model_per_device_and_time

January 26, 2017

In [1]:
```
import pandas as pd
import numpy as np
import calendar

from bokeh.charts import output_notebook, Scatter, Bar, show, output_file, Line, BoxPlot
from bokeh.plotting import figure
from bokeh.layouts import row, column, gridplot
from bokeh.charts import output_notebook, Scatter, Bar, show, output_file, Line, BoxPlot
from bokeh.io import hplot
from bokeh.models.ranges import Range1d

from ML import filter_devices, build_deriv, resample_per_device, subsample_negatives
from fft import fft_peak

from sklearn.model_selection import cross_val_score
from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.preprocessing import Normalizer
from sklearn.metrics import roc_curve, auc
from sklearn.svm import SVC, NuSVC
from sklearn.model_selection import LeavePGroupsOut, GroupShuffleSplit
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, accuracy_score,precision_recall_curve, auc
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold

output_notebook()
```

In [2]:
```
INPUT="data/train.csv"
dataset = pd.read_csv(INPUT,index_col=[0,1],parse_dates=[0])
```

0.1 Build Training set

In [3]:
```
def pre_filter(df):
    res = df.copy()
    del res["attribute1"]
```

```
        del res["attribute3"]
        #del res["attribute5"]
        dt_list = ["attribute2"]#,"attribute3"]
        for c in dt_list:
            deriv = build_deriv(res,c)
            res["dt_%s" % c] = deriv
            res["dt2_%s" % c] =  build_deriv(res,c,2)
        return res.fillna(0)

    def post_filter(df):
        res = df.copy()
        res = filter_devices(res)
        for col in res.columns:
            if "min" in col:
                del res[col]
            if "std" in col:
                del res[col]
        return res
```

In [4]:
```
pre_dataset = pre_filter(dataset)
#print feature_set.columns

features = [f for f in pre_dataset.columns if "att" in f]
def f_to_dict(feature):
    indexes = dict( ("avg_over%i_%s" % (i,feature), lambda df: df[:(i+1)].mean()) for i
    d = {
            "min_%s" % feature:np.min,
            "max_%s" % feature:np.max,
            "mean_%s" % feature :np.mean,
            "std_%s" % feature:np.std
        }
    d.update(indexes)
    dft_list = ["attribute4","attribute5", "attribute6","attribute7","attribute9"]
    if feature in dft_list:
        d["dft_p0_ind%s" % feature] = lambda r : fft_peak(r,p=0,index_no_value=True)
        d["dft_p0_val%s" % feature] = lambda r : fft_peak(r,p=0,index_no_value=False)
    return d

agg_dict = dict( (f,f_to_dict(f)) for f in features )
```

In [5]:
```
# bugfix: rolling aggregation after group by does not handle multiple aggregation per co
# we fix this by flattening the aggregation dict and repeating the data within the dataf
final_columns = [k for c in agg_dict for k in agg_dict[c]]
input_columns = [c for c in agg_dict for k in agg_dict[c]]
flat_agg_dict = dict( (k,agg_dict[c][k]) for c in agg_dict for k in agg_dict[c])
dup_dataset = pre_dataset[input_columns].sort_index(level="date").sort_index(level="devi
dup_dataset.columns = final_columns
```

In [6]:
```
#
```

```
        # instead of simply grouping by, we roll over the dataset...
        # using the "hacky" flat version, to avoid issues.
        # The hack increase the memory consumption quite a lot, but it should still be better th
        # explicitely building the windowed lines before aggregating over it.
        #

        feature_set = resample_per_device(dup_dataset) \
            .groupby(level="device",as_index=False) \
            .rolling(window=360,min_periods=1) \
            .agg(flat_agg_dict) \
            .reset_index(level=0,drop=True)
        feature_set = post_filter(feature_set).sortlevel(level="device")
```

```
In [7]: #
        # feature filtering : removing features with a weak contribution to the last computed mo
        #
        feat_filtering_thres = 5e-3
        try :
            kept_features = feature_imp[feature_imp.importance > feat_filtering_thres]
            print "threshold : %g, kept: %i features" % (feat_filtering_thres,kept_features.size
            filtered = feature_set.filter(items=feature_imp.index.sort_values())
            print "filtering devices"
            feature_set = filtered
        except:
            print "no feature filtering"
```

```
no feature filtering
```

```
In [8]: #
        # use label_window to expand label to neighboring days.
        # basically, a mainrtenance x days before failure is still OK
        #
        label_window = 7

        label_set =  resample_per_device(dataset[["failure"]]) \
            .sortlevel(level="date",ascending=False) \
            .groupby(level="device",as_index=False) \
            .rolling(window=label_window, min_periods=1) \
            .sum() \
            .reset_index(level=0,drop=True)
        label_set = filter_devices(label_set).sortlevel(level="device")
```

```
In [36]: #
         # subsampling the negatives, to balance classes
         #
         negative_subsampling_fraction = 5e-2
         sub_label_set , sub_feature_set = subsample_negatives(negative_subsampling_fraction,lab
```

```
 126681 with 630 positives
new size 6933
```

```
In [9]: # compute groups (devices), to be used when splitting the training set (train/test)
        # This is useful to avoid the bias selection generated by a temporal model
        # (Basically, a device used in train cannot also be in test, because its attributes will
        # this is why the cross-val strategy must split by device, and not merely at random
        devices = sub_label_set.index.get_level_values("device")
        device_index = dict((device,i) for i,device in enumerate(devices.unique()))
        n_dev = len(device_index)
        device_groups = np.array(devices.to_series().map(device_index).tolist())


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-9-35847931f59d> in <module>()
          3 # (Basically, a device used in train cannot also be in test, because its attributes
          4 # this is why the cross-val strategy must split by device, and not merely at random
    ----> 5 devices = sub_label_set.index.get_level_values("device")
          6 device_index = dict((device,i) for i,device in enumerate(devices.unique()))
          7 n_dev = len(device_index)


        NameError: name 'sub_label_set' is not defined
```

0.2 Run model

```
In [ ]: feature_mat = sub_feature_set.as_matrix()
        label_mat = sub_label_set.as_matrix().ravel()

        splitting_strategy = GroupShuffleSplit(n_splits=4,test_size=0.33)

        pca = PCA()
        norm = Normalizer()

        #model=GradientBoostingClassifier()
        model = RandomForestClassifier()
        #model = SVC(probability=True)

        #pipeline= Pipeline([('normalize', norm),('reduce_dim', pca),("model",model)])
        pipeline= Pipeline([("model",model)])
        #pipeline = pipeline_optimizer._fitted_pipeline

        try:
```

```
            # use best parameters if available
            #
            pipeline.set_params(**grid_result.best_params_)
            print "using last optimized model"
        except:
            print "no optim result, or bad ones: let's keep the default ones"
            pass
```

```
In [ ]: scores = cross_val_score(
            pipeline,
            feature_mat,
            label_mat,
            cv=splitting_strategy,
            groups = device_groups,
            verbose=1,
            scoring="f1",
            n_jobs=6)
        print "accurracy: %g, std(%g))" % (scores.mean(), scores.std())
```

0.2.1 Eval Model

```
In [15]: #
         # build the curve of PR, AUC on test and on train
         #

         X_train, X_test, Y_train, Y_test = train_test_split(feature_mat,label_mat,test_size=0.3
         # calculate the fpr and tpr for all thresholds of the classification

         fitted = pipeline.fit(X_train,Y_train)
         probs = fitted.predict_proba(X_test)
         preds = probs[:,1]
         preds_train = fitted.predict_proba(X_train)[:,1]
         fpr, tpr, threshold = roc_curve(Y_test, preds)
         fpr_train, tpr_train, threshold_train = roc_curve(Y_train, preds_train)
         roc_auc = auc(fpr, tpr)
         roc_auc_train = auc(fpr_train, tpr_train)
         precision, recall, ths = precision_recall_curve(Y_test, preds)
         precision_train, recall_train, ths_train = precision_recall_curve(Y_train, preds_train)
```

```
In [16]: #print "auc: %.2g, on train: %.2g" %(roc_auc, roc_auc_train)
         roc_df = pd.DataFrame({"fpr":fpr,"tpr":tpr}).set_index("fpr")
         pr_df = pd.DataFrame({"precision": precision, "recall":recall}).set_index("recall")
         roc_df["diag"] = roc_df.index
         pr_df["random"] = pr_df.precision.iloc[0]

         # roc curve
         roc_f = figure(width=400,height=400,title="roc, auc: %.2g, on train: %.2g"  %(roc_auc,
         roc_f.xaxis.axis_label = "tpr"
```

```
auc_range= Range1d(0,1)
roc_f.x_range = auc_range
roc_f.y_range = auc_range
roc_f.yaxis.axis_label = "fpr"
roc_f.cross(fpr,tpr,size=5)
roc_f.line(fpr,tpr,legend="roc")
roc_f.circle(fpr_train,tpr_train,size=5,color="red", line_width=1)
roc_f.line(fpr_train,tpr_train,color="red",legend="roc on train")
roc_f.line([0,1],[0,1], color="grey")

# pr curve
pr_f = figure(width=400,height=400,title="PR curve")
pr_f.xaxis.axis_label = "recall"
pr_f.yaxis.axis_label = "precision"
pr_f.cross(recall,precision,size=5)
pr_f.line(recall,precision,legend="PR")
pr_f.circle(recall_train,precision_train,size=5,color="red", line_width=1)
pr_f.line(recall_train,precision_train,color="red",legend="PR on train")

show(row(
    pr_f,
    roc_f
))
```

0.2.2 Debug: feature Importance

```
In [ ]: feature_imp = pd.DataFrame({"importance":model.feature_importances_}).set_index(feature_
        feature_imp.sort_values(by="importance",ascending=False)
```

0.2.3 Hyperparameter optimisation: Grid search optim

```
In [46]: # model : GradientBoostingClassifier, parameters:
         #loss : {deviance, exponential},
         #learning_rate : float, optional (default=0.1)
         #n_estimators : int (default=100)
         #max_depth : integer, optional (default=3)
         #min_samples_split : int, float, optional (default=2)
         grids=dict()
         XDB_param_grid = {
             "model__loss":  ["deviance", 'exponential'],
             "model__learning_rate" : [0.1],
             "model__n_estimators" : [100, 150],
             "model__max_depth" : [2,3,5],
             "model__min_samples_split" : [5,10,15]
         }
         grids[GradientBoostingClassifier] = XDB_param_grid

In [47]: # C :penalty
         # kernel : linear, poly, rbf, sigmoid, precomputed
```

```
SVC_param_grid = {
   'model__C': [1e-7,1e-6,1e-5,0.1],
    "model__kernel": ["rbf","linear"],
    #"model_degree" : [1,3,5], # polynomial degrees
    "model__gamma" : ["auto"], # kernel coef (rbf)
    #"coef0" # for poly, signmoid
    "model__tol" : [1e-3,1e-2]
}
grids[SVC] = SVC_param_grid
```

In [48]:
```
# model : RandomForestClassifier, parameters:
# n_estimators : int (default=100)
# criterion : "gini","entropy"
# max_features : auto , fraction
# max_depth : integer, optional (default=3)
# min_samples_split : int, float, optional (default=2)

RF_param_grid = {
    "model__criterion":  ["entropy"],
    "model__n_estimators" : [50,100,150],
    "model__max_features" : ["auto"],
    "model__max_depth" : [20,30,40],
    "model__min_samples_split" : [5,10,15]
}
grids[RandomForestClassifier] = RF_param_grid
```

In [49]:
```
m  = type(dict(pipeline.steps)["model"])
param_grid=grids[m]

splits = splitting_strategy.split(feature_mat,label_mat,device_groups)
#kfold = StratifiedKFold(n_splits=6, shuffle=True)
grid_search = GridSearchCV(
    pipeline,
    param_grid,
    scoring="f1",
    n_jobs=-1,
    verbose=1,
    cv=splits
)
grid_result = grid_search.fit(feature_mat,label_mat)


# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
```

```
Fitting 4 folds for each of 27 candidates, totalling 108 fits
```

```
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   18.1s
[Parallel(n_jobs=-1)]: Done 108 out of 108 | elapsed:   48.6s finished


Best: 0.429978 using {'model__n_estimators': 100, 'model__max_features': 'auto', 'model__max_dep


In [50]: model_name = "device_time"

         from sklearn.externals import joblib
         import pickle
         joblib.dump(grid_result.best_estimator_, 'model_%s.pkl' % model_name)
         #joblib.dump(pipeline, 'model_%s.pkl' % model_name)
         features = feature_set.columns
         with open('model_%s.feat' % model_name, "w+") as f:
             pickle.dump(features,f)

In [ ]: device_mat = sub_feature_set[device_model_features]
        predicted_device = device_model.predict_proba(device_mat)

In [ ]:
```