ProjectGroupBNeutrino

May 17, 2024

0.1 Introduction

0.1.1 Project Overview

This project focuses on neutrino interactions and classifications within the context of data generated by the NOvA experiment. Neutrinos, fundamental particles with near-zero mass and neutral charge, interact via weak processes, classified as either charged-current (CC) or neutral-current (NC) interactions. The NOvA experiment, provides a unique dataset for analysis, predominantly comprising muon neutrinos and offers a comprehensive view of neutrino interactions within a broad spectrum of energies.

0.1.2 Objectives

The primary objective of this mini-project is to develop and refine a machine learning classifier capable of distinguishing between ν_μ charged-current events and other interaction types. We aim to: - Classify neutrino interaction events based on the presence of µ charged-current interactions. - Investigate the classifier's efficiency across different interaction types and energy levels.

0.2 Environment Setup

The computational environment is prepared with necessary Python libraries such as NumPy, Pandas, Matplotlib, and Scikit-learn, essential for data manipulation, visualization, and machine learning modelling. Additionally, TensorFlow is used for Neural Network architecture.

[1]:
```
# Data manipulation and numerical analysis
####################################################################################
import numpy as np
import pandas as pd




# File and URL handling
####################################################################################
import h5py                                                                        ␣
 ↪         # Interface for HDF5 binary data format
import urllib.request                                                              ␣
 ↪         # opening and reading URLs
import os                                                                          ␣
 ↪         # operating system dependent functionality
```

```python
# Machine learning and neural network frameworks
#####################################################################################
import tensorflow as tf
from tensorflow import keras
from keras.models import Model
from keras import layers          # required blocks for NNs
from keras.layers import Input, Conv2D, BatchNormalization, Flatten, Activation, Add, GlobalAveragePooling2D, Dense, MaxPooling2D, concatenate, Dropout
from tensorflow.keras.models import load_model          # Loads saved models
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau # Effective model fitting



# Visualization
#####################################################################################
import matplotlib.pyplot as plt
import seaborn as sns           # data visualization based on matplotlib



# Utility and helper libraries
#####################################################################################
from numpy import loadtxt
from alive_progress import alive_bar          # progress bar
import time          # time-related functions
import random
from collections import Counter          # stores elements as dict. keys, counts as values
from keras import callbacks          # model training utillity
from sklearn.metrics import accuracy_score          # model evaluation utillity
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
```

```
from sklearn.metrics import cohen_kappa_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import confusion_matrix
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel
Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)
instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel
Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)
instructions.
```

```
2024-05-17 15:13:58.727147: I tensorflow/core/platform/cpu_feature_guard.cc:193]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  SSE4.1 SSE4.2
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
```

0.3 Classes

[2]:
```
import enum
class Interaction(enum.Enum):

    kNumuQE =0           # Numu CC QE interaction
    kNumuRes =1           # Numu CC Resonant interaction
    kNumuDIS = 2          # Numu CC DIS interaction
    kNumuOther = 3        # Numu CC, other than above

    kNueQE = 4            # Nue CC QE interaction
    kNueRes = 5           # Nue CC Resonant interaction
    kNueDIS = 6           # Nue CC DIS interaction
    kNueOther = 7         # Nue CC, other than above
    kNutauQE = 8          # Nutau CC QE interaction
    kNutauRes = 9         # Nutau CC Resonant interaction
    kNutauDIS =10         # Nutau CC DIS interaction
    kNutauOther =11       # Nutau CC, other than above
    kNuElectronElastic = 12# NC Nu On E Scattering
    kNC =13                # NC interaction
    kCosmic =14           # Cosmic ray background
    kOther =15            # Something else.  Tau?  Hopefully we don't use this
    kNIntType=16          # Number of interaction types, used like a vector size


class FinalState(enum.Enum):

    kNumu0tr0sh=0          # Numu CC - no track no shower
```

```
    kNumu0tr1sh=1           # Numu CC - no track  1 shower
    kNumu0tr2sh=enum.auto()          # Numu CC - no track  2 shower
    kNumu0trMsh=enum.auto()          # Numu CC - no track 3+ shower
    kNumu1tr0sh=enum.auto()          # Numu CC -  1 track no shower
    kNumu1tr1sh=enum.auto()          # Numu CC -  1 track  1 shower
    kNumu1tr2sh=enum.auto()          # Numu CC -  1 track  2 shower
    kNumu1trMsh=enum.auto()          # Numu CC -  1 track 3+ shower
    kNumu2tr0sh=enum.auto()          # Numu CC -  2 track no shower
    kNumu2tr1sh=enum.auto()          # Numu CC -  2 track  1 shower
    kNumu2tr2sh=enum.auto()          # Numu CC -  2 track  2 shower
    kNumu2trMsh=enum.auto()          # Numu CC -  2 track 3+ shower
    kNumuMtr0sh=enum.auto()          # Numu CC - 3+ track no showe
    kNumuMtr1sh=enum.auto()          # Numu CC - 3+ track  1 shower
    kNumuMtr2sh=enum.auto()          # Numu CC - 3+ track  2 showe
    kNumuMtrMsh=enum.auto()          # Numu CC - 3+ track 3+ shower

    kNue0tr0sh=enum.auto()           # Nue CC - no track no shower
    kNue0tr1sh=enum.auto()           # Nue CC - no track  1 shower
    kNue0tr2sh=enum.auto()           # Nue CC - no track  2 showe
    kNue0trMsh=enum.auto()           # Nue CC - no track 3+ shower
    kNue1tr0sh=enum.auto()           # Nue CC -  1 track no shower
    kNue1tr1sh=enum.auto()           # Nue CC -  1 track  1 shower
    kNue1tr2sh=enum.auto()           # Nue CC -  1 track  2 shower
    kNue1trMsh=enum.auto()           # Nue CC -  1 track 3+ shower
    kNue2tr0sh=enum.auto()           # Nue CC -  2 track no shower
    kNue2tr1sh=enum.auto()           # Nue CC -  2 track  1 shower
    kNue2tr2sh=enum.auto()           # Nue CC -  2 track  2 shower
    kNue2trMsh=enum.auto()           # Nue CC -  2 track 3+ shower
    kNueMtr0sh=enum.auto()           # Nue CC - 3+ track no shower
    kNueMtr1sh=enum.auto()           # Nue CC - 3+ track  1 shower
    kNueMtr2sh=enum.auto()           # Nue CC - 3+ track  2 shower
    kNueMtrMsh=enum.auto()           # Nue CC - 3+ track 3+ shower
    kNC0tr0sh=enum.auto()           # NC CC - no track no shower
    kNC0tr1sh=enum.auto()           # NC CC - no track  1 shower
    kNC0tr2sh=enum.auto()           # NC CC - no track  2 shower
    kNC0trMsh=enum.auto()           # NC CC - no track 3+ shower
    kNC1tr0sh=enum.auto()           # NC CC -  1 track no shower
    kNC1tr1sh=enum.auto()           # NC CC -  1 track  1 shower
    kNC1tr2sh=enum.auto()           # NC CC -  1 track  2 shower
    kNC1trMsh=enum.auto()           # NC CC -  1 track 3+ shower
    kNC2tr0sh=enum.auto()           # NC CC -  2 track no shower
    kNC2tr1sh=enum.auto()           # NC CC -  2 track  1 shower
    kNC2tr2sh=enum.auto()           # NC CC -  2 track  2 shower
    kNC2trMsh=enum.auto()           # NC CC -  2 track 3+ shower
    kNCMtr0sh=enum.auto()           # NC CC - 3+ track no shower
    kNCMtr1sh=enum.auto()           # NC CC - 3+ track  1 shower
    kNCMtr2sh=enum.auto()           # NC CC - 3+ track  2 shower
```

```
    kNCMtrMsh=enum.auto()            # NC CC - 3+ track 3+ shower
    kCosmicFS=enum.auto()            # Cosmic ray background
    kOtherFS=enum.auto()             # Something else.  Tau?  Hopefully we don't␣
↪use this
    kNFStType=enum.auto()             # Number of final state types, used like a␣
↪vector size
```

0.4 Functions Used

[3]:
```
#####################
#                   #
# FUNCTION getFile  #
#                   #
##########################################################################################
def getFile(file_id):

    """
    Retrieves a file based on its ID from a specific URL and saves it locally.

    This function constructs a file name using the provided file ID, downloads␣
↪the
    file from a predefined URL, saves it locally with the same file name, and␣
↪then
    opens the file using h5py for further operations.

    Args:
        file_id (int or str): The ID of the file to be retrieved. This ID is␣
↪used
                              in constructing the file name.

    Returns:
        h5py.File: An open h5py File object of the downloaded file.

    Note:
        This function prints the URL of the file to be downloaded and requires␣
↪an
        internet connection to work. The function depends on the `urllib` and␣
↪`h5py`
        modules for downloading and opening the file, respectively. Ensure these
        modules are installed and imported in your script.
    """

    name = "neutrino"+str(file_id)+".h5"
    addy = 'http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/' + name

    print(addy)
```

```
    # Copy a network object to a local file
    urllib.request.urlretrieve(addy, name)

    #Open the h5 file with h5py
    df=h5py.File(name)

    return df
##############################################################################################


####################
#                  #
# FUNCTION getEvent #
#                  #
##############################################################################################
def getEvent(df, event_id, lepenergy_bool = False, nuenergy_bool = False,␣
 ↪interaction_bool = False, finalstate_bool = False ):

    """
    Retrieves event data from an h5 file based on the specified event ID and␣
 ↪options.

    This function extracts a specific event's data from an h5 file, including␣
 ↪the
    event's image representation and, optionally, its lepton energy, neutrino␣
 ↪energy,
    interaction type, and final state information based on the boolean flags␣
 ↪provided.

    Args:
        df (h5py.File): The h5py File object containing the dataset.
        event_id (int): The ID of the event to retrieve data for.
        lepenergy_bool (bool): If True, retrieve lepton energy data. Default is␣
 ↪False.
        nuenergy_bool (bool): If True, retrieve neutrino energy data. Default␣
 ↪is False.
        interaction_bool (bool): If True, retrieve interaction type data.␣
 ↪Default is False.
        finalstate_bool (bool): If True, retrieve final state data. Default is␣
 ↪False.

    Returns:
        tuple: A tuple containing the following elements based on the provided␣
 ↪flags:
            - np.array: The event's image representation as a reshaped array.
```

```
                - np.array: Lepton energy data, if `lepenergy_bool` is True.␣
↪Empty array otherwise.
                - np.array: Neutrino energy data, if `nuenergy_bool` is True.␣
↪Empty array otherwise.
                - np.array: Interaction type data, if `interaction_bool` is True.
↪ Empty array otherwise.
                - np.array: Final state data, if `finalstate_bool` is True.␣
↪Empty array otherwise.

    Note:
        The function depends on the `numpy` and `h5py` libraries. Ensure these␣
↪libraries are
        installed and imported in your script.
    """

    event=np.array(df['cvnmap'][event_id]).reshape((2,100,80))

    lepenergy = np.array([])
    nuenergy = np.array([])
    interaction = np.array([])
    finalstate = np.array([])

    if lepenergy_bool:

        lepenergy = np.array(df['neutrino']['lepenergy'][event_id])

    if nuenergy_bool:
        nuenergy = np.array(df['neutrino']['nuenergy'][event_id])

    if interaction_bool:
        interaction = np.array(df['neutrino']['interaction'][event_id])

    if finalstate_bool:
        finalstate = np.array(df['neutrino']['finalstate'][event_id])


    return event, lepenergy, nuenergy, interaction, finalstate
##############################################################################################

#########################
#                       #
# FUNCTION getEventType #
#                       #
##############################################################################################
def getEventType(df, event_id):

    """
```

```
    Retrieves the interaction type for a specified event from an h5 file.

    Given an event ID, this function looks up the interaction type of the event
    from the provided h5 dataset and returns it as an `Interaction` object.

    Args:
        df (h5py.File): The h5py File object containing the dataset.
        event_id (int): The ID of the event whose interaction type is to be␣
↪retrieved.

    Returns:
        Interaction: The interaction type of the specified event.

    Note:
        This function assumes the existence of an `Interaction` class or␣
↪enumeration
        that can interpret interaction types from the dataset. Ensure that this␣
↪class
        or enumeration is defined and accessible in your script.
    """


    kind = Interaction(df['neutrino']['interaction'][event_id])
    return kind
###############################################################################################

##########################
#                        #
# FUNCTION plotEventPair #
#                        #
###############################################################################################
def plotEventPair(df, event):

    print(f"file id: {file_id}  ;  event id: {event_id}  ;   type:␣
↪{getEventType(df,event_id)}")
    #Plot the event
    fig, ax = plt.subplots(1,2)
    ax[0].imshow(event[0][1].T)
    ax[1].imshow(event[0][0].T)
###############################################################################################

######################
#                    #
# FUNCTION fetchData #
#                    #
###############################################################################################
def fetchData(from_file_N, to_file_N, balance = True):
```

```
    """
    Fetches and processes event data from specified file ranges for best_model␣
↪training.

    Iterates through a range of files, extracting event images and metadata,
    including lepton energy, neutrino energy, interaction type, and final state.
↪ It supports
    balancing the dataset to equalize the number of events of different types.

    Args:
        from_file_N (int): The starting file number in the range to process.
        to_file_N (int): The ending file number in the range to process,␣
↪inclusive.
        balance (bool): If True, balances the dataset by equalizing the number␣
↪of events
                        of Numu type with all others. Default is True.

    Returns:
        tuple: A tuple containing the following:
            - np.array: Combined and normalized event images for best_model␣
↪input.
            - np.array: Event types as numerical labels.
            - list: Flattened list of lepton energies.
            - list: Flattened list of neutrino energies.
            - list: Flattened list of interaction types.
            - list: Flattened list of final states.

    Note:
        This function relies on several other functions (`getFile`, `getEvent`,␣
↪`getEventType`)
        and assumes their availability in the script. It also uses NumPy,␣
↪alive_progress (for progress
        bars), Seaborn (for plotting), matplotlib, and random libraries. Ensure␣
↪these libraries are
        installed and imported. The function attempts to balance the dataset by␣
↪removing excess events
        of the more frequent type if `balance` is True. The process involves␣
↪random selection, which may
        require adjustment based on specific dataset characteristics and␣
↪desired balance.
    """


    # getting images from a specific file_id file
```

```python
    P0_imgs = [ ] # Plane 0
    P1_imgs = [ ] # Plane 1
    types = [ ]

    #metadata collection
    lepenergy = [ ]
    nuenergy = [ ]
    interaction = [ ]
    finalstate = [ ]

    for file_id in np.arange(from_file_N, to_file_N + 1):
        print(f'File {file_id}:')

        df = getFile(file_id)

        L = df['neutrino']['evt'].shape[0]

        with alive_bar(L, force_tty = True) as bar:

            for evnt in range(L):

                # GetEvent returns [event ( P1, P0 cvms), lepenergy, nuenergy,␣
↪interaction, finalstate]
                event, le, nu, inter, fin = getEvent(df, evnt,
                                                                             lepenergy_bool␣
↪= True,
                                                                             nuenergy_bool␣
↪= True,
                                                                            ␣
↪interaction_bool = True,
                                                                            ␣
↪finalstate_bool = True )

                # Data for best_model
                P1_imgs.append( event[1].T )
                P0_imgs.append( event[0].T )
                types.append( getEventType(df, evnt) )

                # Metadata
                lepenergy.append( le )
                nuenergy.append( nu )
                interaction.append( inter )
                finalstate.append( fin )

                bar()

    # Binary Classifier
```

```python
    types = types.copy()

    for i in range(len(types)):
        if "Numu" in str( types[i] ):
            types[i] = 1 # Numu event is type 1
        else:
            types[i] = 0

    # Make sure the shapes are all the same
    if not len(P0_imgs) == len(P1_imgs) == len(types) == len(lepenergy) ==␣
↪len(nuenergy) == len(interaction) == len(finalstate):
        raise Exception("Lengths are not equal")

    # Convert to arrays
    for el in P0_imgs, P1_imgs, types, lepenergy, nuenergy, interaction,␣
↪finalstate :
        el = np.array(el)

    # Check the distribution
    N_other = (types).count(0)
    N_numu = (types).count(1)

    data = [N_other, N_numu]
    keys = ['Other', 'NuMu']
    # define Seaborn color palette to use
    palette_color = sns.color_palette('dark')

    plt.figure()
    plt.pie(data, labels=keys, colors=palette_color, autopct='%.0f%%')
    plt.title('Unbalanced Distribution')

    if balance:
        # We want to select a number of random numu events that is equal to the␣
↪number of other events (N_other)
        # Generate random indeces in given range until have N_other

        numu_indices = []
        count = 0
        while count != N_numu - N_other-1:

            ind = random.randint(0, len(types)-1)


            # If not in numu_indices and if corresponds to numu event
            if ind not in numu_indices:
                if types[ind] != 0:
                    # Add to numu_indices
```

```
                    numu_indices.append(ind)
                    count+=1

        # Now we want to eradicate these indeces
        # Sort indices in descending order to avoid shifting index issue
        numu_indices.sort(reverse=True)

        for ind in numu_indices:

            P0_imgs.pop(ind)
            P1_imgs.pop(ind)
            types.pop(ind)

            lepenergy.pop(ind)
            nuenergy.pop(ind)
            interaction.pop(ind)
            finalstate.pop(ind)



        # Make sure the shapes are all the same
        if not len(P0_imgs) == len(P1_imgs) == len(types) == len(lepenergy) ==␣
↪len(nuenergy) == len(interaction) == len(finalstate):
            raise Exception("Lengths are not equal")

        # Check the distribution
        N_other = (types).count(0)
        N_numu = (types).count(1)

        # declare cvm_data
        data = [N_other, N_numu]
        keys = ['Other', 'NuMu']

        # plotting cvm_data on chart
        plt.figure()
        plt.pie(data, labels=keys, colors=palette_color, autopct='%.0f%%')
        plt.title('Balanced Distribution')

        # displaying chart
        plt.show()

    # Combine and normalise the cvm arrays
    cvm_data = [[P0_imgs[i]/255.0, P1_imgs[i]/255.0] for i in␣
↪range(len(P0_imgs))]
    cvm_data = np.array(cvm_data)

    types = np.array(types)
```

```python
    nuenergy_flat = []

    # Correct the format of metadata variables
    for i in range(len(nuenergy)):
        for j in range(len(nuenergy[i])):
            nuen = float(nuenergy[i][j])
            nuenergy_flat.append(nuen)

    interaction_flat = []

    for i in range(len(interaction)):
        for j in range(len(interaction[i])):
            nuen = float(interaction[i][j])
            interaction_flat.append(nuen)

    finalstate_flat = []

    for i in range(len(finalstate)):
        for j in range(len(finalstate[i])):
            nuen = float(finalstate[i][j])
            finalstate_flat.append(nuen)

    lepenergy_flat = []

    for i in range(len(lepenergy)):
        for j in range(len(lepenergy[i])):
            nuen = float(lepenergy[i][j])
            lepenergy_flat.append(nuen)

    print('Successfully Obtained Data for', len(types), 'Events')

    return cvm_data, types, lepenergy_flat, nuenergy_flat, interaction_flat,␣
 ↪finalstate_flat
##############################################################################################



##############################################################################################
#                                                                                           ␣
 ↪                          #
#                                FUNCTIONS FOR best_model TESTING & EVALUATION              ␣
 ↪                              #
#                                                                                           ␣
 ↪                          #
##############################################################################################
```

```
###################
#                 #
# FUNCTION binEval #
#                 #
####################################################################################
def binEval(metavar, best_model, cvm_data, types, N):


    """
    Evaluates a best_model's performance across different bins of a specified␣
↪metadata variable.

    Divides the data into bins based on quantiles of a given metadata variable
    (e.g., nuenergy, lepenergy) and evaluates the best_model's performance␣
↪(loss and accuracy) for each bin. This
    approach allows for the analysis of best_model performance across different␣
↪ranges of the metadata
    variable.

    Args:
        metavar (np.array): The metadata variable array used for binning the␣
↪data.
        best_model (best_model): The machine learning best_model to be␣
↪evaluated.
        cvm_data (np.array): The input data for the best_model, corresponding␣
↪to event images.
        types (np.array): The true labels for the data.
        N (int): The number of bins to divide the metadata variable into.

    Returns:
        tuple: A tuple containing two np.arrays:
            - loss_data: An array with midpoints of bins and corresponding loss␣
↪values.
            - acc_data: An array with midpoints of bins and corresponding␣
↪accuracy values.

    Note:
        The bins are created based on quantiles to ensure approximately equal␣
↪numbers of data points
        in each bin. The function calculates the midpoint of each bin for␣
↪plotting purposes. If a bin
        has no data points, it assigns NaN values for both loss and accuracy to␣
↪indicate an empty bin.
```

```
        This setup requires the `best_model` to have an `evaluate` method that accepts data and labels,
        returning loss and accuracy, which is common in frameworks like TensorFlow/Keras.
    """


    # Calculate quantile-based bins with approximately equal number of data points
    bins = np.quantile(metavar, np.linspace(0, 1, N + 1))

    bin_ranges = [(bins[i], bins[i + 1]) for i in range(N)]

    scores = []
    midpoints = []

    for bin_range in bin_ranges:
        indices = [i for i, el in enumerate(metavar) if bin_range[0] <= el < bin_range[1] or (el == bin_range[1] and bin_range[1] == max(bins))]

        # Check to ensure no bin is evaluated with an empty dataset
        if indices:
            cvm = cvm_data[indices]
            labels = types[indices]
            score = best_model.evaluate(cvm, labels, verbose=0)
        else:
            score = [np.nan, np.nan]  # Placeholder for empty bins, though with quantiles, this should be rare

        scores.append(score)
        midpoints.append(np.mean(bin_range))

    loss, acc = zip(*scores)  # Unpacking scores into loss and accuracy

    acc_data = np.array([midpoints, acc])
    loss_data = np.array([midpoints, loss])

    return loss_data, acc_data
###############################################################################################


##########################
#                        #
# FUNCTION EnergyBinPlot #
#                        #
###############################################################################################
def EnergyBinPlot(metavar, N_bins, loss_data, acc_data):
```

```
    """
    Plots best_model performance (loss and accuracy) across energy bins.

    Creates a visualization to display how the best_model's loss and accuracy
    change across different bins of a specified energy variable. It helps in␣
↪understanding
    how well the best_model performs across a range of energy values, which can␣
↪be crucial for
    evaluating best_model bias towards certain energy levels.

    Args:
        metavar (np.array): The metadata variable array used for binning the␣
↪data, typically
                            representing some form of energy in the context of␣
↪the data.
        N_bins (int): The number of bins to use for the histogram of the␣
↪metavar distribution.
        loss_data (np.array): An array containing the midpoints of bins and␣
↪corresponding
                              loss values for each bin.
        acc_data (np.array): An array containing the midpoints of bins and␣
↪corresponding
                             accuracy values for each bin.

    Note:
        The function uses matplotlib for plotting. Ensure that matplotlib is␣
↪installed and
        imported as plt in your script. The histogram represents the␣
↪distribution of the
        metavar across the dataset, and the dashed lines indicate the␣
↪performance metrics
        (loss and accuracy) in each bin. This visualization aids in identifying␣
↪if the best_model's
        performance is consistent across the range of the metavar or if there␣
↪are noticeable
        variations that need to be addressed.
    """


    fig, ax = plt.subplots(figsize=(10, 6))

    # Assuming `metavar` and `best_model` are defined elsewhere in the code,␣
↪and follow the same structure
    ax.hist(metavar, bins=N_bins, color='skyblue', edgecolor='black', alpha=0.7)
```

```
    for x in acc_data[0]:
        ax.axvline(x, color='green', linestyle='dashed', linewidth=1)

    # Adding a vertical line for the mean
    mean_val = np.mean(metavar)
    ax.axvline(mean_val, color='red', linestyle='dashed', linewidth=1)
    min_ylim, max_ylim = plt.ylim()
    ax.text(mean_val*1.1, max_ylim*0.9, f'Mean: {mean_val:.2f}', color = 'red')

    ax.set_title('Distribution of Values')
    ax.set_xlabel('Energy Value')
    ax.set_ylabel('Frequency')
    ax.grid(True, which='major', linestyle='--', color='grey')

    # Additional plot adjustments not included here for brevity

    plt.tight_layout()
    plt.show()
##############################################################################################

#############################
#                           #
# FUNCTION EnergyBinPerform #
#                           #
##############################################################################################
def EnergyBinPerform(metavar, N_bins, loss_data, acc_data):

    """
    Plots best_model performance (loss and accuracy) across energy bins.

    Creates a visualization to display how the best_model's loss and accuracy
    change across different bins of a specified energy variable. It helps in understanding
    how well the best_model performs across a range of energy values, which can be crucial for
    evaluating best_model bias towards certain energy levels.

    Args:
        metavar (np.array): The metadata variable array used for binning the data, typically
                                representing some form of energy in the context of the data.
        N_bins (int): The number of bins to use for the histogram of the metavar distribution.
        loss_data (np.array): An array containing the midpoints of bins and corresponding
```

```
                              loss values for each bin.
        acc_data (np.array): An array containing the midpoints of bins and␣
↪corresponding
                             accuracy values for each bin.

    Note:
        The function uses matplotlib for plotting. Ensure that matplotlib is␣
↪installed and
        imported as plt in your script. The plot demonstrates the relationship␣
↪between the
        metavar bins and the best_model's loss and accuracy, allowing for a␣
↪detailed analysis of
        performance across metavar ranges. This visualization aids in␣
↪identifying if the
        best_model's performance is consistent across the range of the metavar␣
↪or if there are
        noticeable variations that need to be addressed.
    """




    # Performance Plot
    plt.figure(figsize=(10, 6))
    plt.plot(loss_data[0], loss_data[1], '.--', color='blue', label='Loss',␣
↪linewidth=2, markersize=10)
    plt.plot(acc_data[0], acc_data[1], '.--', color='green', label='Accuracy',␣
↪linewidth=2, markersize=10)

    # Enhancing the plot
    plt.title('best_model Performance', fontsize=15)  # Assuming a general␣
↪title; adjust as needed
    plt.xlabel('Energy')
    plt.ylabel('Value')  # Adjust based on what 'loss' and 'accuracy' represent
    plt.legend()

    # Adding grid for better readability
    plt.grid(True, which='major', linestyle='--', color='grey')

    plt.tight_layout()  # Adjust the layout to make room for the labels
    plt.show()

    # Additional plot adjustments not included here for brevity

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

```python
##########################################################################################


####################
#                  #
# FUNCTION catPlot #
#                  #
##########################################################################################
def catPlot(data, catFunc):

    """
    Creates a categorical barplot for the distribution of categories for a meta␣
↪variable.

    Takes a dataset and a categorization function, applies the categorization
    function to each element in the dataset to determine its category, and then␣
↪creates a
    bar chart visualizing the distribution of these categories.

    Args:
        data (iterable): The dataset to be categorized and plotted.
        catFunc (function): A function that takes a single element of `data` as␣
↪input and
                            returns a category as output.

    Note:
        The function uses matplotlib for plotting and the `collections.Counter`␣
↪class to count
        occurrences of each category. Ensure these are installed and imported␣
↪as needed in your
        script. The categorization function's return value should be a string␣
↪or easily convertible
        to a string. The bar chart will display categories on the y-axis and␣
↪their frequencies on
        the x-axis, with categories sorted by frequency in increasing order for␣
↪clarity.

        The plot does not display x-axis tick labels directly but instead␣
↪annotates each bar with
        the category name for better readability, especially when the␣
↪categories are numerous or
        the names are long.
    """

    final_type = []
```

```
    for el in data:
        final_type.append(str(catFunc(el))[12:-2])

    # Count the occurrences of each unique string
    string_counts = Counter(final_type)

    # Sort the counts by frequency in increasing order
    sorted_counts = sorted(string_counts.items(), key=lambda x: x[1])

    fig, ax = plt.subplots(figsize=(12, 6))  # Set figure size for better␣
↪visibility

    # Data for the bar chart
    categories = [x[0] for x in sorted_counts]
    frequencies = [x[1] for x in sorted_counts]

    # Bar chart plot
    bars = ax.bar(categories, frequencies, color='skyblue', edgecolor='black',␣
↪alpha=0.7)

    # Enhancing the plot
    ax.set_title('Distribution of Categories', fontsize=15)  # Title with␣
↪increased font size
    ax.set_ylabel('Frequency', fontsize=12)  # Y-axis label with increased font␣
↪size
    ax.set_xlabel('Final States')  # X-axis label

    # Adding grid for better readability
    ax.grid(True, which='major', linestyle='--', linewidth=0.5, color='grey')
    ax.set_xticks([])  # This hides the x-axis tick labels

    # Annotate labels on top of each bar
    for bar, label in zip(bars, categories):
        height = bar.get_height()+3
        ax.text(bar.get_x() + bar.get_width() / 2., height, label,
                ha='center', va='bottom', fontsize=12, rotation=90)

    plt.tight_layout()
    plt.show()
###############################################################################################


#####################
#                   #
# FUNCTION evalPlot #
#                   #
###############################################################################################
```

```python
def evalPlot(data, catFunc):

    """
    Evaluates and plots the performance of a best_model for different␣
    ↪categories of a meta variables.

    This function segments the data into categories using a specified␣
    ↪categorization function,
    evaluates the best_model's performance (loss and accuracy) for each␣
    ↪category, and plots the results.
    It's useful for analyzing best_model performance across different segments␣
    ↪of the dataset.

    Args:
        data (iterable): The dataset to be categorized and evaluated.
        catFunc (function): A categorization function that assigns a category␣
    ↪to each element
                              in the dataset based on its characteristics.

    Note:
        The function dynamically creates variables for each category to store␣
    ↪indices, data, and
        types, which are then used for best_model evaluation. It assumes the␣
    ↪presence of a pre-trained
        best_model (`best_model`) and expects the categorization function to␣
    ↪provide unique string labels
        for each category.


        The results are visualized using a bar plot with separate bars for loss␣
    ↪and accuracy within
        each category, providing a clear overview of best_model performance␣
    ↪across different dataset
        segments. This approach helps identify categories where the best_model␣
    ↪may be underperforming,
        guiding further best_model improvements or data collection efforts.
    """


    Dic = {}
    for i in range(50):
        try: Dic[i] = str(catFunc(i))[12:-2]
        except: break


    categories = list(Dic.values())
```

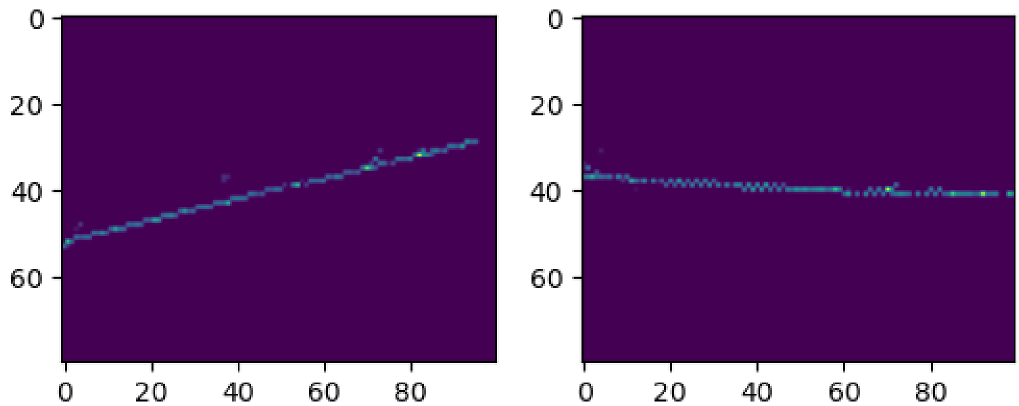
```
    for category in categories:
        globals()[f"{category}_indices"] = []
        globals()[f"{category}_cvm"] = []
        globals()[f"{category}_types"] = []

    categories.sort(key=lambda x: len(globals()[f"{x}_cvm"]))  # Sort␣
↪categories by length of cvm_sub

    data_mapping = {k: globals()[f"{v}_indices"] for k, v in Dic.items()}

    for i, el in enumerate(data):
        if el in data_mapping:
            data_mapping[el].append(i)

    for category in categories:
        indices = globals()[f"{category}_indices"]
        for i in indices:
            globals()[f"{category}_cvm"].append(cvm_data[i])
            globals()[f"{category}_types"].append(types[i])

        globals()[f"{category}_cvm"] = np.array(globals()[f"{category}_cvm"])
        globals()[f"{category}_types"] = np.
↪array(globals()[f"{category}_types"])

    evaluation_results = {}
    for category in categories:
        cvm_sub = globals()[f"{category}_cvm"]
        type_sub = globals()[f"{category}_types"]

        if len(cvm_sub) > 0:
            loss, accuracy = best_model.evaluate(cvm_sub, type_sub, verbose=0)
            evaluation_results[category] = {'loss': loss, 'accuracy': accuracy}
        else:
            print(f'Skipping Category: {category}, cvm_data is empty.')


    # Prepare data for plotting
    plot_data = []
    for category, metrics in evaluation_results.items():
        plot_data.append({'Category': category, 'Metric': 'Loss', 'Value':␣
↪metrics['loss']})
        plot_data.append({'Category': category, 'Metric': 'Accuracy', 'Value':␣
↪metrics['accuracy']}, )

    # Convert results to a DataFrame for easier plotting with seaborn
    df_plot = pd.DataFrame(plot_data)
```

```python
    # Create the plot
    plt.figure(figsize=(12, 6))  # Adjust figure size for better visibility
    ax = sns.barplot(data=df_plot, x='Category', y='Value', hue='Metric',␣
↪palette='Blues_d', edgecolor='black', alpha=0.7)

    # Join the tops of the bars with lines
    sns.pointplot(data=df_plot, x='Category', y='Value', hue='Metric',␣
↪linestyles=["-", "--"], markers=['', ''], ax=ax)

    # Enhancing the plot
    ax.set_title('best_model Performance', fontsize=15)  # Title with increased␣
↪font size
    ax.set_xlabel('Category', fontsize=12)  # X-axis label with increased font␣
↪size
    ax.set_ylabel('Value', fontsize=12)  # Y-axis label with increased font size
    ax.tick_params(axis='x', rotation=45)  # Rotate x-axis labels for better␣
↪visibility
    ax.legend()

    # Adding grid for better readability
    ax.grid(True, which='major', linestyle='--', linewidth=0.5, color='grey')

    plt.tight_layout()  # Adjust the layout to make room for the labels
    plt.show()
```

0.5 Plotting a Single Event

[4]:
```python
file_id = 1
event_id = 0

df = getFile(file_id)
plotEventPair( df, getEvent(df, event_id) )
```

```
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino1.h5
file id: 1  ;  event id: 0  ;   type: Interaction.kNumuQE
```



1 Model Setup

1.1 Getting the Data & Train/Test Split for Training

1.2 Data Loading and Preprocessing

NOvA experiment data - a mix of ν_μ, ν_e neutrinos, and their antiparticles - loaded for analysis using the pre-defined `fetchData` function.

1.3 Motivation for the fetchData Function

The `fetchData` function streamlines loading, preprocessing, and structuring data for the classification of neutrino interactions. This function:

- Iterates over a specified range of data files, extracting relevant features such as event images and metadata (e.g., lepton energy, neutrino energy, interaction type, and final state), necessary for nuanced analysis and classification tasks.
- Implements an option to balance the dataset, crucial for mitigating biases in machine learning models by ensuring equal representation of different event types, specifically by adjusting the prevalence of Numu type events relative to others.
- Prepares the dataset for model training by organizing data into structured arrays for model input and converting interaction types into numerical labels for classification purposes.

For training data, balanced data is chosen: A model trained on unbalanced data might develop a bias towards the majority class, overlooking the minority.

[5]:

```
# Do you want to reload the data for training?
# Set to False for efficiency purposes

reload = False

if reload == True:

    # Number of files we want to read
```

```
    from_file_N = 20
    to_file_N = 30

    cvm_data, types, lepenergy, nuenergy, interaction, finalstate = 
 ↪fetchData(from_file_N, to_file_N)

    # Splitting the cvm_data for train / test
    train_data, test_data, train_labels, test_labels = 
 ↪train_test_split(cvm_data, types, test_size=0.3, random_state=42)

    data_loaded = True

else: data_loaded = False
```

1.4 Model Architecture & Training

The ready model is available via UCL's OneDrive: open access link. Please save in the same file as this notebook, if would prefer to save time. Otherwise, set retrain = True and retrain the model.

The model architecture is tailored to ν_μ binary classification based on two 2D images.

1. The convolutional layers, consisting of 32 and 64 filters respectively, with 3x3 kernel sizes and 'relu' activation functions, capture low-level features such as edges and basic shapes crucial for discriminating between different events.
2. Max-pooling layers, with pool sizes of 2x2, effectively reduce dimensions, enhancing computational efficiency while extracting more robust features.
3. Dense layer with 128 neurons and 'relu' activation facilitates the learning of high-level features essential for nuanced classification tasks.
4. Dropout regularization with a rate of 0.5 mitigates overfitting, thereby improving generalization.
5. Binary output layer with a sigmoid activation function ensures the model's suitability for binary classification tasks. The model is compiled using binary cross-entropy loss and the Adam optimizer with a learning rate of 1×10^{-3} as per best practices for binary classification.

[6]:
```
shape = (2, 80, 100)  # Channels, Height, Width for 2 different plane images

model = keras.Sequential([

    # Input
    layers.Reshape((80, 100, 2), input_shape=shape),

    # A convolutional layer with 32 filters, each with a kernel size of 3x3, 
 ↪using 'relu'
    # Learn to capture low-level features eg edges and basic shapes
    layers.Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),

    # MaxPooling reduces dimensions reducing parameters and computation, 
 ↪extracting more robust features
```

```python
    layers.MaxPooling2D(pool_size=(2, 2)),

    # convolutional layer with 64 filters to capture more complex details
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),

    # MaxPooling reduces dimensions reducing parameters and computation,␣
↪extracting more robust features
    layers.MaxPooling2D(pool_size=(2, 2)),

    # Flatten the output for Dense
    layers.Flatten(),

    # Dense with 128 neurons and relu for high-level features
    layers.Dense(128, activation='relu'),

    # Dropout to prevent overfitting
    layers.Dropout(0.5),

    # Binary output with sigmoid
    layers.Dense(1, activation='sigmoid')
])

# Compile with binary crossentropy & adam (lr of 1e-3) for binary classification
model.compile(loss='binary_crossentropy',
              optimizer=keras.optimizers.Adam(learning_rate=1e-3),
              metrics=['accuracy'])
```

If `retrain` is set to `True` and the data has been successfully loaded, the model will undergo training. If `retrain` is set to `True` but the data has not been loaded in the code above, a respective exception will be raised.

1. Early stopping is implemented to prevent overfitting by monitoring the validation loss. Training halts if the validation loss fails to decrease for a specified number of epochs (patience) to restore the model to its best state.
2. Model checkpointing is employed to save the best-performing model based on validation accuracy during training. This ensures that the model with the highest validation accuracy is retained for further evaluation and potential deployment.
3. Learning rate reduction strategy is adopted using the ReduceLROnPlateau callback. This dynamically adjusts the learning rate when the validation loss reaches a plateau, thereby facilitating convergence towards an optimal solution.

The training process itself involves fitting the model to the training data for 100 epochs. The `callbacks` parameter integrates the aforementioned early stopping, model checkpointing, and learning rate reduction mechanisms into the training pipeline.

Upon completion of training, the trained model is saved to disk, and its performance is evaluated using the test data.

[7]:
```
# Do you want to retrain the model?
# Set to False for efficiency purposes

retrain = False

if retrain == True and data_loaded:

    # Train the model
    early_stopping = EarlyStopping(monitor='val_loss', patience=10, verbose=1,␣
 ↪mode='min', restore_best_weights=True)
    model_checkpoint = ModelCheckpoint('best_model.h5', monitor='val_accuracy',␣
 ↪save_best_only=True, mode='max', verbose=1)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5,␣
 ↪min_lr=1e-5, mode='min', verbose=1)

    model.fit(train_data, train_labels,
              epochs=100,
              batch_size=64,
              validation_split=0.2,
              callbacks=[early_stopping, model_checkpoint, reduce_lr],
              shuffle=True)


    if retrain == True and data_loaded:

        # Setup for early stopping to prevent overfitting. Stops training when␣
 ↪accuracy stagnates
        early_stopping = EarlyStopping(monitor='val_loss', patience=10,␣
 ↪verbose=1, mode='min', restore_best_weights=True)

        # saves model after every epoch where accuracy has improves
        model_checkpoint = ModelCheckpoint('best_model.h5',␣
 ↪monitor='val_accuracy', save_best_only=True, mode='max', verbose=1)

        # ReduceLROnPlateau reduces learning rate (le) when a accuracy stagnates
        reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,␣
 ↪patience=5, min_lr=1e-5, mode='min', verbose=1)

        model.fit(train_data, train_labels,
                  epochs=100,
                  batch_size=64,
                  validation_split=0.2,
                  callbacks=[early_stopping, model_checkpoint, reduce_lr],
                  shuffle=True)

    print("Saved model to disk")
```

```
    scores = model.evaluate(test_data, test_labels, verbose=2)

elif retrain == True and not data_loaded:

    raise Exception('Please return to "Getting the Data & Train/Test Split for␣
 ↪Training" and load the data')
```

1.5 Best Model Retrieval

[8]:
```
# load model
best_model = load_model('best_model.h5')

# summarize model
best_model.summary()
```

```
Model: "sequential_6"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 reshape (Reshape)           (None, 80, 100, 2)        0

 conv2d_8 (Conv2D)           (None, 80, 100, 32)       608

 max_pooling2d_8 (MaxPooling  (None, 40, 50, 32)       0
 2D)

 conv2d_9 (Conv2D)           (None, 40, 50, 64)        18496

 max_pooling2d_9 (MaxPooling  (None, 20, 25, 64)       0
 2D)

 flatten_6 (Flatten)         (None, 32000)             0

 dense_12 (Dense)            (None, 128)               4096128

 dropout_4 (Dropout)         (None, 128)               0

 dense_13 (Dense)            (None, 1)                 129

=================================================================
Total params: 4,115,361
Trainable params: 4,115,361
Non-trainable params: 0
_________________________________________________________________
```

2 Model Testing: investigating how efficiency depends on the meta cvm_data variables below

Label	Description
neutrino/nuenergy	Neutrino Energy (GeV)
neutrino/lepenergy	Lepton Energy (GeV)
neutrino/interaction	Interaction
neutrino/finalstate	Final State

2.0.1 Below, we will focus on both the balanced and unbalanced data

Reflection of Real-world Conditions

- **Unbalanced Data:** Evaluating a model on unbalanced data reflects its expected performance in practical applications, providing insights into how it might perform when deployed. Many real-world datasets inherently exhibit imbalance in their class distributions.
- **Balanced Data:** Evaluating on balanced data allows the assessment of model's ability to learn from each class equally, without bias towards the majority class. This is particularly useful for understanding the model's capabilities in an ideal scenario where each outcome has equal representation.

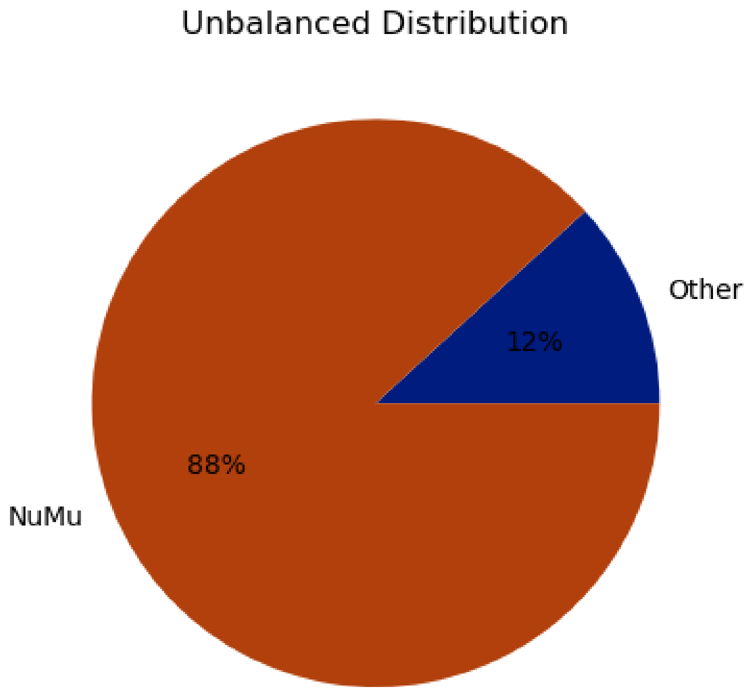
2.1 Using Balanced Data

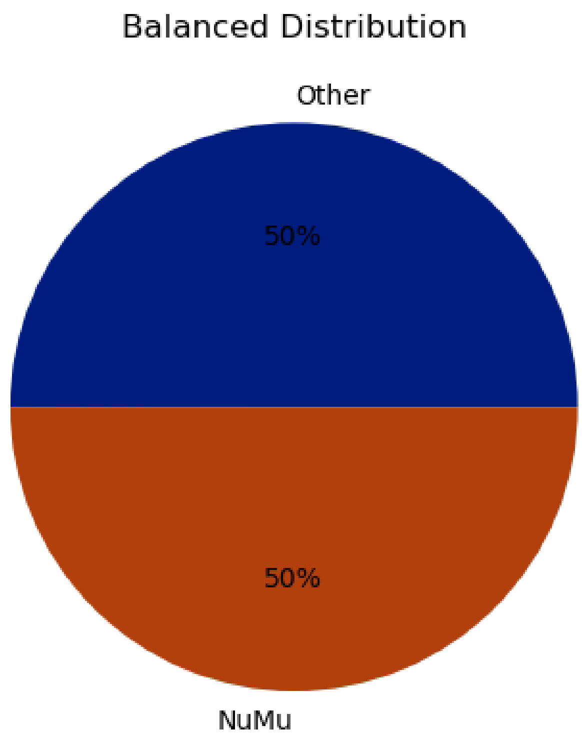
(50% Numu, 50% Other)

[9]:
```
# Get new data for evaluation, from a different set of files
from_file_N, to_file_N = 31,36
cvm_data, types, lepenergy, nuenergy, interaction, finalstate =␣
↪fetchData(from_file_N, to_file_N)
```

```
File 31:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino31.h5
|                         | 6980/6980 [100%] in 46.1s (151.67/s)
File 32:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino32.h5
|                         | 6978/6978 [100%] in 46.3s (150.80/s)
File 33:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino33.h5
|                         | 6928/6928 [100%] in 45.8s (151.44/s)
File 34:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino34.h5
|                         | 6819/6819 [100%] in 44.9s (151.92/s)
File 35:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino35.h5
|                         | 6845/6845 [100%] in 45.2s (151.61/s)
File 36:
```

```
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino36.h5
|                        | 7073/7073 [100%] in 46.8s (151.21/s)
```






```
Successfully Obtained Data for 9827 Events
```

2.1.1 Overal l Performace

[10]:
```
scores = best_model.evaluate(cvm_data, types, verbose=2)
```

```
308/308 - 21s - loss: 0.4785 - accuracy: 0.8073 - 21s/epoch - 68ms/step
```

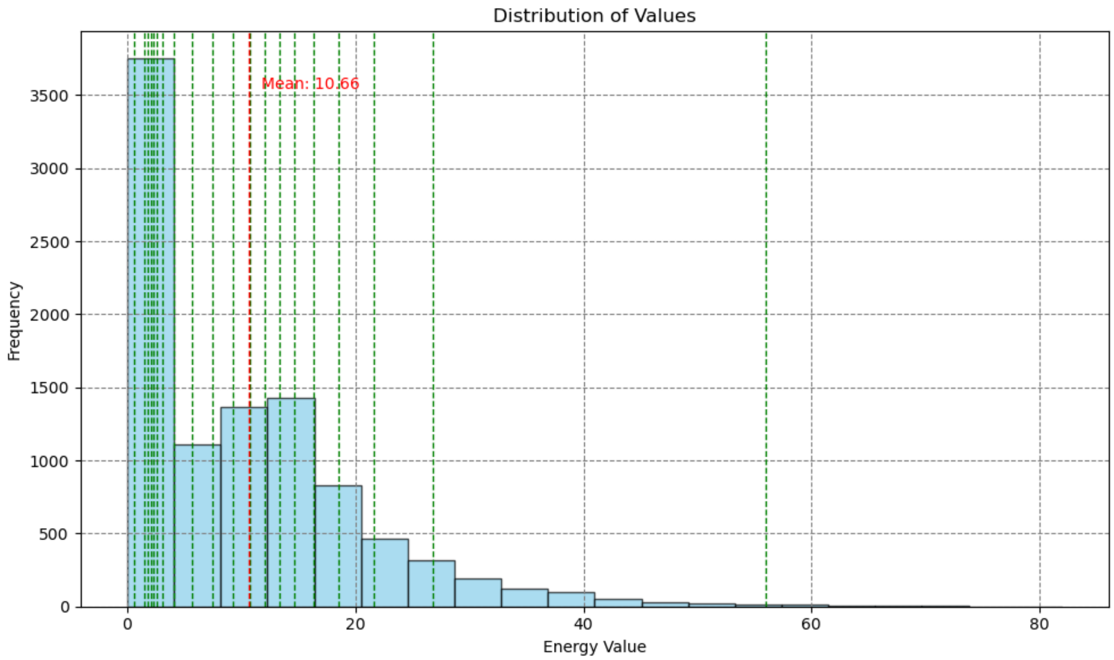
[11]:
```
pred = best_model.predict(cvm_data)
# Now calculate the F1 score
pred_binary = (pred > 0.5).astype(int)
f1 = f1_score(types, pred_binary)
f1
```

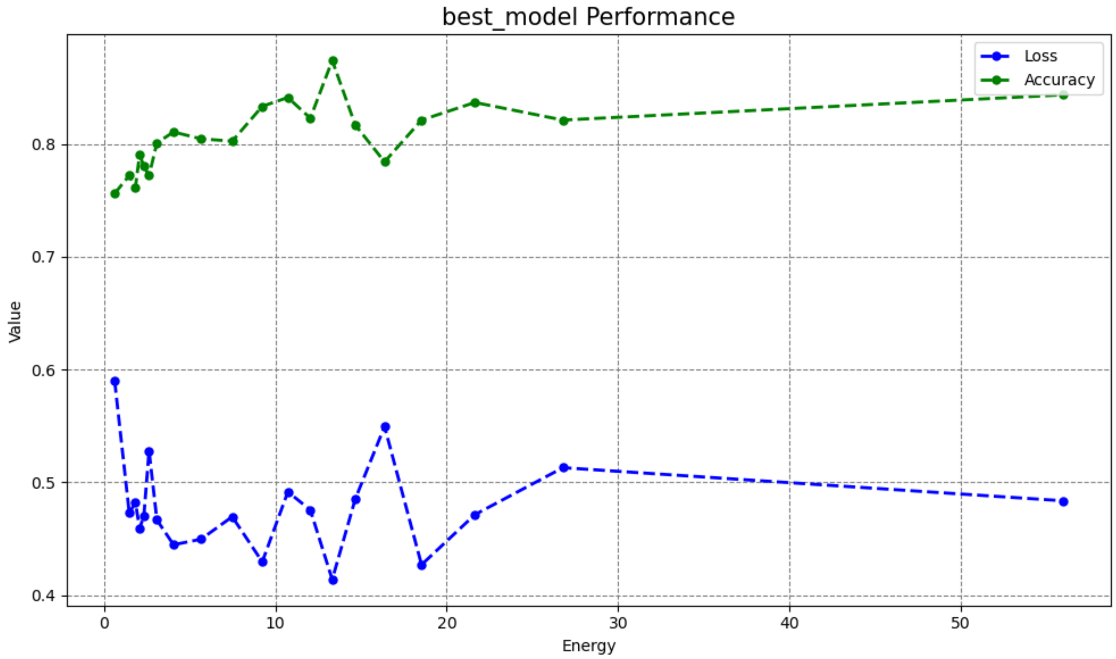
```
308/308 [==============================] - 22s 72ms/step
```

[11]: 0.7923700942775707

2.1.2 Neutrino Energy

[12]:
```
N_bins = 20
metavar = nuenergy

# Evaluating bins now to plot energy value bin boundaries
loss_data, acc_data = binEval(metavar, best_model, cvm_data, types, N_bins)

EnergyBinPlot(metavar, N_bins, loss_data, acc_data)
EnergyBinPerform(metavar, N_bins, loss_data, acc_data)
```

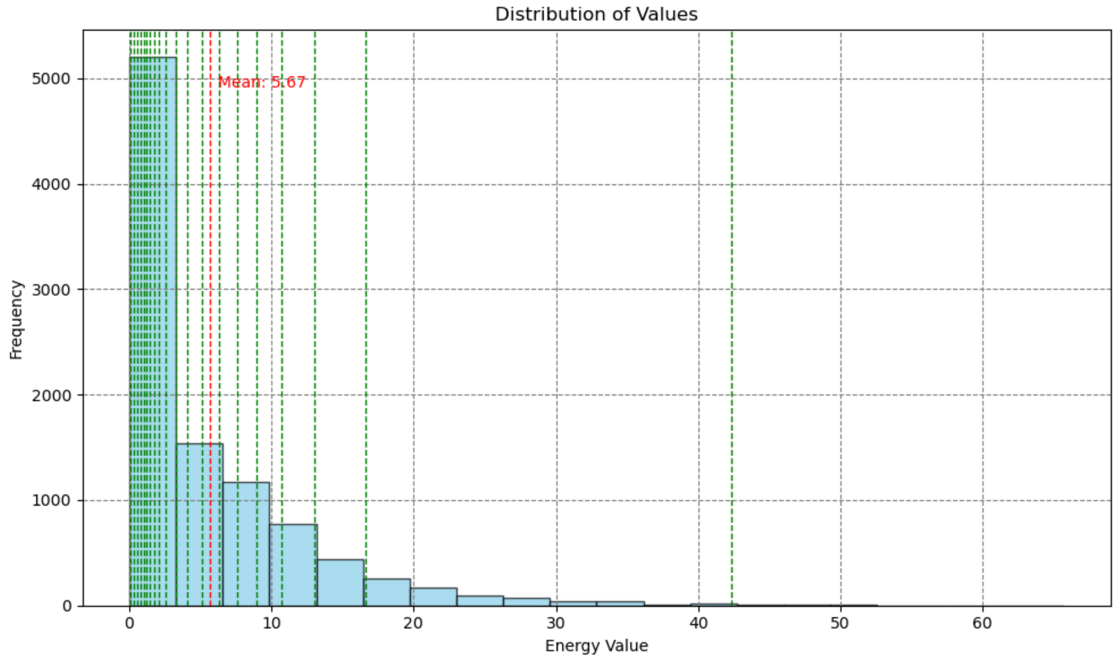


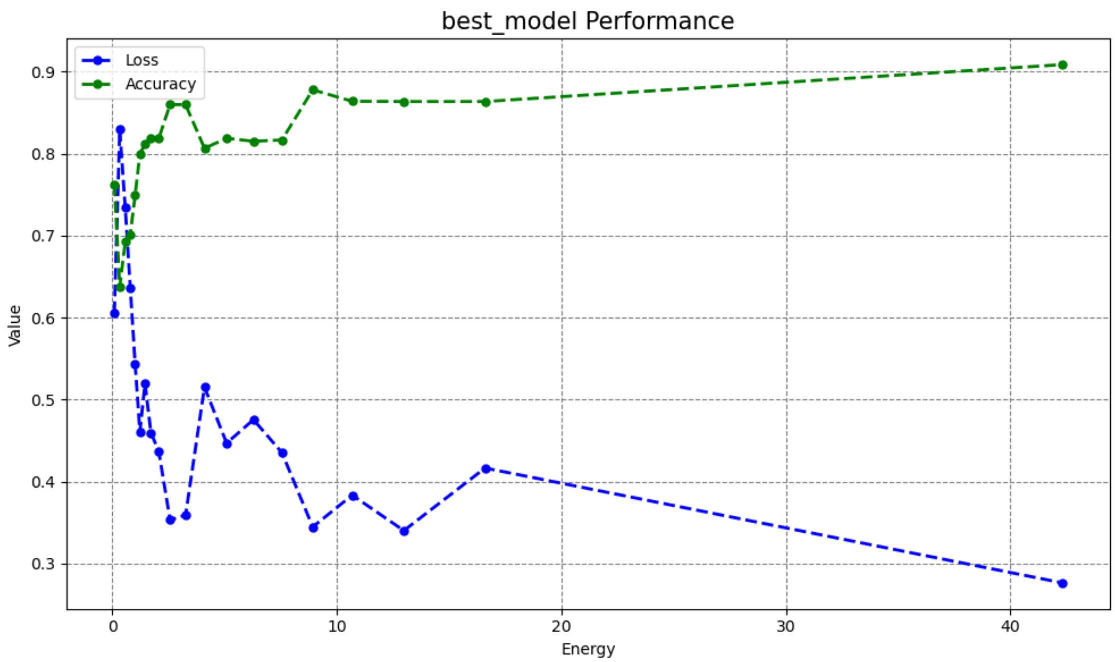



```
<Figure size 640x480 with 0 Axes>
```

2.1.3 Lepton Energy

[13]:
```python
N_bins = 20
metavar = lepenergy

# Evaluating bins now to plot energy value bin boundaries
loss_data, acc_data = binEval(metavar, best_model, cvm_data, types, N_bins)

EnergyBinPlot(metavar, N_bins, loss_data, acc_data)
EnergyBinPerform(metavar, N_bins, loss_data, acc_data)
```

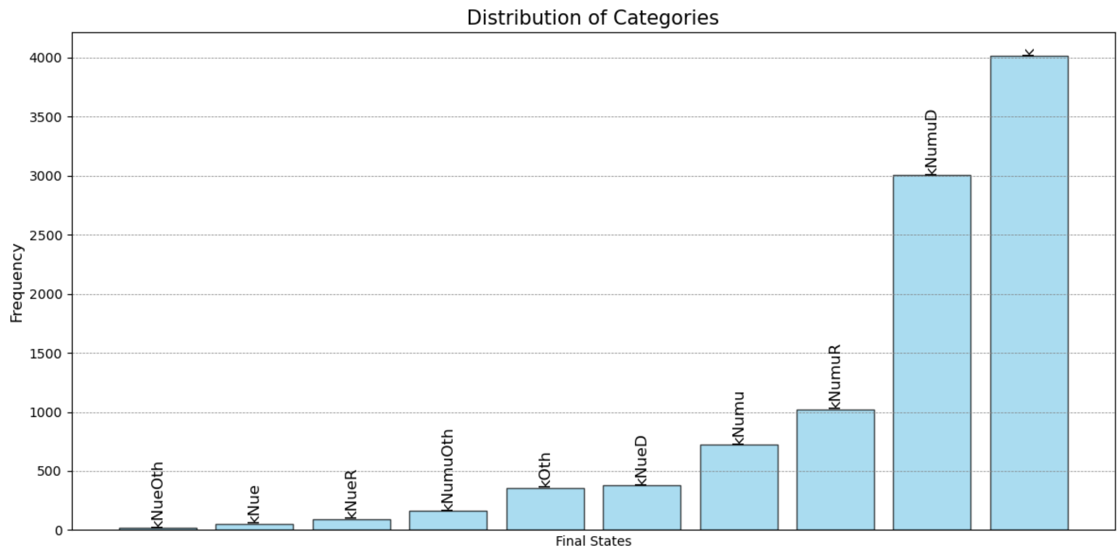





```
<Figure size 640x480 with 0 Axes>
```

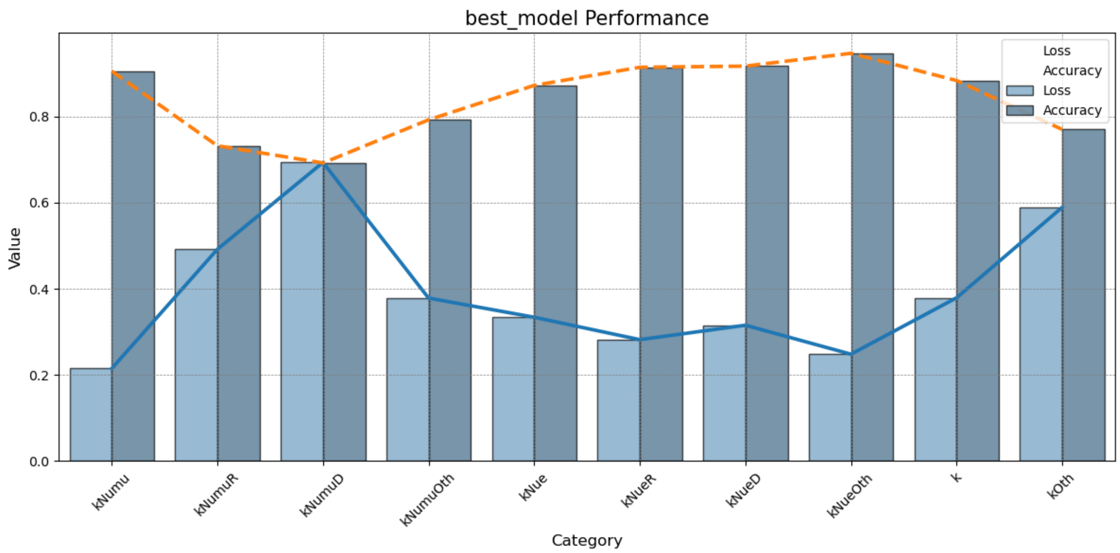
2.1.4 Interaction

[14]:
```
catPlot(interaction, Interaction)
evalPlot(interaction, Interaction)
```



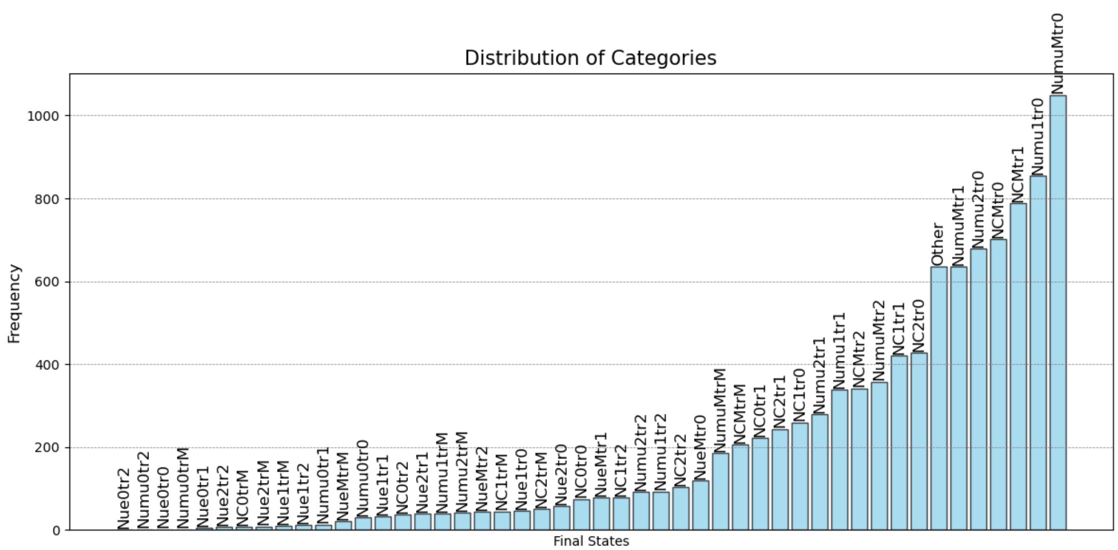

```
Skipping Category: kNutau, cvm_data is empty.
Skipping Category: kNutauR, cvm_data is empty.
Skipping Category: kNutauD, cvm_data is empty.
Skipping Category: kNutauOth, cvm_data is empty.
Skipping Category: kNuElectronElast, cvm_data is empty.
Skipping Category: kCosm, cvm_data is empty.
Skipping Category: kNIntTy, cvm_data is empty.
```

```
/Users/igorbykov/opt/anaconda3/envs/ML/lib/python3.9/site-
packages/seaborn/categorical.py:1728: UserWarning: You passed a
edgecolor/edgecolors ((0.12156862745098039, 0.4666666666666667,
0.7058823529411765)) for an unfilled marker (''). Matplotlib is ignoring the
edgecolor in favor of the facecolor. This behavior may change in the future.
  ax.scatter(x, y, label=hue_level,
/Users/igorbykov/opt/anaconda3/envs/ML/lib/python3.9/site-
packages/seaborn/categorical.py:1728: UserWarning: You passed a
edgecolor/edgecolors ((1.0, 0.4980392156862745, 0.054901960784313725)) for an
unfilled marker (''). Matplotlib is ignoring the edgecolor in favor of the
facecolor. This behavior may change in the future.
  ax.scatter(x, y, label=hue_level,
```



2.1.5 Final State

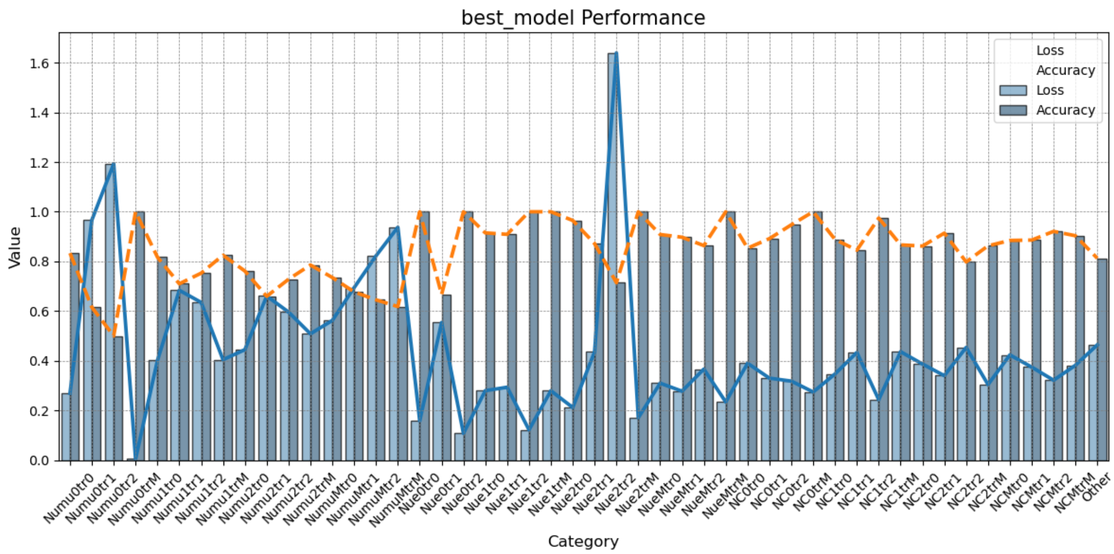
[15]:
```
catPlot(finalstate, FinalState)
evalPlot(finalstate, FinalState)
```




```
Skipping Category: Nue0trM, cvm_data is empty.
Skipping Category: Cosmic, cvm_data is empty.
```

```
/Users/igorbykov/opt/anaconda3/envs/ML/lib/python3.9/site-
packages/seaborn/categorical.py:1728: UserWarning: You passed a
edgecolor/edgecolors ((0.12156862745098039, 0.4666666666666667,
```

```
0.7058823529411765)) for an unfilled marker (''). Matplotlib is ignoring the
edgecolor in favor of the facecolor. This behavior may change in the future.
  ax.scatter(x, y, label=hue_level,
/Users/igorbykov/opt/anaconda3/envs/ML/lib/python3.9/site-
packages/seaborn/categorical.py:1728: UserWarning: You passed a
edgecolor/edgecolors ((1.0, 0.4980392156862745, 0.054901960784313725)) for an
unfilled marker (''). Matplotlib is ignoring the edgecolor in favor of the
facecolor. This behavior may change in the future.
  ax.scatter(x, y, label=hue_level,
```

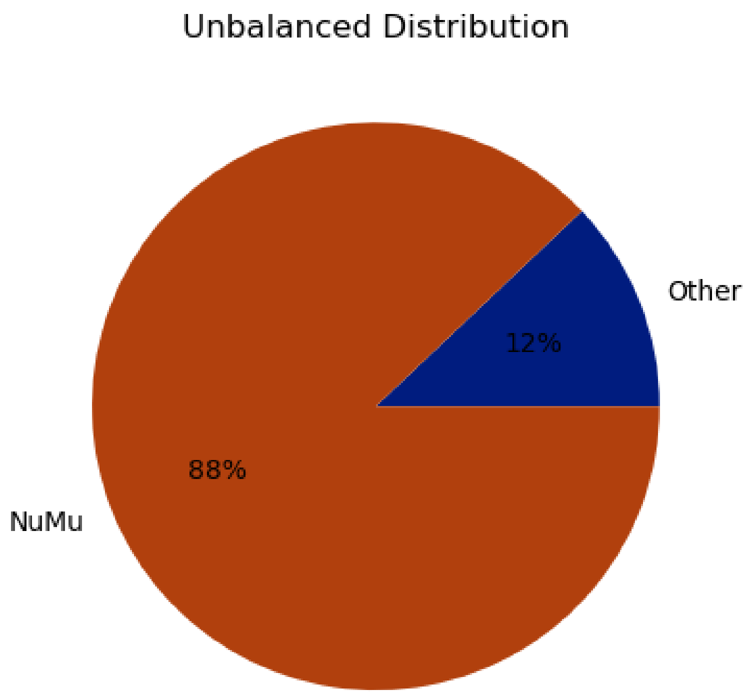


2.2 Using Unbalanced Data

[16]:
```python
# Get new data for evaluation, from a different set of files
from_file_N, to_file_N = 37,42
cvm_data, types, lepenergy, nuenergy, interaction, finalstate =␣
 ↪fetchData(from_file_N, to_file_N, balance = False)
```

```
File 37:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino37.h5
|                        | 6957/6957 [100%] in 46.4s (150.05/s)
File 38:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino38.h5
|                        | 7081/7081 [100%] in 47.3s (149.77/s)
File 39:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino39.h5
|                        | 7057/7057 [100%] in 47.2s (149.55/s)
File 40:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino40.h5
|                        | 6970/6970 [100%] in 46.5s (150.05/s)
```

```
File 41:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino41.h5
|                           | 7066/7066 [100%] in 47.1s (150.04/s)
File 42:
http://www.hep.ucl.ac.uk/undergrad/0056/other/projects/nova/neutrino42.h5
|                           | 6815/6815 [100%] in 45.7s (149.26/s)
Successfully Obtained Data for 41946 Events
```



2.2.1 Overall Performance

[17]:
```python
scores = best_model.evaluate(cvm_data, types, verbose=2)
```

```
1311/1311 - 84s - loss: 0.5462 - accuracy: 0.7554 - 84s/epoch - 64ms/step
```

[18]:
```python
pred = best_model.predict(cvm_data)
# Now calculate the F1 score
pred_binary = (pred > 0.5).astype(int)
f1 = f1_score(types, pred_binary)
f1
```

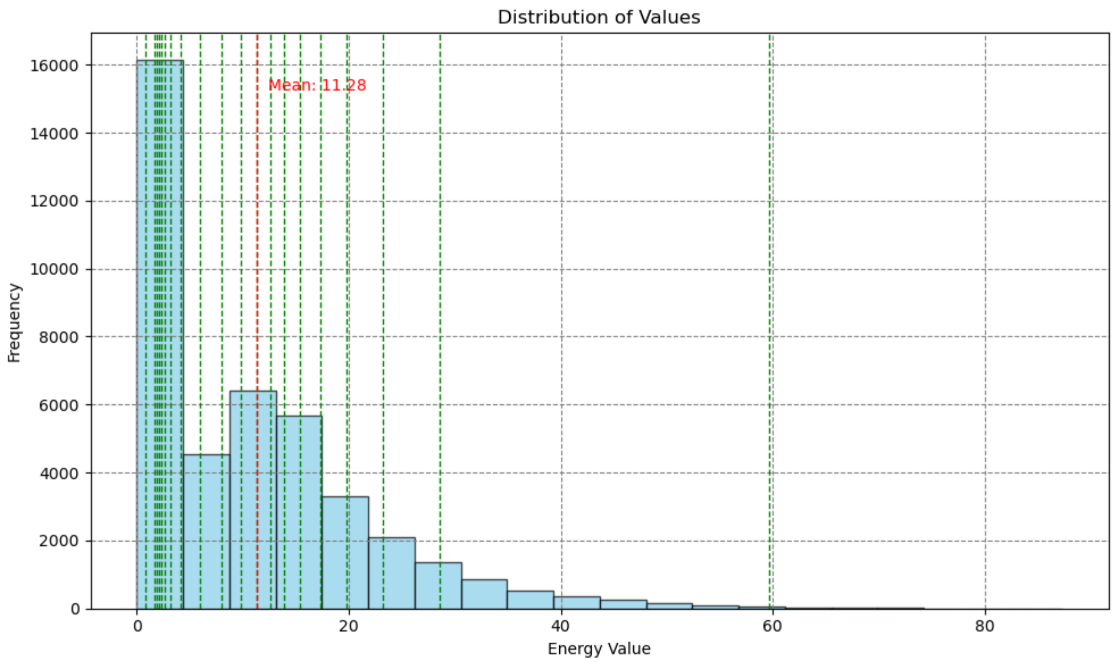
```
1311/1311 [==============================] - 89s 68ms/step
```

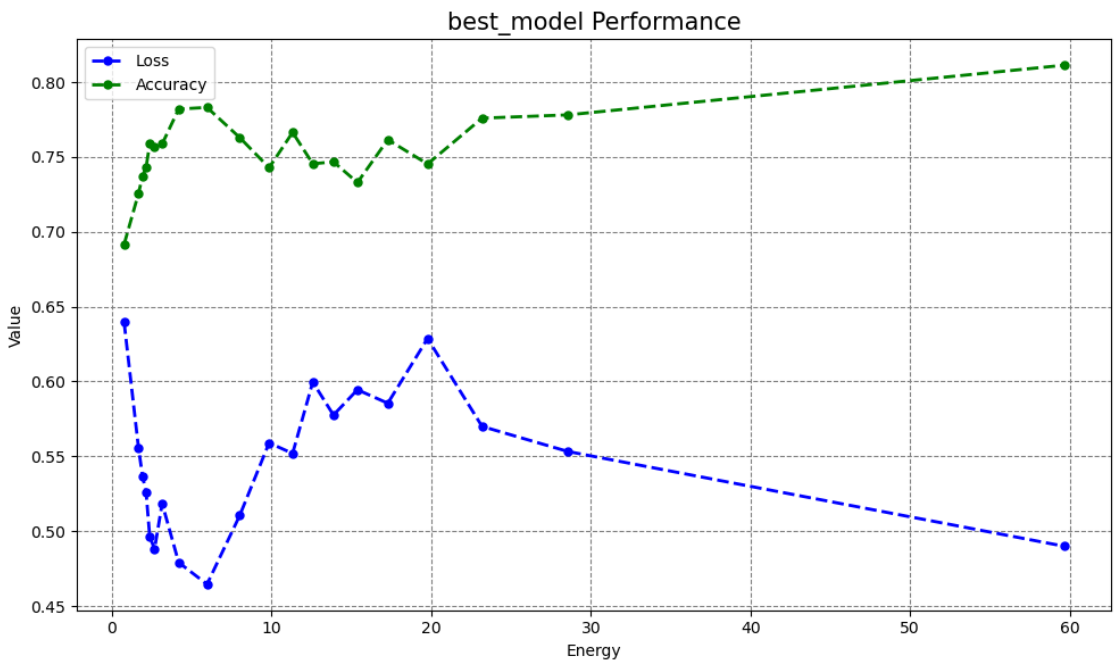
[18]: 0.8414253484562846

[ ]:

2.2.2 NuEnergy

[19]:
```
N_bins = 20
metavar = nuenergy
# Evaluating bins now to plot energy value bin boundaries
loss_data, acc_data = binEval(metavar, best_model, cvm_data, types, N_bins)

EnergyBinPlot(metavar, N_bins, loss_data, acc_data)
EnergyBinPerform(metavar, N_bins, loss_data, acc_data)
```

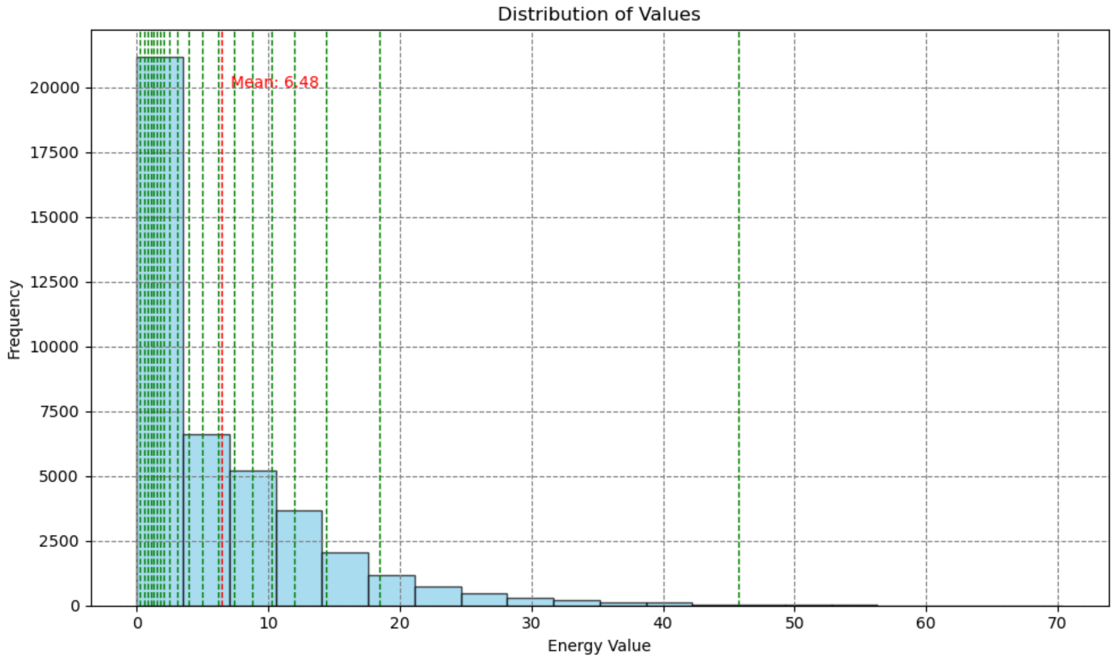


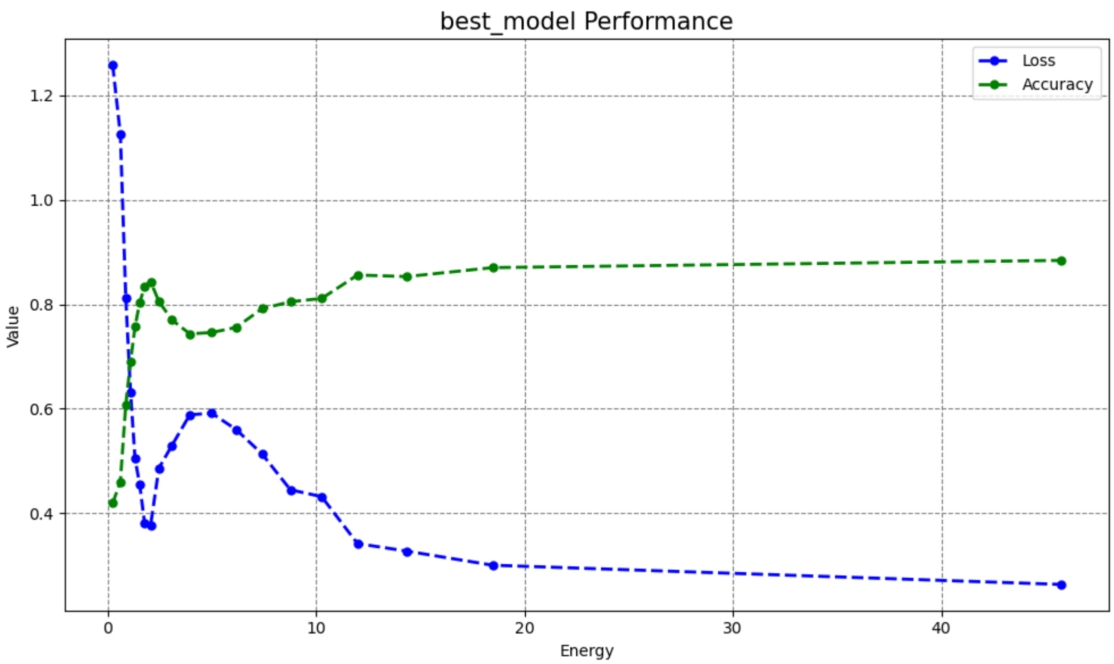


<Figure size 640x480 with 0 Axes>

2.2.3 LepEnergy

[20]:
```
N_bins = 20
metavar = lepenergy

# Evaluating bins now to plot energy value bin boundaries
loss_data, acc_data = binEval(metavar, best_model, cvm_data, types, N_bins)

EnergyBinPlot(metavar, N_bins, loss_data, acc_data)
EnergyBinPerform(metavar, N_bins, loss_data, acc_data)
```

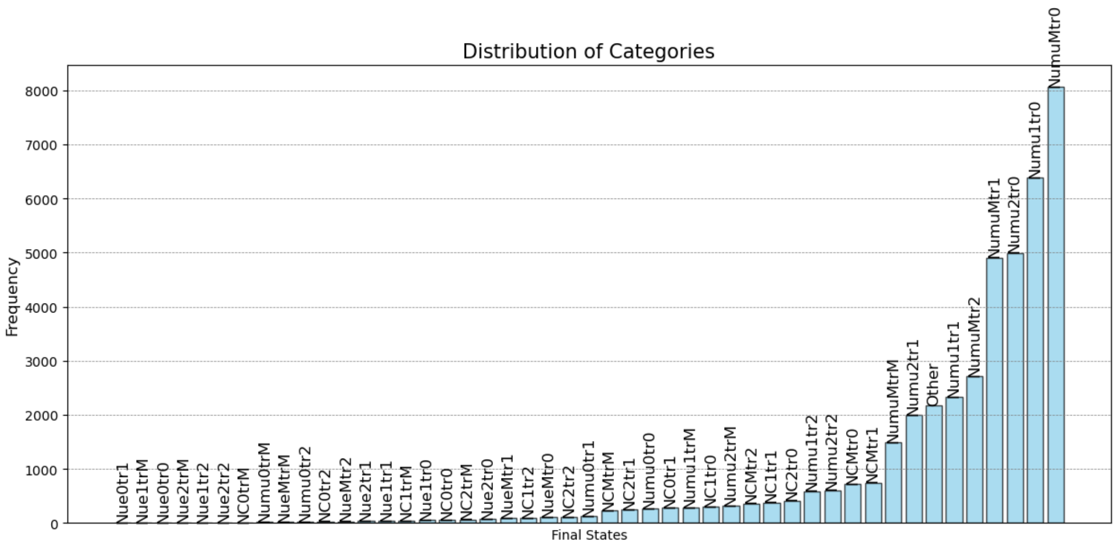





```
<Figure size 640x480 with 0 Axes>
```

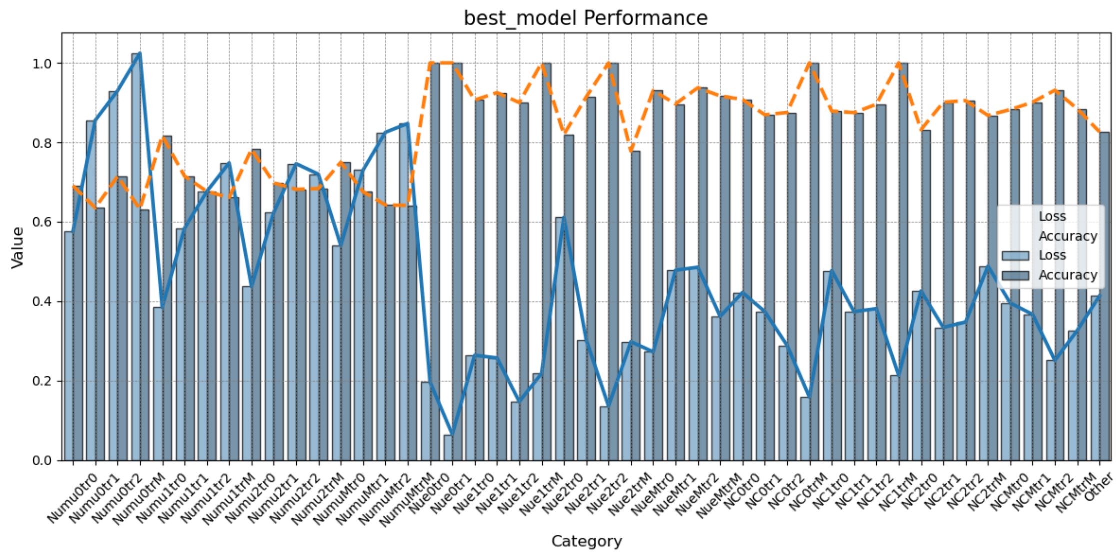
2.2.4 FinalState

[21]:
```
catPlot(finalstate, FinalState)
evalPlot(finalstate, FinalState)
```




```
Skipping Category: Nue0tr2, cvm_data is empty.
Skipping Category: Nue0trM, cvm_data is empty.
Skipping Category: Cosmic, cvm_data is empty.
```

```
/Users/igorbykov/opt/anaconda3/envs/ML/lib/python3.9/site-
packages/seaborn/categorical.py:1728: UserWarning: You passed a
edgecolor/edgecolors ((0.12156862745098039, 0.4666666666666667,
0.7058823529411765)) for an unfilled marker ('').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
  ax.scatter(x, y, label=hue_level,
/Users/igorbykov/opt/anaconda3/envs/ML/lib/python3.9/site-
packages/seaborn/categorical.py:1728: UserWarning: You passed a
edgecolor/edgecolors ((1.0, 0.4980392156862745, 0.054901960784313725)) for an
unfilled marker ('').  Matplotlib is ignoring the edgecolor in favor of the
facecolor.  This behavior may change in the future.
  ax.scatter(x, y, label=hue_level,
```



2.2.5 Interaction

[ ]:
```
catPlot(interaction, Interaction)
evalPlot(interaction, Interaction)
```