1 Lab Notebook 8 - SVM classification in particle physics

In this notebook we will learn about applying SVMs to a larger data set from particle physics.

Data by Sascha Caron (https://www.nikhef.nl/~scaron/). Modified from the Notebook by Viviana Acquaviva (2023). License: BSD-3-clause (https://opensource.org/license/bsd-3-clause/)

In [10]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import rc
from sklearn.svm import SVC, LinearSVC # New algorithm!
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_predict, cross_validate
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn import metrics
from sklearn.model_selection import GridSearchCV # New! This will be used to explore different hyperparameter choices.
```

In [11]:
```
pd.set_option('display.max_columns', 500)
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_colwidth', 100)
rc('text', usetex=False)
```

1.1 Step 1

Read in features and labels from 'ParticleID_features.csv' and 'ParticleID_labels.txt' using pandas. Read in labels from 'ParticleID_labels.txt'. Explore the data set to get an idea of what it looks like (ex: look at first few rows, shape, etc.).

In [12]:
```
from sklearn.model_selection import train_test_split

X = pd.read_csv('ParticleID_features.csv')
y = pd.read_csv('ParticleID_labels.txt',header=None)
data = np.array(y.iloc[:,0])
y = pd.DataFrame(data, columns=['collision'])
```

In [13]:
```
X.head()
```

Out[13]:

	ID	MET	METphi	Type_1	P1	P2	P3	P4	Type_2	P5	P6	P7	P8	Type_3	P9	P10	P11	P12	Type_4	P13	P14
0	0	62803.5	-1.810010	j	137571.0	128444.0	-0.345744	-0.307112	j	174209.0	127932.0	0.826569	2.332000	b	86788.9	84554.9	-0.180795	2.187970	j	140289.0	76955.8
1	1	57594.2	-0.509253	j	161529.0	80458.3	-1.318010	1.402050	j	291490.0	68462.9	-2.126740	-2.582310	e-	44270.1	35139.6	-0.706120	-0.371392	e+	72883.9	26902.2
2	2	82313.3	1.686840	b	167130.0	113078.0	0.937258	-2.068680	j	102423.0	54922.3	1.226850	0.646589	j	60768.9	36244.3	1.102890	-1.434480	j	77714.0	27801.5
3	3	30610.8	2.617120	j	112267.0	61383.9	-1.211050	-1.457800	b	40647.8	39472.0	-0.024646	-2.222800	j	201589.0	32978.6	-2.496040	1.137810	j	90096.7	26964.5
4	4	45153.1	-2.241350	j	178174.0	100164.0	1.166880	-0.018721	j	92351.3	69762.1	0.774114	2.568740	j	61625.2	50086.7	0.652572	-3.012800	j	104193.0	31151.0

In [14]:
```
y.head()
```

Out[14]:

	collision
0	ttbar
1	ttbar
2	ttbar
3	ttbar
4	ttbar

1.2 Step 2

The labels are in the form 'ttbar' and '4top'. Turn these categorical (string-type) labels into an array of zeros and ones, where 'ttbar'=0 and '4top'=1 using 'LabelEncoder' from sklearn.preprocessing. Call this new array "target".

In [15]:
```
from sklearn.preprocessing import LabelEncoder

target = np.where(y['collision'] == 'ttbar', 0, 1)
```

1.3 Step 3

Using describe() on features, look at the "count" row. Some columns only contain fractions of the total number of data set rows, due to the variable number of products in each collision

In [28]:
```
X.describe()
```

Out[28]:

	ID	MET	METphi	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
count	5000.000000	5000.000000	5000.000000	5.000000e+03	5.000000e+03	5000.000000	5000.000000	4.997000e+03	4.997000e+03	4997.000000	4997.000000	4.950000e+03	4950.000000	4950.000000
mean	2499.500000	64071.074332	-0.028916	3.301357e+05	1.540486e+05	-0.039812	-0.003049	2.527799e+05	1.080302e+05	-0.029936	0.007327	2.117980e+05	74863.343131	-0.025104
std	1443.520003	60525.122480	1.819257	3.068202e+05	1.149469e+05	1.361762	1.814855	2.638580e+05	8.136261e+04	1.439105	1.828832	2.510361e+05	46309.512365	1.577316
min	0.000000	290.756000	-3.141010	3.857940e+04	2.825400e+04	-4.110220	-3.140710	1.087540e+04	1.080000e+04	-4.668790	-3.140530	1.221050e+04	10639.800000	-4.520250
25%	1249.750000	24352.375000	-1.619905	1.369522e+05	8.883690e+04	-1.035570	-1.574213	1.007510e+05	6.321840e+04	-1.060500	-1.602460	7.636905e+04	46549.475000	-1.125620
50%	2499.500000	46814.400000	-0.055612	2.263525e+05	1.182015e+05	-0.038731	-0.009037	1.659740e+05	8.584360e+04	-0.057428	0.015111	1.288565e+05	62498.400000	-0.040648
75%	3749.250000	83032.350000	1.537323	4.077158e+05	1.771265e+05	0.943598	1.542370	2.999950e+05	1.238700e+05	1.028340	1.605210	2.421225e+05	89587.500000	1.066302
max	4999.000000	692674.000000	3.141130	3.186360e+06	1.276710e+06	4.141410	3.138540	3.587700e+06	1.146330e+06	4.559150	3.139200	2.800410e+06	788338.000000	4.798090

Therefore, we will consider a subset of the data, so we have limited imputing/manipulation problems. Define "features_lim" as the new limited data set: it only contains the columns

'MET', 'METphi', 'P1', 'P2', 'P3', 'P4', 'P5', 'P6', 'P7', 'P8', 'P9', 'P10', 'P11', 'P12', 'P13', 'P14', 'P15', and 'P16'

There may still be some columns with NaN values, so replace NaN with 0 for the moment.

In [22]:
```
features_lim = X[['MET','METphi','P1','P2','P3','P4','P5','P6','P7','P8','P9', 'P10', 'P11', 'P12', 'P13', 'P14', 'P15', 'P16']]

column_means = features_lim.mean()
features_lim.fillna(column_means, inplace=True)
```

```
C:\Users\kesha\AppData\Local\Temp\ipykernel_16816\3812131856.py:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
  features_lim.fillna(column_means, inplace=True)
```

Out[22]:

	MET	METphi	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
count	5000.000000	5000.000000	5.000000e+03	5.000000e+03	5000.000000	5000.000000	5.000000e+03	5.000000e+03	5000.000000	5000.000000	5.000000e+03	5000.000000	5000.000000	5000.000000
mean	64071.074332	-0.028916	3.301357e+05	1.540486e+05	-0.039812	-0.003049	2.527799e+05	1.080302e+05	-0.029936	0.007327	2.117980e+05	74863.343131	-0.025104	0.011845
std	60525.122480	1.819257	3.068202e+05	1.149469e+05	1.361762	1.814855	2.637788e+05	8.133820e+04	1.438673	1.828283	2.497775e+05	46077.336470	1.569408	1.793677
min	290.756000	-3.141010	3.857940e+04	2.825400e+04	-4.110220	-3.140710	1.087540e+04	1.080000e+04	-4.668790	-3.140530	1.221050e+04	10639.800000	-4.520250	-3.141480
25%	24352.375000	-1.619905	1.369522e+05	8.883690e+04	-1.035570	-1.574213	1.007728e+05	6.322358e+04	-1.059270	-1.599617	7.673668e+04	46708.450000	-1.108390	-1.532478
50%	46814.400000	-0.055612	2.263525e+05	1.182015e+05	-0.038731	-0.009037	1.661110e+05	8.585455e+04	-0.056810	0.012737	1.304675e+05	62857.350000	-0.025104	0.011845
75%	83032.350000	1.537323	4.077158e+05	1.771265e+05	0.943598	1.542370	2.999058e+05	1.238520e+05	1.028055	1.601880	2.406498e+05	89065.300000	1.048617	1.553310
max	692674.000000	3.141130	3.186360e+06	1.276710e+06	4.141410	3.138540	3.587700e+06	1.146330e+06	4.559150	3.139200	2.800410e+06	788338.000000	4.798090	3.139020

Note: replacing NaN with 0 is the simplest but worst possible choice - imputing a constant value skews the model. One step up would be to input the mean or median for each column. However, because only a limited number of instances have missing data, the choice of imputing strategy doesn't matter too much.

Use "describe()" to confirm that the count is the same for all features:

In [23]:
```
features_lim.count()
```

Out[23]:
```
MET       5000
METphi    5000
P1        5000
P2        5000
P3        5000
P4        5000
P5        5000
P6        5000
P7        5000
P8        5000
P9        5000
P10       5000
P11       5000
P12       5000
P13       5000
P14       5000
P15       5000
P16       5000
dtype: int64
```

1.4 Step 4: labels and benchmarking

What percentage of the data has the negative label (0) versus the positive label (1)? What is the accuracy of a classifier that puts everything in the negative class?

In [29]:
```
np.bincount(target)
# We notice that there are 4189 instances of 0 ('4top') and 811 instances of 1 ('ttbar')

lazyClassifier_pred = np.zeros(target.shape[0])
lazyClassifier_accuracy = np.sum(lazyClassifier_pred == target)/target.shape[0]
print(f"Accuracy of the lazy classifier :  {lazyClassifier_accuracy}")
```

```
Accuracy of the lazy classifier :  0.8378
```

For contrast, a random classifier that just assigns a random value according to class distribution has the following accuracy:

In [65]:
```

# Generate an array of random integers
randomClassifier_pred = np.random.randint(0, 2, size=target.size)
randomClassifier_accuracy = np.sum(randomClassifier_pred == target)/target.size
print(f"Accuracy of the random classifier :  {randomClassifier_accuracy}")
```

```
Accuracy of the random classifier :  0.489
```

1.5 Step 5: Let's start with a linear model; model = LinearSVC()

1. Define a benchmark linear model, "bmodel", using LinearSVC(dual=False).
2. Use "StratifiedKFold" with 5 splits, shuffle set to True, and a random state of 101 to produce a cross-validation object, "cv".
3. Run "cross_validate", where scoring = 'accuracy' and return_train_score=True. This will output the fit time, the score time, the test score, and the train score. Print these.
4. Print the mean and standard deviation of the test_score.

In [36]:
```
bmodel = LinearSVC(dual=False)
cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=101)
cross_validate(bmodel,X=features_lim,y=target,scoring='accuracy',cv=cv, return_train_score=True)
```

Out[36]:
```
{'fit_time': array([0.0148387 , 0.01896191, 0.01199746, 0.00999808, 0.01099706]),
 'score_time': array([0.00116134, 0.00200248, 0.00200152, 0.00100207, 0.00199771]),
 'test_score': array([0.836, 0.837, 0.831, 0.829, 0.832]),
 'train_score': array([0.834  , 0.83375, 0.8375 , 0.8375 , 0.83625])}
```

1.6 Step 6

Technically, standardizing/normalizing data using the entire learning set introduces leakage between train and test set (the test set "knows" about the mean and standard deviation of the entire data set). Usually this is not a dramatic effect, but the correct procedure is to derive the scaler within each CV fold (i.e. after separating in train and test), only on the train set, and apply the same transformation to the test set. The model then becomes a pipeline.

Similar to lab 5-6, set up a pipeline with StandardScaler and LinearSVC(dual=False,C=1000), use it in cross_validate and report the results. For the test and train scores, print the mean and standard deviation:

In [50]:
```
from sklearn.pipeline import Pipeline

scaler = StandardScaler()
linear_svc = LinearSVC(dual=False,C=1000)
pipeline = Pipeline([('scaler', scaler), ('svc', linear_svc)])

scores = cross_validate(pipeline,X=features_lim,y=target,scoring='accuracy',cv=cv, return_train_score=True)

train_score_mean = np.mean(scores['train_score'])
train_score_std = np.std(scores['train_score'])

test_score_std = np.std(scores['test_score'])
test_score_mean = np.mean(scores['test_score'])

print(f"The training score is {train_score_mean} +- {train_score_std}")
print(f"The testing score is {test_score_mean} +- {test_score_std}")
```

```
The training score is 0.8946500000000001 +- 0.001991230775173974
The testing score is 0.8922000000000001 +- 0.0035440090293338733
```

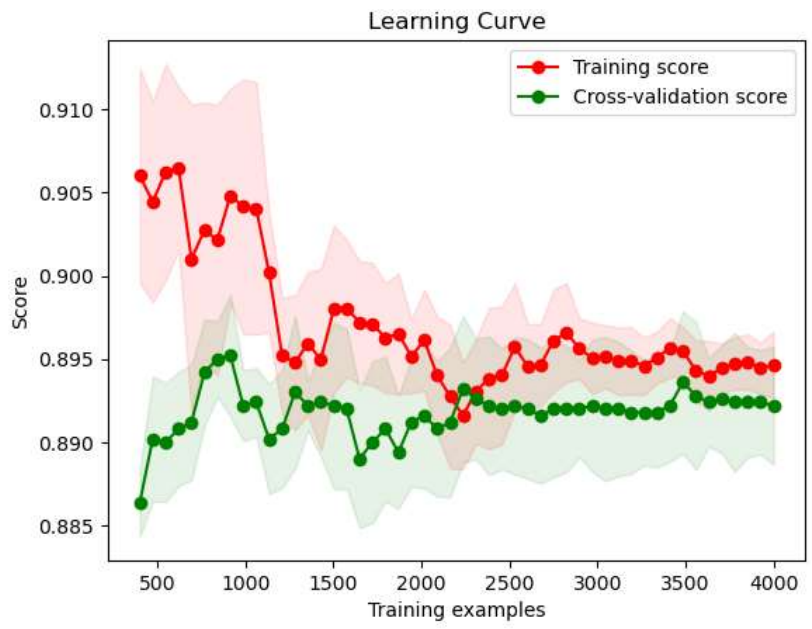
For the test and train scores of benchmark_lim_piped, print the mean and standard deviation:

This should show a significant improvement, and the comparison between test and train scores tells us already something about the problem that we have. We can formalize this by looking at the learning curves, which tell us both about gap between train/test scores, AND whether we need more data.

1.7 Step 7: Learning Curves

As in lab 5-6, construct the learning curve (you can recycle code). What does it tell you?

In [57]:
```
from sklearn.model_selection import learning_curve

train_sizes, train_scores, test_scores = learning_curve(pipeline,features_lim, target, train_sizes=np.linspace(0.1, 1.0, 50), cv=cv)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)

test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, color="r", alpha=0.1)
plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, color="g", alpha=0.1)
plt.plot(train_sizes, train_mean, 'o-', color="r", label="Training score")
plt.plot(train_sizes, test_mean, 'o-', color="g", label="Cross-validation score")

plt.title("Learning Curve")
plt.xlabel("Training examples")
plt.ylabel("Score")
plt.legend(loc="best")
plt.show()
```



Is there anything we can conclude from this graph?

In [66]:
```
# Accuracy is higher than that of the random classifier
```

1.8 Step 8: Parameter optimization via cross-validation

When we optimize parameters with a grid search, we choose the parameters that give the best test scores. This is different from what would happen with new data - to do this fairly, at no point of the training procedure we are allowed to look at the test labels. Therefore, we would need to do **nested cross validation** to avoid leakage between the parameter optimization and the cross validation procedure and properly evaluate the generalization error. For now, we are just looking for the best model so "simple" CV is sufficient.

In [67]:
```
from sklearn.pipeline import make_pipeline

# Given:

piped_model = make_pipeline(StandardScaler(), SVC()) #now using the general SVC so I can change the kernel
piped_model.get_params() #this shows how we can access parameters both for the scaler and the classifier
```

Out[67]:
```
{'memory': None,
 'steps': [('standardscaler', StandardScaler()), ('svc', SVC())],
 'verbose': False,
 'standardscaler': StandardScaler(),
 'svc': SVC(),
 'standardscaler__copy': True,
 'standardscaler__with_mean': True,
 'standardscaler__with_std': True,
 'svc__C': 1.0,
 'svc__break_ties': False,
 'svc__cache_size': 200,
 'svc__class_weight': None,
 'svc__coef0': 0.0,
 'svc__decision_function_shape': 'ovr',
 'svc__degree': 3,
 'svc__gamma': 'scale',
 'svc__kernel': 'rbf',
 'svc__max_iter': -1,
 'svc__probability': False,
 'svc__random_state': None,
 'svc__shrinking': True,
 'svc__tol': 0.001,
 'svc__verbose': False}
```

We can define a dictionary of parameter values to run the optimization.

Note that this might take awhile (5-15 minutes); the early estimates output by this cell may be misleading because more complex models (in particular high gamma) take longer.

Once you run this cell, the "model" object will have attributes "best_score_", "best_params_" and "best_estimator_", which give us access to the optimal estimator (printed out), as well as "cv_results_" that can be used to visualize the performance of all models.

In [68]:
```
# Given:

#optimizing SVC: THIS IS NOT YET NESTED CV

parameters = {'svc__kernel':['poly', 'rbf'], \
              'svc__gamma':['scale', 0.01, 0.1], 'svc__C':[0.1, 1.0, 10.0, 100.0], \
              'svc__degree': [2, 4]}

model = GridSearchCV(piped_model, parameters, cv = StratifiedKFold(n_splits=5, shuffle=True), \
                     verbose = 4, n_jobs = -1, return_train_score=True)

model.fit(features_lim,target)

print('Best params, best score:', "{:.4f}".format(model.best_score_), \
      model.best_params_)
```

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits
Best params, best score: 0.8958 {'svc__C': 1.0, 'svc__degree': 2, 'svc__gamma': 'scale', 'svc__kernel': 'rbf'}
```

1.9 Step 9

Next, we visualize the models in a pandas data frame, and rank them according to their test scores.

You may find it useful to look at:

1. the mean and std of the test scores
2. the mean of the train scores (to evaluate if they differ and the significance of the result)
3. fitting time (we can pick a faster model instead of the best model if the scores are comparable)!

Let "scores_lim" be the dataframe containing "model.cv_results_". Print the columns.

In [74]:
```
scores_lim = pd.DataFrame(data = model.cv_results_)
```

Next, in "scores_lim", sort the columns 'params','mean_test_score','std_test_score','mean_train_score', and 'mean_fit_time' by descending 'mean_test_score':

In [76]:
```
# Sort the DataFrame by 'mean_test_score' in descending order
sorted_df = scores_lim[['params','mean_test_score','std_test_score','mean_train_score','mean_fit_time']].sort_values('mean_test_score', ascending=False)
sorted_df
```

Out[76]:

	params	mean_test_score	std_test_score	mean_train_score	mean_fit_time
13	{'svc__C': 1.0, 'svc__degree': 2, 'svc__gamma': 'scale', 'svc__kernel': 'rbf'}	0.8958	0.004069	0.92175	0.620844
19	{'svc__C': 1.0, 'svc__degree': 4, 'svc__gamma': 'scale', 'svc__kernel': 'rbf'}	0.8958	0.004069	0.92175	0.612268
17	{'svc__C': 1.0, 'svc__degree': 2, 'svc__gamma': 0.1, 'svc__kernel': 'rbf'}	0.8948	0.008158	0.93915	0.745116
23	{'svc__C': 1.0, 'svc__degree': 4, 'svc__gamma': 0.1, 'svc__kernel': 'rbf'}	0.8948	0.008158	0.93915	0.860153
33	{'svc__C': 10.0, 'svc__degree': 4, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}	0.8932	0.007305	0.91150	0.695273
27	{'svc__C': 10.0, 'svc__degree': 2, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}	0.8932	0.007305	0.91150	0.718267
15	{'svc__C': 1.0, 'svc__degree': 2, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}	0.8928	0.003544	0.90045	0.523927
21	{'svc__C': 1.0, 'svc__degree': 4, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}	0.8928	0.003544	0.90045	0.523799
39	{'svc__C': 100.0, 'svc__degree': 2, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}	0.8870	0.007668	0.93375	1.189824
45	{'svc__C': 100.0, 'svc__degree': 4, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}	0.8870	0.007668	0.93375	1.229300
9	{'svc__C': 0.1, 'svc__degree': 4, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}	0.8840	0.003098	0.88515	0.533510
3	{'svc__C': 0.1, 'svc__degree': 2, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}	0.8840	0.003098	0.88515	0.591845
7	{'svc__C': 0.1, 'svc__degree': 4, 'svc__gamma': 'scale', 'svc__kernel': 'rbf'}	0.8834	0.006711	0.88960	0.569997
1	{'svc__C': 0.1, 'svc__degree': 2, 'svc__gamma': 'scale', 'svc__kernel': 'rbf'}	0.8834	0.006711	0.88960	0.601255
31	{'svc__C': 10.0, 'svc__degree': 4, 'svc__gamma': 'scale', 'svc__kernel': 'rbf'}	0.8824	0.008913	0.96660	1.001792
25	{'svc__C': 10.0, 'svc__degree': 2, 'svc__gamma': 'scale', 'svc__kernel': 'rbf'}	0.8824	0.008913	0.96660	0.979785
35	{'svc__C': 10.0, 'svc__degree': 4, 'svc__gamma': 0.1, 'svc__kernel': 'rbf'}	0.8772	0.007467	0.99170	1.021833
29	{'svc__C': 10.0, 'svc__degree': 2, 'svc__gamma': 0.1, 'svc__kernel': 'rbf'}	0.8772	0.007467	0.99170	1.129024
12	{'svc__C': 1.0, 'svc__degree': 2, 'svc__gamma': 'scale', 'svc__kernel': 'poly'}	0.8768	0.007054	0.88325	0.800606
16	{'svc__C': 1.0, 'svc__degree': 2, 'svc__gamma': 0.1, 'svc__kernel': 'poly'}	0.8760	0.008414	0.88625	1.115305
38	{'svc__C': 100.0, 'svc__degree': 2, 'svc__gamma': 0.01, 'svc__kernel': 'poly'}	0.8760	0.008414	0.88625	1.084420
40	{'svc__C': 100.0, 'svc__degree': 2, 'svc__gamma': 0.1, 'svc__kernel': 'poly'}	0.8756	0.008333	0.88970	49.343346
36	{'svc__C': 100.0, 'svc__degree': 2, 'svc__gamma': 'scale', 'svc__kernel': 'poly'}	0.8750	0.008764	0.88995	16.026693
28	{'svc__C': 10.0, 'svc__degree': 2, 'svc__gamma': 0.1, 'svc__kernel': 'poly'}	0.8750	0.008764	0.88925	5.804421
24	{'svc__C': 10.0, 'svc__degree': 2, 'svc__gamma': 'scale', 'svc__kernel': 'poly'}	0.8748	0.009368	0.88815	2.447300
4	{'svc__C': 0.1, 'svc__degree': 2, 'svc__gamma': 0.1, 'svc__kernel': 'poly'}	0.8748	0.007808	0.87985	0.664503
26	{'svc__C': 10.0, 'svc__degree': 2, 'svc__gamma': 0.01, 'svc__kernel': 'poly'}	0.8748	0.007808	0.87985	0.744262
0	{'svc__C': 0.1, 'svc__degree': 2, 'svc__gamma': 'scale', 'svc__kernel': 'poly'}	0.8716	0.005083	0.87430	0.653073
10	{'svc__C': 0.1, 'svc__degree': 4, 'svc__gamma': 0.1, 'svc__kernel': 'poly'}	0.8716	0.007915	0.91740	0.912046
18	{'svc__C': 1.0, 'svc__degree': 4, 'svc__gamma': 'scale', 'svc__kernel': 'poly'}	0.8714	0.007392	0.91680	0.998190
30	{'svc__C': 10.0, 'svc__degree': 4, 'svc__gamma': 'scale', 'svc__kernel': 'poly'}	0.8704	0.004587	0.96175	1.281655
22	{'svc__C': 1.0, 'svc__degree': 4, 'svc__gamma': 0.1, 'svc__kernel': 'poly'}	0.8704	0.004630	0.96235	1.449012
41	{'svc__C': 100.0, 'svc__degree': 2, 'svc__gamma': 0.1, 'svc__kernel': 'rbf'}	0.8674	0.003611	1.00000	1.233072
47	{'svc__C': 100.0, 'svc__degree': 4, 'svc__gamma': 0.1, 'svc__kernel': 'rbf'}	0.8674	0.003611	1.00000	1.426847
14	{'svc__C': 1.0, 'svc__degree': 2, 'svc__gamma': 0.01, 'svc__kernel': 'poly'}	0.8662	0.005269	0.86690	0.600342
37	{'svc__C': 100.0, 'svc__degree': 2, 'svc__gamma': 'scale', 'svc__kernel': 'rbf'}	0.8660	0.004775	0.99700	1.495460
43	{'svc__C': 100.0, 'svc__degree': 4, 'svc__gamma': 'scale', 'svc__kernel': 'rbf'}	0.8660	0.004775	0.99700	1.553468
44	{'svc__C': 100.0, 'svc__degree': 4, 'svc__gamma': 0.01, 'svc__kernel': 'poly'}	0.8652	0.004874	0.88670	0.831908
6	{'svc__C': 0.1, 'svc__degree': 4, 'svc__gamma': 'scale', 'svc__kernel': 'poly'}	0.8650	0.004980	0.88600	0.696102
11	{'svc__C': 0.1, 'svc__degree': 4, 'svc__gamma': 0.1, 'svc__kernel': 'rbf'}	0.8592	0.003970	0.86590	0.684720
5	{'svc__C': 0.1, 'svc__degree': 2, 'svc__gamma': 0.1, 'svc__kernel': 'rbf'}	0.8592	0.003970	0.86590	0.712996
32	{'svc__C': 10.0, 'svc__degree': 4, 'svc__gamma': 0.01, 'svc__kernel': 'poly'}	0.8588	0.004445	0.86425	0.773347
20	{'svc__C': 1.0, 'svc__degree': 4, 'svc__gamma': 0.01, 'svc__kernel': 'poly'}	0.8508	0.002786	0.85295	0.679017
2	{'svc__C': 0.1, 'svc__degree': 2, 'svc__gamma': 0.01, 'svc__kernel': 'poly'}	0.8494	0.002059	0.85030	0.589906
42	{'svc__C': 100.0, 'svc__degree': 4, 'svc__gamma': 'scale', 'svc__kernel': 'poly'}	0.8484	0.005389	0.99600	1.652690
34	{'svc__C': 10.0, 'svc__degree': 4, 'svc__gamma': 0.1, 'svc__kernel': 'poly'}	0.8478	0.005492	0.99620	1.659379
8	{'svc__C': 0.1, 'svc__degree': 4, 'svc__gamma': 0.01, 'svc__kernel': 'poly'}	0.8416	0.003007	0.84230	0.641696
46	{'svc__C': 100.0, 'svc__degree': 4, 'svc__gamma': 0.1, 'svc__kernel': 'poly'}	0.8390	0.005899	1.00000	1.639175

To build some intuition around the results, I find it helpful to ask: what hyperparameter values are common to all the best-performing models? Here, for example, the rbf kernel seems to be constantly preferred, while the values of C and gamma seem to only affect the scores only mildly. Note also that the Grid Search is insensitive to moot parameters combinations; for example, here the first three models are identical, because the degree of the polynomial kernel does not matter when using an rbf kernel. This is less than ideal, of course.

1.9.1 Final diagnosis

The problem here is high bias, which is not that surprising given that we are using only a subset of features.

We can try two things: making up new features which might help, based on what we know about the problem, and using an imputing strategy to include information about the discarded features.

1.9.2 Step 10: Improve the model.

In order to include additional features, use .fillna(0) to replace all nan with zero, and .replace('',0) to replace empty string values. Check the result.

In [79]:
```
features = X.fillna(0)
features.replace('',0,inplace=True)
```

1.9.2.1 Let's start by looking at what kind of particles we have as a product of the collision.

In [80]:
```
np.unique(np.array([features['Type_'+str(i)].values for i in range(1,14)]).astype('str'))
```

Out[80]:
```
array(['0', 'b', 'e+', 'e-', 'g', 'j', 'm+', 'm-'], dtype='<U2')
```

1.9.2.2 Here are the proposed new features (justification can be found in Chapter 4).

1. The total number of particles produced
2. The total number of b jets
3. The total number of jets
4. The total number of leptons (electrons, positron, mu+, mu-)

In [81]:
```
#count number of non-zero types

ntot = np.array([-(np.sum(np.array([features['Type_'+str(i)].values[j] == 0 for i in range(1,14)])) - 13) for j in range(features.shape[0])])
```

In [82]:
```
#count number of b jets

nbtot = np.array([np.sum(np.array([features['Type_'+str(i)].values[j] == 'b' for i in range(1,14)])) for j in range(features.shape[0])])
```

In [83]:
```
#Actually, let's count all types (jets, photons g, e-, e+, mu-, mu+)

njtot = np.array([np.sum(np.array([features['Type_'+str(i)].values[j] == 'j' for i in range(1,14)])) for j in range(features.shape[0])])
```

In [84]:
```
ngtot = np.array([np.sum(np.array([features['Type_'+str(i)].values[j] == 'g' for i in range(1,14)])) for j in range(features.shape[0])])
```

In [85]:
```
n_el_tot = np.array([np.sum(np.array([features['Type_'+str(i)].values[j] == 'e-' for i in range(1,14)])) for j in range(features.shape[0])])
```

In [86]:
```
n_pos_tot = np.array([np.sum(np.array([features['Type_'+str(i)].values[j] == 'e+' for i in range(1,14)])) for j in range(features.shape[0])])
```

In [87]:
```
n_muneg_tot = np.array([np.sum(np.array([features['Type_'+str(i)].values[j] == 'm-' for i in range(1,14)])) for j in range(features.shape[0])])
```

In [88]:
```
n_mupos_tot = np.array([np.sum(np.array([features['Type_'+str(i)].values[j] == 'm+' for i in range(1,14)])) for j in range(features.shape[0])])
```

In [89]:
```
n_lepton_tot = n_el_tot + n_pos_tot + n_muneg_tot + n_mupos_tot
```

Add these new features to the dataset and check the result. How many features are there now?

In [92]:
```
# After adding the data we get a total of 73 features
```

1.9.3 Feature engineering: impact of ad-hoc variables

Define 'features_lim_2' as the original 'features_lim' plus the new five hand-crafted features. Apply the piped model (StandardScaler plus LinearSVC) to this new dataset via cross_validate. Compute the mean and standard deviation of the test scores. Any improvements?

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

Note: Knowledge-informed feature engineering is often very successful, more than hyperparameter optimization. Machine learning methods are often tooted for their ability to learn relevant representations, but non-deep-learning methods are less capable to do so, and providing informative features is very helpful.

We can optimize this model as well; it will take a while, just like the previous time.

In [71]:
```
#optimizing SVC: Takes a few minutes!
piped_model = make_pipeline(StandardScaler(), SVC())

parameters = {'svc__kernel':['poly', 'rbf'], \
              'svc__gamma':['scale', 0.01, 0.1], 'svc__C':[0.1, 1.0, 10.0], 'svc__degree': [2, 4, 8]}

nmodels = np.product([len(el) for el in parameters.values()])
model = GridSearchCV(piped_model, parameters, cv = StratifiedKFold(n_splits=5, shuffle=True), \
                     verbose = 2, n_jobs = -1, return_train_score=True)
model.fit(features_lim_2,target)

print('Best params, best score:', "{:.4f}".format(model.best_score_), \
      model.best_params_)
```

```
Fitting 5 folds for each of 54 candidates, totalling 270 fits
Best params, best score: 0.9462 {'svc__C': 1.0, 'svc__degree': 2, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

In [ ]:

1.9.4 Take-home message: feature engineering often works best if we use subject matter knowledge, and building more features is not necessarily better.