Data Mining and Machine Learning
APS Failure at Scania Trucks Data Set

Javier Rico (jvirico@gmail.com)

4 Dic 2019

PROBLEM DEFINITION

The goal is to minimize maintenance costs of the air pressure system (APS) of Scania trucks. Therefore, failures should be predicted before they occur. Falsely predicting a failure has a cost of 100, missing a failure has a cost of 3500. This leads to the need to cost minimization.

IMPLEMENTATION AND PROCESS

Python 3, Jupyter Notebook and Scikit-learn Machine Learning library have been used to approach this binary classification challenge.

The implementation has a HYPERPARAMETERs section at the beginning to turn on and off almost all decisions considered in the process.

Hyperparameters

```
#### Exploratory Analysis
Analysis = True

#### Column Removal
Remove_Cols_with_many_bad_data = True
NaN_Zero_Threshold = 0.8

#### Outliers
RemoveOutliers = False     #Using IQR
Times_IQR = 1.5            #Normal factor (1.5). Kills mir

## Outliers by class
RemoveOutliers_PositiveClass = False
RemoveOutliers_NegativeClass = False  #It removes >90% of

#### NaN and Zero replacement
NaN_Zero_Replacement = True
NaN_Zero_Replacement_Mode = 'general' #general or perClas
NaN_Zero_Replacement_Operation = 'median' #median or mear
NaN_Zero_Replacement_Sort = True

#### Data Projections
ProjectBins = True
ProjectBin_Operation = "mean"  #sum or mean
RemoveSubBins = True

#### Rounding values
Rounding = False     #Does not change the results and mak
RoudingDecimals = 2

#### Feature Selection
```

IMPORTANT: Note that not all transformations and techniques explained in this document have been applied to the final solution, but many will be explained to expose the process followed. Each solution uploaded to Kaggle competition has the hyperparameters used informed in its description, random seeds, when used, are also informed there.

The process followed has been an iteration of different combinations of the steps listed below. Those steps that resulted in improvements have been consistently used, these are explained in the *Conclusions* and *Notes* below each section:

- **Data Analysis/Discovery**
    - Data types of features
    - Feature Class imbalance
    - Dataset Statistics
    - Outliers
    - NaN and zero analysis

- **Data Preparation/Cleansing**
    - Feature Class to 0-1 values
    - Removing Outliers using Interquartile Ranges (IQR)
    - NaNs and Zeros
    - Removing Features with number of NaNs and Zeros above Threshold
    - Replacing NaNs and Zeros
    - Projections
        - Projecting Bins into 7 new features
    - Feature Selection
        - Removing SubBins
        - High Intercorrelated Features
        - Top correlated features with Class
    - Sampling
        - Up-sampling minority class

- **Model selection**
    - Random Forest
    - Support Vector Machine
    - Bagging

- **Parameter tuning**
    - GridSearch
    - MakeScorer
    - Hyperparameters section

- **Training the model**
    - Favor True class (post processing)

- **Evaluating the model**

- **Prediction**

- **Summary of methods**

DATA ANALYSIS / DISCOVERY

The data consists of a training set with 60000 rows, of which 1000 belong to the positive class, and 171 columns, of which one is the Class column. All the attributes are numeric, except Class that is a Boolean.

- Data types of features
- Dataset Statistics
- Feature Class imbalance
- Outliers
- NaN and Zero analysis

Data types of features

```
#Data types
if(Analysis == True):
    print(train.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 57000 entries, 0 to 56999
Columns: 171 entries, class to eg_000
dtypes: bool(1), float64(169), int64(1)
memory usage: 74.0 MB
None
```

Conclusions:

All features are numeric values.

Dataset Statistics

```
#Statistics
if(Analysis == True):
    stats = train.describe()
    print(stats.round().T)
```

	count	mean	std	min	25%	50%	75%	\
aa_000	57000.0	61074.0	232734.0	0.0	872.0	30840.0	48942.0	
ab_000	13046.0	1.0	3.0	0.0	0.0	0.0	0.0	
ac_000	53781.0	354187065.0	793240902.0	0.0	16.0	152.0	966.0	
ad_000	42725.0	201369.0	41530171.0	0.0	24.0	128.0	432.0	
ae_000	54609.0	7.0	144.0	0.0	0.0	0.0	0.0	
af_000	54609.0	11.0	196.0	0.0	0.0	0.0	0.0	
ag_000	56328.0	224.0	20940.0	0.0	0.0	0.0	0.0	
ag_001	56328.0	1106.0	35294.0	0.0	0.0	0.0	0.0	
ag_002	56328.0	9526.0	161482.0	0.0	0.0	0.0	0.0	
ag_003	56328.0	92700.0	759330.0	0.0	0.0	0.0	0.0	
ag_004	56328.0	445606.0	2217632.0	0.0	310.0	3706.0	50974.0	
ag_005	56328.0	1124752.0	3226828.0	0.0	13986.0	178836.0	923398.0	
ag_006	56328.0	1665498.0	3895117.0	0.0	10804.0	937191.0	1889096.0	
ag_007	56328.0	499825.0	1399681.0	0.0	0.0	118949.0	590420.0	
ag_008	56328.0	36146.0	238897.0	0.0	0.0	1812.0	26796.0	
ag_009	56328.0	5541.0	181772.0	0.0	0.0	0.0	370.0	
ah_000	56365.0	1838969.0	4275079.0	0.0	30300.0	1005844.0	1605868.0	
ai_000	56388.0	9932.0	178279.0	0.0	0.0	0.0	0.0	

Conclusions:

There are features with up to 81% of missing values (0's and NaNs). Almost all features have sparse NaNs and 0s.

Feature Class imbalance

```
#Checking class feature
train["class"].value_counts()
```

```
False    55968
True      1032
Name: class, dtype: int64
```

Conclusions:

The 'class' feature, also the target attribute for our model, is highly imbalanced.

Some approaches that have been tried:

- As it is.
- Up-sampling using **SMOTE** or **ADASYN** to obtain same samples of each class (balanced 50/50).
- Up-sampling using SMOTE or ADASYN using different ratios for True/False samples (example: sampling_strategy = 0.8).

These strategies directly affect the final score since False Positives have different penalty than False Negatives. To deal with this the following approaches have been tried:

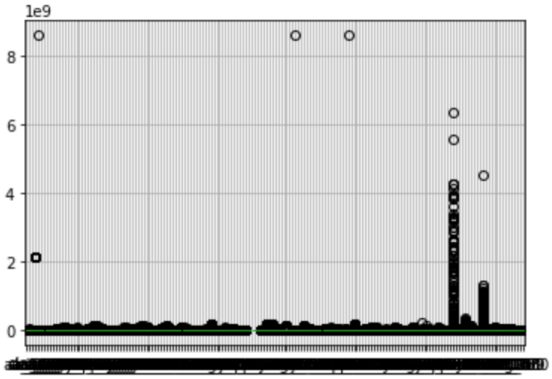
- Scikit-learn **Make Scorer** (sklearn.metrics.make_scorer) have been used in conjunction with GridSearch to favor models with better scores. For this, a custom GetScore function has been coded to fit our case.

```
def getCost(y_test,y_pred):
    tn,fp,fn,tp = confusion_matrix(y_test,y_pred).ravel()
    return (fp*100 + fn*3500)
```

- When Random Forest has been used, different weights have been set up for True and False classes (example: class_weight={0:1,1:35}).

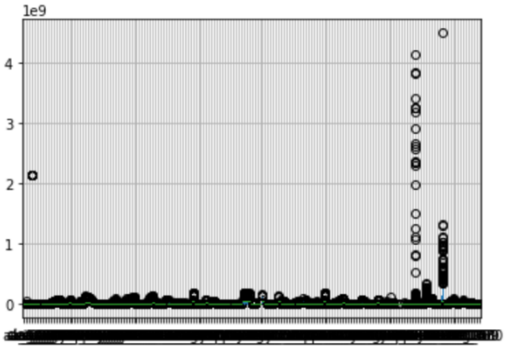
Outliers (all features)

```
#Outliers
if(Analysis == True):
    boxplot = train.boxplot()
#train.columns.values
```

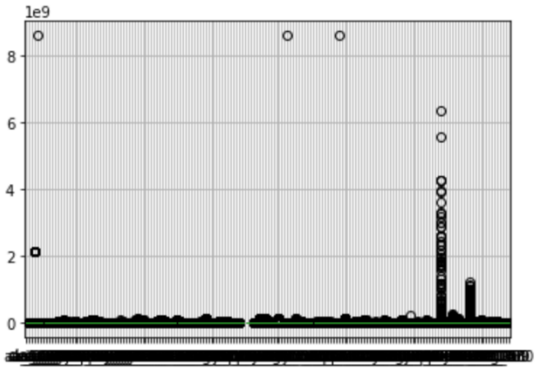


Outliers (Trues profile vs Falses profile)

```
if(Analysis == True):
    positiveData = PositiveData(train)
    negativeData = NegativeData(train)
    boxplot = positiveData.boxplot()
```




```
if(Analysis == True):
    boxplot = negativeData.boxplot()
```



Conclusions:

Many of the features contain outliers.

IQR - **Interquartile Range Rule** has been used to remove them:

- with 1.5 times the third quartile has been used.
- Different thresholds (greater than 1.5x 3rt quartile) have been tried.
- When removed Outliers using 1.5 IQR, True class has been removed when the dataset is not balanced (ex. up-sampled). IQR on True sub set and IQR on False sub set has been also tried, but this cannot be replicated in Test dataset so it has been discarded.
- Not removing outliers. In many cases removing Outliers did not lead to better results.

NaN and zero analysis

Almost all features have sparse NaNs and Zeros.

```
#### Column Removal
Remove_Cols_with_many_bad_data = True
NaN_Zero_Threshold = 0.8
```

```
def ColumnFiltering1_ZerosNaNs_byThreshold(data,threshold):

    columnsNames = list(data.columns.values)
    total_rows = train.shape[0]
    thr = total_rows * threshold

    for i in columnsNames:
        count = 0

        s = data[i]
        if (i!= 'class'):
            for j in s:
                if math.isnan(j) or j == 0.0:
                    count += 1
            #print(i + " " + str(count))
        if count >= thr:
            data.drop(i, axis = 1, inplace = True)
            #print ("Droping " + i)
```

```
if (Remove_Cols_with_many_bad_data == True):
    print (train.shape)
    print (test.shape)
    print("Remove_Cols_with_many_bad_data On")
    ColumnFiltering1_ZerosNaNs_byThreshold(train,NaN_Zero_Threshold)
    test = test[train.columns.values]

print (train.shape)
print (test.shape)
```

```
(57000, 171)
(19000, 171)
Remove_Cols_with_many_bad_data On
(57000, 129)
(19000, 129)
```

Conclusions:

There are features with up to 81% of missing values (0's and NaNs).

Removing columns with more than *Threshold* of missing values has been tried. Most of the times 0.8 has shown good results, as shown above, this setting gets rid of 42 features.

For the rest of NaNs and 0s, replacement (mean/median) has been performed. For NaNs an imputation algorithm has been used (*sklearn.preprocessing.**imputer***).

DATA PREPARATION / CLEANSING

Transformations used:

- Feature Class to 0-1 values
- Removing Outliers using Interquartile Ranges (IQR)
- NaNs and Zeros
- Removing Features with number of NaNs and Zeros above Threshold
- Replacing NaNs and Zeros
- Projections
    - Projecting Bins into 7 new features
- Feature Selection
    - Removing SubBins
    - High Intercorrelated Features
    - Top correlated features with Class
- Sampling
    - Up-sampling minority class

Feature Class to 0-1 values

```
ClassTo01(train)
train["class"] = pd.to_numeric(train["class"])
```

Note:

Applied always.

Removing Outliers using Interquartile Ranges (IQR)

Many of the features contain outliers.

IQR - Interquartile Range Rule has been used to remove outliers:

- with 1.5 times the third quartile has been used.
- Different thresholds (greater than 1.5x 3rt quartile) have been tried.
- When removed Outliers using 1.5 IQR, True class has been removed when the dataset is not balanced (ex. up-sampled). IQR on True sub set and IQR on False sub set has been also tried. This cannot be replicated in Test dataset so it has been discarded.
- Not removing outliers.

Note:

In many cases removing Outliers did not lead to better results.

NaNs and Zeros

There are features with up to 81% of missing values (0's and NaNs). And almost all features have sparse NaNs and Zeros.

(1) Removing columns with more than *Threshold* of missing values has been tried. Most of the times 0.8 has shown good results.

(2) For the rest of NaNs and 0s, replacement (mean/median) has been performed.

Note:

---

(1) Applied in the majority of trials with threshold = 0.8.
(2) Applied always, replacing with mean value most of the times. For this an imputation of missing values using *sklearn.preprocessing.**imputer*** has been performed.

```
def ReplaceNaNsForStats(data,strategy):
    columnsNames = list(data.columns.values)
    imp = Imputer(missing_values = "NaN", strategy = strategy, axis = 0, copy = False)
    d = imp.fit_transform(data)
    aux = pd.DataFrame(d, columns=columnsNames)
    aux.index = data.index
    return aux.copy()
```

Projections

Projecting Bins (histograms) into 7 new features:

- Substitution of the bins for 7 new features, either summarizing or averaging, showed an improvement. The seven new features are highly correlated with the original bins so removing them simplifies the dataset.

- When this is applied, we assume the loss of the distributions of the original bins as histograms of the original attributes.

```
#### Data Projections
ProjectBins = True
ProjectBin_Operation = "mean"  #sum or mean
RemoveSubBins = True
```

```
def NewColFromBin_v2(data,colName,op):
    sum = 0

    if colName + '_000' in data.columns: sum += data[colName + '_000']
    if colName + '_001' in data.columns: sum += data[colName + '_001']
    if colName + '_002' in data.columns: sum += data[colName + '_002']
    if colName + '_003' in data.columns: sum += data[colName + '_003']
    if colName + '_004' in data.columns: sum += data[colName + '_004']
    if colName + '_005' in data.columns: sum += data[colName + '_005']
    if colName + '_006' in data.columns: sum += data[colName + '_006']
    if colName + '_007' in data.columns: sum += data[colName + '_007']
    if colName + '_008' in data.columns: sum += data[colName + '_008']
    if colName + '_009' in data.columns: sum += data[colName + '_009']

    if (op == 'sum'): retorno = sum
    if (op == 'mean'): retorno = sum/10
```

```
if (ProjectBins == True):
    print("ProjectBins On")
    #### Projecting bins into 7 new Features that summarize/average the bin's values.
    train['ag'] = NewColFromBin_v2(train,'ag', ProjectBin_Operation)
    train['ay'] = NewColFromBin_v2(train,'ay', ProjectBin_Operation)
    train['az'] = NewColFromBin_v2(train,'az', ProjectBin_Operation)
    train['ba'] = NewColFromBin_v2(train,'ba', ProjectBin_Operation)
    train['cn'] = NewColFromBin_v2(train,'cn', ProjectBin_Operation)
    train['cs'] = NewColFromBin_v2(train,'cs', ProjectBin_Operation)
    train['ee'] = NewColFromBin_v2(train,'ee', ProjectBin_Operation)

    test['ag'] = NewColFromBin_v2(test,'ag', ProjectBin_Operation)
    test['ay'] = NewColFromBin_v2(test,'ay', ProjectBin_Operation)
    test['az'] = NewColFromBin_v2(test,'az', ProjectBin_Operation)
    test['ba'] = NewColFromBin_v2(test,'ba', ProjectBin_Operation)
    test['cn'] = NewColFromBin_v2(test,'cn', ProjectBin_Operation)
    test['cs'] = NewColFromBin_v2(test,'cs', ProjectBin_Operation)
    test['ee'] = NewColFromBin_v2(test,'ee', ProjectBin_Operation)
```

Note:

Bins projections on 7 new attributes and deletion of original histograms has been applied for most of the trials.

Feature Selection / Correlation / Ranking

Removing histograms:

Bins projections on 7 new attributes and deletion of original histograms has been applied for most of the trials.

High Intercorrelated Features

Feature correlation using Pearson's Correlation has been performed.
The data has a lot of features, because of that, is very difficult to visualize hierarchical graphs. We use a table instead.

```
cor = 0
if (RunCorrelationAnalysis == True):
    #print("RunCorrelationAnalysis On")
    #### Filter Method for Feature Correlation with Class Feature
    #Using Pearson Correlation
    #plt.figure(figsize=(12,10))
    cor = train.corr()
    #sns.heatmap(cor, annot=True, cmap=plt.cm.Reds)
    #plt.show()
cor
```

	class	aa_000	ac_000	ad_000	ag_003	ag_004	ag_005	ag_006	ag_007	ag_008	...
class	1.000000	0.375922	-0.052777	-0.000567	0.470332	0.421640	0.481352	0.384407	0.250419	0.113222	...
aa_000	0.375922	1.000000	-0.043109	-0.001051	0.400179	0.462255	0.528883	0.429416	0.301633	0.130118	...
ac_000	-0.052777	-0.043109	1.000000	-0.001804	-0.050309	-0.070566	-0.061549	-0.036364	-0.012175	-0.010095	...
ad_000	-0.000567	-0.001051	-0.001804	1.000000	-0.000506	-0.000833	-0.001381	-0.001635	-0.001329	-0.000579	...
ag_003	0.470332	0.400179	-0.050309	-0.000506	1.000000	0.849839	0.564479	0.140188	0.059008	0.088435	...
ag_004	0.421640	0.462255	-0.070566	-0.000833	0.849839	1.000000	0.787027	0.236591	0.105144	0.118696	...
ag_005	0.481352	0.528883	-0.061549	-0.001381	0.564479	0.787027	1.000000	0.651116	0.402629	0.213178	...
ag_006	0.384407	0.429416	-0.036364	-0.001635	0.140188	0.236591	0.651116	1.000000	0.851701	0.363803	...
ag_007	0.250419	0.301633	-0.012175	-0.001329	0.059008	0.105144	0.402629	0.851701	1.000000	0.634704	...
ag_008	0.113222	0.130118	-0.010095	-0.000579	0.088435	0.118696	0.213178	0.363803	0.634704	1.000000	...
ag_009	0.081812	0.048350	-0.007640	-0.000125	0.056470	0.053444	0.055777	0.072945	0.115576	0.355399	...
ah_000	0.525860	0.571282	-0.068221	-0.001696	0.437043	0.588085	0.767427	0.674313	0.522468	0.284399	...
ai_000	0.031937	0.037383	-0.009492	-0.000092	0.012544	0.012595	0.055907	0.144853	0.162049	0.122154	...

Top correlated features with Class

Feature ranking is made based on top feature correlations with Class attribute.

```
if (RunCorrelationAnalysis == True):
    #Correlation with output variable
    cor_target = abs(cor["class"])
    #Selecting highly correlated features
    relevant_features = cor_target[cor_target>0.4]
    print(relevant_features.sort_values(ascending=False))
```

```
class     1.000000
ci_000    0.570383
bb_000    0.545215
bv_000    0.543530
bu_000    0.543529
cq_000    0.543529
aq_000    0.533687
cc_000    0.530158
bj_000    0.528382
an_000    0.526455
ah_000    0.525860
bg_000    0.525225
bx_000    0.523317
ao_000    0.522661
by_000    0.521415
ap_000    0.520655
ba_004    0.513682
cc_005    0.509346
```

Several feature sub sets are selected based on top correlated ranking.

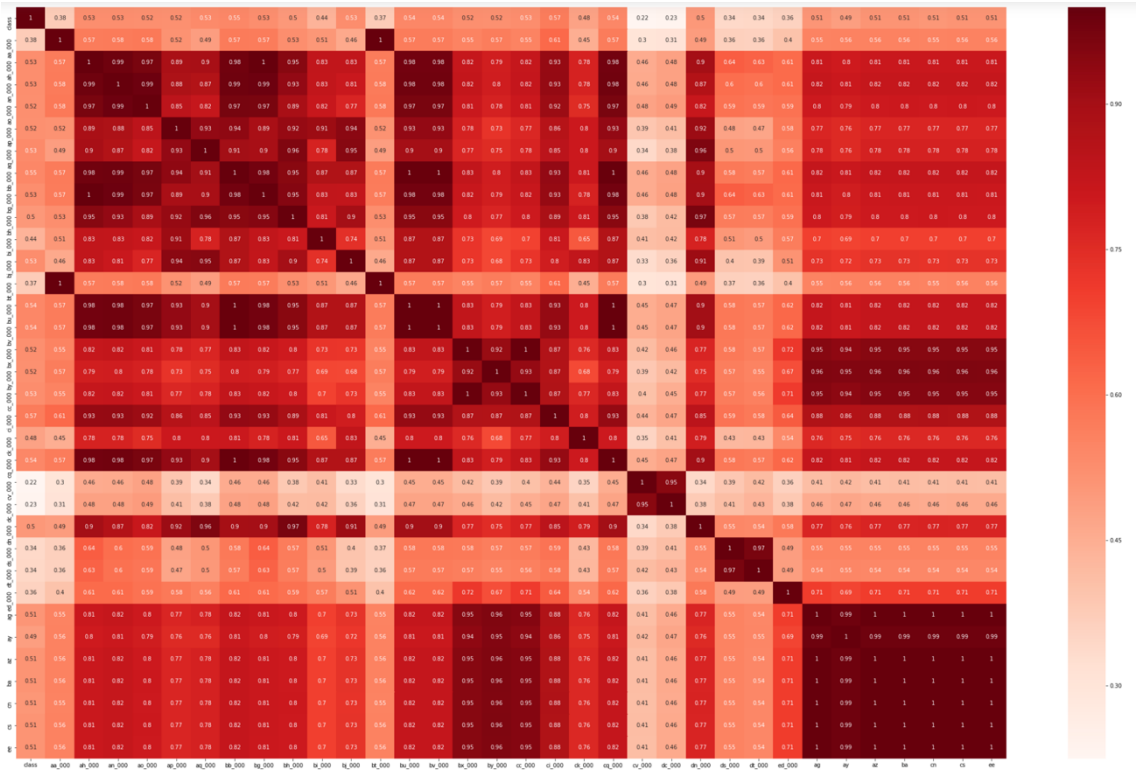
```
#### Feature Selection
RunCorrelationAnalysis = True   #Using Pearson Correlation
Corr_FeatureSelection = 2   #0, 1, 2, 3, 4
```

```
if (Corr_FeatureSelection == 1):
    print("Corr_FeatureSelection = 1")
    #Most correlated features with class (>0.4)
    train = train[['class','aa_000','ah_000','an_000',\
                           'ao_000','ap_000','aq_000',\
                           'bb_000','bg_000','bh_000',\
                           'bi_000','bj_000','bt_000',\
                           'bu_000','bv_000','bx_000',\
                           'by_000','cc_000','ci_000',\
                           'ck_000','cq_000','cv_000',\
                           'dc_000','dn_000','ds_000',\
                           'dt_000','ed_000','ag',\
                           'ay','az','ba',\
                           'cn','cs','ee']].copy()

    test = test[['class','aa_000','ah_000','an_000',\
                           'ao_000','ap_000','aq_000',\
                           'bb_000','bg_000','bh_000',\
                           'bi_000','bj_000','bt_000',\
                           'bu_000','bv_000','bx_000',\
                           'by_000','cc_000','ci_000',\
                           'ck_000','cq_000','cv_000',\
                           'dc_000','dn_000','ds_000',\
                           'dt_000','ed_000','ag',\
                           'ay','az','ba',\
                           'cn','cs','ee']].copy()


if (Corr_FeatureSelection == 2):
    print("Corr_FeatureSelection = 2")

    train = train[['class','bb_000','ck_000','dc_000','ed_000','ba']].copy()
    test = test[['class','bb_000','ck_000','dc_000','ed_000','ba']].copy()


if (Corr_FeatureSelection == 3):
    print("Corr_FeatureSelection = 3")

    train = train[['class','ck_000','dc_000','ed_000','ba']].copy()
    test = test[['class','ck_000','dc_000','ed_000','ba']].copy()
```

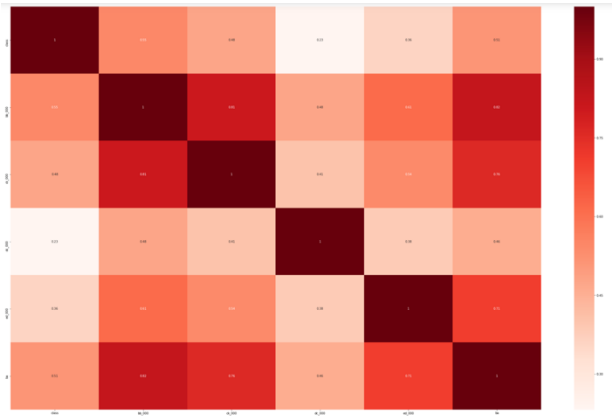
(After Feature Selection)

```
if (RunCorrelationAnalysis == True):
    #Using Pearson Correlation
    plt.figure(figsize=(40,25))
    cor2 = train.corr()
    sns.heatmap(cor2, annot=True, cmap=plt.cm.Reds)
    plt.show()
```

Corr_FeatureSelection = 1



Corr_FeatureSelection = 2



Note:

Best results have been achieved with Corr_FeatureSelection = 1.

Sampling

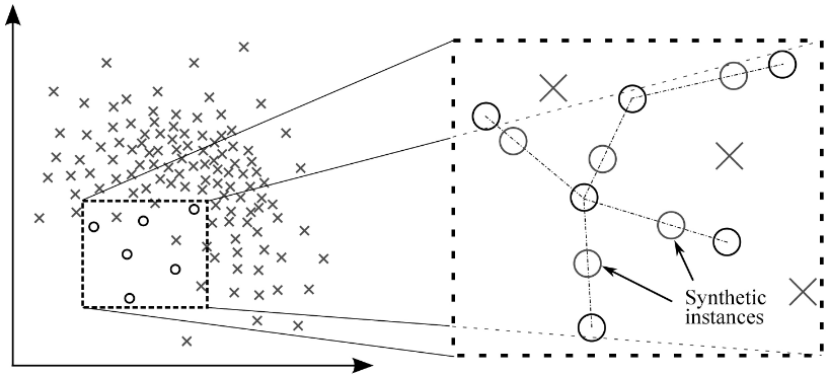
The 'class' feature, also the target attribute for our model, is highly imbalanced.

Some approaches that have been tried:

- As it is.
- Up-sampling using SMOTE or ADASYN to obtain same samples of each class (balanced 50/50).
- Up-sampling using SMOTE or ADASYN using different ratios for True/False samples (example: sampling_strategy = 0.8).

SMOTE and ADASYN:

SMOTE: Synthetic Minority Over Sampling Technique (SMOTE) algorithm applies KNN approach where it selects K nearest neighbors, joins them and creates the synthetic samples in the space. The algorithm takes the feature vectors and its nearest neighbors, computes the distance between these vectors. The difference is multiplied by random number between (0, 1) and it is added back to feature. SMOTE algorithm is a pioneer algorithm and many other algorithms are derived from SMOTE.



ADASYN: ADAptive SYNthetic (ADASYN) is based on the idea of adaptively generating minority data samples according to their distributions using K nearest neighbor. The algorithm adaptively updates the distribution and there are no assumptions made for the underlying distribution of the data. The algorithm uses Euclidean distance for KNN Algorithm. The key difference between ADASYN and SMOTE is that the former uses a density distribution, as a criterion to automatically decide the number of synthetic samples that must be generated for each minority sample by adaptively changing the weights of the different minority samples to compensate for the skewed distributions. The latter generates the same number of synthetic samples for each original minority sample.

```
#### Resampling
Upsampling = True                 #Previous to Bagging
UpsamplingAproach = 'ADASYN'     #ADASYN/SMOTE
_sampling_strategy = 0.8                  #'minority'/double
```

```
if (Upsampling == True):
    print("Upsampling On")

    train_Labels = train["class"]
    columnsNames = list(train.columns.values)

    # Oversampling minority class points, here minority class points are 1s
    if (UpsamplingAproach == "ADASYN"):
        print("ADASYN")
        ada = ADASYN(sampling_strategy = _sampling_strategy)
        train_balanced, train_3_3_balanced_Labels = ada.fit_sample(train, train_Labels)
    if(UpsamplingAproach == "SMOTE"):
        print("SMOTE")
        sm = SMOTE(sampling_strategy = _sampling_strategy)
        train_balanced, train_3_3_balanced_Labels = sm.fit_sample(train, train_Labels)

    train = pd.DataFrame(train_balanced, columns=columnsNames)

train["class"].value_counts()
```

Notes:

ADASYN has shown better results than SMOTE. It has been used most of the time.

Resampling strategies directly affect the final score since False Positives have different penalty than False Negatives. To deal with this the following approaches have been tried:

- Scikit-learn Make Scorer (sklearn.metrics.make_scorer) have been used in conjunction with GridSearch to favor models with better scores. For this, a custom GetScore function has been coded to fit our case.

```
def getCost(y_test,y_pred):
    tn,fp,fn,tp = confusion_matrix(y_test,y_pred).ravel()
    return (fp*100 + fn*3500)
```

- When Random Forest has been used, different weights have been set up for True and False classes (example: class_weight={0:1,1:35}).

MODEL SELECTION
Model selection, ensemble learning.

Two main algorithms have been considered, Support Vector Machine and Random Forest. GridSearch has been used to try different configurations on both.

Bagging has been also used as ensemble learning technique, usually with 10 estimators.

```
#### Model to Use
Multimode = False   # Not implemented
ModelToUse = 'RF'  # RF,SVM

#### Random Forest Parameters (ONLY WHEN GRIDSEARCH False)
_max_depth = 5        #None (default)
_n_estimators = 100
_bootstrap = True
_class_weight ='balanced'
_SEED = 1
_n_jobs = -1

#### Support Vector Machine (ONLY WHEN GRIDSEARCH False)
_C=10.0
_kernel='sigmoid'       #'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'
_degree=3               #3(default)
_gamma='scale'
_coef0=0.0
_shrinking=True
_probability=True
_tol=0.001
_cache_size=200
_class_weight=None
_verbose=True
_max_iter=-1
_decision_function_shape='ovr'
_random_state=None
```

Support Vector Machine

```
## SVM
SVM_Kernels = ['rbf','linear']    #['linear', 'rbf','sigmoid','poly']
SVM_Cs = [0.1, 1, 10]
SVM_gammas = [0.1, 1]
ParamGrid_SVM = {'kernel': SVM_Kernels, 'C': SVM_Cs, 'gamma': SVM_gammas, 'verbose':[2]}
```

Notes:

SVM has shown to be much more computationally expensive compared with the rest of techniques used. In particular, GridSearch + SVM + Bagging has been almost impossible to achieve with a personal computer due to the time needed to train it.

Random Forest

```
## RF
ParamGrid_RF = {'max_depth': [4,5,6,10,20], 'n_estimators':[100], \
                'class_weight':[{0:1,1:35},{1:35,0:1},'balanced'], \
                'verbose':[1], 'bootstrap': [True]};    # 'class_weight':[{0:1,1:35},{1:35,0:1}]
```

Notes:

Since False Positives have different penalty than False Negatives, when Random Forest has been used, different weights have been set up for True and False classes (example: class_weight={0:1,1:35}).

Pruning on Random Forest using *max_depth* has been performed to avoid overfitting. First trials achieved 0.99 accuracy and bad results on Kaggle competition. Once max_depth was properly set overall results were achieved. Pruning has been observed to be sensible to feature selection and resampling, requiring adaptation after such changes.

Bagging

```
#### Bagging
Bagging = True
# BC
_n_bootstraps = 3
_n_estimators = 10
_max_samples = 1.0
_max_features = 1.0
_bootstrap = True
_bootstrap_features = False
_bootstrap = True
_bootstrap_features = False
_oob_score = False
_warm_start = False
_n_jobs = 4
_random_state = None
_verbose = 3
```

Notes:

Bagging has shown small but consistent improvements in many configurations. Decreasing the number of estimators below 10 has shown bad results.

PARAMETER TUNING

GridSearch

GridSearch has been used to try different configurations on both, RF and SVM.

```
#### GridSearch
GridSearch = True
_cv = 5
_verbose = 3
## RF
ParamGrid_RF = {'max_depth': [4,5,6,10,20], 'n_estimators':[100], \
                'class_weight':[{0:1,1:35},{1:35,0:1},'balanced'], \
                'verbose':[1], 'bootstrap': [True]};    # 'class_weight':[{0:1,1:35},{1:35,0:1}]
## SVM
SVM_Kernels = ['rbf','linear']    #['linear', 'rbf','sigmoid','poly']
SVM_Cs = [0.1, 1, 10]
SVM_gammas = [0.1, 1]
ParamGrid_SVM = {'kernel': SVM_Kernels, 'C': SVM_Cs, 'gamma': SVM_gammas, 'verbose':[2]}
```

Notes:

Scikit-learn Make Scorer (sklearn.metrics.make_scorer) has been used in conjunction with GridSearch to favor models with better scores. For this, a custom GetScore function has been coded to fit our case.

```
def getCost(y_test,y_pred):
    tn,fp,fn,tp = confusion_matrix(y_test,y_pred).ravel()
    return (fp*100 + fn*3500)
```

Hyperparameters section

Main parameters of the implementation are located at the beginning of the Jupyter Notebook to facilitate an iterative training using different strategies.

Hyperparameters

```
#### Exploratory Analysis
Analysis = True

#### Column Removal
Remove_Cols_with_many_bad_data = True
NaN_Zero_Threshold = 0.8

#### Outliers
RemoveOutliers = False    #Using IQR
Times_IQR = 1.5           #Normal factor (1.5). Kills minority class if class included!!

## Outliers by class
RemoveOutliers_PositiveClass = False
RemoveOutliers_NegativeClass = False  #It removes >90% of the class

#### NaN and Zero replacement
NaN_Zero_Replacement = True
NaN_Zero_Replacement_Mode = 'general' #general or perClass
NaN_Zero_Replacement_Operation = 'median' #median or mean
NaN_Zero_Replacement_Sort = True

#### Data Projections
ProjectBins = True
ProjectBin_Operation = "mean"  #sum or mean
RemoveSubBins = True

#### Rounding values
Rounding = False     #Does not change the results and makes the process faster
RoudingDecimals = 2

#### Feature Selection
RunCorrelationAnalysis = True   #Using Pearson Correlation
Corr_FeatureSelection = 2   #0, 1, 2, 3, 4
```

```
#### Resampling
Upsampling = True                #Previous to Bagging
UpsamplingAproach = 'ADASYN'    #ADASYN/SMOTE
_sampling_strategy = 0.8                #'minority'/double

#### Train-Test split
EnableSplitting = True    #If False, no RF score and CM can be calculated
_test_size = 0.1

#### GridSearch
GridSearch = True
_cv = 5
_verbose = 3
## RF
ParamGrid_RF = {'max_depth': [4,5,6,10,20], 'n_estimators':[100], \
                'class_weight':[{0:1,1:35},{1:35,0:1},'balanced'], \
                'verbose':[1], 'bootstrap': [True]};    # 'class_weight':[{0:1,1:35},{1:35,0:1}]
## SVM
SVM_Kernels = ['rbf','linear']    #['linear', 'rbf','sigmoid','poly']
SVM_Cs = [0.1, 1, 10]
SVM_gammas = [0.1, 1]
ParamGrid_SVM = {'kernel': SVM_Kernels, 'C': SVM_Cs, 'gamma': SVM_gammas, 'verbose':[2]}

#### Model to Use
Multimode = False   # Not implemented
ModelToUse = 'RF'  # RF,SVM

#### Random Forest Parameters (ONLY WHEN GRIDSEARCH False)
_max_depth = 5        #None (default)
_n_estimators = 100
_bootstrap = True
_class_weight ='balanced'
_SEED = 1
_n_jobs = -1

#### Support Vector Machine (ONLY WHEN GRIDSEARCH False)
_C=10.0
_kernel='sigmoid'       #'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'
_degree=3               #3(default)
_gamma='scale'
_coef0=0.0
_shrinking=True
_probability=True
_tol=0.001
_cache_size=200
_class_weight=None
_verbose=True
_max_iter=-1
_decision_function_shape='ovr'
_random_state=None

#### Bagging
Bagging = True
# BC
_n_bootstraps = 3
_n_estimators = 10
_max_samples = 1.0
_max_features = 1.0
_bootstrap = True
_bootstrap_features = False
_bootstrap = True
_bootstrap_features = False
_oob_score = False
_warm_start = False
_n_jobs = 4
_random_state = None
_verbose = 3

#### Favor True Class (after prediction), not needed if SearchGreed (with scorer) True
FavorTrueClass = False
FavorThreshold = 0.45    #All False predictions between [0.5,FavorThreshold) are switched to True

SaveToFile = True
FileName = "prediction_results_v6_MAC.csv"

#### Use existing model
UseExistingModel = False
modelFileName = "model_v6_MAC.joblib"
if (UseExistingModel == True):
    print("Using model from disk!!!!!!")
```

TRAINING THE MODEL

Two options before and after finally training the model have been used to try to improve the result.

(1) Change the train-test dataset split. Usually preformed with 80/20 or 90/10 when full up-sampling.

```
#### Train-Test split
EnableSplitting = True    #If False, no RF score and CM can be calculated
_test_size = 0.2
```

(2) 'Favor True class', after training the model.

```
#### Favor True Class (after prediction), not needed if SearchGreed (with scorer) True
FavorTrueClass = False
FavorThreshold = 0.45    #All False predictions between [0.5,FavorThreshold) are switched to True
```

When FavorTrueClass = True, all False predictions very close to 50% probability are switched to True class (a threshold is used, usually set to < 0.45).

Meaning that those predictions very close to random are set to True with the aim to favor the score, and avoid some False Negative penalties.

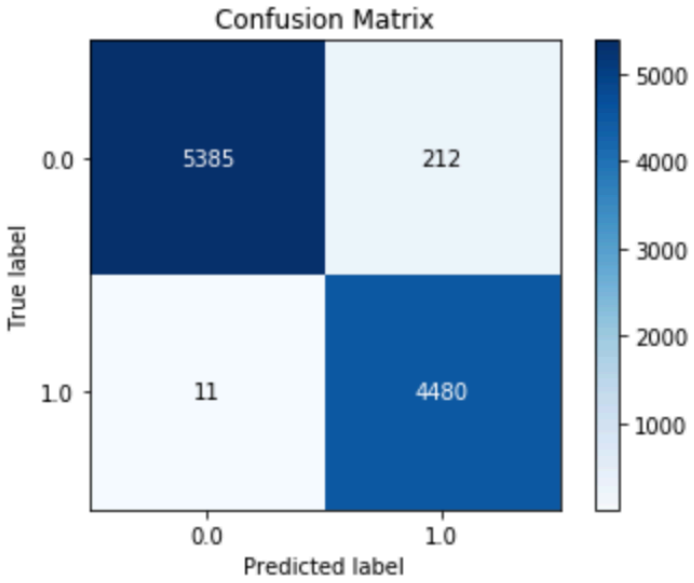
Notes:

(1) Train-Test splits have always been performed using **stratification**.

EVALUATING THE MODEL

Model accuracy, confusion Matrix, and score, are the measures used to evaluate the results.

Confusion Matrix example:



Accuracy example:

```
0.9408788810633866
```

Score example:

```
def getCost(y_test,y_pred):
    tn,fp,fn,tp = confusion_matrix(y_test,y_pred).ravel()
    return (fp*100 + fn*3500)
```

```
print (getCost(y_test, y_pred))
```

```
59700
```

PREDICTION

All data transformations, cleansing, feature selection, projections, except for resampling, are applied to the Test dataset. Those operations that could not be replicated in the Test dataset have been avoided.

Recently trained model is used to predict the Test dataset Class feature. Good results, or sometimes just very different approaches, are uploaded to Kaggle challenge.

Every Kaggle upload has in its description the hyperparameters used, to replicate any upload the only thing needed is to set those options and run it again. Random seeds, when used, are also informed in the description.

SUMMARY OF METHODS

- imblearn.over_sampling.**SMOTE** (resampling)
- imblearn.over_sampling.**ADASYN** (resampling)
- sklearn.impute.**Imputer** (NaNs imputation)
- sklearn.ensemble.**RandomForestClassifier** (classification)
- sklearn.svm.**SVC** (classification)
- sklearn.metrics.**make_scorer** (custom score)
- sklearn.model_selection.**GridSearchCV** (model parametrization)
- sklearn.ensemble.**BaggingClassifier** (model ensemble)
- **Pearson's Correlation** (feature correlation analysis)
- **Stratification** when splitting Train-Test datasets
- **Bootstraping** option for models when possible
- Interquartile Range Rule (**IQR**) for Outlier identification