Credit Card Fraud Detection

The Problem

Our goal is to predict if a credit card transaction is fraudulent or not. Because this data is so skewed, we'll use different techniques to try to make our predictive model work even when there is a majority class.

Collecting Data

This data set came from Kaggle (https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud (https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud)). We'll need to use certain libraries to accomplish our goal moving forward. I'll import Pandas, Numpy, Matplotlib and Seaborn initially.

Importing Libraries

In [9]:

```
# scientific computing libaries
import pandas as pd
import numpy as np

# data mining libaries
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.decomposition import PCA#, FastICA
from sklearn.model_selection import train_test_split, KFold, StratifiedKFold, GridSearchCV, learning_curve
from sklearn import svm
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_curve, auc, confusion_matrix, accuracy_score, f1_score, precision_score, recall_score, r

from imblearn.pipeline import make_pipeline, Pipeline
from imblearn.over_sampling import SMOTE

#plot libaries
import plotly
import plotly.graph_objs as go
import plotly.figure_factory as ff
from plotly.offline import init_notebook_mode
init_notebook_mode(connected=True) # to show plots in notebook
from matplotlib import pyplot as plt
%matplotlib inline
import seaborn as sns

# offline plotly
from plotly.offline import plot, iplot

# do not show any warnings
import warnings
warnings.filterwarnings('ignore')

SEED = 17 # specify seed for reproducable results
pd.set_option('display.max_columns', None) # prevents abbreviation (with '...') of columns in prints

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

Loading the Data frame

In [10]:

```
df = pd.read_csv("creditcard.csv")
df.head()
```

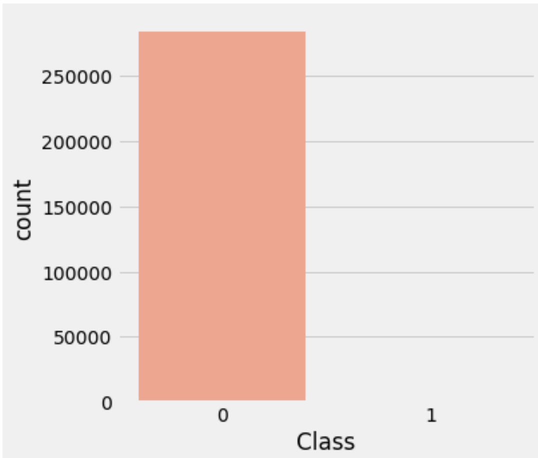
Out[10]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801	-0.991390	-0.3111
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235	0.489095	-0.1437
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.066084	0.717293	-0.1659
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.178228	0.507757	-0.2879
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.538196	1.345852	-1.1196

Checking the target feature, class as Fraud or not fraud

In [11]:

```
# apply the Fivethirtyeight style to plots.
plt.style.use("fivethirtyeight")

# display a frequency distribution for churn.
plt.figure(figsize=(5, 5))
ax = sns.countplot(x=df['Class'], palette='Reds', linewidth=1)
plt.show()
```



The plot shows a class imbalance of the data between Class (fraud) and class (not-fraud). To address this, resampling would be a suitable approach. To keep this case simple, the imbalance is kept forward and specific metrics are chosen for model evaluations.

In [12]:

```
df['Class'].value_counts(normalize=True)
```

Out[12]:

```
0    0.998273
1    0.001727
Name: Class, dtype: float64
```

It looks like they're losing 99.8% of their customers transactions are not fraudulent. Only 0.17% are.

So our original dataset is very imbalanced. Since nearly all transactions are non-fraudulent, our predictive models might make assumptions. We don't want that. We want it to actually catch fraudulent transactions.

To handle class imbalance in this classification problem, several strategies can be employed:

- Collect more data (not applicable in this case)
- Change performance metric:
- Precision, Recall, and F1 Score using confusion matrix
- Kappa for normalized accuracy
- ROC curves to calculate sensitivity/specificity ratio
- Resample the dataset:
- Over-sampling by adding copies of under-represented class (for little data)
- Under-sampling by removing instances from over-represented class (for lots of data)

In [13]:

```
from sklearn.preprocessing import StandardScaler

amount_col = df['Amount'].values.reshape(-1, 1)
scaler = StandardScaler().fit(amount_col)
df['normAmount'] = scaler.transform(amount_col)
df.drop(['Time', 'Amount'], axis=1, inplace=True)
df.head()
```

Out[13]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801	-0.991390	-0.311169	1.
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235	0.489095	-0.143772	0.
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.066084	0.717293	-0.165946	2.
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.178228	0.507757	-0.287924	-0.
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.538196	1.345852	-1.119670	0.

Assign X and Y without resampling

In [14]:

```
X = df.loc[:, df.columns != 'Class']
y = df.loc[:, df.columns == 'Class']
```

Undersample

In [15]:

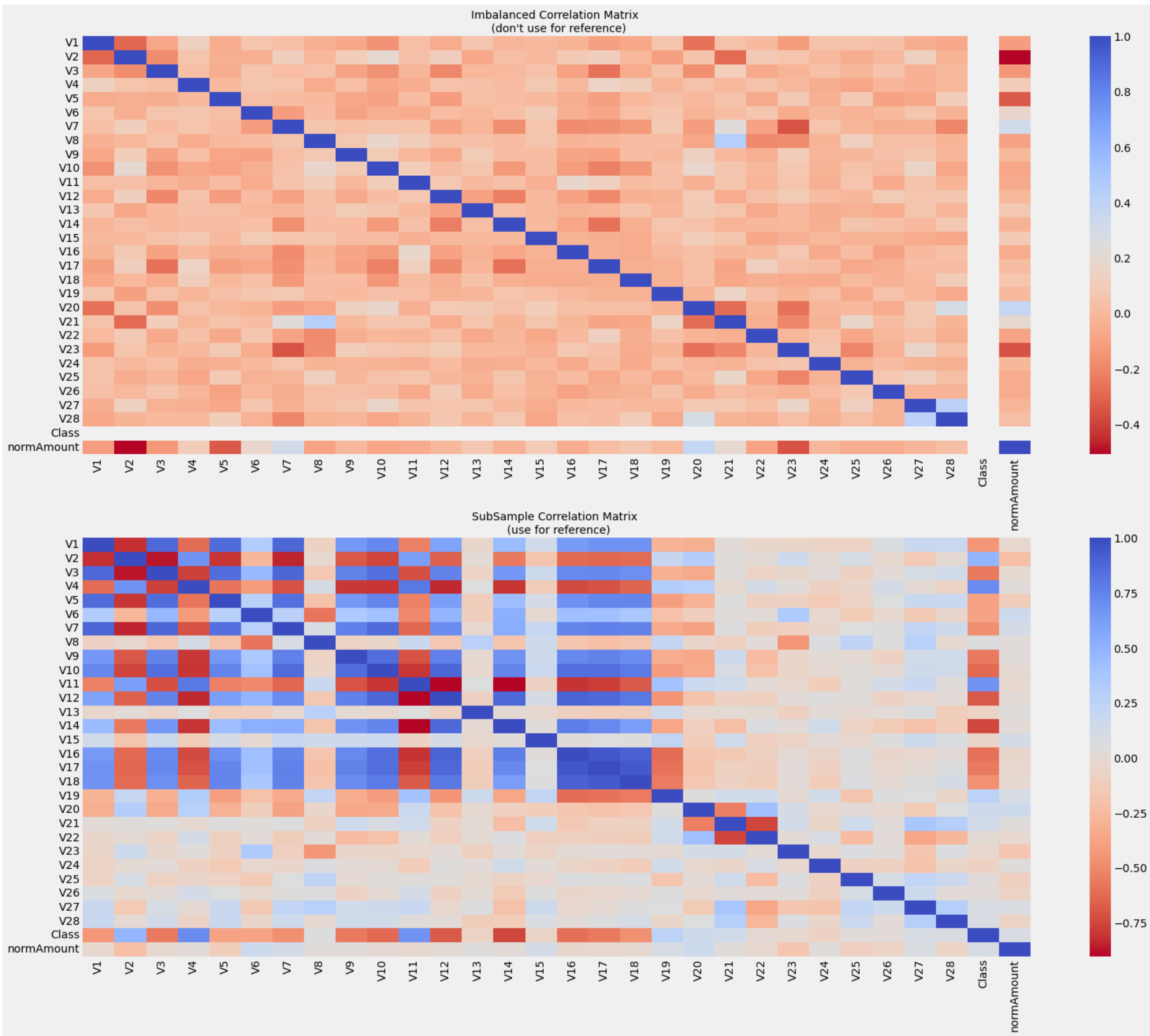
```
# Find the indices of the minority class
fraud_indices = df[df.Class == 1].index

# Find the indices of the majority class
normal_indices = df[df.Class == 0].index

# Sample an equal number of instances from the majority class as the minority class
normal_sample = df.loc[np.random.choice(normal_indices, len(fraud_indices), replace=False), :]

# Concatenate the minority class and the sampled majority class
under_sample_data = pd.concat([df.loc[fraud_indices], normal_sample])

# Split the data into X (features) and y (labels)
X_undersample = under_sample_data.drop("Class", axis=1)
y_undersample = under_sample_data["Class"]

# Calculate class proportions
normal_prop = len(under_sample_data[under_sample_data.Class == 0]) / len(under_sample_data)
fraud_prop = len(under_sample_data[under_sample_data.Class == 1]) / len(under_sample_data)

# Print class proportions
print("Percentage of normal transactions: {:.2f}%".format(normal_prop * 100))
print("Percentage of fraud transactions: {:.2f}%".format(fraud_prop * 100))
print("Total number of transactions in resampled data: {}".format(len(under_sample_data)))
```

```
Percentage of normal transactions: 50.00%
Percentage of fraud transactions: 50.00%
Total number of transactions in resampled data: 984
```

Much better!

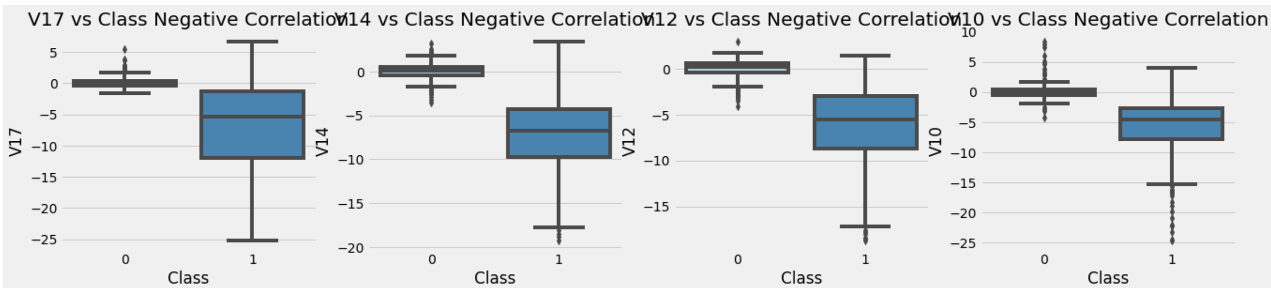
In [16]:

```
# Make sure we use the subsample in our correlation

f, (ax1, ax2) = plt.subplots(2, 1, figsize=(24,20))

# Entire DataFrame
corr = normal_sample.corr()
sns.heatmap(corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax1)
ax1.set_title("Imbalanced Correlation Matrix \n (don't use for reference)", fontsize=14)

sub_sample_corr = under_sample_data.corr()
sns.heatmap(sub_sample_corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax2)
ax2.set_title('SubSample Correlation Matrix \n (use for reference)', fontsize=14)
plt.show()

```



In [17]:

```python
f, axes = plt.subplots(ncols=4, figsize=(20,4))

# Negative Correlations with our Class (The lower our feature value the more likely it will be a fraud transaction)
sns.boxplot(x="Class", y="V17", data=under_sample_data, palette='Blues', ax=axes[0])
axes[0].set_title('V17 vs Class Negative Correlation')

sns.boxplot(x="Class", y="V14", data=under_sample_data, palette='Blues', ax=axes[1])
axes[1].set_title('V14 vs Class Negative Correlation')


sns.boxplot(x="Class", y="V12", data=under_sample_data, palette='Blues', ax=axes[2])
axes[2].set_title('V12 vs Class Negative Correlation')


sns.boxplot(x="Class", y="V10", data=under_sample_data, palette='Blues', ax=axes[3])
axes[3].set_title('V10 vs Class Negative Correlation')

plt.show()
```



Train Test Split

In [18]:

```python
from sklearn.model_selection import train_test_split

# Split the whole dataset into training and testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

# Print the size of each dataset
print("Number of transactions in the training dataset: ", len(X_train))
print("Number of transactions in the testing dataset: ", len(X_test))
print("Total number of transactions: ", len(X_train) + len(X_test))

# Split the undersampled dataset into training and testing
X_train_undersample, X_test_undersample, y_train_undersample, y_test_undersample = train_test_split(X_undersample, y_und

# Print the size of each dataset
print("\nNumber of transactions in the training dataset: ", len(X_train_undersample))
print("Number of transactions in the testing dataset: ", len(X_test_undersample))
print("Total number of transactions: ", len(X_train_undersample) + len(X_test_undersample))
```

```
Number of transactions in the training dataset:  199364
Number of transactions in the testing dataset:  85443
Total number of transactions:  284807

Number of transactions in the training dataset:  688
Number of transactions in the testing dataset:  296
Total number of transactions:  984
```

Logistic Regression

We are highly focused on recall score as it helps us identify the maximum number of fraudulent transactions. Recall, along with accuracy and precision, is a key metric for evaluating a confusion matrix. Recall calculates the number of true positive instances divided by the sum of true positive instances and false negative instances.

Since our dataset is imbalanced, many normal transactions may be predicted as fraudulent, leading to false negatives. Recall helps us address this issue. Increasing recall may lead to a decrease in precision, but that is acceptable in our scenario, as predicting a fraudulent transaction as normal is not as critical as the opposite.

We can also apply a cost function to assign different weights to false negatives and false positives, but we'll leave that for another time.

In [19]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import confusion_matrix,precision_recall_curve,auc,roc_auc_score,roc_curve,recall_score,classificat
```

In [20]:

```
from sklearn.model_selection import KFold
from sklearn.linear_model import LogisticRegression
import numpy as np
import pandas as pd
from sklearn.metrics import recall_score

def print_kfold_scores(x_train_data, y_train_data):
    k_fold = KFold(n_splits=5, shuffle=False)
    c_param_range = [0.01, 0.1, 1, 10, 100]
    results_table = pd.DataFrame(index=range(len(c_param_range)), columns=['C_parameter', 'Mean recall score'])
    j = 0
    for c_param in c_param_range:
        print('C parameter: ', c_param)
        recall_accs = []
        for train, test in k_fold.split(x_train_data):
            lr = LogisticRegression(C=c_param, penalty='l2')
            lr.fit(x_train_data.iloc[train], y_train_data.iloc[train])
            y_pred = lr.predict(x_train_data.iloc[test])
            recall = recall_score(y_train_data.iloc[test], y_pred)
            recall_accs.append(recall)
            print('Recall: ', recall)
        results_table.loc[j, 'C_parameter'] = c_param
        results_table.loc[j, 'Mean recall score'] = np.mean(recall_accs)
        j += 1
    best_c = results_table.loc[results_table['Mean recall score'].astype(float).idxmax()]['C_parameter']
    print('Best model to choose from cross validation is with C parameter =', best_c)
    return best_c

best_c = print_kfold_scores(X_train_undersample, y_train_undersample)
best_c
```

```
C parameter:  0.01
Recall:  0.821917808219178
Recall:  0.8493150684931506
Recall:  0.9152542372881356
Recall:  0.9324324324324325
Recall:  0.8787878787878788
C parameter:  0.1
Recall:  0.8356164383561644
Recall:  0.863013698630137
Recall:  0.9491525423728814
Recall:  0.9324324324324325
Recall:  0.8939393939393939
C parameter:  1
Recall:  0.8356164383561644
Recall:  0.863013698630137
Recall:  0.9661016949152542
Recall:  0.9324324324324325
Recall:  0.8939393939393939
C parameter:  10
Recall:  0.8493150684931506
Recall:  0.863013698630137
Recall:  0.9491525423728814
Recall:  0.9324324324324325
Recall:  0.8939393939393939
C parameter:  100
Recall:  0.8493150684931506
Recall:  0.863013698630137
Recall:  0.9661016949152542
Recall:  0.9459459459459459
Recall:  0.8787878787878788
Best model to choose from cross validation is with C parameter = 100
```

Out[20]: 100

Make a function to create confusion Matrix's

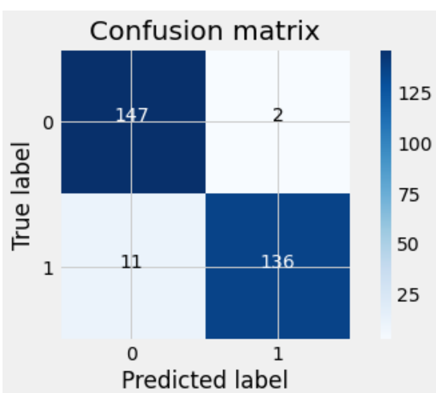
In [21]:

```
import itertools

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=0)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        #print("Normalized confusion matrix")
    else:
        1#print('Confusion matrix, without normalization')

    #print(cm)

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

In [22]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
import numpy as np
import matplotlib.pyplot as plt

def plot_confusion_matrix(cm, classes,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=0)
    plt.yticks(tick_marks, classes)

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Use this C_parameter to build the final model with the whole training dataset and predict the classes in the test
# dataset
lr = LogisticRegression(C=best_c, penalty='l2')
lr.fit(X_train_undersample, y_train_undersample.values.ravel())
y_pred_undersample = lr.predict(X_test_undersample.values)

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test_undersample, y_pred_undersample)
np.set_printoptions(precision=2)

recall = cnf_matrix[1, 1] / (cnf_matrix[1, 0] + cnf_matrix[1, 1])
print("Recall metric in the testing dataset: ", recall)

# Plot non-normalized confusion matrix
class_names = [0, 1]
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, title='Confusion matrix')
plt.show()
```

```
Recall metric in the testing dataset:  0.9251700680272109
```

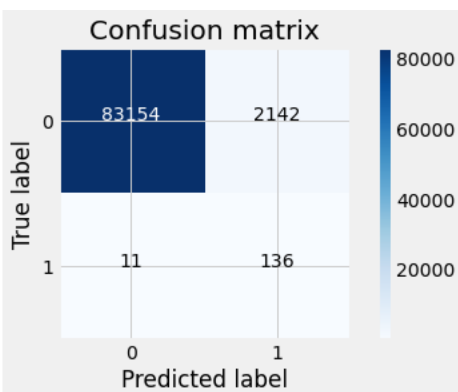


Our model predicts a 93.2% recall on the undersampled test set. We'll try it now with our whole data to see if it still works.

In [23]:

```
def plot_confusion_matrix(cm, classes,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=0)
    plt.yticks(tick_marks, classes)

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Use this C_parameter to build the final model with the whole training dataset and predict the classes in the test
# dataset
lr = LogisticRegression(C=best_c, penalty='l2')
lr.fit(X_train_undersample, y_train_undersample.values.ravel())
y_pred = lr.predict(X_test.values)

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, y_pred)
np.set_printoptions(precision=2)

recall = cnf_matrix[1, 1] / (cnf_matrix[1, 0] + cnf_matrix[1, 1])
print("Recall metric in the testing dataset: ", recall)

# Plot non-normalized confusion matrix
class_names = [0, 1]
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, title='Confusion matrix')
plt.show()

```

```
Recall metric in the testing dataset:  0.9251700680272109
```



Still almost identical! That's great.

We'll check it again with our skewed data using Logistic Regression

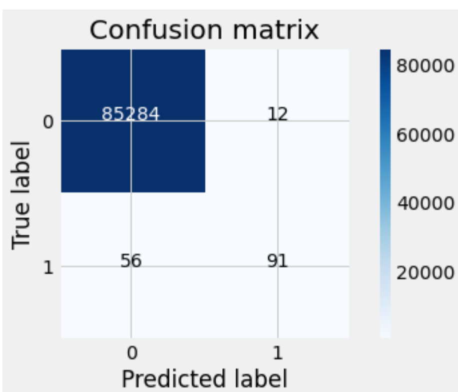
In [24]:
```
best_c = print_kfold_scores(X_train,y_train)
```

```
C parameter:  0.01
Recall:  0.5373134328358209
Recall:  0.6164383561643836
Recall:  0.6666666666666666
Recall:  0.6
Recall:  0.5
C parameter:  0.1
Recall:  0.5522388059701493
Recall:  0.6164383561643836
Recall:  0.7166666666666667
Recall:  0.6153846153846154
Recall:  0.5625
C parameter:  1
Recall:  0.5522388059701493
Recall:  0.6164383561643836
Recall:  0.7333333333333333
Recall:  0.6153846153846154
Recall:  0.575
C parameter:  10
Recall:  0.5522388059701493
Recall:  0.6164383561643836
Recall:  0.7333333333333333
Recall:  0.6153846153846154
Recall:  0.575
C parameter:  100
Recall:  0.5522388059701493
Recall:  0.6164383561643836
Recall:  0.7333333333333333
Recall:  0.6153846153846154
Recall:  0.575
Best model to choose from cross validation is with C parameter = 1
```

In [25]:

```
def plot_confusion_matrix(cm, classes, title):
    """
    Plot confusion matrix using matplotlib.
    """
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=0)
    plt.yticks(tick_marks, classes)

    thresh = cm.max() / 2.
    for i, j in np.ndindex(cm.shape):
        plt.text(j, i, format(cm[i, j], 'd'),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Fit logistic regression model using "best_c" as the regularization strength
lr = LogisticRegression(C=best_c, penalty='l2')
lr.fit(X_train, y_train.values.ravel())

# Make predictions on the test data
y_pred = lr.predict(X_test.values)

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, y_pred)
np.set_printoptions(precision=2)

# Calculate recall metric
recall = cnf_matrix[1, 1] / (cnf_matrix[1, 0] + cnf_matrix[1, 1])
print("Recall metric in the testing dataset: ", recall)

# Plot non-normalized confusion matrix
class_names = [0, 1]
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, title='Confusion matrix')
plt.show()
```

```
Recall metric in the testing dataset:  0.6190476190476191
```



Random Forest

In [31]:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier

# Create a random forest classifier
clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the classifier to the training data
clf.fit(X_train_undersample, y_train_undersample)

# Predict the target variable for the test data
y_pred = clf.predict(X_test)

# Convert the predicted target variable into a Pandas Series
y_pred = pd.Series(y_pred)

# Convert the test target variable into a Pandas Series
y_test = pd.Series(y_test.values.ravel())

# Compare the two Series element-wise
accuracy = np.mean(y_pred == y_test)
print('Accuracy:', accuracy)

```

```
Accuracy: 0.9746146553842913
```

This is so much better.

In [34]:

```
import xgboost as xgb

# Create an XGBoost model
clf = xgb.XGBClassifier(random_state=42)

# Fit the model to the training data
clf.fit(X_train, y_train)

# Predict the target variable for the test data
y_pred = clf.predict(X_test)

# Calculate the accuracy of the model
accuracy = clf.score(X_test, y_test)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.9995318516437859
```

And This is even better!

Conclusion:

Our undersampled dataset performed well in our Logistic Regression models however, it will likely missclassify nonfraudulent transactions as fraudulent which will make our customers unhappy. Our Random Forest model did so much better. But the best model was our XGBoost. It seems to do much better with skewed data.

In [ ]:

```

```