In this notebook we will train the best model obtained from experimentation, save that model, load it from memory. Then we will define a function that will give us the result for a given data point

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

In [2]:
```
# Loading datasets
train_df = pd.read_csv("training.csv")
test_df = pd.read_csv("test.csv")
agree_df = pd.read_csv("check_agreement.csv")
corr_df = pd.read_csv("check_correlation.csv")
```

Check agrrement test

In [3]:
```
# agreement test as mentioned in kaggle resources#
from sklearn.metrics import roc_curve, auc

def __roc_curve_splitted(data_zero, data_one, sample_weights_zero, sample_weights
    """
    Compute roc curve

    :param data_zero: 0-labeled data
    :param data_one:  1-labeled data
    :param sample_weights_zero: weights for 0-labeled data
    :param sample_weights_one:  weights for 1-labeled data
    :return: roc curve
    """
    labels = [0] * len(data_zero) + [1] * len(data_one)
    weights = np.concatenate([sample_weights_zero, sample_weights_one])
    data_all = np.concatenate([data_zero, data_one])
    fpr, tpr, _ = roc_curve(labels, data_all, sample_weight=weights)
    return fpr, tpr

def compute_ks(data_prediction, mc_prediction, weights_data, weights_mc):
    """
    Compute Kolmogorov-Smirnov (ks) distance between real data predictions cdf ar

    :param data_prediction: array-like, real data predictions
    :param mc_prediction: array-like, Monte Carlo data predictions
    :param weights_data: array-like, real data weights
    :param weights_mc: array-like, Monte Carlo weights
    :return: ks value
    """
    assert len(data_prediction) == len(weights_data), 'Data length and weight one
    assert len(mc_prediction) == len(weights_mc), 'Data length and weight one mus

    data_prediction, mc_prediction = np.array(data_prediction), np.array(mc_predi
    weights_data, weights_mc = np.array(weights_data), np.array(weights_mc)

    assert np.all(data_prediction >= 0.) and np.all(data_prediction <= 1.), 'Data
    assert np.all(mc_prediction >= 0.) and np.all(mc_prediction <= 1.), 'MC predi

    weights_data /= np.sum(weights_data)
    weights_mc /= np.sum(weights_mc)

    fpr, tpr = __roc_curve_splitted(data_prediction, mc_prediction, weights_data,

    Dnm = np.max(np.abs(fpr - tpr))
    return Dnm
```

check correlation test

In [4]:
```
# correlation test as mentioned in kaggle resources

def __rolling_window(data, window_size):
    """
    Rolling window: take window with definite size through the array

    :param data: array-like
    :param window_size: size
    :return: the sequence of windows

    Example: data = array(1, 2, 3, 4, 5, 6), window_size = 4
        Then this function return array(array(1, 2, 3, 4), array(2, 3, 4, 5), arr
    """
    shape = data.shape[:-1] + (data.shape[-1] - window_size + 1, window_size)
    strides = data.strides + (data.strides[-1],)
    return np.lib.stride_tricks.as_strided(data, shape=shape, strides=strides)

def __cvm(subindices, total_events):
    """
    Compute Cramer-von Mises metric.
    Compared two distributions, where first is subset of second one.
    Assuming that second is ordered by ascending

    :param subindices: indices of events which will be associated with the first
    :param total_events: count of events in the second distribution
    :return: cvm metric
    """
    target_distribution = np.arange(1, total_events + 1, dtype='float') / total_e
    subarray_distribution = np.cumsum(np.bincount(subindices, minlength=total_eve
    subarray_distribution /= 1.0 * subarray_distribution[-1]
    return np.mean((target_distribution - subarray_distribution) ** 2)

def compute_cvm(predictions, masses, n_neighbours=200, step=50):
    """
    Computing Cramer-von Mises (cvm) metric on background events: take average of
    In each mass bin global prediction's cdf is compared to prediction's cdf in m

    :param predictions: array-like, predictions
    :param masses: array-like, in case of Kaggle tau23mu this is reconstructed ma
    :param n_neighbours: count of neighbours for event to define mass bin
    :param step: step through sorted mass-array to define next center of bin
    :return: average cvm value
    """
    predictions = np.array(predictions)
    masses = np.array(masses)
    assert len(predictions) == len(masses)

    # First, reorder by masses
    predictions = predictions[np.argsort(masses)]

    # Second, replace probabilities with order of probability among other events
    predictions = np.argsort(np.argsort(predictions, kind='mergesort'), kind='mer

    # Now, each window forms a group, and we can compute contribution of each gro
    cvms = []
    for window in __rolling_window(predictions, window_size=n_neighbours)[::step]
```

```
        cvms.append(__cvm(subindices=window, total_events=len(predictions)))
    return np.mean(cvms)
```

In [5]:
```
# feature engineering
def new_feats(df):
    df2 = df.copy()
    df2['isolation_abc'] = df['isolationa'] + df['isolationb'] + df['isolationc']
    df2['isolation_def'] = df['isolationd'] + df['isolatione'] + df['isolationf']
    df2['p_IP'] = df['p0_IP']+df['p1_IP']+df['p2_IP']
    df2['p_p']  = df['p0_p']+df['p1_p']+df['p2_p']
    df2['IP_pp'] = df['IP_p0p2'] + df['IP_p1p2']
    df2['p_IPSig'] = df['p0_IPSig'] + df['p1_IPSig'] + df['p2_IPSig']
    #new feature using 'FlightDu=istance' and LifeTime(from literature)
    df2['FD_LT']=df['FlightDistance']/df['LifeTime']
    #new feature using 'FlightDistance', 'po_p', 'p1_p', 'p2_p'(from literature)
    df2['FD_p0p1p2_p']=df['FlightDistance']/(df['p0_p']+df['p1_p']+df['p2_p'])
    #new feature using 'LifeTime', 'p0_IP', 'p1_IP', 'p2_IP'(from literature)
    df2['NEW5_lt']=df['LifeTime']*(df['p0_IP']+df['p1_IP']+df['p2_IP'])/3
    #new feature using 'p0_track_Chi2Dof', 'p1_track_Chi2Dof', 'p2_track_Chi2Dof
    df2['Chi2Dof_MAX'] = df.loc[:, ['p0_track_Chi2Dof', 'p1_track_Chi2Dof', 'p2_t
    # features from kaggle discussion forum
    df2['flight_dist_sig2'] = (df['FlightDistance']/df['FlightDistanceError'])**2
    df2['flight_dist_sig'] = df['FlightDistance']/df['FlightDistanceError']
    df2['NEW_IP_dira'] = df['IP']*df['dira']
    df2['p0p2_ip_ratio']=df['IP']/df['IP_p0p2']
    df2['p1p2_ip_ratio']=df['IP']/df['IP_p1p2']
    df2['DCA_MAX'] = df.loc[:, ['DOCAone', 'DOCAtwo', 'DOCAthree']].max(axis=1)
    df2['iso_bdt_min'] = df.loc[:, ['p0_IsoBDT', 'p1_IsoBDT', 'p2_IsoBDT']].min(a
    df2['iso_min'] = df.loc[:, ['isolationa', 'isolationb', 'isolationc','isolati
    return df2
```

In [6]:
```
# adding engineered features to training and test datasets
train_df_1 = new_feats(train_df)
test_df_1 = new_feats(test_df)
```

In [7]:
```
# idenifying some features to remove which have been used to engineer new feature
#low importance in EDA
remove = ['id', 'min_ANNmuon', 'production', 'mass', 'signal','SPDhits','CDF1', '
          'p0_pt', 'p1_pt', 'p2_pt','p0_p', 'p1_p', 'p2_p', 'p0_eta', 'p1_eta', '
          'isolationc', 'isolationd', 'isolatione', 'isolationf','p0_IsoBDT', 'p1
          'p2_IP','IP_p0p2', 'IP_p1p2','p0_track_Chi2Dof', 'p1_track_Chi2Dof', 'p
          'p2_IPSig','DOCAone', 'DOCAtwo', 'DOCAthree']
# making a list of features to be used to train the model and make predictions
features = list(f for f in train_df_1.columns if f not in remove)
```

In [9]:
```
# training the actual model
from hep_ml.gradientboosting import UGradientBoostingClassifier
from hep_ml.losses import BinFlatnessLossFunction
loss = BinFlatnessLossFunction(['mass'], n_bins=15, uniform_label=0 , fl_coeffici
model = UGradientBoostingClassifier(loss=loss, n_estimators=900,
                                   max_depth=6,
                                   learning_rate=0.15,
                                   train_features=features,
                                   subsample=0.7)
model.fit(train_df_1[features + ['mass']], train_df_1['signal'])
```

Out[9]:
```
UGradientBoostingClassifier(learning_rate=0.15,
                            loss=BinFlatnessLossFunction(allow_wrong_signs=Tru
e,
                                                         fl_coefficient=15,
                                                         n_bins=15, power=2,
                                                         uniform_features=['mas
s'],
                                                         uniform_label=array
([0])),
                            max_depth=6, max_features=None, max_leaf_nodes=Non
e,
                            min_samples_leaf=1, min_samples_split=2,
                            n_estimators=900,
                            random_state=RandomState(MT19937) at 0x259BE7B3740,
                            splitter...
                            train_features=['LifeTime', 'dira',
                                            'FlightDistance',
                                            'FlightDistanceError', 'IP',
                                            'IPSig', 'VertexChi2', 'pt', 'iso',
                                            'ISO_SumBDT', 'isolation_abc',
                                            'isolation_def', 'p_IP', 'p_p',
                                            'IP_pp', 'p_IPSig', 'FD_LT',
                                            'FD_p0p1p2_p', 'NEW5_lt',
                                            'Chi2Dof_MAX', 'flight_dist_sig2',
                                            'flight_dist_sig', 'NEW_IP_dira',
                                            'p0p2_ip_ratio', 'p1p2_ip_ratio',
                                            'DCA_MAX', 'iso_bdt_min',
                                            'iso_min'],
                            update_tree=True)
```

In [10]:
```
# saving the model to the memory
import pickle
filename = 'finalized_model.sav'
pickle.dump(model, open(filename, 'wb'))
```

In [12]:
```
# Loading the model from memory
loaded_model = pickle.load(open(filename, 'rb'))
```

In [13]:
```
# conducting agreement check test
check_agreement = pd.read_csv("check_agreement.csv")
check_agreement = new_feats(check_agreement)

#check_agreement = pandas.read_csv(folder + 'check_agreement.csv', index_col='id
agreement_probs = loaded_model.predict_proba(check_agreement[features])[:, 1]

ks = compute_ks(
    agreement_probs[check_agreement['signal'].values == 0],
    agreement_probs[check_agreement['signal'].values == 1],
    check_agreement[check_agreement['signal'] == 0]['weight'].values,
    check_agreement[check_agreement['signal'] == 1]['weight'].values)
#print 'KS metric', ks, ks < 0.09
print("KS metric {}".format(ks))
print(ks < 0.09)
```

```
KS metric 0.0777479828637741
True
```

In [14]:
```
# conducting the correlation test
#check_correlation = pandas.read_csv(folder + 'check_correlation.csv', index_col=
check_correlation = pd.read_csv("check_correlation.csv", index_col = "id")
check_correlation = new_feats(check_correlation)
correlation_probs = loaded_model.predict_proba(check_correlation[features])[:, 1]
cvm = compute_cvm(correlation_probs, check_correlation['mass'])
#print 'CvM metric', cvm, cvm < 0.002
print("CvM metric {}".format(cvm))
print(cvm < 0.002)
```

```
CvM metric 0.0018304056413095619
True
```

In [15]:
```
# making submission file with test dataset to be submitted on kaggle
test_probs = loaded_model.predict_proba(test_df_1[features])[:,1]
result = pd.DataFrame({"id": test_df["id"], "prediction": test_probs})
result.to_csv("final_result.csv", index=False)
```

In [ ]:
```

```

In [42]:
```
# defining the function which when given a a data point as a list returns the res
def compute(x):
    """given a data point, x, as a list of values this function returns the label
    df = pd.DataFrame(data = np.array(x).reshape(1, len(x)), columns = list(test_
    df_1 = new_feats(df)
    output = loaded_model.predict(df_1[features])[0]
    if output == 0:
        print("the given point belongs to class {} i.e. it is a background event'
    elif output == 1:
        print("the given point belongs to class {} i.e. it is a signal event".for

    return output
```

In [43]:
```
# testing the function with some test point
test_1 = list(test_df.iloc[0].values)
```

In [44]:
```
output_1 = compute(test_1)
```

```
the given point belongs to class 0 i.e. it is a background event
```

In [45]:
```
output_1
```

Out[45]:
```
0
```

In [46]:
```
test_2 = list(test_df.iloc[222].values)
```

In [47]:
```
output_2 = compute(test_2)
```

```
the given point belongs to class 0 i.e. it is a background event
```

In [48]:
```
test_3 = list(test_df.iloc[111].values)
output_3 = compute(test_3)
```

```
the given point belongs to class 1 i.e. it is a signal event
```

In [ ]:
```

```