# ML_Network_Anomaly_Analysis

October 25, 2024

## 1 Machine Learning Network Anomaly Analysis and Prediction for CPE 400

---

### 1.1 This is my final project. In this project, I am analyzing the network data through the many subplots. After analysis, I provide different predictions of the data in the dataset by using different data algorithms. This includes Naive Bayes, Logistic Regression, Neural Network.

### 1.2 First we need to set up the dataset for training

---

**I am using a dataset from kaggle, therefore, i first need to set up and upload by key credentials, then i can start with the dataset**

**Here is the dataset i am using: https://www.kaggle.com/datasets/ernie55ernie/improved-cicids2017-and-csecicids2018/data**

---

[1]:
```
import os
import shutil
import zipfile
import subprocess

isUsingColab = input("Are you using Google Colab (y/n): ").lower().strip() == "y"
# Ask the user whether they need to download the datasets
askDownload = input("Do you need to download the datasets (y/n)?: ").lower().
 ↪strip() == "y"
mode = int(input("Enter dataset selection (0: Both, 1: CICIDS2017, 2:␣
 ↪CSECICIDS2018): "))
askPlot = input("Process any data (y/n)?: ").lower().strip() == "y"
askCPU = int(input("How many CPU cores would you like to use?: "))

if (askPlot):
  askGraph = input("Process subplot data (y/n)?: ").lower().strip() == "y"

if isUsingColab:
```

```
    if askDownload:
        from google.colab import files

        # Upload the Kaggle API credentials
        print("Please upload your kaggle.json file:")
        uploaded = files.upload()  # Upload file in Colab

        # Save the uploaded kaggle.json file in the appropriate directory
        for filename in uploaded.keys():
            print(f'User uploaded file "{filename}" with length␣
↪{len(uploaded[filename])} bytes')

        # Create the .kaggle directory if it doesn"t exist
        kaggleDir = os.path.expanduser("~/.kaggle")
        if not os.path.exists(kaggleDir):
            os.makedirs(kaggleDir)

        # Define the path where kaggle.json will be copied
        kaggleKeyDest = os.path.join(kaggleDir, "kaggle.json")

        # Save kaggle.json file to the destination
        with open(kaggleKeyDest, "wb") as kaggleFile:
            kaggleFile.write(uploaded[filename])

        # Set the correct permissions for the file (Unix-based systems)
        os.chmod(kaggleKeyDest, 0o600)

        # Check if the datasets already exist before downloading
        dataset2017 = "CICIDS2017_improved"
        dataset2018 = "CSECICIDS2018_improved"
        if not os.path.exists(dataset2017) or not os.path.exists(dataset2018):
            # Install Kaggle API using pip
            subprocess.run(["python", "-m", "pip", "install", "kaggle"],␣
↪check=True)

            # Define the dataset
            dataset = "ernie55ernie/improved-cicids2017-and-csecicids2018"

            # Download the dataset using the Kaggle API
            subprocess.run(["kaggle", "datasets", "download", "-d", dataset],␣
↪check=True)

            # Unzip the downloaded dataset
            zipFile = "improved-cicids2017-and-csecicids2018.zip"
            with zipfile.ZipFile(zipFile, "r") as zipRef:
              print("File is being extracted")
              zipRef.extractall()
```

```
        # Delete the zip file after extraction
            os.remove(zipFile)
            print(f"Dataset downloaded, extracted, and zip file {zipFile}␣
 ↪deleted.")
        else:
            print(f"Dataset already exists in the {dataset2017} and␣
 ↪{dataset2018} folders. No download needed.")
else:
    if askDownload:
        # Ask user to manually place kaggle.json in the correct directory
        kaggleDir = os.path.expanduser("~/.kaggle")
        kaggleKeyDest = os.path.join(kaggleDir, "kaggle.json")

        if not os.path.exists(kaggleKeyDest):
            print(f"Please manually place the kaggle.json file in {kaggleDir}")

        # Check if the datasets already exist before downloading
        dataset2017 = "CICIDS2017_improved"
        dataset2018 = "CSECICIDS2018_improved"
        if not os.path.exists(dataset2017) or not os.path.exists(dataset2018):
            # Install Kaggle API using pip
            subprocess.run(["python", "-m", "pip", "install", "kaggle"],␣
 ↪check=True)

            # Define the dataset
            dataset = "ernie55ernie/improved-cicids2017-and-csecicids2018"

            # Download the dataset using the Kaggle API
            subprocess.run(["kaggle", "datasets", "download", "-d", dataset],␣
 ↪check=True)

            # Unzip the downloaded dataset
            zipFile = "improved-cicids2017-and-csecicids2018.zip"
            with zipfile.ZipFile(zipFile, "r") as zipRef:
                print("File is being extracted")
                zipRef.extractall()

            # Delete the zip file after extraction
            os.remove(zipFile)
            print(f"Dataset downloaded, extracted, and zip file {zipFile}␣
 ↪deleted.")
        else:
            print(f"Dataset already exists in the {dataset2017} and␣
 ↪{dataset2018} folders. No download needed.")
```

```
Are you using Google Colab (y/n):  n
```

```
Do you need to download the datasets (y/n)?:  n
Enter dataset selection (0: Both, 1: CICIDS2017, 2: CSECICIDS2018):  1
Process any data (y/n)?:  n
How many CPU cores would you like to use?:  5
```

### 1.3 Next, I will choose what dataset(s) I would like to use

---

**I am loading the dataset(s) into a variable - dfList**

---

[2]:
```
import glob
from tqdm import tqdm
import pandas as pd
from sklearn.preprocessing import LabelEncoder
import gc

dfList = []
labels = []

projectDir = os.getcwd()
pathToCSV2017 = os.path.join(projectDir, "CICIDS2017_improved")
pathToCSV2018 = os.path.join(projectDir, "CSECICIDS2018_improved")

# Based on the mode, decide which dataset(s) to include
csvCombined = []
if mode == 0:
  # Load both datasets
  csv2017 = glob.glob(os.path.join(pathToCSV2017, "*.csv"))
  csv2018 = glob.glob(os.path.join(pathToCSV2018, "*.csv"))
  csvCombined = csv2017 + csv2018
elif mode == 1:
  # Load only CICIDS2017 dataset
  csv2017 = glob.glob(os.path.join(pathToCSV2017, "*.csv"))
  csvCombined = csv2017
elif mode == 2:
  # Load only CSECICIDS2018 dataset
  csv2018 = glob.glob(os.path.join(pathToCSV2018, "*.csv"))
  csvCombined = csv2018
else:
  raise ValueError("Invalid mode selected. Choose 0 (both), 1 (CICIDS2017), or 2␣
 ↪(CSECICIDS2018).")

# Iterate over files with tqdm for progress tracking
for file in tqdm(csvCombined, desc="Reading CSV files"):
  # Read the CSV file into a DataFrame and append to the list
  df = pd.read_csv(file)
```

```
  dfList.append(df)

print("Finished Reading CSV file(s), Starting Encoding...")
# Encode labels for each DataFrame
aLabels = pd.concat([df["Label"] for df in dfList]).unique()
le = LabelEncoder()
le.fit(aLabels)
for idx in range(len(dfList)):
  oLabels = dfList[idx]["Label"].unique()
  eLabels = le.transform(dfList[idx]["Label"])

  labelMap = {original: le.transform([original])[0] for original in oLabels}
  labels.append(labelMap)

  #print (labelMap)

  dfList[idx]["Label"] = eLabels

print("Finished Encoding.")
```

```
Reading CSV files:
100%|-----------------------------------------------------------| 5/5
[00:09<00:00,  1.80s/it]

Finished Reading CSV file(s), Starting Encoding...
Finished Encoding.
```

### 1.4 After that, I make a function to plot all of the unprocessed data for analysis

---

**I save this data to the Figures folder to be downloaded later**

---

[3]:
```
import numpy as np
import pandas as pd
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import re

def plotData(columns, xlabel, ylabel, sOn, labelMap):
  print("Mapping Colors")
  numColors = 27
  cmap = plt.colormaps.get_cmap("tab20")
  colors = cmap(np.linspace(0, 1, numColors))

  if not os.path.exists("Figures"):
```

```
    os.makedirs("Figures")

  if sOn:
    if not os.path.exists("Figures/SubPlots"):
      os.makedirs("Figures/SubPlots")

    print("Enumerating through our DataFrame List to Plot Data...")
    # Loop through each DataFrame and each column
    for dfIdx, df in enumerate(dfList):
      df = df.dropna()  # Removes rows with NaN values
      for colIdx, col in enumerate(columns):
        # Create a new figure for each dataset and column
        fig, ax = plt.subplots(figsize=(10, 6))  # Adjust size if necessary

        # Plot the data
        ax.plot(df.index, df[col], label=f"{col} (Dataset {dfIdx + 1})",␣
↪color=colors[(dfIdx + colIdx) % numColors], linestyle="-", linewidth=1)

        # Set labels and title for each individual plot
        ax.set_title(f"{col} - Dataset {dfIdx + 1}", fontsize=12)
        ax.set_xlabel(xlabel, fontsize=10)
        ax.set_ylabel(ylabel, fontsize=10)
        ax.grid(True, linestyle=":", linewidth=0.7, color="grey")

        # Adjust layout to avoid overlapping elements
        plt.tight_layout()

        # Save each plot with a unique filename
        plt.savefig(f"Figures/SubPlots/Dataset{dfIdx+1}_{col}.png")
        plt.close(fig)  # Close the figure after saving to avoid memory buildup

  if not os.path.exists("Figures/Histograms"):
    os.makedirs("Figures/Histograms")

  print("Finished Subplots, Enumerating Data to Plot Histograms...")
  for dfIdx, df in enumerate(dfList):
    # Get numerical columns excluding "Label" and "id" or any non-numeric columns
    numericCols = df.select_dtypes(include=[np.number]).columns.tolist()
    numericCols.remove("Label")  # Remove Label

    for col in numericCols:
      plt.figure(figsize=(15, 15))

      # Initialize an empty list to hold average values for each label
      averages = []

      # Plot histogram for each label type
```

```
    for label in df["Label"].unique():
      # Get data for the current label
      data = df[df["Label"] == label][col]
      # Remove NaN and inf values from the data
      data = data[np.isfinite(data)]

      if len(data) > 0:  # Check if there is data to plot
          avgVal = data.mean()

          oLabel = next((orig for orig, enc in labelMap[dfIdx].items() if enc
↪== label), str(label))

          averages.append((oLabel, avgVal))

          # Use colormap to get color for the label
          plt.hist(data, bins=30, alpha=0.5, color=colors[label],
                  label=f"{oLabel} (Avg: {avgVal:.2f})", edgecolor="black")

    # Set titles and labels
    plt.title(f"Histogram of {col} for DataFrame {dfIdx + 1}")
    plt.xlabel(col)
    plt.ylabel("Frequency")
    plt.legend(title="Label (Average Value)")

    # Show grid
    plt.grid(axis="y", alpha=0.75)

    if not os.path.exists(f"Figures/Histograms/{dfIdx + 1}"):
        os.makedirs(f"Figures/Histograms/{dfIdx + 1}")

    sCol = re.sub(r"[^\w\s]", "", col)  # Remove non-alphanumeric characters
    sCol = sCol.replace(" ", "_")  # Replace spaces with underscores

    plt.savefig(f"Figures/Histograms/{dfIdx + 1}/Hist{dfIdx + 1}_{sCol}.png")

    # Close the figure to free up memory
    plt.close()

print("Finished All Plotting.")
```

### 1.5 I also create a function to scale and sample our data sets to make them less affected by outliers

---

[4]:
```
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import RandomOverSampler
```

```
import pickle

def saveIntrm(data, filename):
  with open(filename, 'wb') as f:
    pickle.dump(data, f)

def loadIntrm(filename):
  with open(filename, 'rb') as f:
    return pickle.load(f)

def scaleDS(df, cToDrop, overSample=False):
  X = df.drop(columns=cToDrop)

  # Handle NaN values and Infinite values
  X = X[np.isfinite(X).all(axis=1)]
  y = df[df.columns[-2]][X.index].values  # Align y with the index of X after␣
↪dropping rows

  X.dropna(inplace=True)
  if np.isinf(X).sum().sum() > 0:
    print("\nWarning: Infinite values still present after replacement.\n")

  scaler = StandardScaler()
  X = scaler.fit_transform(X)

  if isinstance(y, np.ndarray):
    y = pd.Series(y)

  if overSample:
    unique = y.unique()
    #print(f"Unique Classes in y: {unique}")

    if len(unique) >= 2:
        ros = RandomOverSampler()
        X, y = ros.fit_resample(X, y)

  if X.shape[0] != len(y):
    print(f"Dimension mismatch: X has {X.shape[0]} rows, y has {len(y)} entries.
↪")

  data = np.hstack((X, np.reshape(y, (-1, 1))))

  return data, X, y
```

### 1.6 After defining those functions, I am actually running them here and preparing for the use of our data

---

[5]:
```
import warnings

# Suppress a specific FutureWarning with a message matching the text
warnings.filterwarnings("ignore", message=".*'DataFrame.swapaxes' is deprecated.
 ↪*")

if (askPlot):
  columns = ["Total Fwd Packet", "Total Bwd packets", "Average Packet Size"]
  try:
    plotData(columns, "Time", "Number of Packets", askGraph, labels)
  except Exception as e:
      # Handle any exceptions or errors
      print(f"An error occurred: {e}")

gc.collect()

colToDrop = ["id", "Flow ID", "Src IP", "Dst IP", "Timestamp"]

# Prepare data for training, validation, and testing
train = []
valid = []
test = []
for df in dfList:
  tr, va, te = np.split(df.sample(frac=1), [int(0.6 * len(df)), int(0.8 *␣
 ↪len(df))])

  train.append(tr)
  valid.append(va)
  test.append(te)

trScale = []
XTrain = []
yTrain = []

vaScale = []
XValid = []
yValid = []

teScale = []
XTest = []
yTest = []
count = 0
```

```
# Scale values relative to mean
for df in dfList:
  # OverSample allows us to balance the amount of data if we want
  trS, XTr, yTr = scaleDS(train[count], colToDrop, overSample=True)
  vaS, XV, yV = scaleDS(valid[count], colToDrop, overSample=False)
  teS, XTe, yTe = scaleDS(test[count], colToDrop, overSample=False)

  # Save intermediate results to disk to free up RAM
  saveIntrm(trS, f'train_scaled_{count}.pkl')
  saveIntrm(XTr, f'X_train_{count}.pkl')
  saveIntrm(yTr, f'y_train_{count}.pkl')

  saveIntrm(vaS, f'validate_scaled_{count}.pkl')
  saveIntrm(XV, f'X_validate_{count}.pkl')
  saveIntrm(yV, f'y_validate_{count}.pkl')

  saveIntrm(teS, f'test_scaled_{count}.pkl')
  saveIntrm(XTe, f'X_test_{count}.pkl')
  saveIntrm(yTe, f'y_test_{count}.pkl')

  '''
  trScale.append(trS)
  XTrain.append(XTr)
  yTrain.append(yTr)

  vaScale.append(vaS)
  XValid.append(XV)
  yValid.append(yV)

  teScale.append(teS)
  XTest.append(XTe)
  yTest.append(yTe)
  '''

  count += 1

del train
del valid
del test
del colToDrop
gc.collect()
```

[5]: 0

### 1.7 When we are finished with our plotting and preparation, I start with the Naive Bayes analysis

**Naive Bayes tries to predict our Labels by using the likelihood of seeing any given Label with respect to the prior Labels and the evidence we already have before us**

**The original mathematical function of the Naive Bayes is given as:**

$$P(C_k|x_1, x_2, ..., x_n) = \frac{P(x_1, x_2, ..., x_n|C_k) * P(C_k)}{P(x_1, x_2, ..., x_n)}$$

**We can then further derive it:**

$$P(C_k|x_1, x_2, ..., x_n) \propto P(x_1, x_2, ..., x_n|C_k) * P(C_k)$$

**Now, since we assume all the probabilities $x_1$, $x_2$, ..., $x_n$ are independent, we can just multiply the probabilities:**

$$P(C_k|x_1, x_2, ..., x_n) \propto (P(x_1|C_k) * P(x_2|C_k) * ... * P(x_n|C_k) * P(C_k)$$

**We can then rewrite this like:**

$$P(C_k|x_1, x_2, ..., x_n) \propto P(C_k) \prod_{i=1}^{n} P(x_i|C_k)$$

---

**Now, to predict the values in our dataset, we utilize the function:**

*Note, argmax is the maximizing function. This is known as the MAP (Maximum A Posteriori)*

$$\hat{y} = argmax * P(C_k) \prod_{i=1}^{n} P(x_i|C_k)$$

$$k \in \{1, k\}$$

---

[6]:
```
from sklearn.naive_bayes import GaussianNB # Gaussian Naive Bayes
from sklearn.metrics import classification_report, confusion_matrix,␣
↪ConfusionMatrixDisplay
from matplotlib.colors import LogNorm, Normalize # To make the Confusion Matrix␣
↪Display easier to read
import matplotlib.ticker as ticker

nbModel = GaussianNB()

# Load data from disk
for dfIdx in range(len(dfList)):
  XTr = loadIntrm(f'X_train_{dfIdx}.pkl')
  yTr = loadIntrm(f'y_train_{dfIdx}.pkl')
  XTe = loadIntrm(f'X_test_{dfIdx}.pkl')
  yTe = loadIntrm(f'y_test_{dfIdx}.pkl')
```

```
  XTrain.append(XTr)
  yTrain.append(yTr)
  XTest.append(XTe)
  yTest.append(yTe)

del XTr
del yTr
del XTe
gc.collect()

# Concatenate all training data
XTrainCom = np.concatenate(XTrain, axis=0)  # Combine all training features
yTrainCom = np.concatenate(yTrain, axis=0)  # Combine all training labels

del XTrain
del yTrain
gc.collect()

# Fit the model using the combined training data
print("Fitting Gaussian Naive Bayes Model...")
nbModel.fit(XTrainCom, yTrainCom)

# Predict using the test sets
print("Finished Fitting, Beginning Prediciton...")
yPredGNB = []
for i in range(len(XTest)):
    preds = nbModel.predict(XTest[i])
    yPredGNB.append(preds)

del nbModel
gc.collect()

# Flatten yPred if you want a single array
yPredGNB = np.concatenate(yPredGNB)
yTestCom = np.concatenate(yTest)

print("\nGaussian Classification Report:\n")
print(classification_report(yTestCom, yPredGNB, zero_division=0))

cm = confusion_matrix(yTestCom, yPredGNB, normalize='true')
allLabels = pd.concat([df["Label"] for df in dfList]).unique()
le.fit(allLabels)
yTrue = le.transform(yTestCom)
reverseLabels = le.inverse_transform(np.unique(yTrue))
maskedCM = np.ma.masked_where(cm == 0.00, cm)

# Display the confusion matrix
```

```
d = ConfusionMatrixDisplay(maskedCM, display_labels=reverseLabels)
fig, axs = plt.subplots(figsize=(25, 25))

# Define a colormap that will give zero values a light color
cmap = plt.cm.inferno
cmap.set_under('lightgray')

# Plot the confusion matrix with LogNorm but allowing very light color for 0␣
 ↪values
im = axs.imshow(maskedCM, interpolation='nearest', cmap=cmap,␣
 ↪norm=LogNorm(vmin=0.1, vmax=cm.max()))

# Add a colorbar and adjust its size to match the height of the plot
cbar = fig.colorbar(im, ax=axs, fraction=0.046, pad=0.04)
cbar.ax.tick_params(labelsize=15)

# Customize the title and axis labels
axs.set_title('Confusion Matrix for Gaussian Naive Bayes', fontsize=30, pad=20)
axs.set_xlabel('Predicted label', fontsize=20, labelpad=20)
axs.set_ylabel('True label', fontsize=20, labelpad=20)

# Customize the tick labels and display class names
axs.set_xticks(np.arange(len(reverseLabels)))
axs.set_yticks(np.arange(len(reverseLabels)))
axs.set_xticklabels(reverseLabels, fontsize=15, rotation=90)
axs.set_yticklabels(reverseLabels, fontsize=15)

# Rotate the tick labels and set their alignment
plt.setp(axs.get_xticklabels(), ha="right", rotation_mode="anchor")

# Add spacing to ticks
axs.xaxis.set_major_locator(ticker.MultipleLocator(1))
axs.yaxis.set_major_locator(ticker.MultipleLocator(1))

# Add text annotations inside the confusion matrix cells
thresh = cm.max() / 2.  # Threshold for text color (white vs black)
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        axs.text(j, i, format(cm[i, j], '.2f'),
                 ha="center", va="center",
                 color="black" if cm[i, j] > thresh else "white")

# Adjust plot layout to ensure everything fits
plt.subplots_adjust(left=0.2, right=0.8, top=0.9, bottom=0.1)

if not os.path.exists(f"Figures/Confusion_Matrix"):
    os.makedirs(f"Figures/Confusion_Matrix")
```

```
plt.savefig(f"Figures/Confusion_Matrix/GaussianNB.png")

del yPredGNB
gc.collect()
```

```
Fitting Gaussian Naive Bayes Model...
Finished Fitting, Beginning Prediciton...

Gaussian Classification Report:

              precision    recall  f1-score   support

           0       0.81      1.00      0.89    316418
           1       0.00      0.00      0.00       146
           2       0.00      0.00      0.00       777
           3       0.00      0.00      0.00     18920
           4       0.00      0.00      0.00      1525
           5       0.00      0.00      0.00        18
           6       1.00      0.88      0.93     31711
           7       0.00      0.00      0.00       108
           8       0.00      0.00      0.00       377
           9       1.00      0.81      0.90       704
          10       0.00      0.00      0.00       788
          11       1.00      0.08      0.15       353
          12       0.00      0.00      0.00       834
          13       1.00      1.00      1.00         2
          14       0.00      0.00      0.00         1
          15       1.00      0.25      0.40        12
          16       1.00      0.86      0.92         7
          17       1.00      0.01      0.01     14403
          18       0.00      0.00      0.00     31850
          19       0.00      0.00      0.00       627
          20       1.00      1.00      1.00         6
          21       0.00      0.00      0.00         9
          22       1.00      1.00      1.00       250
          23       0.00      0.00      0.00         3
          24       1.00      0.50      0.67         2
          25       0.00      0.00      0.00         6
          26       1.00      1.00      1.00       139

    accuracy                           0.82    419996
   macro avg       0.44      0.31      0.33    419996
weighted avg       0.72      0.82      0.75    419996
```

```
[6]: 2173
```

### 1.8 After the Naive Bayes implementation, I move onto the Logistic Regression implementation

---

**Logistic Regression tries to predict our Labels by using the probability of any given point being above a given line so we can determine it as a Label**

**We know that the slope of a regular regression line is given as:**

$$\hat{y} = mx + b$$

**When using Logistic Regression, our line cant just be defined by $\hat{y}$. We instead have to start with:**

$$p = mx + b$$

**Now, since $mx + b$ ranges from $-\infty$ to $\infty$ while probability has to be between 0 and 1, we set the "odds" of something being over or under our line:**

$$\ln \frac{p}{1-p} = mx + b$$

**To solve for p:**

$$e^{\ln \frac{p}{1-p}} = e^{mx+b}$$
$$\frac{p}{1-p} = e^{mx+b}$$
$$p = e^{mx+b}(1-p)$$
$$p = e^{mx+b} - pe^{mx+b}$$
$$p(1 + e^{mx+b}) = e^{mx+b}$$
$$p = \frac{e^{mx+b}}{1 + e^{mx+b}}$$

**Since we want a numerator of 1:**

$$p = \frac{e^{mx+b}}{1 + e^{mx+b}} * \frac{e^{-(mx+b)}}{e^{-(mx+b)}}$$
$$p = \frac{1}{1 + e^{-(mx+b)}}$$

---

**This gives us the special form similar to a Sigmoid function (S):**

$$S(x) = \frac{1}{1 + e^{-(x)}}$$

**So we can rewrite our function as:**

$$S(y) = \frac{1}{1 + e^{-(y)}}$$

---

[7]:
```
from sklearn.linear_model import LogisticRegression # Logistic Regression

warnings.filterwarnings("ignore", message="Setting penalty=None will ignore the␣
 ↪C and l1_ratio parameters")

def plot_logistic_results(history, penalty, regularization, dual):
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.plot(history['loss'], label='loss')
    ax.plot(history['accuracy'], label='accuracy')
    ax.set_xlabel('Iterations')
    ax.set_ylabel('Metrics')
    plt.legend()
    ax.grid(True)

    # Create directory for saving figures if it doesn't exist
    if not os.path.exists("Figures/Logistic_Regression"):
        os.makedirs("Figures/Logistic_Regression")

    plt.savefig(f"Figures/Logistic_Regression/
 ↪Results_{penalty}_{regularization}_{dual}.png")
    plt.show()

# This is a function i developed if you wanted to try and make the logistic␣
 ↪regression even more accurate
def trainLR(X_train, y_train, X_test, y_test, num_cores, penalties,␣
 ↪regularizations, random_state):
    best_model = None
    best_val_score = 0
    history_results = {'loss': [], 'accuracy': []}

    # Suppress warnings during conversion and reshaping
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")

        # Convert X_train and X_test to NumPy arrays if they are lists
        X_train = np.array(X_train)
        X_test = np.array(X_test)

        # Ensure X_train and X_test are 2D arrays
        if len(X_train.shape) == 1:
            X_train = X_train.reshape(-1, 1)
        if len(X_test.shape) == 1:
            X_test = X_test.reshape(-1, 1)


    for penalty in penalties:
        for reg in regularizations:
```

```
            if penalty == 'l2':
                for dual in [False, True]:
                    print(f"Penalty: {penalty}, Regularization: {reg}, Dual:␣
↪{dual}")
                    lr_model = LogisticRegression(
                        penalty=penalty, C=reg,
                        solver='saga', dual=dual, random_state=random_state,␣
↪n_jobs=(num_cores // 2 + 1)
                    )

                    # Fit the logistic regression model
                    print("Fitting Model...")
                    lr_model.fit(X_train, y_train)

                    # Evaluate on test set and collect metrics
                    print("Finished Fitting, Evaluating Prediction and Metrics...
↪")
                    y_pred = lr_model.predict(X_test)
                    acc_score = np.mean(y_pred == y_test)
                    history_results['accuracy'].append(acc_score)
                    print(classification_report(y_test, y_pred))

                    # Track best model
                    if acc_score > best_val_score:
                        best_val_score = acc_score
                        best_model = lr_model

                    history_results['loss'].append(lr_model.n_iter_[0])
                    plot_logistic_results(history_results, penalty, reg, dual)
            else:
                print(f"Penalty: {penalty}, Regularization: {reg}")
                lr_model = LogisticRegression(
                    penalty=penalty, C=reg,
                    solver='saga', random_state=random_state, n_jobs=(num_cores /
↪/ 2 + 1)
                )

                # Fit the logistic regression model
                print("Fitting Model...")
                lr_model.fit(X_train, y_train)

                # Evaluate on test set and collect metrics
                print("Finished Fitting, Evaluating Prediction and Metrics...")
                y_pred = lr_model.predict(X_test)
                acc_score = np.mean(y_pred == y_test)
                history_results['accuracy'].append(acc_score)
                print(classification_report(y_test, y_pred))
```

```
                # Track best model
                if acc_score > best_val_score:
                    best_val_score = acc_score
                    best_model = lr_model

                history_results['loss'].append(lr_model.n_iter_[0])
                plot_logistic_results(history_results, penalty, reg, None)

    return best_model
```

[8]:
```
penalties = [None, 'l1', 'l2', 'elasticnet']
regularizations = [0.85, 1, 1.15]
random_state = 1

# bestLRModel = trainLR(XTrainCom, yTrainCom, XTest, yTestCom, askCPU,␣
 ↪penalties, regularizations, random_state)

# -2 for the number of jobs causes the program to use all but 1 CPU cores
lrModel = LogisticRegression('l2', solver='saga', random_state=random_state,␣
 ↪n_jobs=askCPU)

print("Fitting Logistic Regression Model...")
lrModel.fit(XTrainCom, yTrainCom)

print("Finished Fitting, Beginning Prediciton...")
yPred = []
for i in range(len(XTest)):
    preds = lrModel.predict(XTest[i])
    yPred.append(preds)

yPred = np.concatenate(yPred)
yTestCom = np.concatenate(yTest)

# del bestLRModel
gc.collect()

print("\Logistic Regression Classification Report:\n")
print(classification_report(yTestCom, yPred))

cm = confusion_matrix(yTestCom, yPred, normalize='true')
maskedCM = np.ma.masked_where(cm == 0.00, cm)

# Display the confusion matrix
d = ConfusionMatrixDisplay(maskedCM, display_labels=reverseLabels)
fig, axs = plt.subplots(figsize=(25, 25))
```

```
# Define a colormap that will give zero values a light color
cmap = plt.cm.plasma
cmap.set_under('mediumpurple')

# Plot the confusion matrix with LogNorm but allowing very light color for 0␣
 ↪values
im = axs.imshow(maskedCM, interpolation='nearest', cmap=cmap,␣
 ↪norm=LogNorm(vmin=0.1, vmax=cm.max()))

# Add a colorbar and adjust its size to match the height of the plot
cbar = fig.colorbar(im, ax=axs, fraction=0.046, pad=0.04)
cbar.ax.tick_params(labelsize=15)

# Customize the title and axis labels
axs.set_title('Confusion Matrix for Logistic Regression', fontsize=30, pad=20)
axs.set_xlabel('Predicted label', fontsize=20, labelpad=20)
axs.set_ylabel('True label', fontsize=20, labelpad=20)

# Customize the tick labels and display class names
axs.set_xticks(np.arange(len(reverseLabels)))
axs.set_yticks(np.arange(len(reverseLabels)))
axs.set_xticklabels(reverseLabels, fontsize=15, rotation=90)
axs.set_yticklabels(reverseLabels, fontsize=15)

# Rotate the tick labels and set their alignment
plt.setp(axs.get_xticklabels(), ha="right", rotation_mode="anchor")

# Add spacing to ticks
axs.xaxis.set_major_locator(ticker.MultipleLocator(1))
axs.yaxis.set_major_locator(ticker.MultipleLocator(1))

# Add text annotations inside the confusion matrix cells
thresh = cm.max() / 2.  # Threshold for text color (white vs black)
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        axs.text(j, i, format(cm[i, j], '.2f'),
                 ha="center", va="center",
                 color="black" if cm[i, j] > thresh else "white")

# Adjust plot layout to ensure everything fits
plt.subplots_adjust(left=0.2, right=0.8, top=0.9, bottom=0.1)

if not os.path.exists(f"Figures/Confusion_Matrix"):
    os.makedirs(f"Figures/Confusion_Matrix")

plt.savefig(f"Figures/Confusion_Matrix/LogRegression.png")
```

```
gc.collect()
```

```
Fitting Logistic Regression Model...

C:\Users\Carter\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\linear_model\_sag.py:349: ConvergenceWarning: The max_iter was
reached which means the coef_ did not converge
  warnings.warn(

Finished Fitting, Beginning Prediciton...
\Logistic Regression Classification Report:

              precision    recall  f1-score   support

           0       1.00      0.99      1.00    316418
           1       0.07      0.99      0.13       146
           2       1.00      1.00      1.00       777
           3       0.98      1.00      0.99     18920
           4       0.86      0.99      0.92      1525
           5       0.09      1.00      0.16        18
           6       1.00      1.00      1.00     31711
           7       0.90      1.00      0.95       108
           8       0.89      0.99      0.94       377
           9       1.00      1.00      1.00       704
          10       0.98      0.99      0.99       788
          11       0.98      1.00      0.99       353
          12       1.00      1.00      1.00       834
          13       0.67      1.00      0.80         2
          14       0.14      1.00      0.25         1
          15       0.38      0.67      0.48        12
          16       0.55      0.86      0.67         7
          17       0.99      0.98      0.99     14403
          18       0.99      1.00      1.00     31850
          19       0.99      1.00      1.00       627
          20       0.33      1.00      0.50         6
          21       0.53      1.00      0.69         9
          22       0.92      0.98      0.95       250
          23       0.18      1.00      0.30         3
          24       0.22      1.00      0.36         2
          25       0.86      1.00      0.92         6
          26       0.81      0.99      0.89       139

    accuracy                           0.99    419996
   macro avg       0.72      0.98      0.77    419996
weighted avg       1.00      0.99      0.99    419996
```

[8]: 40

### 1.9 After going over Naive Bayes and Logistic Regression, we start by creating a loss-plotting function and an accuracy-plotting function

---

[9]:
```
def plot_hist(history, numNode, dropProb, learnRate, batchSize):
  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

  ax1.plot(history.history['loss'], label='loss')
  ax1.plot(history.history['val_loss'], label='val_loss')
  ax1.set_xlabel('Epoch')
  ax1.set_ylabel('Binary Cross Entropy')
  ax1.grid(True)

  ax2.plot(history.history['accuracy'], label='accuracy')
  ax2.plot(history.history['val_accuracy'], label='val_accuracy')
  ax2.set_xlabel('Epoch')
  ax2.set_ylabel('Accuracy')
  plt.legend()
  ax2.grid(True)

  if not os.path.exists("Figures/Neural_Network"):
    os.makedirs("Figures/Neural_Network")
    plt.savefig(f"Figures/Neural_Network/
↪Loss_And_Acc_{numNode}_{dropProb}_{learnRate}_{batchSize}.png")

  #plt.show()
```

### 1.10 After defining those functions, we will start on the Neural Network

---

A neural network has a bunch of nodes called neurons.

In a neural network, we have a bunch of input features $x_1, x_2, ..., x_n$ to process.

We sum all of these inputs with their respective weights, which then goes into each neuron. This neuron can have a specified bias applied to it to shift the values somewhat.

The output of the weighted input values being passed into the neurons with the bias all gets passed to the activation function. After applying the activation function, we get our output prediction.

Now, I wasn't actually able to run the Neural Network(s) in their entirety because it takes way too long. For a single model to get trained on the dataset as I have it now, it would take somewhere from 3 to 5 hours on just 25 epochs. The functionality is their, though, as I was able to see what a couple of the models looked like.

---

[10]:
```
import tensorflow as tf

def trainNN(XTrainSet, yTrainSet, numNodes, dropoutProb, learningRate,␣
↪batchSize, epochs):
  # Linearlly stack layers as a model
  nnModel = tf.keras.Sequential([
    tf.keras.layers.Dense(numNodes, activation='relu', input_shape=(86,)),    #␣
↪First layer uses RELU and 32 nodes
    tf.keras.layers.Dropout(dropoutProb),
    tf.keras.layers.Dense(numNodes, activation='relu'),                       #␣
↪Next layer is the same
    tf.keras.layers.Dropout(dropoutProb),

    tf.keras.layers.Dense(1, activation='sigmoid')                        # Last␣
↪layer uses Signmoid function
  ])

  # Compile the Neural Network with the Adam activation function using binary␣
↪cross entropy as our loss
  # We will also have another metric stored for us, accuracy
  print("Compiling Neural Network...")
  nnModel.compile(optimizer=tf.keras.optimizers.Adam(learningRate),
                  loss='binary_crossentropy',
                  metrics=['accuracy']
  )

  print("Finished Compiling, Fitting Neural Network Model...")
  history = nnModel.fit(
    XTrainCom, yTrainCom,
    epochs=epochs, batch_size=batchSize,
    validation_split=0.2, verbose=0
  )

  return nnModel, history
```

### 1.11 After defining the Neural Network function, we can use it with customized values to see what gets the best results

---

[ ]:
```
for dfIdx in range(len(dfList)):
  XVa = loadIntrm(f'X_validate_{dfIdx}.pkl')
  yVa = loadIntrm(f'y_validate_{dfIdx}.pkl')

  XValid.append(XVa)
  yValid.append(yVa)
```

```
del XVa
del yVa
gc.collect()

XValidCom = np.concatenate(XValid)
yValidCom = np.concatenate(yValid)

del XValid
del yValid
gc.collect()

leastValLoss = float('inf')
leastLossModel = None

epoch = 25
for numNode in [4, 8, 16, 32, 64]:
  for dropProb in [0, 0.1, 0.2]:
    for learnRate in [0.005, 0.001, 0.1]:
      for batchSize in [16, 32, 64, 128]:
        print(f"Nodes: {numNode}, Drop Probability: {dropProb}, Learn Rate:␣
↪{learnRate}, Batch Size: {batchSize}")
        model, history = trainNN(XTrainCom, yTrainCom, numNode, dropProb,␣
↪learnRate, batchSize, epoch)
        print("Finished Fitting, Plotting Data...")
        plot_hist(history, numNode, dropProb, learnRate, batchSize)

        valLoss = model.evaluate(XValidCom, yValidCom)[0]
        if valLoss < leastValLoss:
          leastValLoss = valLoss
          leastLossModel = model

yPr = leastLossModel.predict(XTest)
yPr = (yPr > 0.5).astype(int).reshape(-1,)
yPr = np.concatenate(yPr)
print(classification_report(yTestCom, yPr))
```

```
Nodes: 4, Drop Probability: 0, Learn Rate: 0.005, Batch Size: 16
Compiling Neural Network...
Finished Compiling, Fitting Neural Network Model...
```

### 1.12 After finishing the predictions, we zip the Figures folder and provide a download link.

---

[ ]:
```
if isUsingColab:
    from IPython.display import FileLink
```

```
    # Create a zip archive of the Figures folder
    shutil.make_archive("Figures", 'zip', "Figures")

    # Provide the download link for the zipped figures
    FileLink("Figures.zip")
```