IIT Bombay

PH 303: Student Learning Project

Gluon and Quark Jet Classification

Student Name	**Student ID**
1. Nahush Rajesh Kolhe	210260034

Professor in charge:

Prof. Sadhana Dash

Date : 14/04/24

Contents

Abstract

Understanding the origin of jets in high-energy particle collisions is fundamental to unraveling the mysteries of Quantum Chromodynamics (QCD). In our project, *Gluon and Quark Jet Classification using ML*, we delve into this realm by simulating events that yield jets initiated by quarks and gluons using Pythia and FastJet. Through meticulous analysis, we delineate distinctive features pertinent to their classification. Leveraging a diverse array of Machine Learning models, we endeavor to discern between quark-initiated and gluon-initiated jets on a per-event basis. Our findings underscore the efficacy of our approach, culminating in an achieved accuracy of 78.6%.

1 Introduction

The exploration of jet substructure at the CERN Large Hadron Collider (LHC) has been pivotal in advancing our understanding of the standard model (SM) of particle physics, particularly through the analysis of jets generated by Quantum Chromodynamics (QCD). Consequently, there has been extensive research into characterizing and distinguishing between light quark- and gluon-initiated jets that constitute QCD jets.

The data collected at the LHC exists in the form of detector-level information. This data, originating from various subdetectors, undergoes a process of combination and processing to reconstruct particle-level information, representing the 4-momenta of all resolved particles resulting from proton–proton collisions at the LHC. Subsequently, a jet clustering algorithm is applied to group nearby particles into a jet object. Several jet-level features can be derived to describe the jet as a whole, including its shape and particle multiplicity.

Accurately distinguishing between quark-initiated and gluon-initiated jets at the LHC holds significant promise, particularly given that signatures of physics beyond the standard model often exhibit a prevalence of quarks. Additionally, differentiating between quark and gluon jets within a single event can help reduce combinatorial ambiguity. Identifying the nature of jets in an event can constrain their position in proposed decay topologies or classify them as initial-state radiation.

In recent years, machine learning (ML) techniques have gained prominence for this classification task, offering a means to extract valuable insights. State-of-the-art methods typically employ jet images as input, created by pixelating particle-level data into a grid resembling a histogram-like image that captures detector geometry. These images are then fed into convolutional neural networks (CNNs) for classification.

However, our approach focuses on utilizing jet features as input to a Feed Forward Neural Network, as well as other classifiers such as random forest and XGBoost, for jet classification. We employ Pythia 3.8 to simulate events involving gluon-initiated and quark-initiated jets, utilizing FastJet tools to cluster them into jets. The features extracted from these jets serve as input for our ML models. Through this classification process, we aim to gain valuable insights into the distinguishing features of quark- and gluon-initiated jets, reaffirming established theoretical results and experimental observations.

2 Theory

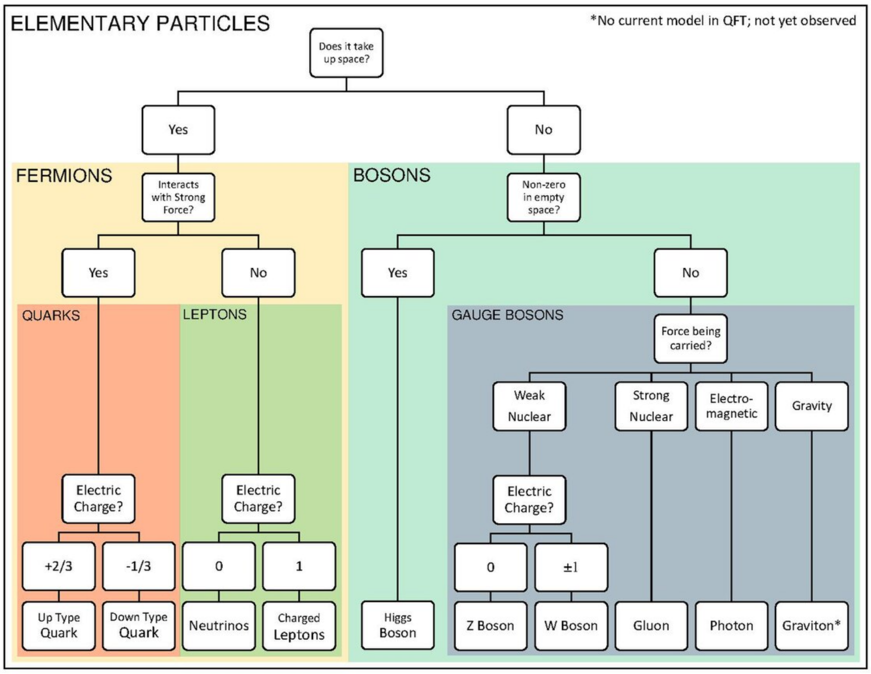

Figure 1: Standard Model Flowchart

- **Elementary Particles**:
  - Quarks: Quarks are fundamental particles that are the building blocks of hadrons, such as protons and neutrons. There are six types, or "flavors", of quarks: up (u), down (d), charm (c), strange (s), top (t), and bottom (b).
  - Gluons: Gluons are the force carriers of the strong nuclear force. They mediate the interactions between quarks and are responsible for holding them together inside hadrons. Gluons carry color charge and participate in strong interactions.
- **Hadrons**:
  - Hadrons are composite particles made up of quarks and/or antiquarks held together by the strong force. There are two main types of hadrons: baryons and mesons.
  - Baryons: Made up of three quarks, such as protons and neutrons.
  - Mesons: Composed of one quark and one antiquark, such as pions.
- **Color Charge**:
  - Quarks and gluons carry a property called *color charge*, analogous to electric charge in electromagnetism. However, color charge comes in

three types: red, green, and blue, as well as their respective anticolors (antired, antigreen, and antiblue).

  - Quarks can have either one "color" and one "anticolor" (making them color-neutral) or three colors (forming a "color triplet"). Gluons carry both a color and an anticolor, allowing them to interact with other quarks and gluons.

- **Quantum Chromodynamics (QCD)**:
  - QCD is the quantum field theory that describes the strong force and its interactions between quarks and gluons, resulting in the formation of hadrons and the rich spectrum of phenomena observed in particle physics experiments.
  - The photon is described in QED as the “force-carrier” particle that mediates or transmits the electromagnetic force. By analogy with QED, quantum chromodynamics predicts the existence of force-carrier particles called gluons, which transmit the strong force between particles of matter that carry “colour,” a form of strong “charge.” The strong force is therefore limited in its effect to the behaviour of elementary subatomic particles called quarks and of composite particles built from quarks

- **Hard/Soft Scattering**: Hard/Soft scattering refers to interactions between particles where the exchanged momentum is large/small compared to the overall energy scale of the process. In QCD, soft scattering typically involves the exchange of low-momentum gluons between quarks, leading to relatively gentle changes in the momentum of the interacting particles. Hard scattering processes typically involve high-energy collisions between partons (quarks and gluons), resulting in significant momentum transfers and the ***production of jets*** or other high-momentum final states.

- **Momentum Transfer**: Momentum transfer, also known as momentum exchange or transverse momentum, is the transfer of momentum between particles during a collision or interaction. In QCD, momentum transfer plays a crucial role in determining the kinematics and dynamics of scattering processes, influencing the final state particles' momenta and angles.

- **Center-of-Mass (COM) Frame Analysis in QCD**: In QCD, the center-of-mass frame is a reference frame where the total momentum of the colliding particles is zero. This frame is commonly used in high-energy physics experiments to extract information about fundamental particle properties and interactions.

- **Parton Distribution Functions (PDFs)**: Parton distribution functions are mathematical descriptions of the internal structure of protons and neutrons in terms of their constituent partons (quarks and gluons). PDFs quantify the probability of finding a parton with a specific momentum fraction inside a hadron, providing essential information for predicting the outcomes of high-energy scattering processes involving hadrons, such as proton-proton collisions at particle accelerators like the Large Hadron Collider (LHC).

- **Jets**:

- **Definition**: A jet is a narrow cone of hadrons and other particles produced by the hadronization of quarks and gluons in a particle physics or heavy ion experiment.
- Jets are created in high-energy collisions, such as proton-proton collisions at particle accelerators like the (LHC). When energetic partons are produced in these collisions, they subsequently undergo a process called fragmentation, where they produce showers of secondary particles. These particles are clustered together to form jets.
- Jets typically exhibit cone-like structures in detector measurements, reflecting the angular distribution of the particles within the jet. They carry information about the initial parton's energy and momentum, as well as insights into the dynamics of the collision process.
- **Gluon-initiated jets/Quark-initiated jets** are jets primarily initiated by the fragmentation of high-energy gluons/quarks produced in a collision.
- Gluons typically undergo more frequent branching and radiation compared to quarks due to their self-interaction. As a result, gluon-initiated jets tend to have a ***higher particle multiplicity***, with more secondary particles produced in the fragmentation process.
- Gluons can radiate gluons, leading to a softer energy spectrum compared to quark-initiated jets, which often exhibit a ***harder energy distribution***.

3 Dataset Generation using Pythia

This section will extensively cover how we have created the dataset containing features of gluon and quark initiated jets for classification.

3.1 Introduction

I have used the following software frameworks for obtaining this dataset.

- **ROOT CERN**: is a powerful software framework developed by CERN for data processing, analysis, and visualization in high-energy physics (HEP) experiments. It provides a comprehensive set of tools and libraries for manipulating large datasets, performing statistical analysis, and creating high-quality graphical representations of scientific data. With its advanced graphics capabilities, ROOT enables users to create complex plots, histograms, and 3D visualizations to explore and interpret experimental results effectively.

  In our project we will mainly use ROOT for graphing purposes particularly 1D and 2D histograms of jet features.

- **Pythia**: is an event generator, which is an evolving physics tool used to answer fundamental questions in particle physics. The program is most often used to generate high-energy-physics collision "events", i.e. sets of particles produced in association with the collision of two incoming high-energy particles, but has several uses beyond that. It simulates the full event evolution, including initial-state radiation, parton showering, hadronization, and decay processes, based on theoretical models derived from perturbative QCD.

Pythia offers extensive customization options, allowing users to configure various aspects of the event generation process, such as the choice of PDF sets, parton distribution functions, and underlying event models. It is mainly written in C++

In our project we will use Pythia extensively to generate events which gives rise to quark and gluon initiated jets and extract their features.

- **FastJet**: is a popular software package designed for efficient and robust jet clustering in high-energy physics applications. It implements several jet clustering algorithms, including the well-known anti-kT and Cambridge/Aachen algorithms, which are widely used for reconstructing jets from particle-level data in collider experiments.

  In our project, Fastjet will be heavily utilized to cluster particles into jets using the anti-kT jet clustering algorithm. These resulting jets will possess distinct features, which we will leverage for generating our dataset.

3.2 Software Setup

The following steps outline the setup process for ROOT, Pythia, and Fastjet on my laptop running Windows 11 OS.

1. Initially, I installed Windows Subsystem for Linux 2 (WSL-2), which provides a compatibility layer for running Linux binary executables natively on Windows 11 and other operating systems. This integration enables the execution of Linux command-line tools and utilities directly on Windows machines without the need for a traditional virtual machine or dual-boot setup.

2. With the transition to a Linux environment facilitated by WSL-2, I proceeded to install ROOT from the official ROOT CERN website, adhering to the prescribed instructions. It was imperative to add the necessary environment variables, include the ROOT bin directory in the PATH, and configure other environment variables like ROOTSYS.

3. Subsequently, I installed Pythia from pythia.org, following the provided instructions. During the installation process, I opted for the 'with root' function to seamlessly configure ROOT with Pythia.

4. Lastly, I installed Fastjet from its official website and followed the provided quick start instructions.

5. With the installation of all three components completed, the setup was nearly finalized, requiring the configuration of Pythia with Fastjet. This involved editing the Makefile.inc located in the examples folder of Pythia, the

primary directory for my coding endeavors. In the Makefile.inc, I added the requisite paths pointing towards various Fastjet files. Similarly, if ROOT was not already configured, similar adjustments could be made. The following code snippets demonstrate this configuration:

```
FASTJET3_USE=true
FASTJET3_CONFIG=fastjet-config
FASTJET3_BIN=
FASTJET3_INCLUDE=-I/home/nahush/fastjet-install/include
FASTJET3_LIB=-L/home/nahush/fastjet-install/lib -Wl,-rpath,/home/na

ROOT_USE=true
ROOT_CONFIG=root-config
ROOT_BIN=/home/nahush/root-6.30.04-install/bin/
ROOT_INCLUDE=-I/home/nahush/root-6.30.04-install/include
ROOT_LIB=-L/home/nahush/root-6.30.04-install/lib -Wl,-rpath,/home/r
```

6. Finally, I added a rule to the makefile for all the files I intended to create and execute. This rule included the necessary ROOT and Fastjet paths, allowing for the seamless execution of these files using the make command. Below is an example of this rule, utilizing 'test' and 'test2' files for coding purposes.

```
test test2: $(PYTHIA) $$@.cc
ifeq ($(FASTJET3_USE),true)
	$(CXX) $@.cc -o $@ -w $(CXX_COMMON) $(ROOT_LIB) $(shell $(ROOT_CON
else
	$(error Error: $@ requires FASTJET3)
endif
```

3.3 Generating Dataset

In this section, I describe the Pythia settings utilized for generating features of gluon and quark initiated jets.

I establish two event generators: `pythia_g` and `pythia_q` for generating gluon and quark initiated jets, respectively. Both generators are configured with a Beam Center of Mass Energy of 13 TeV and a phase space $\hat{p}_{t_{min}}$ of 20 GeV, emphasizing high-energy interactions.

To optimize internal parameters for our particular situation, the `tune` parameter is set to 14, which ensures somewhat more accurate outcomes. To further improve the realism of the produced data, Initial State Radiation is set up to imitate the emission of extra partons (quarks or gluons) from the arriving protons prior to the hard scattering process.

For `pythia_g`, processes that explicitly produce gluons in their products are enabled, resulting in gluon-initiated jets. These processes include `HardQCD:qqbar2gg`, `HardQCD:gg2gg`, and `HardQCD:gg2ggg`. These processes are categorized as Hard Scattering, as jets are commonly observed in such interactions due to the higher momentum transfer, leading to jet formation.

Similarly, for `pythia_q`, processes such as `HardQCD:gg2qqbar` and `HardQCD:qq2qq` are enabled to generate quark-initiated jets.

Jets are defined using the FastJet library with the anti-k_t algorithm and a jet radius of 0.5.

3.3.1 Anti-kT Algorithm

The anti-k_t algorithm is a jet clustering algorithm commonly used in particle physics to reconstruct jets from particles produced in high-energy collisions. It is particularly useful for identifying jets originating from hard scattering processes like ours. Here's a brief explanation of how it works:

1. The process begins by obtaining input particles, each of which has a position and momentum. Then, using the formula $\Delta R = \sqrt{(\Delta\eta)^2 + (\Delta\phi)^2}$, it determines the distance (measure of angular separation) between each pair of particles in the event, where $\Delta\eta$ and $\Delta\phi$ represent variations in pseudorapidity and azimuthal angle, respectively.

2. The pairs of particles are iteratively combined according to their distances as the algorithm moves forward. It tends to group soft, close particles with high-momentum particles (seeds) and gives priority to the development of jets around these seeds.

3. **Jet Definition**: The jet radius, or parameter R, defined by the anti-k_t algorithm controls the size of the ensuing jets. A particle is said to be a component of the same jet if it is located within R distance of a seed particle. This parameter helps lessen the impacts of soft radiation and permits control over the size of the jet cone.

4. Following the clustering of all particles into jets, the technique gives a vector sum of the momenta of the particles that make up each jet's momentum.

3.3.2 Features of Dataset

1. **no_of_jets**: The number of jets produced in a collision event.

2. **lead_jet_pt**: The transverse momentum (momentum perpendicular to the beam axis) of the leading jet, indicating its energy and direction. We know that quarks have higher energy spectrum compared to gluons, hence this is an important feature.

3. **lead_jet_eta**: The pseudorapidity of the leading jet, which measures its angle relative to the beam axis.

4. **lead_jet_phi**: The azimuthal angle of the leading jet, indicating its direction perpendicular to the beam axis.

5. **lead_jet_mass**: The invariant mass of the leading jet, which characterizes its total energy and momentum.

6. **lead_jet_energy**: The total energy of the leading jet, including both its kinetic and rest energy. Similar to lead_jet_pt, quarks have higher energy spectrum as compared to gluons.

7. **lead_jet_multiplicity**: The number of particles within the leading jet, reflecting its complexity. Here we know that gluon jets tend to have higher jet multiplicity, hence this is also an important feature.

8. **lead_jet_m/pt**: The ratio of the leading jet's mass to its transverse momentum, providing insight into its massiveness relative to its momentum. It is also known as Jet Shape. The jet shapes is generally high for gluons as compared to quarks.

9. **avg_pt**: The average transverse momentum of particles within the leading jet, indicating the typical momentum of its constituent particles.

10. **var_pt**: The variance of transverse momentum within the leading jet, measuring the spread or uniformity of particle momenta.

11. **avg_distance**: The average distance between particles within the leading jet, offering a measure of jet compactness.

12. **var_distance**: The variance of distances within the leading jet, indicating the spread or clustering of its constituent particles.

13. **var_phi**: The variance of azimuthal angles within the leading jet, providing information about its directional spread.

14. **jet_width_1**: It is basically the linear radial moment as defined in [4], as $\sum_{i \in \text{jet}} \frac{p_{iT}}{p_{\text{jet}T}} \cdot r_i$ which gives idea about the width of the jet.

15. **jet_width_2**: $\sum_{i \in \text{jet}} \frac{p_{iT}}{p_{\text{jet}T}} \cdot \sqrt{r}_i$

16. **sublead_jet_pt**: The transverse momentum of the subleading jet, providing insight into the kinematics of secondary jets.

17. **sublead_jet_eta**: The pseudorapidity of the subleading jet, offering information about its angular distribution.

18. **sublead_jet_phi**: The azimuthal angle of the subleading jet, indicating its direction relative to the beam axis.

19. **target**: The target variable or label indicating whether the jet is initiated by a gluon for which target is 1, or a quark for which the target is 0.

3.3.3 Main Event Loop

With consideration of these predetermined features, we embark on our primary event loop focused on gluons, encompassing 1,000,000 events. Each event's particle momentum, defined by its energy and directional components, is encapsulated within a Pseudojet class instance named "particles." Subsequently, we traverse through each particle within the event, converting them into pseudojets. Utilizing the original jet definition, we apply FastJet clustering to these particles, stipulating a minimum momentum threshold of 5 GeV.

Upon attaining all jets within the event, we proceed to extract their respective features following prescribed formulas, subsequently storing this information within a CSV file, constituting our dataset.

A similar iterative process is executed for quarks, extracting their features and appending them to the same CSV file. Notably, an additional target column is introduced in the CSV file, assigning a value of 1 to gluon-initiated jets and 0 to quark-initiated jets.

Simultaneously, as feature extraction transpires, we define histograms for pertinent features using ROOT, populating them through the event loop. Subsequently, these histograms can be preserved as ROOT files or transformed into image formats such as PNG. Subsequent analysis will focus on interpreting the obtained histograms.

3.4 Graphs and their Analysis

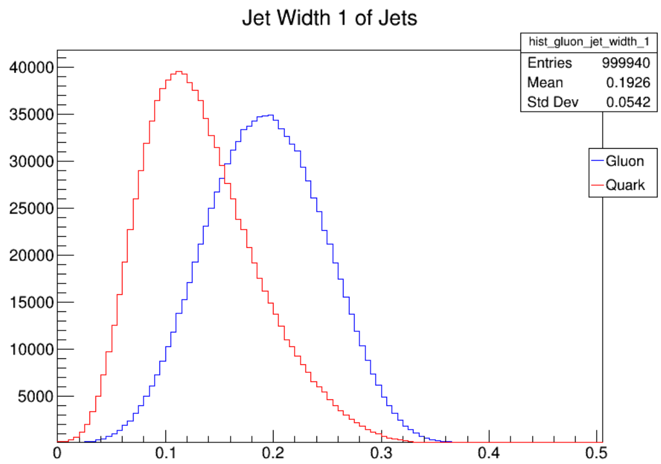

Figure 2: Jet Width 1 in phi-eta plane distance units

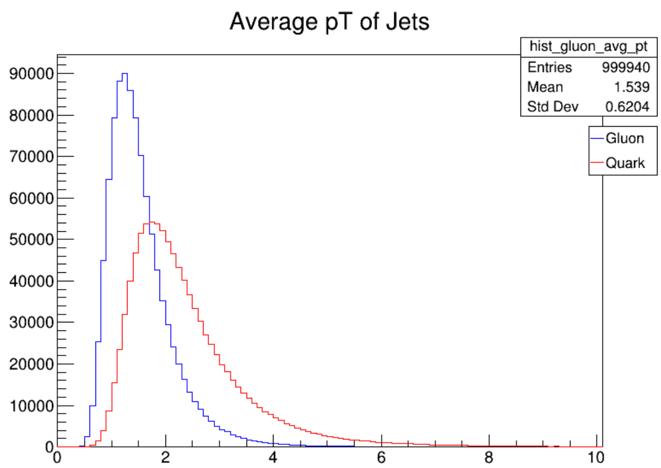

Figure 3: Average pt in GeV

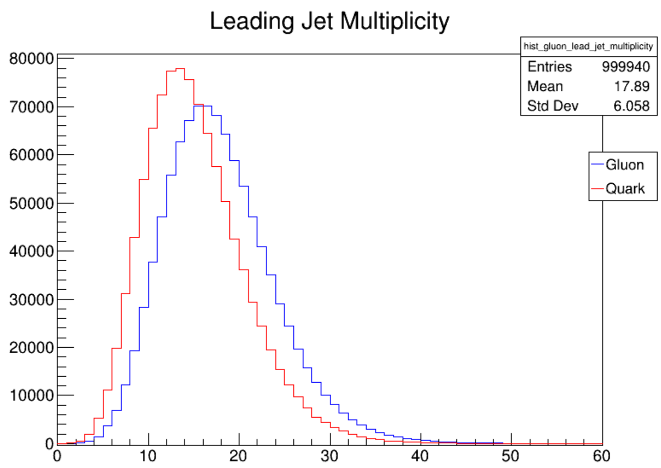

Figure 4: Lead Jet Multipicity

Gluon jets exhibit higher width and multiplicity due to their diffuse energy distribution and tendency to radiate additional gluons and quark-antiquark pairs. This results in lower average transverse momentum and energy compared to quark jets, reflecting the inherent differences in the properties and interactions of gluons and quarks.

We can clearly see that the Jet width and jet multiplicity of gluons are higher than the quarks, pt is higher for quarks. This aligns with our known results (theoretical/experimental), and hence these are some good features which we can use for distinguishing the jets. Similar observations can be made to other features whose graphs I have included in the Appendix.

4 Classification Using ML

In this section, we'll use various machine learning methods to classify our dataset. It's a binary classification task where we have 18 features, some are floating point numbers, others are whole numbers. The target variable is either 1 for gluon-initiated jets or 0 for quark-initiated jets.

4.1 Data Analysis

The Python code utilizes libraries such as PyTorch and Scikit-learn. We start by importing the dataset and dividing it into features and target columns. Next, we employ the train-test split method to divide the data into training and testing subsets, allocating 20% of the data for testing. Additionally, we convert these subsets into tensors, as required by PyTorch.

To gain insights into the dataset, we generate a correlation heatmap. This heatmap reveals the correlations between features and provides insights into their importance by comparing them with the target variable.

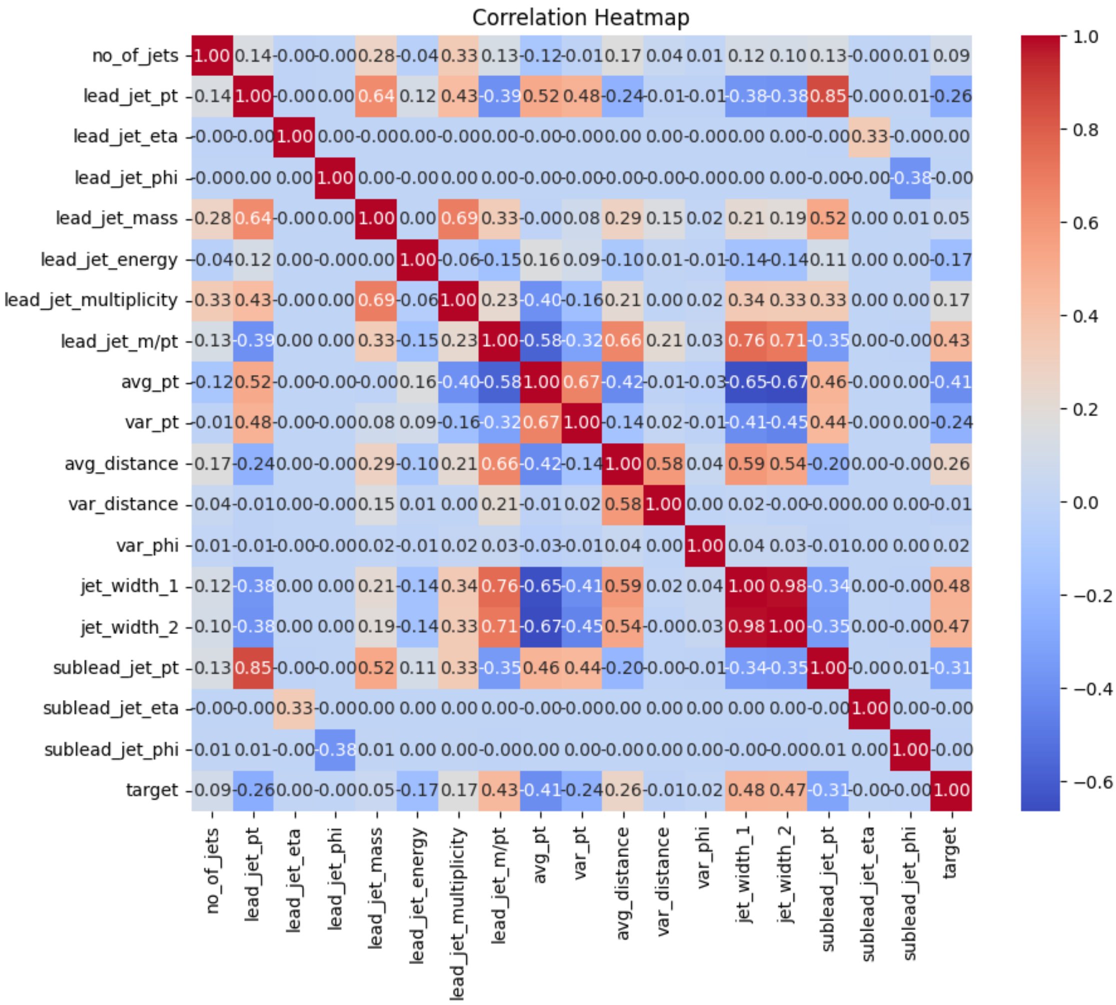

Figure 5: Correlation Heat map of the Dataset

Indeed, certain features such as jet width, mass/pt, average pt, average distance, and multiplicity exhibit relatively strong correlations with the target variable. This suggests that these features hold significance in distinguishing between gluon and quark-initiated jets.

Lets us now move to classifying our dataset

4.2 Logistic Regression

Logistic regression is a method for binary classification, predicting the probability of an event occurring based on input features. It models the relationship between the independent variables X and the probability of the binary outcome Y using the logistic function $g(z)$:

$$g(z) = \frac{1}{1 + e^{-z}}$$

where z is a linear combination of the input features:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

Here, $\beta_0, \beta_1, \ldots, \beta_n$ are the coefficients or weights of the model, and $x_1, x_2, \ldots, x_n$ are the input features.

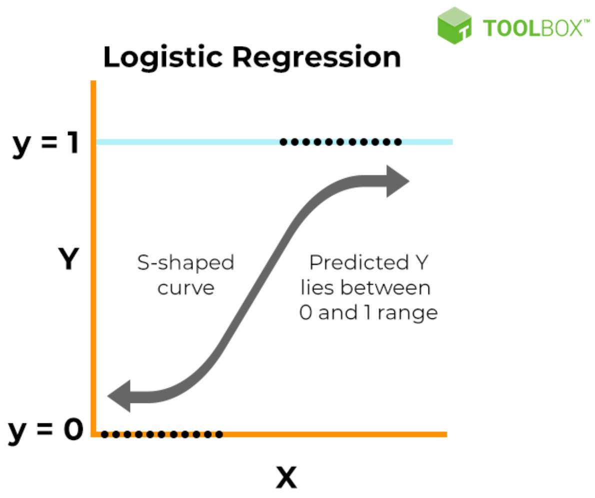

Figure 6: Sigmoid Function

The logistic regression model predicts the probability $P(Y = 1|X)$ of the positive class given the input features X as:

$$P(Y = 1|X) = g(z)$$

where $g(z)$ is the logistic function.

Optimization Objective: To train the logistic regression model, we minimize the logistic loss function, also known as the cross-entropy loss:

$$\mathcal{L}(\hat{y}, y) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

where $\hat{y}$ represents the predicted probabilities, y is the true labels, and N is the number of samples.

Logistic regression assumes that the relationship between the independent variables and the log-odds of the dependent variable is linear. It requires independent observations, a binary dependent variable, no outliers, and a sufficiently large sample size.

We use this method by directly importing the logistic regression class from sklearn and fit it on our train data and test it.

4.3 Support Vector Machines

Support Vector Machine (SVM) is a supervised learning algorithm used for classification and regression tasks. It aims to find the optimal hyperplane that best separates the classes in the feature space by maximizing the margin between classes while minimizing classification errors.

Mathematical Formulations:
Linear SVM: In linear SVM, the decision boundary is represented as a linear combination of input features:

$$f(x) = w^T x + b$$

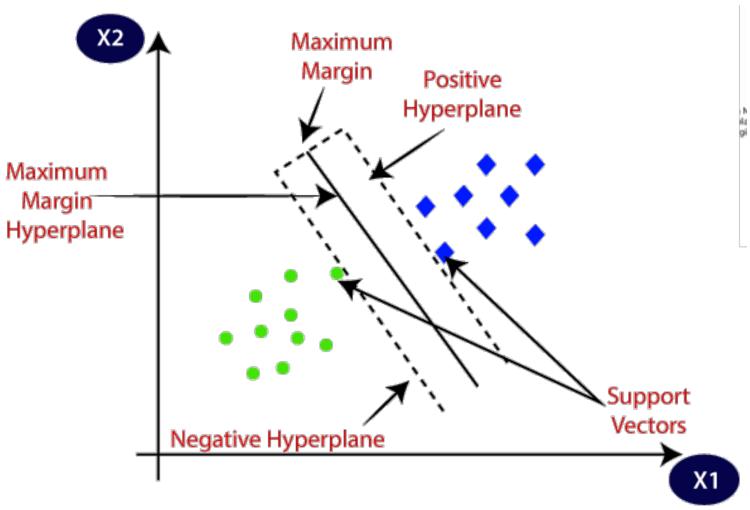

Figure 7: Support Vector Machines

The goal is to find the optimal weight vector w and bias term b that maximizes the margin between classes.

Optimization Objective: For hard-margin linear SVM:

$$\underset{w,b}{\text{minimize}} \frac{1}{2}||w||^2$$

$$\text{subject to } y_i(w^T x_i + b) \geq 1 \text{ for } i = 1, 2, \ldots, m$$

For soft-margin linear SVM:

$$\underset{w,b,\xi}{\text{minimize}} \frac{1}{2}||w||^2 + C\sum_{i=1}^{m} \xi_i$$

$$\text{subject to } y_i(w^T x_i + b) \geq 1 - \xi_i, \xi_i \geq 0$$

Non-Linear SVM (RBF Kernel): The Radial Basis Function (RBF) kernel measures the similarity between two samples based on their Euclidean distance:

$$K(x_i, x_j) = \exp\left(-\gamma||x_i - x_j||^2\right)$$

The decision function for SVM with RBF kernel:

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b$$

Optimization Objective (Dual Formulation): The dual problem of SVM involves maximizing the following objective function:

$$\underset{\alpha}{\text{maximize}} \left(\sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j)\right)$$

$$\text{subject to } \sum_{i=1}^{n} \alpha_i y_i = 0, \text{ and } 0 \leq \alpha_i \leq C \text{ for } i = 1, 2, \ldots, n$$

In this project, **the RBF kernel is being used**, which is suitable for capturing complex non linear relationships in the data. In the project, SVM is imported from scikit-learn library and utilized for classification tasks.

4.4 Random Forest

Random Forest (RF) is an ensemble learning algorithm for classification and regression tasks, utilizing decision trees. It constructs a multitude of decision trees during training and outputs the mode of the classification classes.

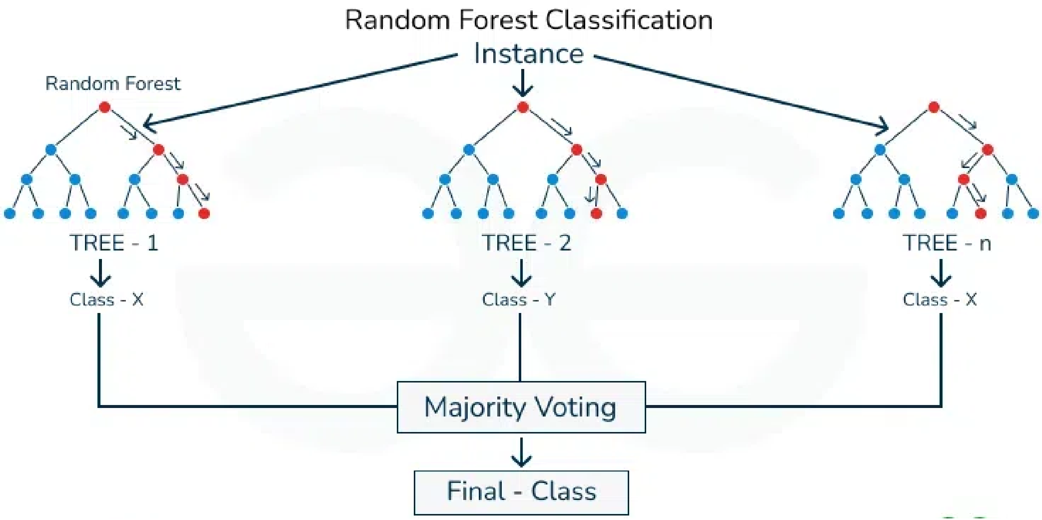

Figure 8: Random FOrest Working

Algorithm:

- RF builds decision trees using a subset of training data and a random subset of features, reducing overfitting.
- Each tree is built recursively, selecting the best feature at each split point based on criteria such as information gain or Gini impurity.
- The final prediction is made by aggregating the predictions of individual trees.

Decision Trees:

- A decision tree splits the feature space into regions based on feature values, with each node representing a decision.
- At each split, the algorithm selects the feature and split point that minimizes impurity in child nodes.
- Gini impurity, a measure of node purity, is often used in classification trees:

$$\text{Gini}(t) = 1 - \sum_{i=1}^{C} p(i|t)^2$$

where C is the number of classes and $p(i|t)$ is the probability of class i in node t.

Feature Selection:

- RF uses feature bagging, selecting a random subset of features for each tree, promoting a diverse set of features.
- The algorithm evaluates feature subsets at each split point, selecting the best feature based on criteria such as information gain or Gini impurity.

Random Forest is imported from scikit-learn library for classification tasks. Grid search is used for hyperparameter tuning.

4.5 Neural Network

Neural networks are versatile machine learning models capable of approximating complex functions. Neural networks consist of layers of interconnected neurons, each performing a specific computation and passing their outputs to subsequent layers.

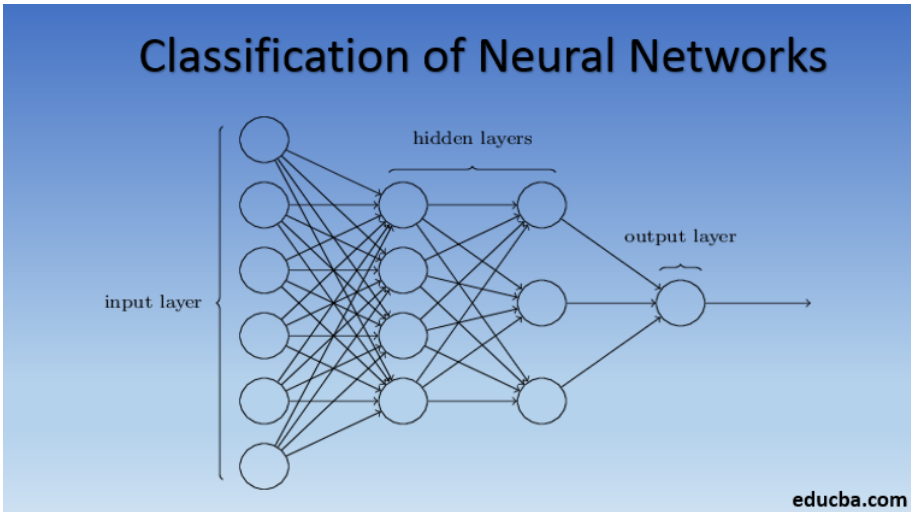

Figure 9: Neural Network For Classifictaion

Feedforward Neural Networks: A feedforward neural network, also known as a multi-layer perceptron (MLP), is a type of neural network where information flows in one direction, from input to output layers, without forming loops. The network approximates functions using the formula:

$$y = f(x; \theta)$$

where x represents the input, θ denotes the parameters (weights and biases) of the network, and f is the activation function.

Working Principle: In a simplified form, a single-layer perceptron, a basic unit of a feedforward neural network, multiplies input values with corresponding weights, sums them up, and applies an activation function to produce an output. Mathematically, this can be represented as:

$$y = \text{activation}(\sum_{i=1}^{n} w_i x_i + b)$$

where w_i are the weights, x_i are the input values, b is the bias, and the activation function determines the neuron's output.

Activation Function: Activation functions introduce nonlinearity into the neural network, allowing it to learn complex patterns. Common activation functions include:

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
- Tanh: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- ReLU (Rectified Linear Unit): $\text{ReLU}(z) = \max(0, z)$

Gradient Descent: During training, neural networks adjust their parameters using optimization algorithms such as gradient descent. The weights and biases are updated iteratively to minimize the cost function by computing the gradient of the cost function with respect to each parameter. The update rule for gradient descent is given by:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla J(\theta)$$

where θ represents the parameters, η is the learning rate, and $\nabla J(\theta)$ is the gradient of the cost function.

Universal Approximation Theorem: The universal approximation theorem states that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of the input space, under mild assumptions. This theorem underscores the remarkable expressive power of neural networks.

Model Architecture: In this code, we define a neural network model called ComplexModel using PyTorch's nn.Module class. This model is designed for our binary classification task of classifying gluon and quark initiated jets.

```

class ComplexModel(nn.Module):
    def __init__(self, input_size):
        super(ComplexModel, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.dropout1 = nn.Dropout(0.1)
        self.fc2 = nn.Linear(64, 128)
        self.dropout2 = nn.Dropout(0.1)
        self.fc3 = nn.Linear(128, 64)
        self.dropout3 = nn.Dropout(0.1)
        self.fc4 = nn.Linear(64, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.dropout1(x)
        x = torch.relu(self.fc2(x))
        x = self.dropout2(x)
        x = torch.relu(self.fc3(x))
        x = self.dropout3(x)
        x = self.fc4(x)
        x = self.sigmoid(x)
        return x
```

- The input size is specified when initializing the model.
- The model consists of three fully connected (linear) layers (fc1, fc2, fc3) with ReLU activation functions, which introduce non-linearity into the model.
- Dropout layers (dropout1, dropout2, dropout3) are added after each hidden layer to prevent overfitting. Dropout randomly sets a fraction of input units to zero during training, reducing the interdependence of neurons and preventing the network from relying too heavily on specific features.
- The final layer (fc4) is a fully connected layer with a single output unit, followed by a sigmoid activation function (sigmoid). Sigmoid activation function squashes the output to a range between 0 and 1, making it suitable for binary classification tasks where the output represents the probability of belonging to the positive class.

In summary, this model utilizes a feedforward neural network architecture with ReLU activations for non-linearity, dropout layers for regularization, and a sigmoid output for binary classification.

In our code, the Binary Cross Entropy Loss (BCELoss) criterion is employed for calculating the loss, while the Adam optimizer with a learning rate of 0.001 is utilized for optimizing the model parameters.

5 Results

We train all the above models excpet the Neural Network on 5% of the data due to high computational cost. For Neural Netwrok we use the entire 2 million data points for training. However it is observed that even Neural Network when trained for 5% dataset, gives very close results.
The following are the resultant accuracies and metrics obtained for the above described models:

Table 1: Performance Comparison of Different Models

Model	**Precision**	**Recall**	**F1 Score**	**Accuracy**
Logistic Regression	0.76	0.76	0.76	75.5%
SVM (RBF Kernel)	0.78	0.78	0.78	78%
Random Forest	0.78	0.78	0.78	77.8%
Neural Network	**0.79**	**0.79**	**0.79**	**78.6%**

Additional Training Details:

- For the support vector machines, we utilized the radial basis function (RBF) kernel to capture non-linearities in the model.
- In the random forest, we employed grid search to fine-tune the hyperparameters. The best configuration we found was:
  - `max_depth`: None
  - `min_samples_leaf`: 2
  - `min_samples_split`: 2

- n_estimators: 300

- When designing the Neural Network models, we experimented with various architectures. We observed that using too few or too many parameters adversely affected performance. Eventually, we settled on the 'ComplexModel' architecture defined earlier, which struck a balance.

- We also experimented with different optimizers and learning rates for training the Neural Network. While Stochastic Gradient Descent and RMS Prop yielded similar results, they were slightly inferior to the Adam optimizer. Therefore, we opted for the Adam optimizer with a learning rate of 0.001. We trained the model with a batch size of 10,000 and for a total of 50 epochs.

5.1 Graphs

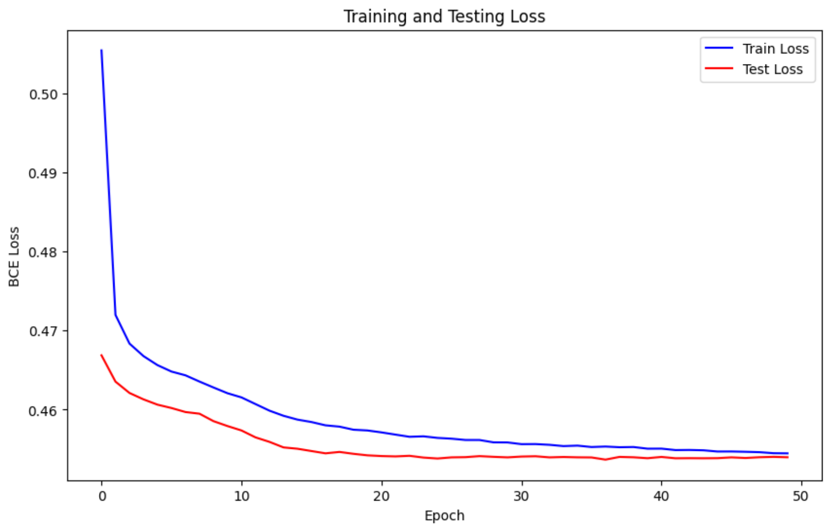

Figure 10: Loss of Neural Network over epochs

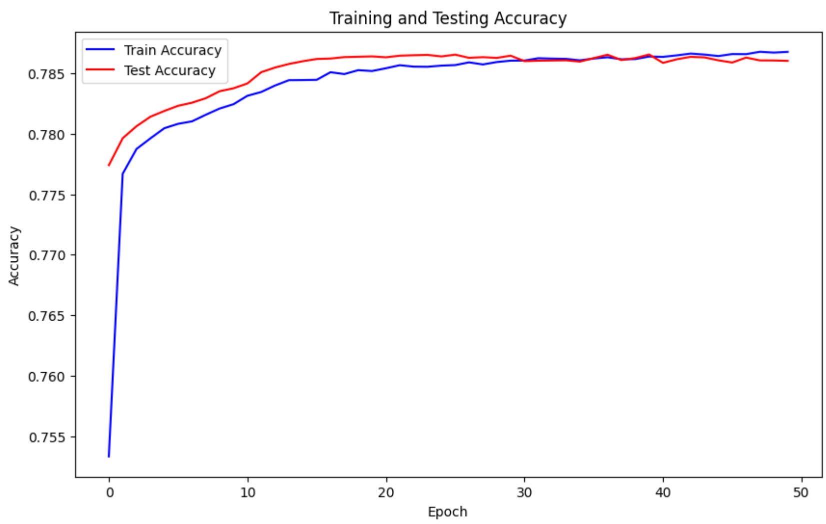

Figure 11: Accuracy of Neural Network over epochs

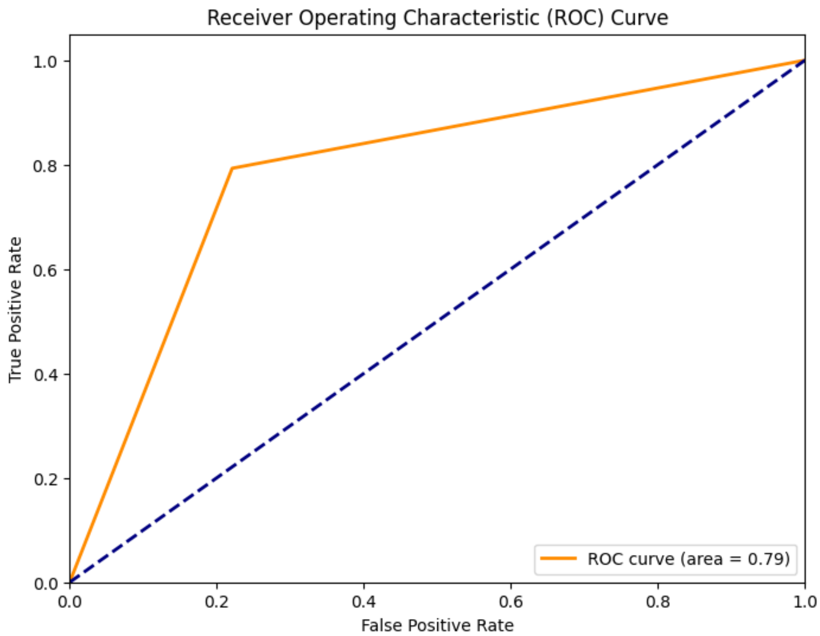

Figure 12: Receiver Operator Curve of Neural Network

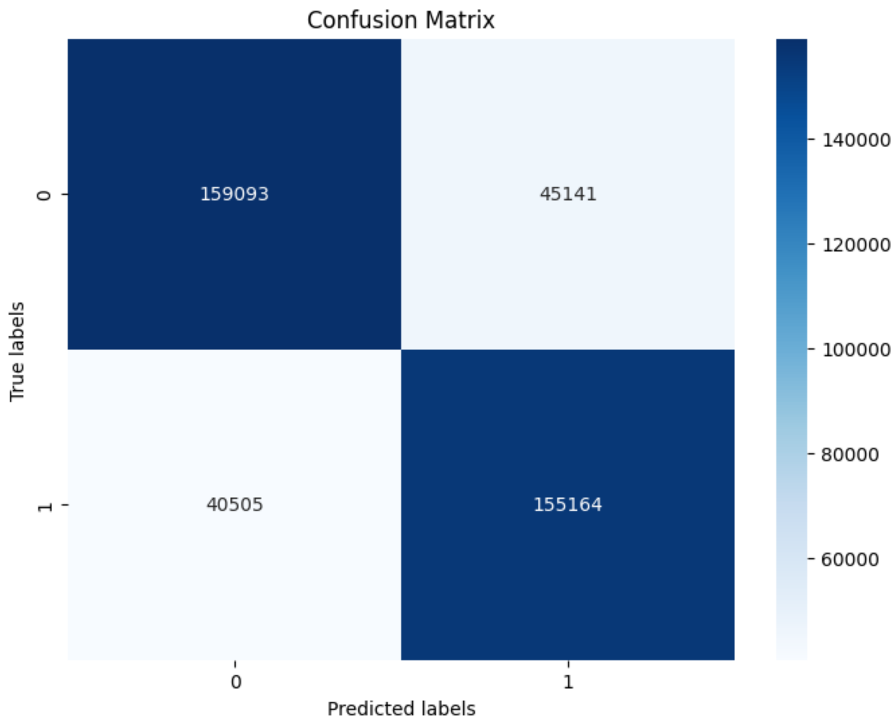

Figure 13: Confusion Matrix of Neural Network

6 Conclusion

In this project, we collected data using Pythia event generators, focusing on features relevant for distinguishing between gluon and quark initiated jets. We then applied machine learning techniques, including logistic regression, support vector machines with RBF kernel, random forest, and neural networks, to classify these jets.

Among these methods, neural networks showed a slight advantage over the others, likely due to their flexibility and ability to fine-tune various parameters. Overall, our study demonstrates the effectiveness of machine learning in distinguishing between quark and gluon jets, with neural networks showing particular promise in this task.

For future endeavors, a deeper analysis of particle-level features within jets could offer valuable insights. Additionally, exploring the application of convolutional neural networks (CNNs) for classification tasks could further enhance our understanding and classification accuracy.

References

[1] M. Andrews et al. "End-to-end jet classification of quarks and gluons with the CMS Open Data". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 977 (2020), p. 164304. ISSN: 0168-9002. DOI: `https://doi.org/10.1016/j.nima.2020.164304`. URL: `https://www.sciencedirect.com/science/article/pii/S0168900220307002`.

[2] Christian Bierlich et al. *A comprehensive guide to the physics and usage of PYTHIA 8.3*. 2022. arXiv: `2203.11601 [hep-ph]`.

[3] Matteo Cacciari, Gavin P Salam, and Gregory Soyez. "The anti-ktjet clustering algorithm". In: *Journal of High Energy Physics* 2008.04 (Apr. 2008), pp. 063–063. ISSN: 1029-8479. DOI: `10.1088/1126-6708/2008/04/063`. URL: `http://dx.doi.org/10.1088/1126-6708/2008/04/063`.

[4] Jason Gallicchio and Matthew D. Schwartz. "Quark and gluon jet substructure". In: *Journal of High Energy Physics* 2013.4 (Apr. 2013). ISSN: 1029-8479. DOI: `10.1007/jhep04(2013)090`. URL: `http://dx.doi.org/10.1007/JHEP04(2013)090`.

[5] *PYTHIA 8.3 - PYTHIA 8.3 — pythia.org*. `https://pythia.org/`. [Accessed 15-04-2024].

[6] ROOT team. *ROOT: analyzing petabytes of data, scientifically. — root.cern*. `https://root.cern/`. [Accessed 15-04-2024].

[7] Seungjin Yang. *Discriminating quark/gluon jets with deep learning*. `https://indico.cern.ch/event/661284/contributions/2699312/attachments/1521324/2376721/ML_Workshop.pdf`. [Accessed 15-04-2024].

7 Appendix

The link to the code is on this GitHub Repo: https://github.com/Nahush-27/Gluon-and-Jets-Classification-SLP

7.1 Additional Histograms

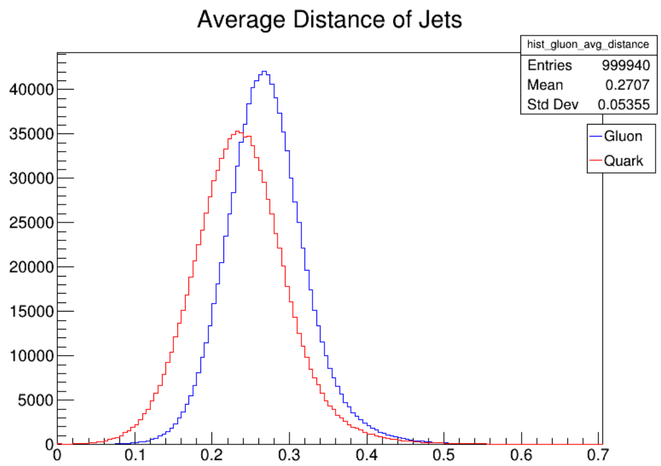

Figure 14: Average distance of Jets in eta-phi plane distance

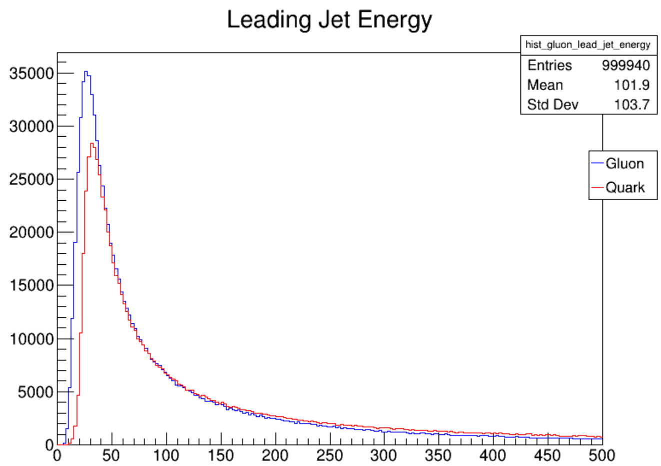

Figure 15: Lead Jet Energy in GeV

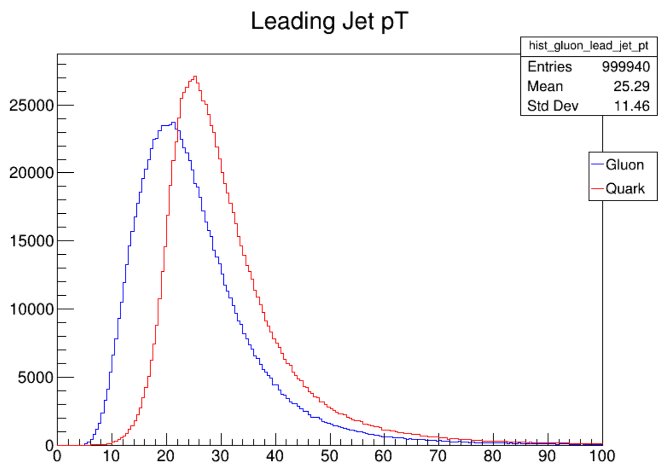

Figure 16: Lead Jet pt in GeV

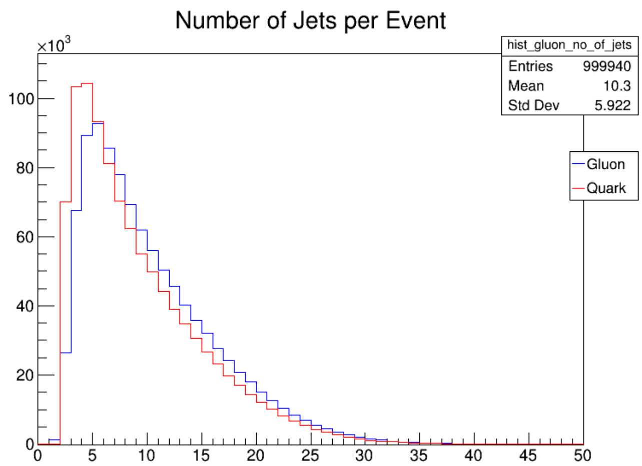

Figure 17: No of jets in the event

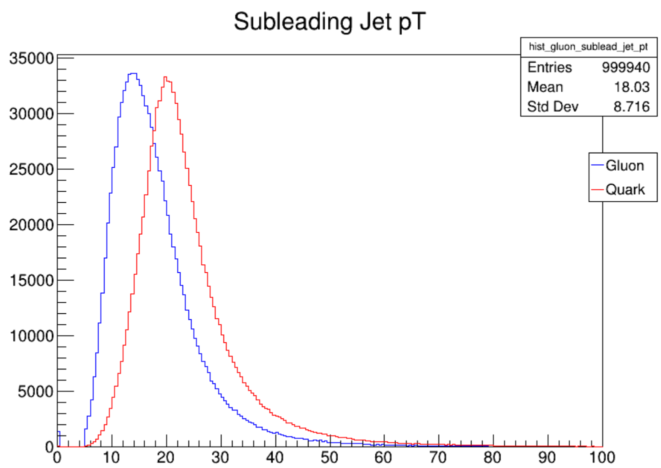

Figure 18: Sublead Jet pt in GeV

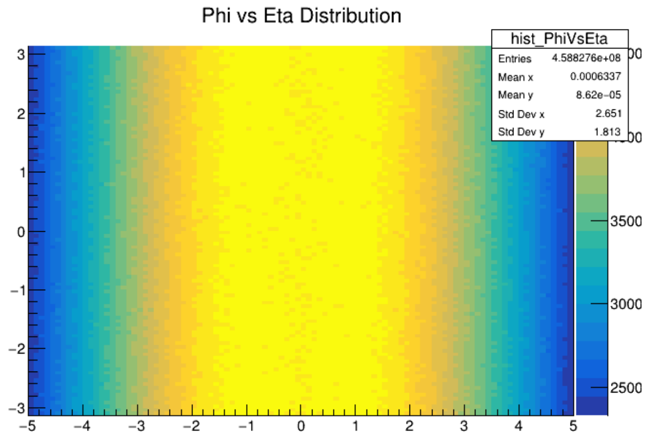

Figure 19: Quark phi vs eta 2-D histogram

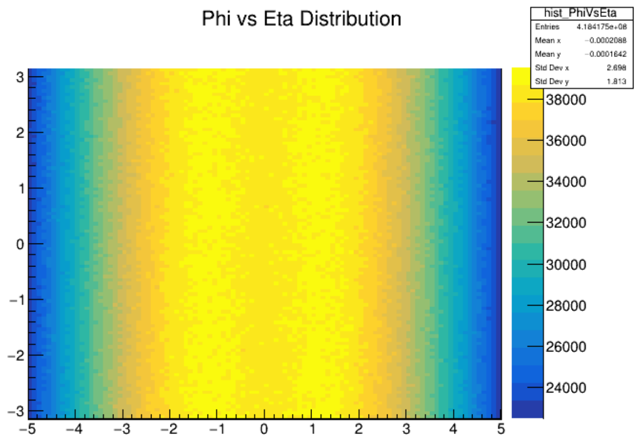

Figure 20: Quark phi vs eta 2-D histogram

7.2 Pythia Code

```

#include "Pythia8/Pythia.h"
#include "fastjet/ClusterSequence.hh"
#include "fastjet/AreaDefinition.hh"
#include "fastjet/GhostedAreaSpec.hh"
#include "TH1F.h"
#include "TH2F.h"
#include "TFile.h"
#include "TCanvas.h"
#include "TLegend.h"
#include <fstream>
#include <iostream>
#include <cmath>
#include <vector>

void saveHistogram2DAsROOT(TH2F &histogram, const char* filename)
    {
    // Create a canvas
    TCanvas canvas("canvas", "Histogram Canvas");

    // Draw the histogram on the canvas
    histogram.Draw("colz"); // "colz" option for a 2D color plot
```

```

    // Save the canvas and histogram in a ROOT file
    std::string fullFilename = std::string("Histograms") + "/" + std::string(filename);
    TFile outputFile(fullFilename.c_str(), "RECREATE");
    canvas.Write();
    histogram.Write();
    outputFile.Close();
}

void plotOverlayHistograms(TH1F& hist1, TH1F& hist2, const char* label1, const char* label2, const char* filename) {
    // Create a canvas
    TCanvas canvas("canvas", filename, 800, 600);

    // Draw the first histogram with blue color
    hist1.SetLineColor(kBlue);
    hist1.Draw();

    // Draw the second histogram with red color, overlaid on top of the first histogram
    hist2.SetLineColor(kRed);
    hist2.Draw("same");

    // Add a legend
    TLegend legend(0.88, 0.6, 0.98, 0.7);
    legend.AddEntry(&hist1, label1, "l");
    legend.AddEntry(&hist2, label2, "l");
    legend.Draw();

    // Save the canvas as a ROOT file
    std::string fullFilename = std::string("Histograms") + "/" + std::string(filename);
    TFile outputFile(fullFilename.c_str(), "RECREATE");
    canvas.Write();
    hist1.Write();
    hist2.Write();
    outputFile.Close();
}

// Function to calculate Euclidean distance between two points in the eta-phi plane
double euclideanDistance(double eta1, double phi1, double eta2, double phi2) {
    double deta = eta1 - eta2;
    double dphi = phi1 - phi2;
    while (dphi > M_PI) dphi -= 2 * M_PI;
    while (dphi <= -M_PI) dphi += 2 * M_PI;
    return sqrt(deta * deta + dphi * dphi);
}

int main() {
    // Create Pythia instance
    Pythia8::Pythia pythia_g;
    Pythia8::Pythia pythia_q;
    int event_size = 1000000;

    // Set up Pythia configuration
    pythia_g.readString("Beams:eCM = 13000."); // Center-of-mass energy
```

```
    pythia_g.readString("PhaseSpace:pTHatMin = 20.");
    pythia_g.readString("Tune:pp = 14");
    pythia_g.readString("PartonLevel:ISR = on");  // Enable initial-state radiation

    pythia_q.readString("Beams:eCM = 13000."); // Center-of-mass energy
    pythia_q.readString("PhaseSpace:pTHatMin = 20.");
    pythia_q.readString("Tune:pp = 14");
    pythia_q.readString("PartonLevel:ISR = on");  // Enable initial-state radiation

    pythia_g.readString("HardQCD:qqbar2gg = on");  // Processes which emmit only Gluons
    pythia_g.readString("HardQCD:gg2gg = on");
    pythia_g.readString("HardQCD:gg2ggg = on");

    pythia_q.readString("HardQCD:gg2qqbar = on"); // Processes which emmit only Quarks
    pythia_q.readString("HardQCD:qq2qq = on");


    // Initialize Pythia
    if (!pythia_g.init()) {
        std::cerr << "Initialization failed!" << std::endl;
        return 1;
    }

    // Define jet radius
    double jet_R = 0.4;
    fastjet::Strategy               strategy = fastjet::Best;
    fastjet::RecombinationScheme    recombScheme = fastjet::E_scheme;

    // Initialize FastJet jet definition
    fastjet::JetDefinition jet_def(fastjet::antikt_algorithm, jet_R,
                                   recombScheme, strategy);

    // Initialize FastJet area definition
    fastjet::AreaDefinition area_def(fastjet::active_area_explicit_ghosts, fastjet::GhostedAreaSpec(5.0));

    // TCanvas canvas("canvas", "Number of Jets", 800, 600);
    // Open a file to write CSV data
    std::ofstream outFile("event_properties_2M.csv");

    // Write header line with column names
    outFile << "no_of_jets,lead_jet_pt,lead_jet_eta,lead_jet_phi,lead_jet_mass,lead_jet_energy,lead_jet_multiplicity,lead_jet_m/pt,avg_pt,var_pt,avg_distance,var_distance,var_phi,jet_width_1,jet_width_2,sublead_jet_pt,sublead_jet_eta,sublead_jet_phi,target" << std::endl;

    TH2F gluon_phi_eta_hist("hist_PhiVsEta", "Phi vs Eta Distribution", 100, -5, 5, 100, -3.14, 3.14);
    TH1F gluon_no_of_jets("hist_gluon_no_of_jets", "Number of Jets per Event", 50, 0, 50);
    TH1F gluon_lead_jet_pt("hist_gluon_lead_jet_pt", "Leading Jet pT", 200, 0, 100); //change
```

```cpp
    TH1F gluon_lead_jet_energy("hist_gluon_lead_jet_energy", "Leading Jet Energy", 200, 0, 500);
    TH1F gluon_lead_jet_multiplicity("hist_gluon_lead_jet_multiplicity", "Leading Jet Multiplicity", 100, 0, 100); //change
    TH1F gluon_lead_jet_mass_over_pt("hist_gluon_lead_jet_mass_over_pt", "Leading Jet Mass/pT", 200, 0, 2); //change
    TH1F gluon_avg_pt("hist_gluon_avg_pt", "Average pT of Jets", 200, 0, 20);
    TH1F gluon_var_pt("hist_gluon_var_pt", "Variance of pT of Jets", 200, 0, 100);
    TH1F gluon_avg_distance("hist_gluon_avg_distance", "Average Distance of Jets", 200, 0, 1);
    TH1F gluon_jet_width_1("hist_gluon_jet_width_1", "Jet Width 1 of Jets", 200, 0, 1);  // chnage
    TH1F gluon_sublead_jet_pt("hist_gluon_sublead_jet_pt", "Subleading Jet pT", 200, 0, 100); //change


    // Loop over events
    for (int iEvent = 0; iEvent < event_size; ++iEvent) {
        // Generate event
        if (!pythia_g.next()) continue;

        // Convert Pythia particles to FastJet PseudoJets
        std::vector<fastjet::PseudoJet> particles;
        for (int i = 0; i < pythia_g.event.size(); ++i) {
            if (pythia_g.event[i].isFinal() && pythia_g.event[i].isVisible()) {
                particles.push_back(fastjet::PseudoJet(pythia_g.event[i].px(), pythia_g.event[i].py(), pythia_g.event[i].pz(), pythia_g.event[i].e()));
                gluon_phi_eta_hist.Fill(pythia_g.event[i].eta(), pythia_g.event[i].phi());
            }
        }

        // Cluster particles into jets using FastJet
        fastjet::ClusterSequence cs(particles, jet_def);
        std::vector<fastjet::PseudoJet> jets = fastjet::sorted_by_pt(cs.inclusive_jets(5));
        // Process each jet

        if (!jets.empty()) {
            double lead_jet_pt = jets[0].pt();
            double lead_jet_eta = jets[0].eta();
            double lead_jet_phi = jets[0].phi();
            double lead_jet_mass = jets[0].m();
            double lead_jet_energy = jets[0].e();
            double lead_jet_multiplicity = jets[0].constituents().size();
            double lead_jet_mass_over_pt = lead_jet_mass / lead_jet_pt;

            // Calculate average pt and variance
            double avg_pt = 0.0;
            double var_pt = 0.0;
            double avg_distance = 0.0;
            double var_distance = 0.0;
```

```
            double sum_weighted_pt_1 = 0.0;
            double sum_weighted_pt_2 = 0.0;
            double sum_phi_sq = 0.0;
            double sum_phi = 0.0;

            for (const auto& constituent : jets[0].constituents()) {
                double pt = constituent.pt();
                avg_pt += pt;
                var_pt += pt * pt;
                double distance = euclideanDistance(lead_jet_eta, lead_jet_phi, constituent.eta(), constituent.phi());
                avg_distance += distance;
                var_distance += distance * distance;
                sum_weighted_pt_1 += pt * distance;
                sum_weighted_pt_2 += pt * std::sqrt(distance);
                sum_phi += constituent.phi();
                sum_phi_sq += pow(constituent.phi(),2);
            }
            avg_pt /= lead_jet_multiplicity;
            var_pt = var_pt / lead_jet_multiplicity - avg_pt * avg_pt;
            avg_distance /= lead_jet_multiplicity;
            var_distance = var_distance / lead_jet_multiplicity - avg_distance * avg_distance;
            double jet_width_1 = sum_weighted_pt_1 / (avg_pt*lead_jet_multiplicity);
            double jet_width_2 = sum_weighted_pt_2 / (avg_pt*lead_jet_multiplicity);
            double var_phi = sum_phi_sq / lead_jet_multiplicity - pow(sum_phi / lead_jet_multiplicity, 2);

            // Calculate properties for subleading jet
            double sublead_jet_pt = 0.0;
            double sublead_jet_eta = 0.0;
            double sublead_jet_phi = 0.0;
            if (jets.size() > 1) {
                sublead_jet_pt = jets[1].pt();
                sublead_jet_eta = jets[1].eta();
                sublead_jet_phi = jets[1].phi();
            }

            // Write properties to CSV file
            outFile << jets.size() << ","
                    << lead_jet_pt << ","
                    << lead_jet_eta << ","
                    << lead_jet_phi << ","
                    << lead_jet_mass << ","
                    << lead_jet_energy << ","
                    << lead_jet_multiplicity << ","
                    << lead_jet_mass_over_pt << ","
                    << avg_pt << ","
                    << var_pt << ","
                    << avg_distance << ","
                    << var_distance << ","
                    << var_phi << ","
                    << jet_width_1 << ","
                    << jet_width_2 << ","
                    << sublead_jet_pt << ","
                    << sublead_jet_eta << ","
```

```
                 << sublead_jet_phi << ","
                 << 1 << std::endl;

                 gluon_no_of_jets.Fill(jets.size());
                 gluon_lead_jet_pt.Fill(lead_jet_pt);
                 gluon_lead_jet_energy.Fill(lead_jet_energy);
                 gluon_lead_jet_multiplicity.Fill(lead_jet_multiplicity);
                 gluon_lead_jet_mass_over_pt.Fill(lead_jet_mass_over_pt);
                 gluon_avg_pt.Fill(avg_pt);
                 gluon_var_pt.Fill(var_pt);
                 gluon_avg_distance.Fill(avg_distance);
                 gluon_jet_width_1.Fill(jet_width_1);
                 gluon_sublead_jet_pt.Fill(sublead_jet_pt);

     } else {
         // If no jets found, write placeholder values to CSV file
         outFile << "0,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,1" << std::endl;
     }


 }

 std::ofstream statsFile("pythia_g_stats.txt");

 // Redirect the output of pythia.stat() to the file stream
 std::streambuf* orig_cout = std::cout.rdbuf();  // Save original cout buffer
 std::cout.rdbuf(statsFile.rdbuf());             // Redirect cout to the file

 // Call pythia.stat() to print statistics
 pythia_g.stat();

 // Restore the original cout buffer
 std::cout.rdbuf(orig_cout);

 // Close the file
 statsFile.close();


 if (!pythia_q.init()) {
     std::cerr << "Initialization failed!" << std::endl;
     return 1;
 }

 TH2F quark_phi_eta_hist("hist_PhiVsEta", "Phi vs Eta Distribution", 100, -5, 5, 100, -3.14, 3.14);
 TH1F quark_no_of_jets("hist_quark_no_of_jets", "Number of Jets per Event", 50, 0, 50);
 TH1F quark_lead_jet_pt("hist_quark_lead_jet_pt", "Leading Jet pT", 200, 0, 100); //change
 TH1F quark_lead_jet_energy("hist_quark_lead_jet_energy", "Leading Jet Energy", 200, 0, 500);
 TH1F quark_lead_jet_multiplicity("hist_quark_lead_jet_multiplicity", "Leading Jet Multiplicity", 100, 0, 100); //change
```

```
    TH1F quark_lead_jet_mass_over_pt("hist_quark_lead_jet_mass_over_pt", "Leading Jet Mass/pT", 200, 0, 2); //change
    TH1F quark_avg_pt("hist_quark_avg_pt", "Average pT of Jets", 200, 0, 20);
    TH1F quark_var_pt("hist_quark_var_pt", "Variance of pT of Jets", 200, 0, 100);
    TH1F quark_avg_distance("hist_quark_avg_distance", "Average Distance of Jets", 200, 0, 1);
    TH1F quark_jet_width_1("hist_quark_jet_width_1", "Jet Width 1 of Jets", 200, 0, 1);  // change
    TH1F quark_sublead_jet_pt("hist_quark_sublead_jet_pt", "Subleading Jet pT", 200, 0, 100); // change


    // Loop over events
    for (int iEvent = 0; iEvent < event_size; ++iEvent) {
        // Generate event
        if (!pythia_q.next()) continue;

        // Convert Pythia particles to FastJet PseudoJets
        std::vector<fastjet::PseudoJet> particles;
        for (int i = 0; i < pythia_q.event.size(); ++i) {
            if (pythia_q.event[i].isFinal() && pythia_q.event[i].isVisible()) {
                particles.push_back(fastjet::PseudoJet(pythia_q.event[i].px(), pythia_q.event[i].py(), pythia_q.event[i].pz(), pythia_q.event[i].e()));
                quark_phi_eta_hist.Fill(pythia_q.event[i].eta(), pythia_q.event[i].phi());
            }
        }

        // Cluster particles into jets using FastJet
        fastjet::ClusterSequence cs(particles, jet_def);
        std::vector<fastjet::PseudoJet> jets = fastjet::sorted_by_pt(cs.inclusive_jets(5));
        // Process each jet


        if (!jets.empty()) {
            double lead_jet_pt = jets[0].pt();
            double lead_jet_eta = jets[0].eta();
            double lead_jet_phi = jets[0].phi();
            double lead_jet_mass = jets[0].m();
            double lead_jet_energy = jets[0].e();
            double lead_jet_multiplicity = jets[0].constituents().size();
            double lead_jet_mass_over_pt = lead_jet_mass / lead_jet_pt;

            // Calculate average pt and variance
            double avg_pt = 0.0;
            double var_pt = 0.0;
            double avg_distance = 0.0;
            double var_distance = 0.0;
            double sum_weighted_pt_1 = 0.0;
            double sum_weighted_pt_2 = 0.0;
            double sum_phi_sq = 0.0;
            double sum_phi = 0.0;
```

```

            for (const auto& constituent : jets[0].constituents()) {
                double pt = constituent.pt();
                avg_pt += pt;
                var_pt += pt * pt;
                double distance = euclideanDistance(lead_jet_eta, lead_jet_phi, constituent.eta(), constituent.phi());
                avg_distance += distance;
                var_distance += distance * distance;
                sum_weighted_pt_1 += pt * distance;
                sum_weighted_pt_2 += pt * std::sqrt(distance);
                sum_phi += constituent.phi();
                sum_phi_sq += pow(constituent.phi(),2);
            }
            avg_pt /= lead_jet_multiplicity;
            var_pt = var_pt / lead_jet_multiplicity - avg_pt * avg_pt;
            avg_distance /= lead_jet_multiplicity;
            var_distance = var_distance / lead_jet_multiplicity - avg_distance * avg_distance;
            double jet_width_1 = sum_weighted_pt_1 / (avg_pt*lead_jet_multiplicity);
            double jet_width_2 = sum_weighted_pt_2 / (avg_pt*lead_jet_multiplicity);
            double var_phi = sum_phi_sq / lead_jet_multiplicity - pow(sum_phi / lead_jet_multiplicity, 2);

            // Calculate properties for subleading jet
            double sublead_jet_pt = 0.0;
            double sublead_jet_eta = 0.0;
            double sublead_jet_phi = 0.0;
            if (jets.size() > 1) {
                sublead_jet_pt = jets[1].pt();
                sublead_jet_eta = jets[1].eta();
                sublead_jet_phi = jets[1].phi();
            }

            // Write properties to CSV file
            outFile << jets.size() << ","
                    << lead_jet_pt << ","
                    << lead_jet_eta << ","
                    << lead_jet_phi << ","
                    << lead_jet_mass << ","
                    << lead_jet_energy << ","
                    << lead_jet_multiplicity << ","
                    << lead_jet_mass_over_pt << ","
                    << avg_pt << ","
                    << var_pt << ","
                    << avg_distance << ","
                    << var_distance << ","
                    << var_phi << ","
                    << jet_width_1 << ","
                    << jet_width_2 << ","
                    << sublead_jet_pt << ","
                    << sublead_jet_eta << ","
                    << sublead_jet_phi << ","
                    << 0 << std::endl;

                    quark_no_of_jets.Fill(jets.size());
```

```
                    quark_lead_jet_pt.Fill(lead_jet_pt);
                    quark_lead_jet_energy.Fill(lead_jet_energy);
                    quark_lead_jet_multiplicity.Fill(
lead_jet_multiplicity);
                    quark_lead_jet_mass_over_pt.Fill(
lead_jet_mass_over_pt);
                    quark_avg_pt.Fill(avg_pt);
                    quark_var_pt.Fill(var_pt);
                    quark_avg_distance.Fill(avg_distance);
                    quark_jet_width_1.Fill(jet_width_1);
                    quark_sublead_jet_pt.Fill(sublead_jet_pt);


      } else {
          // If no jets found, write placeholder values to CSV
file
          outFile << "0,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A,N/A
,N/A,N/A,N/A,N/A,N/A,N/A,N/A,0" << std::endl;
      }



 }

 std::ofstream statsFile_q("pythia_q_stats.txt");

 // Redirect the output of pythia.stat() to the file stream
 std::streambuf* orig_cout_q = std::cout.rdbuf();  // Save
original cout buffer
 std::cout.rdbuf(statsFile_q.rdbuf());             // Redirect
 cout to the file

 // Call pythia.stat() to print statistics
 pythia_q.stat();

 // Restore the original cout buffer
 std::cout.rdbuf(orig_cout_q);

 // Close the file
 statsFile.close();

 outFile.close();

 // Save histograms
 plotOverlayHistograms(gluon_no_of_jets, quark_no_of_jets, "
Gluon", "Quark", "no_of_jets_overlay.root");
 plotOverlayHistograms(gluon_lead_jet_pt, quark_lead_jet_pt, "
Gluon", "Quark", "lead_jet_pt_overlay.root"); //
 plotOverlayHistograms(gluon_lead_jet_energy,
quark_lead_jet_energy, "Gluon", "Quark", "
lead_jet_energy_overlay.root");
 plotOverlayHistograms(gluon_lead_jet_multiplicity,
quark_lead_jet_multiplicity, "Gluon", "Quark", "
lead_jet_multiplicity_overlay.root"); //
 plotOverlayHistograms(gluon_lead_jet_mass_over_pt,
quark_lead_jet_mass_over_pt, "Gluon", "Quark", "
lead_jet_mass_over_pt_overlay.root"); //
 plotOverlayHistograms(gluon_avg_pt, quark_avg_pt, "Gluon", "
Quark", "avg_pt_overlay.root");
```

```
    plotOverlayHistograms(gluon_var_pt, quark_var_pt, "Gluon", "Quark", "var_pt_overlay.root");
    plotOverlayHistograms(gluon_avg_distance, quark_avg_distance, "Gluon", "Quark", "avg_distance_overlay.root");
    plotOverlayHistograms(gluon_jet_width_1, quark_jet_width_1, "Gluon", "Quark", "jet_width_1_overlay.root"); //
    plotOverlayHistograms(gluon_sublead_jet_pt, quark_sublead_jet_pt, "Gluon", "Quark", "sublead_jet_pt_overlay.root"); //

    saveHistogram2DAsROOT(quark_phi_eta_hist, "quark_phi_eta_hist.root");
    saveHistogram2DAsROOT(gluon_phi_eta_hist, "gluon_phi_eta_hist.root");


    return 0;
}
```