest_relResult(statistic=0.4459895260999597, pvalue=0.6661410151408587)
Ttest_relResult(statistic=-0.3058274573434334, pvalue=0.7666918485390652)
```

A p-value is more than 0.05 .The data given is not enough for the validation .As p-value is not statistically significant.

In [54]:
```
pred_KNN = gs_p_KNN.predict(test_data)
```

In [55]:
```
Data_test_transformed = PowerTransformer().fit_transform(test_data)
pred_NB = gs_p_NB.predict(Data_test_transformed)
```

In [56]:
```
pred_DT = gs_p_DT2.predict(test_data)
```

In [57]:
```
from sklearn import metrics
print("\nClassification report for K-Nearest Neighbor")
print(metrics.classification_report(test_target, pred_KNN))
print("\nClassification report for Naive Bayes")
print(metrics.classification_report(test_target, pred_NB))
print("\nClassification report for Decision Tree")
print(metrics.classification_report(test_target, pred_DT))
```

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

           0       0.81      0.99      0.89      1202
           1       0.66      0.09      0.16       298

   micro avg       0.81      0.81      0.81      1500
   macro avg       0.74      0.54      0.53      1500
weighted avg       0.78      0.81      0.75      1500


Classification report for Naive Bayes
              precision    recall  f1-score   support

           0       0.85      0.95      0.89      1202
           1       0.59      0.31      0.41       298

   micro avg       0.82      0.82      0.82      1500
   macro avg       0.72      0.63      0.65      1500
```

```
weighted avg       0.80      0.82      0.80      1500


Classification report for Decision Tree
              precision    recall  f1-score   support

           0       0.87      0.94      0.91      1202
           1       0.66      0.46      0.54       298

   micro avg       0.84      0.84      0.84      1500
   macro avg       0.77      0.70      0.72      1500
weighted avg       0.83      0.84      0.83      1500
```

Confusion matrices are given below:

```
In [58]: from sklearn import metrics
         print("\nConfusion matrix for K-Nearest Neighbor")
         print(metrics.confusion_matrix(test_target, pred_KNN))
         print("\nConfusion matrix for Naive Bayes")
         print(metrics.confusion_matrix(test_target, pred_NB))
         print("\nConfusion matrix for Decision Tree")
         print(metrics.confusion_matrix(test_target, pred_DT))
```

```
Confusion matrix for K-Nearest Neighbor
[[1188   14]
 [ 271   27]]

Confusion matrix for Naive Bayes
[[1138   64]
 [ 206   92]]

Confusion matrix for Decision Tree
[[1131   71]
 [ 162  136]]
```

Here recall score which is equivalent to the true positive rate is considered as the performance metric. In this context, DT is the best performer since it has produced the highest recall score . If we check with the AUC score DT has the highest AUC score.According to our findings DT is the best performer when it comes to the AUC score and recall score

Chapter 3

Summary

The Decision Tree model with 10 best features which are selected by Random Forest Importance (RFI) produces the highest cross-validation AUC score on the train data. But when evaluated on the test data the Decision Tree model is almost closer to Naive Bayes with respect to AUC. However, the Decision Tree model gives the highest recall score on the test data. We can also observe that the p-value is greater than 0.05, which indicates the features that we used from the dataset to check the performance are not enough.

Chapter 4

References

[1] Shruti Iyer. Churn Modelling: Classification Data Set