**Project Documentation**

**1.Data Collection:**

**Approach Used:**

For our project, Match Prediction, we chose to collect synthetic data. This means we created our own dataset instead of using existing real-world data. Even though it's synthetic, we designed it to reflect real scenarios as closely as possible.

To do this, we created a Google Form with multiple-choice questions related to match outcomes. The questions were based on what users might actually input in a real system, and the answer options matched the kind of labels we wanted the model to learn from.

**Challenges Faced:**

• Low Response Count:

We were expecting around 300 to 500 responses, which would have given us more data to train the model.

However, we only received about 140 responses.

• Less Training Data:

Because of the smaller dataset, we had to be more careful with how we used the data. We focused on keeping the questions clear and the labels balanced so that the model could still learn effectively.

**2.Data Labelling:**

For labelling the data in our Match Prediction project, we used an unsupervised learning approach.

**Why Unsupervised Learning?**

We chose unsupervised learning because it allowed us to automatically find patterns or groupings in the data without needing manually written rules. In our case, a rule-based method wouldn’t be effective since it depends on fixed, human-defined logic—which can be limiting and may not work well with the kind of data we collected.

**3.Label Encoding:**

To convert our labels into numbers that the model can understand, we used Label Encoder.

**Why Label Encoder?**

Since we had a small number of data points, using Label Encoder was the simplest and most efficient choice. It assigns a unique number to each label, which works well when the dataset is small and the labels are not too complex.

If we had a larger dataset (like nominal or ordinal data), we would have considered using other methods like One-Hot Encoding or Ordinal Encoding depending on the label type.

**4.Model Architecture:**

We used an Artificial Neural Network (ANN) for our model architecture.

Initial Design

• The model started with Input layer, 2 hidden layers and output layer

• Activation functions and dense layers were used for basic learning.

• However, due to our limited and unbalanced dataset, we quickly faced overfitting.

How We Improved It

To fix these issues, we made several improvements:

• L2 Regularization: To reduce overfitting by penalizing large weights.

• Kernel Initializer: Helped in better weight initialization to stabilize learning.

• Batch Normalization: Improved training speed and stability.

• Dropout Layers: Randomly dropped neurons during training to prevent the model from becoming too dependent on specific paths.

**5.Saving the Model:**

After training the model, it was important to save the best version for future use—especially for making predictions later.

We used callbacks during training, the Model Checkpoint callback from Keras. This helped us automatically save the model whenever it performed better on the validation data.

Instead of just saving the last model (which might not be the best), we set the callback to monitor validation accuracy—so the model with the highest validation accuracy was saved.

**6.Inference Script:**

The main goal of inference script is to measure how much time a pre-trained model takes to make predictions on new input data.

• We used Python’s argparse module to allow users to pass arguments from the terminal. First, we created the parser object (parser = argparse.ArgumentParser() ).

• Then we added the arguments –weigths\_path (path to the saved model file), --data\_path (path to the data file) and –num\_preds (number of predictions to make).

• For each argument includes required = True (Makes the argument mandatory), type (Specifies the data type (e.g., str, int)), default is given for only the data path and help (Describes what the argument is for).

• We used the following commands to load the saved model (model = tensorflow.keras.models.load\_model(weights\_path) ).

• To measure how long the model takes to generate predictions, we used python ‘s built in time module (import time start = time.time()

predictions = model.predict(data) end = time.time() print(f"Prediction time: {end - start:.4f} seconds") ).

**7. Summary Report**

I performed a series of training experiments using an Artificial Neural Network (ANN) and optimized it using Optuna with three different search strategies: Grid Search, Random Search, and BayesianOptimization. The goal was to determine the best hyperparameters for predicting match outcomes with high precision and generalization.

1. Best Trial Summary (Grid Search)

Number of Hidden Layers: 2 Layer

Configurations:

Layer 1: 12 units, PReLU

Layer 2: 12 units, tanh

Output Layer: 1 unit, sigmoid Optimizer: SGD

Performance (Best Trial)

Final Training Accuracy: 96.50%

Final Training Loss: 0.1510

Final Validation Accuracy: 90.00%

Final Validation Loss: 0.2342

Precision: 0.9225

Recall: 0.9943

Val Precision: 1.0000

Val Recall: 0.7000

1. Best Trial Summary (Random Search)

Number of Hidden Layers: 1 Layer

Configurations:

Layer 1: 15 units, tanh

Output Layer: 1 unit, sigmoid Optimizer: Adam

Performance (Best Trial)

Final Training Accuracy: 98.06%

Final Training Loss: 0.1183

Final Validation Accuracy: 93.33%

Final Validation Loss: 0.2543

Precision: 0.9507

Recall: 1.0000

Val Precision: 1.0000

Val Recall: 0.8000

1. Best Trial Summary (Bayesian Optimization)

Number of Hidden Layers: 2 Layer

Configurations:

Layer 1: 8 units, PReLU

Layer 2: 8 units, tanh

Output Layer: 1 unit, sigmoid Optimizer: Adam

Performance (Best Trial)

Final Training Accuracy: 98.35%

Final Training Loss: 0.1468

Final Validation Accuracy: 93.33%

Final Validation Loss: 0.2549

Precision: 0.9616

Recall: 1.0000

Val Precision: 1.0000

Val Recall: 0.8000