Project 2 - Classification

Machine Learning Spring 2021

Default of Credit card

Source of Dataset

https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients (https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients)

Feature Details

LIMIT_BAL: Amount of given credit in NT dollars (includes individual and family/supplementary credit)

SEX: Gender (1=male, 2=female)

EDUCATION: (0=?, 1=graduate school, 2=university, 3=high school, 4=others, 5=unknown, 6=unknown)

MARRIAGE: Marital status (0=?,1=married, 2=single, 3=others)

AGE: Age in years

PAY0: Repayment status in September, 2005 (-1=pay duly, 1=payment delay for one month, 2=payment delay for two months, ... 8=payment delay for eight months, 9=payment delay for nine months and above)

PAY2: Repayment status in August, 2005

PAY3: Repayment status in July, 2005

PAY4: Repayment status in June, 2005

PAY5: Repayment status in May, 2005

PAY6: Repayment status in April, 2005

BILL_AMT1: Bill in September, 2005 (NT dollar)

BILL_AMT2: Bill in August, 2005 (NT dollar)

BILL_AMT3: Bill in July, 2005 (NT dollar)

BILL_AMT4: Bill in June, 2005 (NT dollar)

BILL_AMT5: Bill statement in May, 2005 (NT dollar)

BILL_AMT6: Bill statement in April, 2005 (NT dollar)

PAY_AMT1: Amount of previous payment in September, 2005 (NT dollar)

PAY_AMT2: Amount of previous payment in August, 2005 (NT dollar)

PAY_AMT3: Amount of previous payment in July, 2005 (NT dollar)

PAY_AMT4: Amount of previous payment in June, 2005 (NT dollar)

PAY_AMT5: Amount of previous payment in May, 2005 (NT dollar)

PAY_AMT6: Amount of previous payment in April, 2005 (NT dollar)

Default_Payment: Default payment for next month (1=yes, 0=no)

Classification Task:

1) Apply two voting classifiers - one with hard voting and one with soft voting.

2) Apply any two models with bagging and any two models with pasting.

3) Apply any two models with AdaBoost boosting.

4) Apply one model with gradient boosting.

5) Apply PCA on data and then apply all the models in project 1 again on data you get from PCA. Compare your results with results in project 1. You don't need to apply all the models twice. Just copy the result table from project 1, prepare a similar table for all the models after PCA and compare both tables. Does PCA help in getting better results?

6) Apply deep learning models covered in class.

7) In all the classification tasks, consider the evaluation function you used in Project 1.

Import Modules

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn import model_selection
from sklearn.preprocessing import MinMaxScaler,StandardScaler
from sklearn import svm
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import recall_score, precision_score, f1_score
from sklearn.metrics import precision_recall_curve
from sklearn import svm
import seaborn as sns
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

```

Load Data

In [2]:

```
dcc = pd.read_excel('default of credit card clients.xls',skiprows=1)
dcc.drop(['ID'], axis=1, inplace=True)
```

In [3]:

```
dcc.head(5)
```

Out[3]:

	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_5	...
0	20000	2	2	1	24	2	2	-1	-1	-2	...
1	120000	2	2	2	26	-1	2	0	0	0	...
2	90000	2	2	2	34	0	0	0	0	0	...
3	50000	2	2	1	37	0	0	0	0	0	...
4	50000	1	2	1	57	-1	0	-1	0	0	...

5 rows × 24 columns

In [4]:

```
dcc.describe()
```

Out[4]:

	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0
count	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000
mean	167484.322667	1.603733	1.853133	1.551867	35.485500	-0.016700
std	129747.661567	0.489129	0.790349	0.521970	9.217904	1.123802
min	10000.000000	1.000000	0.000000	0.000000	21.000000	-2.000000
25%	50000.000000	1.000000	1.000000	1.000000	28.000000	-1.000000
50%	140000.000000	2.000000	2.000000	2.000000	34.000000	0.000000
75%	240000.000000	2.000000	2.000000	2.000000	41.000000	0.000000
max	1000000.000000	2.000000	6.000000	3.000000	79.000000	8.000000

8 rows × 24 columns

In [5]:

```
dcc.isnull().sum()
```

Out[5]:

```
LIMIT_BAL                   0
SEX                         0
EDUCATION                   0
MARRIAGE                    0
AGE                         0
PAY_0                       0
PAY_2                       0
PAY_3                       0
PAY_4                       0
PAY_5                       0
PAY_6                       0
BILL_AMT1                   0
BILL_AMT2                   0
BILL_AMT3                   0
BILL_AMT4                   0
BILL_AMT5                   0
BILL_AMT6                   0
PAY_AMT1                    0
```

The original dataset does not have any null value, hence we have to manually insert null values randomly in the feature set.

Let's delete 5.5% of random values from the below mentioned columns , Categorical variables and Age are left alone

In [6]:

```
dcc.columns[5:25]
```

Out[6]:

```
Index(['PAY_0', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1',
       'BILL_AMT2', 'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6',
       'PAY_AMT1', 'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT
6',
       'default payment next month'],
      dtype='object')
```

In [7]:

```
np.random.seed(seed=0)
masking_array= np.random.randint(100,size=(dcc.shape[0], 19)) < 94.5
masking_array
```

Out[7]:

```
array([[ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       ...,
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True, False,  True]])
```

In [8]:

```
dcc[dcc.columns[5:25]]=dcc[dcc.columns[5:25]].where(masking_array, np.nan)
```

Checking the missing values in each columns

In [9]:

```
dcc.isna().any()[lambda x: x]
```

Out[9]:

```
PAY_0                         True
PAY_2                         True
PAY_3                         True
PAY_4                         True
PAY_5                         True
PAY_6                         True
BILL_AMT1                     True
BILL_AMT2                     True
BILL_AMT3                     True
BILL_AMT4                     True
BILL_AMT5                     True
BILL_AMT6                     True
PAY_AMT1                      True
PAY_AMT2                      True
PAY_AMT3                      True
PAY_AMT4                      True
PAY_AMT5                      True
PAY_AMT6                      True
default payment next month    True
dtype: bool
```

Now we need to check the percentage of total missing data in the dataset, for this purpose we will create a function

In [10]:

```
def missing_data_percentage(df):
    x = ['column_name','missing_values', 'missing_in_percentage']
    missing_data = pd.DataFrame(columns=x)
    columns = dcc.columns
    for col in columns:
        iscolumn_name = col
        ismissing_values = dcc[col].isnull().sum()
        ismissing_in_percentage = (dcc[col].isnull().sum()/dcc[col].shape[0])*100

        missing_data.loc[len(missing_data)] = [iscolumn_name, ismissing_values, ismiss
    print(missing_data.round(2))
```

In [11]:

```
missing_data_percentage(dcc)
```

```
           column_name missing_values  missing_in_percentage
0            LIMIT_BAL              0                   0.00
1                  SEX              0                   0.00
2            EDUCATION              0                   0.00
3             MARRIAGE              0                   0.00
4                  AGE              0                   0.00
5                PAY_0           1496                   4.99
6                PAY_2           1490                   4.97
7                PAY_3           1550                   5.17
8                PAY_4           1502                   5.01
9                PAY_5           1437                   4.79
10               PAY_6           1474                   4.91
11           BILL_AMT1           1459                   4.86
12           BILL_AMT2           1473                   4.91
13           BILL_AMT3           1514                   5.05
14           BILL_AMT4           1432                   4.77
15           BILL_AMT5           1462                   4.87
16           BILL_AMT6           1530                   5.10
17            PAY_AMT1           1466                   4.89
```

Data Transformation

In [12]:

```
dcc.rename(columns={'default payment next month':'Default_Payment'}, inplace=True)
```

a) Let's address the redundant data in the columns to simplify data analysis.

Let's check the column "EDUCATION" first. As mentioned in the dataset description EDUCATION: (0=?, 1=graduate school, 2=university, 3=high school, 4=others, 5=unknown, 6=unknown). In this case we can combine 0,4,5,6 under a common lablel '0'.

In [13]:

```
print(dcc.apply(lambda col: col.unique()))
```

```
LIMIT_BAL          [20000, 120000, 90000, 50000, 500000, 100000, ...
SEX                                                            [2, 1]
EDUCATION                                         [2, 1, 3, 5, 4, 6, 0]
MARRIAGE                                                   [1, 2, 3, 0]
AGE                [24, 26, 34, 37, 57, 29, 23, 28, 35, 51, 41, 3...
PAY_0              [2.0, -1.0, 0.0, nan, 1.0, -2.0, 3.0, 4.0, 8.0...
PAY_2              [2.0, 0.0, nan, -2.0, -1.0, 3.0, 5.0, 7.0, 4.0...
PAY_3              [-1.0, 0.0, 2.0, -2.0, nan, 3.0, 4.0, 6.0, 7.0...
PAY_4              [-1.0, 0.0, -2.0, 2.0, nan, 3.0, 4.0, 5.0, 7.0...
PAY_5              [-2.0, 0.0, -1.0, 2.0, nan, 3.0, 5.0, 4.0, 7.0...
PAY_6              [-2.0, 2.0, 0.0, -1.0, nan, 3.0, 4.0, 6.0, 7.0...
BILL_AMT1          [3913.0, 2682.0, 29239.0, 46990.0, 8617.0, 644...
BILL_AMT2          [3102.0, 1725.0, 14027.0, 48233.0, 5670.0, 570...
BILL_AMT3          [689.0, 2682.0, 13559.0, 49291.0, 35835.0, 576...
BILL_AMT4          [0.0, 3272.0, 14331.0, 28314.0, 20940.0, 19394...
BILL_AMT5          [0.0, 3455.0, 14948.0, 28959.0, 19146.0, nan, ...
BILL_AMT6          [0.0, 3261.0, 15549.0, 29547.0, 19131.0, 20024...
PAY_AMT1           [0.0, 1518.0, 2000.0, 2500.0, 55000.0, 380.0, ...
PAY_AMT2           [689.0, 1000.0, 1500.0, 2019.0, 36681.0, 1815....
```

In [14]:

```
dcc.loc[dcc.EDUCATION >= 4, 'EDUCATION'] = 0
```

In [15]:

```
#checking the unique values in the column 'EDUCATION'
dcc['EDUCATION'].unique()
```

Out[15]:

```
array([2, 1, 3, 0], dtype=int64)
```

Now let's check the next column, 'MARRIAGE'. From the dataset description we realize that

MARRIAGE: Marital status (0=?,1=married, 2=single, 3=others) In this case we can combine 0,3 under a common lablel '0'.

In [16]:

```
dcc.loc[dcc.MARRIAGE == 3, 'MARRIAGE'] = 0
```

In [17]:

```
#checking the unique values in the column 'EDUCATION'
dcc['MARRIAGE'].unique()
```

Out[17]:

```
array([1, 2, 0], dtype=int64)
```

b) Let's fill up the missing values now

For columns (PAY_0, PAY_2, PAY_4, PAY_5, PAY_6, default payment next month) the values are

categorical because it's best representation of the central tendency without creating ambigūties in case of categorical values

In [18]:

```
dcc['PAY_0'].fillna(dcc['PAY_0'].mode()[0],inplace= True)
dcc['PAY_2'].fillna(dcc['PAY_2'].mode()[0],inplace= True)
dcc['PAY_3'].fillna(dcc['PAY_3'].mode()[0],inplace= True)
dcc['PAY_4'].fillna(dcc['PAY_4'].mode()[0],inplace= True)
dcc['PAY_5'].fillna(dcc['PAY_5'].mode()[0],inplace= True)
dcc['PAY_6'].fillna(dcc['PAY_6'].mode()[0],inplace= True)
```

In [19]:

```
dcc['Default_Payment'].fillna(dcc['Default_Payment'].mode()[0],inplace= True)
```

For columns BILL AMOUNT and PAY AMOUNT the values are continuous and for columns containing countinuous values no matter how many times we add mean, we are in a way replacing the unknown value by the average of observed data for that variable.

In [20]:

```
dcc['PAY_AMT1'].fillna(dcc['PAY_AMT1'].mean(),inplace= True)
dcc['PAY_AMT2'].fillna(dcc['PAY_AMT2'].mean(),inplace= True)
dcc['PAY_AMT3'].fillna(dcc['PAY_AMT3'].mean(),inplace= True)
dcc['PAY_AMT4'].fillna(dcc['PAY_AMT4'].mean(),inplace= True)
dcc['PAY_AMT5'].fillna(dcc['PAY_AMT5'].mean(),inplace= True)
dcc['PAY_AMT6'].fillna(dcc['PAY_AMT6'].mean(),inplace= True)
```

In [21]:

```
dcc['BILL_AMT1'].fillna(dcc['BILL_AMT1'].mean(),inplace= True)
dcc['BILL_AMT2'].fillna(dcc['BILL_AMT1'].mean(),inplace= True)
dcc['BILL_AMT3'].fillna(dcc['BILL_AMT1'].mean(),inplace= True)
dcc['BILL_AMT4'].fillna(dcc['BILL_AMT1'].mean(),inplace= True)
dcc['BILL_AMT5'].fillna(dcc['BILL_AMT1'].mean(),inplace= True)
dcc['BILL_AMT6'].fillna(dcc['BILL_AMT1'].mean(),inplace= True)
```

Let's check now if any column has null values

In [22]:

```
dcc.isna().sum()
```

Out[22]:

```
LIMIT_BAL          0
SEX                0
EDUCATION          0
MARRIAGE           0
AGE                0
PAY_0              0
PAY_2              0
PAY_3              0
PAY_4              0
PAY_5              0
PAY_6              0
BILL_AMT1          0
BILL_AMT2          0
BILL_AMT3          0
BILL_AMT4          0
BILL_AMT5          0
BILL_AMT6          0
PAY_AMT1           0
PAY_AMT2           0
PAY_AMT3           0
PAY_AMT4           0
PAY_AMT5           0
PAY_AMT6           0
Default_Payment    0
dtype: int64
```

Exploratory Data Analaysis Using Data Visualization

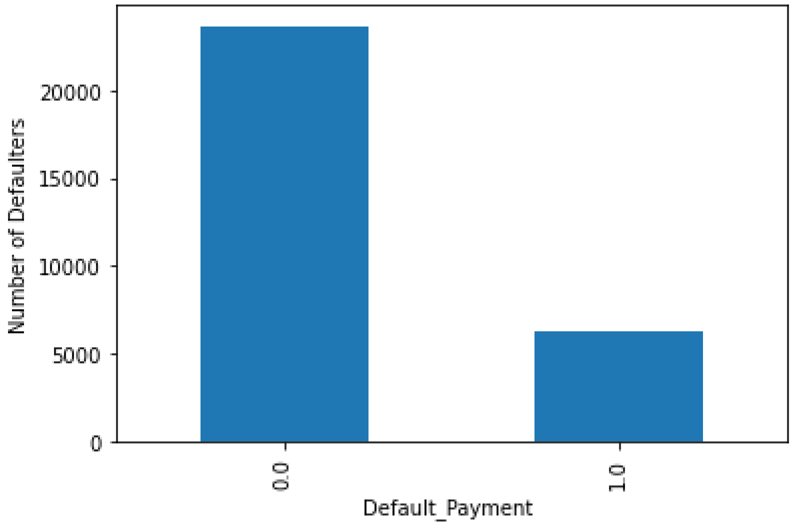
a) Distribution of target classes

In [23]:

```
dcc['Default_Payment'].value_counts().plot(kind='bar')
plt.xlabel('Default_Payment ')
plt.ylabel('Number of Defaulters')
```

Out[23]:

```
Text(0, 0.5, 'Number of Defaulters')
```



We can cleary see that Distribution of target classes is highly imbalanced and most people pay credit cards bills on time

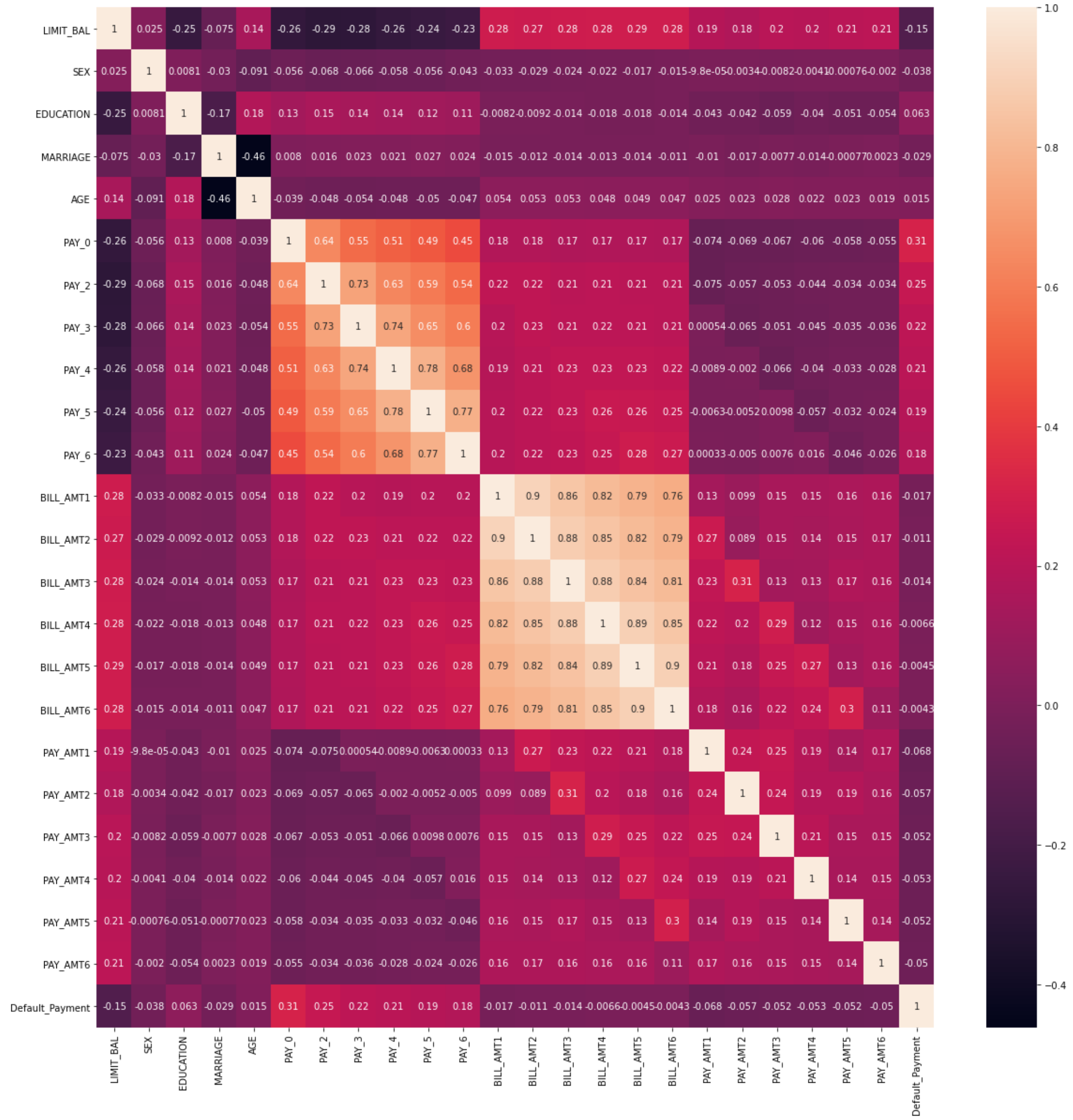
b) Co-relation of feature-set

In [24]:

```
plt.figure(figsize=(20,20))
sns.heatmap(dcc.corr(), annot=True)
```

Out[24]:

<AxesSubplot:>



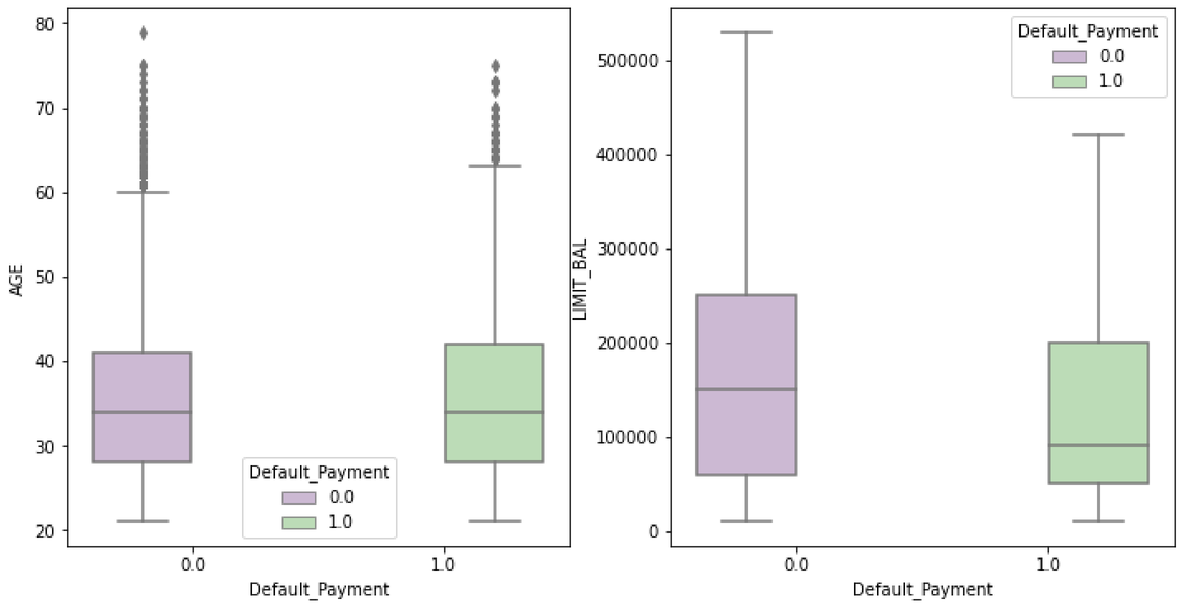
1)PAY_0, PAY_2, PAY_3, PAY_4, PAY_5, PAY_6 which represent payment status from April 2005 to September 2005 are highly co-related to each other indicating late payment in one month could lead to late payment in subsequent months as well

2) BILL_AMT1 ,BILL_AMT2, BILL_AMT3, BILL_AMT4, BILL_AMT5, BILL_AMT6 are the Amount of bill statement from April to September are again strongly co-related

c) Default Payment by AGE and Limit Balance

In [25]:

```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12,6))
s = sns.boxplot(ax = ax1, x="Default_Payment", y="AGE", hue="Default_Payment",data=dcc_
s = sns.boxplot(ax = ax2, x="Default_Payment", y="LIMIT_BAL", hue="Default_Payment",dat
plt.show();
```



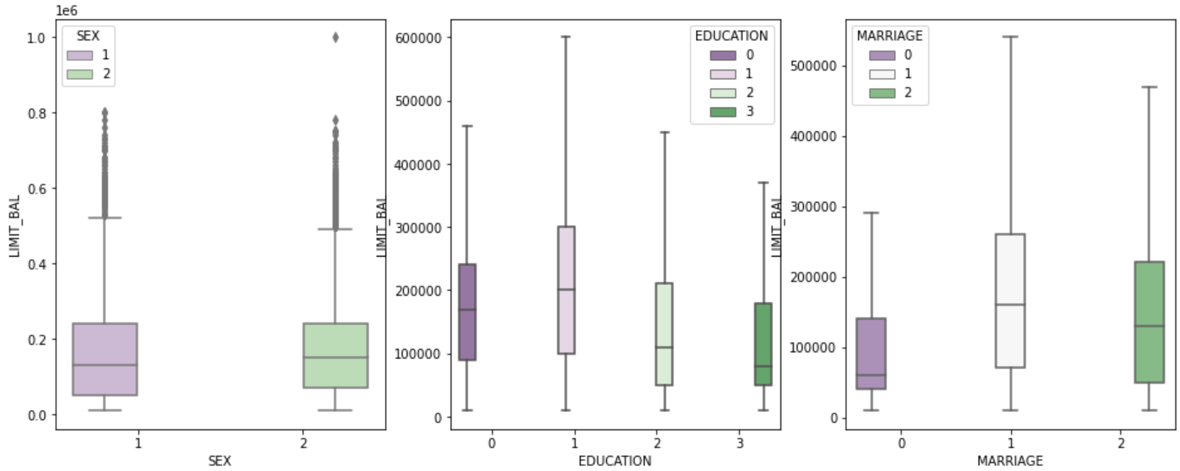
1) The dataset mostly contains people in their late-20s to late-40s who are both defaulters and non-defaulters

2) Majority of Defaulters have Limit Balance of less the 200000

d) Credit Limit by SEX

In [26]:

```
fig, (ax1, ax2, ax3) = plt.subplots(ncols=3, figsize=(16,6))
s = sns.boxplot(ax = ax1, x="SEX", y="LIMIT_BAL", hue="SEX",data=dcc, palette="PRGn",sh
s = sns.boxplot(ax = ax2, x="EDUCATION", y="LIMIT_BAL", hue="EDUCATION",data=dcc, palet
s = sns.boxplot(ax = ax3, x="MARRIAGE", y="LIMIT_BAL", hue="MARRIAGE",data=dcc, palette
plt.show()
```



1) The dataset contains almost similar distribution LIMIT_BAL for both male

2) The median LIMIT_BAL of the people who have graduate school degree in highest.

3) People who are married are observed to have greater median of LIMIT_BAL than single and others.

There are no recommended changes

Data Prepartion for Analysis and Classification

Train Test Split

In [27]:

```
X = dcc.drop('Default_Payment',axis =1)
y = dcc['Default_Payment']
X_train_org, X_test_org, y_train, y_test = train_test_split(X, y, random_state = 0)
```

In [28]:

```
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train_org)
X_test = scaler.transform(X_test_org)
```

CLASSIFICATION TASKS

For all classification tasks we will have the follwing approach

1. Find the best Hyperparaemeters for the base class using Grid Search
2. Find the best Hyperparaemeters for the voting/bagging/booster class using Grid Search
3. Combine the best parameters to create a new model
4. Train and Test the new model on the dataset and find the train score, test score, accuracy, f1 score.

1) Apply two voting classifiers - one with hard voting and one with soft voting.

Since each model participating in soft voting should have an ability to predict probablity as a part of model training, we are going to use Logistic Regression and Support Vector Machine as two voting classifier for this task.

HARD VOTING : Class prediction with the largest sum of votes from base models

In [29]:

```
param_grid_logit = { 'max_iter' : range(1,120), 'penalty' : ['l1','l2'],'C' : [0.1, 1,
grid_search_logit = GridSearchCV(estimator=LogisticRegression(random_state = 0), param_
                                 cv = 5, verbose = 1, n_jobs = -1, return_train_score =
grid_search_logit.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 1190 candidates, totalling 5950 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    3.8s
[Parallel(n_jobs=-1)]: Done 640 tasks      | elapsed:   12.4s
[Parallel(n_jobs=-1)]: Done 890 tasks      | elapsed:   17.9s
[Parallel(n_jobs=-1)]: Done 1280 tasks      | elapsed:   24.9s
[Parallel(n_jobs=-1)]: Done 2146 tasks      | elapsed:   44.4s
[Parallel(n_jobs=-1)]: Done 3152 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 4258 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done 5464 tasks      | elapsed:  2.1min
[Parallel(n_jobs=-1)]: Done 5950 out of 5950 | elapsed:  2.5min finished
```

Out[29]:

```
GridSearchCV(cv=5, estimator=LogisticRegression(random_state=0), n_jobs=-1,
             param_grid={'C': [0.1, 1, 10, 100, 1000],
                         'max_iter': range(1, 120), 'penalty': ['l1', 'l
2']},
             return_train_score=True, verbose=1)
```

In [30]:

```
print("Best parameters Logistic Regression: {}".format(grid_search_logit.best_params_))
```

```
Best parameters Logistic Regression: {'C': 1000, 'max_iter': 90, 'penalty':
'l2'}
```

In [31]:

```
logistic_hvoting = LogisticRegression(C=1000, random_state=0, max_iter=90, penalty='l2
logistic_hvoting.fit(X_train, y_train)

svm_hvoting = SVC(C = 10, probability=True, random_state=0)
svm_hvoting.fit(X_train, y_train)

h_voting_classifier = VotingClassifier(estimators=[('lr', logistic_hvoting), ('svc', sv
h_voting_classifier.fit(X_train, y_train)
```

Out[31]:

```
VotingClassifier(estimators=[('lr',
                              LogisticRegression(C=1000, max_iter=90,
                                                 random_state=0)),
                             ('svc',
                              SVC(C=10, probability=True, random_state=0))])
```

In [32]:

```
for hvoting in (logistic_hvoting, svm_hvoting, h_voting_classifier):
    hvoting.fit(X_train, y_train)
    y_pred = hvoting.predict(X_test)
    print(hvoting.__class__.__name__, accuracy_score(y_test, y_pred).round(3))
```

```
LogisticRegression 0.817
SVC 0.823
VotingClassifier 0.815
```

SOFT VOTING : Class prediction with the average probability from models.

In [33]:

```
logistic_svoting = LogisticRegression( random_state=0, C=1000, max_iter=90, penalty='l2
logistic_svoting.fit(X_train, y_train)

svm_svoting = SVC(C = 10, probability=True, random_state=0)
svm_svoting.fit(X_train, y_train)

s_voting_classifier = VotingClassifier(estimators=[('lr', logistic_svoting), ('svc', sv
s_voting_classifier.fit(X_train, y_train)

```

Out[33]:

```
VotingClassifier(estimators=[('lr',
                              LogisticRegression(C=1000, max_iter=90,
                                                 random_state=0)),
                             ('svc',
                              SVC(C=10, probability=True, random_state=0))],
                 voting='soft')
```

In [34]:

```
for svoting in (logistic_svoting, svm_svoting, s_voting_classifier):
    svoting.fit(X_train, y_train)
    y_pred = svoting.predict(X_test)
    print(svoting.__class__.__name__, accuracy_score(y_test, y_pred).round(3))
```

```
LogisticRegression 0.817
SVC 0.823
VotingClassifier 0.82
```

2) Apply any two models with bagging and any two models with pasting.

Bagging : Bootstrap = 'True'

This ensemble machine learning algorithm combines the result of multiple decision tress derived from running the model on the samples in the bags (collection of random samples) . The sample used for each decision tree is randomly selected and everytime a new bag is created all the sample in training dataset are available for the bagging process, allowing it to be selected again and perhaps multiple times in the new bags

a) Decision Tree Bagging Classifier

In [35]:

```
param_grid_dtree = {'max_depth': range(1,15)}
dtree_grid_search = GridSearchCV(DecisionTreeClassifier(random_state=0), cv=7,
                                 param_grid = param_grid_dtree, return_train_score=True
dtree_grid_search.fit(X_train, y_train)
```

Out[35]:

```
GridSearchCV(cv=7, estimator=DecisionTreeClassifier(random_state=0),
             param_grid={'max_depth': range(1, 15)}, return_train_score=Tru
e)
```

In [36]:

```
print("Best parameters : Decision Tree grid search : {}".format(dtree_grid_search.best_
```

```
Best parameters : Decision Tree grid search : {'max_depth': 4}
```

In [37]:

```
param_grid_bag = {'n_estimators':[200, 300, 400, 500], 'max_samples':[0.1, 0.2, 0.3, 0.

bag_dtree_classifier = BaggingClassifier(DecisionTreeClassifier(max_depth = 4, random_s
bag_grid_search = GridSearchCV(bag_dtree_classifier, param_grid = param_grid_bag, cv =
bag_grid_search.fit(X_train, y_train)
```

Out[37]:

```
GridSearchCV(cv=5,
             estimator=BaggingClassifier(base_estimator=DecisionTreeClassifi
er(max_depth=4,

random_state=0)),
             n_jobs=-1,
             param_grid={'max_samples': [0.1, 0.2, 0.3, 0.4, 0.5],
                         'n_estimators': [200, 300, 400, 500]})
```

In [38]:

```
print("Best parameters : Decision Tree Bagging : {}".format(bag_grid_search.best_params
```

```
Best parameters : Decision Tree Bagging : {'max_samples': 0.5, 'n_estimator
s': 200}
```

Combining the best parameters of base model and bagging to create a new model.

.

In [39]:

```
best_dtree = DecisionTreeClassifier(max_depth = 4, random_state=0)
best_bagging = BaggingClassifier(best_dtree, n_estimators=200, max_samples=0.5, oob_sco
                                 bootstrap=True, n_jobs=-1, random_state=0)
best_bagging.fit(X_train, y_train)
y_pred = best_bagging.predict(X_test)
```

In [40]:

```
print('Train score: %.3f'%best_bagging.score(X_train, y_train))
print('Test score: %.3f'%best_bagging.score(X_test, y_test))
print('Out-of-bag score: %.3f'%best_bagging.oob_score_)
```

```
Train score: 0.826
Test score: 0.830
Out-of-bag score: 0.823
```

In [41]:

```
print(classification_report(y_test, best_bagging.predict(X_test), target_names=["Defaul
```

```
              precision    recall  f1-score   support

     Default       0.85      0.95      0.90      5950
  No Default       0.67      0.35      0.46      1550

    accuracy                           0.83      7500
   macro avg       0.76      0.65      0.68      7500
weighted avg       0.81      0.83      0.81      7500
```

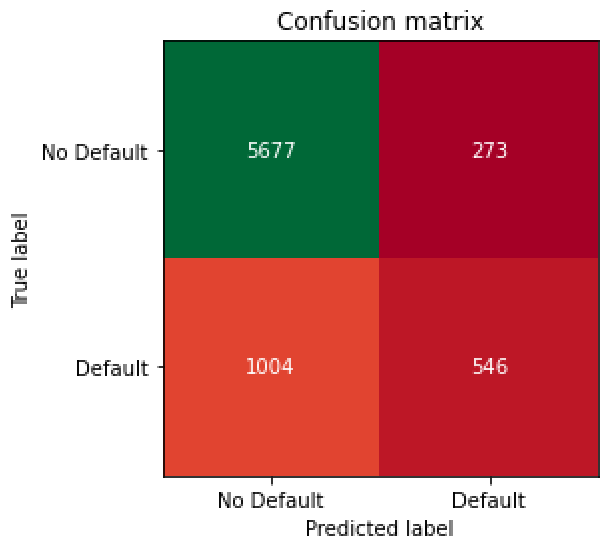
In [42]:

```
print(confusion_matrix(y_pred = y_pred, y_true = y_test))
```

```
[[5677  273]
 [1004  546]]
```

In [43]:

```
import mglearn
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```



In [44]:

```
print('Precision score : {:.3f}'.format(precision_score(y_test, best_bagging.predict(X_
```

```
Precision score : 0.667
```

In [45]:

```
print('Recall score : {:.3f} '.format(recall_score(y_test, best_bagging.predict(X_test)
```

```
Recall score : 0.352
```

In [46]:

```
print('f1 score : {:.3f} '.format(f1_score(y_test, best_bagging.predict(X_test))))
```

```
f1 score : 0.461
```

In [47]:

```
print('Accuracy : {:.3f} '.format(accuracy_score(y_test, best_bagging.predict(X_test)))
```

```
Accuracy : 0.830
```

b) Random Forest Bagging Classifier

In [48]:

```
from sklearn.ensemble import RandomForestClassifier
param_grid_rf = {'max_depth': range(1,10),'n_estimators':[100, 300, 500],
             'max_features':[0.3, 0.5],'max_samples':[0.1, 0.5]}
rf_grid_search = GridSearchCV(RandomForestClassifier(random_state = 0), param_grid = pa
rf_grid_search.fit(X_train, y_train)
```

Out[48]:

```
GridSearchCV(cv=5, estimator=RandomForestClassifier(random_state=0),
             param_grid={'max_depth': range(1, 10), 'max_features': [0.3, 0.
5],
                         'max_samples': [0.1, 0.5],
                         'n_estimators': [100, 300, 500]})
```

In [49]:

```
print("Best parameters : Random Forest Bagging grid search :{}".format(rf_grid_search.b
```

```
Best parameters : Random Forest Bagging grid search :{'max_depth': 5, 'max_f
eatures': 0.5, 'max_samples': 0.5, 'n_estimators': 500}
```

Combining the best paratmeters of base model and bagging to create a new model

In [50]:

```
best_rf = RandomForestClassifier(bootstrap = True, max_depth = 5, max_features = 0.5,
                                 max_samples = 0.5, n_estimators = 500, random_state =
best_rf.fit(X_train, y_train)
y_pred = best_rf.predict(X_test)
```

In [51]:

```
print('Test score: %.3f'%best_rf.score(X_test, y_test))
print('Train score: %.3f'%best_rf.score(X_train, y_train))
```

```
Test score: 0.830
Train score: 0.827
```

In [52]:

```
print(classification_report(y_test, y_pred=best_rf.predict(X_test), target_names=["Defa
```

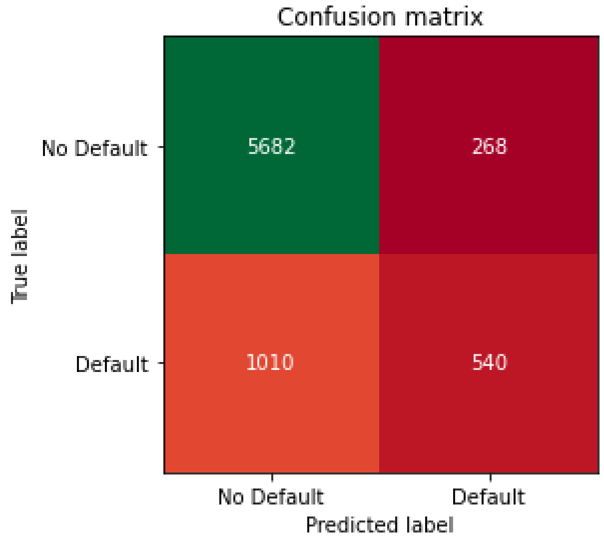
```
              precision    recall  f1-score   support

     Default       0.85      0.95      0.90      5950
  No Default       0.67      0.35      0.46      1550

    accuracy                           0.83      7500
   macro avg       0.76      0.65      0.68      7500
weighted avg       0.81      0.83      0.81      7500
```

In [53]:

```
import mglearn
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = best_rf.predict(X_test), y_true = y_test), xlabel = 'Pred
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```



In [54]:

```
print('Precision score : {:.3f}'.format(precision_score(y_test, best_rf.predict(X_test)
```

```
Precision score : 0.668
```

In [55]:

```
print('Recall score : {:.3f} '.format(recall_score(y_test, best_rf.predict(X_test))))
```

```
Recall score : 0.348
```

In [56]:

```
print('f1 score : {:.3f} '.format(f1_score(y_test, best_rf.predict(X_test))))
```

```
f1 score : 0.458
```

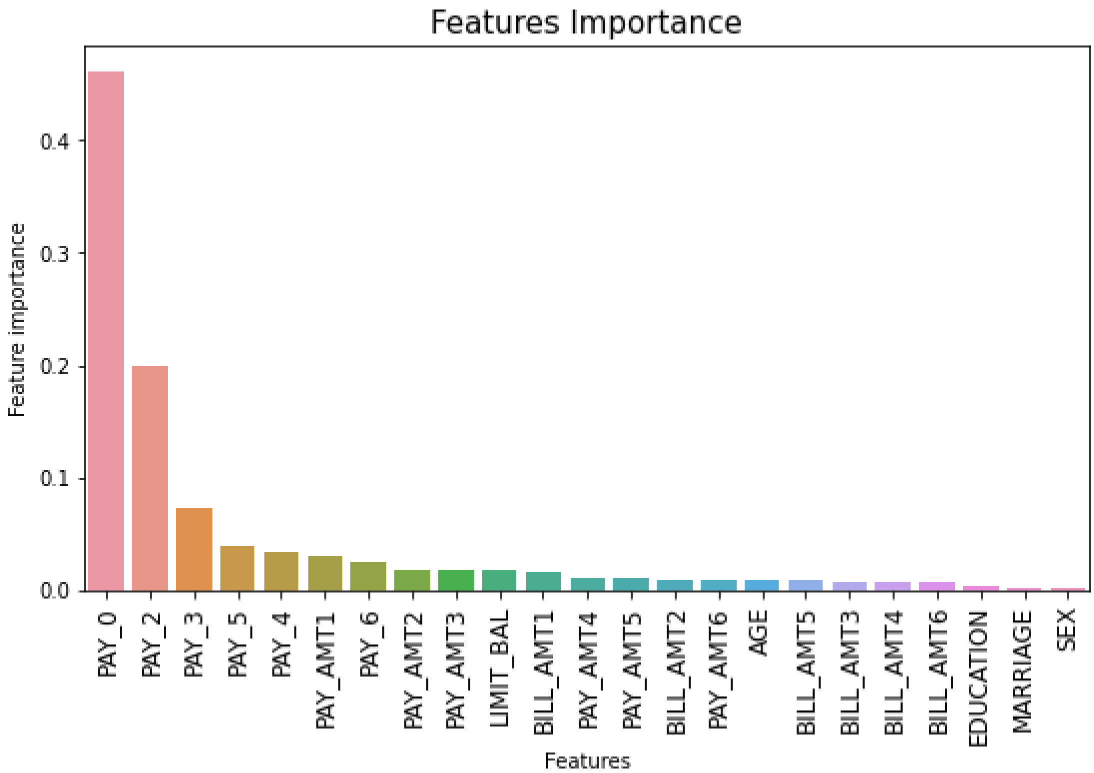
In [57]:

```
print('Accuracy : {:.3f} '.format(accuracy_score(y_test, best_rf.predict(X_test))))
```

```
Accuracy : 0.830
```

In [58]:

```
figt = pd.DataFrame({'Features': X.columns, 'Feature importance': best_rf.feature_impor
figt = figt.sort_values(by='Feature importance',ascending=False)
plt.figure(figsize = (9,5))
plt.title('Features Importance',fontsize=15)
s = sns.barplot(x='Features',y='Feature importance',data=figt)
s.set_xticklabels(s.get_xticklabels(), fontsize=12, rotation=90 )
plt.show()
```



Pasting : Bootstrap = 'False'

This ensemble machine learning algorithm combines the result of multiple decision tress derived the bags (collection of random samples) . The sample used for each decision tree is randomly selected and are unavailable for subsequent bagging cycles. In other words the collection of sample in each bag is unique.

a) Decision Tree Pasting Classifier

We have already found best parameters for the base models while performing Bagging, Hence for the purpose of pasting we will simply combine the best parameters that are already found above with the only difference, Bootstrap ='False'

In [59]:

```
print("Best parameters : Decision Tree grid search : {}".format(dtree_grid_search.best_
```

```
Best parameters : Decision Tree grid search : {'max_depth': 4}
```

In [60]:

```
print("Best parameters : Decision Tree Pasting : {}".format(bag_grid_search.best_params
```

```
Best parameters : Decision Tree Pasting : {'max_samples': 0.5, 'n_estimator
s': 200}
```

Combining the best parameters of base model and pasting to create a new model.

.

In [61]:

```
best_dtree = DecisionTreeClassifier(max_depth = 4, random_state=0)
best_pasting = BaggingClassifier(best_dtree, bootstrap=False, n_estimators=200, max_sar

best_pasting.fit(X_train, y_train)
y_pred = best_pasting.predict(X_test)
```

In [62]:

```
print('Train score: %.3f'%best_pasting.score(X_train, y_train))
print('Test score: %.3f'%best_pasting.score(X_test, y_test))
```

```
Train score: 0.826
Test score: 0.829
```

In [63]:

```
print(classification_report(y_test, best_pasting.predict(X_test), target_names=["Defau
```

```
              precision    recall  f1-score   support

     Default       0.85      0.95      0.90      5950
  No Default       0.66      0.35      0.46      1550

    accuracy                           0.83      7500
   macro avg       0.76      0.65      0.68      7500
weighted avg       0.81      0.83      0.81      7500
```

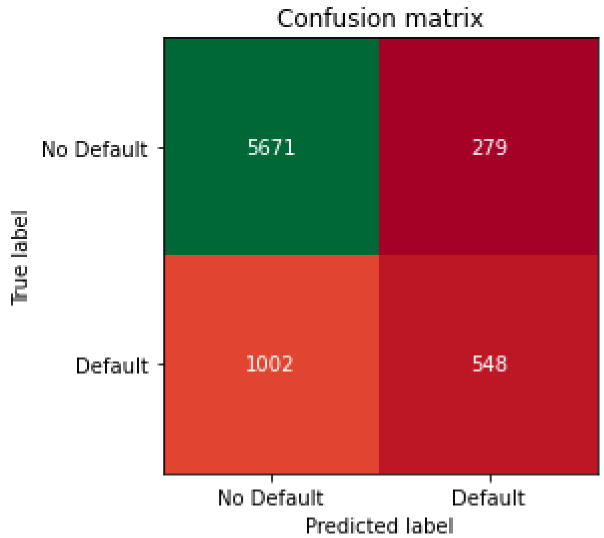
In [64]:

```
print(confusion_matrix(y_pred = y_pred, y_true = y_test))
```

```
[[5671  279]
 [1002  548]]
```

In [65]:

```
import mglearn
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```



In [66]:

```
print('Precision score : {:.3f}'.format(precision_score(y_test, best_pasting.predict(X_
```

```
Precision score : 0.663
```

In [67]:

```
print('Recall score : {:.3f} '.format(recall_score(y_test, best_pasting.predict(X_test)
```

```
Recall score : 0.354
```

In [68]:

```
print('f1 score : {:.3f} '.format(f1_score(y_test, best_pasting.predict(X_test))))
```

```
f1 score : 0.461
```

In [69]:

```
print('Accuracy : {:.3f} '.format(accuracy_score(y_test, best_pasting.predict(X_test)))
```

```
Accuracy : 0.829
```

b) Random Forest Pasting Classifier

In [70]:

```
print("Best parameters : Random Forest grid search :{}".format(rf_grid_search.best_para
```

```
Best parameters : Random Forest grid search :{'max_depth': 5, 'max_feature
s': 0.5, 'max_samples': 0.5, 'n_estimators': 500}
```

Combining the best parameters of base model and pasting to create a new model

In [71]:

```
best_pasting_rf = RandomForestClassifier(bootstrap = False, max_depth = 5, max_features
best_pasting_rf.fit(X_train, y_train)
y_pred = best_pasting_rf.predict(X_test)
```

In [72]:

```
print('Test score: %.3f'%best_pasting_rf.score(X_test, y_test))
print('Train score: %.3f'%best_pasting_rf.score(X_train, y_train))
```

```
Test score: 0.830
Train score: 0.827
```

In [73]:

```
print(classification_report(y_test, y_pred=best_pasting_rf.predict(X_test), target_name
```

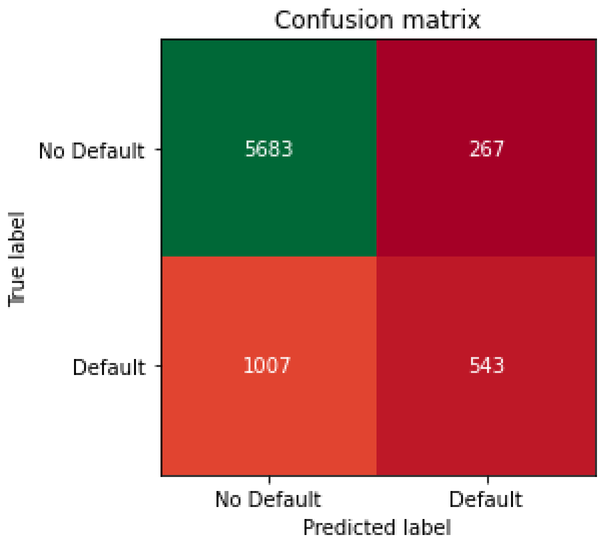
```
              precision    recall  f1-score   support

     Default       0.85      0.96      0.90      5950
  No Default       0.67      0.35      0.46      1550

    accuracy                           0.83      7500
   macro avg       0.76      0.65      0.68      7500
weighted avg       0.81      0.83      0.81      7500
```

In [74]:

```
import mglearn
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = best_pasting_rf.predict(X_test), y_true = y_test), xlabel
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```



In [75]:

```
print('Precision score : {:.3f}'.format(precision_score(y_test, best_pasting_rf.predict
```

Precision score : 0.670

In [76]:

```
print('Recall score : {:.3f} '.format(recall_score(y_test, best_pasting_rf.predict(X_te
```

Recall score : 0.350

In [77]:

```
print('f1 score : {:.3f} '.format(f1_score(y_test, best_pasting_rf.predict(X_test))))
```

f1 score : 0.460

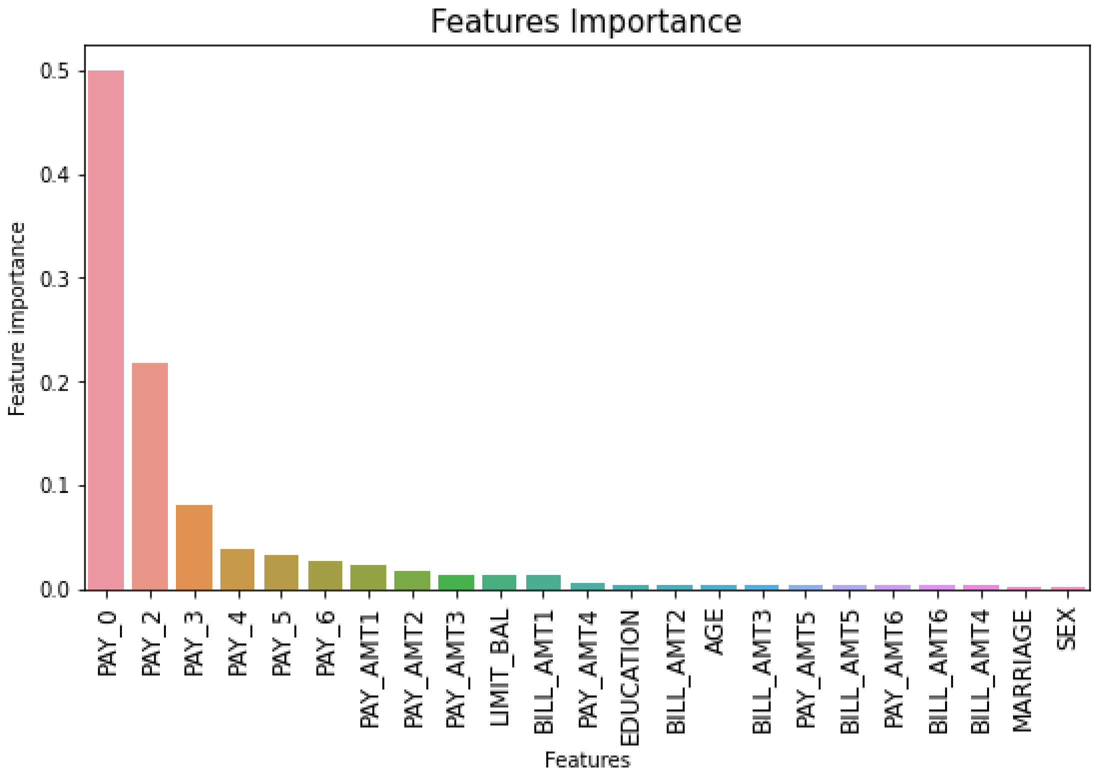
In [78]:

```
print('Accuracy : {:.3f} '.format(accuracy_score(y_test, best_pasting_rf.predict(X_test
```

Accuracy : 0.830

In [79]:

```
figt = pd.DataFrame({'Features': X.columns, 'Feature importance': best_pasting_rf.featu
figt = figt.sort_values(by='Feature importance',ascending=False)
plt.figure(figsize = (9,5))
plt.title('Features Importance',fontsize=15)
s = sns.barplot(x='Features',y='Feature importance',data=figt)
s.set_xticklabels(s.get_xticklabels(), fontsize=12, rotation=90 )
plt.show()
```



3. ADABOOSTING

The goal in AdaBoosting is to change the cost function in every iteration by giving importance to those function which haven't been predicted correctly

a) Logistic Regression AdaBoost

In [80]:

```
param_grid_logit = { 'max_iter' : range(1,150), 'penalty' : ['l1','l2'], 'C' : [0.01, 0
logit_class_CV = GridSearchCV(LogisticRegression(random_state=0), param_grid = param_gr
logit_class_CV.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 1788 candidates, totalling 8940 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    0.8s
[Parallel(n_jobs=-1)]: Done 1032 tasks      | elapsed:   12.0s
[Parallel(n_jobs=-1)]: Done 2178 tasks      | elapsed:   26.9s
[Parallel(n_jobs=-1)]: Done 2528 tasks      | elapsed:   34.6s
[Parallel(n_jobs=-1)]: Done 2978 tasks      | elapsed:   44.4s
[Parallel(n_jobs=-1)]: Done 3984 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 5090 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done 5840 tasks      | elapsed:  2.1min
[Parallel(n_jobs=-1)]: Done 7146 tasks      | elapsed:  2.7min
[Parallel(n_jobs=-1)]: Done 8552 tasks      | elapsed:  3.4min
[Parallel(n_jobs=-1)]: Done 8940 out of 8940 | elapsed:  3.8min finished
```

Out[80]:

```
GridSearchCV(cv=5, estimator=LogisticRegression(random_state=0), n_jobs=-1,
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000],
                         'max_iter': range(1, 150), 'penalty': ['l1', 'l
2']},
             return_train_score=True, verbose=1)
```

In [81]:

```
print("Best parameters: Logistic Regression {}".format(logit_class_CV.best_params_))
```

```
Best parameters: Logistic Regression {'C': 1000, 'max_iter': 90, 'penalty':
'l2'}
```

In [82]:

```
from sklearn.ensemble import AdaBoostClassifier
param_grid = {'n_estimators': [200,250,500,600], 'learning_rate': [.02, .05, 0.5, 1]}

log_ada_gs = GridSearchCV(AdaBoostClassifier(LogisticRegression(C=1000, max_iter=90, pe
                                             random_state = 0), param_grid, cv=7, retur
log_ada_gs.fit(X_train, y_train)
```

```
Fitting 7 folds for each of 16 candidates, totalling 112 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent work
ers.
[Parallel(n_jobs=1)]: Done 112 out of 112 | elapsed: 13.5min finished
```

Out[82]:

```
GridSearchCV(cv=7,
             estimator=AdaBoostClassifier(base_estimator=LogisticRegression
(C=1000,

max_iter=90),
                                          random_state=0),
             param_grid={'learning_rate': [0.02, 0.05, 0.5, 1],
                         'n_estimators': [200, 250, 500, 600]},
             return_train_score=True, verbose=True)
```

In [83]:

```
print("Best parameters: Logistic Regression AdaBoost {}".format(log_ada_gs.best_params_
```

```
Best parameters: Logistic Regression AdaBoost {'learning_rate': 1, 'n_estima
tors': 200}
```

Combining Best Parameters for AdaBoosting

In [84]:

```
best_logit_ada = AdaBoostClassifier(LogisticRegression(random_state=0), random_state=0,
                                    algorithm="SAMME.R", learning_rate=1)
best_logit_ada.fit(X_train, y_train)
y_pred = best_logit_ada.predict(X_test)
```

In [85]:

```
print("Accuracy :",accuracy_score(y_test, y_pred).round(3))
```

```
Accuracy : 0.801
```

In [86]:

```
print(classification_report(y_test, y_pred= best_logit_ada.predict(X_test), target_name
```

```
              precision    recall  f1-score   support

     Default       0.80      0.99      0.89      5950
  No Default       0.72      0.06      0.12      1550

    accuracy                           0.80      7500
   macro avg       0.76      0.53      0.50      7500
weighted avg       0.79      0.80      0.73      7500
```

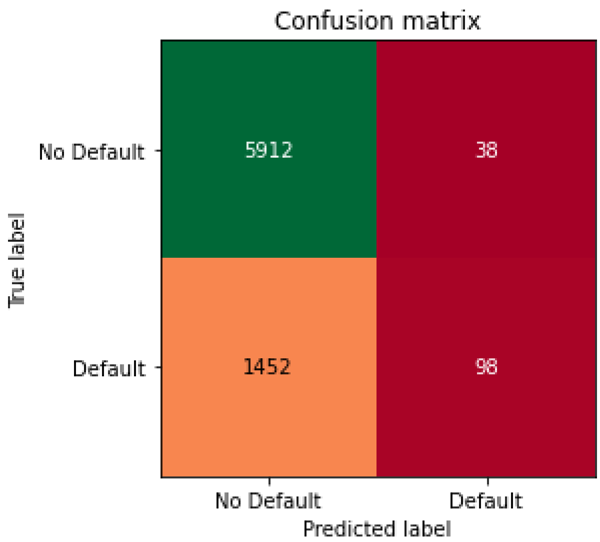
In [87]:

```
print(confusion_matrix(y_pred = y_pred, y_true = y_test))
```

```
[[5912   38]
 [1452   98]]
```

In [88]:

```
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```



b) Decision Tree AdaBoost

In [89]:

```
param_grid_dtree = {'max_depth': range(1,10),'criterion':['gini','entropy'],'min_sample
GS_results_dtrees = GridSearchCV(DecisionTreeClassifier(random_state=0), cv = 7, param_
GS_results_dtrees.fit(X_train, y_train)
```

```
Fitting 7 folds for each of 864 candidates, totalling 6048 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    4.2s
[Parallel(n_jobs=-1)]: Done 312 tasks      | elapsed:    7.1s
[Parallel(n_jobs=-1)]: Done 812 tasks      | elapsed:   16.1s
[Parallel(n_jobs=-1)]: Done 1512 tasks      | elapsed:   35.2s
[Parallel(n_jobs=-1)]: Done 2412 tasks      | elapsed:  1.2min
[Parallel(n_jobs=-1)]: Done 3512 tasks      | elapsed:  1.8min
[Parallel(n_jobs=-1)]: Done 4812 tasks      | elapsed:  2.5min
[Parallel(n_jobs=-1)]: Done 5800 tasks      | elapsed:  3.5min
[Parallel(n_jobs=-1)]: Done 6048 out of 6048 | elapsed:  3.8min finished
```

Out[89]:

```
GridSearchCV(cv=7, estimator=DecisionTreeClassifier(random_state=0), n_jobs=
-1,
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': range(1, 10),
                         'min_samples_leaf': range(2, 50)},
             verbose=True)
```

In [90]:

```
print("Best parameters: Decision Tree Adaboost {}".format(GS_results_dtrees.best_params
```

```
Best parameters: Decision Tree Adaboost {'criterion': 'entropy', 'max_dept
h': 4, 'min_samples_leaf': 48}
```

In [91]:

```
param_grid = {'n_estimators': [100,150,250,400], 'learning_rate': [.02, .05, 0.5, 1]}
dtree_ada_gs = GridSearchCV(AdaBoostClassifier(DecisionTreeClassifier(criterion ='entro
                                               algorithm="SAMME.R",random_state = 0),
                            param_grid, cv=7, return_train_score=True, verbose=True)
dtree_ada_gs.fit(X_train, y_train)
```

```
Fitting 7 folds for each of 16 candidates, totalling 112 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent work
ers.
[Parallel(n_jobs=1)]: Done 112 out of 112 | elapsed: 46.7min finished
```

Out[91]:

```
GridSearchCV(cv=7,
             estimator=AdaBoostClassifier(base_estimator=DecisionTreeClassif
ier(criterion='entropy',

max_depth=4,

min_samples_leaf=48),
                                          random_state=0),
             param_grid={'learning_rate': [0.02, 0.05, 0.5, 1],
                         'n_estimators': [100, 150, 250, 400]},
             return_train_score=True, verbose=True)
```

In [92]:

```
print("Best parameters: Decision Tree AdaBoost {}".format(dtree_ada_gs.best_params_))
```

```
Best parameters: Decision Tree AdaBoost {'learning_rate': 0.02, 'n_estimator
s': 250}
```

Combining Best Parameters for AdaBoosting

In [93]:

```
best_dtree_ada = AdaBoostClassifier(DecisionTreeClassifier(criterion ='entropy',max_dep
                                    random_state=0, n_estimators=250, algorithm="SAMME.
best_dtree_ada.fit(X_train, y_train)
y_pred = best_dtree_ada.predict(X_test)
```

In [94]:

```
print("Accuracy :",accuracy_score(y_test, y_pred).round(3))
```

```
Accuracy : 0.83
```

In [95]:

```
print(classification_report(y_test, y_pred= best_dtree_ada.predict(X_test), target_name
```

```
              precision    recall  f1-score   support

     Default       0.85      0.96      0.90      5950
  No Default       0.67      0.35      0.46      1550

    accuracy                           0.83      7500
   macro avg       0.76      0.65      0.68      7500
weighted avg       0.81      0.83      0.81      7500
```

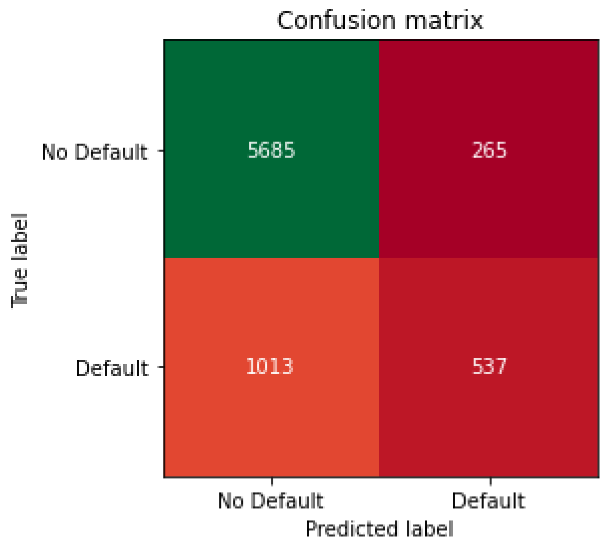
In [96]:

```
print(confusion_matrix(y_pred = y_pred, y_true = y_test))
```

```
[[5685  265]
 [1013  537]]
```

In [97]:

```
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```



4. Gradient Boosting

In [98]:

```
from sklearn.ensemble import GradientBoostingClassifier
param_grid = {'n_estimators': [25,50,100,200],'learning_rate': [0.05,0.1,0.2,0.5,1], 'r
gb_grid_search = GridSearchCV(GradientBoostingClassifier(random_state = 0), param_grid,
                              cv=10, return_train_score=True, n_jobs=-1, verbose=True)
gb_grid_search.fit(X_train, y_train)
```

```
Fitting 10 folds for each of 100 candidates, totalling 1000 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   29.0s
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed:  5.3min
[Parallel(n_jobs=-1)]: Done 434 tasks      | elapsed: 13.5min
[Parallel(n_jobs=-1)]: Done 784 tasks      | elapsed: 25.4min
[Parallel(n_jobs=-1)]: Done 1000 out of 1000 | elapsed: 33.3min finished
```

Out[98]:

```
GridSearchCV(cv=10, estimator=GradientBoostingClassifier(random_state=0),
             n_jobs=-1,
             param_grid={'learning_rate': [0.05, 0.1, 0.2, 0.5, 1],
                         'max_depth': array([1, 2, 3, 4, 5]),
                         'n_estimators': [25, 50, 100, 200]},
             return_train_score=True, verbose=True)
```

In [99]:

```
print("Best parameters Gradient Boosting: {}".format(gb_grid_search.best_params_))
```

```
Best parameters Gradient Boosting: {'learning_rate': 0.1, 'max_depth': 4, 'n
_estimators': 50}
```

In [100]:

```
best_gbrt = GradientBoostingClassifier(random_state=0, n_estimators=50, learning_rate=0
best_gbrt.fit(X_train, y_train)
y_pred = best_gbrt.predict(X_test)
```

In [101]:

```
print("Accuracy :",accuracy_score(y_test, y_pred).round(3))
```

```
Accuracy : 0.831
```

In [102]:

```
print(classification_report(y_test, y_pred= best_gbrt.predict(X_test), target_names=["D
```

```
              precision    recall  f1-score   support

     Default       0.85      0.95      0.90      5950
  No Default       0.67      0.36      0.47      1550

    accuracy                           0.83      7500
   macro avg       0.76      0.66      0.68      7500
weighted avg       0.81      0.83      0.81      7500
```

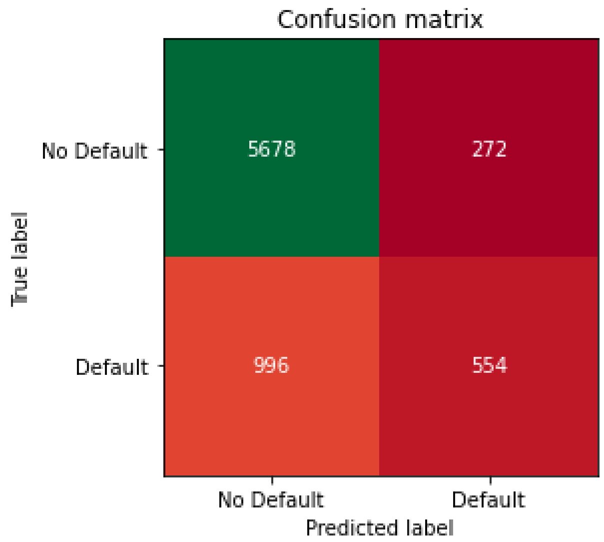
In [103]:

```
print(confusion_matrix(y_pred = y_pred, y_true = y_test))
```

```
[[5678  272]
 [ 996  554]]
```

In [104]:

```
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```



PCA

Train Test Split

In [105]:

```
X = dcc.drop('Default_Payment',axis =1)
y = dcc['Default_Payment']
X_train_org, X_test_org, y_train, y_test = train_test_split(X, y, random_state = 0)
```

In [106]:

```
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train_org)
X_test = scaler.transform(X_test_org)
```

In [107]:

```
X_train.shape
```

Out[107]:

```
(22500, 23)
```

With an Explained Variance ratio of 95% let's perform Principal Component analyisis to reduce the dimensionality

In [108]:

```
from sklearn.decomposition import PCA
pca_classification = PCA(n_components = 0.95, random_state = 0)
X_train_pca = pca_classification.fit_transform(X_train)
X_test_pca = pca_classification.transform(X_test)
```

Let's check how much of dimensionality is reduced using Explained Variance ratio of 95%.

In [109]:

```
X_train_pca.shape
```

Out[109]:

```
(22500, 8)
```

As we can see that total 15 columns were dropped as a part of dimensionality reduction

In [110]:

```
pca_classification.n_components_
```

Out[110]:

```
8
```

1. k-nearest neighbors (KNN) - PCA

Finding best parameters for the model using grid search

In [111]:

```
gs_knn_para = {'n_neighbors':range(1,20),'weights': ['uniform','distance'],
              'metric': ['euclidean','manhattan']}

gs_knn = GridSearchCV(KNeighborsClassifier(), gs_knn_para, verbose = 1, cv = 7, n_jobs
gs_knn.fit(X_train_pca, y_train)
```

```
Fitting 7 folds for each of 76 candidates, totalling 532 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    4.8s
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed:   12.5s
[Parallel(n_jobs=-1)]: Done 434 tasks      | elapsed:   27.9s
[Parallel(n_jobs=-1)]: Done 532 out of 532 | elapsed:   35.6s finished
```

Out[111]:

```
GridSearchCV(cv=7, estimator=KNeighborsClassifier(), n_jobs=-1,
             param_grid={'metric': ['euclidean', 'manhattan'],
                         'n_neighbors': range(1, 20),
                         'weights': ['uniform', 'distance']},
             verbose=1)
```

In [112]:

```
gs_knn.best_score_
```

Out[112]:

```
0.8067113758997345
```

In [113]:

```
print("KNN grid search Best Parameters {}".format(gs_knn.best_params_))
```

```
KNN grid search Best Parameters {'metric': 'manhattan', 'n_neighbors': 13,
'weights': 'uniform'}
```

Applying best parameters 'metric': 'manhattan', 'n_neighbors': 13, 'weights': 'uniform' obtained using grid search

In [114]:

```
knn_pca = KNeighborsClassifier(metric ='manhattan', n_neighbors=13, weights= 'uniform')
knn_pca.fit(X_train_pca, y_train)
y_pred = knn_pca.predict(X_test_pca)
```

In [115]:

```
print('Training score: {:.3f}'.format(knn_pca.score(X_train_pca,y_train)))
print('Testing score: {:.3f}'.format(knn_pca.score(X_test_pca,y_test)))
```

```
Training score: 0.823
Testing score: 0.808
```

In [116]:

```
print(classification_report(y_pred = y_pred, y_true = y_test))
```

```
              precision    recall  f1-score   support

         0.0       0.83      0.95      0.89      5950
         1.0       0.57      0.27      0.37      1550

    accuracy                           0.81      7500
   macro avg       0.70      0.61      0.63      7500
weighted avg       0.78      0.81      0.78      7500
```

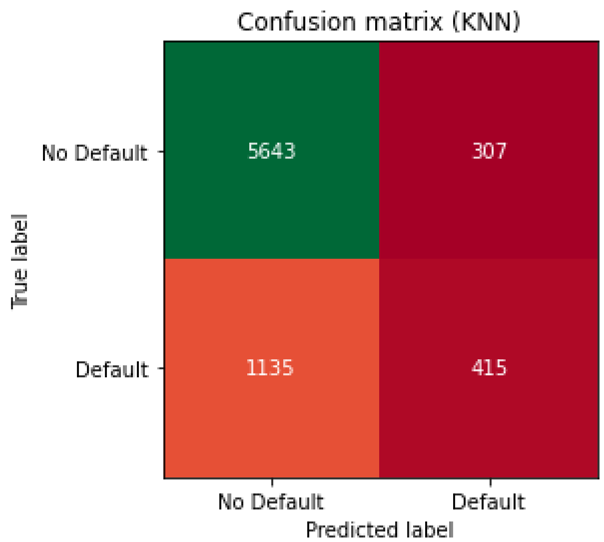
In [117]:

```
print(confusion_matrix(y_pred = y_pred, y_true = y_test))
```

```
[[5643  307]
 [1135  415]]
```

In [118]:

```
import mglearn
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix (KNN)")
plt.gca().invert_yaxis()
```



In [119]:

```
print('KNN-PCA f1-score : {:.3f} '.format(f1_score(y_test,knn_pca.predict(X_test_pca))

```

```
KNN-PCA f1-score : 0.365
```

In [120]:

```
print('KNN-PCA Accuracy : {:.3f} '.format(accuracy_score(y_test, knn_pca.predict(X_test
```

```
KNN-PCA Accuracy : 0.808
```

2. Logistic Regression - PCA

Finding best parameters for the model using grid search

In [121]:

```
gs_logit_para = { 'max_iter' : range(1,200), 'penalty' : ['l1','l2'], 'C' : [0.001, 0.0
gs_logit = GridSearchCV(LogisticRegression(), param_grid = gs_logit_para, cv = 5, verbo
                        n_jobs = -1, return_train_score = True)
gs_logit.fit(X_train_pca, y_train)
```

```
Fitting 5 folds for each of 2786 candidates, totalling 13930 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.2s
[Parallel(n_jobs=-1)]: Done 1200 tasks      | elapsed:    5.6s
[Parallel(n_jobs=-1)]: Done 3200 tasks      | elapsed:   16.7s
[Parallel(n_jobs=-1)]: Done 6000 tasks      | elapsed:   38.2s
[Parallel(n_jobs=-1)]: Done 9600 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 13874 tasks      | elapsed:  1.7min
[Parallel(n_jobs=-1)]: Done 13930 out of 13930 | elapsed:  1.7min finished
```

Out[121]:

```
GridSearchCV(cv=5, estimator=LogisticRegression(), n_jobs=-1,
             param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
                         'max_iter': range(1, 200), 'penalty': ['l1', 'l
2']},
             return_train_score=True, verbose=1)
```

In [122]:

```
gs_logit.best_score_
```

Out[122]:

```
0.8048
```

In [123]:

```
print("Logistic Regression -PCA grid search Best Parameters {}".format(gs_logit.best_pa
```

```
Logistic Regression -PCA grid search Best Parameters {'C': 100, 'max_iter':
9, 'penalty': 'l2'}
```

Applying best parameters 'C': 100, 'max_iter': 9, 'penalty': 'l2' obtained using grid search

In [124]:

```
logit_pca = LogisticRegression(C=100, max_iter=9, penalty='l2')
logit_pca.fit(X_train_pca, y_train)
y_pred = logit_pca.predict(X_test_pca)
```

In [125]:

```
print('Training score: {:.3f}'.format(logit_pca.score(X_train_pca,y_train)))
print('Testing score: {:.3f}'.format(logit_pca.score(X_test_pca,y_test)))
```

```
Training score: 0.805
Testing score: 0.809
```

In [126]:

```
print(classification_report(y_pred = y_pred, y_true = y_test))
```

```
              precision    recall  f1-score   support

         0.0       0.82      0.98      0.89      5950
         1.0       0.66      0.15      0.25      1550

    accuracy                           0.81      7500
   macro avg       0.74      0.57      0.57      7500
weighted avg       0.78      0.81      0.76      7500
```

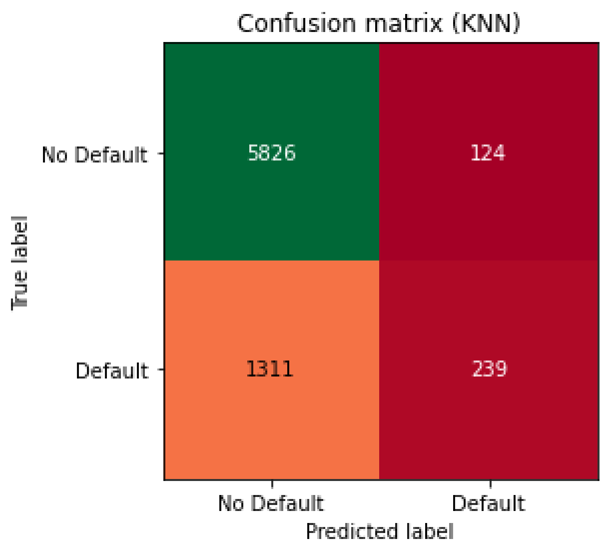
In [127]:

```
print(confusion_matrix(y_pred = y_pred, y_true = y_test))
```

```
[[5826  124]
 [1311  239]]
```

In [128]:

```
import mglearn
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix (KNN)")
plt.gca().invert_yaxis()
```



In [129]:

```
print('Logistic Regression-PCA f1-score : {:.3f} '.format(f1_score(y_test,logit_pca.pre
```

```
Logistic Regression-PCA f1-score : 0.250
```

In [130]:

```
print('Logistic Regression-PCA Accuracy : {:.3f} '.format(accuracy_score(y_test, logit_
```

```
Logistic Regression-PCA Accuracy : 0.809
```

3. Linear Support Vector Machine Classifier- PCA

Finding best parameters for the model using grid search

In [131]:

```
from sklearn.svm import LinearSVC
gs_linearsvm_para= { 'max_iter' : range(1,150),'C' : [ 0.001,0.01, 0.1, 1, 10, 100,1000
gs_linearsvm = GridSearchCV(LinearSVC(), param_grid = gs_linearsvm_para, cv = 5, verbos
                            n_jobs = -1, return_train_score = True)
gs_linearsvm.fit(X_train_pca, y_train)
```

```
Fitting 5 folds for each of 1043 candidates, totalling 5215 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  52 tasks      | elapsed:    0.7s
[Parallel(n_jobs=-1)]: Done 352 tasks      | elapsed:    5.4s
[Parallel(n_jobs=-1)]: Done 852 tasks      | elapsed:   13.0s
[Parallel(n_jobs=-1)]: Done 1552 tasks      | elapsed:   22.8s
[Parallel(n_jobs=-1)]: Done 2452 tasks      | elapsed:   42.5s
[Parallel(n_jobs=-1)]: Done 3048 tasks      | elapsed:  1.5min
[Parallel(n_jobs=-1)]: Done 3698 tasks      | elapsed:  2.4min
[Parallel(n_jobs=-1)]: Done 4448 tasks      | elapsed:  3.5min
[Parallel(n_jobs=-1)]: Done 5215 out of 5215 | elapsed:  4.6min finished
```

Out[131]:

```
GridSearchCV(cv=5, estimator=LinearSVC(), n_jobs=-1,
             param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
                         'max_iter': range(1, 150)},
             return_train_score=True, verbose=1)
```

In [132]:

```
gs_linearsvm.best_score_
```

Out[132]:

```
0.8049777777777779
```

In [133]:

```
print("Linear SVM - PCA grid search Best Parameters {}".format(gs_linearsvm.best_params
```

```
Linear SVM - PCA grid search Best Parameters {'C': 10, 'max_iter': 69}
```

.

Applying best parameters 'C': 10, 'max_iter': 69 obtained using grid search

In [134]:

```
linearsvm_pca = LinearSVC(C= 10, max_iter= 69)
linearsvm_pca.fit(X_train_pca, y_train)
y_pred = linearsvm_pca.predict(X_test_pca)
```

In [135]:

```
print('Training score: {:.3f}'.format(linearsvm_pca.score(X_train_pca,y_train)))
print('Testing score: {:.3f}'.format(linearsvm_pca.score(X_test_pca,y_test)))
```

```
Training score: 0.800
Testing score: 0.808
```

In [136]:

```
print(classification_report(y_pred = y_pred, y_true = y_test))
```

```
              precision    recall  f1-score   support

         0.0       0.81      0.98      0.89      5950
         1.0       0.69      0.13      0.22      1550

    accuracy                           0.81      7500
   macro avg       0.75      0.56      0.55      7500
weighted avg       0.79      0.81      0.75      7500
```

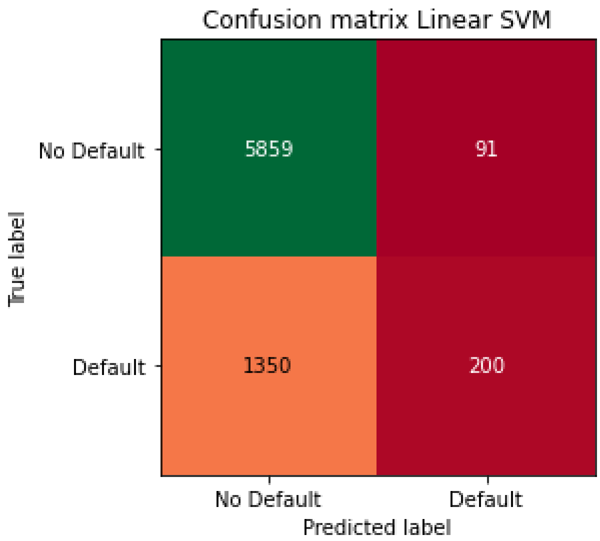
In [137]:

```
print(confusion_matrix(y_pred = y_pred, y_true = y_test))
```

```
[[5859   91]
 [1350  200]]
```

In [138]:

```
import mglearn
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix Linear SVM")
plt.gca().invert_yaxis()
```



In [139]:

```
print('Linear Support Vector Machine-PCA f1-score : {:.3f} '.format(f1_score(y_test,lin
```

Linear Support Vector Machine-PCA f1-score : 0.217

In [140]:

```
print('Linear Support Vector Machine -PCA Accuracy : {:.3f} '.format(accuracy_score(y_t
```

Linear Support Vector Machine -PCA Accuracy : 0.808

4.Kernalized Support Vector Machine (rbf, poly, and linear)- PCA

Reducing sample size to 1000 samples with subsampling.Let's map the training samples with random feature mapping to obtain training set and train linear SVMs in parallel to get a unified model on the training set.

In [141]:

```
dcc_k = dcc.sample(n = 1000, random_state= 0)
```

In [142]:

```
dcc_k.shape
```

Out[142]:

```
(1000, 24)
```

In [143]:

```
X_k = dcc_k.drop(['Default_Payment'],axis =1)
y_k = dcc_k['Default_Payment']
X_train_org_k, X_test_org_k, y_train_k, y_test_k = train_test_split(X_k, y_k, random_st
```

In [144]:

```
scaler = MinMaxScaler()
X_train_k = scaler.fit_transform(X_train_org_k)
X_test_k = scaler.transform(X_test_org_k)
```

In [145]:

```
X_train_k.shape
```

Out[145]:

```
(750, 23)
```

With an Explained Variance ratio of 95% let's perform Principal Component analyisis to reduce the dimensionality

In [146]:

```
from sklearn.decomposition import PCA
pca_classification_svm = PCA(n_components = 0.95, random_state = 0)
X_train_k_pca = pca_classification_svm.fit_transform(X_train_k)
X_test_k_pca = pca_classification_svm.transform(X_test_k)
```

Let's check how much of dimensionality is reduced using Explained Variance ratio of 95%.

In [147]:

```
X_train_k_pca.shape
```

Out[147]:

```
(750, 10)
```

As we can see that total 13 columns were dropped as a part of dimensionality reduction

In [148]:

```
pca_classification_svm.n_components_
```

Out[148]:

```
10
```

In [149]:

```
gs_kernel_para= {'gamma':[0.001, 0.01, 0.1, 1, 10, 100],'C' : [ 0.001,0.01, 0.1, 1, 10,
gs_kernel = GridSearchCV(SVC(), param_grid = gs_kernel_para, cv = 5, verbose = 1, n_jot
gs_kernel.fit(X_train_k_pca, y_train_k)
```

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

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    4.2s
[Parallel(n_jobs=-1)]: Done 180 out of 180 | elapsed:    4.7s finished
```

Out[149]:

```
GridSearchCV(cv=5, estimator=SVC(), n_jobs=-1,
             param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100],
                         'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
             return_train_score=True, verbose=1)
```

In [150]:

```
print("SVC grid search Best Parameters {}".format(gs_kernel.best_params_))
```

```
SVC grid search Best Parameters {'C': 10, 'gamma': 1}
```

'C': 10, 'gamma': 1, kernel = poly

In [151]:

```
kernel_poly = SVC(C = 10, gamma = 1, kernel = 'poly', verbose = 1)
kernel_poly.fit(X_train_k,y_train_k)
y_pred = kernel_poly.predict(X_test_k)
```

```
[LibSVM]
```

In [152]:

```
print('Training score: {:.3f}'.format(kernel_poly.score(X_train_k, y_train_k)))
print('Testing score: {:.3f}'.format(kernel_poly.score(X_test_k, y_test_k)))
```

```
Training score: 0.913
Testing score: 0.788
```

In [153]:

```
print(classification_report(y_pred,y_test_k))
```

```
              precision    recall  f1-score   support

         0.0       0.96      0.80      0.87       227
         1.0       0.25      0.65      0.36        23

    accuracy                           0.79       250
   macro avg       0.60      0.73      0.62       250
weighted avg       0.89      0.79      0.83       250
```

In [154]:

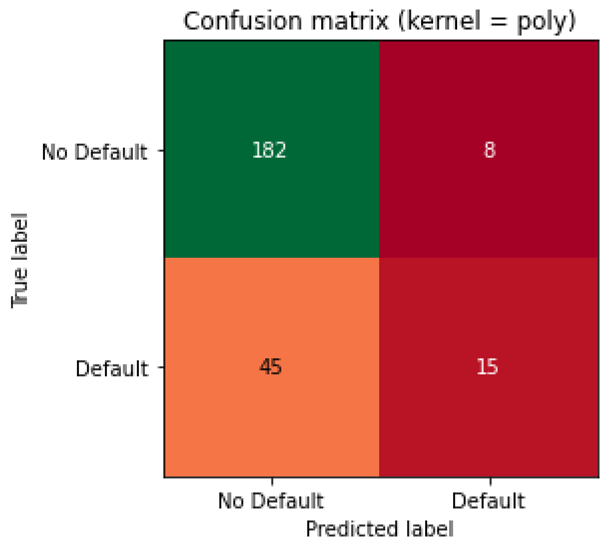
```
print(confusion_matrix(y_pred=y_pred,y_true = y_test_k))
```

```
[[182   8]
 [ 45  15]]
```

In [155]:

```
%matplotlib inline

heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test_k), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix (kernel = poly)")
plt.gca().invert_yaxis()
```



In [156]:

```
print('Support Vector Machine(kernel = poly)-PCA f1-score: {:.3f} '.format(f1_score(y_t
```

```
Support Vector Machine(kernel = poly)-PCA f1-score: 0.361
```

In [157]:

```
print('Support Vector Machine(kernel = poly)-PCA Accuracy : {:.3f} '.format(accuracy_sc
```

```
Support Vector Machine(kernel = poly)-PCA Accuracy : 0.788
```

'C': 10, 'gamma': 1, kernel = linear

In [158]:

```
kernel_lin = SVC(C = 10, gamma = 1, kernel = 'linear', verbose = 1)
kernel_lin.fit(X_train_k,y_train_k)
y_pred = kernel_lin.predict(X_test_k)
```

```
[LibSVM]
```

In [159]:

```
print('Training score: {:.3f}'.format(kernel_lin.score(X_train_k, y_train_k)))
print('Testing score: {:.3f}'.format(kernel_lin.score(X_test_k, y_test_k)))
```

```
Training score: 0.849
Testing score: 0.792
```

In [160]:

```
print(classification_report(y_pred,y_test_k))
```

```
              precision    recall  f1-score   support

         0.0       0.98      0.79      0.88       236
         1.0       0.18      0.79      0.30        14

    accuracy                           0.79       250
   macro avg       0.58      0.79      0.59       250
weighted avg       0.94      0.79      0.85       250
```

In [161]:

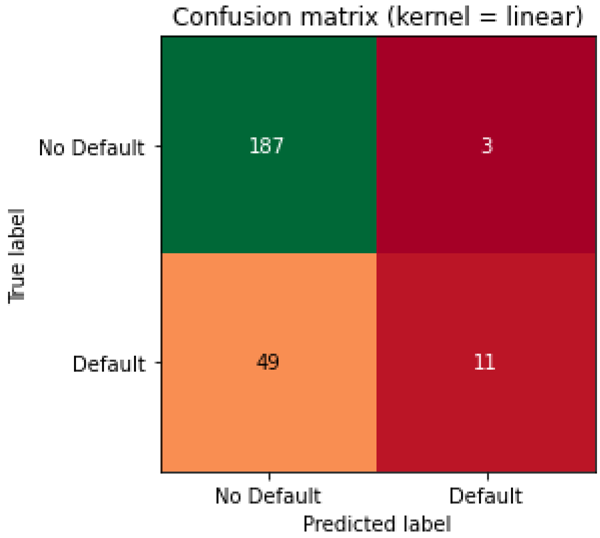
```
print(confusion_matrix(y_pred=y_pred,y_true = y_test_k))
```

```
[[187   3]
 [ 49  11]]
```

In [162]:

```
%matplotlib inline

heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test_k), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix (kernel = linear)")
plt.gca().invert_yaxis()
```



In [163]:

```
print('Support Vector Machine(kernel = linear)-f1 score: {:.3f} '.format(f1_score(y_tes
```

```
Support Vector Machine(kernel = linear)-f1 score: 0.297
```

In [164]:

```
print('Support Vector Machine(kernel = linear)-PCA Accuracy : {:.3f} '.format(accuracy_
```

```
Support Vector Machine(kernel = linear)-PCA Accuracy : 0.792
```

'C': 10, 'gamma': 1, kernel = rbf

In [165]:

```
kernel_rbf = SVC(C = 10, gamma = 1, kernel = 'rbf', verbose = 1)
kernel_rbf.fit(X_train_k,y_train_k)
y_pred = kernel_rbf.predict(X_test_k)
```

```
[LibSVM]
```

In [166]:

```
print('Training score: {:.3f}'.format(kernel_rbf.score(X_train_k, y_train_k)))
print('Testing score: {:.3f}'.format(kernel_rbf.score(X_test_k, y_test_k)))
```

```
Training score: 0.907
Testing score: 0.796
```

In [167]:

```
print(classification_report(y_pred,y_test_k))
```

```
              precision    recall  f1-score   support

         0.0       0.97      0.80      0.88       229
         1.0       0.25      0.71      0.37        21

    accuracy                           0.80       250
   macro avg       0.61      0.76      0.62       250
weighted avg       0.91      0.80      0.84       250
```

In [168]:

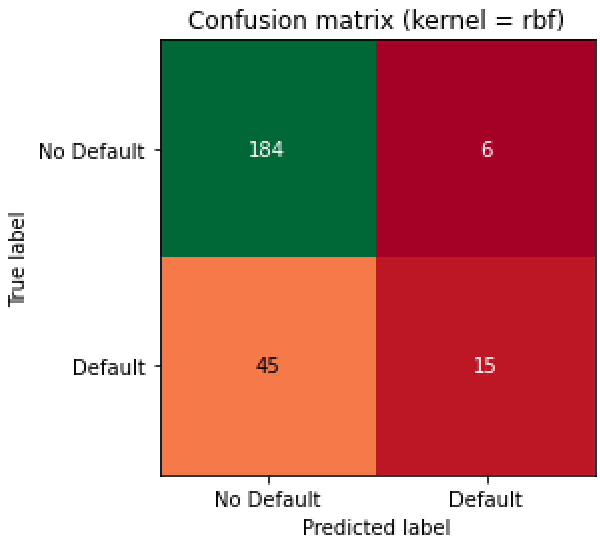
```
print(confusion_matrix(y_pred=y_pred,y_true = y_test_k))
```

```
[[184   6]
 [ 45  15]]
```

In [169]:

```
%matplotlib inline

heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test_k), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix (kernel = rbf)")
plt.gca().invert_yaxis()
```



In [170]:

```
print('Support Vector Machine(kernel = rbf)-PCA f1-score : {:.3f} '.format(f1_score(y_t
```

Support Vector Machine(kernel = rbf)-PCA f1-score : 0.370

In [171]:

```
print('Support Vector Machine(kernel = rbf)-PCA Accuracy : {:.3f} '.format(accuracy_sco
```

Support Vector Machine(kernel = rbf)-PCA Accuracy : 0.796

5.Decision Tree Classification - PCA

In [172]:

```
gs_dtree_para = {'max_depth': range(1,20),'criterion':['gini','entropy'],'min_samples_l
gs_dtree= GridSearchCV(DecisionTreeClassifier(), cv = 5, param_grid = gs_dtree_para , \
gs_dtree.fit(X_train_pca, y_train)
```

```
Fitting 5 folds for each of 1824 candidates, totalling 9120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.4s
[Parallel(n_jobs=-1)]: Done 656 tasks      | elapsed:    5.7s
[Parallel(n_jobs=-1)]: Done 1656 tasks      | elapsed:   26.1s
[Parallel(n_jobs=-1)]: Done 2096 tasks      | elapsed:   39.4s
[Parallel(n_jobs=-1)]: Done 2546 tasks      | elapsed:   53.2s
[Parallel(n_jobs=-1)]: Done 3096 tasks      | elapsed:  1.2min
[Parallel(n_jobs=-1)]: Done 3746 tasks      | elapsed:  1.7min
[Parallel(n_jobs=-1)]: Done 4496 tasks      | elapsed:  2.2min
[Parallel(n_jobs=-1)]: Done 6092 tasks      | elapsed:  2.8min
[Parallel(n_jobs=-1)]: Done 7616 tasks      | elapsed:  4.2min
[Parallel(n_jobs=-1)]: Done 8666 tasks      | elapsed:  5.3min
[Parallel(n_jobs=-1)]: Done 9120 out of 9120 | elapsed:  5.8min finished
```

Out[172]:

In [173]:

```
gs_dtree.best_score_
```

Out[173]:

```
0.8061333333333334
```

In [174]:

```
print("Decision Tree Grid Search Best Parameters {}".format(gs_dtree.best_params_))
```

```
Decision Tree Grid Search Best Parameters {'criterion': 'entropy', 'max_dept
h': 7, 'min_samples_leaf': 20}
```

Applying best parameters 'criterion': 'entropy', 'max_depth': 7, 'min_samples_leaf': 20 obtained using grid search

In [175]:

```
dtree_pca = DecisionTreeClassifier(criterion ='entropy', max_depth= 7, min_samples_leaf
dtree_pca.fit(X_train_pca, y_train)
y_pred = dtree_pca.predict(X_test_pca)
```

In [176]:

```
print('Training score: {:.3f}'.format(dtree_pca.score(X_train_pca,y_train)))
print('Testing score: {:.3f}'.format(dtree_pca.score(X_test_pca,y_test)))
```

```
Training score: 0.823
Testing score: 0.808
```

In [177]:

```
print(classification_report(y_pred = y_pred, y_true = y_test))
```

```
              precision    recall  f1-score   support

         0.0       0.84      0.94      0.89      5950
         1.0       0.57      0.29      0.38      1550

    accuracy                           0.81      7500
   macro avg       0.70      0.62      0.63      7500
weighted avg       0.78      0.81      0.78      7500
```

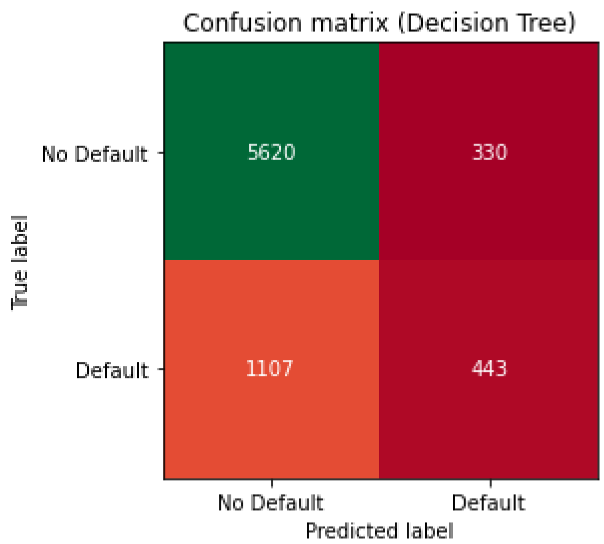
In [178]:

```
print(confusion_matrix(y_pred = y_pred, y_true = y_test))
```

```
[[5620  330]
 [1107  443]]
```

In [179]:

```
import mglearn
heatmap = mglearn.tools.heatmap(
    confusion_matrix(y_pred = y_pred, y_true = y_test), xlabel = 'Predicted label',
    ylabel='True label', xticklabels = ['No Default','Default'], yticklabels=['No Defau
plt.title("Confusion matrix (Decision Tree)")
plt.gca().invert_yaxis()
```



In [180]:

```
print('Decision tree f1-score : {:.3f} '.format(f1_score(y_test, dtree_pca.predict(X_te
```

```
Decision tree f1-score : 0.381
```

In [181]:

```
print('Decision tree Accuracy : {:.3f} '.format(accuracy_score(y_test, dtree_pca.predic
```

```
Decision tree Accuracy : 0.808
```

Let's compare the values of models from Project 1(without PCA) and Project 2(with PCA)

Based on our data, we are more concerened about False Positives and False negative and hence one of the measure to evaluate the model performance can be F1 score. The highest possible value of an F-score is 1.0, indicating perfect precision and recall, and the lowest possible value is 0, if either the precision or the recall is zero

Accuracy score before and after PCA

In [182]:

```
print('KNN-PCA Accuracy - Without PCA :  0.81 ')
print('Logistic Regression-PCA Accuracy - Without PCA : 0.82')
print('Linear Support Vector Machine Accuracy - Without PCA : 0.81')
print('Support Vector Machine(kernel = poly)Accuracy - Without PCA : 0.77')
print('Support Vector Machine(kernel = linear)Accuracy - Without PCA: 0.76')
print('Support Vector Machine(kernel = rbf)Accuracy - Without PCA: 0.76')
print('Decision tree Accuracy - Without PCA: : 0.83')
```

```
KNN-PCA Accuracy - Without PCA :  0.81
Logistic Regression-PCA Accuracy - Without PCA : 0.82
Linear Support Vector Machine Accuracy - Without PCA : 0.81
Support Vector Machine(kernel = poly)Accuracy - Without PCA : 0.77
Support Vector Machine(kernel = linear)Accuracy - Without PCA: 0.76
Support Vector Machine(kernel = rbf)Accuracy - Without PCA: 0.76
Decision tree Accuracy - Without PCA: : 0.83
```

In [183]:

```
print('KNN-PCA Accuracy - PCA : {:.2f} '.format(accuracy_score(y_test, knn_pca.predict(
print('Logistic Regression-PCA Accuracy : {:.2f} '.format(accuracy_score(y_test, logit_
print('Linear Support Vector Machine -PCA Accuracy : {:.2f} '.format(accuracy_score(y_t
print('Support Vector Machine(kernel = poly)-PCA Accuracy : {:.2f} '.format(accuracy_sc
print('Support Vector Machine(kernel = linear)-PCA Accuracy : {:.2f} '.format(accuracy_
print('Support Vector Machine(kernel = rbf)-PCA Accuracy : {:.2f} '.format(accuracy_sco
print('Decision tree Accuracy - PCA : {:.2f} '.format(accuracy_score(y_test, dtree_pca.
```

```
KNN-PCA Accuracy - PCA : 0.81
Logistic Regression-PCA Accuracy : 0.81
Linear Support Vector Machine -PCA Accuracy : 0.81
Support Vector Machine(kernel = poly)-PCA Accuracy : 0.79
Support Vector Machine(kernel = linear)-PCA Accuracy : 0.79
Support Vector Machine(kernel = rbf)-PCA Accuracy : 0.80
Decision tree Accuracy - PCA : 0.81
```

f1- score before and after PCA

In [184]:

```
print('KNN-PCA f1_score - Without PCA :  0.35 ')
print('Logistic f1_score-PCA Accuracy - Without PCA : 0.30')
print('Linear Support Vector Machine f1_score - Without PCA : 0.17')
print('Support Vector Machine(kernel = poly) f1_score - Without PCA : 0.17')
print('Support Vector Machine(kernel = linear) f1_score - Without PCA: 0')
print('Support Vector Machine(kernel = rbf) f1_score - Without PCA: 0')
print('Decision tree f1_score- Without PCA: : 0.46')
```

```
KNN-PCA f1_score - Without PCA :  0.35
Logistic f1_score-PCA Accuracy - Without PCA : 0.30
Linear Support Vector Machine f1_score - Without PCA : 0.17
Support Vector Machine(kernel = poly) f1_score - Without PCA : 0.17
Support Vector Machine(kernel = linear) f1_score - Without PCA: 0
Support Vector Machine(kernel = rbf) f1_score - Without PCA: 0
Decision tree f1_score- Without PCA: : 0.46
```

In [185]:

```
print('KNN-PCA f1_score - PCA : {:.2f} '.format(f1_score(y_test, knn_pca.predict(X_test
print('Logistic Regression-PCA f1_score : {:.2f} '.format(f1_score(y_test, logit_pca.pr
print('Linear Support Vector Machine -PCA f1_score : {:.2f} '.format(f1_score(y_test, l
print('Support Vector Machine(kernel = poly)-PCA f1_score : {:.2f} '.format(f1_score(y_
print('Support Vector Machine(kernel = linear)-PCA f1_score : {:.2f} '.format(f1_score(
print('Support Vector Machine(kernel = rbf)-PCA f1_score : {:.2f} '.format(f1_score(y_t
print('Decision tree f1_score - PCA : {:.2f} '.format(f1_score(y_test, dtree_pca.predic
```

```
KNN-PCA f1_score - PCA : 0.37
Logistic Regression-PCA f1_score : 0.25
Linear Support Vector Machine -PCA f1_score : 0.22
Support Vector Machine(kernel = poly)-PCA f1_score : 0.36
Support Vector Machine(kernel = linear)-PCA f1_score : 0.30
Support Vector Machine(kernel = rbf)-PCA f1_score : 0.37
Decision tree f1_score - PCA : 0.38
```

Conclusion

The objective of Principal Component Analysis is to find a low-dimension set of axes that summarize data. We can clearly observe the following things by comparing similar models that were designed with and without PCA

1. The accuracy score remains the almost same for all the models after PCA, except in case of SVM where the accuracy score gets better after we reduce the dimension using PCA. Primararily because principal components are linear combinations of original variable so when we peform SVM on dimensionally reduced data we are not working on the original data and hence the accuracy of the SVM on dimensionally reduced data is high due to increasing interpretability and information Rloss.
2. Based on our dataset our focus is on people who are 'defaulters [category = 1]'. Being an imbalanced dataset,the f1-score is a better parameter to compare the model performance as f1-score is more concerned about False Positives and False negatives and the score becomes high only when both precision and recall are high. Hence f1 score of models can better help me compare the model performance.
3. The f1-score slighlty reduces in the models using PCA processed data possibly because while reducing the dimensionality of data we are also reducing a bit of noise from it.
4. Overall we are able to achieve similar result for both PCA and Non-PCA dataset but by performing PCA we are achieving similar results by using a much much smaller computation power.

6. Apply deep learning models

In [186]:

```
# To stack layers on each other we are using sequential model
from keras.models import Sequential

#To establish connectivity between the nodes we are using Dense
from keras.layers import Dense

#for reproductability we are going to assign seed value of 10
np.random.seed(10)
```

In [187]:

```
model = Sequential()
# input layer,  the number of the nodes we would like to consider in the input layer wh
model.add(Dense(12, input_dim=23, activation='relu'))

#To connect the input and output layer we need a hidden layer, the number of nodes in t
model.add(Dense(8, activation='relu'))

#For the output layer since we have two disjoint classes (0 or 1), we can use the sigmo
model.add(Dense(1, activation='sigmoid'))
```

Now let's proceed toward compiling the model, for this purpose we need three parameter, they are objective function, optimizer and evaluation metrics

In [188]:

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

For fitting the model we need two parameters to be defined, they are epochs and batch_size

In [189]:

```
model.fit(X_train, y_train, epochs=150, batch_size=10)
```

```
Epoch 1/150
2250/2250 [==============================] - 2s 667us/step - loss: 0.4998 -
accuracy: 0.7808
Epoch 2/150
2250/2250 [==============================] - 1s 612us/step - loss: 0.4551 -
accuracy: 0.8107
Epoch 3/150
2250/2250 [==============================] - 1s 663us/step - loss: 0.4430 -
accuracy: 0.8160
Epoch 4/150
2250/2250 [==============================] - 2s 758us/step - loss: 0.4410 -
accuracy: 0.8152
Epoch 5/150
2250/2250 [==============================] - 2s 717us/step - loss: 0.4457 -
accuracy: 0.8137
Epoch 6/150
2250/2250 [==============================] - 1s 639us/step - loss: 0.4390 -
accuracy: 0.8177
Epoch 7/150
2250/2250 [==============================] - 2s 760us/step - loss: 0.4404 -
accuracy: 0.8161
Epoch 8/150
2250/2250 [==============================] - 1s 615us/step - loss: 0.4416 -
accuracy: 0.8159
Epoch 9/150
2250/2250 [==============================] - 2s 688us/step - loss: 0.4342 -
accuracy: 0.8196
Epoch 10/150
2250/2250 [==============================] - 2s 745us/step - loss: 0.4331 -
accuracy: 0.8197
Epoch 11/150
2250/2250 [==============================] - 1s 591us/step - loss: 0.4338 -
accuracy: 0.8169
Epoch 12/150
2250/2250 [==============================] - 2s 695us/step - loss: 0.4297 -
accuracy: 0.8231
Epoch 13/150
2250/2250 [==============================] - 2s 686us/step - loss: 0.4357 -
accuracy: 0.8160
Epoch 14/150
2250/2250 [==============================] - 2s 717us/step - loss: 0.4354 -
accuracy: 0.8199
Epoch 15/150
2250/2250 [==============================] - 2s 671us/step - loss: 0.4395 -
accuracy: 0.8176
Epoch 16/150
2250/2250 [==============================] - 2s 676us/step - loss: 0.4346 -
accuracy: 0.8200
Epoch 17/150
2250/2250 [==============================] - 2s 846us/step - loss: 0.4365 -
accuracy: 0.8163
Epoch 18/150
2250/2250 [==============================] - 2s 732us/step - loss: 0.4349 -
accuracy: 0.8190
Epoch 19/150
2250/2250 [==============================] - 2s 749us/step - loss: 0.4359 -
accuracy: 0.8160
```

```
Epoch 20/150
2250/2250 [==============================] - 2s 706us/step - loss: 0.4375 -
accuracy: 0.8162
Epoch 21/150
2250/2250 [==============================] - 2s 711us/step - loss: 0.4333 -
accuracy: 0.8184
Epoch 22/150
2250/2250 [==============================] - 2s 754us/step - loss: 0.4320 -
accuracy: 0.8206
Epoch 23/150
2250/2250 [==============================] - 2s 709us/step - loss: 0.4320 -
accuracy: 0.8184
Epoch 24/150
2250/2250 [==============================] - 2s 701us/step - loss: 0.4243 -
accuracy: 0.8253
Epoch 25/150
2250/2250 [==============================] - 2s 708us/step - loss: 0.4291 -
accuracy: 0.8182
Epoch 26/150
2250/2250 [==============================] - 2s 670us/step - loss: 0.4344 -
accuracy: 0.8190
Epoch 27/150
2250/2250 [==============================] - 2s 847us/step - loss: 0.4318 -
accuracy: 0.8202
Epoch 28/150
2250/2250 [==============================] - 1s 619us/step - loss: 0.4274 -
accuracy: 0.8226
Epoch 29/150
2250/2250 [==============================] - 1s 666us/step - loss: 0.4323 -
accuracy: 0.8179
Epoch 30/150
2250/2250 [==============================] - 2s 675us/step - loss: 0.4193 -
accuracy: 0.8263
Epoch 31/150
2250/2250 [==============================] - 1s 661us/step - loss: 0.4271 -
accuracy: 0.8219
Epoch 32/150
2250/2250 [==============================] - 2s 671us/step - loss: 0.4316 -
accuracy: 0.8175
Epoch 33/150
2250/2250 [==============================] - 1s 666us/step - loss: 0.4296 -
accuracy: 0.8207
Epoch 34/150
2250/2250 [==============================] - 2s 692us/step - loss: 0.4254 -
accuracy: 0.8207
Epoch 35/150
2250/2250 [==============================] - 2s 717us/step - loss: 0.4289 -
accuracy: 0.8186
Epoch 36/150
2250/2250 [==============================] - 2s 676us/step - loss: 0.4322 -
accuracy: 0.8180
Epoch 37/150
2250/2250 [==============================] - 2s 760us/step - loss: 0.4202 -
accuracy: 0.8248
Epoch 38/150
2250/2250 [==============================] - 2s 753us/step - loss: 0.4261 -
accuracy: 0.8233
Epoch 39/150
2250/2250 [==============================] - 2s 670us/step - loss: 0.4365 -
accuracy: 0.8166
Epoch 40/150
```

```
2250/2250 [==============================] - 2s 790us/step - loss: 0.4257 -
accuracy: 0.8208
Epoch 41/150
2250/2250 [==============================] - 1s 647us/step - loss: 0.4304 -
accuracy: 0.8194
Epoch 42/150
2250/2250 [==============================] - 1s 644us/step - loss: 0.4325 -
accuracy: 0.8187
Epoch 43/150
2250/2250 [==============================] - 1s 642us/step - loss: 0.4288 -
accuracy: 0.8196
Epoch 44/150
2250/2250 [==============================] - 1s 644us/step - loss: 0.4336 -
accuracy: 0.8185
Epoch 45/150
2250/2250 [==============================] - 1s 661us/step - loss: 0.4233 -
accuracy: 0.8248
Epoch 46/150
2250/2250 [==============================] - 2s 676us/step - loss: 0.4299 -
accuracy: 0.8205
Epoch 47/150
2250/2250 [==============================] - 2s 719us/step - loss: 0.4251 -
accuracy: 0.8231
Epoch 48/150
2250/2250 [==============================] - 2s 828us/step - loss: 0.4228 -
accuracy: 0.8258
Epoch 49/150
2250/2250 [==============================] - 2s 669us/step - loss: 0.4298 -
accuracy: 0.8194
Epoch 50/150
2250/2250 [==============================] - 1s 664us/step - loss: 0.4237 -
accuracy: 0.8236
Epoch 51/150
2250/2250 [==============================] - 1s 655us/step - loss: 0.4257 -
accuracy: 0.8218
Epoch 52/150
2250/2250 [==============================] - 2s 687us/step - loss: 0.4279 -
accuracy: 0.8218
Epoch 53/150
2250/2250 [==============================] - 1s 626us/step - loss: 0.4237 -
accuracy: 0.8244
Epoch 54/150
2250/2250 [==============================] - 1s 616us/step - loss: 0.4216 -
accuracy: 0.8238
Epoch 55/150
2250/2250 [==============================] - 1s 659us/step - loss: 0.4232 -
accuracy: 0.8246
Epoch 56/150
2250/2250 [==============================] - 2s 706us/step - loss: 0.4281 -
accuracy: 0.8220
Epoch 57/150
2250/2250 [==============================] - 2s 682us/step - loss: 0.4254 -
accuracy: 0.8240
Epoch 58/150
2250/2250 [==============================] - 2s 799us/step - loss: 0.4321 -
accuracy: 0.8174
Epoch 59/150
2250/2250 [==============================] - 2s 716us/step - loss: 0.4299 -
accuracy: 0.8184
Epoch 60/150
2250/2250 [==============================] - 2s 801us/step - loss: 0.4198 -
```

```
accuracy: 0.8235
Epoch 61/150
2250/2250 [==============================] - 2s 677us/step - loss: 0.4273 -
accuracy: 0.8171
Epoch 62/150
2250/2250 [==============================] - 2s 756us/step - loss: 0.4277 -
accuracy: 0.8190
Epoch 63/150
2250/2250 [==============================] - 2s 675us/step - loss: 0.4264 -
accuracy: 0.8222
Epoch 64/150
2250/2250 [==============================] - 2s 694us/step - loss: 0.4252 -
accuracy: 0.8239
Epoch 65/150
2250/2250 [==============================] - 2s 694us/step - loss: 0.4314 -
accuracy: 0.8183
Epoch 66/150
2250/2250 [==============================] - 2s 708us/step - loss: 0.4208 -
accuracy: 0.8230
Epoch 67/150
2250/2250 [==============================] - 2s 732us/step - loss: 0.4223 -
accuracy: 0.8238
Epoch 68/150
2250/2250 [==============================] - 2s 881us/step - loss: 0.4303 -
accuracy: 0.8175
Epoch 69/150
2250/2250 [==============================] - 2s 744us/step - loss: 0.4226 -
accuracy: 0.8249
Epoch 70/150
2250/2250 [==============================] - 2s 670us/step - loss: 0.4226 -
accuracy: 0.8239
Epoch 71/150
2250/2250 [==============================] - 2s 685us/step - loss: 0.4219 -
accuracy: 0.8247
Epoch 72/150
2250/2250 [==============================] - 2s 705us/step - loss: 0.4237 -
accuracy: 0.8218
Epoch 73/150
2250/2250 [==============================] - 2s 701us/step - loss: 0.4253 -
accuracy: 0.8206
Epoch 74/150
2250/2250 [==============================] - 2s 689us/step - loss: 0.4244 -
accuracy: 0.8223
Epoch 75/150
2250/2250 [==============================] - 2s 680us/step - loss: 0.4190 -
accuracy: 0.8244
Epoch 76/150
2250/2250 [==============================] - 1s 657us/step - loss: 0.4216 -
accuracy: 0.8236
Epoch 77/150
2250/2250 [==============================] - 2s 716us/step - loss: 0.4199 -
accuracy: 0.8258
Epoch 78/150
2250/2250 [==============================] - 2s 885us/step - loss: 0.4155 -
accuracy: 0.8278
Epoch 79/150
2250/2250 [==============================] - 2s 687us/step - loss: 0.4240 -
accuracy: 0.8209
Epoch 80/150
2250/2250 [==============================] - 2s 734us/step - loss: 0.4228 -
accuracy: 0.8235
```

```
Epoch 81/150
2250/2250 [==============================] - 2s 693us/step - loss: 0.4239 -
accuracy: 0.8223
Epoch 82/150
2250/2250 [==============================] - 2s 746us/step - loss: 0.4186 -
accuracy: 0.8254
Epoch 83/150
2250/2250 [==============================] - 2s 785us/step - loss: 0.4267 -
accuracy: 0.8223
Epoch 84/150
2250/2250 [==============================] - 2s 718us/step - loss: 0.4238 -
accuracy: 0.8204
Epoch 85/150
2250/2250 [==============================] - 2s 715us/step - loss: 0.4256 -
accuracy: 0.8198
Epoch 86/150
2250/2250 [==============================] - 2s 736us/step - loss: 0.4168 -
accuracy: 0.8272
Epoch 87/150
2250/2250 [==============================] - 2s 724us/step - loss: 0.4185 -
accuracy: 0.8254
Epoch 88/150
2250/2250 [==============================] - 2s 733us/step - loss: 0.4232 -
accuracy: 0.8197
Epoch 89/150
2250/2250 [==============================] - 2s 677us/step - loss: 0.4220 -
accuracy: 0.8235
Epoch 90/150
2250/2250 [==============================] - 2s 697us/step - loss: 0.4232 -
accuracy: 0.8216
Epoch 91/150
2250/2250 [==============================] - 2s 685us/step - loss: 0.4168 -
accuracy: 0.8253
Epoch 92/150
2250/2250 [==============================] - 1s 651us/step - loss: 0.4231 -
accuracy: 0.8216
Epoch 93/150
2250/2250 [==============================] - 1s 650us/step - loss: 0.4309 -
accuracy: 0.8162
Epoch 94/150
2250/2250 [==============================] - 2s 668us/step - loss: 0.4145 -
accuracy: 0.8266
Epoch 95/150
2250/2250 [==============================] - 2s 710us/step - loss: 0.4215 -
accuracy: 0.8229
Epoch 96/150
2250/2250 [==============================] - 1s 662us/step - loss: 0.4296 -
accuracy: 0.8209
Epoch 97/150
2250/2250 [==============================] - 2s 718us/step - loss: 0.4159 -
accuracy: 0.8272
Epoch 98/150
2250/2250 [==============================] - 2s 879us/step - loss: 0.4264 -
accuracy: 0.8214
Epoch 99/150
2250/2250 [==============================] - 2s 740us/step - loss: 0.4217 -
accuracy: 0.8215
Epoch 100/150
2250/2250 [==============================] - 2s 676us/step - loss: 0.4235 -
accuracy: 0.8260
Epoch 101/150
```

```
2250/2250 [==============================] - 2s 701us/step - loss: 0.4203 -
accuracy: 0.8219
Epoch 102/150
2250/2250 [==============================] - 2s 766us/step - loss: 0.4227 -
accuracy: 0.8229
Epoch 103/150
2250/2250 [==============================] - 2s 687us/step - loss: 0.4198 -
accuracy: 0.8247
Epoch 104/150
2250/2250 [==============================] - 1s 631us/step - loss: 0.4214 -
accuracy: 0.8240
Epoch 105/150
2250/2250 [==============================] - 1s 643us/step - loss: 0.4255 -
accuracy: 0.8199
Epoch 106/150
2250/2250 [==============================] - 1s 639us/step - loss: 0.4180 -
accuracy: 0.8231
Epoch 107/150
2250/2250 [==============================] - 1s 644us/step - loss: 0.4175 -
accuracy: 0.8264
Epoch 108/150
2250/2250 [==============================] - 2s 762us/step - loss: 0.4128 -
accuracy: 0.8292
Epoch 109/150
2250/2250 [==============================] - 1s 649us/step - loss: 0.4223 -
accuracy: 0.8224
Epoch 110/150
2250/2250 [==============================] - 1s 650us/step - loss: 0.4175 -
accuracy: 0.8263
Epoch 111/150
2250/2250 [==============================] - 1s 644us/step - loss: 0.4197 -
accuracy: 0.8236
Epoch 112/150
2250/2250 [==============================] - 1s 644us/step - loss: 0.4197 -
accuracy: 0.8224
Epoch 113/150
2250/2250 [==============================] - 2s 703us/step - loss: 0.4205 -
accuracy: 0.8231
Epoch 114/150
2250/2250 [==============================] - 2s 703us/step - loss: 0.4197 -
accuracy: 0.8239
Epoch 115/150
2250/2250 [==============================] - 2s 739us/step - loss: 0.4184 -
accuracy: 0.8226
Epoch 116/150
2250/2250 [==============================] - 2s 749us/step - loss: 0.4203 -
accuracy: 0.8246
Epoch 117/150
2250/2250 [==============================] - 1s 633us/step - loss: 0.4241 -
accuracy: 0.8197
Epoch 118/150
2250/2250 [==============================] - 2s 708us/step - loss: 0.4180 -
accuracy: 0.8236
Epoch 119/150
2250/2250 [==============================] - 1s 614us/step - loss: 0.4143 -
accuracy: 0.8254
Epoch 120/150
2250/2250 [==============================] - 1s 615us/step - loss: 0.4251 -
accuracy: 0.8193
Epoch 121/150
2250/2250 [==============================] - 1s 658us/step - loss: 0.4196 -
```

```
accuracy: 0.8237
Epoch 122/150
2250/2250 [==============================] - 1s 661us/step - loss: 0.4216 -
accuracy: 0.8205
Epoch 123/150
2250/2250 [==============================] - 1s 660us/step - loss: 0.4193 -
accuracy: 0.8245
Epoch 124/150
2250/2250 [==============================] - 1s 638us/step - loss: 0.4207 -
accuracy: 0.8218
Epoch 125/150
2250/2250 [==============================] - 2s 680us/step - loss: 0.4256 -
accuracy: 0.81990s - loss: 0.4259 - accuracy: 0.
Epoch 126/150
2250/2250 [==============================] - 1s 648us/step - loss: 0.4136 -
accuracy: 0.8254
Epoch 127/150
2250/2250 [==============================] - 2s 726us/step - loss: 0.4188 -
accuracy: 0.8244
Epoch 128/150
2250/2250 [==============================] - 2s 752us/step - loss: 0.4186 -
accuracy: 0.8246
Epoch 129/150
2250/2250 [==============================] - 2s 901us/step - loss: 0.4301 -
accuracy: 0.8195
Epoch 130/150
2250/2250 [==============================] - 2s 908us/step - loss: 0.4166 -
accuracy: 0.8266
Epoch 131/150
2250/2250 [==============================] - 3s 1ms/step - loss: 0.4213 - ac
curacy: 0.8240
Epoch 132/150
2250/2250 [==============================] - 2s 795us/step - loss: 0.4191 -
accuracy: 0.8241
Epoch 133/150
2250/2250 [==============================] - 2s 1ms/step - loss: 0.4267 - ac
curacy: 0.8187
Epoch 134/150
2250/2250 [==============================] - 2s 1ms/step - loss: 0.4191 - ac
curacy: 0.8240
Epoch 135/150
2250/2250 [==============================] - 3s 1ms/step - loss: 0.4137 - ac
curacy: 0.8285
Epoch 136/150
2250/2250 [==============================] - 2s 857us/step - loss: 0.4176 -
accuracy: 0.8236
Epoch 137/150
2250/2250 [==============================] - 2s 721us/step - loss: 0.4188 -
accuracy: 0.8254
Epoch 138/150
2250/2250 [==============================] - 2s 685us/step - loss: 0.4144 -
accuracy: 0.8275
Epoch 139/150
2250/2250 [==============================] - 2s 714us/step - loss: 0.4190 -
accuracy: 0.8231
Epoch 140/150
2250/2250 [==============================] - 2s 685us/step - loss: 0.4187 -
accuracy: 0.8243
Epoch 141/150
2250/2250 [==============================] - 1s 658us/step - loss: 0.4166 -
accuracy: 0.8249
```

```
Epoch 142/150
2250/2250 [==============================] - 1s 638us/step - loss: 0.4161 -
accuracy: 0.8237
Epoch 143/150
2250/2250 [==============================] - 1s 639us/step - loss: 0.4287 -
accuracy: 0.8190
Epoch 144/150
2250/2250 [==============================] - 1s 636us/step - loss: 0.4196 -
accuracy: 0.8211
Epoch 145/150
2250/2250 [==============================] - 1s 663us/step - loss: 0.4221 -
accuracy: 0.8211
Epoch 146/150
2250/2250 [==============================] - 2s 689us/step - loss: 0.4211 -
accuracy: 0.8232
Epoch 147/150
2250/2250 [==============================] - 1s 659us/step - loss: 0.4159 -
accuracy: 0.8250
Epoch 148/150
2250/2250 [==============================] - 2s 1ms/step - loss: 0.4212 - ac
curacy: 0.8225
Epoch 149/150
2250/2250 [==============================] - 2s 786us/step - loss: 0.4201 -
accuracy: 0.8237
Epoch 150/150
2250/2250 [==============================] - 2s 859us/step - loss: 0.4173 -
accuracy: 0.8236
```

Out[189]:

```
<tensorflow.python.keras.callbacks.History at 0x2996330a760>
```

Model Evaluation

In [190]:

```
scores = model.evaluate(X_test, y_test)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
235/235 [==============================] - 0s 604us/step - loss: 0.4261 - ac
curacy: 0.8272

accuracy: 82.72%
```

End of Project 2 : Classification

Initials:

-rp