Particle Collision Classification using Support Vector Machines

Introduction

This notebook explores the application of Support Vector Machines (SVM) to classify particle collisions from the HIGGS dataset. The dataset contains simulated collision events, with each event either being a signal process producing Higgs bosons (class 1) or a background process (class 0).

Dataset Overview

The HIGGS dataset contains 28 features derived from particle physics measurements, including:

- Low-level features: Raw measurements like particle momentum (pT), pseudorapidity (eta), and azimuthal angle (phi)
- High-level features: Derived quantities like invariant masses of particle combinations

SVM Classification Approach

We'll use Support Vector Machines with different kernel functions to classify these collision events:

- Linear kernel: For linearly separable data
- RBF (Gaussian) kernel: For non-linear decision boundaries
- Polynomial kernel: For capturing higher-order feature interactions
- Custom hybrid kernel: Combining multiple kernels to leverage different aspects of the data

The choice of kernel is crucial as it determines how the SVM algorithm maps the input features to a higher-dimensional space where the classes become separable. We'll experiment with various kernel parameters to optimize the classification performance.

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest, f_classif
```

Reading the dataset

The dataset

The dataset HIGGs does not have the column headings, which means we need to set "header = None" so that we do not lose one data point when converting to pandas dataframe

In [3]:
```
# The following lines were used to unzip the dataset and convert to csv (need no
# uploaded files has the smaller version of it)

# df = pd.read_csv('HIGGS.csv', header = None)
# df

# read only the target column of the dataset
# y_main = pd.read_csv('../HIGGS.csv', header = None, usecols=[0])
# y_main.columns = ['target']
```

In [4]:
```
# column_names = ['class',
#     'lepton_pT', 'lepton_eta', 'lepton_phi', 'missing_energy_magnitude', 'miss
#     'jet_1_pt', 'jet_1_eta', 'jet_1_phi', 'jet_1_btag',
#     'jet_2_pt', 'jet_2_eta', 'jet_2_phi', 'jet_2_btag',
#     'jet_3_pt', 'jet_3_eta', 'jet_3_phi', 'jet_3_btag',
#     'jet_4_pt', 'jet_4_eta', 'jet_4_phi', 'jet_4_btag',
#     'm_jj', 'm_jjj', 'm_lv', 'm_jlv', 'm_bb', 'm_wbb', 'm_wwbb'
# ]

# df.columns = column_names
```

Exploratory Data Analysis and Preprocessing

Dataset Sampling

**Stratified Sampling* - Stratified random sampling is a method of sampling that involves the division of a population into smaller subgroups known as strata. In stratified random sampling, or stratification, the strata are formed based on members’ shared attributes or characteristics. Stratified sampling is used to highlight differences among groups in a population. This is different from simple random sampling, which treats all members of a population as equal, with an equal likelihood of being sampled.

In the following commented code (using which initially the entire HIGGS dataset was contracted to smaller 1% set), we first group the instances by class and within them, sample 1% of points (essentially what stratified sampling does).

In [5]:
```
# df = df.groupby('class').apply(lambda x: x.sample(frac=.01, random_state=42)).
# df.to_csv('HIGGS_smaller_updated.csv', index = False)
```

In [13]:
```
df = pd.read_csv('HIGGS_smaller_updated.csv')
df
```

Out[13]:

	class	lepton_pT	lepton_eta	lepton_phi	missing_energy_magnitude	missing_er
0	0.0	0.795907	0.521993	1.266055	1.193820	
1	0.0	0.439039	-0.246468	-1.633201	2.165616	
2	0.0	0.792429	-0.225041	0.476488	0.216189	
3	0.0	2.508689	0.605754	0.057012	1.908093	
4	0.0	0.956040	-0.425679	0.380497	0.382192	
...	...	...	...	...	...	
109995	1.0	1.326267	-1.027591	-0.629456	1.147094	
109996	1.0	0.534753	-1.201932	-0.766507	0.721336	
109997	1.0	0.524138	1.794970	0.293383	0.789934	
109998	1.0	0.971595	0.186948	-0.499616	1.675635	
109999	1.0	0.649865	0.728474	-1.178214	2.513985	

110000 rows × 29 columns

Constants

The section below sets all the constants that might be required in the notebook.

In [14]:
```
RANDOM_STATE = 42
y_col = "class"
```

Quick Feature Analysis

Data Types

Verifying whether all the data points have numerical values (since the original data did not have header)

In [15]:
```
df.dtypes
```

Out[15]:

	0
class	float64
lepton_pT	float64
lepton_eta	float64
lepton_phi	float64
missing_energy_magnitude	float64
missing_energy_phi	float64
jet_1_pt	float64
jet_1_eta	float64
jet_1_phi	float64
jet_1_btag	float64
jet_2_pt	float64
jet_2_eta	float64
jet_2_phi	float64
jet_2_btag	float64
jet_3_pt	float64
jet_3_eta	float64
jet_3_phi	float64
jet_3_btag	float64
jet_4_pt	float64
jet_4_eta	float64
jet_4_phi	float64
jet_4_btag	float64
m_jj	float64
m_jjj	float64
m_lv	float64
m_jlv	float64
m_bb	float64
m_wbb	float64
m_wwbb	float64

dtype: object

Missing Value Analysis

None of the columns have missing values, hence no need to perform any data imputations

```
In [16]: df.isna().sum()
```

Out[16]:

	0
class	0
lepton_pT	0
lepton_eta	0
lepton_phi	0
missing_energy_magnitude	0
missing_energy_phi	0
jet_1_pt	0
jet_1_eta	0
jet_1_phi	0
jet_1_btag	0
jet_2_pt	0
jet_2_eta	0
jet_2_phi	0
jet_2_btag	0
jet_3_pt	0
jet_3_eta	0
jet_3_phi	0
jet_3_btag	0
jet_4_pt	0
jet_4_eta	0
jet_4_phi	0
jet_4_btag	0
m_jj	0
m_jjj	0
m_lv	0
m_jlv	0
m_bb	0
m_wbb	0
m_wwbb	0

dtype: int64

Target Variable Analysis

Class with value 1 represents SIGNAL (53% of the data), and Class with value 0 represents BACKGROUND (47% of the data)

Looks like there is very little imbalance in data.

In [17]:
```
df[y_col].value_counts() / len(df)
```

Out[17]:

class	count
1.0	0.529918
0.0	0.470082

dtype: float64

Simple Statistcal Analysis

Using the `describe` function we can see some simple statistics of all the columns. Given there are so many columns, it is hard to see understand the distrubtion and what each columns represent. Hence we will then move to plots to understand better.

In [18]:
```
df.describe()
```

Out[18]:

	class	lepton_pT	lepton_eta	lepton_phi	missing_energy_mag
count	110000.000000	110000.000000	110000.000000	110000.000000	110000
mean	0.529918	0.990136	-0.005788	0.001553	1
std	0.499106	0.566492	1.008337	1.005915	0
min	0.000000	0.274697	-2.433028	-1.742508	0
25%	0.000000	0.588740	-0.744166	-0.870266	0
50%	1.000000	0.851724	-0.006872	0.000971	0
75%	1.000000	1.233481	0.732370	0.875988	1
max	1.000000	8.170258	2.434868	1.743236	9

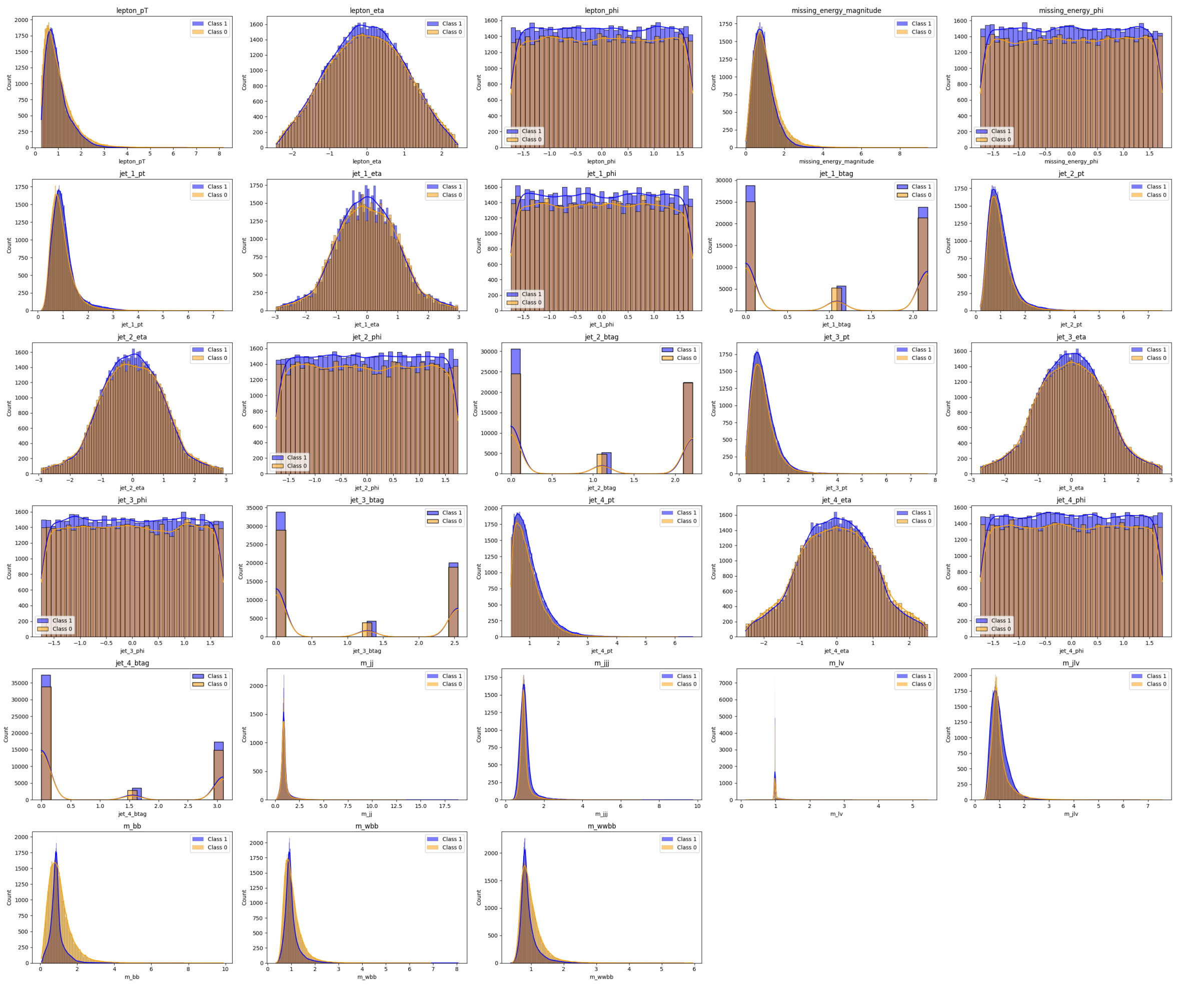
8 rows × 29 columns

Plots and Outlier Handling

Histogram Plots

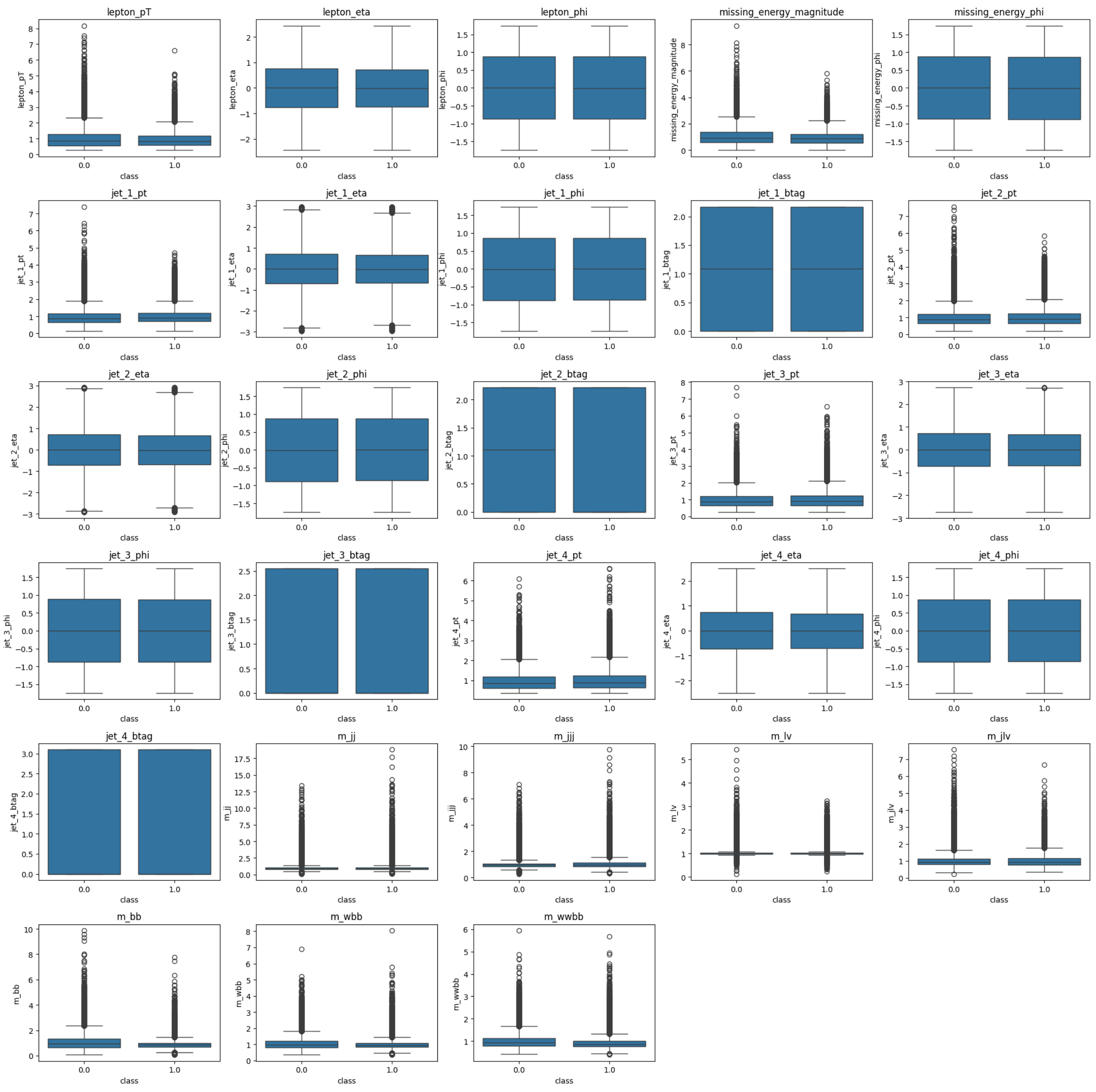
Given all the fields are numerical, it makes sense to first see what the data distribution (histogram) looks like. We plot the histograms in differnt colors for different class to see if there is anything we can understand from the data.

In [19]:
```
# Plot distributions using seaborn for better visualization
total_columns = len(df.columns) - 1
num_cols = 5
num_rows = (total_columns // num_cols) + 1

plt.figure(figsize=(30, 25))
for i, col in enumerate(df.drop(columns=[y_col]).columns):
    plt.subplot(num_rows, num_cols, i+1)

    # Using seaborn's kdeplot for smoother distribution visualization
    sns.histplot(data=df[df[y_col] == 1][col], color='blue', label='Class 1', fi
    sns.histplot(data=df[df[y_col] == 0][col], color='orange', label='Class 0',

    plt.title(col, fontsize=12)
    plt.legend()

plt.tight_layout()
plt.show()
```



The plots tell us that within the features, an instance is classified almost uniformly. By this, it means that there are no "outstanding" features or feature values, that "prefer" one class over another. Thus, we can perform outlier removal techniques since removing the extreme points will not let us lose any categorizable information.

In [20]:
```
# conditional boxplots of all the features with respect to the classes 0 and 1
plt.figure(figsize=(20, 20))
for i, col in enumerate(df.drop(columns=[y_col]).columns):
    plt.subplot(6, 5, i+1)
    sns.boxplot(x='class', y=col, data=df)
    plt.title(col)
plt.tight_layout()
plt.show()
```



The box plots of the features conditional on the class show that the features have no charactersitic differences between the classes

We can also see that nearly half the features have values outside the whiskers of the box plots, but the classes are still almost uniformly distributed. Since these points are extreme points, we can remove them and continue.

We can apply outlier removal for the features that have more points around the median and very few outside the whiskers. The technique used will be Z-score. Using this, we find that the number of rows reduce from 110000 to 89387.

**Z-score*: Z-score is a statistical measurement that describes a value's relationship to the mean of a group of values. Z-score is measured in terms of standard deviations from the mean.

In [21]:
```
skewed_columns = []
for col in df.columns:
    if 'btag' in col:
        continue
    t = df[col].skew()

    if t > 1.5:
        skewed_columns.append(col)

print(skewed_columns)
```

```
['lepton_pT', 'missing_energy_magnitude', 'jet_1_pt', 'jet_2_pt', 'jet_3_pt', 'je
t_4_pt', 'm_jj', 'm_jjj', 'm_lv', 'm_jlv', 'm_bb', 'm_wbb', 'm_wwbb']
```

In [22]:

```
# Function to remove outliers based on z-score
def remove_outliers_zscore(df, column, threshold=3):
    z_scores = (df[column] - df[column].mean()) / df[column].std()
    return df[abs(z_scores) < threshold]

# Apply outlier removal to these columns
for col in skewed_columns:
    df = remove_outliers_zscore(df, col)

df = df.reset_index(drop = True)

df
```

Out[22]:

	class	lepton_pT	lepton_eta	lepton_phi	missing_energy_magnitude	missing_ene
0	0.0	0.795907	0.521993	1.266055	1.193820	-1
1	0.0	0.439039	-0.246468	-1.633201	2.165616	-1
2	0.0	0.792429	-0.225041	0.476488	0.216189	-(
3	0.0	2.508689	0.605754	0.057012	1.908093	(
4	0.0	0.956040	-0.425679	0.380497	0.382192	1
...	...	...	...	...	...	
89382	1.0	1.367993	-0.729557	0.365515	0.409989	(
89383	1.0	1.326267	-1.027591	-0.629456	1.147094	-(
89384	1.0	0.534753	-1.201932	-0.766507	0.721336	-1
89385	1.0	0.524138	1.794970	0.293383	0.789934	1
89386	1.0	0.971595	0.186948	-0.499616	1.675635	-(

89387 rows × 29 columns

In [23]:

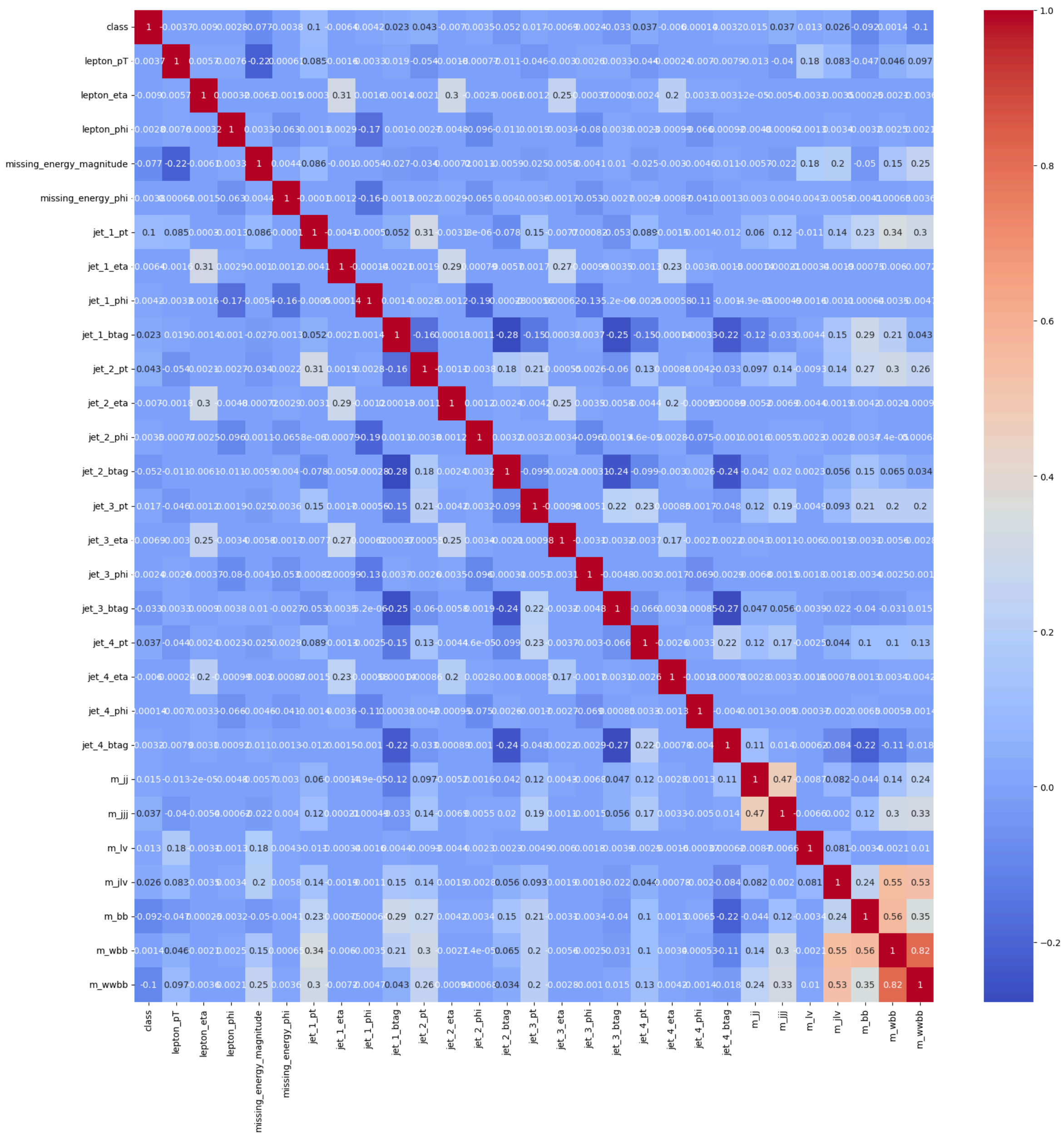
```
# splitting the data into features and target
X = df.drop(columns = ['class'])
y = df[['class']]

X.head()
```

Out[23]:

	lepton_pT	lepton_eta	lepton_phi	missing_energy_magnitude	missing_energy_phi	j
0	0.795907	0.521993	1.266055	1.193820	-1.345527	1.
1	0.439039	-0.246468	-1.633201	2.165616	-1.280181	1.
2	0.792429	-0.225041	0.476488	0.216189	-0.653596	0.
3	2.508689	0.605754	0.057012	1.908093	0.369292	1.
4	0.956040	-0.425679	0.380497	0.382192	1.287647	1.

5 rows × 28 columns

In [24]:
```
# correlation matrix
corr_matrix = df.corr()

plt.figure(figsize=(20, 20))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.show()
```



As evident in the correlation matrix, in general the features are not correlated with each other, but m_wbb and m_wwbb are highly correlated with each other which means we can drop one of them (choosing wbb arbitrarily)

In [25]:
```
# drop the column m_wbb
X_reduced = pd.DataFrame(X, columns = X.drop(columns=['m_wbb']).columns)

# X_reduced.head()
```

Feature Engineering

Feature Engineering for this project involves creating some new features to gain a few insights on how the features are affecting the classification. Here, the features that involve transverse momentum (pT), and the angles (eta, phi) have been added to create one new feature.

Another feature was added that combined 2 invariant masses m_bb and m_wwbb. We will see if these features add some improvement in the correlation or not.

In [26]:
```
X_feat_eng = X_reduced.copy()

# adding the sum of the transverse momenta of the jets
X_feat_eng['sum_jet_pt'] = X_feat_eng['jet_1_pt'] + X_feat_eng['jet_2_pt'] + X_f

# adding the sum of the btag values of the jets
X_feat_eng['sum_jet_eta'] = X_feat_eng['jet_1_eta'] + X_feat_eng['jet_2_eta'] +
X_feat_eng['sum_jet_phi'] = X_feat_eng['jet_1_phi'] + X_feat_eng['jet_2_phi'] +

X_feat_eng['mass_combination'] = X_feat_eng['m_bb'] + X_feat_eng['m_wwbb']
X_feat_eng['missing_energy_lepton_diff'] = X_feat_eng['missing_energy_magnitude'

X_feat_eng.head()
```

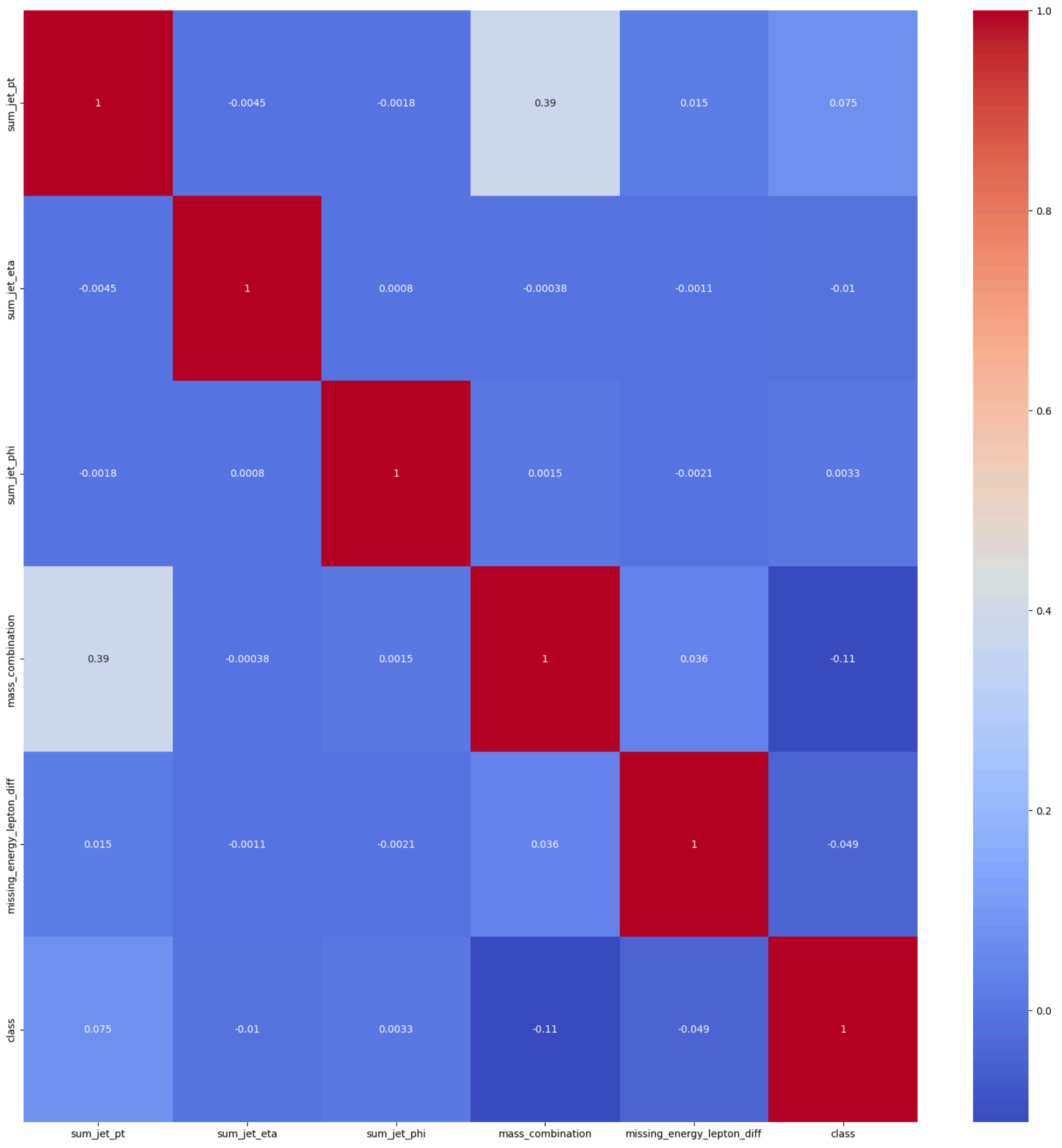
Out[26]:

	lepton_pT	lepton_eta	lepton_phi	missing_energy_magnitude	missing_energy_phi	j
0	0.795907	0.521993	1.266055	1.193820	-1.345527	1.
1	0.439039	-0.246468	-1.633201	2.165616	-1.280181	1.
2	0.792429	-0.225041	0.476488	0.216189	-0.653596	0.
3	2.508689	0.605754	0.057012	1.908093	0.369292	1.
4	0.956040	-0.425679	0.380497	0.382192	1.287647	1.

5 rows × 32 columns

In [27]:
```
# see correlation of new features with the target
# create a temporary df to store only the new features and the target
temp_df = X_feat_eng.iloc[:, 27:33]
temp_df['class'] = y

corr_matrix = temp_df.corr()
```

```
plt.figure(figsize=(20, 20))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.show()
```



In [28]:
```
# the correlation seems to improve a little for sum_jet_pt and mass_combination
# thus we can keep these features and drop the rest

X_feat_eng = X_feat_eng.drop(columns=['sum_jet_eta', 'sum_jet_phi', 'missing_ene

X_feat_eng.head()
```

Out[28]:

	lepton_pT	lepton_eta	lepton_phi	missing_energy_magnitude	missing_energy_phi	j
0	0.795907	0.521993	1.266055	1.193820	-1.345527	1.
1	0.439039	-0.246468	-1.633201	2.165616	-1.280181	1.
2	0.792429	-0.225041	0.476488	0.216189	-0.653596	0.
3	2.508689	0.605754	0.057012	1.908093	0.369292	1.
4	0.956040	-0.425679	0.380497	0.382192	1.287647	1.

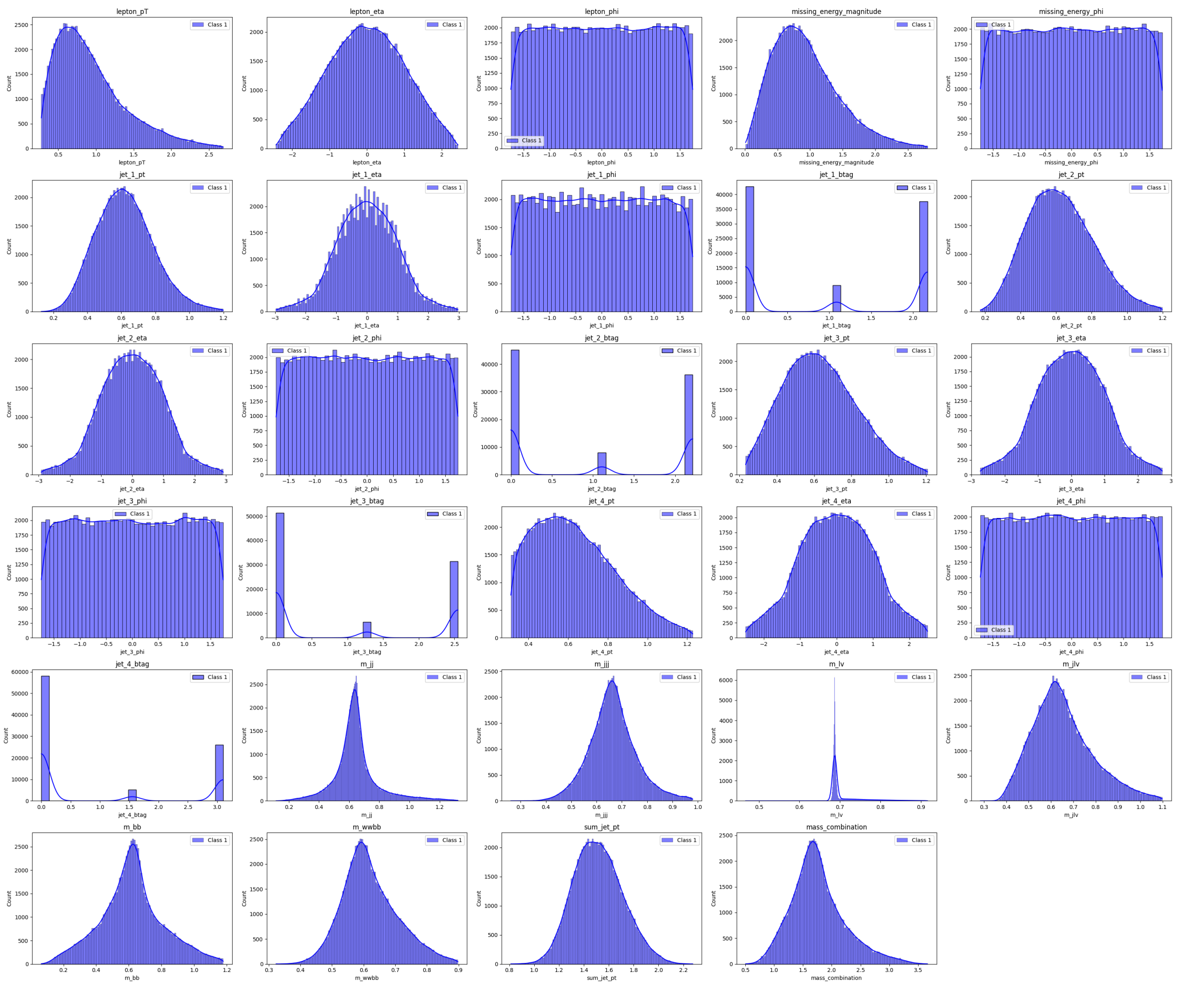
5 rows × 29 columns

In [29]:

```
# Apply log-transformation to the feautres having either "pt" or "m_" in their n
log_transform_columns = [col for col in X_feat_eng.columns if (('pt' in col and

#calculate the skewness of the columns
skewness = X_feat_eng[log_transform_columns].apply(lambda x: x.skew())

X_log_transformed = X_feat_eng.copy()

X_log_transformed[log_transform_columns] = np.log1p(X_feat_eng[log_transform_col

#calculate the skewness of the columns after log transformation
skewness_after = X_log_transformed[log_transform_columns].apply(lambda x: x.skew

print("Skewness before log transformation: \n", skewness)
print("\nSkewness after log transformation: \n", skewness_after)
```

```
Skewness before log transformation:
 jet_1_pt      0.881885
jet_2_pt      0.832616
jet_3_pt      0.825316
jet_4_pt      0.981074
m_jj          1.935039
m_jjj         0.645414
m_lv          2.455376
m_jlv         1.028096
m_bb          0.783329
m_wwbb        0.734528
sum_jet_pt    0.774100
dtype: float64

Skewness after log transformation:
 jet_1_pt      0.358661
jet_2_pt      0.343230
jet_3_pt      0.353184
jet_4_pt      0.543047
m_jj          0.964680
m_jjj         0.279852
m_lv          2.311788
m_jlv         0.643812
m_bb          0.160005
m_wwbb        0.480200
sum_jet_pt    0.221756
dtype: float64
```

In [30]:
```
# visualize the distributions of the features after the changes

plt.figure(figsize=(30, 25))
for i, col in enumerate(X_log_transformed.columns):
    plt.subplot(num_rows, num_cols, i+1)

    # Using seaborn's kdeplot for smoother distribution visualization
    sns.histplot(data=X_log_transformed[col], color='blue', label='Class 1', fil

    plt.title(col, fontsize=12)
    plt.legend()

plt.tight_layout()
plt.show()
```



Data Normalization

We will normalize the data such that mean is 0 and standard deviation is 1

In [31]:
```
for col in X_log_transformed.columns:
    X_log_transformed[col] = (X_log_transformed[col] - X_log_transformed[col].me

X_log_transformed.describe()
```

Out[31]:

	lepton_pT	lepton_eta	lepton_phi	missing_energy_magnitude	missing
count	8.938700e+04	8.938700e+04	8.938700e+04	8.938700e+04	8
mean	-1.317954e-16	-1.399035e-17	-1.764692e-17	-1.310005e-16	
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	
min	-1.413199e+00	-2.402162e+00	-1.734294e+00	-1.815803e+00	
25%	-7.523984e-01	-7.263369e-01	-8.675218e-01	-7.419861e-01	
50%	-2.236805e-01	-2.202859e-03	-6.450224e-04	-1.540092e-01	
75%	5.136500e-01	7.286807e-01	8.705380e-01	5.761204e-01	
max	3.712374e+00	2.417041e+00	1.729591e+00	3.653865e+00	

8 rows × 29 columns

Feature Selection

We will use the SelectKBest as our selection algorithm

**SelectKBest*: It is a filter-based feature selection method, which relies on statistical measures to score and rank the features. It uses statistical tests like chi-squared test, ANOVA F-test, or mutual information score. Then, it selects the K features with the highest scores to be included in the final feature subset.

In [32]:
```
# using f_classif as the scoring function
selector = SelectKBest(f_classif, k=20)
X_new = selector.fit_transform(X_log_transformed, y)

X_new = pd.DataFrame(X_new, columns=X_log_transformed.columns[selector.get_suppo

# get the selected features
selected_features = X_log_transformed.columns[selector.get_support()]
selected_features
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/utils/validation.py:1339: DataCon
versionWarning: A column-vector y was passed when a 1d array was expected. Please
change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

Out[32]:
```
Index(['lepton_eta', 'missing_energy_magnitude', 'jet_1_pt', 'jet_1_eta',
       'jet_1_btag', 'jet_2_pt', 'jet_2_eta', 'jet_2_btag', 'jet_3_pt',
       'jet_3_eta', 'jet_3_btag', 'jet_4_pt', 'm_jj', 'm_jjj', 'm_lv', 'm_jlv',
       'm_bb', 'm_wwbb', 'sum_jet_pt', 'mass_combination'],
      dtype='object')
```

Model Training

In [33]:
```
!pip install scikit-optimize
```

```
Collecting scikit-optimize
  Downloading scikit_optimize-0.10.2-py2.py3-none-any.whl.metadata (9.7 kB)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.10/dist-pac
kages (from scikit-optimize) (1.4.2)
Collecting pyaml>=16.9 (from scikit-optimize)
  Downloading pyaml-24.9.0-py3-none-any.whl.metadata (11 kB)
Requirement already satisfied: numpy>=1.20.3 in /usr/local/lib/python3.10/dist-pa
ckages (from scikit-optimize) (1.26.4)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.10/dist-pac
kages (from scikit-optimize) (1.13.1)
Requirement already satisfied: scikit-learn>=1.0.0 in /usr/local/lib/python3.10/d
ist-packages (from scikit-optimize) (1.5.2)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.10/dist-
packages (from scikit-optimize) (24.1)
Requirement already satisfied: PyYAML in /usr/local/lib/python3.10/dist-packages
(from pyaml>=16.9->scikit-optimize) (6.0.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.10/
dist-packages (from scikit-learn>=1.0.0->scikit-optimize) (3.5.0)
Downloading scikit_optimize-0.10.2-py2.py3-none-any.whl (107 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 107.8/107.8 kB 3.2 MB/s eta 0:00:00
Downloading pyaml-24.9.0-py3-none-any.whl (24 kB)
Installing collected packages: pyaml, scikit-optimize
Successfully installed pyaml-24.9.0 scikit-optimize-0.10.2
```

In [34]:
```python
import time
from sklearn.base import clone
from sklearn.model_selection import cross_validate

from sklearn.svm import LinearSVC, SVC
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_sc
from sklearn.base import BaseEstimator, ClassifierMixin

from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer

import warnings
warnings.filterwarnings("ignore")

from sklearn.exceptions import DataConversionWarning
warnings.filterwarnings(action='ignore', category=DataConversionWarning)
```

Determining the Scoring metrics on which the results will be found.

Following is a dictionary that maps the metric to the corresponding library function

In [35]:
```python
SCORING_METRICS = {
    'accuracy': accuracy_score,
    'f1': f1_score,
    'precision': precision_score,
    'recall': recall_score,
    'roc_auc': roc_auc_score
}
```

Splitting the data into train and test sets

In [36]:
```
# split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X_new, y, test_size=0.2, ran

print("Train shape: ", X_train.shape)
print("Test shape: ", X_test.shape)
```

```
Train shape:  (71509, 20)
Test shape:  (17878, 20)
```

Defining 3 functions:

1. run_cross_validation(): It uses the built-in function cross_validate instead of cross_val_score because we can pass a list of keys that represent the scoring metrics (instead of just 1 metric)
2. run_grid_search(): Performs grid search on the parameter_grid and the model that is passed to it as a paramemter. We clone the model to _model, since there may be scenarios where the model gets updated during the execution which may affect the outcome.
3. run_bayesian_optimzation_search(): a similar function that performs bayesian optimization on the search_space provided and the model passed.

In [37]:
```
def run_cross_validation(model, X_train, y_train):

    _model = clone(model)
    cv_results = cross_validate(_model, X_train, y_train, cv=5, scoring = list(S
    avg_cv_results = {}
    for key in cv_results:
        avg_cv_results[f'avg_{key.replace("test", "val")}'] = np.mean(cv_results

    return avg_cv_results

def run_grid_search(model, X_train, y_train, params, scoring):
    _model = clone(model)

    grid_search_model = GridSearchCV(estimator=_model, param_grid=params, cv=5,
    grid_search_model.fit(X_train, y_train)

    grid_search_results = []

    i = 0
    for params in grid_search_model.cv_results_['params']:
        data = {}
        for param in params:
            data[param] = params[param]
        for key in SCORING_METRICS:
            data[f'mean_val_{key}'] = grid_search_model.cv_results_[f'mean_test_

        data[f'mean_fit_time'] = grid_search_model.cv_results_['mean_fit_time'][
        data[f'mean_score_time'] = grid_search_model.cv_results_['mean_score_tim
        i += 1
        grid_search_results.append(data)

    return grid_search_model, grid_search_results

def run_bayesian_optimzation_search(model, X_train, y_train, params, scoring):
    _model = clone(model)

    bayes_search_model = BayesSearchCV(estimator=_model, search_spaces=params, c
    bayes_search_model.fit(X_train, y_train)

    bayes_search_results = []

    i = 0
    for params in bayes_search_model.cv_results_['params']:
        data = {}
        for param in params:
            data[param] = params[param]
        for key in SCORING_METRICS:
            data[f'mean_val_{key}'] = bayes_search_model.cv_results_[f'mean_test

        data[f'mean_fit_time'] = bayes_search_model.cv_results_['mean_fit_time']
        data[f'mean_score_time'] = bayes_search_model.cv_results_['mean_score_ti
        i += 1

        bayes_search_results.append(data)

    return bayes_search_model, bayes_search_results


def evaluate_model_predictions(y_true, y_pred):
    test_results = {}
    for key in SCORING_METRICS:
        test_results[key] = SCORING_METRICS[key](y_true, y_pred)
    return test_results
```

Linear SVM

Using the LinearSVC function to perform SVM classification with linear kernel and using cross_validation to evaluate the performance

In [38]:
```
linear_svm = LinearSVC(random_state = RANDOM_STATE)

cv_results = run_cross_validation(linear_svm, X_train, y_train)
print("Cross Validation Evaluation:")
display(pd.DataFrame([cv_results]))

print("Avg Validation Score:", cv_results['avg_val_accuracy'])
```

```
Cross Validation Evaluation:
```

	avg_fit_time	avg_score_time	avg_val_accuracy	avg_val_f1	avg_val_precision	avg_val_r
0	0.807773	0.193984	0.632662	0.694825	0.631208	0.77

```
Avg Validation Score: 0.6326616272231839
```

Scalability and Efficiency

LinearSVC is a relatively fast functionality. However if we use SVC with kernel='linear', it takes a lot of time. This is why there are methods such as Stochastic Gradient Descent, or Mini-batch learning that train the models very quickly. For comparison, we shall see the difference in training times for SGD and LinearSVC.

In [39]:
```
# perform SGD with hinge loss

from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(loss='hinge', random_state=RANDOM_STATE)

cv_results = run_cross_validation(sgd_clf, X_train, y_train)
print("Cross Validation Evaluation:")
display(pd.DataFrame([cv_results]))

print("Avg Validation Score:", cv_results['avg_val_accuracy'])
```

```
Cross Validation Evaluation:
```

	avg_fit_time	avg_score_time	avg_val_accuracy	avg_val_f1	avg_val_precision	avg_val_r
0	1.274936	0.205667	0.628173	0.691768	0.627427	0.77

```
Avg Validation Score: 0.6281726770786247
```

Other Kernels

Due to less computation power and high training time taken by other kernels when used with SVC, only a small number of training examples had to be used (5% of even the 1% ~ 3k-4K data points).

In [40]:
```
X_train_small = X_train.sample(frac=0.03, random_state=RANDOM_STATE)
y_train_small = y_train.loc[X_train_small.index]
```

Polynomial Kernel SVM

The polynomial kernel maps input data points to a higher-dimensional feature space using polynomial functions. In this new space, a hyperplane can separate the data points into classes, which corresponds to a non-linear decision boundary in the original input space. Thus, a polynomial kernel is used to model non-linear relationships between input features.

In [41]:
```
poly_svc = SVC(random_state = RANDOM_STATE, kernel = 'poly')

param_grid = {
    'degree': [2, 3, 4],
}

grid_search_model, cv_results = run_grid_search(poly_svc, X_train_small, y_train
print("Grid Search Evaluation:")
display(pd.DataFrame(cv_results))

print("Best Model:")
display(grid_search_model.best_estimator_)

print("Best Validation Score:")
print(grid_search_model.best_score_)

cv_results_poly = cv_results
```

```
Grid Search Evaluation:
```

	degree	mean_val_accuracy	mean_val_f1	mean_val_precision	mean_val_recall	mean_va
0	2	0.627972	0.709691	0.626434	0.818828	
1	3	0.606061	0.701123	0.605987	0.832179	
2	4	0.589277	0.699401	0.589444	0.859917	

```
Best Model:
```

SVC

```
SVC(degree=2, kernel='poly', random_state=42)
```

```
Best Validation Score:
0.627972027972028
```

In [42]:
```
poly_svc = SVC(random_state = RANDOM_STATE, kernel = 'poly', degree = 2)

param_grid = {
    'C': [1, 10, 100, 1000]
}

grid_search_model, cv_results = run_grid_search(poly_svc, X_train_small, y_train

print("Grid Search Evaluation:")
display(pd.DataFrame(cv_results))

print("Best Model:")
display(grid_search_model.best_estimator_)

print("Best Validation Score:")
print(grid_search_model.best_score_)

cv_results_poly_c = cv_results
```

```
Grid Search Evaluation:
```

	C	mean_val_accuracy	mean_val_f1	mean_val_precision	mean_val_recall	mean_val_r
0	1	0.627972	0.709691	0.626434	0.818828	0.
1	10	0.626573	0.687089	0.643020	0.738297	0.
2	100	0.621911	0.678522	0.643441	0.718139	0.
3	1000	0.624709	0.678319	0.647930	0.712264	0.

```
Best Model:
```

SVC

```
SVC(C=1, degree=2, kernel='poly', random_state=42)
```

```
Best Validation Score:
0.627972027972028
```

RBF Kernel SVM

The RBF (Radial Basis Function) kernel function calculates the similarity between two points, X_1 and X_2, by computing the squared Euclidean distance between them. The kernel's value decreases with distance. It is used to deal with overlapping data. It behaves like a weighted nearest neighbor model, where the closest observations have the most influence on how a new observation is classified.

In [43]:
```
rbf_svc = SVC(random_state = RANDOM_STATE, kernel = 'rbf')

param_grid = {
    'gamma': [0.01, 0.1, 1, 10],
}

grid_search_model, cv_results = run_grid_search(rbf_svc, X_train_small, y_train_

print("Grid Search Evaluation:")
display(pd.DataFrame(cv_results))

print("Best Model:")
display(grid_search_model.best_estimator_)

print("Best Validation Score:")
print(grid_search_model.best_score_)

cv_results_rbf_gamma = cv_results
```

```
Grid Search Evaluation:
```

	gamma	mean_val_accuracy	mean_val_f1	mean_val_precision	mean_val_recall	mean_v
0	0.01	0.638695	0.721867	0.630814	0.843961	
1	0.10	0.661538	0.717848	0.668485	0.775159	
2	1.00	0.555711	0.714413	0.555711	1.000000	
3	10.00	0.555711	0.714413	0.555711	1.000000	

```
Best Model:
```

SVC

```
SVC(gamma=0.1, random_state=42)
```

```
Best Validation Score:
0.6615384615384616
```

In [44]:

```python
# seeing the effect of C on rbf kernel by keeping gamma fixed to the best value

rbf_svc = SVC(random_state = RANDOM_STATE, kernel = 'rbf', gamma = 0.1)

param_grid = {
    'C': [0.1, 1, 10, 100],
}

grid_search_model, cv_results = run_grid_search(rbf_svc, X_train_small, y_train_

print("Grid Search Evaluation:")
display(pd.DataFrame(cv_results))

print("Best Model:")
display(grid_search_model.best_estimator_)

print("Best Validation Score:")
print(grid_search_model.best_score_)

cv_results_rbf_c = cv_results
```

```
Grid Search Evaluation:
```

	C	mean_val_accuracy	mean_val_f1	mean_val_precision	mean_val_recall	mean_val_
0	0.1	0.561305	0.715969	0.559179	0.994972	0
1	1.0	0.661538	0.717848	0.668485	0.775159	0
2	10.0	0.637762	0.682676	0.665477	0.701375	0
3	100.0	0.629837	0.675261	0.658920	0.692982	0

```
Best Model:
```

SVC

```
SVC(C=1, gamma=0.1, random_state=42)
```

```
Best Validation Score:
0.6615384615384616
```

Custom Kernel SVM

So far, the RBF kernel outperforms the other kernels. This justified creating a hybrid kernel with one part as rbf and the other as linear.

Now, we will create a custom kernel by combining svm with rbf kernel and linear kernel. This hybrid kernel performs the best out of all the models.

https://scikit-learn.org/stable/developers/develop.html#rolling-your-own-estimator

In [45]:
```
from sklearn.utils.multiclass import unique_labels

# Custom hybrid kernel combining RBF and Linear
def hybrid_kernel(X, Y, alpha=0.5, beta=0.5, gamma=0.1):
    # Ensure inputs are NumPy arrays
    X = np.array(X)
    Y = np.array(Y)

    # RBF component
    rbf_part = np.exp(-gamma * np.sum((X[:, None, :] - Y[None, :, :]) ** 2, axis

    # Linear component
    linear_part = np.dot(X, Y.T)

    # Weighted combination of RBF and Linear
    return alpha * rbf_part + beta * linear_part

class HybridKernelSVC(ClassifierMixin, BaseEstimator):
    def __init__(self, C=1.0, alpha=0.5, beta=0.5, gamma=0.1):
        self.C = C
        self.alpha = alpha
        self.beta = beta
        self.gamma = gamma
        self.svc = None


    def _kernel_wrapper(self, X1, X2):
        return hybrid_kernel(X1, X2, alpha = self.alpha, beta = self.beta, gamma

    def fit(self, X, y):
        self.classes_ = unique_labels(y)
        self.X_ = X
        self.y_ = y
        self.svc = SVC(kernel=self._kernel_wrapper, C=self.C, random_state = RAN
        self.svc.fit(X, y)
        return self

    def predict(self, X):
        return self.svc.predict(X)

    def decision_function(self, X):
        return self.svc.decision_function(X)
```

In [46]:
```
hyb_svm = HybridKernelSVC()
```

```
cv_results = run_cross_validation(hyb_svm, X_train_small, y_train_small)
print("Cross Validation Evaluation:")
display(pd.DataFrame([cv_results]))

print("Avg Validation Score:", cv_results['avg_val_accuracy'])
```

```
Cross Validation Evaluation:
```

	avg_fit_time	avg_score_time	avg_val_accuracy	avg_val_f1	avg_val_precision	avg_val_r
0	0.705823	0.290794	0.672727	0.729431	0.674579	0.79

```
Avg Validation Score: 0.6727272727272727
```

The hybrid takes quite some time when all parameters are inserted in the grid (apha, beta, gamma) since it goes through all the combinations. That is why, gamma is kept to default and best performing value (with respect to rbf, that is, 0.1) and the relation is seen jsut with alpha and beta

In [47]:
```
hybrid_kernel_svc = HybridKernelSVC()

param_grid = {
    'alpha': [0.1, 1, 10],
    'beta': [0.01, 0.1, 1],
}

grid_search_model, cv_results = run_grid_search(hybrid_kernel_svc, X_train_small

print("Grid Search Evaluation:")
with pd.option_context('display.max_rows', None):
    display(pd.DataFrame(cv_results))

print("Best Model:")
display(grid_search_model.best_estimator_)

print("Best Validation Score:")
print(grid_search_model.best_score_)

cv_results_hybrid = cv_results
```

```
Grid Search Evaluation:
```

	alpha	beta	mean_val_accuracy	mean_val_f1	mean_val_precision	mean_val_recall	me
0	0.1	0.01	0.647086	0.728780	0.636182	0.853205	
1	0.1	0.10	0.644755	0.722459	0.638320	0.832239	
2	0.1	1.00	0.654079	0.727383	0.647063	0.830547	
3	1.0	0.01	0.660140	0.714266	0.669923	0.765100	
4	1.0	0.10	0.660140	0.713808	0.670500	0.763426	
5	1.0	1.00	0.668531	0.720860	0.677172	0.770979	
6	10.0	0.01	0.640559	0.684942	0.668206	0.703048	
7	10.0	0.10	0.643357	0.686313	0.672048	0.702208	
8	10.0	1.00	0.643823	0.686185	0.673129	0.700527	

```
Best Model:
```

```
HybridKernelSVC
HybridKernelSVC(alpha=1, beta=1)
```

```
Best Validation Score:
0.6685314685314685
```

Here we will fix alpha and beta found before and vary C.

In [48]:
```python
hybrid_kernel_svc = HybridKernelSVC(alpha = 1, beta = 0.01, gamma = 0.1)

param_grid = {
    'C': [0.1, 1, 10],
}

grid_search_model, cv_results = run_grid_search(hybrid_kernel_svc, X_train_small

print("Grid Search Evaluation:")
with pd.option_context('display.max_rows', None):
    display(pd.DataFrame(cv_results))

print("Best Model:")
display(grid_search_model.best_estimator_)

print("Best Validation Score:")
print(grid_search_model.best_score_)

cv_results_hybrid_c = cv_results
```

```
Grid Search Evaluation:
```

	C	mean_val_accuracy	mean_val_f1	mean_val_precision	mean_val_recall	mean_val_r
0	0.1	0.618182	0.734269	0.598607	0.949675	0.6
1	1.0	0.660140	0.714266	0.669923	0.765100	0.7
2	10.0	0.643357	0.686313	0.672048	0.702208	0.6

```
Best Model:
```

```
HybridKernelSVC
HybridKernelSVC(C=1, alpha=1, beta=0.01)
```

```
Best Validation Score:
0.6601398601398601
```

Hyperparameter tuning

Performing Bayesian Optimization on the best performing model (should have been hybrid, but have to use RBF, as mentioned above).

In [49]:
```
hyb_svc = HybridKernelSVC()

param_grid = {
    'C': Real(1e-1, 1e+1, prior='log-uniform'),
    'gamma': Real(1e-2, 1e+1, prior='log-uniform')
}

bayes_search_model, cv_results = run_bayesian_optimzation_search(hyb_svc, X_trai

print("Grid Search Evaluation:")
display(pd.DataFrame(cv_results))

print("Best Model:")
display(bayes_search_model.best_estimator_)

y_pred = bayes_search_model.best_estimator_.predict(X_test)
print("Test Data Evaluation:")
display(pd.DataFrame([evaluate_model_predictions(y_test, y_pred)]))

cv_results_hybrid_bayes = cv_results
```

```
Grid Search Evaluation:
```

	C	gamma	mean_val_accuracy	mean_val_f1	mean_val_precision	mean_val_reca
0	0.210901	0.040609	0.644755	0.724091	0.636991	0.83894
1	0.114192	0.057753	0.643823	0.723779	0.636128	0.83979
2	0.285972	0.241828	0.651748	0.727910	0.643412	0.83811
3	3.731244	0.317200	0.662471	0.717515	0.670163	0.77264
4	1.069017	0.011402	0.653147	0.730361	0.642840	0.84565
5	1.898417	0.159895	0.670396	0.723550	0.677320	0.77685
6	6.395840	0.045505	0.662937	0.710482	0.679791	0.74411
7	0.211967	0.174946	0.648019	0.725567	0.640233	0.83727
8	3.913899	0.049695	0.668998	0.718428	0.681268	0.76005
9	2.507596	2.520417	0.643357	0.709574	0.647966	0.78438

```
Best Model:
```

```
HybridKernelSVC
HybridKernelSVC(C=1.8984174942761238, gamma=0.1598951682624043)
```

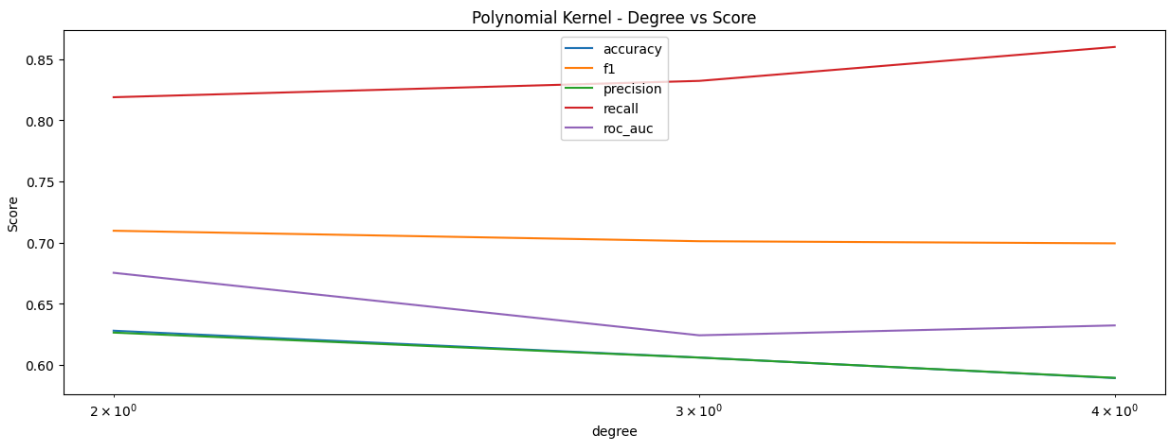
```
Test Data Evaluation:
```

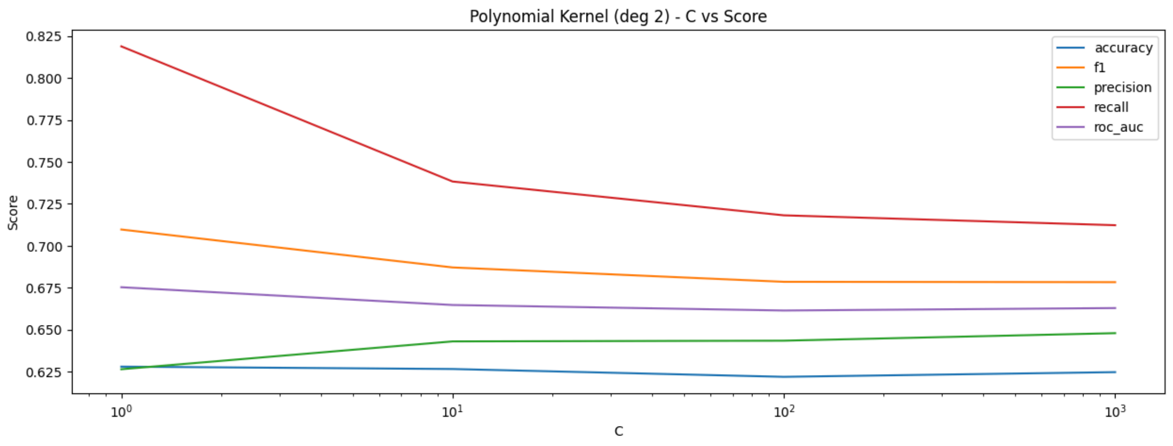
	accuracy	f1	precision	recall	roc_auc
0	0.650129	0.701703	0.651408	0.760413	0.640234

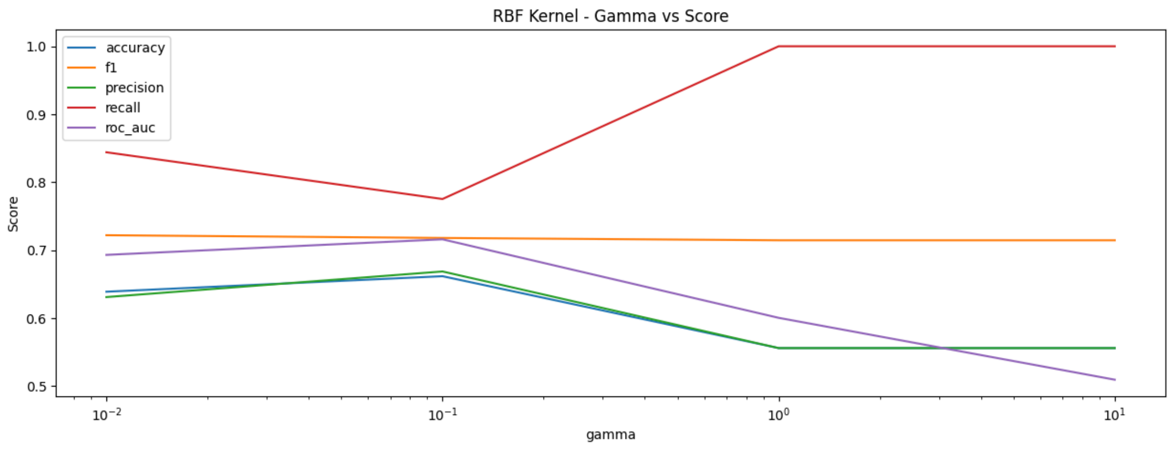
Sensitivity Analysis

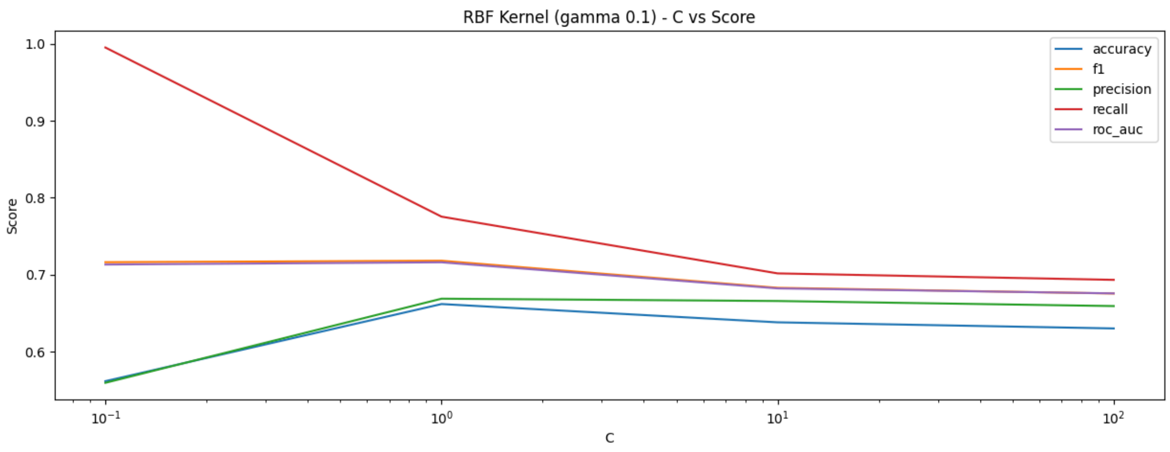
Now we will plot the cv_results of different kernels and see how the SVM model performs for different values of the hyperparameter.

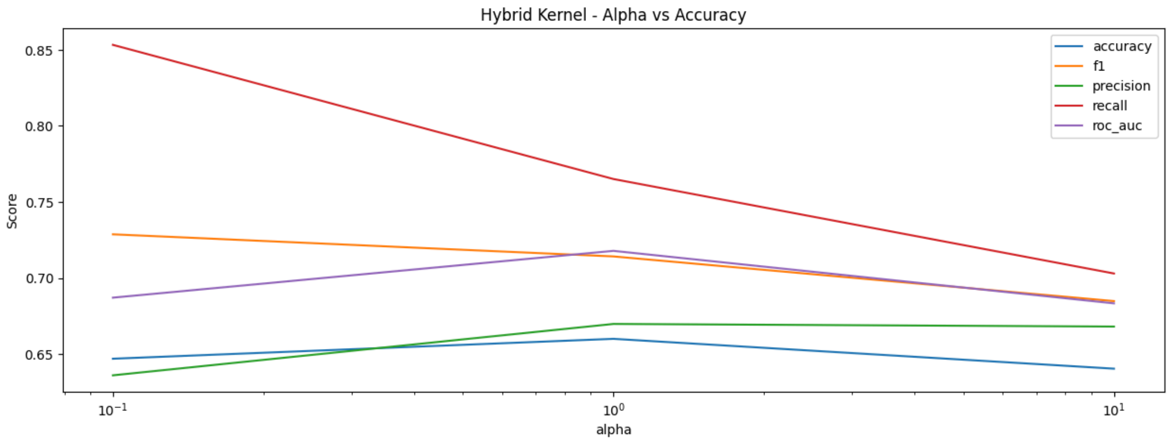
In [50]:
```
# plot the accuracy vs parameter graphs for different cv_results stored before

def plot_cv_results(cv_results, param_name, param_values, title):
    plt.figure(figsize=(15, 5))
    # print(cv_results)
    # print(param_values)
    for key in SCORING_METRICS:
        plt.plot(param_values, [result[f'mean_val_{key}'] for result in cv_resul

    plt.xscale('log')
    plt.xlabel(param_name)
    plt.ylabel("Score")
    plt.title(title)
    plt.legend()
    plt.show()

# plot the results for the polynomial kernel
plot_cv_results(cv_results_poly, 'degree', [2, 3, 4], 'Polynomial Kernel - Degre
plot_cv_results(cv_results_poly_c, 'C', [1, 10, 100, 1000], 'Polynomial Kernel (
# plot the results for the rbf kernel
plot_cv_results(cv_results_rbf_gamma, 'gamma', [0.01, 0.1, 1, 10], 'RBF Kernel -
plot_cv_results(cv_results_rbf_c, 'C', [0.1, 1, 10, 100], 'RBF Kernel (gamma 0.1

# plot the results for the hybrid kernel for beta = 0.01
plot_cv_results(list(filter(lambda x: x["beta"] == 0.01, cv_results_hybrid)), 'a
plot_cv_results(cv_results_hybrid_c, 'C', [0.1, 1, 10], 'Hybrid Kernel (alpha 1,
```

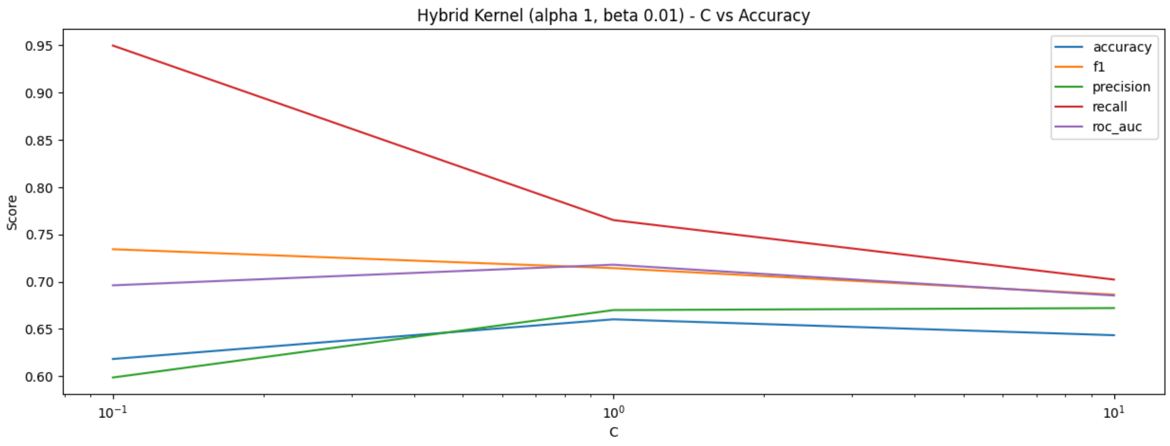








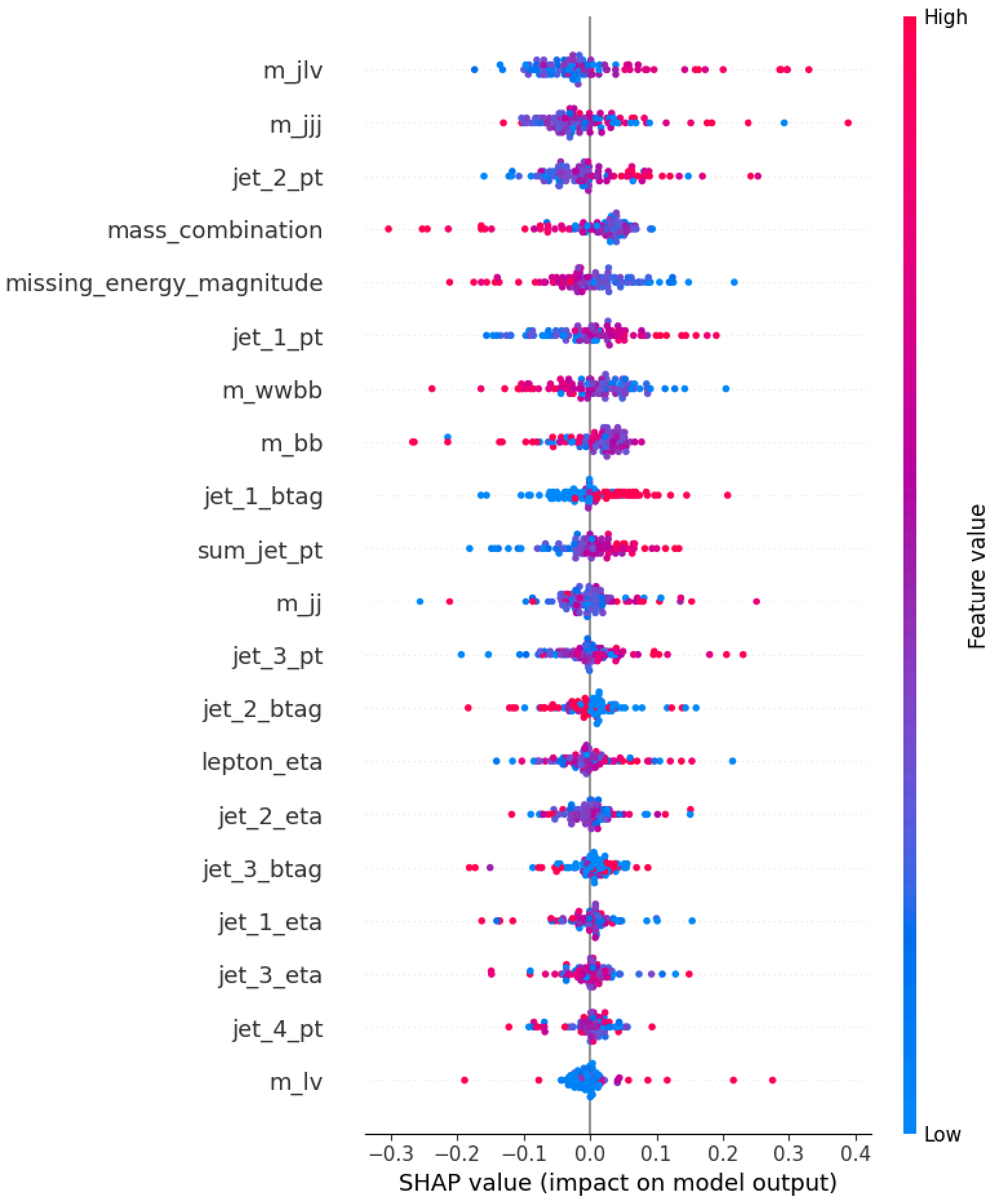


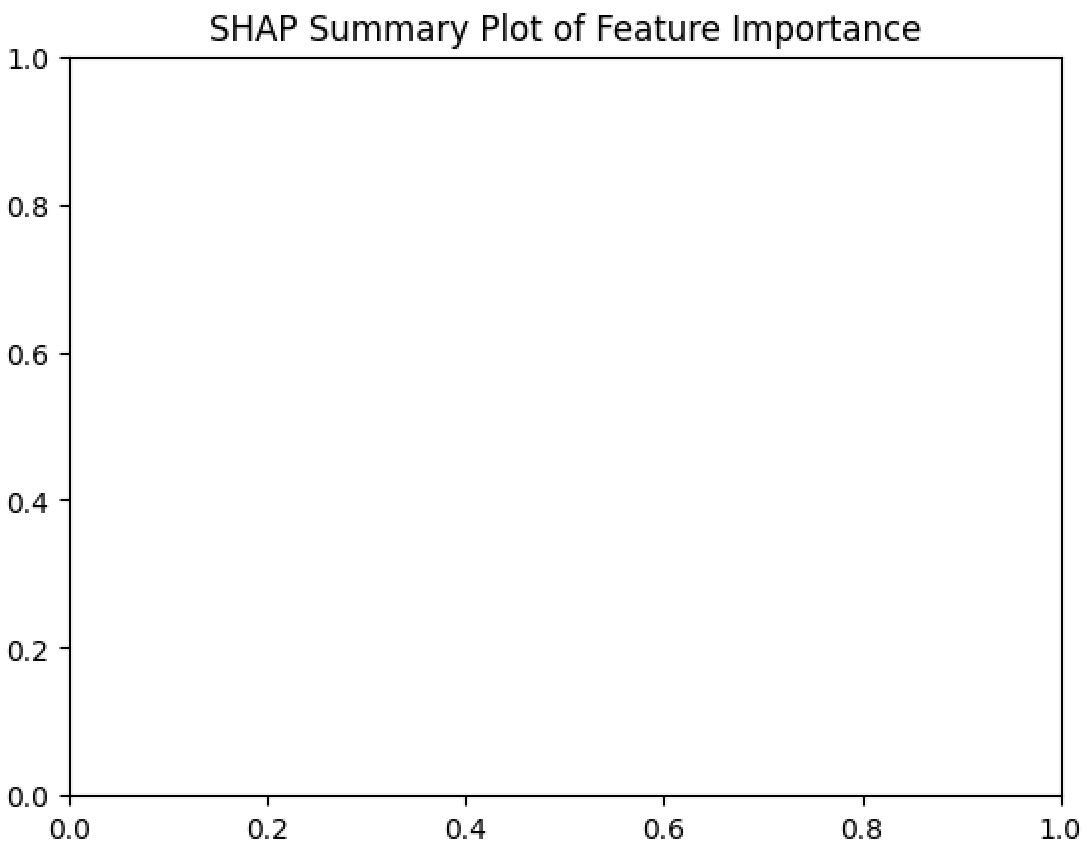


In [53]:
```python
# Thus the best model is Hybrid, with alpha = 1, beta = 0.01, gamma = 0.16, C =
# With this model, we should apply SHAP, but since SHAP takes a lot of time, I w
# SVC with rbf kernel and the same gamma and c values, and also reduce input sam

import shap

# Train the best model
best_model = SVC(random_state = RANDOM_STATE, kernel = 'rbf', gamma = 0.16, C =
best_model.fit(X_train_small, y_train_small)

# Explain model prediction using SHAP
explainer = shap.Explainer(best_model.predict, X_train_small[:100])
shap_values = explainer(X_train_small[:100])
```

```
PermutationExplainer explainer: 101it [20:09, 12.10s/it]
```

In [54]:
```python
plt.figure(figsize=(10, 6))
shap.summary_plot(shap_values, X_train_small[:100], feature_names=list(selected_
plt.title("SHAP Summary Plot of Feature Importance")
plt.show()
```





In [57]:
```
rf_resultX = pd.DataFrame(shap_values.values, columns=list(selected_features))

vals = np.abs(rf_resultX.values).mean(0)

shap_importance = pd.DataFrame(list(zip(list(selected_features), vals)), columns
shap_importance.sort_values(by=['feature_importance_vals'], ascending=False, inp
shap_importance
```

Out[57]:

	col_name	feature_importance_vals
15	m_jlv	0.064746
13	m_jjj	0.053758
5	jet_2_pt	0.051992
19	mass_combination	0.050558
1	missing_energy_magnitude	0.050442
2	jet_1_pt	0.050413
17	m_wwbb	0.048337
16	m_bb	0.043683
4	jet_1_btag	0.040879
18	sum_jet_pt	0.039004
12	m_jj	0.037858
8	jet_3_pt	0.037196
7	jet_2_btag	0.033842
0	lepton_eta	0.033438
6	jet_2_eta	0.028383
10	jet_3_btag	0.027258
3	jet_1_eta	0.026033
9	jet_3_eta	0.025246
11	jet_4_pt	0.022904
14	m_lv	0.022537

In [ ]: