DBSCAN Unsupervised

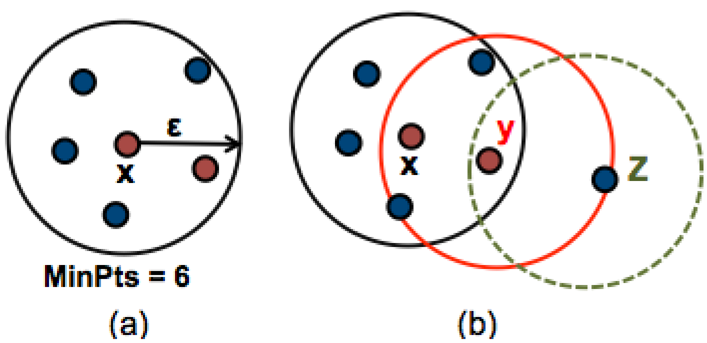
DBSCAN - Density-Based Spatial Clustering of Applications with Noise. Finds core samples of high density and expands clusters from them. Good for data which contains clusters of similar density.

The goal is to identify dense regions, which can be measured by the number of objects close to a given point.

Two important parameters are required for DBSCAN: epsilon ("eps") and minimum points ("MinPts"). The parameter eps defines the radius of neighborhood around a point x. It's called called the ϵ-neighborhood of x. The parameter MinPts is the minimum number of neighbors within "eps" radius.

Any point x in the dataset, with a neighbor count greater than or equal to MinPts, is marked as a core point. We say that x is border point, if the number of its neighbors is less than MinPts, but it belongs to the ϵ-neighborhood of some core point z. Finally, if a point is neither a core nor a border point, then it is called a noise point or an outlier.

The figure below shows the different types of points (core, border and outlier points) using MinPts = 6. Here x is a core point because neighboursϵ(x)=6, y is a border point because neighboursϵ(y)<MinPts, but it belongs to the ϵ-neighborhood of the core point x. Finally, z is a noise point.



We define 3 terms, required for understanding the DBSCAN algorithm:

Direct density reachable: A point “A” is directly density reachable from another point “B” if: i) “A” is in the ϵ-neighborhood of “B” and ii) “B” is a core point. Density reachable: A point “A” is density reachable from “B” if there are a set of core points leading from “B” to “A. Density connected: Two points “A” and “B” are density connected if there are a core point “C”, such that both “A” and “B” are density reachable from “C”. A density-based cluster is defined as a group of density connected points. The algorithm of density-based clustering (DBSCAN) works as follow:

The algorithm of density-based clustering works as follow:

For each point xi, compute the distance between xi and the other points. Finds all neighbor points within distance eps of the starting point (xi). Each point, with a neighbor count greater than or equal to MinPts, is marked as core point or visited. For each core point, if it’s not already assigned to a cluster, create a new cluster. Find recursively all its density connected points and assign them to the same cluster as the core point. Iterate through the remaining unvisited points in the dataset. Those points that do not belong to any cluster are treated as outliers or noise.

Parameter estimation for DBSCAN

To choose good parameters we need to understand how they are used and have at least a basic previous knowledge about the data set that will be used.

eps: if the eps value chosen is too small, a large part of the data will not be clustered. It will be considered outliers because don’t satisfy the number of points to create a dense region. On the other hand, if the value that was chosen is too high, clusters will merge and the majority of objects will be in the same cluster. The eps should be chosen based on the distance of the dataset (we can use a k-distance graph to find it), but in general small eps values are preferable. minPoints: As a general rule, a minimum minPoints can be derived from a number of dimensions (D) in the data set, as minPoints ≥ D + 1. Larger values are usually better for data sets with noise and will form more significant clusters. The minimum value for the minPoints must be 3, but the larger the data set, the larger the minPoints value that should be chosen.

In [1]:
```
from sklearn import datasets
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn import cluster
from sklearn.cluster import KMeans
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
from sklearn import svm, neighbors
from sklearn.neighbors import NearestNeighbors
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import recall_score
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
from sklearn.metrics import accuracy_score
from sklearn import metrics

import pandas as pd
import numpy as np
import itertools
import matplotlib.pyplot as plt
import datetime

%matplotlib inline
```

In [2]:
```
train_split = 0.80
nrows = 250_000
path = 'c:/users/ugy1/abs/'
df=pd.read_csv(path+'datasets/processed_abs_loan_'+str(nrows)+'.csv',
               #usecols=use_list,
               #sep='\t',
               #compression=bz2,
               nrows=nrows,
               low_memory=False,
              index_col=0,
               parse_dates=True
              )
df.shape
```

Out[2]: (237024, 58)

In [3]:
```
column_list=df.columns.tolist()
```

In [4]:
```
df.head()
```

Out[4]:

	originalloanamount	originalloanterm	originalinterestratepercentage	graceperiodnuml
0	66711.84	60	3.29	1
1	16258.45	60	0.90	0
2	31930.41	72	2.90	1
3	26065.02	65	0.90	0
4	42091.00	72	3.90	0

5 rows × 58 columns

In [5]:
```
# prepare label for scikit-learn
Y=df.label.values
Y.shape
```

Out[5]: (237024,)

In [6]:
```
# prepare input data for scikit-learn
input=df.values
input.shape
```

Out[6]: (237024, 58)

In [7]:
```
# calculate train/test split

len_train = int(len(input)*train_split)
print(len_train)
```

189619

In [8]:
```
# apply train/test split to labels
y_train = Y[0:len_train]
y_test = Y[len_train:]
x_train = input[0:len_train]
x_test = input[len_train:]
x_train.shape
```

Out[8]: (189619, 58)

In [9]:
```
export_x_test = pd.DataFrame(data=x_test)
```

In [10]:
```
export_x_test.columns=column_list
export_x_test.rename(columns={'label':'True Label'}, inplace=True)
export_x_test.head()
```

Out[10]:

	originalloanamount	originalloanterm	originalinterestratepercentage	graceperiodnuml
0	36863.24	72.0	1.00	1.0
1	23811.32	60.0	1.90	0.0
2	30669.00	48.0	1.00	1.0
3	54083.21	72.0	1.00	0.0
4	31557.75	72.0	3.89	1.0

5 rows × 58 columns

In [11]:
```
#from sklearn.preprocessing import MinMaxScaler
# from sklearn.preprocessing import minmax_scale
# from sklearn.preprocessing import MaxAbsScaler
from sklearn.preprocessing import StandardScaler
# from sklearn.preprocessing import RobustScaler
# from sklearn.preprocessing import Normalizer
# from sklearn.preprocessing import QuantileTransformer
# from sklearn.preprocessing import PowerTransformer
```

In [12]:
```
x_scaler=StandardScaler()
x_train = x_scaler.fit_transform(x_train)
x_test = x_scaler.fit_transform(x_test)
```

In [13]:
```
dbscan = cluster.DBSCAN(eps=0.3, algorithm='auto').fit(x_test)
```

In [14]:
```
#x_pred = x_test
```

In [15]:
```
prediction_dbscan = dbscan.labels_
```

In [16]:
```
np.unique(prediction_dbscan)
```

Out[16]:
```
array([-1,  0], dtype=int64)
```

In [17]:
```
s=np.absolute(np.array(prediction_dbscan))
np.bincount(s.reshape(1,s.size)[0])
```

Out[17]:
```
array([    5, 47400], dtype=int64)
```

In [18]:
```
n_clusters=len(np.bincount(s.reshape(1,s.size)[0]))
print('Number of Clusters: ', n_clusters)
```

```
Number of Clusters:  2
```

In [19]:
```
export_x_test['Predicted_Label']=prediction_dbscan
```

In [20]:
```
export_x_test.Predicted_Label.replace(0,1, inplace=True)
export_x_test.Predicted_Label.replace(-1,0, inplace=True)
```

In [21]:
```
export_x_test.head()
```

Out[21]:

	originalloanamount	originalloanterm	originalinterestratepercentage	graceperiodnuml
0	36863.24	72.0	1.00	1.0
1	23811.32	60.0	1.90	0.0
2	30669.00	48.0	1.00	1.0
3	54083.21	72.0	1.00	0.0
4	31557.75	72.0	3.89	1.0

5 rows × 59 columns

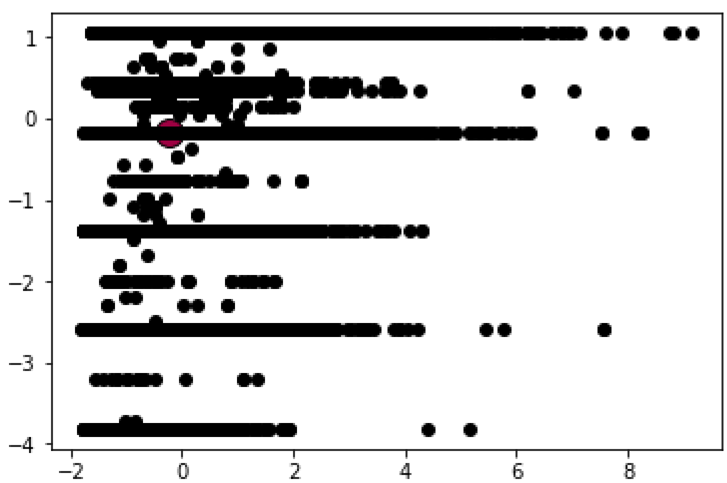
In [22]:
```
export_x_test.shape
```

Out[22]: (47405, 59)

In [23]:
```
export_x_test.to_csv(path+"prediction/dbscan/predicated_dbscan_abs_loans_"+str
(nrows)+".csv", chunksize=10000)
```

In [30]:
```
prediction_dbscan=export_x_test.Predicted_Label
```

In [37]:
```
labels= dbscan.labels_
core_samples_mask = np.zeros_like(dbscan.labels_, dtype=bool)
core_samples_mask[dbscan.core_sample_indices_] = True
# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = x_test[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)

    xy = x_test[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

#plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```



In [32]:
```
def plot_confusion_matrix(cm, title, classes=['Current', 'Non_Current'],
                          cmap=plt.cm.Blues, save=False, saveas="MyFigure.pn
g"):

    # print Confusion matrix with blue gradient colours

    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=90)
    plt.yticks(tick_marks, classes)

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

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

    if save:
        plt.savefig(saveas, dpi=100)
```

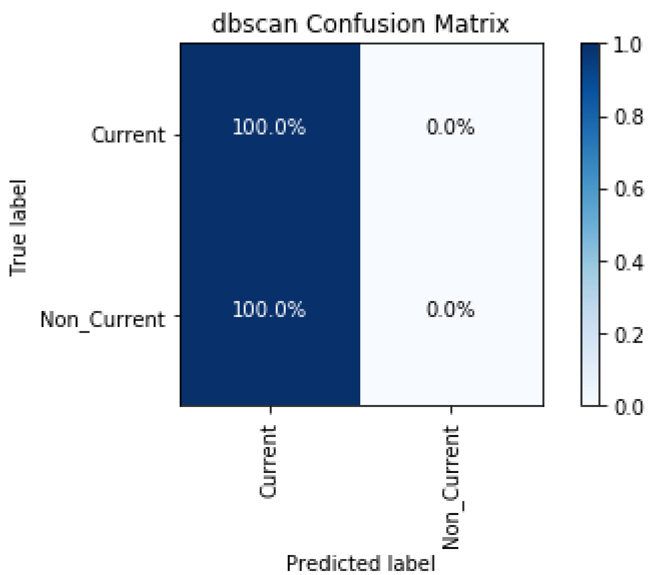
In [33]:
```
def plot_gridsearch_cv(results, estimator, x_min, x_max, y_min, y_max,save=F
alse, saveas="MyFigure.png"):

    # print GridSearch cross-validation for parameters

    plt.figure(figsize=(10,8))
    plt.title("GridSearchCV for "+estimator, fontsize=24)

    plt.xlabel(estimator)
    plt.ylabel("Score")
    plt.grid()

    ax = plt.axes()
    ax.set_xlim(x_min, x_max)
    ax.set_ylim(y_min, y_max)

    pad = 0.005
    X_axis = np.array(results["param_"+estimator].data, dtype=float)

    for scorer, color in zip(sorted(scoring), ['b', 'k']):
        for sample, style in (('train', '--'), ('test', '-')):
            sample_score_mean = results['mean_%s_%s' % (sample, scorer)]
            sample_score_std = results['std_%s_%s' % (sample, scorer)]
            ax.fill_between(X_axis, sample_score_mean - sample_score_std,
                            sample_score_mean + sample_score_std,
                            alpha=0.1 if sample == 'test' else 0, color=color)
            ax.plot(X_axis, sample_score_mean, style, color=color,
                alpha=1 if sample == 'test' else 0.7,
                label="%s (%s)" % (scorer, sample))

        best_index = np.nonzero(results['rank_test_%s' % scorer] == 1)[0][0]
        best_score = results['mean_test_%s' % scorer][best_index]

        # Plot a dotted vertical line at the best score for that scorer mark
ed by x
        ax.plot([X_axis[best_index], ] * 2, [0, best_score],
            linestyle='-.', color=color, marker='x', markeredgewidth=3, ms=8
)

        # Annotate the best score for that scorer
        ax.annotate("%0.2f" % best_score,
                (X_axis[best_index], best_score+pad))

    plt.legend(loc="best")
    plt.grid('off')
    plt.tight_layout()
    if save:
        plt.savefig(saveas, dpi=100)

    plt.show()
```

In [34]:
```
np.unique(prediction_dbscan)
```

Out[34]:
```
array([0, 1], dtype=int64)
```

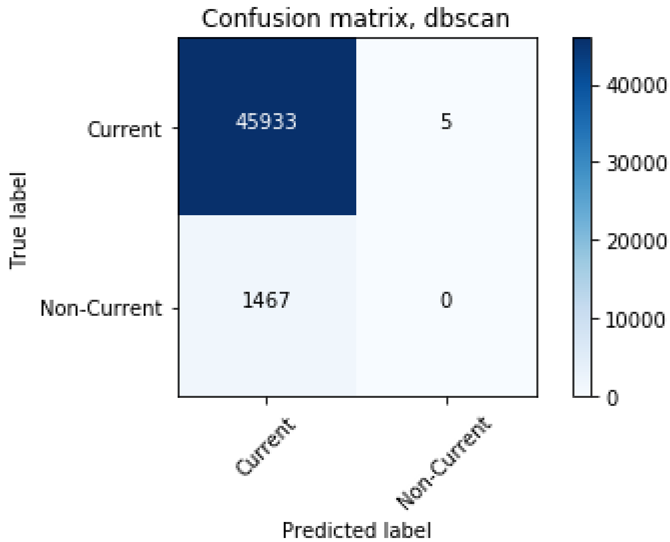
In [35]:
```
print(classification_report(y_test, prediction_dbscan, target_names=['Current'
, 'Non_Current']))
print ("AUC: ", "{:.1%}".format(roc_auc_score(y_test, prediction_dbscan)))
cm = confusion_matrix(y_test, prediction_dbscan)
plot_confusion_matrix(cm, title="dbscan Confusion Matrix",save=True,
                      saveas='prediction/dbscan/cm'+str(' dbscan Accuracy-')+s
tr(nrows)+'.jpg')
```

```
             precision    recall  f1-score   support

    Current       0.97      1.00      0.98     45938
Non_Current       0.00      0.00      0.00      1467

avg / total       0.94      0.97      0.95     47405

AUC:  50.0%
```



In [36]:
```
class_names = ['Current', 'Non-Current']

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

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

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

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


print('ROC_AUC_SCORE ; ', roc_auc_score(y_test, prediction_dbscan))
# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, prediction_dbscan)
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, title= 'Confusion mat
rix, dbscan')
plt.savefig('prediction/dbscan/cm'+str(' dbscan Complete-')+str(nrows)+'.jp
g')
plt.show()
```

```
ROC_AUC_SCORE ;  0.499945578824
Confusion matrix, without normalization
[[45933     5]
 [ 1467     0]]
```



In [ ]: