Heart Beat Classification Report

Data Processing

1) The training set has 87554 samples, each made up of 187 data entries and one label. Similarly, the test set has 21892 samples.

   Upon analysis of the provided MITBIH dataset, it as apparent that the class of data labelled 'N' or 'Normal', vastly outweighs other classes in both the training and testing datasets. The 'N' class has more than 70,000 data entries in the training set, while the least common class has below 700 entries. The class distribution in both datasets is identical, and can be seen in Figure 1 and Figure 2.

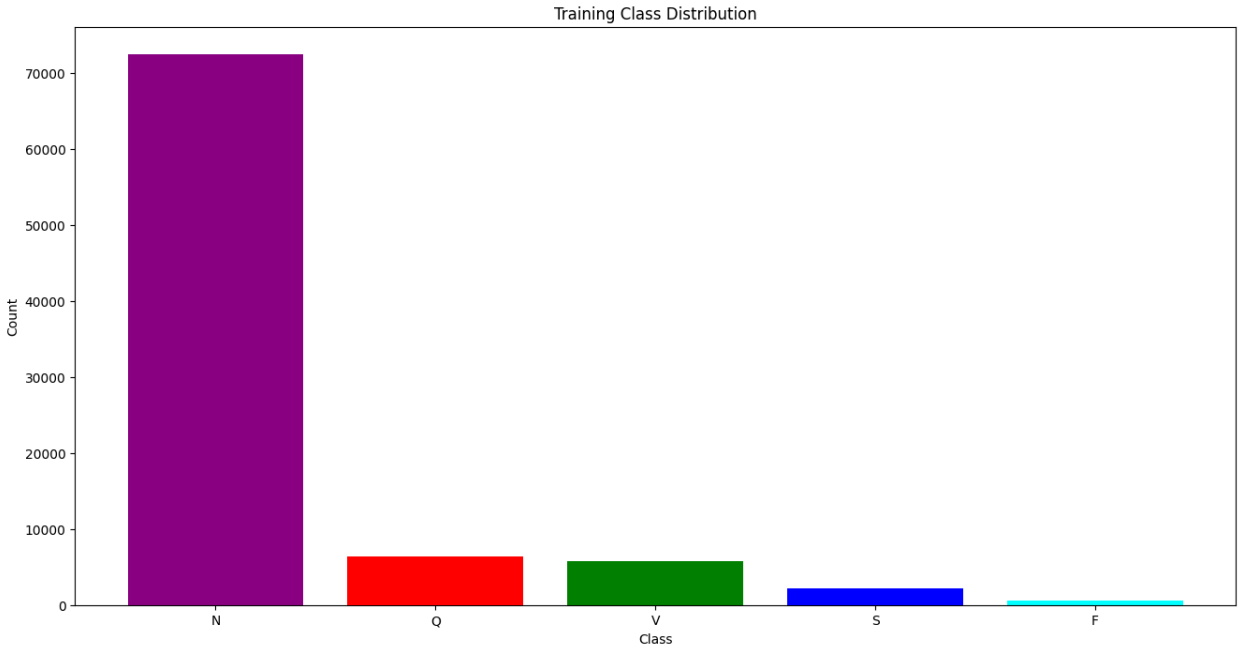

Figure 1: Training Data Distribution

It is important to note that a majority of samples in both datasets are zero padded to achieve a uniform length of 187 data points. Plotting sample signals shows a segmented section of the raw ECG signal, containing a normalized amplitude heartbeat, followed by a 0 amplitude signal courtesy of padding. To an untrained eye, signal appearance across classes not labelled 'normal', seems hard to separate, as well as varied. Identifying temporal as well as frequency-domain features requires further domain-specific research. Sample signals from both datasets can be found in Figure 3 and Figure 4.

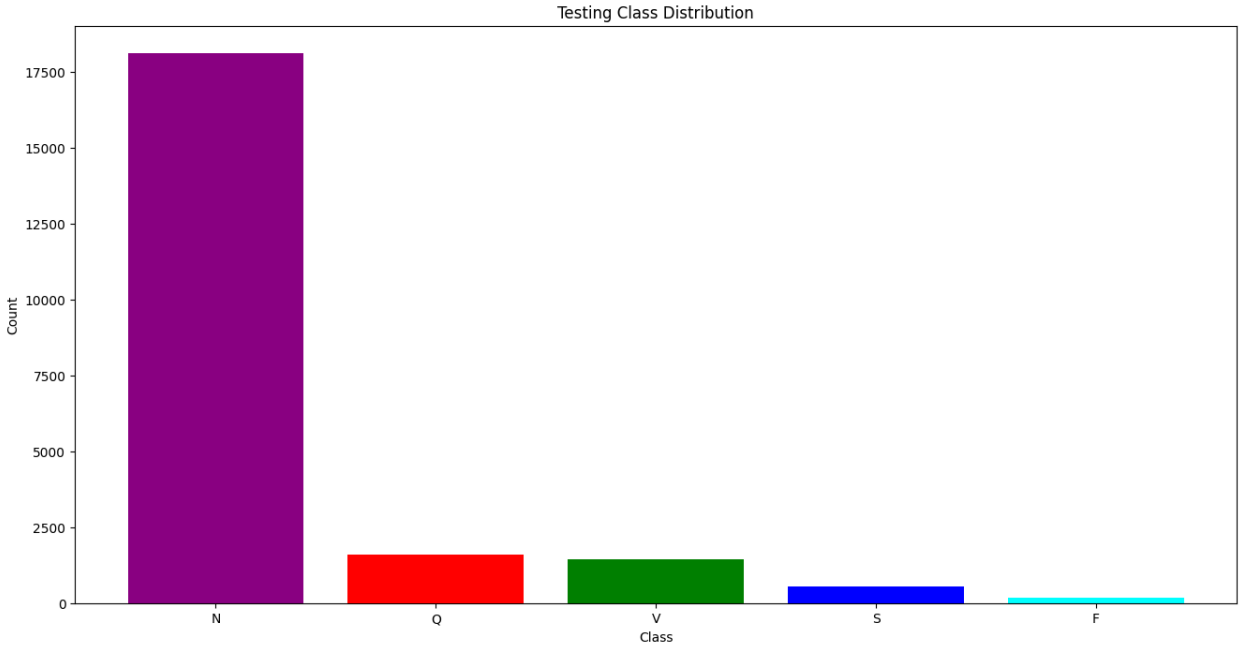

Figure 2: Testing Data Distribution

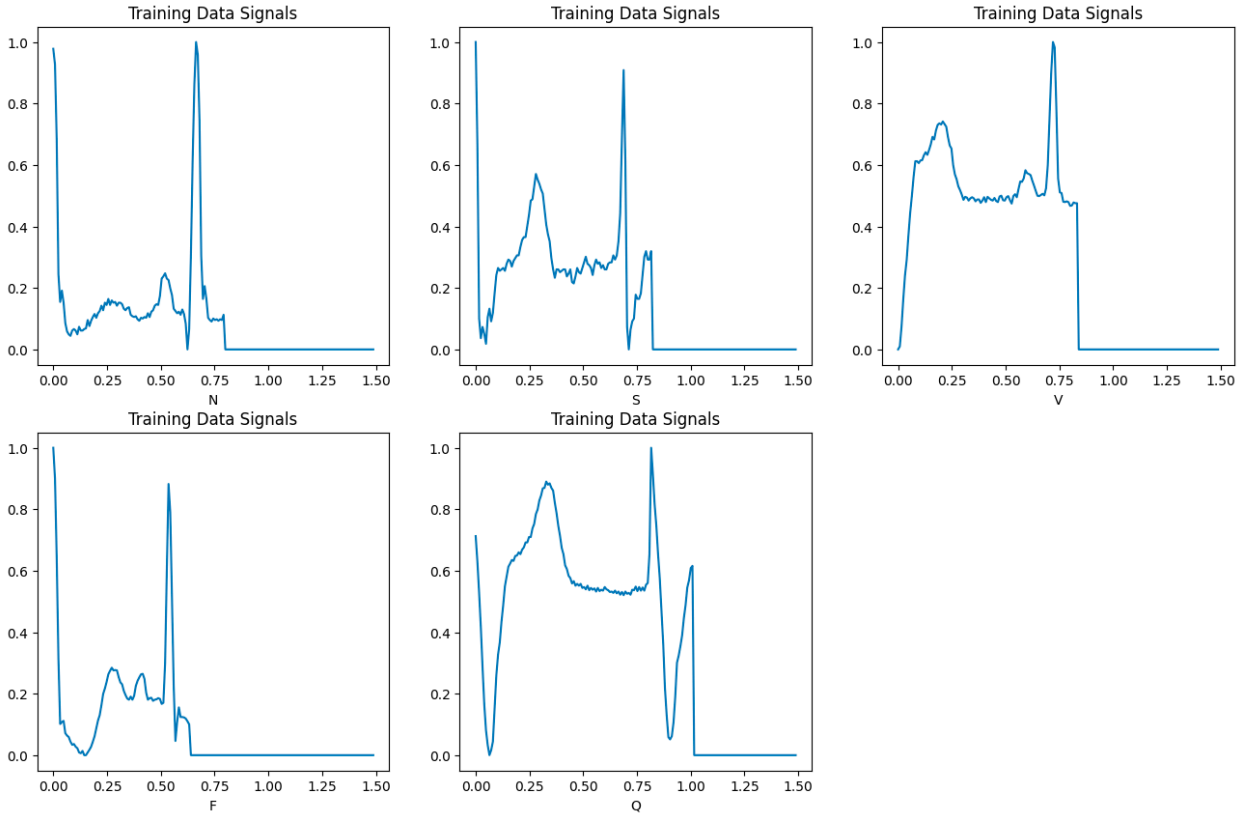

Figure 3: Training Data Signal Visualization

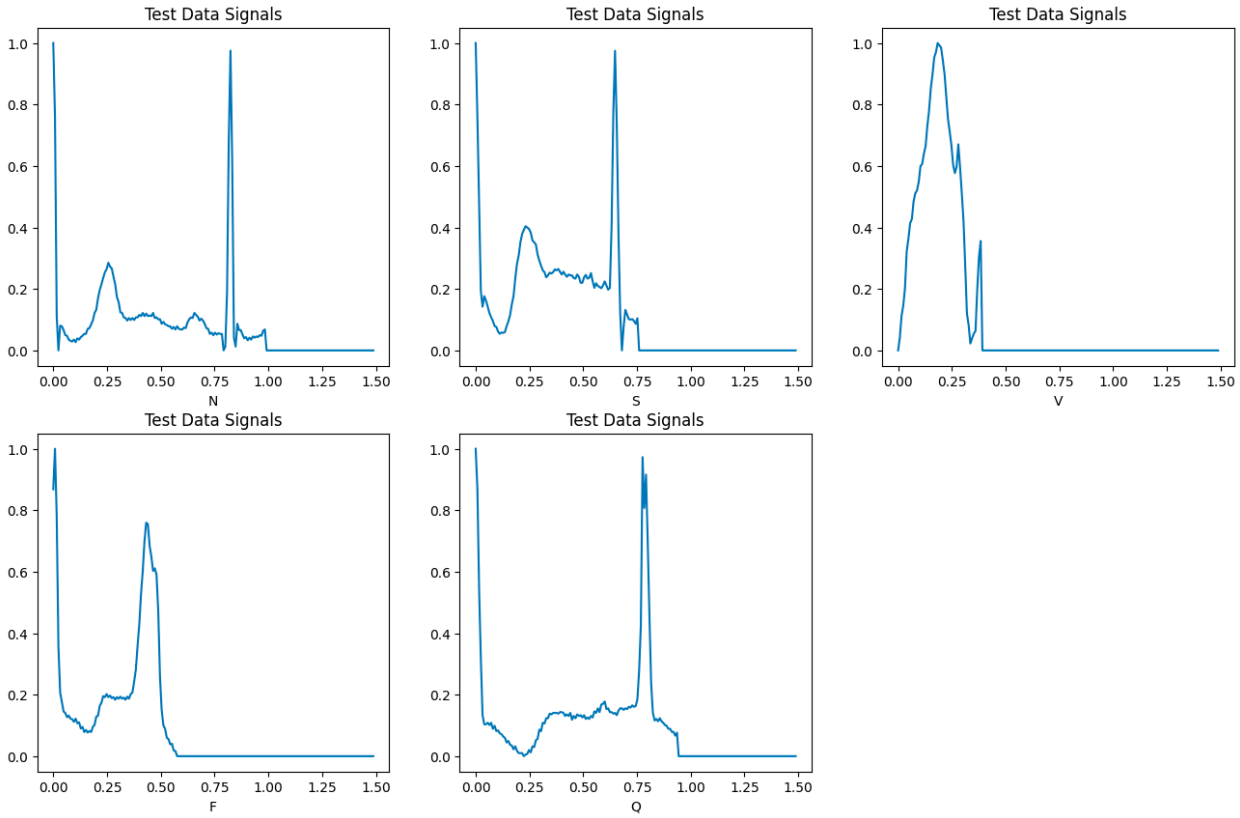

Figure 4: Test Data Signal Visualization

2) Unbalanced datasets lead to skewed predictions, with the dominant class overwhelming predictions. I chose to resample the data to combat this issue.

First, I split the training dataset into a training and validation set. It is important for the validation set to have the same distribution as the initial data, therefore, I did not want to manipulate it with resampling

Next, I chose to upsample less frequent classes, and downsample the dominant class, to achieve a balanced distribution across all classes. I choose to have 10,000 samples from each class, making the dataset a total of 50,000 samples. This method does exclude a lot of the data from the dominant class however, in my opinion, upsampling the less frequent classes to a higher number (to maintain equal proportion) would have caused severe overfitting in my final model.

Following the resampling, because upsampling of less frequent classes often leads to the same example being seen multiple times in the dataset, overfitting becomes a big concern. Anticipating this issue, I decided to add a small amount of random gaussian noise to each of my samples - theoretically this should vary identically sampled data slightly, and therefore reduce the chances of overfitting.

Reducing the possibility of overfitting theoretically leads to better model performance on unseen data i.e. the validation and test data, as well as possibly data in production. With overfitting, even with a low training loss and high training accuracy/confusion matrix performance, model performance on unseen data tends to be very low as the model merely memorizes noisy patterns in training data and overfits to it.

3) Based on a quick literature analysis, most traditional ML techniques (Random Forests, Ensemble SVMs), are based on statistical features extracted from raw data. These features are primarily based on 'R-R' peak related measures such as time period, variance, standard deviation, the energy captured by R-R peaks, as well as frequency domain metrics obtained through Fourier Analysis of dominant frequencies.

These techniques however, rely on pre-processing of raw data (such as peak detection), and then extract these characteristics from raw data. The data provided to me for this problem has already been preprocessed and segmented therefore, without additional information regarding these prior steps, I believe I would be making unsubstantiated assumptions by calculating statistical measures in the small segments of data I have access too. Additionally, these techniques require specialized domain knowledge.

Due to the drawbacks and limitations regarding the application of traditional ML techniques for this problem, I have chosen to go with a CNN architecture. Instead of explicit feature engineering and training models on chosen feature, CNNs inherently extract feature maps from the data, through convolutional layers, and then train on these feature maps.

4) I choose to tackle missing data through a simple data frame sweep of 'NaN' values, along with a search of empty rows (all zero samples) in the data frame. These techniques revealed no missing data in either dataset.

I chose not to shorten the signal, considering most samples were at least 30% padded with zeros. I made this decision because I came across a non-negligible number of samples that did actually have non-zero values throughout the signal and was not sure if this could be an indicator for a particular class or underlying characteristic.

In general, the impact of missing data on model performance depends on various factors, including the amount and pattern of missing data, the specific modeling technique used, and the nature of the data itself.

Non-random missing data might introduce a bias during model training. Additionally lesser training data also contribute to a further loss of information due to fewer samples being available to train on. Additionally, missing data can cause unstable training on models that rely on dense representations.

Likewise, incorrect imputation techniques tend to introduce a noise as well as bias into the model, and could possibly lead to overfitting.

Model Training and Fine-Tuning

1) I chose a CNN architecture for my model. The architecture was inspired by the following paper: https://arxiv.org/pdf/1805.00794.pdf. In addition to the baseline architecture, I added a dew batch normalization and dropout layers, to aid with easier training and regularization.

I chose this architecture as CNNs lend themselves well to feature map extraction. CNNs excel at capturing local patterns due to their inherent convolutional layers, which enable the

automatic extraction of meaningful features from the input signals. Deep CNN architectures have the ability to learn hierarchical representations of the input data, capturing both low-level and high-level features

CNNs can also be easily scaled to deeper architectures by stacking multiple convolutional layers and incorporating pooling layers, due to parameter sharing across convolutional filters, as well as reduced computation due to pooling layers. This property reduces the number of parameters to learn, making CNNs more efficient and effective for tasks with limited data, such as ECG heartbeat categorization.

However, CNN's also face their own challenges. Limited data can potentially affect the generalization capabilities of CNN models. Deep learning architectures perform better with a large quantity of well-balanced training data. Data augmentation techniques, as discussed earlier, can help address this challenge to some extent.

Additionally, CNNs are known for their black-box nature, making it difficult to interpret the decision-making process. Interpreting the learned representations and understanding the reasoning behind the model's predictions can be crucial in medical applications.

2) I tuned the following hyper parameters in my model, through varied methods:

a) Learning rate
b) Batch size
c) Dropout probability as well as amount of dropout layers
d) epochs

Due to time constraints, along with lack of computational resources, I only conducted automated fine-tuning on batch size and learning rates using the Weights and Biases 'Sweep' tool, carrying out 30 training runs. Dropout and epoch fine-tuning was conducted empirically, by manually changing parameters. Each of these hyper parameters were evaluated on validation loss

Dropout: I manually changed the model architecture to asses the best placement of dropout layers. I maintained dropout for each of the fully connected layers, while changing the use of dropout on convolutional layers. I experimented with using dropout on both convolutional layers of the residual block, as well as just the second convolutional layer. Having dropout on both layers resulted in sever degradation of validation as well as training performance, possible due to underfitting. Additionally, using a dropout value of more than $P = 0.2$ on the convolutional layers also led to underfitting. As such, I chose to use a value of $P = 0.2$ and apply dropout to just the second convolutional layer in the residual block. I used the theoretically optimal $P = 0.5$ for dropout in the fully-connected layers.

Epochs: Due to computational and time constraints, I only train up to 100 epochs on two runs, and trained up to 50 epochs when using the Weights and Biases Sweep tool. The tool suggested however, that an increased number of epochs indicated better validation performance.

Learning rate and Batch size: I used automated hyper parameter fine-tuning for these through the Weights and Biases Sweep tool. I limited the learning rate sweep between the theoretical optimum of 0.001 (mentioned in the paper mentioned earlier) to 0.1. I did not include lower values because even at a learning rate of 0.001, loss values start decreasing slowly after about 20 epochs.

For batch size I used sweep parameters of 16, 32 and 64. I constrained the sweep to these values due to computational constraints.

The Weights and Biases Sweep tool indicated that the parameter that seemed to most affect validation loss is the number of epochs for which the model is run. This is followed by batch size, with higher batch sizes yielding better validation loss performance. Finally, Learning rate was deemed least important of the chosen hyperparameters, as long as it was constrained to a value below 0.01. A visualization of the Weights and Biases Sweep can be seen using the following link: Batch size vs Lr (effect on validation loss): https://wandb.ai/sdhaka/lightning_logs/reports/-23-06-02-12-35-20---Vmlldzo0NTQzNzQy. This sweep can also be viewed in the figure below.

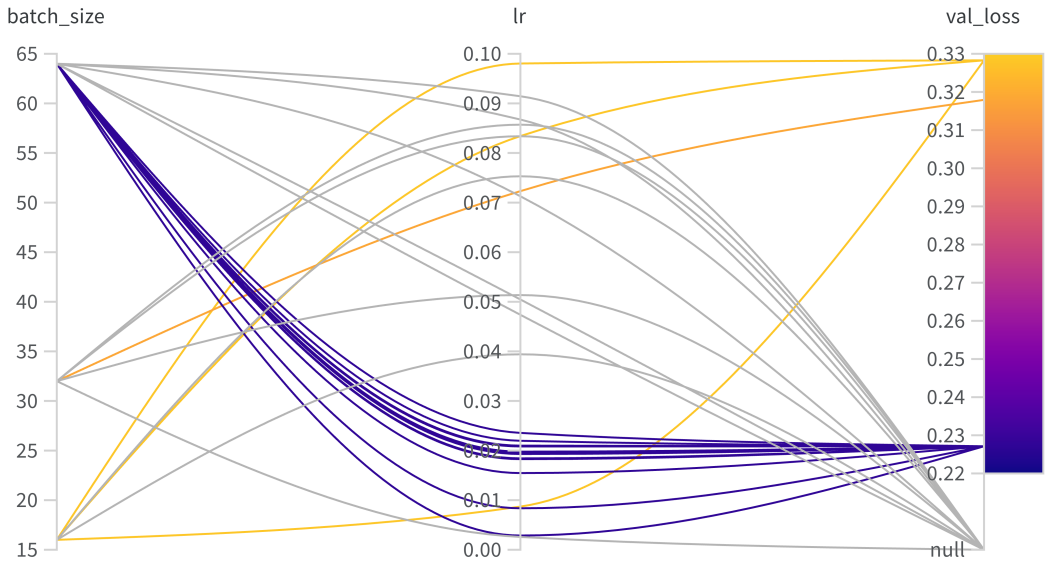

Figure 5: A sweep of 30 runs, comparing effects of various values of batch size and learning rate on validation loss

In summary, it seems that a higher number of epochs, along with a larger batch size and a learning rate below 0.01 is optimum to train this model. With more computational resources and time, I would ideally train the model for longer and on larger batch sizes.

Furthermore, beta values for the Adam optimizer can also be fine-tuned. I empirically observed better performance using higher beta values for a short training run that I executed.

Additionally, there is also potential to fine-tune the amount of Gaussian noise that is added to the training data upon resampling.

3) To address overfitting, I have incorporated the following strategies:

a) Noise addition to resampled data: Upon resampling, some rare classes containing only 650 samples are upsampled to 10,000 samples. Because upsampling is with replacement, the same data points are seen by the model several times, leading to overfitting to those data

points. To solve this, I decided to randomly add gaussian noise to each sample, so their would be some natural variance even if the same data point is upsampled multiple times.

b) Dropout: I have used a dropout layer with P = 0.2 for convolutional layers and P = 0.5 for fully connected layers. Dropout randomly 'switches off' certain nodes in fully connected layers and turns off activations in convolutional layers. It is believed that this reduces interdependencies between nodes, as well as redundancies, leading to regularization of the model, which reduces overfitting

c) Early Stopping: I will cover this in the next section

d) Data Resampling: Because the training data was heavily imbalances, training on this data would cause the model to overfit and skew predictions towards the dominant class. To counter this, I decided to resample data in order to train on a balanced dataset.

In order to address undercutting, I employed the following strategies:

a) Using a Deep CNN: Deep NN often yield feature rich maps that are able to capture both local and global interdependencies of data, and offer an architecture that better captures the complexity of data, when compared to traditional ML models.

b) Using a Residual Block: Residual blocks aid with combatting issues related to vanishing gradients. The skip connections in residual blocks are effectively able to propagate gradients from earlier layers in the network, enabling them to have an impact in gradient updates that lead to the network weights being updated.

4) Early stopping is usually implemented based on a validation metric. It helps prevent overfitting as it stops model training if a certain validation metric does not improve over a defined period of time (patience). It is possible or training metrics to continuously improve, without affecting, and possibly degrading validation metrics. This is the classic sign of overfitting, and early stopping helps combat this issue.

Callbacks in general as useful for model optimization. Several callbacks can be defined, such as model checkpointing based off of best validation performance.

5) During training, the model was evaluated based on evaluation loss. It was important to monitor the evaluation loss, to see if it deviated from the training loss, thereby indicating an overfitting scenario. A learning rate of 0.002 was chosen with a 64 batch size, for 50 epochs.

In terms of performance metrics, accuracy, precision, recall, and F1 score were considered on the test set. A Classification report as well as a confusion matrix was computed based on the predictions in the test set. The classification report is shown below:

Classification Report:

	precision	**recall**	**f1-score**	**support**
0	**0.99**	**0.89**	**0.94**	**18118**
1	**0.34**	**0.83**	**0.48**	**556**
2	**0.86**	**0.92**	**0.89**	**1448**
3	**0.16**	**0.93**	**0.27**	**162**
4	**0.94**	**0.98**	**0.96**	**1608**

It is integral not to solely rely on accuracy metrics on unbalanced datasets. A model could potentially predict all values from the dominant class and still yield a high accuracy on an unbalanced dataset. The accuracy for my model on the test set was **90.1%**.

In my case, the Classification Report results show that the model had a high recall for all classes - the model correctly captured positive occurrences of all classes in the test data.

However, it seems that less frequent classes had a low precision value. This again is an indication of a difference of distribution in training and testing data. The test data was resampled to balance all classes, leading to higher recall performance. The test data however, is severely imbalanced, and is mostly comprised of the 'N' class. Although the model correctly classifies positive occurrences of classes in the test data, it labels a lot of false positives for the less frequent classes.

These low precision errors can be reduced by ensuring a similar data distribution between training and test data. Making the training and test data however will likely reduce recall values and therefore, care must be taken into correctly tuning these tradeoffs. In a medical setting, it is generally beneficial to have high recall values, as to not miss any false negatives.

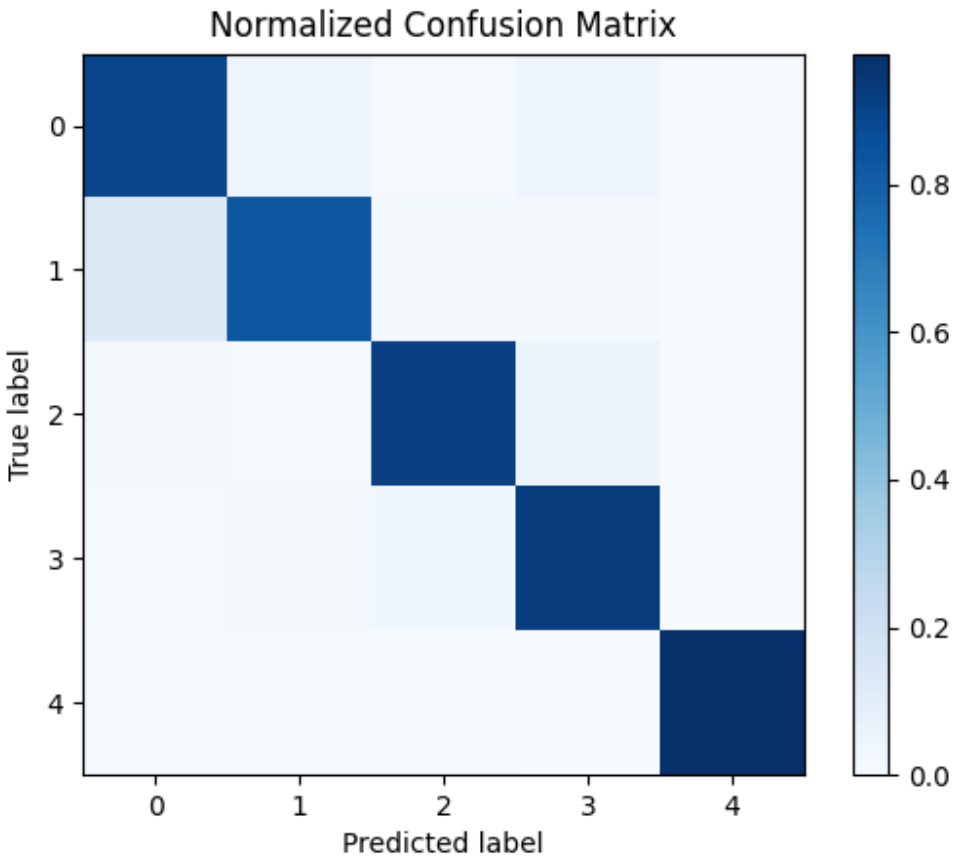

Figure 6: Test data confusion matrix

Deployment Strategies

1) Deep Learning models have intensive computational requirements, often requiring multiple GPU usage. This infrastructure would be my primary concern during deployment. I would suggest use of cloud services such as AWS EC2 instances to both train, and deploy the model in a distributed setting. It is important. To handle concurrent requests, the nodes on which the model is deployed need to have auto-scaling and load-balancing capabilities, a service offered by AWS Sagemaker.

   For implementation, the model first needs to be serialized and saved to disk - the model would be served client-side via Flask, for inference purposes.

   The model pipeline should be uploaded to the cloud as a containerized package (Docker). The deployment setup should include necessary data preprocessing steps used during training. This may include data normalization, resizing/resampling of input data, and handling missing or corrupted data. It is important to implement proper input handling and validation to handle different input formats, handle edge cases, and provide meaningful error messages to users if the input data is invalid.

   Privacy considerations need to be taken into account, especially if the data includes sensitive or personally identifiable information (PII), which is common when dealing with healthcare data.

   The deployment process should be automated and well-documented, including dependencies, environment setup, and configuration details, and using Makefiles for automated deployment.

   Finally, a decision needs to be made on whether to configure the model for offline or online training. In this specific example, offline training might be beneficial as it allows for human verification of input data, and requires an intensive labelling process. Additionally, offline model deployments are often cheaper, with only model inference being served via Flask API.

2) It is integral to implement monitoring mechanisms to track the performance of a deployed model in production. Metrics such as inference latency, throughput, error rates, and model drift need to be monitored to ensure the model continues to perform well over time.

Inference latency, such as the time it takes for a request to be processed and the number of requests the model can handle per unit of time, needs to be monitored to ensure performance expectations. This can easily be done using a Grafana or Prometheus dashboard. Furthermore, synthetic loads can be applied to the containerized service using Gatling, to simulate deployment loads. Without such monitoring, it is possible for out distributed system to become unstable during periods of heavy traffic

Logging and error tracking is automatically set up if the model is deployed on the cloud using Kubernetes for orchestration, to capture any issues or errors that occur during inference. This information can help in debugging. Furthermore, error tracking in terms of model performance also should be consistently logged for analysis, through tools such as wandb, MLflow etc.

Additionally, data as well as model drift needs to be monitored. Data drift might result in changes in the input data distribution over time. Data drift can impact model performance and may require model retraining or updates to pre-processing steps such as resampling. Data drift is often followed by model drift, which results in model performance degrading over time due

to data assumptions being rendered false as the data changes over time. This can be overcome by regularly analyzing statistical properties of input data, especially class imbalances.

In terms of model versioning, tools such as Git and MLflow support model lifecycle versioning. This allows for tracking of different iterations of the model and rollback to previous versions if needed.