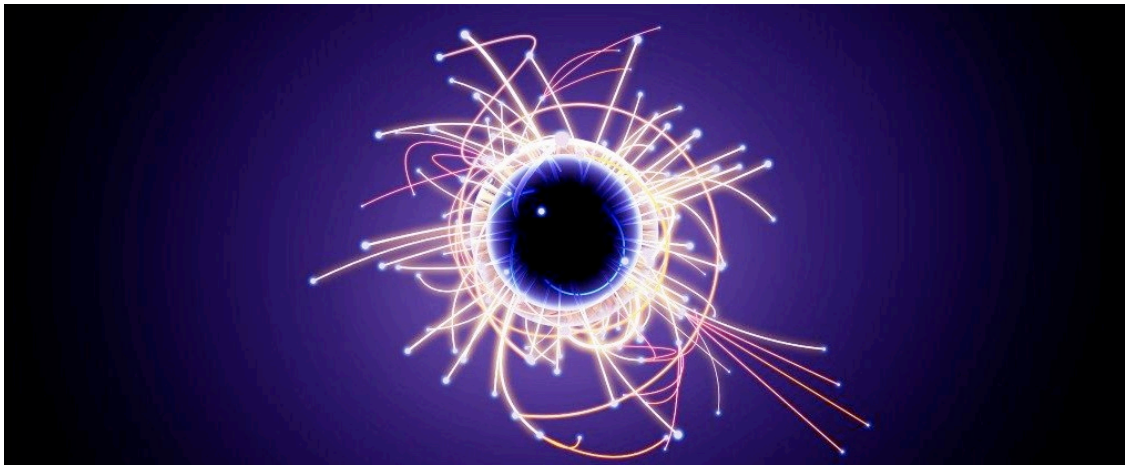
Universitat de Barcelona, Facultat de Matemàtiques i Informàtic

Postraduate in Data Science and Machine Learning

Data Science in Particle Physics

The Higgs Boson



Capstone Project 2023/2024

Ricardo Vicente

The Higgs boson is an elementary particle in the Standard Model of particle physics. It is the *quantum* excitation of the Higgs field,

which permeates all of space. The Higgs field is responsible for giving mass to other elementary particles through the mechanism of spontaneous symmetry breaking. Without the Higgs field, particles like quarks and electrons would be massless, and the universe would be vastly different. At high energies, the Higgs field has a symmetrical state with zero value. As the universe cooled, the field took on a non-zero value in all of space, breaking the symmetry. Particles interacting with the Higgs field gain mass. The more strongly a particle interacts with the Higgs field, the more massive it becomes.

The Higgs boson was first proposed by Peter Higgs in 1964, along with independent contributions by François Englert and Robert Brout, as well as other physicists. The prediction emerged from the need to explain how particles acquire mass. The electroweak theory, which unifies the electromagnetic force and weak nuclear force, required a mechanism to endow particles with mass while preserving the gauge symmetry of the theory. The Higgs mechanism provided this solution.

The confirmation of the Higgs boson took place at CERN (the European Organization for Nuclear Research) in Switzerland, using the Large Hadron Collider (LHC). The LHC is the world's largest and most powerful particle collider, capable of accelerating protons to near-light speeds and then smashing them together. Two main experiments at the LHC, ATLAS and CMS, were designed to detect the Higgs boson among other particles. These detectors are vast, sophisticated machines capable of tracking the particles produced by collisions. Protons were collided at very high energies, producing a wide range of particles. The Higgs boson is unstable and decays into other particles almost immediately. Physicists looked for specific patterns in the decay products that would indicate the presence of a Higgs boson. Some key decay channels included: two photons ($\gamma\gamma$), two Z bosons (ZZ) which further decay into four leptons, two W bosons (WW), two tau leptons ($\tau\tau$). Enormous amounts of data were analyzed to identify excess events over the expected background that matched the predicted properties of the Higgs boson. On July 4, 2012, CERN announced that both the ATLAS and CMS experiments had observed a new particle with a mass around 125-126 GeV (giga-electronvolts), consistent with the predicted Higgs boson.

Data Science played a crucial role in the discovery of the Higgs boson. The process involved massive amounts of data collection, processing, and analysis, which required advanced Data Science techniques.

1. **Data Collection**

ATLAS and CMS Detectors collected data from billions of proton-proton collisions per second. Each collision produced a vast amount of data, leading to petabytes of data being generated annually.

2. **Data Processing**

Real-time data filtering systems called triggers were used to identify potentially interesting events. These systems reduced the data from billions of events per second to a manageable number of events for further analysis. Sophisticated algorithms selected events that matched the expected signature of Higgs boson decays. The Worldwide LHC Computing Grid (WLCG) was used to store and process the massive amounts of data. It linked computing resources from institutions around the world.

3. **Data Analysis**

Scientists used hypothesis testing to determine if the observed data deviated significantly from the background noise and matched the expected signal for the Higgs boson. P-values were calculated to determine the statistical significance of the observed excess. A discovery required a 5-sigma significance level, meaning the probability of the result being due to random chance was less than one in 3.5 million. Machine learning algorithms were employed to recognize complex patterns in the data and to distinguish between signal (potential Higgs boson events) and background (other types of events); supervised learning techniques were used to classify events based on their characteristics, improving the accuracy of identifying Higgs boson events.

Phase 1: Frame the Problem

Collisions in high-energy particle accelerators are crucial for discovering exotic particles and require Data Analysis approaches to solve 'signal' (1) versus 'background' (0) classification problems. **'Shallow' Machine Learning models** have been limited in their ability to learn complex nonlinear functions, leading to a meticulous search for **manually constructed nonlinear features**.

Recent advances in the field of **Deep Learning** have allowed for learning more complex functions **directly from the data**, without the need for manually constructed features. Using benchmark datasets, Deep Learning methods have been shown to significantly improve classification metrics by up to 8% over the best current approaches.

While current approaches in high-energy physics have improved with manually constructed physics-inspired features, Deep Learning has proven to be more effective in automatically discovering powerful combinations of nonlinear features. Advances in Deep Learning, such as the use of neural networks with multiple hidden layers, have shown promise in improving the discrimination capability between signals and backgrounds in high-energy physics.

These developments in Deep Learning could inspire new advances in Data Analysis tools to address more complex problems in high-energy physics and could have applications in identifying decay products in high-dimensional raw detector data.

1. Presentation of the chosen dataset

The Higgs boson detection data has been produced using Monte Carlo simulations. The first 17 features (columns 1-17) represent kinematic properties measured by particle detectors in the accelerator (Low-Level). The last 7 features (columns 18-24) are functions of the first 17, designed by physicists to differentiate between the two classes (High-Level).

The variable set includes Low-Level features, which are direct measurements from particle detectors, and High-Level features, which are combinations of the former. These variables describe various aspects of particle collisions, ranging from the energy and momentum of individual particles to combined properties of multiple particles.

2. General Features

This dataset belongs to the "Classification in Numerical Features" benchmark. It is a smaller version (about 2.14%) of the original dataset, which contains 44 million rows. This version corrects the previous one in the definition of the class attribute, which is categorical, not numerical.

The author of this version is Daniel Whiteson from the University of California, Irvine, who selected 50% of experiments with 'background' and 50% with 'signal'; it is, therefore, a balanced dataset.

3. Definition of the variables

The variables are divided into 3 groups: kinematic or Low-Level, derived or High-Level, and the class label:

i. Kinematic or Low-Level features (numeric):

1. **lepton_pT** (Lepton transverse momentum)
2. **lepton_eta** (Lepton pseudorapidity)
3. **lepton_phi** (Lepton azimuthal angle)
4. **missing_energy_magnitude** (Missing energy magnitude)
5. **missing_energy_phi** (Missing energy azimuthal angle)
6. **jet_1_pt** (First jet transverse momentum)
7. **jet_1_eta** (First jet pseudorapidity)

8. **jet_1_phi** (First jet azimuthal angle)
9. **jet_2_pt** (Second jet transverse momentum)
10. **jet_2_eta** (Second jet pseudorapidity)
11. **jet_2_phi** (Second jet azimuthal angle)
12. **jet_3_pt** (Third jet transverse momentum)
13. **jet_3_eta** (Third jet pseudorapidity)
14. **jet_3_phi** (Third jet azimuthal angle)
15. **jet_4_pt** (Fourth jet transverse momentum)
16. **jet_4_eta** (Fourth jet pseudorapidity)
17. **jet_4_phi** (Fourth jet azimuthal angle)

ii. Derived or High-Level features engineered by physicists (numeric):

18. **m_jj** (Invariant mass of the dijet system)
19. **m_jjj** (Invariant mass of the trijet system)
20. **m_lv** (Invariant mass of the lepton-neutrino system)
21. **m_jlv** (Invariant mass of the jet-lepton-neutrino system)
22. **m_bb** (Invariant mass of the two b-tagged jets system)
23. **m_wbb** (Invariant mass of the jet-lepton-neutrino and two b-tagged jets system)
24. **m_wwbb** (Invariant mass of the jet-lepton-neutrino and four b-tagged jets system)

iii. Class label (categorical):

25. **label** (Used to distinguish between signal [1] and background [0].)

4. Presentation of the objectives

This is a classification problem to distinguish between a **signal** process produced by Higgs bosons and a **background** process that does not.

1. The first objective of this work is to analyze the **behavior of two types of models**: one more **'superficial'**, such as the Extreme Gradient Boosting Classifier, and others of **Deep Learning**, such as the Multilayer Perceptrons, using the libraries Keras/Tensorflow and Scikit-learn.

2. The second objective concerns the **variables**. Which model better adapts to the Low/High-Level variables? With which types of variables do the models make better predictions? Will the models be capable of learning complex patterns in **kinematic data** to accurately classify observations into different categories (signal or background), without requiring manual intervention by physicists to create **additional features**?

5. Project phases

1. Frame the Problem
2. Get the Data
3. Visualize the Data
4. Prepare the Data
5. Select, Train and Fine-Tune the Models
6. Conclusions

6. Reference

Baldi, P. et al. *Searching for exotic particles in high-energy physics with deep learning.* Nat. Commun. 5:4308 doi: 10.1038/ncomms5308 (2014).

Baldi, P. et al. *Enhanced Higgs to $\tau + \tau -$ Search with Deep Learning.* Physical Review Letters 114(11) doi: 10.1103/PhysRevLett.114.111801 (2014).

Phase 2: Get the Data

The "Get the Data" phase in a Python data science project marks the crucial initial step of acquiring the dataset necessary for analysis and modeling. This phase involves setting up the project environment, including installing essential Python libraries and tools, and obtaining the dataset itself.

In this phase, we'll typically:

1. **Setup Workspace**: Create a dedicated directory or workspace for the project. This ensures organization and separation from other projects.

2. **Install Dependencies**: Install required Python libraries such as Jupyter, NumPy, Pandas, Matplotlib, Keras, Tensorflow, and Scikit-Learn. These libraries form the backbone of data manipulation, visualization, and Machine/ Deep Learning tasks.

3. **Download Dataset**: Obtain the dataset needed for analysis. In the case of this project, this involves downloading a single file (dataset_higgs.csv).

4. **First visualizations**: Plotting histograms for numerical attributes provides a quick overview of the dataset's distribution patterns.

By completing the "Get the Data" phase effectively, we establish a solid foundation for our data science project. We ensure that we have access to the necessary data and the tools required for analysis and modeling. This sets the stage for subsequent phases such as data exploration, preprocessing, model building, and evaluation.

1. Load the data

This code loads and inspects a dataset from a CSV file using the pandas library. It specifies the file path to dataset_higgs.csv, reads the file into a DataFrame, and uses the first row as column names. The code then displays the first rows of the dataset to provide an initial view of the data's structure and contents.

In [4]:
```
# Load the data and print the first line
import csv
import pandas as pd

# File path of the dataset
file_path = "./higgsboson/dataset_higgs.csv"

# Read the CSV file and use the first row as column names
df = pd.read_csv(file_path, header=0)

# Display the DataFrame
df.head(10)
```

Out[4]:

	lepton_pT	lepton_eta	lepton_phi	missing_energy_magnitude	missing_energy_phi	jet_1_pt	jet_1_eta	jet_1_phi	jet_2_pt
0	0.730938	-0.862016	1.304896	1.203239	-0.937731	1.263905	-0.407010	-0.044292	0.738928
1	0.824456	-0.017586	-1.094985	0.338968	-1.236657	1.254561	-0.678333	0.087198	1.610054
2	0.592217	-0.614628	0.600777	1.598253	0.388530	0.829687	1.490271	-1.401960	0.669303
3	0.474177	-0.863964	0.021501	1.243103	-1.368377	0.524176	0.152470	1.473136	1.470679
4	1.349509	1.182344	1.483561	1.152123	-1.447959	0.587569	0.345565	-0.912446	1.609298
5	0.654440	1.307986	-0.389754	0.269518	-1.615687	0.829503	-1.506165	1.386654	0.367009
6	1.014419	-0.524049	0.950894	0.545483	0.334042	0.734690	-0.219856	0.246858	1.015915
7	1.089087	-0.997398	-0.904668	1.422371	-0.165290	0.602409	-0.137667	1.290192	0.507769
8	0.994105	-1.279849	1.436398	0.210038	-1.354840	0.871001	-0.976392	-0.548773	0.869112
9	1.079754	-1.445424	-0.358681	1.481180	0.146186	1.013267	-2.503327	-0.166255	1.530483

10 rows × 25 columns

The line df.info() in the code is used to display a concise summary of the DataFrame df. This method provides essential information about the DataFrame, including: Index Range, Column Names and Types, and Non-Null Counts.

In [6]:
```
# Display quick description of the data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 940160 entries, 0 to 940159
Data columns (total 25 columns):
 #   Column                     Non-Null Count   Dtype  
---  ------                     --------------   -----  
 0   lepton_pT                  940160 non-null  float64
 1   lepton_eta                 940160 non-null  float64
 2   lepton_phi                 940160 non-null  float64
 3   missing_energy_magnitude   940160 non-null  float64
 4   missing_energy_phi         940160 non-null  float64
 5   jet_1_pt                   940160 non-null  float64
 6   jet_1_eta                  940160 non-null  float64
 7   jet_1_phi                  940160 non-null  float64
 8   jet_2_pt                   940160 non-null  float64
 9   jet_2_eta                  940160 non-null  float64
 10  jet_2_phi                  940160 non-null  float64
 11  jet_3_pt                   940160 non-null  float64
 12  jet_3_eta                  940160 non-null  float64
 13  jet_3_phi                  940160 non-null  float64
 14  jet_4_pt                   940160 non-null  float64
 15  jet_4_eta                  940160 non-null  float64
 16  jet_4_phi                  940160 non-null  float64
 17  m_jj                       940160 non-null  float64
 18  m_jjj                      940160 non-null  float64
 19  m_lv                       940160 non-null  float64
 20  m_jlv                      940160 non-null  float64
 21  m_bb                       940160 non-null  float64
 22  m_wbb                      940160 non-null  float64
 23  m_wwbb                     940160 non-null  float64
 24  label                      940160 non-null  int64  
dtypes: float64(24), int64(1)
memory usage: 179.3 MB
```

The dataset, as summarized by df.info(), consists of 940,160 rows and 25 columns. All columns have complete data without any missing values. Most columns store numerical data in floating-point format, representing measurements like particle transverse momentum (lepton_pT), energy (missing_energy_magnitude), and angles (lepton_phi, jet_1_phi, etc.). The label column is integer-type, likely indicating the classification of each entry, such as distinguishing between different types of particle interactions. Overall, this dataset appears well-suited for machine learning tasks, particularly classification based on its comprehensive and numerical feature set.

The following df.describe() function in pandas provides a statistical summary of the numerical attributes (columns) and label in the DataFrame df. It includes key statistical measures such as count, mean, standard deviation (std), minimum (min), quartiles (25th, 50th, and 75th percentiles), and maximum (max) values for each column. This summary helps analysts quickly grasp the central tendency, spread, and distribution of the data, identify potential outliers, and make informed decisions about data preprocessing or modeling strategies for tasks such as machine learning, deep learning and statistical analysis.

In [9]:
```
# Display summary of the attributes
df.describe()
```

Out[9]:

	lepton_pT	lepton_eta	lepton_phi	missing_energy_magnitude	missing_energy_phi	jet_1_pt	jet_1
count	940160.000000	940160.000000	940160.000000	940160.000000	940160.000000	940160.000000	940160.000
mean	0.993551	0.000814	-0.000067	1.002481	-0.000698	0.989046	0.000
std	0.569061	1.009169	1.006939	0.604771	1.006263	0.474202	1.008
min	0.274697	-2.434976	-1.742508	0.001232	-1.743938	0.140892	-2.969
25%	0.590204	-0.738322	-0.873041	0.578123	-0.872272	0.677985	-0.687
50%	0.853737	0.000920	0.002081	0.894025	-0.000980	0.892804	-0.00
75%	1.239521	0.740162	0.872104	1.297610	0.870569	1.169091	0.688
max	9.745601	2.434868	1.743236	9.900929	1.743246	8.382610	2.969

8 rows × 25 columns

Based on the dataset summary provided, here are the overall tendencies observed across the attributes:

1. **Symmetry in Distributions**: Attributes such as lepton_eta, lepton_phi, and missing_energy_phi exhibit distributions symmetrically centered around zero. This symmetry indicates that measurements related to angles and directions (eta and phi) are evenly distributed across positive and negative values.

2. **Variable Ranges**: Most attributes, including lepton_pT, missing_energy_magnitude, and various jet attributes (jet_1_pt to jet_4_phi), cover a wide range of values. This variability suggests significant diversity in the physical quantities measured, reflecting the complexity of particle interactions.

3. **Moderate to High Variability**: Attributes such as lepton_pT, missing_energy_magnitude, and m_jj, m_jjj, m_lv, m_jlv, m_bb, m_wbb, m_wwbb (mass measurements) display moderate to high variability as indicated by their standard deviations. This variability implies that these attributes can significantly differ from one observation to another.

4. **Balanced Dataset**: The label attribute, which likely represents a binary classification (possibly signal vs. background events), has a mean around 0.5. This suggests a balanced distribution between the two classes, which is important for training classifiers without a class imbalance bias.

5. **Typical Values and Central Tendencies**: Attributes like lepton_pT and missing_energy_magnitude have means close to 1, indicating that their typical values are centered around this average. This central tendency helps in understanding the baseline values for these measurements.

6. **Standard Deviations**: The standard deviations across various attributes provide insights into the spread or dispersion of data points around their respective means. Higher standard deviations indicate greater variability, while lower ones suggest more clustered data points.

7. **Measurement Specificity**: Attributes related to jets (jet_1_pt to jet_4_phi) show specific measurements related to momentum (pt), pseudorapidity (eta), and azimuthal angle (phi) for up to four jets per event. Each of these attributes contributes uniquely to understanding the characteristics of jet formations in particle collisions.

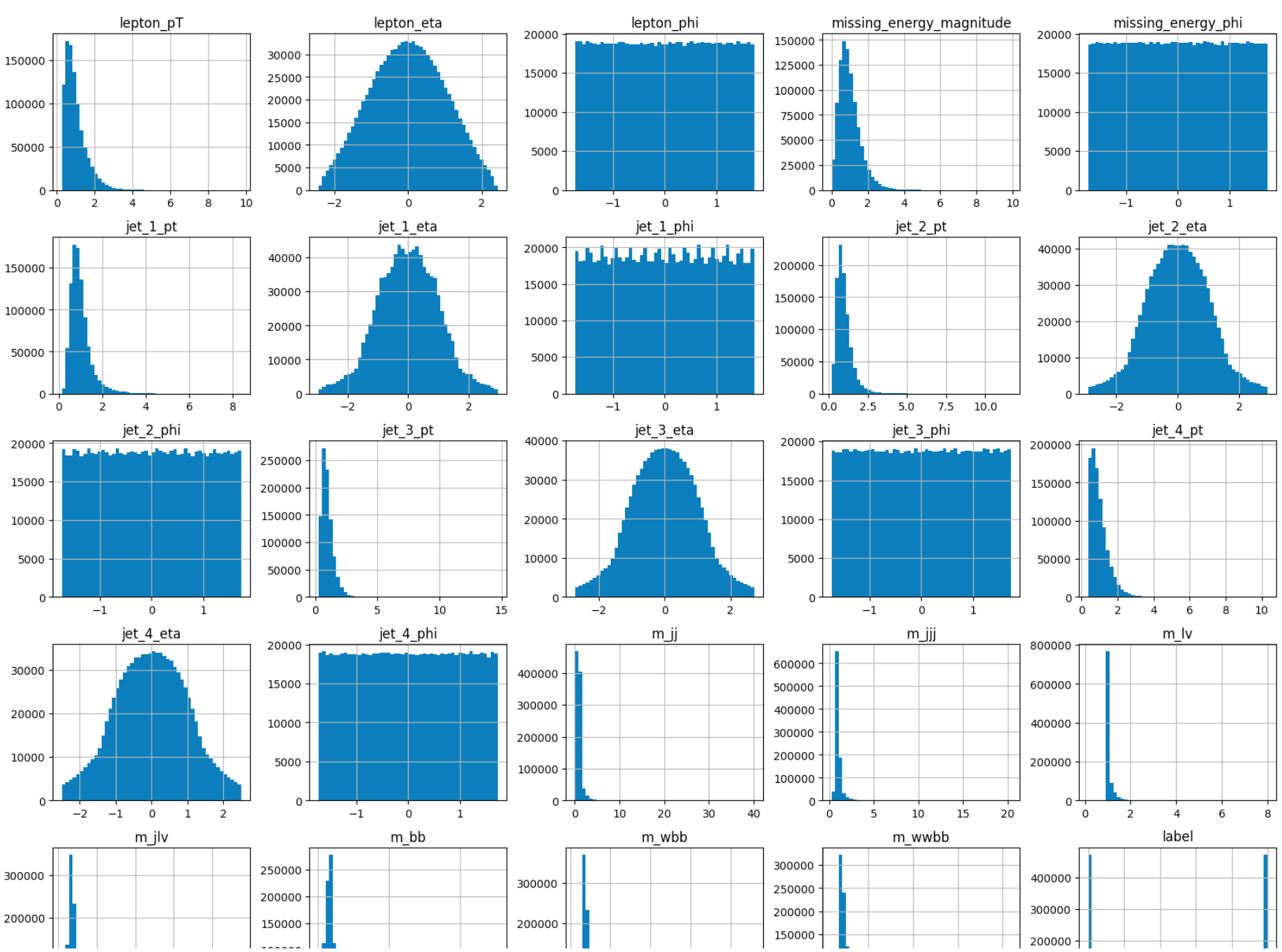
These overall tendencies provide a foundational understanding of the dataset's composition and variability, which is crucial for subsequent data exploration, feature engineering, and model development tasks in particle physics analysis.

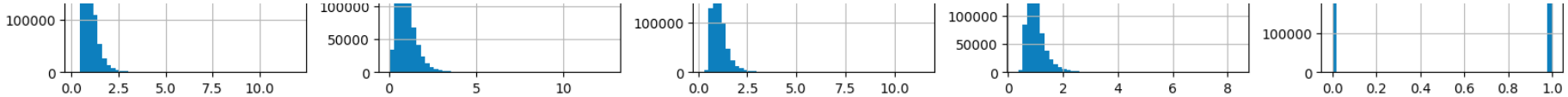
2. First Visualizations

This code snippet is used to visually explore and understand the distribution of the data attributes and label in the dataset. It helps in identifying the spread of values, detecting any skewness or outliers, gaining initial insights into the data characteristics, in conclusion, confirming the initial conclusions drawn earlier.

In [12]:
```
# Enables inline plotting in Jupyter notebooks
%matplotlib inline

# Importing necessary libraries
import matplotlib.pyplot as plt
```

```
# Plotting histograms for each column in the original DataFrame
df.hist(bins=50, figsize=(20, 15))
plt.suptitle('First Visualizations of All Features', fontsize=23)
plt.subplots_adjust(top=0.93)
plt.show()
```


First Visualizations of All Features
lepton_pT
lepton_eta
lepton_phi
missing_energy_magnitude
missing_energy_phi
jet_1_pt
jet_1_eta
jet_1_phi
jet_2_pt
jet_2_eta
jet_2_phi
jet_3_pt
jet_3_eta
jet_3_phi
jet_4_pt
jet_4_eta
jet_4_phi
m_jj
m_jjj
m_lv
m_jlv
m_bb
m_wbb
m_wwbb
label



Phase 3: Visualize the Data

Now, let's dive deeper into the data to unearth valuable insights, namely:

1. **Identifying Correlations utilizing Pearson's correlation coefficient (r)**. A positive r value close to 1 indicates a strong positive correlation, such as higher values of one feature correlating with higher values of another feature. Conversely, a negative r value near -1 indicates a strong negative correlation, where higher values of one feature are associated with lower values of another feature. Coefficients close to zero suggest no linear correlation between the features being analyzed.

2. **Employ Pandas' scatter_matrix function** for a visual representation of correlations between numerical attributes. The main diagonal presents histograms for individual attributes.

The following code first shuffles the entire dataset df to randomize the order of its rows. Then, it selects a random sample containing 5% of the rows from the shuffled dataset (shuffled_df). Finally, it prints the sizes of both the shuffled dataset (shuffled_df) and the sampled subset (shuffled_sampled_df) to confirm the number of rows in each after these operations.

Shuffling data involves randomly reordering the dataset, which helps remove any inherent sequence biases. Sampling, on the other hand, involves selecting a subset of the shuffled data for visualization purposes or specific analysis tasks. These steps are crucial for ensuring data integrity, reducing biases, and facilitating effective data exploration through visual representations.

In [15]:
```
# Shuffle the train_set
shuffled_df = df.sample(frac=1)

# Take a sample of 10% of the shuffled train_set
shuffled_sampled_df = shuffled_df.sample(frac=0.05)

print('Size of the train_set:', len(shuffled_df))
print('Size of the shuffled sample:', len(shuffled_sampled_df))
```

```
Size of the train_set: 940160
Size of the shuffled sample: 47008
```

1. Plot the Correlation Matrix

The CorrelationMatrix function calculates and visualizes the correlation matrix for specified columns in a Pandas DataFrame. It imports seaborn and matplotlib.pyplot for plotting. The function computes the correlation matrix for given columns (COLS) and the 'label' column from DATAFRAME. It then plots the matrix using sns.heatmap to show pairwise correlations. The plot uses colors to represent correlation strength (annot=True shows values), with adjustments for clarity (figsize=(45, 45)). It identifies top 5 variables correlated with 'label' and prints their values. This helps in exploring relationships and selecting features for analysis or modeling.

In [18]:
```
def CorrelationMatrix(DATAFRAME, COLS, columns):
    """
    Plotting the correlation matrix for the specified columns in the dataframe.

    Parameters:
        DATAFRAME (DataFrame): Pandas DataFrame containing the data.
        COLS (list): List of columns for which correlation matrix is calculated.
        columns (str): String to describe the columns being analyzed.

    Returns:
        top_variables (list): List of top 5 variables with the highest correlation to 'label'.
    """
    # Importing necessary libraries
    import seaborn as sns
    import matplotlib.pyplot as plt

    # Calculate the correlation matrix
    correlation_matrix = DATAFRAME[COLS + ['label']].corr()

    # Plot the correlation matrix
    plt.figure(figsize=(45, 45))
    heatmap = sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".4f", annot_kws={"size": 25}, xtick
    plt.xticks(rotation=45)
    plt.yticks(rotation=45)
    plt.tick_params(axis='both', which='major', labelsize=20)
    plt.title(f"Correlation Matrix for {columns}", fontsize=60)
    plt.show()

    # Find correlations of the 'label' with other variables
```

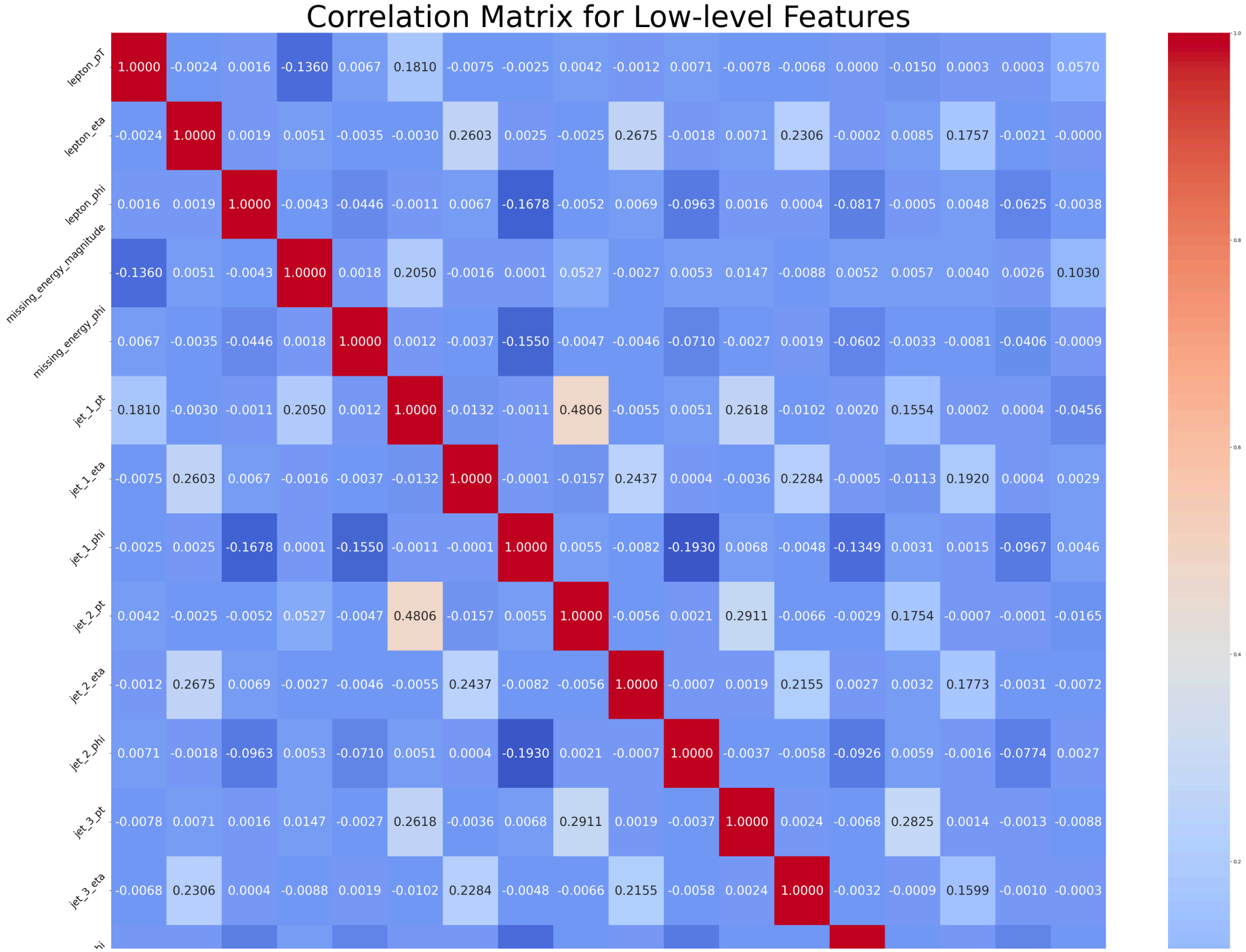
```
    label_correlations = correlation_matrix['label'].abs().sort_values(ascending=False)
    label_correlations = label_correlations[label_correlations.index != 'label']  # Exclude self-correlation

    # Print the labels of the top 5 variables with the highest correlation to the label
    print(f"\nTop 5 variables with the highest relation to 'label':\n")
    for index in label_correlations.head(5).index:
        correlation = label_correlations[index]
        print(f"\t{index}: {correlation:.4f}")

    top_variables = label_correlations.head(5).index.tolist()

    return top_variables
```

We define the following lists of column names (LowLevelCols and HighLevelCols) to categorize or group specific types of features or variables from our dataset. This organization can serve several purposes in data analysis tasks, such as 'Feature Selection'. LowLevelCols represents raw measurements or basic attributes, while HighLevelCols represents derived features or higher-level summaries.

In [20]:
```
# Definition of variable sets
LowLevelCols = ['lepton_pT', 'lepton_eta', 'lepton_phi', 'missing_energy_magnitude', 'missing_energy_phi', \
          'jet_1_pt', 'jet_1_eta', 'jet_1_phi', 'jet_2_pt', 'jet_2_eta', 'jet_2_phi', 'jet_3_pt', 'jet_3_eta', \
          'jet_3_phi', 'jet_4_pt', 'jet_4_eta', 'jet_4_phi']

HighLevelCols = ['m_jj', 'm_jjj', 'm_lv', 'm_jlv', 'm_bb', 'm_wbb', 'm_wwbb']
```

This line analyzes correlations among Low-Level features in shuffled_sampled_df, displaying them as a heatmap and identifying the top variables correlated with the 'label':

In [22]:
```
# Correlation Matrix for Low Level variables
top_low_variables = CorrelationMatrix(shuffled_sampled_df, LowLevelCols, 'Low-level Features')
```


Correlation Matrix for Low-level Features
lepton_pT 1.0000 -0.0024 0.0016 -0.1360 0.0067 0.1810 -0.0075 -0.0025 0.0042 -0.0012 0.0071 -0.0078 -0.0068 0.0000 -0.0150 0.0003 0.0003 0.0570
lepton_eta -0.0024 1.0000 0.0019 0.0051 -0.0035 -0.0030 0.2603 0.0025 -0.0025 0.2675 -0.0018 0.0071 0.2306 -0.0002 0.0085 0.1757 -0.0021 -0.0000
lepton_phi 0.0016 0.0019 1.0000 -0.0043 -0.0446 -0.0011 0.0067 -0.1678 -0.0052 0.0069 -0.0963 0.0016 0.0004 -0.0817 -0.0005 0.0048 -0.0625 -0.0038
missing_energy_magnitude -0.1360 0.0051 -0.0043 1.0000 0.0018 0.2050 -0.0016 0.0001 0.0527 -0.0027 0.0053 0.0147 -0.0088 0.0052 0.0057 0.0040 0.0026 0.1030
missing_energy_phi 0.0067 -0.0035 -0.0446 0.0018 1.0000 0.0012 -0.0037 -0.1550 -0.0047 -0.0046 -0.0710 -0.0027 0.0019 -0.0602 -0.0033 -0.0081 -0.0406 -0.0009
jet_1_pt 0.1810 -0.0030 -0.0011 0.2050 0.0012 1.0000 -0.0132 -0.0011 0.4806 -0.0055 0.0051 0.2618 -0.0102 0.0020 0.1554 0.0002 0.0004 -0.0456
jet_1_eta -0.0075 0.2603 0.0067 -0.0016 -0.0037 -0.0132 1.0000 -0.0001 -0.0157 0.2437 0.0004 -0.0036 0.2284 -0.0005 -0.0113 0.1920 0.0004 0.0029
jet_1_phi -0.0025 0.0025 -0.1678 0.0001 -0.1550 -0.0011 -0.0001 1.0000 0.0055 -0.0082 -0.1930 0.0068 -0.0048 -0.1349 0.0031 0.0015 -0.0967 0.0046
jet_2_pt 0.0042 -0.0025 -0.0052 0.0527 -0.0047 0.4806 -0.0157 0.0055 1.0000 -0.0056 0.0021 0.2911 -0.0066 -0.0029 0.1754 -0.0007 -0.0001 -0.0165
jet_2_eta -0.0012 0.2675 0.0069 -0.0027 -0.0046 -0.0055 0.2437 -0.0082 -0.0056 1.0000 -0.0007 0.0019 0.2155 0.0027 0.0032 0.1773 -0.0031 -0.0072
jet_2_phi 0.0071 -0.0018 -0.0963 0.0053 -0.0710 0.0051 0.0004 -0.1930 0.0021 -0.0007 1.0000 -0.0037 -0.0058 -0.0926 0.0059 -0.0016 -0.0774 0.0027
jet_3_pt -0.0078 0.0071 0.0016 0.0147 -0.0027 0.2618 -0.0036 0.0068 0.2911 0.0019 -0.0037 1.0000 0.0024 -0.0068 0.2825 0.0014 -0.0013 -0.0088
jet_3_eta -0.0068 0.2306 0.0004 -0.0088 0.0019 -0.0102 0.2284 -0.0048 -0.0066 0.2155 -0.0058 0.0024 1.0000 -0.0032 -0.0009 0.1599 -0.0010 -0.0003
1.0
0.8
0.6
0.4
0.2

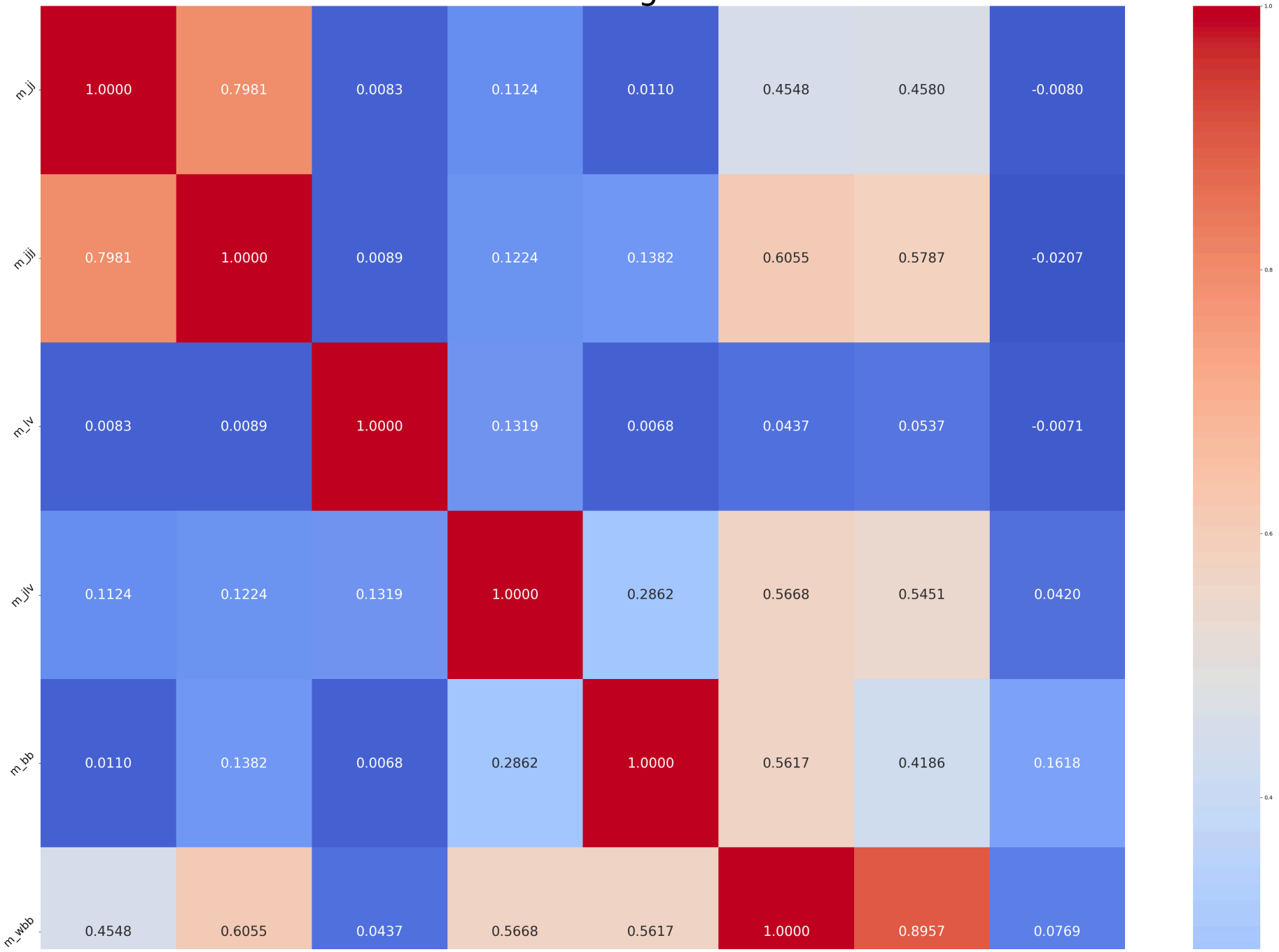

```
Top 5 variables with the highest relation to 'label':

        missing_energy_magnitude: 0.1030
        lepton_pT: 0.0570
        jet_1_pt: 0.0456
        jet_4_pt: 0.0279
        jet_2_pt: 0.0165
```

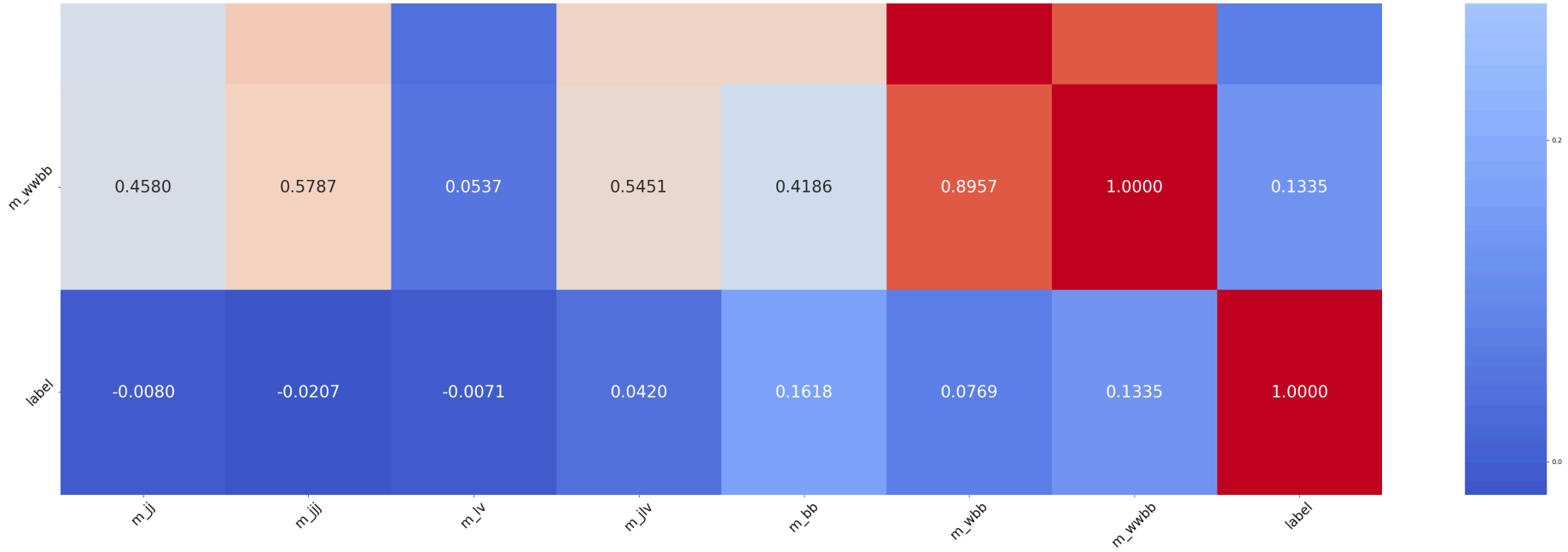
While these correlations suggest some level of relationship between each Low-Level variable and the label, they are not considered high. Typically, correlation coefficients around 0.1 to 0.3 are considered moderate, while those below 0.1 are considered weak. Therefore, in the context of correlation strength, these coefficients can be categorized as moderate to weak, with missing energy magnitude showing the strongest correlation among them.

This following line analyzes correlations among High-Level features in shuffled_sampled_df, displaying them as a heatmap and identifying the top variables correlated with the 'label'.

In [25]:
```
# Correlation Matrix for High Level variables
top_high_variables = CorrelationMatrix(shuffled_sampled_df, HighLevelCols, 'High-level Features')
```

Correlation Matrix for High-level Features






```
Top 5 variables with the highest relation to 'label':

        m_bb: 0.1618
        m_wwbb: 0.1335
        m_wbb: 0.0769
        m_jlv: 0.0420
        m_jjj: 0.0207
```

From this matrix, it can be observed that m_bb and m_wwbb show moderate correlations with the label, indicating they might exert a more significant influence compared to m_wbb, m_jlv, and m_jjj, which display weaker correlations. These relationships are crucial for understanding how variations in these variables could affect the label in your analysis.

These findings highlight that High-Level variables such as m_bb and m_wwbb are more strongly correlated with the Label compared to Low-Level variables. This suggests that changes in High-Level variables may have a more pronounced impact on the Label in your dataset analysis.

2. Boxplot the Top Features

The BoxPlot function generates boxplots for columns in a DataFrame based on a binary classification label. It imports matplotlib.pyplot

for plotting, calculates subplot layout based on COLS, and creates subplots. For each column, it extracts data for 'label' 0 and 1, then plots boxplots with labels '0' and '1'. Titles and axis labels are set, a main title for the figure is added, and the plot is displayed using plt.show().

In [30]:
```python
def BoxPlot(DATAFRAME, COLS, string):
    """
    Generate boxplots for specified columns in a dataframe, labeling them based on a binary classification.

    Parameters:
        DATAFRAME (DataFrame): The pandas DataFrame containing the data.
        COLS (list): List of column names for which boxplots are to be generated.
        string (str): String describing the columns being analyzed.

    Returns:
        None
    """
    # Import libraries
    import matplotlib.pyplot as plt

    # Calculate number of rows and columns needed for subplots
    num_cols = len(COLS)
    num_rows = 1

    # Create subplots
    fig, axs = plt.subplots(nrows=num_rows, ncols=num_cols, figsize=(15, 5))

    # Flatten axs if there is only one row
    axs = axs.flatten() if num_rows == 1 else axs

    # Iterate over columns
    for i, col in enumerate(COLS):
        # Extract data for label 0 and label 1
        label0_data = DATAFRAME.loc[DATAFRAME['label'] == 0, col]
        label1_data = DATAFRAME.loc[DATAFRAME['label'] == 1, col]

        # Create boxplot for each column
        axs[i].boxplot([label0_data, label1_data], positions=[0, 1], widths=0.6, tick_labels=['0', '1'])
        axs[i].set_title(col)  # Set subplot title
```

```
        # Set labels and title
        axs[i].set_xlabel('label')
        axs[i].set_ylabel(col)
        axs[i].set_xticks([0, 1])

    # Add main title
    plt.suptitle(f"Boxplots for top {string}", fontsize=20)

    # Adjust layout
    plt.tight_layout()

    # Show plot
    plt.show()
```

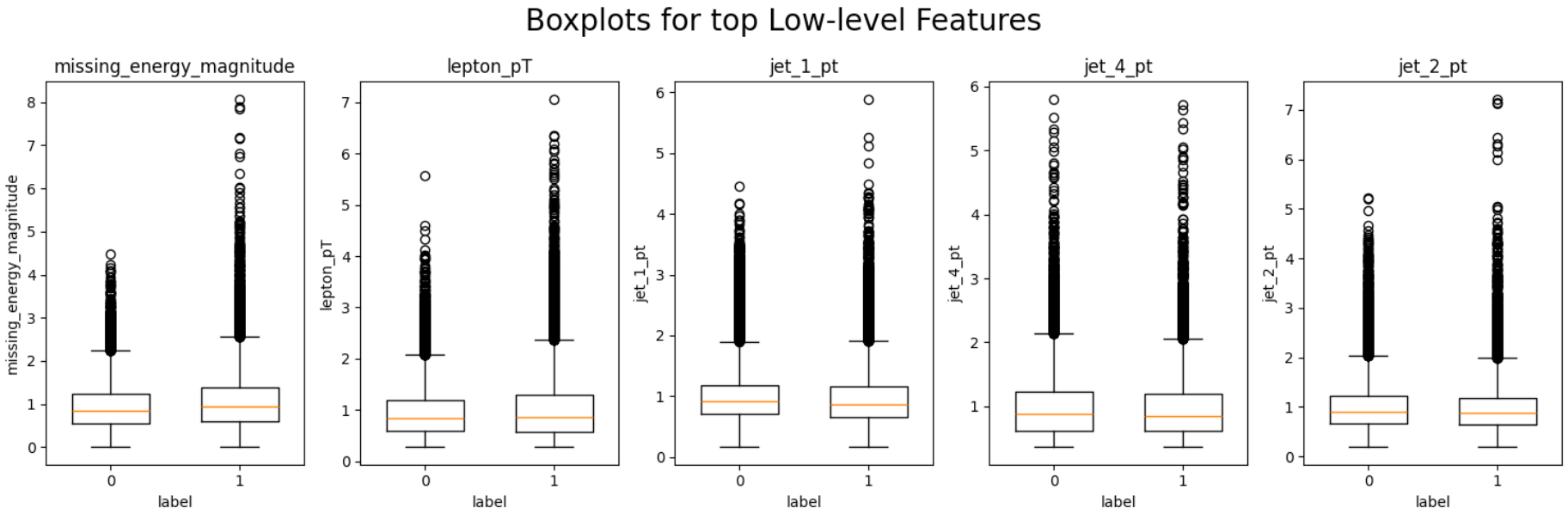
In the following boxplots for the top features related to the label, the significance of outliers and differences between the boxes can provide critical insights into the data distribution and group distinctions:

1. **Outliers**: Outliers in boxplots are individual data points that are significantly different from the rest of the data. They can indicate variability or errors in measurement but can also represent important anomalies or extremes in your dataset. Understanding outliers helps in assessing the overall distribution and potential impacts on statistical analyses.

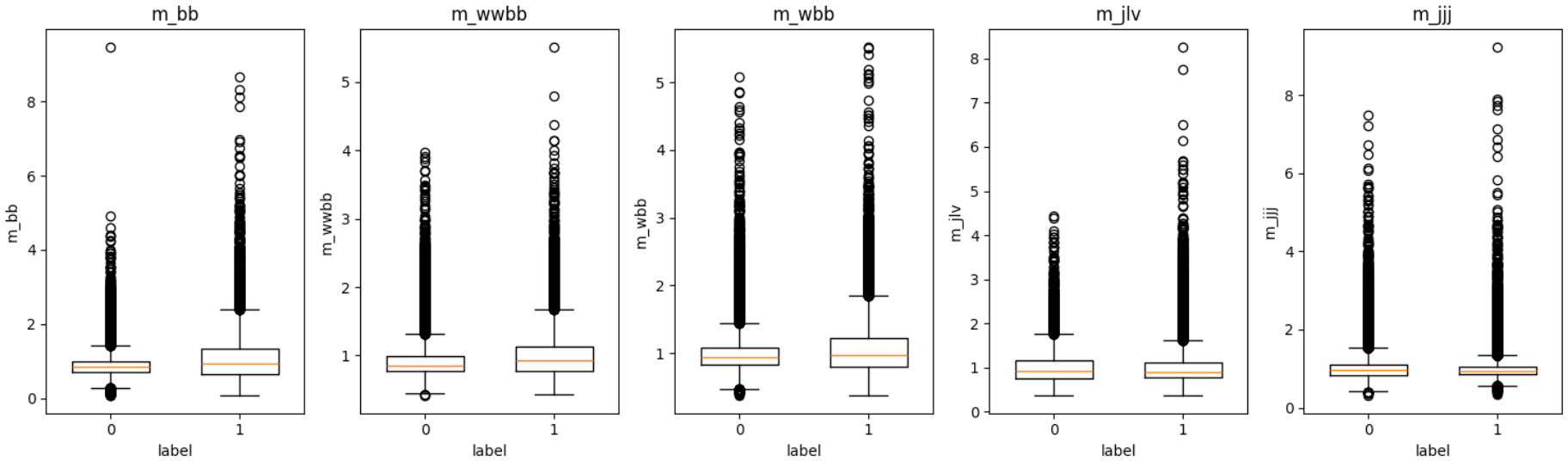
2. **Differences Between Boxes**: The differences between the boxes (especially their median, interquartile range, and whiskers) highlight variations between different categories or groups in your data. In the context of binary classification (labels '0' and '1'), these differences can signify how distinct the distributions of the data are between the two groups. Significant differences suggest that the variable (represented by the boxplot) may have predictive or discriminatory power in relation to the classification label.

3. **Overall Importance**: Boxplots provide a visual summary of the distributional characteristics of your data, allowing us to quickly compare different variables or groups. They help identify outliers, visualize central tendency, spread, skewness, and potential differences between groups. Understanding these aspects is crucial for making informed decisions in statistical modeling, hypothesis testing, or feature selection tasks.

In [32]:
```
# Scatter Plot for the 5 top Low Level variables
BoxPlot(shuffled_sampled_df, top_low_variables, 'Low-level Features')
```



In [33]:
```
# Scatter Plot for the 5 top High Level variables
BoxPlot(shuffled_sampled_df, top_high_variables, 'High-level Features')
```



The box plots underscore the superior correlation between the High-Level variables and the Label, through the significant differences in box distributions and medians between the two categories.

Phase 4: Prepare the Data

Before applying Machine Learning algorithms, it's necessary to prepare the data. This process involves several essential steps, like:

1. **Handling Missing Values (NaN)**: Prior to feeding the data into Machine/ Deep Learning algorithms, it's crucial to address missing values (NaNs). This can be done by either removing the rows or columns containing NaNs, filling them with a specific value (like the mean or median of the column), or using more advanced techniques such as imputation.

2. **Normalization**: Normalizing the data ensures that all features have the same scale. This step is crucial for many algorithms, specially in Deep Learning. Normalization typically involves scaling numerical features to a standard range, or using standardization, which centers the feature columns at mean 0 with standard deviation 1.

By preparing the data properly, we ensure it's clean, consistent, and appropriately scaled for effective model training and performance.

1. Handling Missing Values (NaNs)

The following code checks for NaN values in each column of the DataFrame df and print whether each column contains NaNs. This helps identify which columns may have missing data that need to be handled in data preprocessing or analysis.

In [38]:
```
# Verify if the set has NaN values
print("Do the columns in the dataset have NaN values?")
print(df.isna().any())
```

```
Do the columns in the dataset have NaN values?
lepton_pT                   False
lepton_eta                  False
lepton_phi                  False
missing_energy_magnitude    False
missing_energy_phi          False
jet_1_pt                    False
jet_1_eta                   False
jet_1_phi                   False
jet_2_pt                    False
jet_2_eta                   False
jet_2_phi                   False
jet_3_pt                    False
jet_3_eta                   False
jet_3_phi                   False
jet_4_pt                    False
jet_4_eta                   False
jet_4_phi                   False
m_jj                        False
m_jjj                       False
m_lv                        False
m_jlv                       False
m_bb                        False
m_wbb                       False
m_wwbb                      False
label                       False
dtype: bool
```

All columns in the dataset do not have NaN values. Each column shows a "False" result, indicating no NaNs are present.

2. Normalization

The following code prepares df for Machine/ Deep Learning analysis by applying standardization to the numerical features, ultimately optimizing the data for predictive modeling tasks. The significance of this process lies in its ability to enhance model performance and ensure consistent preprocessing of numerical features across different datasets:

1. **Enhanced Model Performance**: Standardization improves model performance by ensuring that features are on a comparable scale, which aids in better convergence and interpretation of model parameters.

2. **Preprocessing Consistency**: By standardizing features, the code facilitates consistent handling of numerical data across different features, enhancing the reliability and interpretability of machine learning models.

In [41]:
```
# Feature Scaling: Standardization
from sklearn.preprocessing import StandardScaler

# Initialize StandardScaler
scaler = StandardScaler()

# Fit and transform the selected columns
scaled_data = scaler.fit_transform(df[LowLevelCols + HighLevelCols])

# Convert the scaled data back to a DataFrame
df_std = pd.DataFrame(scaled_data, columns=LowLevelCols + HighLevelCols)

# Add the 'label' column back to the scaled DataFrame
df_std['label'] = df['label']
```

In [42]:
```
df_std.describe()
```

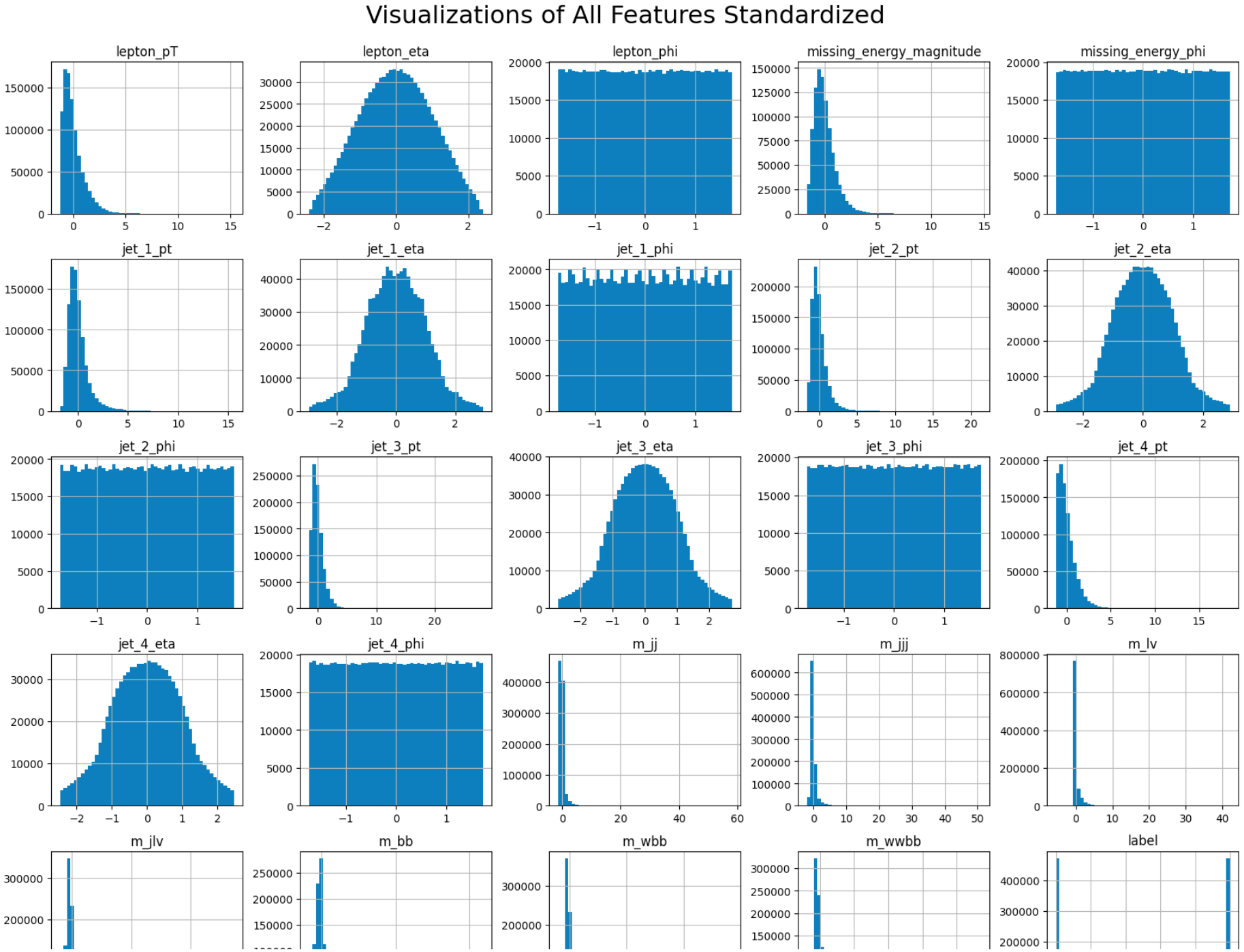
Out[42]:

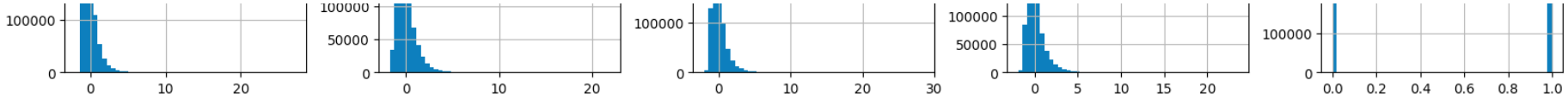
	lepton_pT	lepton_eta	lepton_phi	missing_energy_magnitude	missing_energy_phi	jet_1_pt	jet_1_e
count	9.401600e+05	9.401600e+05	9.401600e+05	9.401600e+05	9.401600e+05	9.401600e+05	9.401600e+(
mean	5.084806e-17	-8.003582e-18	3.524901e-17	1.104026e-16	8.328562e-18	-1.101003e-16	-3.317821e-
std	1.000001e+00	1.000001e+00	1.000001e+00	1.000001e+00	1.000001e+00	1.000001e+00	1.000001e+(
min	-1.263230e+00	-2.413660e+00	-1.730434e+00	-1.655583e+00	-1.732392e+00	-1.788595e+00	-2.943547e+(
25%	-7.087936e-01	-7.324211e-01	-8.669578e-01	-7.016842e-01	-8.661504e-01	-6.559684e-01	-6.812494e-
50%	-2.456915e-01	1.051763e-04	2.133592e-03	-1.793335e-01	-2.797749e-04	-2.029566e-01	-1.087861e-(
75%	4.322384e-01	7.326314e-01	8.661609e-01	4.880024e-01	8.658450e-01	3.796806e-01	6.820181e-
max	1.537982e+01	2.411941e+00	1.731290e+00	1.471375e+01	1.733091e+00	1.559161e+01	2.943335e+(

8 rows × 25 columns

The previous output and the following graphics show that all columns are already normalized, with mean centered around zero and standard deviation of one, ensuring consistent scaling across features.

In [44]:
```
# Enables inline plotting in Jupyter notebooks
%matplotlib inline

# Importing necessary libraries
import matplotlib.pyplot as plt

# Plotting histograms for each column in the standardized DataFrame
df_std.hist(bins=50, figsize=(20, 15))
plt.suptitle('Visualizations of All Features Standardized', fontsize=23)
plt.subplots_adjust(top=0.93)
plt.show()
```


Visualizations of All Features Standardized
lepton_pT
lepton_eta
lepton_phi
missing_energy_magnitude
missing_energy_phi
jet_1_pt
jet_1_eta
jet_1_phi
jet_2_pt
jet_2_eta
jet_2_phi
jet_3_pt
jet_3_eta
jet_3_phi
jet_4_pt
jet_4_eta
jet_4_phi
m_jj
m_jjj
m_lv
m_jlv
m_bb
m_wbb
m_wwbb
label



Phase 5: Select and Train the Models

This stage involves choosing suitable Machine/ Deep Learning algorithms to solve the problem at hand and then training it on the available data. Once the preparatory steps are complete (preprocess the data, split it into training and test sets, and perform feature engineering), we select the models that we believe might perform well on the task based on our understanding of the problem and the characteristics of the data. Then, we train the selected models on the training set, optimizing its parameters to minimize error and/or maximize performance metrics.

We could use here Grid-Search. It is a method used for fine-tuning the hyperparameters of a Machine/ Deep Learning model. Hyperparameters are parameters that are not directly learned from the data during training but instead control the learning process. Grid search involves defining a grid of hyperparameter values to explore, and then systematically training and evaluating the model for each combination of hyperparameters. This exhaustive search allows us to identify the combination of hyperparameters that results in the best performance on a validation set. Grid Search is a systematic way to find optimal hyperparameters and improve the model's performance; however, as it is computationally expensive, especially with a considerably large dataset, and we do not have the technical conditions to do so, it will not be carried out in this work. Instead, the tuning of the models will be done manually, through experimentation, that is, trial and error.

1. Create the Training and the Test Sets

The process of creating the Training and the Test sets involves random or stratified sampling to ensure that both sets represent the overall distribution of the data. Techniques like stratified sampling, as seen in the following code, are used to maintain the proportion of classes (if dealing with classification tasks) in both the training and test sets. This ensures that the model's evaluation on the test set reflects its true performance on unseen data.

In [47]:
```
# Importing necessary library
from sklearn.model_selection import StratifiedShuffleSplit

# Initialize StratifiedShuffleSplit for stratified sampling
split = StratifiedShuffleSplit(n_splits=1, test_size=0.3, random_state=42)
```

```
# Splitting data into training and test sets based on 'label' to maintain class proportionality
for train_index, test_index in split.split(df_std, df_std['label']):
    train_set_orig = df_std.loc[train_index]
    test_set = df_std.loc[test_index]

# Printing sizes of the training and test sets
print("train_set_orig size:", len(train_set_orig))
print("test_set size:", len(test_set))

# Displaying label category proportions in the training and test sets
print('\nLabel category proportions in the train_set_orig:')
print(train_set_orig['label'].value_counts() * 100 / len(train_set_orig))
print('\nLabel category proportions in the test_set:')
print(test_set['label'].value_counts() * 100 / len(test_set))
```

```
train_set_orig size: 658112
test_set size: 282048

Label category proportions in the train_set_orig:
label
0    50.0
1    50.0
Name: count, dtype: float64

Label category proportions in the test_set:
label
0    50.0
1    50.0
Name: count, dtype: float64
```

The label category proportions in both sets indicate the percentage distribution of each class label (0 and 1) in their respective datasets. Both the training and test sets exhibit an equal distribution of labels, with each label comprising 50% of the dataset, mirroring the original dataset.

2. Create the Validation Set

The validation set plays a crucial role in ensuring that Machine/ Deep Learning models generalize well to new data and in optimizing their performance before final evaluation on the test set.

In [51]:
```
# Split train_set_orig into training and validation sets
split = StratifiedShuffleSplit(n_splits=1, test_size=0.3, random_state=42)

# Reset index before splitting to avoid issues with indexing
train_set_orig.reset_index(drop=True, inplace=True)

# Perform the stratified split
for train_index, val_index in split.split(train_set_orig, train_set_orig['label']):
    train_set = train_set_orig.loc[train_index]
    val_set = train_set_orig.loc[val_index]

# Display sizes of the resulting sets
print("train_set size:", len(train_set))
print("val_set size:", len(val_set))

# Display label category proportions in the train_set and val_set
print('\nLabel category proportions in the train_set:')
print(train_set['label'].value_counts() * 100 / len(train_set))
print('\nLabel category proportions in the val_set:')
print(val_set['label'].value_counts() * 100 / len(val_set))
```

```
train_set size: 460678
val_set size: 197434

Label category proportions in the train_set:
label
1    50.0
0    50.0
Name: count, dtype: float64

Label category proportions in the val_set:
label
0    50.0
1    50.0
Name: count, dtype: float64
```

The validation set also shows an equal distribution of both classes (0 and 1), each making up 50% of the validation set. This ensures

that both sets maintain the same class distribution as the original dataset, facilitating unbiased model training and evaluation.

3. Drop the Label

Dropping the label refers to removing or excluding the column that contains the target variable (or label) from the dataset. This is typically done when preparing the data for machine learning tasks, where the goal is to separate the features (independent variables) from the target variable (dependent variable or label). By dropping the label, you ensure that the model does not inadvertently learn from the target variable during training, which could lead to biased results or overfitting. The label is usually dropped from the input features and stored separately as the target variable that the model aims to predict based on the input data.

In [54]:
```
# Separate the predictors and the labels in the training set
X_train = train_set.drop("label", axis=1)
y_train = train_set["label"].copy()

# Separate the predictors and the labels in the validation set
X_val = val_set.drop("label", axis=1)
y_val = val_set["label"].copy()

# Separate the predictors and the labels in the test set
X_test = test_set.drop("label", axis=1)
y_test = test_set["label"].copy()
```

4. Implement recurrent functions

At this moment, we are going to implement three functions: One function will compute **Statistics** such as accuracy, precision, and recall, which evaluate the performance and correctness of a classifier model. The second function will plot the **Receiver Operating Characteristic** (ROC) curve and calculate the Area Under the Curve (AUC), which assesses the model's ability to distinguish between classes based on predicted probabilities. The last function will **plot results** (Accuracy, Precision, and Recall) using bar plots.

4.1. Statistics for binary classification

Accuracy, precision, and recall are fundamental metrics used to evaluate the performance of classification models. Each metric provides unique insights into different aspects of the model's predictive ability, especially in scenarios where the distribution of classes in the dataset varies:

1. **Accuracy** measures the overall correctness of the model but can be misleading for imbalanced datasets:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Number of Instances}}$$

2. **Precision** measures the correctness of positive predictions:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

3. **Recall** measures the ability to capture all positive instances:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

In [57]:
```python
def Statistics(y_test, y_pred):
    """
    Generates statistical data: Accuracy, Precision and Recall.

    Parameters:
        y_test (DataFrame): Contains the true class labels for the data points in the test set.
        y_pred (DataFrame): Contains the predicted class labels for the data points in the test set, generated by t

    Returns:
        tuple (float, float, float): A tuple containing accuracy, precision, and recall scores.
    """
    # Import libraries
    from sklearn.metrics import accuracy_score, precision_score, recall_score

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy: {accuracy:.3f}")

    # Calculate precision
    precision = precision_score(y_test, y_pred)
    print(f"Precision: {precision:.3f}")

    # Calculate recall
```

```
    recall = recall_score(y_test, y_pred)
    print(f"Recall: {recall:.3f}")

    return accuracy, precision, recall
```

4.2. ROC (Receiver Operating Characteristic) Curve and AUC (Area Under the ROC Curve)

__ROC (Receiver Operating Characteristic) Curve__ is a graphical representation used to evaluate the performance of a binary classification model. It illustrates the trade-off between the true positive rate (TPR) and the false positive rate (FPR) at various threshold settings. Also known as sensitivity or recall, **True Positive Rate (TPR)** is the ratio of correctly predicted positive observations to the actual positives. **False Positive Rate (FPR)** is the ratio of incorrectly predicted positive observations to the actual negatives. The ROC curve is generated by varying the threshold for the classification model to determine what is considered a positive or negative prediction.

$$\text{FPR or Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

Interpreting the ROC Curve:

- The curve plots TPR (y-axis) against FPR (x-axis).
- A model with good predictive power will have a ROC curve that hugs the top left corner of the plot. This indicates high TPR and low FPR.
- A random guess model would produce a diagonal line (45-degree line) from the bottom left to the top right, representing a TPR equal to FPR.

__AUC (Area Under the ROC Curve)__ is a single scalar value that summarizes the performance of the ROC curve, reflecting the model's ability to distinguish between positive and negative classes.

Interpreting the AUC Area:

- 1.0: Perfect model. The model perfectly distinguishes between positive and negative classes.
- 0.8 to 0.9: Considered excellent. Models with AUC in this range have a high capability to differentiate between classes.

- 0.7 to 0.8: Considered acceptable. These models generally perform adequately in distinguishing between classes.
- 0.5 to 0.7: Considered reasonable. The closer to 0.7, the better the model's performance in distinguishing between classes.
- 0.5: No discrimination. The model performs no better than random guessing.
- < 0.5: Indicates a model that performs worse than random guessing, which typically indicates issues with the model or data.

In [59]:
```python
def ROC_AUC(y_test, y_pred_proba_list, model):
    """
    Plot ROC curve and calculate AUC for multiple probability predictions.

    Parameters:
        y_test (np.ndarray): True labels.
        y_pred_proba_list (List[List[np.ndarray]]): List of lists of predicted probabilities from different models.
        model (str): String to describe the algorithm being used.

    Returns:
        None.
    """
    # Import libraries
    from sklearn.metrics import roc_curve, roc_auc_score
    import matplotlib.pyplot as plt

    # Create a figure with a size of 7x7 inches
    plt.figure(figsize=(6, 6))

    # Features composed by each subset
    features = ['Low Level', 'High Level']

    # Define the colormap for each model type
    color_maps = [plt.get_cmap('Reds'), plt.get_cmap('Greens'), plt.get_cmap('Blues')]
    model_names = ['XGB', 'MLPK', 'MLPS']

    # Iterate through each model type (XGB, MLPK, MLPS)
    for j, model_preds in enumerate(y_pred_proba_list):
        color_map = color_maps[j]
        colors = [color_map((i+1) / len(model_preds)) for i in range(len(model_preds))]

        # Plot ROC for each set of predictions
        for i, y_pred_proba in enumerate(model_preds):
            fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
```

```
            roc_auc = roc_auc_score(y_test, y_pred_proba)
            plt.plot(fpr, tpr, color=colors[i], lw=2,
                     label=f"ROC curve (AUC = {roc_auc:.3f}) - {features[i]} {model_names[j]}")

    # Plot the diagonal line
    plt.plot([0, 1], [0, 1], 'k--', lw=2)

    # Plot the curves
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.0])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f"Receiver Operating Characteristic (ROC) with {model}")
    plt.legend(loc="lower right")
    plt.show()
```

4.3. Plotting the results report

The ResultsPlot function generates bar plots for model performance metrics (Accuracy, Precision, and Recall) from a provided report, allowing easy comparison of different models and feature sets.

In [61]:
```
def ResultsPlot(report):
    """Plot the results from a report.

    Parameters:
        report (dict): Dictionary with Models and Metrics (Accuracy, Precision, and Recall).

    Returns:
        None.
    """
    # Import libraries
    import matplotlib.pyplot as plt
    import matplotlib.cm as cm

    # Create subplots
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))

    # Define the metrics and their corresponding subplot positions
    metrics = ['Accuracy', 'Precision', 'Recall']
```

```
# Flatten the list of model names
models_flat = [model for sublist in report['model'] for model in sublist]

# Create colormaps for different model types
color_maps = [cm.Reds, cm.Greens, cm.Blues]

# Loop through the metrics and plot them
for i, metric in enumerate(metrics):
    # Plot the metric for each model
    for j, model_type in enumerate(report['model']):
        color_map = color_maps[j]
        # Determine color based on model index within each model type
        for k, model_name in enumerate(model_type):
            color = color_map((k + 1) / len(model_type))
            x_pos = k + j * len(report['model'][0])
            axs[i].bar(x_pos, report[metric][j][k], color=color, label=model_name if i == 0 else "")

            # Add values on top of each bar
            axs[i].text(x_pos, report[metric][j][k], '{:.2f}'.format(report[metric][j][k]), ha='center', va='bo

    axs[i].set_title(metric, fontsize=16)
    axs[i].set_xlabel('Features_Model', fontsize=14)
    axs[i].set_ylabel(metric, fontsize=14)
    axs[i].set_xticks(range(len(models_flat)))
    axs[i].set_xticklabels(models_flat, rotation=45, ha='right')

# Adjust the spacing between subplots
plt.tight_layout()
plt.show()
```

5. Train and Predict

5.1. Extreme Gradient Boosting

Extreme Gradient Boosting (XGBoost) is a powerful boosting algorithm known for its efficiency and versatility:

1. **Boosting Technique**: Sequentially combines weak learners to create a strong learner, improving predictive accuracy iteratively.

2. **Gradient Boosting**: Optimizes model parameters using gradient descent, reducing prediction errors by fitting new models to residuals.

3. **Regularization**: Prevents overfitting and enhances generalization with techniques like tree pruning and feature subsampling.

4. **Efficiency**: Utilizes parallel and distributed computing for scalability, handling large datasets efficiently.

5. **Cross-validation and Early Stopping**: Supports hyperparameter tuning and prevents overfitting with cross-validation and early stopping techniques.

We can also manage the behavior of this method using manually hyperparameters, such as:

1. **n_estimators**: This hyperparameter controls the number of trees in the ensemble, influencing the model's overall complexity and performance.

2. **learning_rate**: By adjusting this hyperparameter, we determine the step size of each tree's contribution during the boosting process, affecting the rate of convergence and the model's generalization capability.

3. **max_depth**: This hyperparameter specifies the maximum depth of each tree in the ensemble, influencing the complexity of individual trees and, consequently, the model's ability to capture complex patterns in the data.

4. **early_stopping_rounds**: This hyperparameter allows the model to automatically halt training when the performance metric on a validation dataset stops improving for a specified number of consecutive boosting rounds (early_stopping_rounds). This helps prevent overfitting, and ensures efficient use of computational resources. It enhances model robustness by stopping training at the point where further iterations no longer improve validation performance, thereby optimizing model training and performance.

In the following function, **tracking time** (start_time, end_time, run_time) ensures transparency regarding the computational cost of training. The predicted probabilities (y_pred_proba) are crucial for tasks such as computing Receiver Operating Characteristic (ROC) curves and calculating the Area Under the Curve (AUC).

In [64]:
```
def ExtremeGradientBoosting(X_train, y_train, X_val, y_val, X_test, y_test, COLS, \
                            n_estimators, learning_rate, max_depth, \
                            early_stopping_rounds=70, eval_metric="merror"):
    """
    Train and predict with Extreme Gradient Boosting.
```

```
    Parameters:
        X_train (pd.DataFrame): Training features.
        y_train (Union[pd.Series, np.ndarray]): Training labels.
        X_val (pd.DataFrame): Validation features.
        y_val (Union[pd.Series, np.ndarray]): Validation labels.
        X_test (pd.DataFrame): Test features.
        y_test (Union[pd.Series, np.ndarray]): Test labels.
        COLS (List[str]): Set of features/columns with which the model is trained and predicts.
        n_estimators (int): Controls the number of trees in the ensemble.
        learning_rate (float): Determines the step size of each tree's contribution.
        max_depth (int): Specifies the maximum depth of each tree, influencing its complexity.
        early_stopping_rounds (int, optional): Number of rounds of boosting to perform before stopping. Default is
        eval_metric (str, optional): Evaluation metric used to monitor performance on the validation set. Default i

    Returns:
        Tuple: A tuple containing the following elements:
            - statistics (Tuple[float, float, float]): Evaluation metrics for the test set (accuracy, precision, re
            - evals_result (Dict[str, Dict[str, List[float]]]): Evaluation results of the model on the training and
            - y_pred_proba (np.ndarray): Predicted probabilities for the test set.
            - xgb_clf (xgb.XGBClassifier): Trained XGBoost model.
    """
    # Import libraries
    import time
    import xgboost as xgb
    import matplotlib.pyplot as plt

    # Print start time
    print("Start time:", time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))

    # Record start time
    start_time = time.time()

    # Initialize the Gradient Boosting Classifier with specified hyperparameters
    xgb_clf = xgb.XGBClassifier(
        objective='multi:softmax',
        num_class=2,
        n_estimators=n_estimators,
        learning_rate=learning_rate,
        max_depth=max_depth,
        early_stopping_rounds=early_stopping_rounds,
        eval_metric=eval_metric,
```

```
    )

    # Train the classifier on the training data with evaluation set
    xgb_clf.fit(
        X_train[COLS],
        y_train,
        eval_set=[(X_train[COLS], y_train), (X_val[COLS], y_val)],
        verbose=True
    )

    # Get evaluation results
    evals_result = xgb_clf.evals_result()

    # Predict on the test data
    y_pred = xgb_clf.predict(X_test[COLS])

    # Predict on the test data for ROC/AUC
    y_pred_proba = xgb_clf.predict_proba(X_test[COLS])[:, 1]

    # Record end time
    end_time = time.time()

    # Calculate and print time taken
    run_time = end_time - start_time
    print(f"\nTime taken for running Extreme Gradient Boosting: {run_time/60:.2f} minutes.\n")

    # Return metrics, evaluation results, predictions and model
    return Statistics(y_test, y_pred), evals_result, y_pred_proba, xgb_clf
```

The following function is useful for monitoring the training process of an XGBoost model by visually inspecting how both training and validation errors evolve over the course of training. It helps in diagnosing issues such as overfitting or underfitting, and assessing the overall training progress.

In [66]:
```
def XGBTrainValidationLossPlot(evals_result, columns):
    """
    Plot training and validation loss for a Extreme Gradient Boosting model.

    Parameters:
        evals_result (dict): Provides a dictionary containing the evaluation results of the model during training.
```

```
        columns (str): String to describe the columns being analyzed.

    Returns:
        None.
    """
    # Plot learning curves
    epochs = len(evals_result['validation_0']['merror'])
    x_axis = range(0, epochs)

    # Plot training and validation loss
    plt.figure(figsize=(8, 3))
    plt.plot(x_axis, evals_result['validation_0']['merror'], label='Train')
    plt.plot(x_axis, evals_result['validation_1']['merror'], label='Validation')
    plt.legend()
    plt.xlabel('Boosting Rounds')
    plt.ylabel('Loss')
    plt.title(f"XGBoost Loss with {columns}")
    plt.show()
```

5.1.1. Low Level features

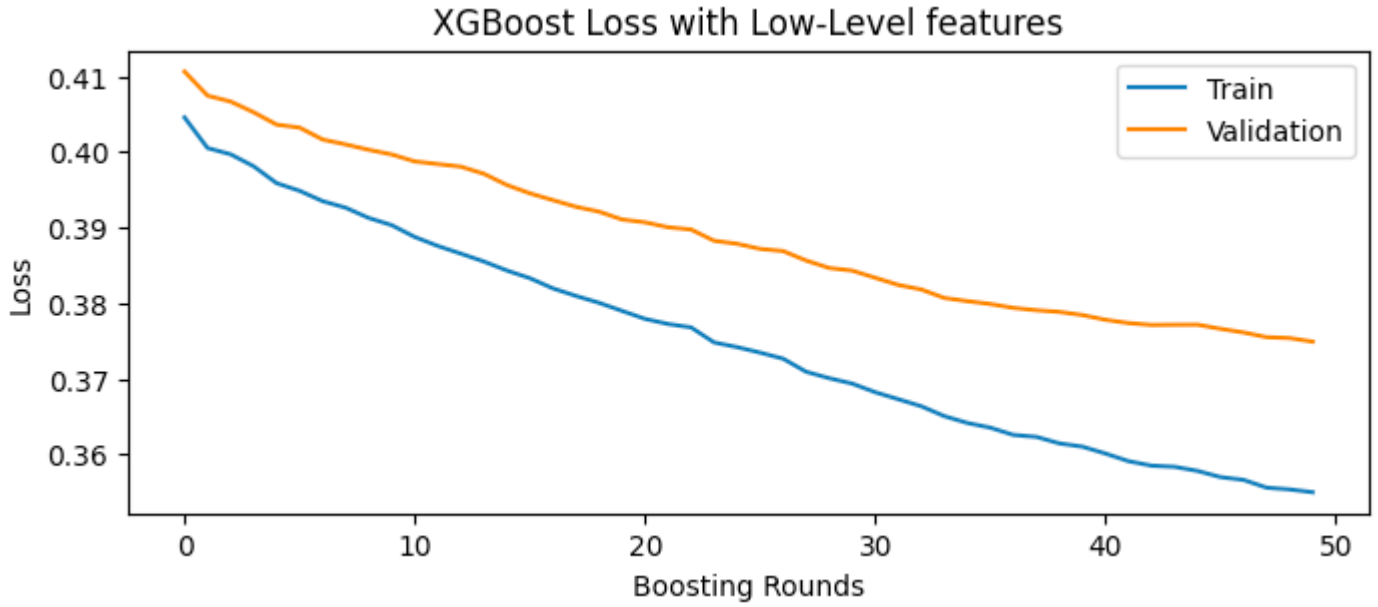
We start by training and predicting with the Low-Level features using the Extreme Gradient Boosting model.

In [68]:
```
# Extreme Gradient Boosting Classifier with Low Level features
(acc_XGB_ll, prec_XGB_ll, rec_XGB_ll), evals_result_XGB_ll, y_pred_proba_XGB_ll, clf_XGB_ll = ExtremeGradientBoosti
    X_train, y_train, X_val, y_val, X_test, y_test, LowLevelCols, \
    50, 0.1, 8
)
```

```
Start time: 2024-07-03 22:47:55
[0]     validation_0-merror:0.40457     validation_1-merror:0.41064
[1]     validation_0-merror:0.40052     validation_1-merror:0.40745
[2]     validation_0-merror:0.39970     validation_1-merror:0.40667
[3]     validation_0-merror:0.39811     validation_1-merror:0.40526
[4]     validation_0-merror:0.39588     validation_1-merror:0.40362
[5]     validation_0-merror:0.39486     validation_1-merror:0.40322
[6]     validation_0-merror:0.39349     validation_1-merror:0.40165
[7]     validation_0-merror:0.39263     validation_1-merror:0.40100
[8]     validation_0-merror:0.39127     validation_1-merror:0.40029
[9]     validation_0-merror:0.39032     validation_1-merror:0.39967
[10]    validation_0-merror:0.38876     validation_1-merror:0.39875
[11]    validation_0-merror:0.38757     validation_1-merror:0.39840
[12]    validation_0-merror:0.38655     validation_1-merror:0.39806
[13]    validation_0-merror:0.38551     validation_1-merror:0.39712
[14]    validation_0-merror:0.38432     validation_1-merror:0.39564
[15]    validation_0-merror:0.38330     validation_1-merror:0.39453
[16]    validation_0-merror:0.38196     validation_1-merror:0.39363
[17]    validation_0-merror:0.38096     validation_1-merror:0.39275
[18]    validation_0-merror:0.38005     validation_1-merror:0.39211
[19]    validation_0-merror:0.37898     validation_1-merror:0.39107
[20]    validation_0-merror:0.37792     validation_1-merror:0.39070
[21]    validation_0-merror:0.37725     validation_1-merror:0.39004
[22]    validation_0-merror:0.37680     validation_1-merror:0.38975
[23]    validation_0-merror:0.37481     validation_1-merror:0.38826
[24]    validation_0-merror:0.37419     validation_1-merror:0.38786
[25]    validation_0-merror:0.37346     validation_1-merror:0.38718
[26]    validation_0-merror:0.37266     validation_1-merror:0.38689
[27]    validation_0-merror:0.37093     validation_1-merror:0.38564
[28]    validation_0-merror:0.37007     validation_1-merror:0.38466
[29]    validation_0-merror:0.36936     validation_1-merror:0.38431
[30]    validation_0-merror:0.36822     validation_1-merror:0.38336
[31]    validation_0-merror:0.36729     validation_1-merror:0.38242
[32]    validation_0-merror:0.36636     validation_1-merror:0.38181
[33]    validation_0-merror:0.36506     validation_1-merror:0.38069
[34]    validation_0-merror:0.36415     validation_1-merror:0.38027
[35]    validation_0-merror:0.36353     validation_1-merror:0.37989
[36]    validation_0-merror:0.36257     validation_1-merror:0.37938
[37]    validation_0-merror:0.36231     validation_1-merror:0.37906
[38]    validation_0-merror:0.36145     validation_1-merror:0.37885
[39]    validation_0-merror:0.36102     validation_1-merror:0.37843
```

```
[40]    validation_0-merror:0.36011     validation_1-merror:0.37780
[41]    validation_0-merror:0.35910     validation_1-merror:0.37737
[42]    validation_0-merror:0.35850     validation_1-merror:0.37712
[43]    validation_0-merror:0.35835     validation_1-merror:0.37714
[44]    validation_0-merror:0.35781     validation_1-merror:0.37715
[45]    validation_0-merror:0.35699     validation_1-merror:0.37661
[46]    validation_0-merror:0.35662     validation_1-merror:0.37613
[47]    validation_0-merror:0.35559     validation_1-merror:0.37550
[48]    validation_0-merror:0.35536     validation_1-merror:0.37538
[49]    validation_0-merror:0.35499     validation_1-merror:0.37491

Time taken for running Extreme Gradient Boosting: 0.37 minutes.

Accuracy: 0.624
Precision: 0.626
Recall: 0.617
```

In [69]:
```
XGBTrainValidationLossPlot(evals_result_XGB_ll, 'Low-Level features')
```



The validation loss curve closely follows the train loss curve. The process is halted at boosting round 50 because the values start to diverge.

5.1.2. High Level features

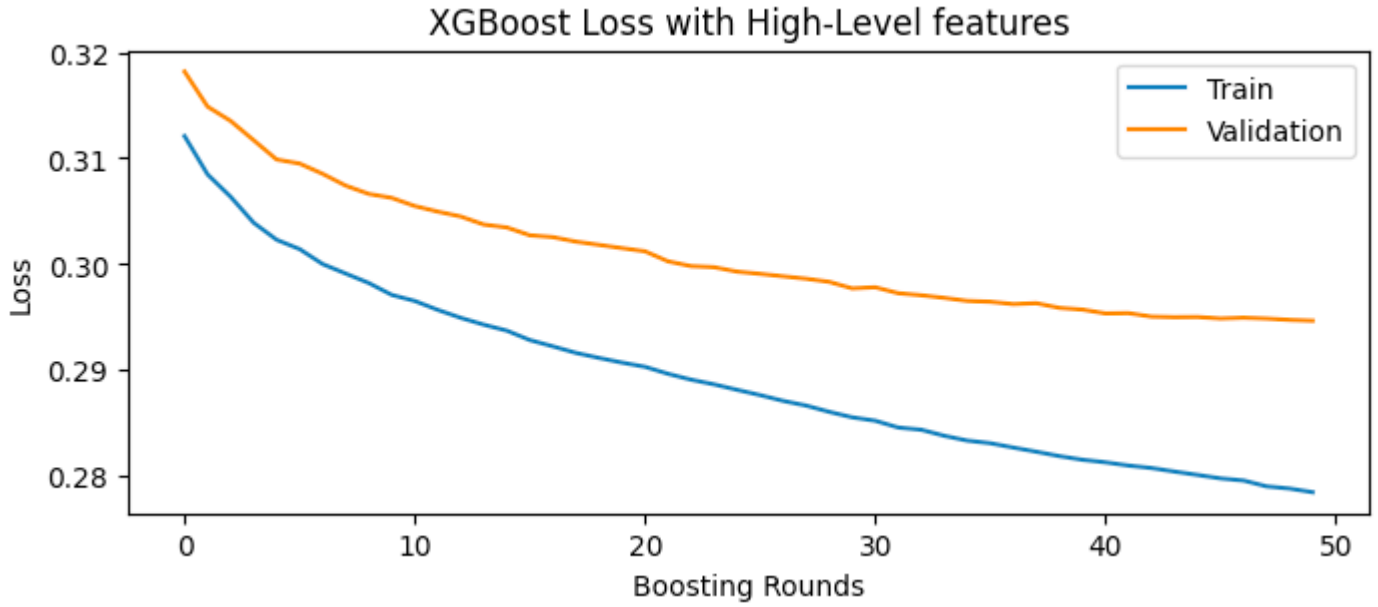
We finally train and predict using the Extreme Gradient Boosting model with the High-Level features.

In [72]:
```
# Extreme Gradient Boosting Classifier with High Level features
(acc_XGB_hl, prec_XGB_hl, rec_XGB_hl), evals_result_XGB_hl, y_pred_proba_XGB_hl, clf_XGB_hl = ExtremeGradientBoosti
    X_train, y_train, X_val, y_val, X_test, y_test, HighLevelCols, \
    50, 0.1, 8
)
```

```
Start time: 2024-07-03 22:48:18
[0]     validation_0-merror:0.31209     validation_1-merror:0.31818
[1]     validation_0-merror:0.30847     validation_1-merror:0.31487
[2]     validation_0-merror:0.30637     validation_1-merror:0.31352
[3]     validation_0-merror:0.30392     validation_1-merror:0.31170
[4]     validation_0-merror:0.30231     validation_1-merror:0.30988
[5]     validation_0-merror:0.30143     validation_1-merror:0.30950
[6]     validation_0-merror:0.30001     validation_1-merror:0.30853
[7]     validation_0-merror:0.29912     validation_1-merror:0.30741
[8]     validation_0-merror:0.29823     validation_1-merror:0.30663
[9]     validation_0-merror:0.29711     validation_1-merror:0.30626
[10]    validation_0-merror:0.29653     validation_1-merror:0.30549
[11]    validation_0-merror:0.29569     validation_1-merror:0.30496
[12]    validation_0-merror:0.29494     validation_1-merror:0.30450
[13]    validation_0-merror:0.29429     validation_1-merror:0.30374
[14]    validation_0-merror:0.29373     validation_1-merror:0.30347
[15]    validation_0-merror:0.29284     validation_1-merror:0.30273
[16]    validation_0-merror:0.29225     validation_1-merror:0.30256
[17]    validation_0-merror:0.29163     validation_1-merror:0.30213
[18]    validation_0-merror:0.29116     validation_1-merror:0.30183
[19]    validation_0-merror:0.29072     validation_1-merror:0.30152
[20]    validation_0-merror:0.29032     validation_1-merror:0.30122
[21]    validation_0-merror:0.28966     validation_1-merror:0.30027
[22]    validation_0-merror:0.28911     validation_1-merror:0.29982
[23]    validation_0-merror:0.28866     validation_1-merror:0.29972
[24]    validation_0-merror:0.28815     validation_1-merror:0.29930
[25]    validation_0-merror:0.28765     validation_1-merror:0.29909
[26]    validation_0-merror:0.28711     validation_1-merror:0.29886
[27]    validation_0-merror:0.28666     validation_1-merror:0.29863
[28]    validation_0-merror:0.28607     validation_1-merror:0.29834
[29]    validation_0-merror:0.28555     validation_1-merror:0.29772
[30]    validation_0-merror:0.28523     validation_1-merror:0.29783
[31]    validation_0-merror:0.28459     validation_1-merror:0.29726
[32]    validation_0-merror:0.28438     validation_1-merror:0.29707
[33]    validation_0-merror:0.28381     validation_1-merror:0.29682
[34]    validation_0-merror:0.28336     validation_1-merror:0.29654
[35]    validation_0-merror:0.28311     validation_1-merror:0.29646
[36]    validation_0-merror:0.28270     validation_1-merror:0.29624
[37]    validation_0-merror:0.28231     validation_1-merror:0.29632
[38]    validation_0-merror:0.28189     validation_1-merror:0.29588
[39]    validation_0-merror:0.28155     validation_1-merror:0.29572
```

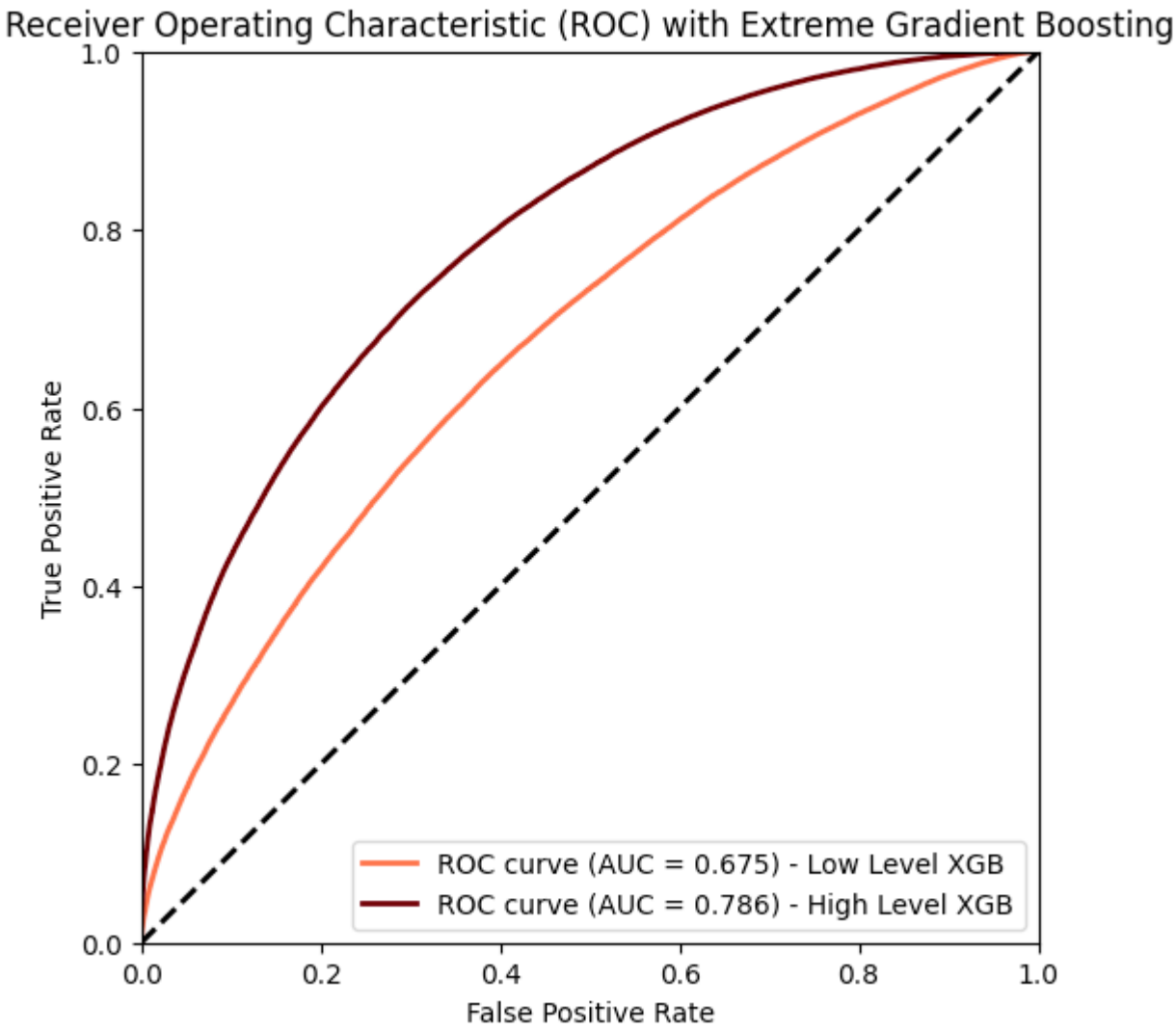
```
[40]    validation_0-merror:0.28131     validation_1-merror:0.29536
[41]    validation_0-merror:0.28100     validation_1-merror:0.29538
[42]    validation_0-merror:0.28077     validation_1-merror:0.29506
[43]    validation_0-merror:0.28043     validation_1-merror:0.29500
[44]    validation_0-merror:0.28012     validation_1-merror:0.29502
[45]    validation_0-merror:0.27979     validation_1-merror:0.29488
[46]    validation_0-merror:0.27959     validation_1-merror:0.29495
[47]    validation_0-merror:0.27904     validation_1-merror:0.29487
[48]    validation_0-merror:0.27883     validation_1-merror:0.29475
[49]    validation_0-merror:0.27849     validation_1-merror:0.29467

Time taken for running Extreme Gradient Boosting: 0.24 minutes.

Accuracy: 0.708
Precision: 0.703
Recall: 0.720
```

In [73]:
```
XGBTrainValidationLossPlot(evals_result_XGB_hl, 'High-Level features')
```



The validation loss curve closely follows the train loss curve. The process is halted at boosting round 50 because the values start to diverge.

In [75]:
```
ROC_AUC(y_test, [[y_pred_proba_XGB_ll, y_pred_proba_XGB_hl],], 'Extreme Gradient Boosting')
```



We can interpret the AUC values for this model as follows:

1. **Low-Level Extreme Gradient Boosting AUC = 0.675**:

This value falls between 0.5 and 0.7, which is considered reasonable. It indicates that the model's performance in distinguishing between classes is not very strong but better than random guessing. It suggests that while the Low-Level features model has some ability to differentiate between positive and negative classes, there is significant room for improvement.

2. **High-Level Extreme Gradient Boosting AUC = 0.786**:

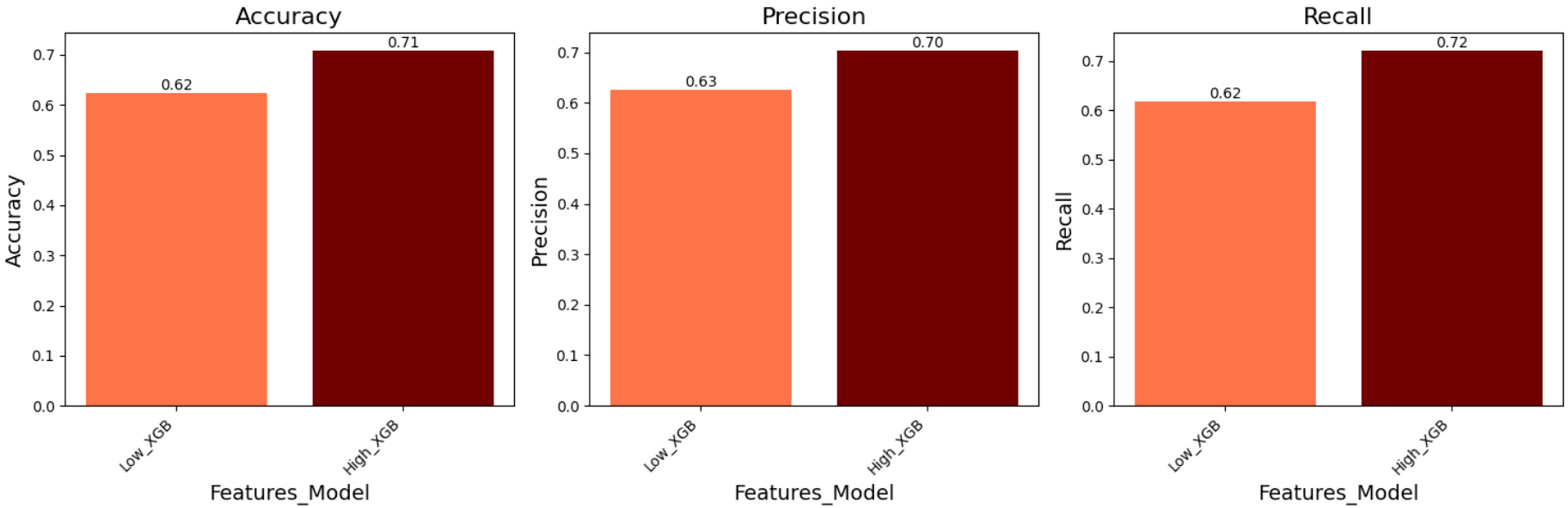
This value falls between 0.7 and 0.8, which is considered acceptable. It indicates that the model performs adequately in distinguishing between classes. The High-Level features model shows a better capability to differentiate between classes compared to the low-level feature model, reflecting a noticeable improvement in performance.

Based on the AUC values and their interpretation, the conclusion for this model's performance is: **This model, although powerful, has however difficulties discovering patterns in the Low-Level features**, as indicated by the AUC of 0.675, which is considered reasonable but not strong. On the other hand, the model performs well with High-Level features, as demonstrated by the AUC of 0.786, which falls into the acceptable range, indicating a good capability to distinguish between Background and Signal classes. This suggests that the model benefits significantly from High-Level features and can effectively leverage them to improve its predictive performance.

5.1.4. Results

Now we plot the statistical results (Accuracy, Precion and Recall) for the Extreme Gradient Boosting model.

In [78]:
```
# Dictionary to store the results
report = {'model': [['Low_XGB', 'High_XGB']],
          'Accuracy': [[acc_XGB_ll, acc_XGB_hl]],
          'Precision': [[prec_XGB_ll, prec_XGB_hl]],
          'Recall': [[rec_XGB_ll, rec_XGB_hl]]
         }

ResultsPlot(report)
```



5.2. Multilayer Perceptrons

Multilayer Perceptrons (MLP) are a type of artificial neural network composed of multiple layers of interconnected neurons, structured as follows:

1. **Neurons (Units)**: Neurons receive input signals, process them, and produce output signals. They are organized into layers: input, hidden, and output.

2. **Layers**:

   - *Input Layer*: Receives raw input data, with each neuron representing a feature.
   - *Hidden Layers*: Perform nonlinear transformations of input data to learn abstract representations.
   - *Output Layer*: Produces final predictions or outputs based on learned representations.

3. **Connections and Weights**: Neurons in adjacent layers are connected by weighted connections, determining signal strength. Weights are adjusted during training to minimize prediction errors.

4. **Activation Functions**: Introduce nonlinearity into the network, enabling complex mappings between inputs and outputs. Common

functions include sigmoid, tanh, and ReLU.

5. **Feedforward Propagation**: MLP use feedforward propagation to compute predictions. Input data passes through the network, undergoing weighted sums and activation functions at each layer.

6. **Training**: Typically conducted using supervised learning, where the network learns to map input data to target outputs. Training involves forward propagation, backpropagation for gradient computation, and weight optimization.

7. **Versatility and Performance**: MLP are versatile models capable of learning complex patterns in data. They may require careful tuning and regularization to prevent overfitting, especially with high-dimensional or noisy data.

Below are the differences between the two different MLP implementation architectures used: in this project:

1. **Keras/TensorFlow**. Keras is a high-level neural network API that runs on top of TensorFlow, a powerful framework for building, training, and deploying Deep Learning models. It offers extensive flexibility in defining neural network architectures, including the ability to create complex models with various types of layers (e.g., Dense, Convolutional, Recurrent) and custom activation functions. Keras also supports advanced configurations like dropout regularization, batch normalization, and custom training loops. It is well-suited for tasks requiring deep customization and control over the neural network architecture.

2. **Scikit-learn**. Scikit-learn, on the other hand, is a general-purpose machine learning library in Python focused on simplicity and ease of use. It includes an MLP implementation primarily designed for traditional machine learning tasks like classification and regression. The MLP in Scikit-learn is defined using parameters such as hidden_layer_sizes, activation, and solver, offering a simpler approach to constructing neural networks compared to Keras. Scikit-learn integrates well with other scientific Python libraries and provides straightforward methods for model training, evaluation, and deployment.

In the following two MLP models, the importance given to Early Stopping and Loss Monitoring can be explained as follows:

1. **Early Stopping**: This mechanism helps prevent overfitting by stopping training when validation performance no longer improves. By doing so, it ensures that the model can generalize well to new data, as evidenced by a consistent reduction in validation loss.

2. **Train/Validation Loss**: Monitoring these losses provides crucial insights into the model's performance. Significant discrepancies between training and validation losses may indicate overfitting, where the model memorizes specific details of the training data instead of learning general patterns.

These practices are essential for developing models that not only fit the training data well but also generalize effectively to unseen data, thereby enhancing their utility and reliability in practical applications.

5.2.1. Multilayer Perceptrons with Keras/TensorFlow

To fine-tune the number of neurons in dense layers, dropout rate, and batch size in your MLPKeras function, we follow these guidelines:

1. **Number of Neurons** in Dense layers: Adjusts the value to scale neurons in each dense layer. It's typical to start with a baseline number of neurons per layer, such as, for larger datasets, 128, or 256.

2. **Dropout Rate** in Dropout layers: It refers to a regularization technique used to prevent overfitting during the training process, and works by randomly deactivating (or "dropping out") a fraction of neurons in a layer during each training iteration.

3. **Batch Size** in Fit function: Determines the number of samples processed before updating model weights, influencing training speed and generalization. If we have a large dataset and ample GPU memory, larger batch sizes might be suitable for faster training; for smaller datasets or when training on CPUs, smaller batch sizes might be necessary to fit into memory and ensure the model learns well.

In the following function, **tracking time** (start_time, end_time, run_time) ensures transparency regarding the computational cost of training.

In [81]:
```
def MLPKeras(X_train, y_train, X_val, y_val, X_test, y_test, COLS, \
             neurons_base, dropout_rate_coef, batch_size):
    """
    Train and predict with Multilayer Perceptrons with Keras.

    Parameters:
        X_train (pd.DataFrame): Training features.
        y_train (Union[pd.Series, np.ndarray]): Training labels.
        X_val (pd.DataFrame): Validation features.
        y_val (Union[pd.Series, np.ndarray]): Validation labels.
        X_test (pd.DataFrame): Test features.
        y_test (Union[pd.Series, np.ndarray]): Test labels.
        COLS (List[str]): Set of features/columns with which the model is trained and predicts.
        neurons_base (int): Base value for neurons.
        dropout_rate_coef (float): Dropout Rate Coefficient.
```

```
    batch_size (int): Batch Size.

Returns:
    Tuple: A tuple containing the following elements:
        - statistics (Tuple[float, float, float]): Evaluation metrics for the test set (accuracy, precision, re
        - history (tf.keras.callbacks.History): Training history of the model.
        - y_pred_proba (np.ndarray): Predicted probabilities for the test set.
        - mlpk_clf (tensorflow.keras.Model): Trained Keras MLP model.
"""
# Import necessary libraries
import time
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

# Print start time
print("Start time:", time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))

# Record start time
start_time = time.time()

# Initialize MLPKeras model using the Functional API
input_shape = len(COLS)
inputs = Input(shape=(input_shape,))
x = Dense(neurons_base * 1, activation='relu')(inputs)
x = Dropout(dropout_rate_coef * 0.1)(x)
x = Dense(neurons_base * 2, activation='relu')(x)
x = Dropout(dropout_rate_coef * 0.1)(x)
x = Dense(neurons_base * 4, activation='relu')(x)
x = Dropout(dropout_rate_coef * 0.1)(x)
x = Dense(neurons_base * 8, activation='relu')(x)
x = Dropout(dropout_rate_coef * 0.1)(x)
x = Dense(neurons_base * 4, activation='relu')(x)
x = Dropout(dropout_rate_coef * 0.1)(x)
outputs = Dense(1, activation='sigmoid')(x)

mlpk_clf = Model(inputs=inputs, outputs=outputs)

# Print summary of the model
mlpk_clf.summary()
```

```
# Compile the model
print("Compiling the Model...")
mlpk_clf.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# Define early stopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
    verbose=1,
    restore_best_weights=True
)

# Train the model on the training data
print("Training the Model...")
history = mlpk_clf.fit(
    X_train[COLS],
    y_train,
    epochs=100,
    batch_size=batch_size,
    validation_data=(X_val[COLS], y_val),
    callbacks=[early_stopping]
)

# Predict using the trained model
print("Predicting on the Testset...")
y_pred_proba = mlpk_clf.predict(X_test[COLS])
y_pred = (y_pred_proba > 0.5).astype(int)

# Record end time
end_time = time.time()

# Calculate and print time taken
run_time = end_time - start_time
print(f"Time taken for running Multilayer Perceptrons with Keras: {run_time/60:.2f} minutes.")

# Return metrics, history, predicted labels and model
```

```
    return Statistics(y_test, y_pred), history, y_pred_proba, mlpk_clf
```

The following UMAP function visualizes the UMAP projection of the second-to-last dense layer outputs from a trained Keras MLP model. Applying UMAP (Uniform Manifold Approximation and Projection) to the penultimate layer of Deep Learning models is particularly useful for several reasons:

1. **Capturing Abstract Representations**: The penultimate layer of a deep neural network often contains abstract and semantic representations of the input data. As data is processed through the layers of the network, it is transformed into higher-level representations that encapsulate complex and task-relevant features.

2. **Dimensionality Reduction**: Representations in the penultimate layer typically have a relatively high dimensionality, which can make visualization and direct interpretation of learned patterns challenging. Reducing the dimensionality of these representations with UMAP preserves important data relationships while projecting them into a lower-dimensional space, facilitating visualization and analysis.

3. **Interpretability**: Applying UMAP to the penultimate layer allows exploration of how the model organizes and separates data in a simpler and more interpretable representation space. This can reveal natural clusters in the data, separation patterns between classes, or important features learned by the model.

4. **Efficient Visualization**: Visualizing outputs from the penultimate layer provides insights into how the model processes and understands data. UMAP enables the creation of visualizations that help data scientists and researchers better understand the model's behavior and interpret its decisions.

5. **Model Comparison**: By applying UMAP to outputs from the penultimate layer of different models, it is possible to compare how different neural network architectures or training parameters result in different data representations. This can be useful for model tuning or identifying specific aspects of the problem that each model captures.

In summary, applying UMAP to the penultimate layer of deep learning is a powerful technique for exploring and interpreting the internal representations learned by the model, providing a clearer and more intuitive view of the learning process and data characteristics.

In [83]:
```
def MLPKerasUMAP(mlpk_clf, TrainSet, y_train, columns):
    """
    Visualizes the UMAP projection of the second-to-last dense layer outputs from a given Keras MLP model.

    Parameters:
```

```
    mlpk_clf (tensorflow.keras.Model): Trained Keras MLP model.
    TrainSet (numpy.ndarray): Training data used to obtain the intermediate layer outputs.
    y_train (numpy.ndarray): Labels corresponding to the training data.
    columns (str): String to describe the columns being analyzed.

Returns:
    None
"""
# Import libraries
import time
from tensorflow.keras import Model
import umap.umap_ as umap
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

# Print start time
print("Start time:", time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))

# Record start time
start_time = time.time()

# Create a model to output the second-to-last dense layer
intermediate_layer_model = Model(inputs=mlpk_clf.input, outputs=mlpk_clf.layers[-3].output)

# Get the intermediate layer outputs for the training data
print("Predicting on the Second-to-last Dense Layer...")
intermediate_output = intermediate_layer_model.predict(TrainSet)

# Use UMAP to reduce dimensionality
reducer = umap.UMAP()
print("Embedding...")
embedding = reducer.fit_transform(intermediate_output)

# Plot the UMAP embedding
print("Plotting...")
plt.figure(figsize=(8, 6))
cmap = plt.cm.viridis
plt.scatter(embedding[:, 0], embedding[:, 1], c=y_train, cmap=cmap, s=5, alpha=0.5)

# Create custom legend
legend_elements = [
```

```
        Line2D([0], [0], marker='o', color='w', markerfacecolor=cmap(cmap.N), markersize=10, label='Signal'),
        Line2D([0], [0], marker='o', color='w', markerfacecolor=cmap(0), markersize=10, label='Background')    ]
    plt.legend(handles=legend_elements, title='Classes', loc='lower right')

    plt.title(f"UMAP projection of the second-to-last layer outputs for {columns} with Keras MLP")
    plt.xlabel('UMAP Dimension 1')
    plt.ylabel('UMAP Dimension 2')
    plt.show()

    # Record end time
    end_time = time.time()

    # Calculate and print time taken
    run_time = end_time - start_time
    print(f"Time taken for Embedding and Plotting: {run_time/60:.2f} minutes.")
```

The following Train/Validation Loss Plot function is used to visualize the training and validation loss curves of a Keras MLP model across epochs. It takes a history object as input, which contains recorded metrics from the model training process, such as loss values for both training and validation data. By plotting these loss curves, the function provides insights into how well the model is learning over time. The x-axis represents the number of epochs, while the y-axis shows the corresponding loss values. The plot helps in understanding:

1. **Training Progress**: It shows whether the model is improving its performance on the training data as epochs progress.
2. **Generalization Capability**: By comparing the training and validation loss curves, it indicates how well the model generalizes to unseen validation data. Large discrepancies between these curves might suggest overfitting or underfitting.

This visualization, being crucial for monitoring training progress and assessing model convergence and potential issues like overfitting or underfitting, helps make informed decisions regarding model adjustments or training strategies.

In [85]:
```
def MLPKerasTrainValidationLossPlot(history, columns):
    """
    Plot training and validation loss for a MLPKeras model.

    Parameters:
        history (History class): History object returned by the fit method of a Keras model.
        columns (str): String to describe the columns being analyzed.

    Returns:
```

```
        None.
    """
    # Import libraries
    import matplotlib.pyplot as plt
    import numpy as np

    # Get the number of epochs
    epochs = range(1, len(history.history['loss']) + 1)

    # Plot the training and validation loss
    plt.figure(figsize=(8, 3))
    plt.plot(epochs, history.history['loss'], label='Training Loss')
    plt.plot(epochs, history.history['val_loss'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')

    # Set x-ticks at regular intervals
    plt.xticks(np.arange(0, max(epochs), 5))

    plt.title(f"Training and Validation Loss for {columns} with Keras MLP")
    plt.legend()
    plt.show()
```

5.2.1.1. Low Level features

We start by training and predicting with the Low-Level features using Keras Multilayer Perceptrons model.

In [87]:
```
# Multilayer Perceptrons with Keras with Low Level features
(acc_MLPK_ll, prec_MLPK_ll, rec_MLPK_ll), history_MLPK_ll, y_pred_proba_MLPK_ll, clf_MLPK_ll = MLPKeras(
    X_train, y_train, X_val, y_val, X_test, y_test, LowLevelCols, \
    128, 1, 64
)
```

```
2024-07-03 22:48:40.095699: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized
to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other operations, rebuild TensorFlow with the
appropriate compiler flags.
Start time: 2024-07-03 22:48:44
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 17)	0
dense (Dense)	(None, 128)	2,304
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 256)	33,024
dropout_1 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 512)	131,584
dropout_2 (Dropout)	(None, 512)	0
dense_3 (Dense)	(None, 1024)	525,312
dropout_3 (Dropout)	(None, 1024)	0
dense_4 (Dense)	(None, 512)	524,800
dropout_4 (Dropout)	(None, 512)	0
dense_5 (Dense)	(None, 1)	513

Total params: 1,217,537 (4.64 MB)

Trainable params: 1,217,537 (4.64 MB)

Non-trainable params: 0 (0.00 B)

```
Compiling the Model...
Training the Model...
Epoch 1/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 194s 27ms/step - accuracy: 0.5908 - loss: 0.6655 - val_accuracy: 0.6321 - val_loss:
0.6369
Epoch 2/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 249s 35ms/step - accuracy: 0.6320 - loss: 0.6380 - val_accuracy: 0.6433 - val_loss:
0.6274
Epoch 3/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 242s 32ms/step - accuracy: 0.6416 - loss: 0.6303 - val_accuracy: 0.6482 - val_loss:
0.6217
Epoch 4/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 237s 33ms/step - accuracy: 0.6483 - loss: 0.6252 - val_accuracy: 0.6522 - val_loss:
0.6239
Epoch 5/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 233s 29ms/step - accuracy: 0.6503 - loss: 0.6221 - val_accuracy: 0.6531 - val_loss:
0.6182
Epoch 6/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 222s 31ms/step - accuracy: 0.6521 - loss: 0.6206 - val_accuracy: 0.6541 - val_loss:
0.6183
Epoch 7/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 226s 31ms/step - accuracy: 0.6548 - loss: 0.6192 - val_accuracy: 0.6564 - val_loss:
0.6162
Epoch 8/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 232s 32ms/step - accuracy: 0.6573 - loss: 0.6173 - val_accuracy: 0.6563 - val_loss:
0.6149
Epoch 9/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 232s 32ms/step - accuracy: 0.6595 - loss: 0.6149 - val_accuracy: 0.6574 - val_loss:
0.6138
Epoch 10/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 233s 32ms/step - accuracy: 0.6601 - loss: 0.6134 - val_accuracy: 0.6587 - val_loss:
0.6150
Epoch 11/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 251s 31ms/step - accuracy: 0.6612 - loss: 0.6125 - val_accuracy: 0.6594 - val_loss:
0.6149
Epoch 12/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 262s 31ms/step - accuracy: 0.6623 - loss: 0.6111 - val_accuracy: 0.6595 - val_loss:
0.6131
Epoch 13/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 261s 31ms/step - accuracy: 0.6636 - loss: 0.6092 - val_accuracy: 0.6577 - val_loss:
0.6135
```

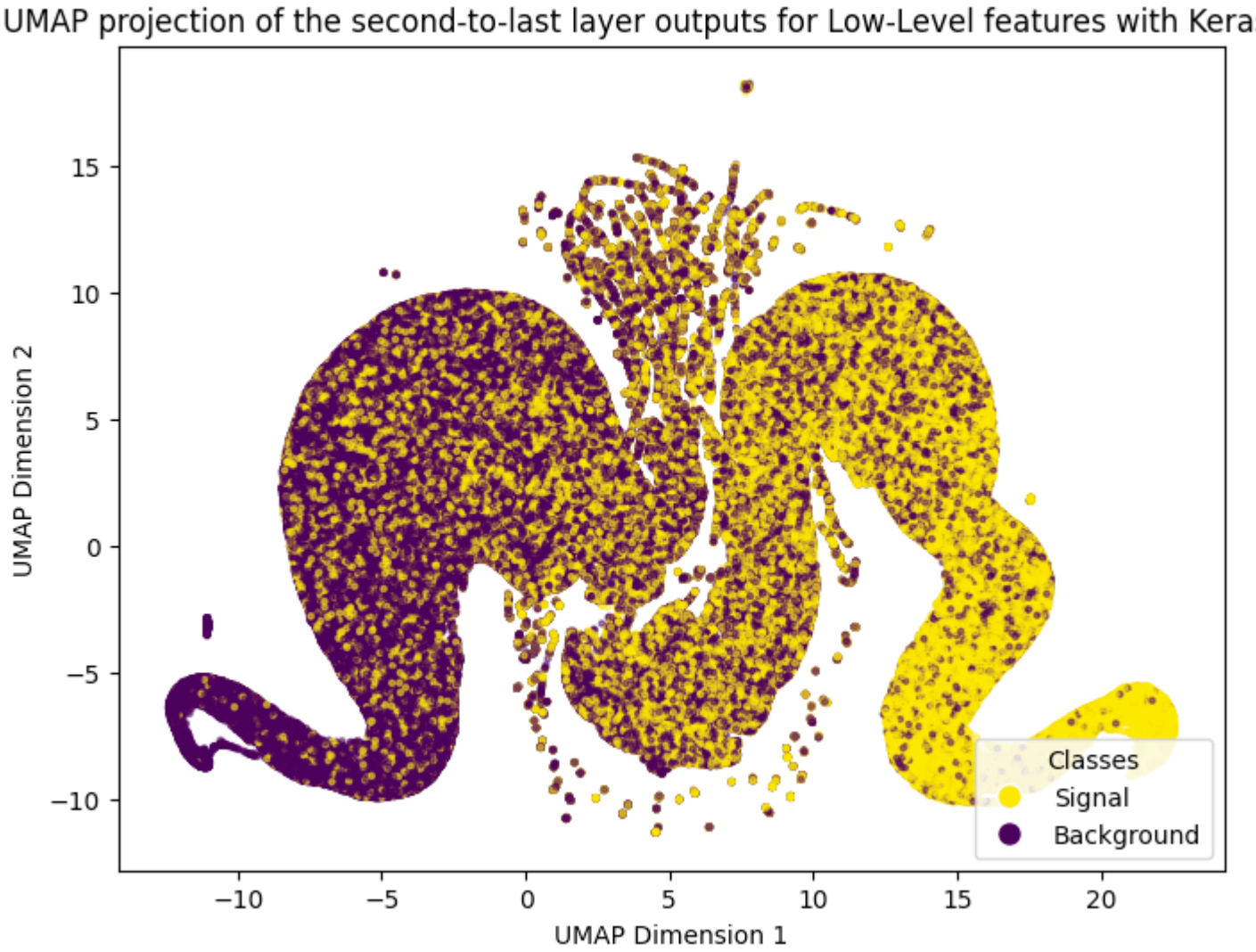
```
Epoch 14/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 238s 33ms/step - accuracy: 0.6633 - loss: 0.6093 - val_accuracy: 0.6609 - val_loss:
0.6111
Epoch 15/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 234s 32ms/step - accuracy: 0.6633 - loss: 0.6097 - val_accuracy: 0.6616 - val_loss:
0.6128
Epoch 16/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 239s 33ms/step - accuracy: 0.6657 - loss: 0.6078 - val_accuracy: 0.6601 - val_loss:
0.6111
Epoch 17/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 239s 33ms/step - accuracy: 0.6661 - loss: 0.6080 - val_accuracy: 0.6628 - val_loss:
0.6107
Epoch 18/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 236s 33ms/step - accuracy: 0.6684 - loss: 0.6060 - val_accuracy: 0.6602 - val_loss:
0.6141
Epoch 19/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 246s 34ms/step - accuracy: 0.6675 - loss: 0.6061 - val_accuracy: 0.6610 - val_loss:
0.6119
Epoch 20/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 250s 33ms/step - accuracy: 0.6688 - loss: 0.6050 - val_accuracy: 0.6618 - val_loss:
0.6129
Epoch 21/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 241s 33ms/step - accuracy: 0.6700 - loss: 0.6045 - val_accuracy: 0.6630 - val_loss:
0.6101
Epoch 22/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 243s 34ms/step - accuracy: 0.6716 - loss: 0.6026 - val_accuracy: 0.6631 - val_loss:
0.6091
Epoch 23/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 253s 33ms/step - accuracy: 0.6706 - loss: 0.6025 - val_accuracy: 0.6628 - val_loss:
0.6090
Epoch 24/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 240s 33ms/step - accuracy: 0.6721 - loss: 0.6019 - val_accuracy: 0.6636 - val_loss:
0.6104
Epoch 25/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 241s 33ms/step - accuracy: 0.6719 - loss: 0.6025 - val_accuracy: 0.6626 - val_loss:
0.6088
Epoch 26/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 247s 34ms/step - accuracy: 0.6708 - loss: 0.6033 - val_accuracy: 0.6632 - val_loss:
0.6099
Epoch 27/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 241s 34ms/step - accuracy: 0.6713 - loss: 0.6016 - val_accuracy: 0.6642 - val_loss:
```

```
0.6095
Epoch 28/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 238s 33ms/step - accuracy: 0.6740 - loss: 0.5999 - val_accuracy: 0.6637 - val_loss:
0.6085
Epoch 29/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 239s 33ms/step - accuracy: 0.6747 - loss: 0.5995 - val_accuracy: 0.6625 - val_loss:
0.6094
Epoch 30/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 238s 33ms/step - accuracy: 0.6742 - loss: 0.5996 - val_accuracy: 0.6636 - val_loss:
0.6085
Epoch 31/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 238s 33ms/step - accuracy: 0.6744 - loss: 0.5989 - val_accuracy: 0.6622 - val_loss:
0.6080
Epoch 32/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 238s 33ms/step - accuracy: 0.6732 - loss: 0.5993 - val_accuracy: 0.6634 - val_loss:
0.6089
Epoch 33/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 237s 33ms/step - accuracy: 0.6740 - loss: 0.5989 - val_accuracy: 0.6626 - val_loss:
0.6091
Epoch 34/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 241s 34ms/step - accuracy: 0.6748 - loss: 0.5986 - val_accuracy: 0.6619 - val_loss:
0.6094
Epoch 35/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 236s 33ms/step - accuracy: 0.6759 - loss: 0.5967 - val_accuracy: 0.6628 - val_loss:
0.6088
Epoch 36/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 236s 33ms/step - accuracy: 0.6756 - loss: 0.5979 - val_accuracy: 0.6640 - val_loss:
0.6076
Epoch 37/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 237s 33ms/step - accuracy: 0.6744 - loss: 0.5983 - val_accuracy: 0.6638 - val_loss:
0.6078
Epoch 38/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 240s 33ms/step - accuracy: 0.6762 - loss: 0.5974 - val_accuracy: 0.6645 - val_loss:
0.6070
Epoch 39/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 236s 33ms/step - accuracy: 0.6745 - loss: 0.5977 - val_accuracy: 0.6640 - val_loss:
0.6075
Epoch 40/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 238s 33ms/step - accuracy: 0.6778 - loss: 0.5953 - val_accuracy: 0.6627 - val_loss:
0.6093
Epoch 41/100
```

```
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 239s 33ms/step - accuracy: 0.6770 - loss: 0.5959 - val_accuracy: 0.6645 - val_loss:
0.6072
Epoch 42/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 241s 34ms/step - accuracy: 0.6775 - loss: 0.5959 - val_accuracy: 0.6631 - val_loss:
0.6105
Epoch 43/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 246s 34ms/step - accuracy: 0.6768 - loss: 0.5949 - val_accuracy: 0.6650 - val_loss:
0.6083
Epoch 43: early stopping
Restoring model weights from the end of the best epoch: 38.
Predicting on the Testset...
8814/8814 ━━━━━━━━━━━━━━━━━━━━ 45s 5ms/step
Time taken for running Multilayer Perceptrons with Keras: 172.26 minutes.
Accuracy: 0.664
Precision: 0.672
Recall: 0.640
```

In [88]:
```
MLPKerasUMAP(clf_MLPK_ll, X_train[LowLevelCols], y_train, 'Low-Level features')
```

```
Start time: 2024-07-04 01:41:11
Predicting on the Second-to-last Dense Layer...
14397/14397 ━━━━━━━━━━━━━━━━━━━━ 78s 5ms/step
Embedding...
Plotting...
```




```
Time taken for Embedding and Plotting: 25.23 minutes.
```

In the UMAP projection for the Low-Level features, the following observations can be made:

1. **Two Clusters**: There are two main clusters:
   - One at the left bottom, labeled as "Background".
   - One at the right bottom, labeled as "Signal".

2. **Mixed Main Area**: The central area of the UMAP projection is occupied by a mix of data points from both "Signal" and "Background" labels. This suggests that there is no clear separation between the two clusters in this region, indicating significant overlap or similarity in features.

3. **Outliers**: There are two sets of outliers that are distinct from the main area and clusters. These outliers are situated away from both the central mixed area and the main clusters, indicating they have unique characteristics.

We can draw the following conclusions about this clustering:

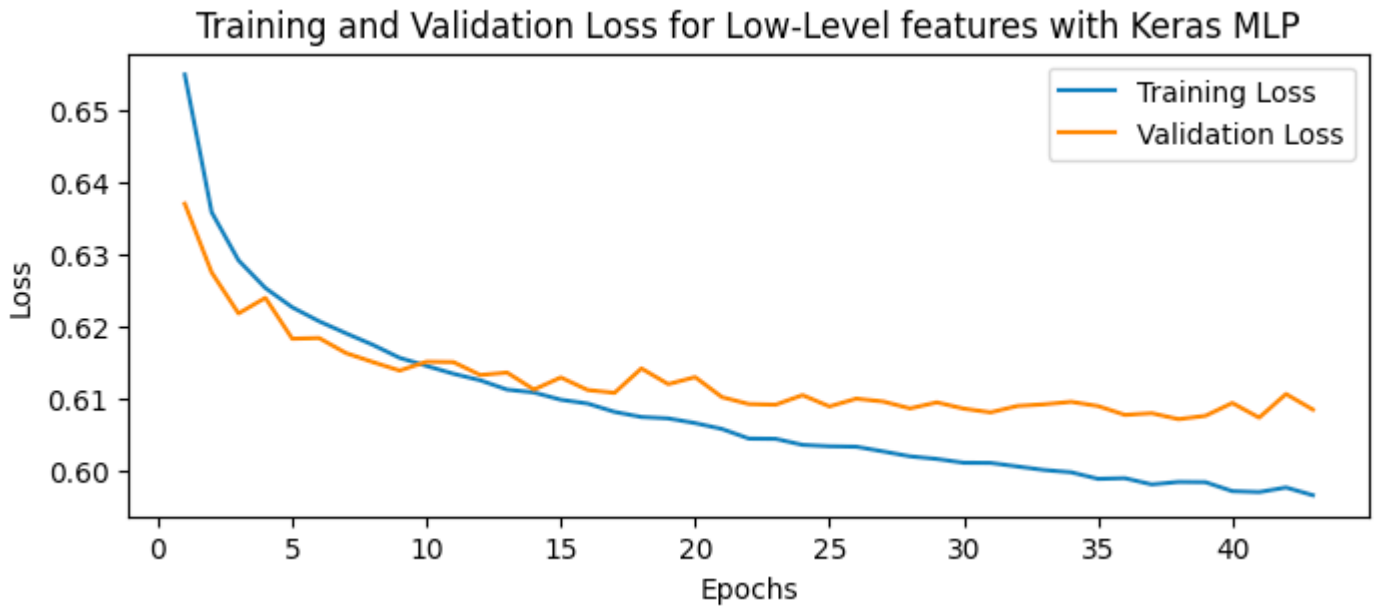
1. **Distinct Clusters with Mixed Region**: The data points form two primary clusters, "Signal" and "Background," but the main area of the UMAP projection shows a mix of both labels. This implies that while there are distinct groups, there is a significant portion of the data where the features of "Signal" and "Background" overlap.

2. **Overlap Indicates Ambiguity**: The mixing of "Signal" and "Background" labels in the central area suggests that the boundary between these two clusters is not well-defined. This could indicate that the features used for clustering do not completely differentiate between the two labels, leading to ambiguity in classification.

3. **Outliers**: The presence of outliers, which do not fit into the main clusters or the mixed central area, indicates there are data points with unique features. These outliers may represent noise, rare cases, or potentially new subgroups that are not captured by the main clusters.

4. **Implications for Further Analysis**: The overlap in the main area suggests that the model requires a larger dataset. It could be also beneficial to analyze the features of the data points in the mixed region to understand the source of overlap better and to explore if additional features or a different clustering approach might improve separation. The outliers should be investigated to determine their nature and relevance to the overall data structure.

In [90]:
```
MLPKerasTrainValidationLossPlot(history_MLPK_ll, 'Low-Level features')
```



The validation loss curve closely follows the train loss curve. The process is halted at epoch 43 because the values start to diverge.

5.2.1.2. High Level features

We finally train and predict using Keras Multilayer Perceptrons model with the High-Level features.

In [93]:
```
# Multilayer Perceptrons with Keras with High Level features
(acc_MLPK_hl, prec_MLPK_hl, rec_MLPK_hl), history_MLPK_hl, y_pred_proba_MLPK_hl, clf_MLPK_hl = MLPKeras(
    X_train, y_train, X_val, y_val, X_test, y_test, HighLevelCols, \
    128, 1, 64
)
```

```
Start time: 2024-07-04 02:06:37
Model: "functional_5"
```

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 7)	0
dense_6 (Dense)	(None, 128)	1,024
dropout_5 (Dropout)	(None, 128)	0
dense_7 (Dense)	(None, 256)	33,024
dropout_6 (Dropout)	(None, 256)	0
dense_8 (Dense)	(None, 512)	131,584
dropout_7 (Dropout)	(None, 512)	0
dense_9 (Dense)	(None, 1024)	525,312
dropout_8 (Dropout)	(None, 1024)	0
dense_10 (Dense)	(None, 512)	524,800
dropout_9 (Dropout)	(None, 512)	0
dense_11 (Dense)	(None, 1)	513

Total params: 1,216,257 (4.64 MB)

Trainable params: 1,216,257 (4.64 MB)

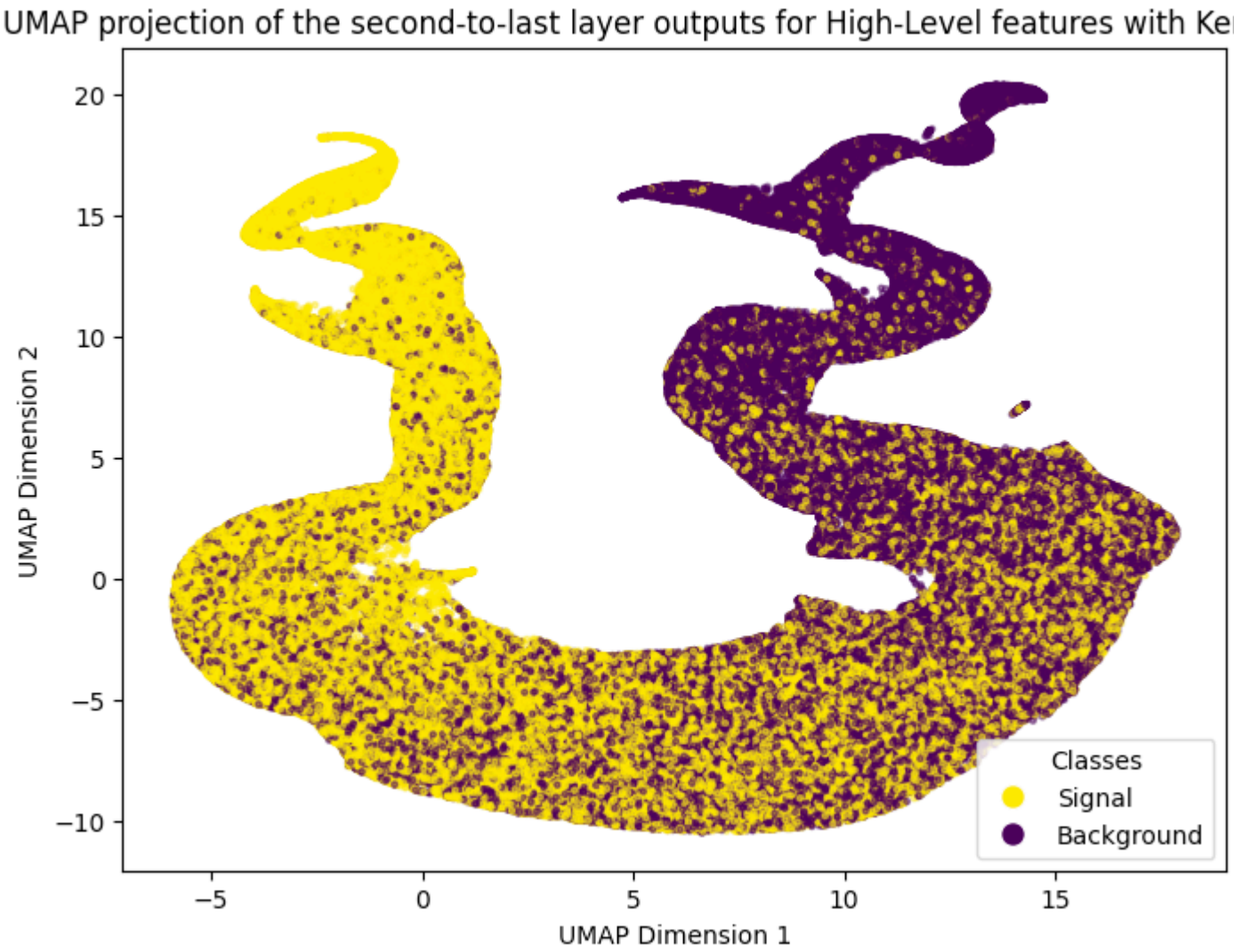
Non-trainable params: 0 (0.00 B)

```
Compiling the Model...
Training the Model...
Epoch 1/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 193s 26ms/step - accuracy: 0.6816 - loss: 0.5867 - val_accuracy: 0.6989 - val_loss:
0.5613
Epoch 2/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 191s 26ms/step - accuracy: 0.6982 - loss: 0.5658 - val_accuracy: 0.7008 - val_loss:
0.5622
Epoch 3/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 203s 27ms/step - accuracy: 0.7032 - loss: 0.5602 - val_accuracy: 0.7047 - val_loss:
0.5581
Epoch 4/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 201s 28ms/step - accuracy: 0.7030 - loss: 0.5595 - val_accuracy: 0.7054 - val_loss:
0.5568
Epoch 5/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 193s 27ms/step - accuracy: 0.7043 - loss: 0.5577 - val_accuracy: 0.7055 - val_loss:
0.5557
Epoch 6/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 195s 27ms/step - accuracy: 0.7055 - loss: 0.5571 - val_accuracy: 0.7065 - val_loss:
0.5559
Epoch 7/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 195s 27ms/step - accuracy: 0.7058 - loss: 0.5565 - val_accuracy: 0.7071 - val_loss:
0.5548
Epoch 8/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 196s 27ms/step - accuracy: 0.7067 - loss: 0.5560 - val_accuracy: 0.7075 - val_loss:
0.5524
Epoch 9/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 203s 28ms/step - accuracy: 0.7081 - loss: 0.5556 - val_accuracy: 0.7065 - val_loss:
0.5537
Epoch 10/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 199s 28ms/step - accuracy: 0.7088 - loss: 0.5536 - val_accuracy: 0.7067 - val_loss:
0.5534
Epoch 11/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 199s 28ms/step - accuracy: 0.7082 - loss: 0.5530 - val_accuracy: 0.7079 - val_loss:
0.5585
Epoch 12/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 202s 28ms/step - accuracy: 0.7078 - loss: 0.5535 - val_accuracy: 0.7063 - val_loss:
0.5545
Epoch 13/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 203s 28ms/step - accuracy: 0.7088 - loss: 0.5523 - val_accuracy: 0.7085 - val_loss:
0.5515
```

```
Epoch 14/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 204s 28ms/step - accuracy: 0.7076 - loss: 0.5530 - val_accuracy: 0.7077 - val_loss:
0.5552
Epoch 15/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 206s 29ms/step - accuracy: 0.7096 - loss: 0.5513 - val_accuracy: 0.7069 - val_loss:
0.5551
Epoch 16/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 206s 29ms/step - accuracy: 0.7088 - loss: 0.5523 - val_accuracy: 0.7068 - val_loss:
0.5529
Epoch 17/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 207s 29ms/step - accuracy: 0.7088 - loss: 0.5522 - val_accuracy: 0.7073 - val_loss:
0.5531
Epoch 18/100
7199/7199 ━━━━━━━━━━━━━━━━━━━━ 209s 29ms/step - accuracy: 0.7085 - loss: 0.5524 - val_accuracy: 0.7088 - val_loss:
0.5539
Epoch 18: early stopping
Restoring model weights from the end of the best epoch: 13.
Predicting on the Testset...
8814/8814 ━━━━━━━━━━━━━━━━━━━━ 37s 4ms/step
Time taken for running Multilayer Perceptrons with Keras: 60.74 minutes.
Accuracy: 0.711
Precision: 0.719
Recall: 0.694
```

In [94]:
```
MLPKerasUMAP(clf_MLPK_hl, X_train[HighLevelCols], y_train, 'High-Level features')
```

```
Start time: 2024-07-04 03:07:22
Predicting on the Second-to-last Dense Layer...
14397/14397 ━━━━━━━━━━━━━━━━━━━━ 58s 4ms/step
Embedding...
Plotting...
```




```
Time taken for Embedding and Plotting: 14.25 minutes.
```

In the UMAP projection for the High-Level features, the following observations can be made:

1. **Larger Clusters**: Compared to the clusters obtained with the Low-Level features, the two main clusters have become slightly larger:
    - One on the top left labeled as "Signal."

- One on the top right labeled as "Background."

2. **Mixed Main Area**: The central area, extending from the left to the right, is occupied by a mix of data points from both "Signal" and "Background" labels. This suggests significant overlap or similarity in features in this region.

3. **Reduced Outliers**: The majority of the outliers observed in the Low-Level features projection have disappeared. The data points are now more concentrated within the two main clusters and the mixed central area.

We can draw the following conclusions about this clustering:

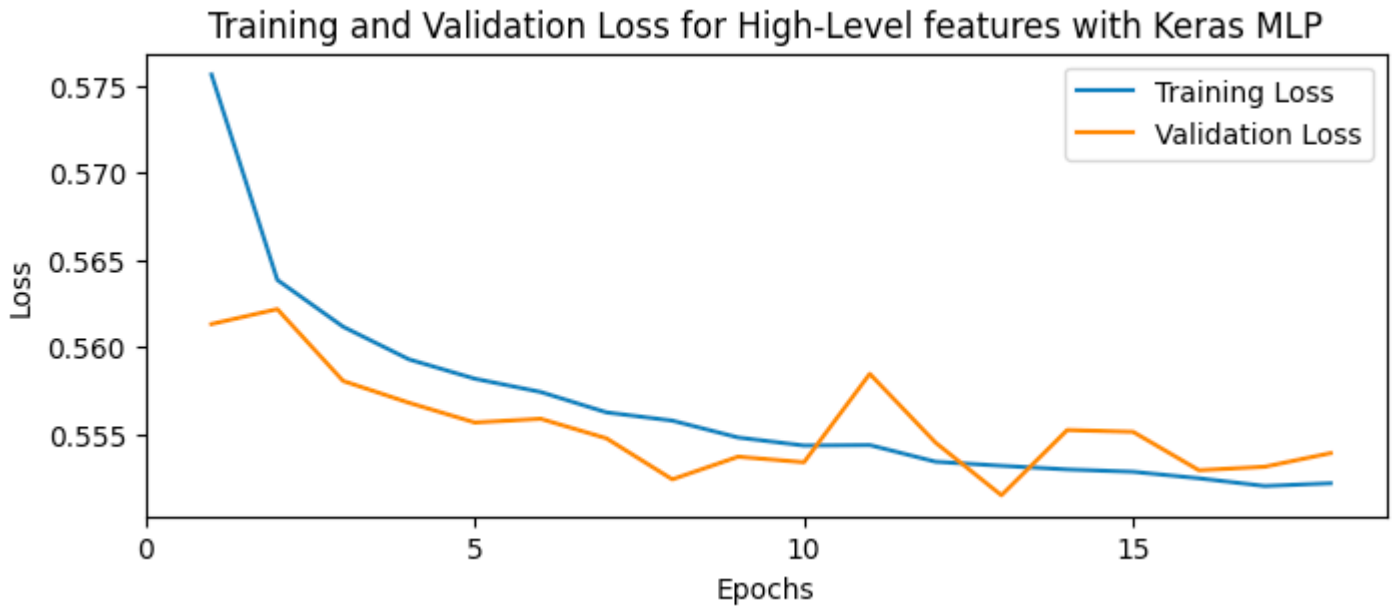
1. **Expanded Clusters**: The "Background" and "Signal" clusters are now slightly larger, indicating that the High-Level features have helped to group more data points into the main clusters, reducing ambiguity.

2. **Decreased Ambiguity in Overlap**: The central mixed area, which extends from the bottom left to the bottom right, is still present, but with a significantly lower number of outliers, suggesting that the High-Level features provide a clearer differentiation between the two labels. The overlap remains, indicating some ambiguity, but it has decreased compared to the Low-Level features.

3. **Outliers Reduction**: The reduction in the number of outliers implies that the High-Level features have captured more relevant information, bringing previously outlying points into the main clusters or mixed area. This suggests a more robust clustering model with better-defined groups.

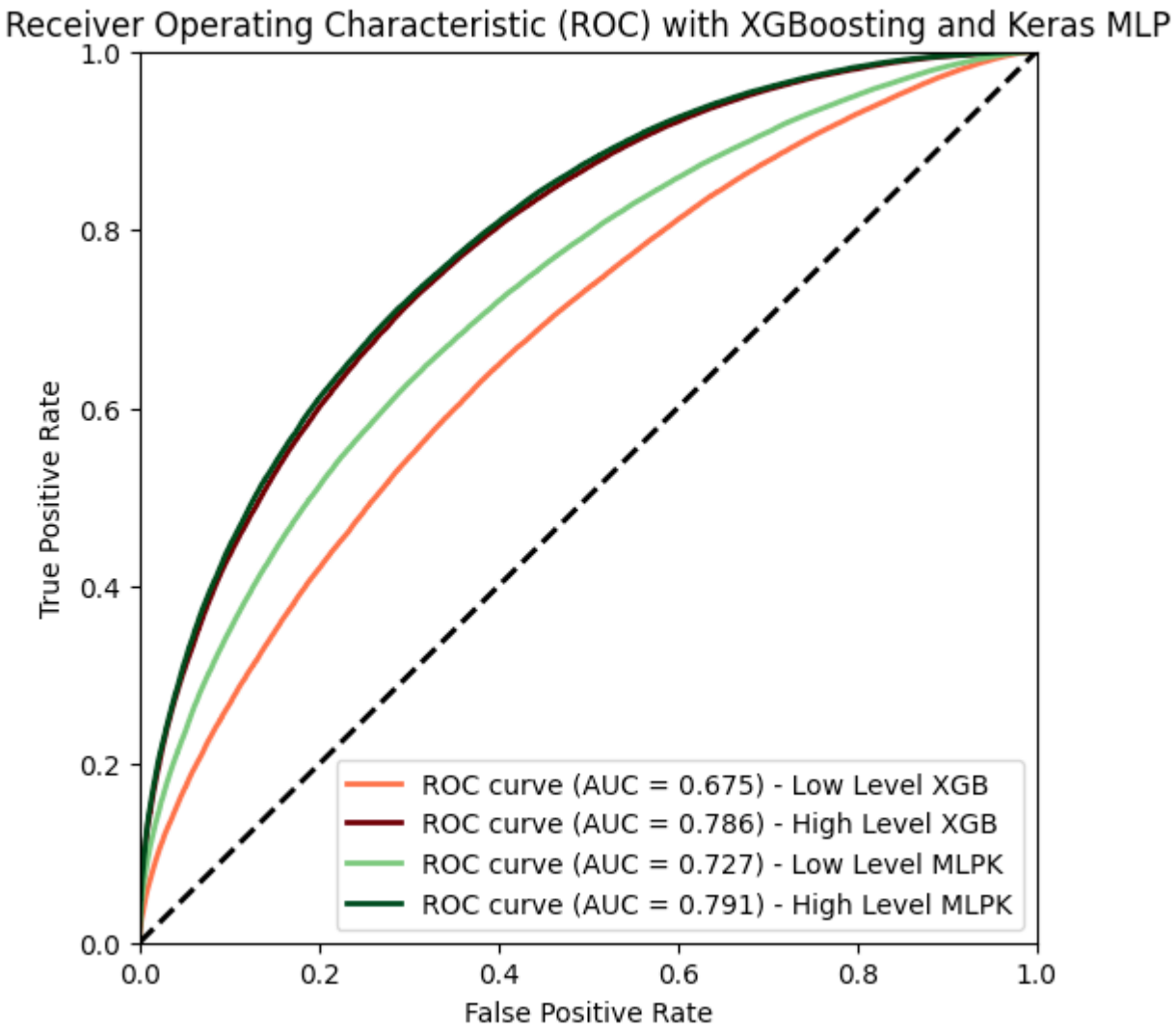
4. **Implications for Further Analysis**: The persistent central mixing region, extending from the bottom left to the bottom right, indicates that there is still some overlap between the features of the two labels; analyzing the data points in this area could provide insights into the nature of this overlap. The decrease in outliers suggests that the model's representation of the data has improved; further refinement could focus on enhancing feature extraction to further reduce the central overlap and increase cluster distinctiveness.

In [96]:
```
MLPKerasTrainValidationLossPlot(history_MLPK_hl, 'High-Level features')
```



The validation loss curve closely follows the train loss curve. The process is halted at epoch 18 because the values start to diverge.

In [98]:
```
ROC_AUC(y_test, [[y_pred_proba_XGB_ll, y_pred_proba_XGB_hl], \
                 [y_pred_proba_MLPK_ll, y_pred_proba_MLPK_hl],], 'XGBoosting and Keras MLP')
```



We can interpret the AUC values for the model MLP with Keras as follows:

1. **Low-Level Multilayer Perceptrons (MLP) with Keras AUC = 0.727:**

This value falls between 0.7 and 0.8, which is considered acceptable. It indicates that the model performs adequately in distinguishing between classes. The Low-Level features model shows a better capability to differentiate between positive and negative classes

compared to random guessing, reflecting a noticeable improvement in performance over the Low-Level XGB model.

2. **High-Level Multilayer Perceptrons (MLP) with Keras AUC = 0.791:**

This value also falls between 0.7 and 0.8, which is considered acceptable. It indicates that the model performs well in distinguishing between classes. The High-Level features model shows a slightly better capability to differentiate between classes compared to the High-Level XGB model, indicating a good capability to distinguish between Background and Signal classes.

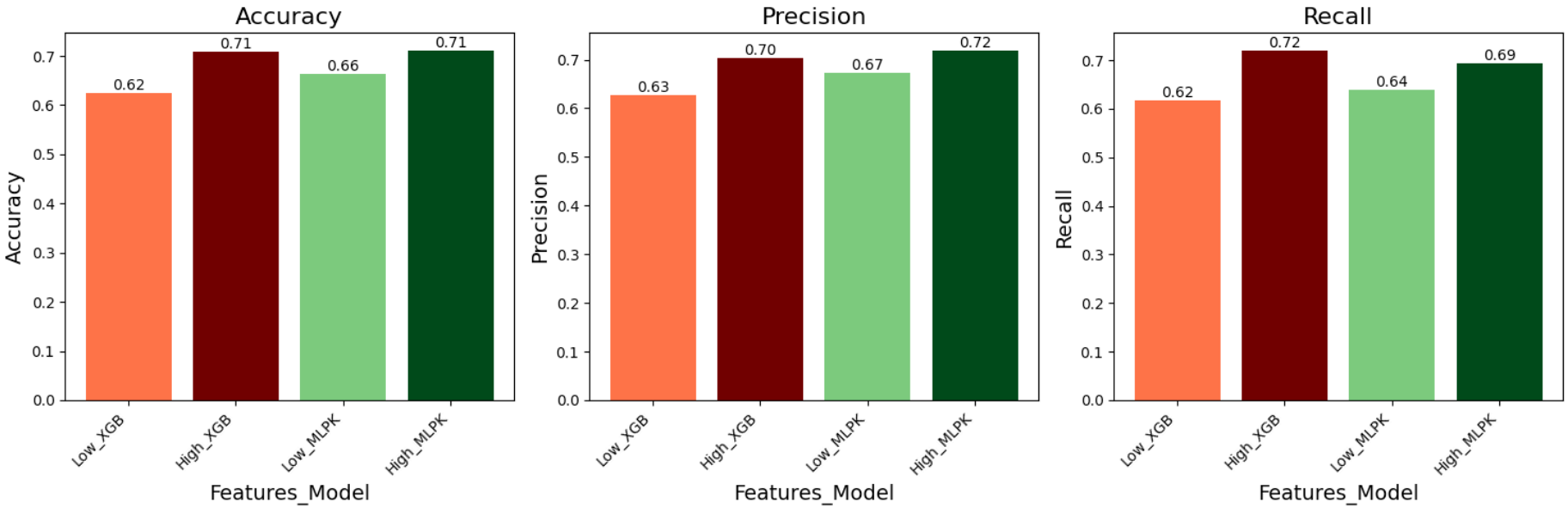
Based on the AUC values and their interpretation, the conclusion for XGB and MLPK models' performance is:

1. **Low-Level Features:** The Low-Level MLP model has an AUC of 0.727, which falls into the acceptable range. This indicates that the MLP with Keras has a reasonable capability to differentiate between classes, showing better performance than the Low-Level XGB model with an AUC of 0.675. It suggests that the MLP model is better at discovering patterns in the Low-Level features compared to the XGB model, though there is still room for improvement, eventually with a larger dataset.

2. **High-Level Features:** The High-Level MLP model has an AUC of 0.791, which is in the acceptable range and slightly better than the High-Level XGB model's AUC of 0.786. This indicates that the MLP with Keras performs well with High-Level features, showing a good capability to distinguish between Background and Signal classes. The performance is comparable to the XGB model, suggesting that both models benefit significantly from High-Level features and can effectively leverage them to improve predictive performance.

In [100…

```
# Dictionary to store the results
report = {'model': [['Low_XGB', 'High_XGB'],['Low_MLPK', 'High_MLPK']],
          'Accuracy': [[acc_XGB_ll, acc_XGB_hl],[acc_MLPK_ll, acc_MLPK_hl]],
          'Precision': [[prec_XGB_ll, prec_XGB_hl],[prec_MLPK_ll, prec_MLPK_hl]],
          'Recall': [[rec_XGB_ll, rec_XGB_hl],[rec_MLPK_ll, rec_MLPK_hl]]
         }

ResultsPlot(report)
```



5.2.2. Multilayer Perceptrons with Scikit-learn

Scikit-learn, a widely-used machine learning library, does not have built-in functionalities specifically developed for loss monitoring in its MLPClassifier for Multilayer Perceptrons. This capability is typically found in Deep Learning frameworks like TensorFlow or PyTorch, which offer more extensive tools for neural network training and optimization. So, in the following MLPScikit function, the manual implementation of early stopping with loss monitoring serves several critical purposes to ensure effective model training and performance evaluation:

1. **Early Stopping**: Early stopping is crucial for preventing overfitting. It terminates the training process when the validation accuracy stops improving, thereby preventing the model from learning noise or specific details of the training data that do not generalize well. In the function, early stopping is manually implemented by monitoring the validation accuracy (val_accuracy). If there is no improvement in validation accuracy for a specified number of epochs (patience=5), training is halted early to avoid overfitting.

2. **Train/Validation Loss Monitoring**: Monitoring these losses provides insights into how well the model is learning and generalizing from the training data to unseen validation data. The Training Loss is calculated using the logarithmic loss (log_loss) between the predicted probabilities (y_train_pred_proba) and actual training labels (y_train); this metric helps assess how well the model fits the training data. Similarly, the Validation Loss is calculated using log_loss between predicted probabilities (y_val_pred_proba) and actual validation labels (y_val). It indicates how well the model generalizes to unseen validation data.

In the following function, the following hyperparameters are finely tuned:

1. **Activation**: Specifies the activation function used in the MLP (e.g., 'relu', 'sigmoid'). Activation functions introduce non-linearity to the model, enabling it to capture complex patterns in the data effectively.

2. **Solver**: Determines the optimization algorithm for weight optimization ('adam', 'lbfgs', 'sgd'). Each solver has its advantages depending on the dataset size, complexity, and available computational resources.

3. **Epochs**: Refers to the number of complete passes through the training dataset. Each epoch involves forward and backward propagation, updating model parameters to minimize training loss and enhance performance iteratively.

Also, **tracking time** (start_time1, start_time2, end_time1, end_time2, run_time1, run_time2) ensures transparency regarding the computational cost of training.

In [109…

```
def MLPScikit(X_train, y_train, X_val, y_val, X_test, y_test, COLS, \
              activation, solver, epochs):
    """
    Train and predict with Scikit-learn MLP Model.

    Parameters:
      X_train (pd.DataFrame): Training features.
      y_train (Union[pd.Series, np.ndarray]): Training labels.
      X_val (pd.DataFrame): Validation features.
      y_val (Union[pd.Series, np.ndarray]): Validation labels.
      X_test (pd.DataFrame): Test features.
      y_test (Union[pd.Series, np.ndarray]): Test labels.
      COLS (List[str]): Set of features/columns with which the model is trained and predicts.
      activation (str): Activation function.
      solver (str): Solver algorithm.
      epochs (int): Number of epochs for training.

    Returns:
      Tuple: A tuple containing the following elements:
          - statistics (Tuple[float, float, float]): Evaluation metrics for the test set (accuracy, precision, reca
          - history (List[List[float]]): Training history of the model (training loss, validation loss).
          - y_pred_proba (np.ndarray): Predicted probabilities for the test set.
          - best_mlps_model (MLPClassifier): Best trained Scikit Multilayer Perceptron model.
    """
```

```
# Import necessary libraries
import time
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, log_loss
import numpy as np

# Print start time
print("Start time:", time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))

# Record start time
start_time1 = time.time()

# Initialize the MLPClassifier
mlps_model = MLPClassifier(
    hidden_layer_sizes=(128, 256, 512, 1024, 512, 1),
    alpha=0.001, # L2 regularization
    activation=activation,
    solver=solver,
    random_state=42,
    early_stopping=False  # Turn off built-in early stopping
)

# Parameters of the model
print("Parameters of MLP Scikit model in use:")
params = mlps_model.get_params()
for key, value in params.items():
    print(f"{key:<20} {value}")
print()

# Initialize variables for early stopping
best_val_accuracy = 0
patience = 5
best_mlps_model = None
best_epoch = None
epochs_without_improvement = 0
training_loss = []
validation_loss = []

# Train the model
for epoch in range(epochs):
    # Record start time
```

```
start_time2 = time.time()
print(f"Epoch {epoch+1:03}:", end=" ")

# Fit the model to the training data
print(f"Fitting...", end=" ")
mlps_model.partial_fit(X_train[COLS], y_train, classes=np.unique(y_train))

# Track training loss
print(f"Train_loss:", end=" ")
y_train_pred_proba = mlps_model.predict_proba(X_train[COLS])
train_loss = log_loss(y_train, y_train_pred_proba)
print(f"{train_loss:.4f}", end=" ")
training_loss.append(train_loss)

# Calculate validation accuracy
print(f"Val_accuracy:", end=" ")
y_val_pred = mlps_model.predict(X_val[COLS])
val_accuracy = accuracy_score(y_val, y_val_pred)
accuracy_improvement = val_accuracy - best_val_accuracy
print(f"{val_accuracy:.4f} Acc_improv.: {accuracy_improvement:+.5f}", end=" ")

# Calculate validation loss
print(f"Val_loss:", end=" ")
y_val_pred_proba = mlps_model.predict_proba(X_val[COLS])
val_loss = log_loss(y_val, y_val_pred_proba)
print(f"{val_loss:.4f}", end=" ")
validation_loss.append(val_loss)

# Check for epochs without improvement
if val_accuracy > best_val_accuracy:
    best_val_accuracy = val_accuracy
    best_mlps_model = mlps_model
    best_epoch = epoch
    epochs_without_improvement = 0
else:
    epochs_without_improvement += 1

# Print epochs without improvement
print(f"Epochs w/o improv.: {epochs_without_improvement}.", end=" ")

# Record end time
```

```
        end_time2 = time.time()

        # Calculate and print time taken
        run_time2 = end_time2 - start_time2
        print(f"{run_time2/60:.2f} min.")

        # Check for break in early stopping
        if epochs_without_improvement >= patience:
            print(f"Early stopping at epoch {epoch+1}.")
            break

    # Create history
    history = [training_loss, validation_loss]

    # Use the best model for predictions
    print(f"\nPredicting with the model of epoch {best_epoch+1}...")
    y_pred = best_mlps_model.predict(X_test[COLS])

    # Predict on the test data for ROC/AUC
    y_pred_proba = best_mlps_model.predict_proba(X_test[COLS])[:, 1]

    # Record end time
    end_time1 = time.time()

    # Calculate and print time taken
    run_time1 = end_time1 - start_time1
    print(f"Time taken for running Scikit-learn MLP Model: {run_time1/60:.2f} minutes.")

    # Return metrics and predicted labels
    return Statistics(y_test, y_pred), history, y_pred_proba, best_mlps_model
```

The following function provides a visual UMAP representation of how well the Scikit-learn MLP model separates classes in a lower-dimensional space, based on its learned representations from the second-to-last dense layer outputs. We can adjust sample_fraction to control the proportion of data used for embedding, depending on the computational resources available and the desired level of detail in the visualization.

In the following function, tracking time (start_time, end_time, run_time) ensures transparency regarding the computational cost of training.

In [104…

```
def MLPScikitUMAP(mlps_model, TrainSet, y_train, columns, sample_fraction=0.1):
    """
    Visualizes the UMAP projection of the second-to-last dense layer outputs
    from a given Scikit-learn MLP model, using a sample of the training data.

    Parameters:
        mlps_model (sklearn.neural_network.MLPClassifier): Trained Scikit-learn MLP model.
        TrainSet (numpy.ndarray): Training data used to obtain the second-to-last layer outputs.
        y_train (numpy.ndarray): Labels corresponding to the training data.
        sample_fraction (float): Fraction of the data to use for embedding (default is 0.1, i.e., 10%).
        columns (str): String to describe the columns being analyzed.

    Returns:
        None
    """
    # Import necessary libraries
    import time
    import numpy as np
    import umap
    import matplotlib.pyplot as plt
    from matplotlib.lines import Line2D

    # Print start time
    print("Start time:", time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))

    # Record start time
    start_time = time.time()

    # Define the function to extract the intermediate output
    def get_intermediate_output(model, data):
        # Perform a forward pass up to the second-to-last layer
        layer_activations = data
        for i in range(len(model.coefs_) - 1):  # Exclude the last layer
            layer_activations = np.dot(layer_activations, model.coefs_[i]) + model.intercepts_[i]
            if model.activation == 'relu':
                layer_activations = np.maximum(0, layer_activations)
            elif model.activation == 'logistic':
                layer_activations = 1 / (1 + np.exp(-layer_activations))
            elif model.activation == 'tanh':
                layer_activations = np.tanh(layer_activations)
```

```
        elif model.activation == 'identity':
            pass
    return layer_activations

# Get the intermediate layer outputs for the training data
print("Predicting on the Second-to-last Layer...")
intermediate_output = get_intermediate_output(mlps_model, TrainSet)

# Reset the index of y_train to make it compatible with the random indices
y_train_reset = y_train.reset_index(drop=True)

# Sample the data
np.random.seed(42)  # For reproducibility
sample_size = int(len(intermediate_output) * sample_fraction)
sample_indices = np.random.choice(len(intermediate_output), sample_size, replace=False)
sample_output = intermediate_output[sample_indices]
sample_labels = y_train_reset[sample_indices]

# Use UMAP to reduce dimensionality
reducer = umap.UMAP(n_neighbors=15, metric='euclidean', n_epochs=200, init='spectral', n_jobs=-1, verbose=True)
print("Embedding...")
embedding = reducer.fit_transform(sample_output)

# Plot the UMAP embedding
print("Plotting...")
plt.figure(figsize=(8, 6))
cmap = plt.cm.viridis
plt.scatter(embedding[:, 0], embedding[:, 1], c=sample_labels, cmap=cmap, s=5, alpha=0.5)

# Create custom legend
legend_elements = [Line2D([0], [0], marker='o', color='w', markerfacecolor=cmap(0), markersize=10, label='Backg
                   Line2D([0], [0], marker='o', color='w', markerfacecolor=cmap(cmap.N), markersize=10, label='
                  ]
plt.legend(handles=legend_elements, title='Classes')

# Add title, x-axis label, y-axis label, and display the plot
plt.title(f"UMAP projection of the second-to-last layer outputs for {columns} with Scikit-learn MLP")
plt.xlabel('UMAP Dimension 1')
plt.ylabel('UMAP Dimension 2')
plt.show()
```

```
    # Record end time
    end_time = time.time()

    # Calculate and print time taken
    run_time = end_time - start_time
    print(f"Time taken for Embedding and Plotting: {run_time/60:.2f} minutes.")
```

The following function is essential for visualizing the Training and Validation Loss trends over epochs, in the Scikit-learn MLP model, aiding in assessing the model's training progress and identifying potential issues such as overfitting or underfitting.

In [106…

```
def MLPScikitTrainValidationLossPlot(history, columns):
    """
    Plot training and validation loss for a Scikit-learn MLP model.

    Parameters:
        history (List[List[float]]): List containing training and validation loss (history from MLP function).
        columns (str): String to describe the columns being analyzed.

    Returns:
        None.
    """
    # Import libraries
    import matplotlib.pyplot as plt

    # Plot the training and validation loss
    plt.figure(figsize=(8, 3))
    plt.plot(history[0], label='Training Loss')
    plt.plot(history[1], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(f"Training and Validation Loss for {columns} with Scikit-learn MLP")
    plt.legend()
    plt.show()
```

5.2.2.1. Low Level features

We start by training and predicting with the Low-Level features using Scikit-learn Multilayer Perceptrons model.

In [111…

```
# Multilayer Perceptrons with Scikit-learn with Low Level features
```

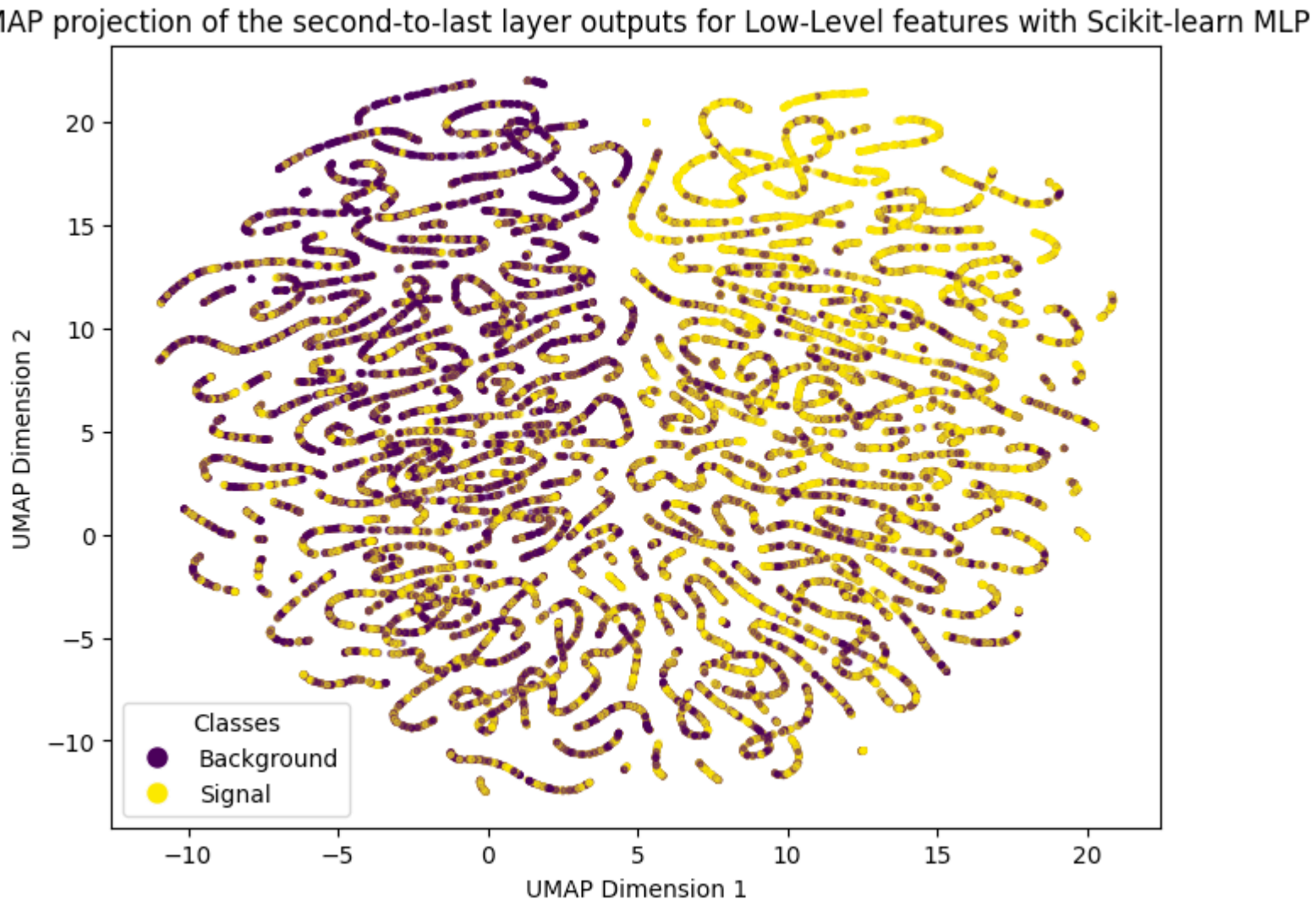
```
(acc_MLPS_ll, prec_MLPS_ll, rec_MLPS_ll), history_MLPS_ll, y_pred_proba_MLPS_ll, clf_MLPS_ll = MLPScikit(
    X_train, y_train, X_val, y_val, X_test, y_test, LowLevelCols, \
    'tanh', 'sgd', 25
)
```

```
Start time: 2024-07-04 09:21:31
Parameters of MLP Scikit model in use:
activation           tanh
alpha                0.001
batch_size           auto
beta_1               0.9
beta_2               0.999
early_stopping       False
epsilon              1e-08
hidden_layer_sizes   (128, 256, 512, 1024, 512, 1)
learning_rate        constant
learning_rate_init   0.001
max_fun              15000
max_iter             200
momentum             0.9
n_iter_no_change     10
nesterovs_momentum   True
power_t              0.5
random_state         42
shuffle              True
solver               sgd
tol                  0.0001
validation_fraction  0.1
verbose              False
warm_start           False

Epoch 001: Fitting... Train_loss: 0.6779 Val_accuracy: 0.5675 Acc_improv.: +0.56749 Val_loss: 0.6788 Epochs w/o impr
ov.: 0. 4.57 min.
Epoch 002: Fitting... Train_loss: 0.6729 Val_accuracy: 0.5747 Acc_improv.: +0.00721 Val_loss: 0.6737 Epochs w/o impr
ov.: 0. 4.62 min.
Epoch 003: Fitting... Train_loss: 0.6671 Val_accuracy: 0.5864 Acc_improv.: +0.01168 Val_loss: 0.6683 Epochs w/o impr
ov.: 0. 4.51 min.
Epoch 004: Fitting... Train_loss: 0.6622 Val_accuracy: 0.5959 Acc_improv.: +0.00948 Val_loss: 0.6636 Epochs w/o impr
ov.: 0. 4.33 min.
Epoch 005: Fitting... Train_loss: 0.6587 Val_accuracy: 0.6026 Acc_improv.: +0.00671 Val_loss: 0.6601 Epochs w/o impr
ov.: 0. 4.32 min.
Epoch 006: Fitting... Train_loss: 0.6558 Val_accuracy: 0.6071 Acc_improv.: +0.00451 Val_loss: 0.6574 Epochs w/o impr
ov.: 0. 4.37 min.
Epoch 007: Fitting... Train_loss: 0.6536 Val_accuracy: 0.6105 Acc_improv.: +0.00344 Val_loss: 0.6552 Epochs w/o impr
ov.: 0. 7.58 min.
Epoch 008: Fitting... Train_loss: 0.6518 Val_accuracy: 0.6125 Acc_improv.: +0.00200 Val_loss: 0.6535 Epochs w/o impr
```

```
ov.: 0. 5.42 min.
Epoch 009: Fitting... Train_loss: 0.6502 Val_accuracy: 0.6136 Acc_improv.: +0.00113 Val_loss: 0.6520 Epochs w/o impr
ov.: 0. 4.36 min.
Epoch 010: Fitting... Train_loss: 0.6486 Val_accuracy: 0.6158 Acc_improv.: +0.00215 Val_loss: 0.6505 Epochs w/o impr
ov.: 0. 4.60 min.
Epoch 011: Fitting... Train_loss: 0.6470 Val_accuracy: 0.6180 Acc_improv.: +0.00223 Val_loss: 0.6489 Epochs w/o impr
ov.: 0. 4.31 min.
Epoch 012: Fitting... Train_loss: 0.6453 Val_accuracy: 0.6206 Acc_improv.: +0.00261 Val_loss: 0.6473 Epochs w/o impr
ov.: 0. 7.60 min.
Epoch 013: Fitting... Train_loss: 0.6436 Val_accuracy: 0.6221 Acc_improv.: +0.00148 Val_loss: 0.6457 Epochs w/o impr
ov.: 0. 7.76 min.
Epoch 014: Fitting... Train_loss: 0.6418 Val_accuracy: 0.6245 Acc_improv.: +0.00233 Val_loss: 0.6440 Epochs w/o impr
ov.: 0. 9.10 min.
Epoch 015: Fitting... Train_loss: 0.6398 Val_accuracy: 0.6271 Acc_improv.: +0.00267 Val_loss: 0.6422 Epochs w/o impr
ov.: 0. 9.37 min.
Epoch 016: Fitting... Train_loss: 0.6382 Val_accuracy: 0.6283 Acc_improv.: +0.00111 Val_loss: 0.6407 Epochs w/o impr
ov.: 0. 10.28 min.
Epoch 017: Fitting... Train_loss: 0.6369 Val_accuracy: 0.6300 Acc_improv.: +0.00173 Val_loss: 0.6395 Epochs w/o impr
ov.: 0. 6.73 min.
Epoch 018: Fitting... Train_loss: 0.6358 Val_accuracy: 0.6312 Acc_improv.: +0.00123 Val_loss: 0.6385 Epochs w/o impr
ov.: 0. 5.43 min.
Epoch 019: Fitting... Train_loss: 0.6347 Val_accuracy: 0.6326 Acc_improv.: +0.00141 Val_loss: 0.6375 Epochs w/o impr
ov.: 0. 9.97 min.
Epoch 020: Fitting... Train_loss: 0.6336 Val_accuracy: 0.6340 Acc_improv.: +0.00135 Val_loss: 0.6365 Epochs w/o impr
ov.: 0. 9.97 min.
Epoch 021: Fitting... Train_loss: 0.6326 Val_accuracy: 0.6354 Acc_improv.: +0.00146 Val_loss: 0.6356 Epochs w/o impr
ov.: 0. 7.86 min.
Epoch 022: Fitting... Train_loss: 0.6316 Val_accuracy: 0.6362 Acc_improv.: +0.00077 Val_loss: 0.6347 Epochs w/o impr
ov.: 0. 8.38 min.
Epoch 023: Fitting... Train_loss: 0.6306 Val_accuracy: 0.6370 Acc_improv.: +0.00075 Val_loss: 0.6338 Epochs w/o impr
ov.: 0. 7.42 min.
Epoch 024: Fitting... Train_loss: 0.6296 Val_accuracy: 0.6379 Acc_improv.: +0.00096 Val_loss: 0.6330 Epochs w/o impr
ov.: 0. 6.39 min.
Epoch 025: Fitting... Train_loss: 0.6286 Val_accuracy: 0.6390 Acc_improv.: +0.00111 Val_loss: 0.6322 Epochs w/o impr
ov.: 0. 8.22 min.

Predicting with the model of epoch 25...
Time taken for running Scikit-learn MLP Model: 169.42 minutes.
Accuracy: 0.637
Precision: 0.630
Recall: 0.663
```

In [112…
```
MLPScikitUMAP(clf_MLPS_ll, X_train[LowLevelCols], y_train, 'Low-Level features')
```

```
Start time: 2024-07-04 12:10:58
Predicting on the Second-to-last Layer...
Embedding...
UMAP(n_epochs=200, verbose=True)
Thu Jul  4 12:17:45 2024 Construct fuzzy simplicial set
Thu Jul  4 12:17:45 2024 Finding Nearest Neighbors
Thu Jul  4 12:17:45 2024 Building RP forest with 16 trees
Thu Jul  4 12:17:46 2024 NN descent for 15 iterations
         1  /  15
         2  /  15
        Stopping threshold met -- exiting after 2 iterations
Thu Jul  4 12:17:48 2024 Finished Nearest Neighbor Search
Thu Jul  4 12:17:49 2024 Construct embedding
Epochs completed:   0%|            0/200 [00:00]
        completed  0  /  200 epochs
        completed  20  /  200 epochs
        completed  40  /  200 epochs
        completed  60  /  200 epochs
        completed  80  /  200 epochs
        completed  100  /  200 epochs
        completed  120  /  200 epochs
        completed  140  /  200 epochs
        completed  160  /  200 epochs
        completed  180  /  200 epochs
Thu Jul  4 13:10:36 2024 Finished embedding
Plotting...
```




```
Time taken for Embedding and Plotting: 59.66 minutes.
```

We examine this UMAP visualization derived from an MLP model implemented using the scikit-learn library, with a sample size of 10% of the total dataset with Low-Level features (by saving resources, taking into account the requirement of this model):

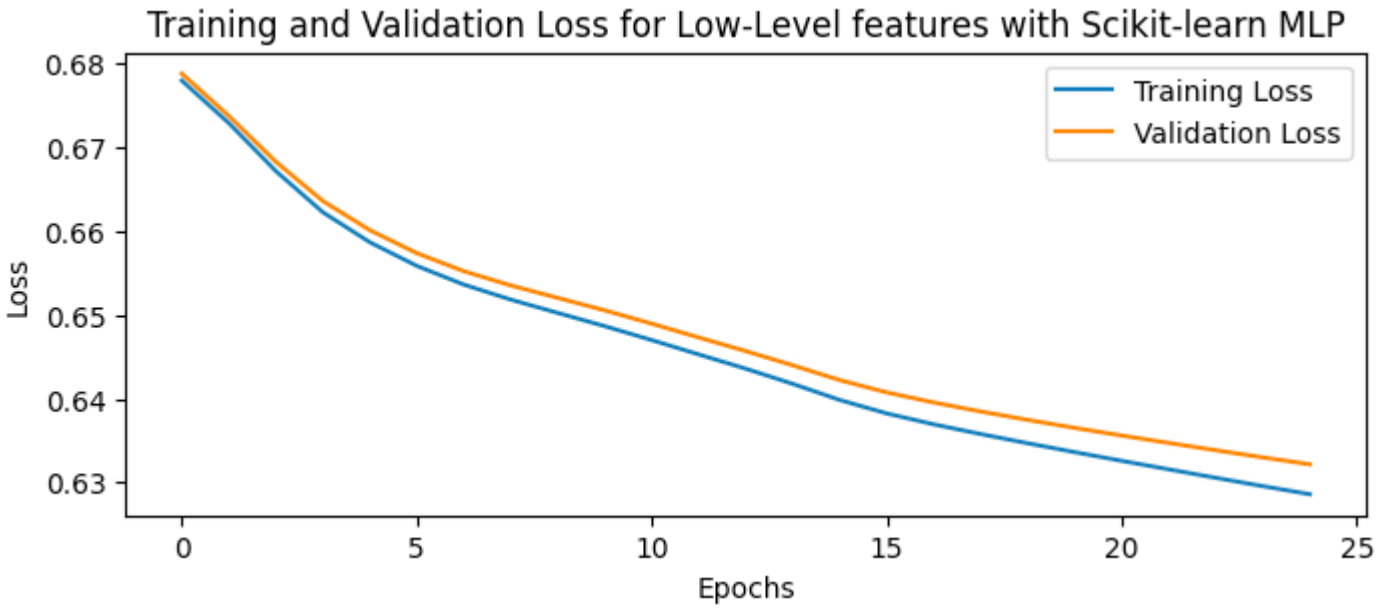
1. **Little clusters on top ("Background" on left, "Signal" on right)**: The UMAP visualization reveals small clusters located at the top of the plot. These clusters represent groups of similar data points, with a distinct distribution: Background data points on the left

and Signal data points on the right. This separation suggests a potential pattern or relationship within the data.

2. **Majority mix of both**: The overall UMAP plot shows a blend of both Background and Signal data points across most of the visualization. This intermingling indicates that the data is not completely separable into distinct groups, but rather a mix of both types of data points is present, suggesting overlap or integration.

3. **Comparison with Keras**: It is noted that these clusters are slightly smaller compared to those observed when using a Multilayer Perceptron with Keras, indicating a bit worse performance than the Keras implementation.

In [113…
```
MLPScikitTrainValidationLossPlot(history_MLPS_ll, 'Low-Level features')
```



The validation loss curve closely follows the train loss curve. The process is halted at epoch 25 because the values start to diverge.

5.2.2.2. High Level features

We finally train and predict using Scikit-learn Multilayer Perceptrons model with the High-Level features.

In [123…
```
# Multilayer Perceptrons with Scikit-learn with High Level features
```

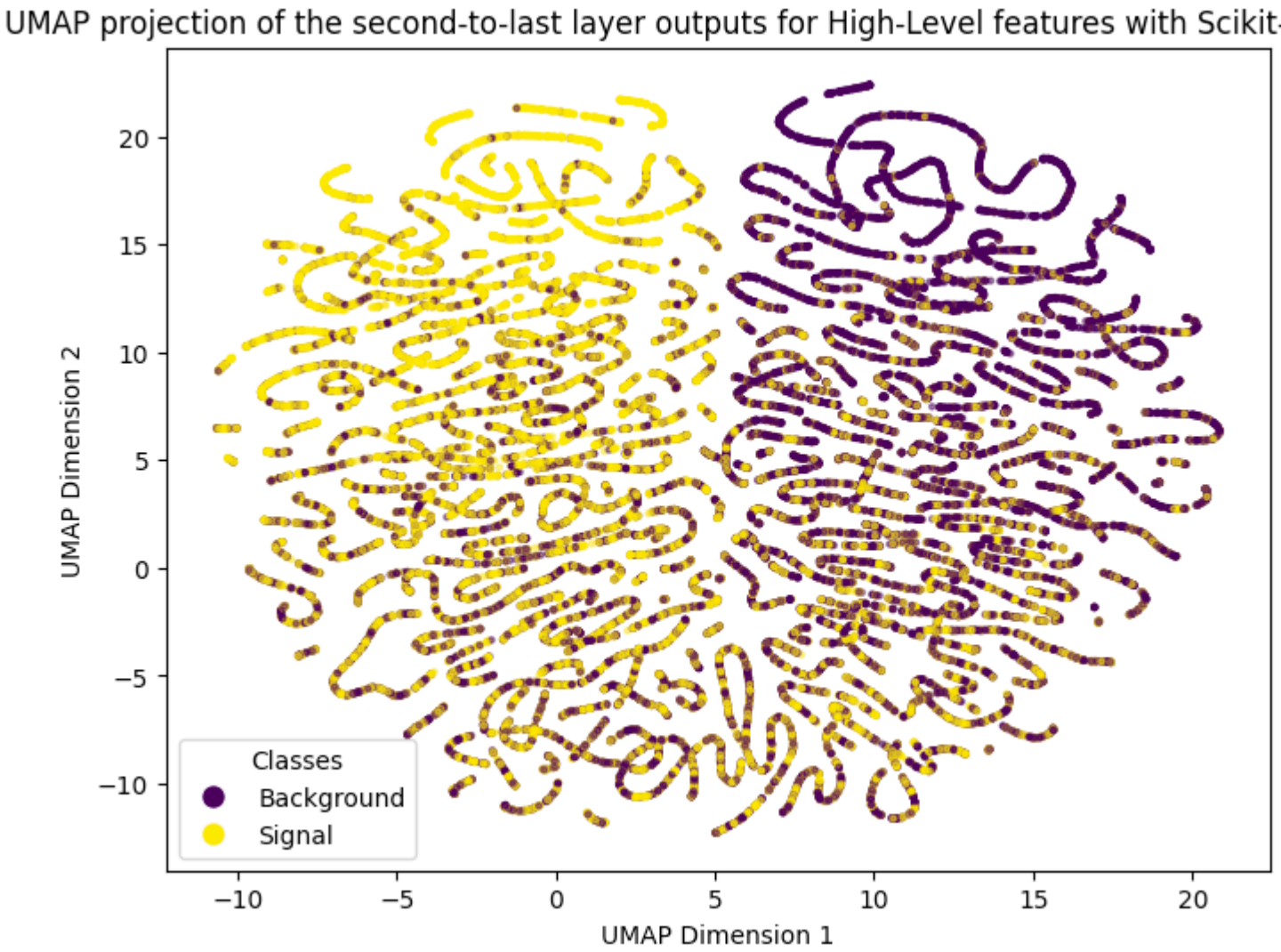
```
(acc_MLPS_hl, prec_MLPS_hl, rec_MLPS_hl), history_MLPS_hl, y_pred_proba_MLPS_hl, clf_MLPS_hl = MLPScikit(
    X_train, y_train, X_val, y_val, X_test, y_test, HighLevelCols, \
    'tanh', 'sgd', 25
)
```

```
Start time: 2024-07-04 13:31:24
Parameters of MLP Scikit model in use:
activation           tanh
alpha                0.001
batch_size           auto
beta_1               0.9
beta_2               0.999
early_stopping       False
epsilon              1e-08
hidden_layer_sizes   (128, 256, 512, 1024, 512, 1)
learning_rate        constant
learning_rate_init   0.001
max_fun              15000
max_iter             200
momentum             0.9
n_iter_no_change     10
nesterovs_momentum   True
power_t              0.5
random_state         42
shuffle              True
solver               sgd
tol                  0.0001
validation_fraction  0.1
verbose              False
warm_start           False

Epoch 001: Fitting... Train_loss: 0.6492 Val_accuracy: 0.6308 Acc_improv.: +0.63078 Val_loss: 0.6497 Epochs w/o impr
ov.: 0. 8.50 min.
Epoch 002: Fitting... Train_loss: 0.6414 Val_accuracy: 0.6375 Acc_improv.: +0.00668 Val_loss: 0.6423 Epochs w/o impr
ov.: 0. 10.35 min.
Epoch 003: Fitting... Train_loss: 0.6284 Val_accuracy: 0.6441 Acc_improv.: +0.00667 Val_loss: 0.6300 Epochs w/o impr
ov.: 0. 7.74 min.
Epoch 004: Fitting... Train_loss: 0.6128 Val_accuracy: 0.6573 Acc_improv.: +0.01317 Val_loss: 0.6148 Epochs w/o impr
ov.: 0. 7.45 min.
Epoch 005: Fitting... Train_loss: 0.6014 Val_accuracy: 0.6682 Acc_improv.: +0.01094 Val_loss: 0.6035 Epochs w/o impr
ov.: 0. 8.14 min.
Epoch 006: Fitting... Train_loss: 0.5925 Val_accuracy: 0.6758 Acc_improv.: +0.00756 Val_loss: 0.5951 Epochs w/o impr
ov.: 0. 7.13 min.
Epoch 007: Fitting... Train_loss: 0.5859 Val_accuracy: 0.6810 Acc_improv.: +0.00526 Val_loss: 0.5890 Epochs w/o impr
ov.: 0. 6.25 min.
Epoch 008: Fitting... Train_loss: 0.5810 Val_accuracy: 0.6849 Acc_improv.: +0.00385 Val_loss: 0.5843 Epochs w/o impr
```

```
ov.: 0. 5.92 min.
Epoch 009: Fitting... Train_loss: 0.5773 Val_accuracy: 0.6878 Acc_improv.: +0.00288 Val_loss: 0.5808 Epochs w/o impr
ov.: 0. 8.02 min.
Epoch 010: Fitting... Train_loss: 0.5745 Val_accuracy: 0.6899 Acc_improv.: +0.00210 Val_loss: 0.5781 Epochs w/o impr
ov.: 0. 7.12 min.
Epoch 011: Fitting... Train_loss: 0.5724 Val_accuracy: 0.6915 Acc_improv.: +0.00158 Val_loss: 0.5760 Epochs w/o impr
ov.: 0. 9.60 min.
Epoch 012: Fitting... Train_loss: 0.5706 Val_accuracy: 0.6927 Acc_improv.: +0.00120 Val_loss: 0.5742 Epochs w/o impr
ov.: 0. 6.79 min.
Epoch 013: Fitting... Train_loss: 0.5690 Val_accuracy: 0.6937 Acc_improv.: +0.00100 Val_loss: 0.5727 Epochs w/o impr
ov.: 0. 7.58 min.
Epoch 014: Fitting... Train_loss: 0.5675 Val_accuracy: 0.6951 Acc_improv.: +0.00148 Val_loss: 0.5713 Epochs w/o impr
ov.: 0. 9.19 min.
Epoch 015: Fitting... Train_loss: 0.5662 Val_accuracy: 0.6959 Acc_improv.: +0.00077 Val_loss: 0.5700 Epochs w/o impr
ov.: 0. 8.28 min.
Epoch 016: Fitting... Train_loss: 0.5650 Val_accuracy: 0.6962 Acc_improv.: +0.00027 Val_loss: 0.5688 Epochs w/o impr
ov.: 0. 8.61 min.
Epoch 017: Fitting... Train_loss: 0.5640 Val_accuracy: 0.6968 Acc_improv.: +0.00057 Val_loss: 0.5678 Epochs w/o impr
ov.: 0. 9.11 min.
Epoch 018: Fitting... Train_loss: 0.5631 Val_accuracy: 0.6974 Acc_improv.: +0.00060 Val_loss: 0.5669 Epochs w/o impr
ov.: 0. 8.92 min.
Epoch 019: Fitting... Train_loss: 0.5623 Val_accuracy: 0.6977 Acc_improv.: +0.00036 Val_loss: 0.5662 Epochs w/o impr
ov.: 0. 8.16 min.
Epoch 020: Fitting... Train_loss: 0.5616 Val_accuracy: 0.6983 Acc_improv.: +0.00061 Val_loss: 0.5655 Epochs w/o impr
ov.: 0. 6.46 min.
Epoch 021: Fitting... Train_loss: 0.5609 Val_accuracy: 0.6986 Acc_improv.: +0.00032 Val_loss: 0.5649 Epochs w/o impr
ov.: 0. 10.04 min.
Epoch 022: Fitting... Train_loss: 0.5603 Val_accuracy: 0.6989 Acc_improv.: +0.00028 Val_loss: 0.5643 Epochs w/o impr
ov.: 0. 11.87 min.
Epoch 023: Fitting... Train_loss: 0.5597 Val_accuracy: 0.6994 Acc_improv.: +0.00051 Val_loss: 0.5638 Epochs w/o impr
ov.: 0. 8.68 min.
Epoch 024: Fitting... Train_loss: 0.5592 Val_accuracy: 0.6998 Acc_improv.: +0.00041 Val_loss: 0.5632 Epochs w/o impr
ov.: 0. 7.39 min.
Epoch 025: Fitting... Train_loss: 0.5586 Val_accuracy: 0.7001 Acc_improv.: +0.00027 Val_loss: 0.5627 Epochs w/o impr
ov.: 0. 14.05 min.

Predicting with the model of epoch 25...
Time taken for running Scikit-learn MLP Model: 213.35 minutes.
Accuracy: 0.704
Precision: 0.706
Recall: 0.702
```

In [124…
```
MLPScikitUMAP(clf_MLPS_hl, X_train[HighLevelCols], y_train, 'High-Level features')
```

```
Start time: 2024-07-04 17:04:47
Predicting on the Second-to-last Layer...
Embedding...
UMAP(n_epochs=200, verbose=True)
Thu Jul  4 17:13:28 2024 Construct fuzzy simplicial set
Thu Jul  4 17:13:28 2024 Finding Nearest Neighbors
Thu Jul  4 17:13:28 2024 Building RP forest with 16 trees
Thu Jul  4 17:13:29 2024 NN descent for 15 iterations
         1  /  15
         2  /  15
        Stopping threshold met -- exiting after 2 iterations
Thu Jul  4 17:13:35 2024 Finished Nearest Neighbor Search
Thu Jul  4 17:13:36 2024 Construct embedding
Epochs completed:   0%|            0/200 [00:00]
        completed  0  /  200 epochs
        completed  20  /  200 epochs
        completed  40  /  200 epochs
        completed  60  /  200 epochs
        completed  80  /  200 epochs
        completed  100  /  200 epochs
        completed  120  /  200 epochs
        completed  140  /  200 epochs
        completed  160  /  200 epochs
        completed  180  /  200 epochs
Thu Jul  4 18:22:16 2024 Finished embedding
Plotting...
```




```
Time taken for Embedding and Plotting: 77.53 minutes.
```

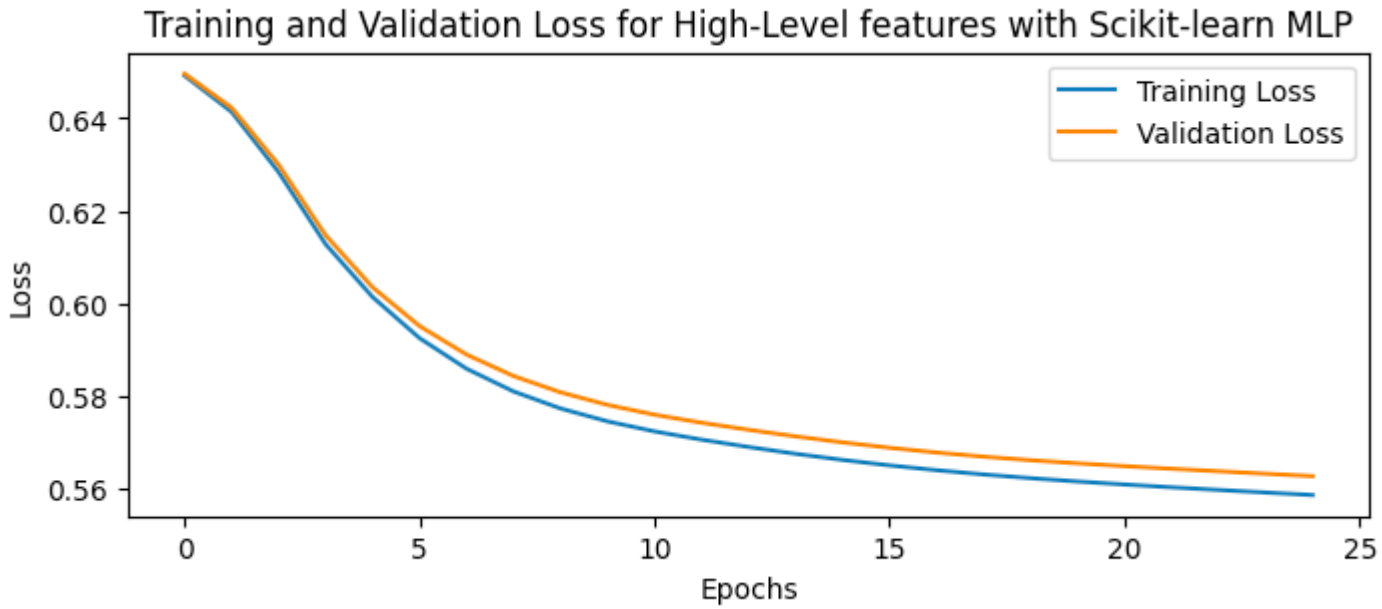
We now examine this UMAP visualization derived from an MLP model implemented using the scikit-learn library, with a sample size of 10% of the total dataset with High-Level features:

1. **Comparison with Low-Level features**: In comparison to Low-Level features, the UMAP plot with High-Level features shows slightly larger clusters at the top. This suggests that the use of High-Level features has resulted in a different clustering pattern,

possibly indicating a more distinct separation between Background and Signal data points. However, both feature types exhibit a majority mix of both Background and Signal data points throughout the visualization.

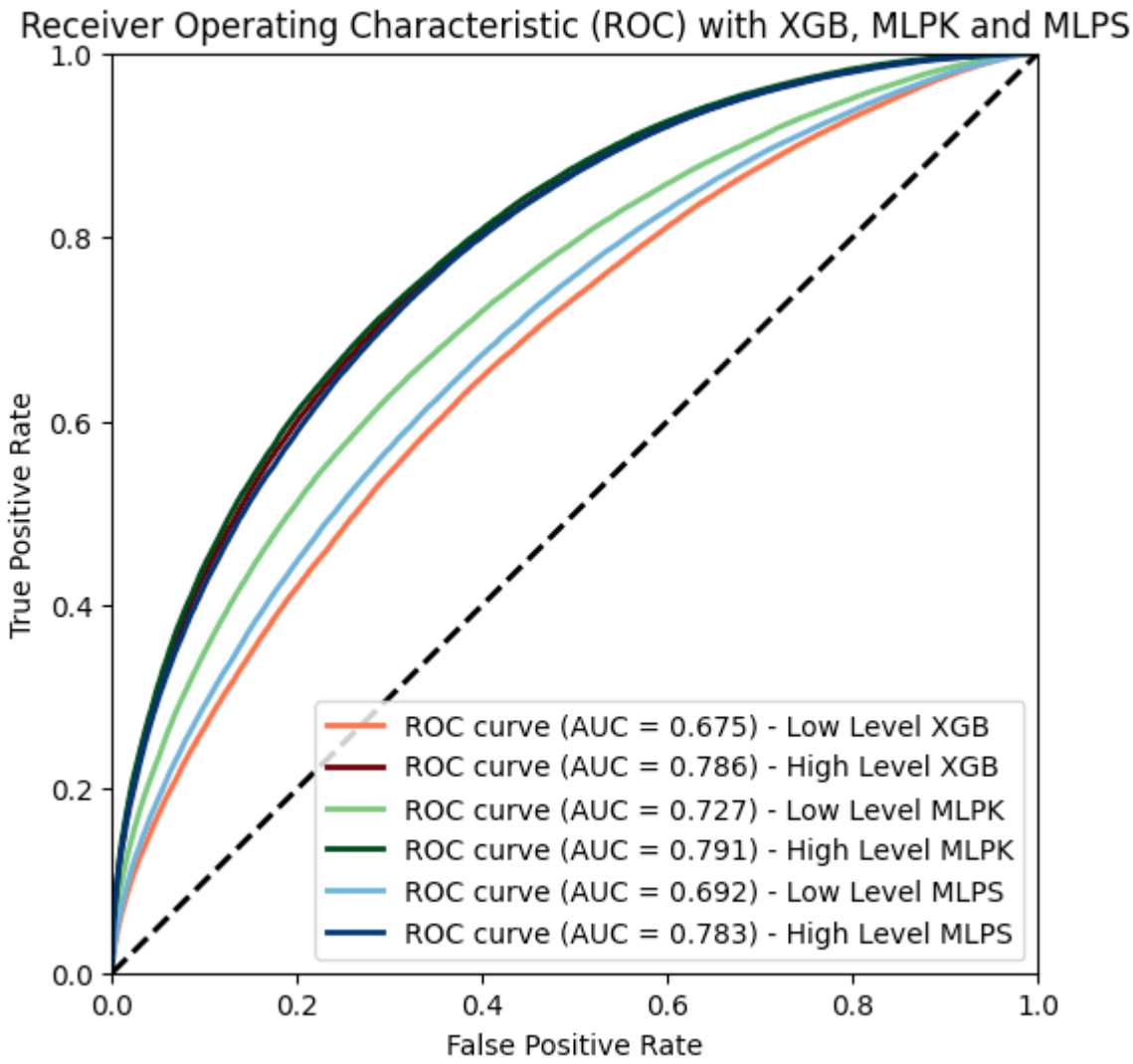
2. **Comparison with Keras**: When compared with a Multilayer Perceptron implemented using Keras, the clusters observed in the UMAP plot with High-Level features are slightly smaller. This indicates that the scikit-learn MLP with High-Level features may have lower performance in capturing the underlying structure of the data compared to the Keras implementation.

In [125…
```
MLPScikitTrainValidationLossPlot(history_MLPS_hl, 'High-Level features')
```



The validation loss curve closely tracks the training loss curve. The process is halted at epoch 25 because the improvements that are made are very minimal and do not justify the processing cost.

In [126…
```
ROC_AUC(y_test, [[y_pred_proba_XGB_ll, y_pred_proba_XGB_hl], \
                 [y_pred_proba_MLPK_ll, y_pred_proba_MLPK_hl], \
                 [y_pred_proba_MLPS_ll, y_pred_proba_MLPS_hl],], 'XGB, MLPK and MLPS')
```



Based on the previous ROC/AUC plot, the following observations can be made:

1. **Low-Level Features: AUC = 0.692**

The AUC value of 0.692 for the Scikit-learn MLP model with Low-Level features suggests moderate performance in distinguishing between classes. Falling within the acceptable range (typically 0.7 to 0.8), it indicates decent discriminatory power but also suggests

potential for improvement.

2. **High-Level Features: AUC = 0.783**

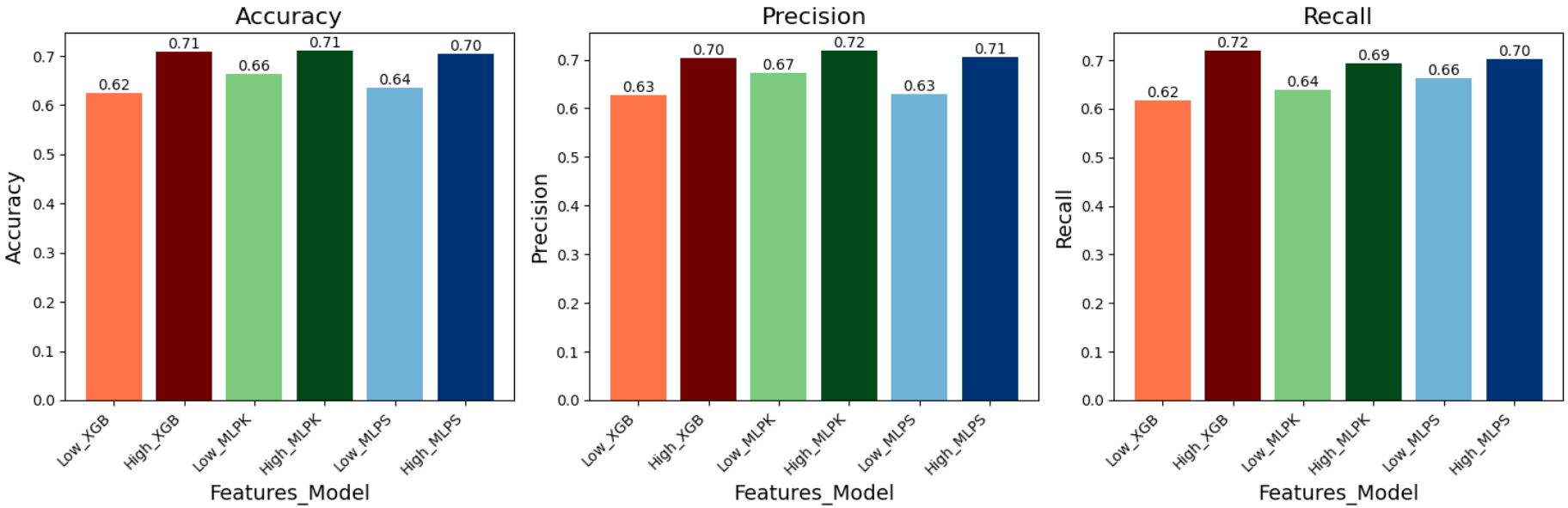
The AUC value of 0.783 for the Scikit-learn MLP model with High-Level features indicates better performance compared to the Low-Level model. This higher AUC value suggests that the model performs well in distinguishing between classes and demonstrates a stronger discriminatory ability.

The difference in performance between the MLP with Scikit-learn and the MLP with Keras, with Keras being better, could be attributed to several factors:

- **Model Complexity:** Keras allows for more flexibility and customization in model architecture and training parameters compared to Scikit-learn. This flexibility may enable Keras models to capture complex patterns in data more effectively.
- **Backend Implementation:** Keras utilizes TensorFlow or another backend framework optimized for neural network computations. This backend optimization can lead to improved performance compared to Scikit-learn's more generalized approach.
- **Feature Handling:** Keras models may handle features differently or more optimally than Scikit-learn MLPs. This difference in feature handling can influence how well the models discriminate between classes, potentially giving Keras models a performance advantage.

In [127…

```
# Dictionary to store the results
report = {'model': [['Low_XGB', 'High_XGB'],['Low_MLPK', 'High_MLPK'],['Low_MLPS', 'High_MLPS']],
          'Accuracy': [[acc_XGB_ll, acc_XGB_hl],[acc_MLPK_ll, acc_MLPK_hl],[acc_MLPS_ll, acc_MLPS_hl]],
          'Precision': [[prec_XGB_ll, prec_XGB_hl],[prec_MLPK_ll, prec_MLPK_hl],[prec_MLPS_ll, prec_MLPS_hl]],
          'Recall': [[rec_XGB_ll, rec_XGB_hl],[rec_MLPK_ll, rec_MLPK_hl],[rec_MLPS_ll, rec_MLPS_hl]]
         }

ResultsPlot(report)
```



Phase 6: Conclusions

Based on the analysis and interpretations of AUC values for Extreme Gradient Boosting (XGB), Multilayer Perceptrons (MLP) with Keras, and MLP with Scikit-learn, let's delve deeper into how Deep Learning methods, especially MLPs, excel over shallow models like XGB, particularly when dealing with raw or Low-Level features in high-energy physics data:

Deep Learning vs. Shallow Models in Handling Low-Level Features

1. **Feature Engineering and Model Performance**

- **XGB Performance (Low-Level):** XGB achieves an AUC of 0.675 with Low-Level features, indicating moderate performance. This suggests that XGB, while capable, struggles to extract complex patterns directly from raw or minimally processed data without additional feature engineering.
- **MLP with Scikit-learn Performance (Low-Level):** Similarly, MLP with Scikit-learn achieves an AUC of 0.786 with Low-Level features, slightly better than XGB. This indicates that Scikit-learn MLP, with its neural network architecture, can leverage some nonlinear relationships but still requires feature engineering to optimize performance.

2. **Deep Learning Advantages (Low-Level Features)**

- **MLP with Keras Performance (Low-Level):** In contrast, MLP with Keras achieves an AUC of 0.727 with Low-Level features, demonstrating improved capability compared to XGB and Scikit-learn MLP, even with a relatively small dataset of 1 million observations. This improvement suggests that Keras, with its TensorFlow backend, allows for more flexible and deeper model architectures that can learn intricate patterns directly from raw data.

3. **Complex Pattern Learning**

- **Capability of Deep Learning:** Deep learning, particularly MLPs implemented with Keras, excels in learning complex nonlinear functions directly from data without relying heavily on manually constructed features. This is crucial in high-energy physics where discovering subtle patterns in raw detector data is paramount.
- **Advantages Over Shallow Models:** MLPs with Keras can automatically discover and leverage powerful combinations of features, especially Low-Level, to enhance classification accuracy. This ability reduces the dependence on domain expertise for feature engineering, thereby accelerating model development and improving adaptability to varying data characteristics.

4. **Model Flexibility and Optimization**

- **Backend and Architecture:** Keras provides a robust framework optimized for neural network computations, which enhances the efficiency of model training and inference. This backend optimization is particularly advantageous in handling large-scale, high-dimensional data typical in particle physics experiments.

Implications

Deep Learning methods, especially MLPs implemented with Keras/TensorFlow, outperform shallow models like XGB when dealing with Low-Level features in high-energy physics data classification tasks. The ability of MLPs to autonomously learn complex patterns from raw data underscores their superiority in capturing nuanced distinctions between Signal and Background classes.

For future advancements in high-energy physics data analysis:

- **Automated Feature Discovery:** Deep Learning's capability to automatically extract discriminative features from raw data can lead to more accurate and robust classification models.
- **Reduced Manual Intervention:** The reduced need for manual feature engineering allows physicists to focus more on scientific insights and less on preprocessing data, potentially accelerating discoveries of exotic particles and phenomena.

In summary, leveraging Deep Learning, particularly MLPs with Keras, represents a significant step forward in addressing the complexities of high-energy physics data analysis, enhancing both accuracy and efficiency in classification tasks.