CS 636 FINAL PROJECT

PLASTICC KAGGLE COMPETITION

CODE

In [ ]:
```
#Load Libraries

import sys, os
import argparse
import time
from datetime import datetime as dt
import gc; gc.enable()
from functools import partial, wraps

import pandas as pd
import numpy as np
np.warnings.filterwarnings('ignore')

from sklearn.model_selection import StratifiedKFold
from tsfresh.feature_extraction import extract_features
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier

from numba import jit
```

Processing the metadata file and summarizing

In [ ]:
```
def process_meta(filename):
    meta_df = pd.read_csv(filename)

    meta_dict = dict()
    # distance
    meta_dict.update(haversine_plus(meta_df['ra'].values, meta_df['decl'].valu
es,
                   meta_df['gal_l'].values, meta_df['gal_b'].values))
    #
    meta_dict['hostgal_photoz_certain'] = np.multiply(
            meta_df['hostgal_photoz'].values,
             np.exp(meta_df['hostgal_photoz_err'].values))

    meta_df = pd.concat([meta_df, pd.DataFrame(meta_dict, index=meta_df.index
)], axis=1)
    return meta_df
```

Preprocessing train_set and test_set

In [ ]:
```
def featurize(df, df_meta, aggs, fcp, n_jobs=4):
    df = process_flux(df)

    agg_df = df.groupby('object_id').agg(aggs)
    agg_df.columns = [ '{}_{}'.format(k, agg) for k in aggs.keys() for agg in
aggs[k]]
    agg_df = process_flux_agg(agg_df)

    agg_df_ts_flux_passband = extract_features(df,
                                               column_id='object_id',
                                               column_sort='mjd',
                                               column_kind='passband',
                                               column_value='flux',
                                               default_fc_parameters=fcp['flux
_passband'], n_jobs=n_jobs)

    agg_df_ts_flux = extract_features(df,
                                      column_id='object_id',
                                      column_value='flux',
                                      default_fc_parameters=fcp['flux'], n_job
s=n_jobs)

    agg_df_ts_flux_by_flux_ratio_sq = extract_features(df,
                                      column_id='object_id',
                                      column_value='flux_by_flux_ratio_sq',
                                      default_fc_parameters=fcp['flux_by_flux_
ratio_sq'], n_jobs=n_jobs)


    df_det = df[df['detected']==1].copy()
    agg_df_mjd = extract_features(df_det,
                                  column_id='object_id',
                                  column_value='mjd',
                                  default_fc_parameters=fcp['mjd'], n_jobs=n_j
obs)
    agg_df_mjd['mjd_diff_det'] = agg_df_mjd['mjd__maximum'].values - agg_df_mj
d['mjd__minimum'].values
    del agg_df_mjd['mjd__maximum'], agg_df_mjd['mjd__minimum']

    agg_df_ts_flux_passband.index.rename('object_id', inplace=True)
    agg_df_ts_flux.index.rename('object_id', inplace=True)
    agg_df_ts_flux_by_flux_ratio_sq.index.rename('object_id', inplace=True)
    agg_df_mjd.index.rename('object_id', inplace=True)
    agg_df_ts = pd.concat([agg_df,
                           agg_df_ts_flux_passband,
                           agg_df_ts_flux,
                           agg_df_ts_flux_by_flux_ratio_sq,
                           agg_df_mjd], axis=1).reset_index()

    result = agg_df_ts.merge(right=df_meta, how='left', on='object_id')
    return result
```

Creating new fields

In [ ]:
```
@jit
def haversine_plus(lon1, lat1, lon2, lat2):
    #Convert decimal degrees to Radians:
    lon1 = np.radians(lon1)
    lat1 = np.radians(lat1)
    lon2 = np.radians(lon2)
    lat2 = np.radians(lat2)

    #Implementing Haversine Formula:
    dlon = np.subtract(lon2, lon1)
    dlat = np.subtract(lat2, lat1)

    a = np.add(np.power(np.sin(np.divide(dlat, 2)), 2),
                          np.multiply(np.cos(lat1),
                                      np.multiply(np.cos(lat2),
                                                  np.power(np.sin(np.divide(dl
on, 2)), 2))))

    haversine = np.multiply(2, np.arcsin(np.sqrt(a)))
    return {
        'haversine': haversine,
        'latlon1': np.subtract(np.multiply(lon1, lat1), np.multiply(lon2, lat2
)),
   }


@jit
def process_flux(df):
    flux_ratio_sq = np.power(df['flux'].values / df['flux_err'].values, 2.0)

    df_flux = pd.DataFrame({
        'flux_ratio_sq': flux_ratio_sq,
        'flux_by_flux_ratio_sq': df['flux'].values * flux_ratio_sq,},
        index=df.index)

    return pd.concat([df, df_flux], axis=1)


@jit
def process_flux_agg(df):
    flux_w_mean = df['flux_by_flux_ratio_sq_sum'].values / df['flux_ratio_sq_s
um'].values
    flux_diff = df['flux_max'].values - df['flux_min'].values

    df_flux_agg = pd.DataFrame({
        'flux_w_mean': flux_w_mean,
        'flux_diff1': flux_diff,
        'flux_diff2': flux_diff / df['flux_mean'].values,
        'flux_diff3': flux_diff /flux_w_mean,
        }, index=df.index)

    return pd.concat([df, df_flux_agg], axis=1)
```

Prediction functions

In [ ]:
```
def predict_chunk(df_, clfs_, meta_, features, featurize_configs, train_mean):

    # process all features
    full_test = featurize(df_, meta_,
                          featurize_configs['aggs'],
                          featurize_configs['fcp'])
    full_test.fillna(0, inplace=True)

    # Make predictions
    preds_ = None
    for clf in clfs_:
        if preds_ is None:
            preds_ = clf.predict_proba(full_test[features])
        else:
            preds_ += clf.predict_proba(full_test[features])

    preds_ = preds_ / len(clfs_)

    preds_99 = np.ones(preds_.shape[0])
    for i in range(preds_.shape[1]):
        preds_99 *= (1 - preds_[:, i])

    preds_df_ = pd.DataFrame(preds_,
                             columns=['class_{}'.format(s) for s in clfs_[0].c
lasses_])
    preds_df_['object_id'] = full_test['object_id']
    preds_df_['class_99'] = 0.14 * preds_99 / np.mean(preds_99)
    return preds_df_


def process_test(clfs,
                 features,
                 featurize_configs,
                 train_mean,
                 filename='predictions.csv',
                 chunks=5000000):
    start = time.time()

    meta_test = process_meta('test_set_metadata.csv')

    remain_df = None
    for i_c, df in enumerate(pd.read_csv('test_set.csv', chunksize=chunks, ite
rator=True)):
        unique_ids = np.unique(df['object_id'])

        new_remain_df = df.loc[df['object_id'] == unique_ids[-1]].copy()
        if remain_df is None:
            df = df.loc[df['object_id'].isin(unique_ids[:-1])]
        else:
            df = pd.concat([remain_df, df.loc[df['object_id'].isin(unique_ids
[:-1])]], axis=0)
        remain_df = new_remain_df

        preds_df = predict_chunk(df_=df,
                                 clfs_=clfs,
                                 meta_=meta_test,
```

```
                                  features=features,
                                  featurize_configs=featurize_configs,
                                  train_mean=train_mean)

        if i_c == 0:
            preds_df.to_csv(filename, header=True, mode='a', index=False)
        else:
            preds_df.to_csv(filename, header=False, mode='a', index=False)

        del preds_df
        gc.collect()
        print('{:15d} done in {:5.1f} minutes' .format(
                chunks * (i_c + 1), (time.time() - start) / 60), flush=True)


    preds_df = predict_chunk(df_=remain_df,
                             clfs_=clfs,
                             meta_=meta_test,
                             features=features,
                             featurize_configs=featurize_configs,
                             train_mean=train_mean)

    preds_df.to_csv(filename, header=False, mode='a', index=False)
    return
```

Modelling

In [ ]:
```
def multi_weighted_logloss(y_true, y_preds, classes, class_weights):
    y_p = y_preds.reshape(y_true.shape[0], len(classes), order='F')
    y_ohe = pd.get_dummies(y_true)
    y_p = np.clip(a=y_p, a_min=1e-15, a_max=1 - 1e-15)
    y_p_log = np.log(y_p)
    y_log_ones = np.sum(y_ohe.values * y_p_log, axis=0)
    nb_pos = y_ohe.sum(axis=0).values.astype(float)
    class_arr = np.array([class_weights[k] for k in sorted(class_weights.keys
())])
    y_w = y_log_ones * class_arr / nb_pos

    loss = - np.sum(y_w) / np.sum(class_arr)
    return loss


def lgbm_multi_weighted_logloss(y_true, y_preds):
    classes = [6, 15, 16, 42, 52, 53, 62, 64, 65, 67, 88, 90, 92, 95]
    class_weights = {6: 1, 15: 2, 16: 1, 42: 1, 52: 1, 53: 1, 62: 1, 64: 2, 65
: 1, 67: 1, 88: 1, 90: 1, 92: 1, 95: 1}

    loss = multi_weighted_logloss(y_true, y_preds, classes, class_weights)
    return 'wloss', loss, False


def xgb_multi_weighted_logloss(y_predicted, y_true, classes, class_weights):
    loss = multi_weighted_logloss(y_true.get_label(), y_predicted,
                                  classes, class_weights)
    return 'wloss', loss


def save_importances(importances_):
    mean_gain = importances_[['gain', 'feature']].groupby('feature').mean()
    importances_['mean_gain'] = importances_['feature'].map(mean_gain['gain'])
    return importances_


def xgb_modeling_cross_validation(params,
                                  full_train,
                                  y,
                                  classes,
                                  class_weights,
                                  nr_fold=5,
                                  random_state=1):
    # Compute weights
    w = y.value_counts()
    weights = {i : np.sum(w) / w[i] for i in w.index}

    # Loss function
    func_loss = partial(xgb_multi_weighted_logloss,
                        classes=classes,
                        class_weights=class_weights)

    clfs = []
    importances = pd.DataFrame()
    folds = StratifiedKFold(n_splits=nr_fold,
                            shuffle=True,
```

```
                            random_state=random_state)

    oof_preds = np.zeros((len(full_train), np.unique(y).shape[0]))
    for fold_, (trn_, val_) in enumerate(folds.split(y, y)):
        trn_x, trn_y = full_train.iloc[trn_], y.iloc[trn_]
        val_x, val_y = full_train.iloc[val_], y.iloc[val_]

        clf = XGBClassifier(**params)
        clf.fit(
            trn_x, trn_y,
            eval_set=[(trn_x, trn_y), (val_x, val_y)],
            eval_metric=func_loss,
            verbose=100,
            early_stopping_rounds=50,
            sample_weight=trn_y.map(weights)
        )
        clfs.append(clf)

        oof_preds[val_, :] = clf.predict_proba(val_x, ntree_limit=clf.best_ntr
ee_limit)
        print('no {}-fold loss: {}'.format(fold_ + 1,
              multi_weighted_logloss(val_y, oof_preds[val_, :],
                                     classes, class_weights)))

        imp_df = pd.DataFrame({
                'feature': full_train.columns,
                'gain': clf.feature_importances_,
                'fold': [fold_ + 1] * len(full_train.columns),
                })
        importances = pd.concat([importances, imp_df], axis=0)

    score = multi_weighted_logloss(y_true=y, y_preds=oof_preds,
                                   classes=classes, class_weights=class_weight
s)
    print('MULTI WEIGHTED LOG LOSS: {:.5f}'.format(score))
    df_importances = save_importances(importances_=importances)
    df_importances.to_csv('xgb_importances.csv', index=False)

    return clfs, score


def lgbm_modeling_cross_validation(params,
                                   full_train,
                                   y,
                                   classes,
                                   class_weights,
                                   nr_fold=5,
                                   random_state=1):

    w = y.value_counts()
    weights = {i : np.sum(w) / w[i] for i in w.index}

    clfs = []
    importances = pd.DataFrame()
    folds = StratifiedKFold(n_splits=nr_fold,
                            shuffle=True,
                            random_state=random_state)
```

```
    oof_preds = np.zeros((len(full_train), np.unique(y).shape[0]))
    for fold_, (trn_, val_) in enumerate(folds.split(y, y)):
        trn_x, trn_y = full_train.iloc[trn_], y.iloc[trn_]
        val_x, val_y = full_train.iloc[val_], y.iloc[val_]

        clf = LGBMClassifier(**params)
        clf.fit(
            trn_x, trn_y,
            eval_set=[(trn_x, trn_y), (val_x, val_y)],
            eval_metric=lgbm_multi_weighted_logloss,
            verbose=100,
            early_stopping_rounds=50,
            sample_weight=trn_y.map(weights)
        )
        clfs.append(clf)

        oof_preds[val_, :] = clf.predict_proba(val_x, num_iteration=clf.best_i
teration_)
        print('no {}-fold loss: {}'.format(fold_ + 1,
              multi_weighted_logloss(val_y, oof_preds[val_, :],
                                     classes, class_weights)))

        imp_df = pd.DataFrame({
                'feature': full_train.columns,
                'gain': clf.feature_importances_,
                'fold': [fold_ + 1] * len(full_train.columns),
                })
        importances = pd.concat([importances, imp_df], axis=0)

    score = multi_weighted_logloss(y_true=y, y_preds=oof_preds,
                                   classes=classes, class_weights=class_weight
s)
    print('MULTI WEIGHTED LOG LOSS: {:.5f}'.format(score))
    df_importances = save_importances(importances_=importances)
    df_importances.to_csv('lgbm_importances.csv', index=False)

    return clfs, score
```

Main code

In [ ]:
```
aggs = {
    'flux': ['min', 'max', 'mean', 'median', 'std', 'skew'],
    'flux_err': ['min', 'max', 'mean', 'median', 'std', 'skew'],
    'detected': ['mean'],
    'flux_ratio_sq':['sum', 'skew'],
    'flux_by_flux_ratio_sq':['sum','skew'],
}


fcp = {
    'flux': {
        'longest_strike_above_mean': None,
        'longest_strike_below_mean': None,
        'mean_change': None,
        'mean_abs_change': None,
        'length': None,
    },

    'flux_by_flux_ratio_sq': {
        'longest_strike_above_mean': None,
        'longest_strike_below_mean': None,
    },

    'flux_passband': {
        'fft_coefficient': [
                {'coeff': 0, 'attr': 'abs'},
                {'coeff': 1, 'attr': 'abs'}
            ],
        'kurtosis' : None,
        'skewness' : None,
    },

    'mjd': {
        'maximum': None,
        'minimum': None,
        'mean_change': None,
        'mean_abs_change': None,
    },
}

best_params = {
        'device': 'cpu',
        'objective': 'multiclass',
        'num_class': 14,
        'boosting_type': 'gbdt',
        'n_jobs': -1,
        'max_depth': 7,
        'n_estimators': 500,
        'subsample_freq': 2,
        'subsample_for_bin': 5000,
        'min_data_per_group': 100,
        'max_cat_to_onehot': 4,
        'cat_l2': 1.0,
        'cat_smooth': 59.5,
        'max_cat_threshold': 32,
        'metric_freq': 10,
```

```
        'verbosity': -1,
        'metric': 'multi_logloss',
        'xgboost_dart_mode': False,
        'uniform_drop': False,
        'colsample_bytree': 0.5,
        'drop_rate': 0.173,
        'learning_rate': 0.0267,
        'max_drop': 5,
        'min_child_samples': 10,
        'min_child_weight': 100.0,
        'min_split_gain': 0.1,
        'num_leaves': 7,
        'reg_alpha': 0.1,
        'reg_lambda': 0.00023,
        'skip_drop': 0.44,
        'subsample': 0.75}

meta_train = process_meta('training_set_metadata.csv')

train = pd.read_csv('training_set.csv')
full_train = featurize(train, meta_train, aggs, fcp)

if 'target' in full_train:
    y = full_train['target']
    del full_train['target']

classes = sorted(y.unique())

class_weights = {c: 1 for c in classes}
class_weights.update({c:2 for c in [64, 15]})
print('Unique classes : {}, {}'.format(len(classes), classes))
print(class_weights)

if 'object_id' in full_train:
    oof_df = full_train[['object_id']]
    del full_train['object_id']
    del full_train['hostgal_specz']
    del full_train['ra'], full_train['decl'], full_train['gal_l'], full_train[
'gal_b']
    del full_train['ddf']

train_mean = full_train.mean(axis=0)

pd.set_option('display.max_rows', 500)
print(full_train.describe().T)

full_train.fillna(0, inplace=True)

eval_func = partial(lgbm_modeling_cross_validation,
                    full_train=full_train,
                    y=y,
                    classes=classes,
                    class_weights=class_weights,
                    nr_fold=7,
                    random_state=7)

best_params.update({'n_estimators': 1100})
```

```
clfs, score = eval_func(best_params)

filename = "kaggle_submission_final.csv"
print('save to {}'.format(filename))

process_test(clfs,
             features=full_train.columns,
             featurize_configs={'aggs': aggs, 'fcp': fcp},
             train_mean=train_mean,
             filename=filename,
             chunks=5000000)

z = pd.read_csv(filename)
print("Shape BEFORE grouping: {}".format(z.shape))
z = z.groupby('object_id').mean()
print("Shape AFTER grouping: {}".format(z.shape))
z.to_csv(filename, index=True)
```