Project Name: Medical Appointment No Shows

Sajjad Zangiabadi

Introduction

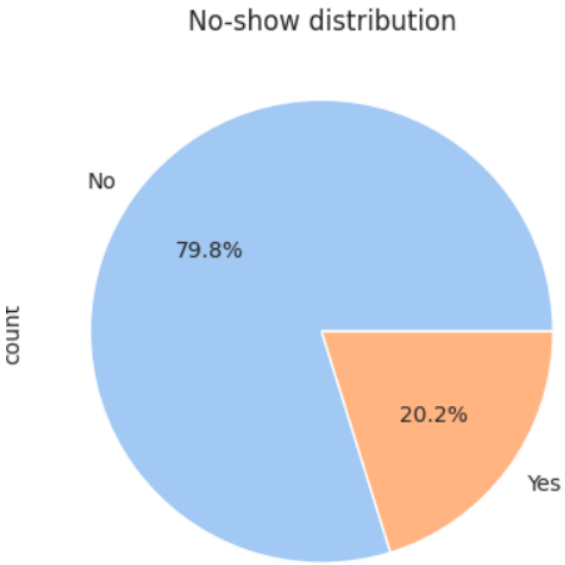
- This report analyzes a medical appointment dataset from Kaggle to understand why patients miss scheduled appointments (no-shows). The dataset contains information on over 110,000 appointments with 14 associated variables, including patient characteristics, health conditions, and communication details. By analyzing these factors, we can identify patterns and develop strategies to reduce no-show rates.

Data Description:

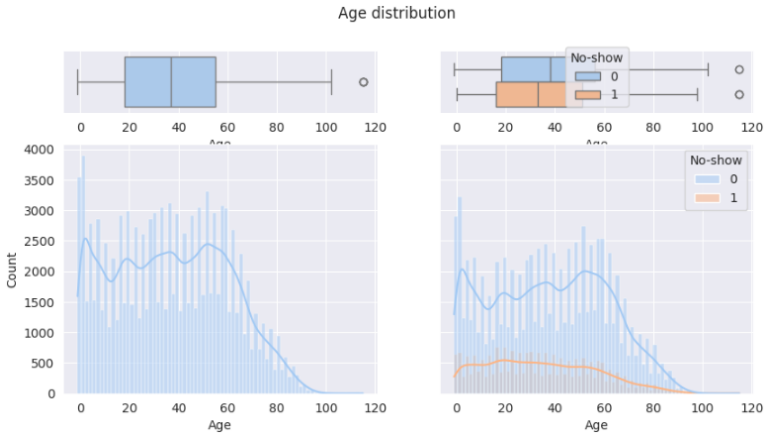
- The dataset includes the following features:
  - **Patient ID**: Unique identifier for each patient
  - **Appointment ID**: Unique identifier for each appointment
  - **Gender**: Male or Female
  - **Appointment Date**: Date of the scheduled appointment
  - **Scheduling Date**: Date the appointment was scheduled.
  - **Age**: Patient's age
  - **Neighborhood**: Area where the appointment takes place
  - **Scholarship**: Whether the patient receives social benefits.
  - **Hypertension**: Whether the patient has hypertension (high blood pressure)
  - **Diabetes**: Whether the patient has diabetes
  - **Alcoholism**: Whether the patient has alcoholism
  - **Handicap**: Whether the patient has a disability
  - **SMS Received**: Indicates if the patient received appointment reminders via SMS.
  - **No-Show**: Indicates whether the patient did not show up for the appointment or attended

- **No-show:**

The No-show target is imbalanced, with approximately 80% of appointments marked as "No" and 20% marked as "Yes." This imbalance is important to consider when analyzing the data and building predictive models.
This imbalance can pose challenges for some algorithms, making it more difficult to learn patterns accurately from the minority class. X

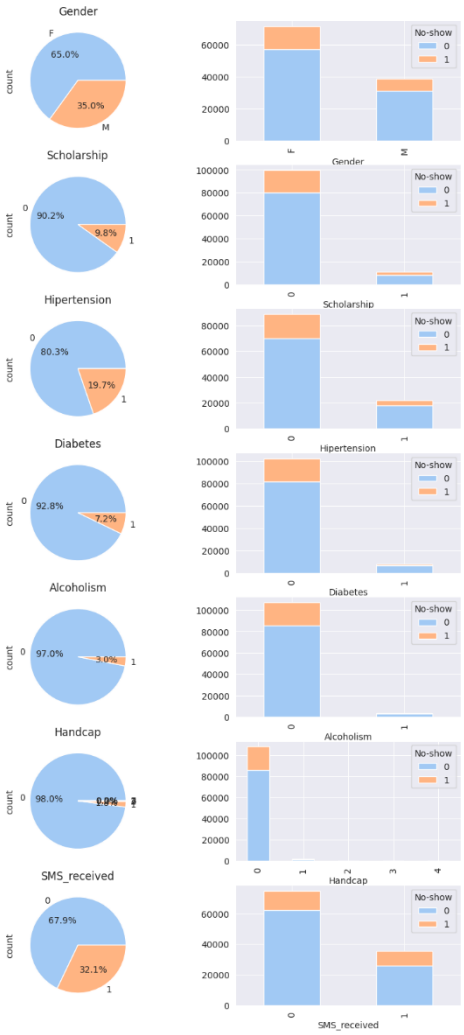


- **Age**



This visual analysis explores the potential correlation between patient age and appointment attendance. The boxplots (Figures 1 & 2) provide a comparative overview of the age distribution for the entire patient population and segmented by attendance status (attended vs. missed appointments). The histograms (Figures 3 & 4) further illustrate the age distribution, with a specific focus on differentiating the distribution between attending and non-attending patients. Notably, the boxplots reveal a potentially significant finding: the median age of patients who missed appointments may be lower compared to those who attended.

- **Other features:**



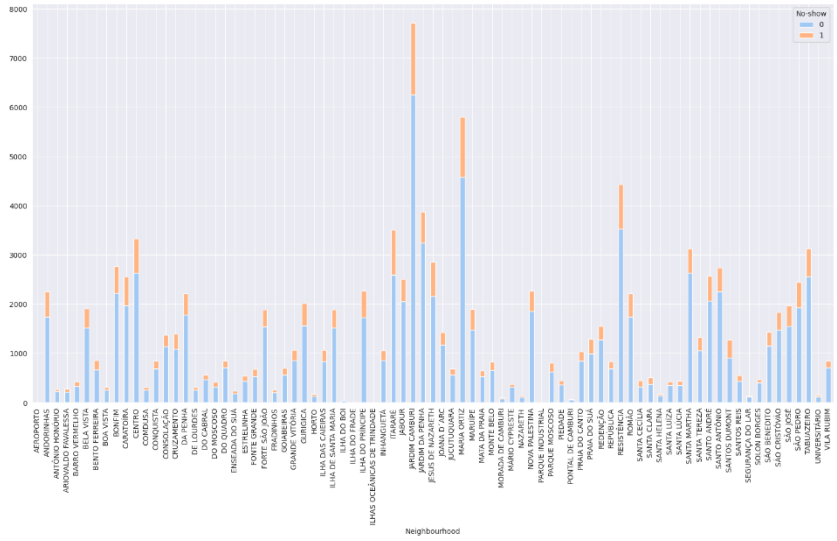
Pie Charts (Left Column):

Each pie chart represents the distribution of a single feature (e.g., Gender, Scholarship).
Slice sizes and labels indicate the percentage of data points belonging to each category within that feature.

Stacked Bar Charts (Right Column):

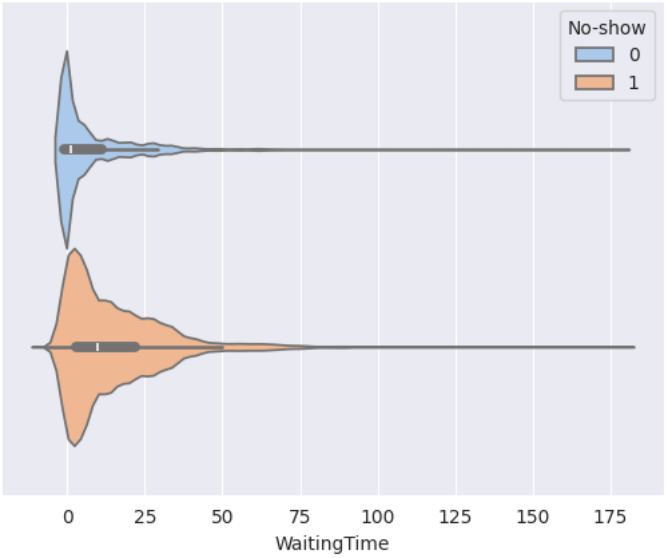
These charts visualize how the categories of each feature break down in relation to the target variable ("No-show").
The x-axis represents the categories of the feature being analyzed. The y-axis shows the total count of data points. Each bar is stacked, with segments representing the number of patients who did or did not show up for appointments within each feature category.

The x-axis represents the different neighborhood categories present in your data.
Due to the large number of categories.
The y-axis shows the total count of data points.
Each bar is stacked, with segments representing the number of patients who, did or did not show up for appointments within each neighborhood category.



- **WaintingTime:**
  - first converts both "ScheduledDay" and "AppointmentDay" columns into datetime,
  - A new feature, "WaitingTime", is calculated by subtracting the "ScheduledDay" from the "AppointmentDay" and converting the difference to the number of days.
  - And then “ScheduledDay” and “AppointmentDay” are dropped.

The visualization shows how long people waited for appointments. The x-axis shows the waiting time, and the y-axis shows how many people waited that long. Interestingly, people who didn't attend (no-shows) had a significantly longer median wait time than those who did attend.



Preprocessing:

- **Train-Test Split:**
    - The initial split separates the data into two primary sets: training (train_df) and testing (test_df). This is achieved using the train_test_split function from the scikit-learn library.
        1. **Training Set (80%)**: This larger portion of the data is used to train the machine learning model. The model learns patterns and relationships within the training data to make predictions.
        2. **Testing Set (20%)**: This unseen data is used to evaluate the model's performance on data it hasn't encountered during training. This helps assess the model's generalizability and robustness.
    - **Stratified Splitting on "No-show" Class:** An important aspect of the split process is the use of stratification based on the "No-show" feature. Stratification ensures that the proportion of classes (e.g., appointments with and without a no-show) is preserved in both training and testing sets. This is crucial when dealing with imbalanced datasets, where one class might have significantly fewer instances than others. By stratifying the split, the model is exposed to a representative sample of the classes during training, leading to more reliable performance evaluation.
    - **Train-Validation Split (Optional):** The second train_test_split function creates an additional split within the training data, resulting in training (train_df) and validation (dev_df) sets
        1. **Training Set (Further Split):** This subset is used to train the model.
        2. **Validation Set:** This unseen portion of the training data is used to fine-tune the model's hyperparameters.
        3. **So it's advantageous:**
            1. **Improved Model Tuning**: The validation set provides a held-out set of data specifically for fine-tuning the model's hyperparameters. Hyperparameters are settings that control the model's learning process, and the validation set helps us identify the configuration that leads to the best performance on unseen data. By iteratively training the model on the training set and evaluating it on the validation set, we can optimize the hyperparameters without overfitting to the training data.
            2. **Early Stopping**: The validation set allows for early stopping. During training, the model's performance on the validation set is monitored. If the performance stops improving or starts to decline, it indicates overfitting. Early stopping helps us halt the training process at the point where the model generalizes best to unseen data, preventing overfitting to the training data.
- **Encoding "Neighbourhood":**
    - This step encodes the categorical "Neighbourhood" feature for all datasets (training, validation, testing). Encoding transforms neighborhood text labels into numerical values for machine learning models. A dictionary maps unique neighborhoods to indices (based on frequency in training data) and is applied consistently across all sets.
- **RobustScaling of "Age"**
    - This step applies a RobustScaler to the "Age" feature. Feature scaling normalizes data, which is important for machine learning models.
    - RobustScaler is a good choice here because:

- **Handles outliers**: It uses medians and IQRs, making it less sensitive to outliers that can distort traditional scaling methods.
- **Preserves importance**: It maintains the relative importance of features, even those with outliers.

- **Log Transformation of "WaitingTime"**
  - This step transforms "WaitingTime" using log1p after RobustScaler scaling. It addresses positive skew in waiting times, a common issue that can hinder machine learning models. Log transformation compresses longer times and preserves zeros, making the distribution more symmetrical and suitable for modeling. The combination of RobustScaler and log1p improves data normality and potentially leads to better model performance.
- **Split to X, y**

Finally, we have 12 features and 1 target, and there is 70736, 17685 and 22106 samples for train, dev and test sets in order.

Modeling (Traditional ML):

- **SMOTE (Synthetic Minority Oversampling Technique):**
  - We address the potential issue of class imbalance in the data using SMOTE. This technique oversamples data points from the minority class, creating synthetic instances to balance the representation of each class within the training data. This helps the model learn effectively from the minority class, improving its ability to classify such instances accurately.
- **Cross-Validation and Hyperparameter Tuning**:
  - To prevent overfitting and ensure robust model performance, we leverage cross-validation. This technique splits the training data into folds. The model is trained on each fold except one (validation fold) and evaluated on the held-out validation fold. This process is repeated for all folds, providing a more reliable estimate of the model's generalizability. Additionally, a RandomizedSearchCV is implemented within the cross-validation framework to efficiently search through a defined hyperparameter grid for the Random Forest Classifier. This optimizes the model's configuration for the specific dataset, potentially leading to improved performance.
- **These are parameters** of the random forest model with the highest f1-score which is found throughout the cross-validation:

```
{'rf_classifier__n_estimators': 200, 'rf_classifier__min_samples_split': 5, 'rf_classifier__min_samples_leaf': 4
, 'rf_classifier__max_features': 'sqrt', 'rf_classifier__max_depth': 10, 'rf_classifier__bootstrap': True}
```

- **Random Forest:**
  - is an ensemble learning method that constructs a collection of decision trees at training time. Each tree is built on a random subset of features and data points drawn with replacement (bootstrapping) from the original dataset. This process introduces diversity into the ensemble, reducing variance and improving the overall generalization performance of the model compared to a single decision tree.
  - Key Hyperparameters:
    1. **n_estimators:** More trees generally improve performance but increase training time.
    2. **max_depth**: Controls complexity - deeper trees can overfit.
    3. **max_features**: Limits considered features to prevent overfitting.
    4. **min_samples_split, min_samples_leaf**: Avoids splitting nodes with too few data points.
    5. **bootstrap**: Use bagging for more diverse trees.

- Evaluating Model Performance On Test Data: Classification Report
    - The classification_report reports four key metrics:
        1. **Precision**: This metric reflects the proportion of positive predictions that were truly positive. In simpler terms, it tells you how accurate your model is when it predicts a positive class.
        2. **Recall**: Recall focuses on the completeness of the model's positive predictions. It represents the proportion of actual positive cases that were correctly identified by the model. In other words, it tells you how good the model is at finding all the positive examples.
        3. **F1-score**: The F1-score is a harmonic mean between precision and recall. It strikes a balance between the two, providing a single metric to evaluate the model's overall effectiveness. A higher F1-score indicates better overall classification performance.

	precision	recall	f1-score	support
0	0.90	0.60	0.72	17642
1	0.32	0.73	0.44	4464
accuracy			0.63	22106
macro avg	0.61	0.67	0.58	22106
weighted avg	0.78	0.63	0.67	22106

Modeling (Neural Network):

- **Define Custom Dataset:**
    - Class CustomDataset, is designed to prepare a dataset specifically tailored for training a machine learning model in PyTorch. It inherits from the Dataset class provided by PyTorch's torch.utils.data module, ensuring compatibility with the framework's data loading functionalities.

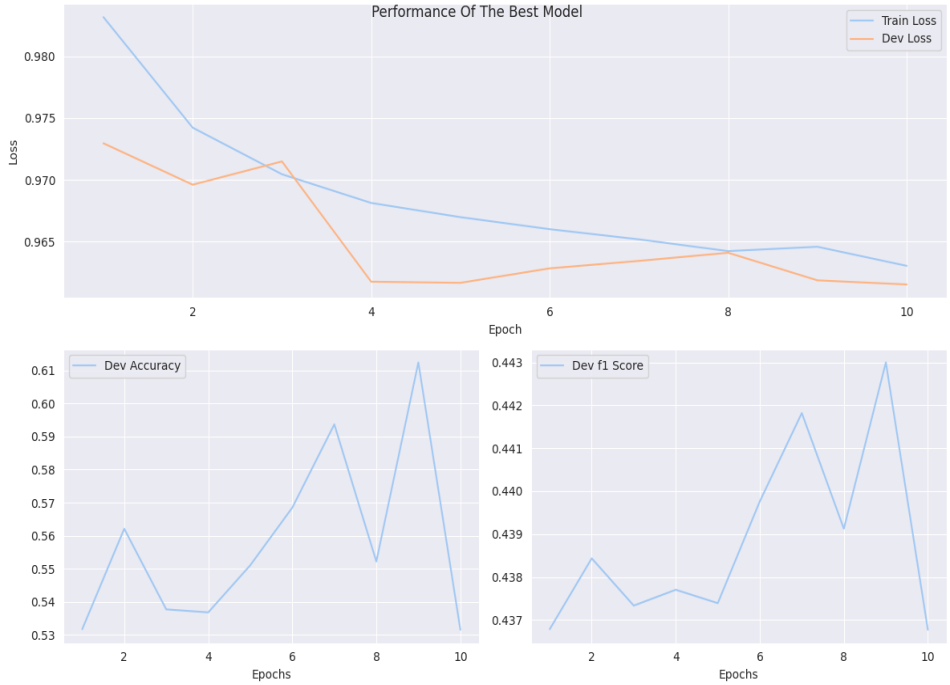
    - Functionality:

        1. Dataset Ingestion: It takes a pandas DataFrame containing your data as input during initialization.
        2. Data Preprocessing: Within the __getitem__ method, it extracts a single sample (data point) at the specified index (idx) from the DataFrame. It then separates the features (X) from the target variable (y) you're aiming to predict, which in this case is the "No-show" label.
        3. Tensor Conversion: The class converts both the features (X) and the target variable (y) into PyTorch tensors. This is crucial for efficient processing within PyTorch models, which operate on tensors. The features are converted to floating-point tensors, while the target is also converted to a float tensor.
        4. Dataset Length: The __len__ method simply returns the total number of samples (rows) in the provided DataFrame, allowing the data loader to iterate through the dataset effectively.

        By encapsulating these operations within a custom dataset class, you make your data preparation code more modular, reusable, and maintainable.

- **Define classifier**:
  - The model's forward function takes input data x and passes it through each layer sequentially with ReLU activations, finally returning a single number as the logit output.
  - **Architecture**:
    1. **Input layer**: Accepts input data with 12 features.
    2. **Hidden layers:** There are four hidden layers with sizes specified by hidd1, hidd2, hidd3, and hidd4. Each hidden layer is followed by a ReLU activation function.
    3. **Output layer**: has an output size of 1. This indicates that the model predicts a single continuous value. (logit)
  - **Logit Output**:
    1. The model does not apply a final activation function like sigmoid or softmax. Instead, it directly returns the output from the last linear layer (x). This output represents the logits, which are the raw, unnormalized scores for each class (in this case, likely a binary classification problem).
- **Define BCEwithLogits Loss**:
  - By returning logits, the model is prepared to work seamlessly with the BCEWithLogitsLoss function in PyTorch. This loss function is specifically designed for binary classification tasks where the output is in the form of logits. It calculates the binary cross-entropy loss directly from the logits, eliminating the need for an additional sigmoid activation to convert the outputs to probabilities.
- **Training and Evaluation over Epochs**:
  - Iterative Training: The train_evaluate_over_epochs function iterates through a defined number of epochs. Within each epoch:
  - **Training Step**:
    1. The train function performs the forward pass on the training data loader (train_data_loader).
    2. It calculates the loss using the provided loss function (loss_fn).
    3. The optimizer (optimizer) updates the model weights based on the calculated loss.
  - **Evaluation Step**:
    1. The evaluate function conducts the forward pass on the validation data loader
    2. It calculates the validation loss and various performance metrics (accuracy, precision, recall, F1-score) using the provided loss function and the true labels.
  - **Metrics Tracking**:
    1. The function maintains a dictionary (metrics_history) to track training and validation losses, along with all the performance metrics across epochs. This allows for analysis of the model's learning behavior.
  - **Best Model Selection**:
    1. The code keeps track of the best model based on the F1-score.
- **Grid Search for Hyperparameter Optimization**:
  - **Hyperparameter Space Definition**: The param_grid dictionary defines the search space for four hyperparameters corresponding to the hidden layer sizes (hidd1 to hidd4) of classifer
  - **Grid Search Execution:**
    1. The code employs a Parallel object from the joblib library to leverage parallel processing for an efficient grid search.
    2. The delayed function from joblib wraps the wrapper function that takes a combination of hyperparameters, etc.
    3. By running these wrapped functions in parallel, the grid search explores all hyperparameter combinations concurrently, significantly reducing the time required compared to a sequential approach and finally the best model is chosen.

- Performance Analysis and Visualization:
    - **Loss Curves (Top Plot):**
        - This plot showcases the training and validation losses across epochs.
        - A downward trend in both lines indicates the model is learning effectively and reducing its loss on both the training and validation datasets.
        - A significant gap between the two lines could suggest overfitting, where the model performs well on the training data but poorly on unseen validation data.
    - **Dev Accuracy (Bottom Left Plot):**
        - This plot monitors the model's accuracy on the validation set throughout training.
        - An increasing trend signifies the model's ability to correctly classify examples from the validation data.
    - **Dev F1-Score (Bottom Right Plot):**
        - This plot tracks the F1-score, a balanced measure of precision and recall, on the validation set.
        - A consistently high F1-score indicates the model achieves good performance in terms of both precision (identifying true positives) and recall (capturing all relevant examples).



This plot is performance of the best model which is chosen throug grid search on train and dev set, and its parameters are:

```
Classifier(
(linear1): Linear(in_features=12, out_features=32, bias=True)
(linear2): Linear(in_features=32, out_features=64, bias=True)
(linear3): Linear(in_features=64, out_features=32, bias=True)
(linear4): Linear(in_features=32, out_features=8, bias=True)
(linear5): Linear(in_features=8, out_features=1, bias=True)
(relu): ReLU()
)
```

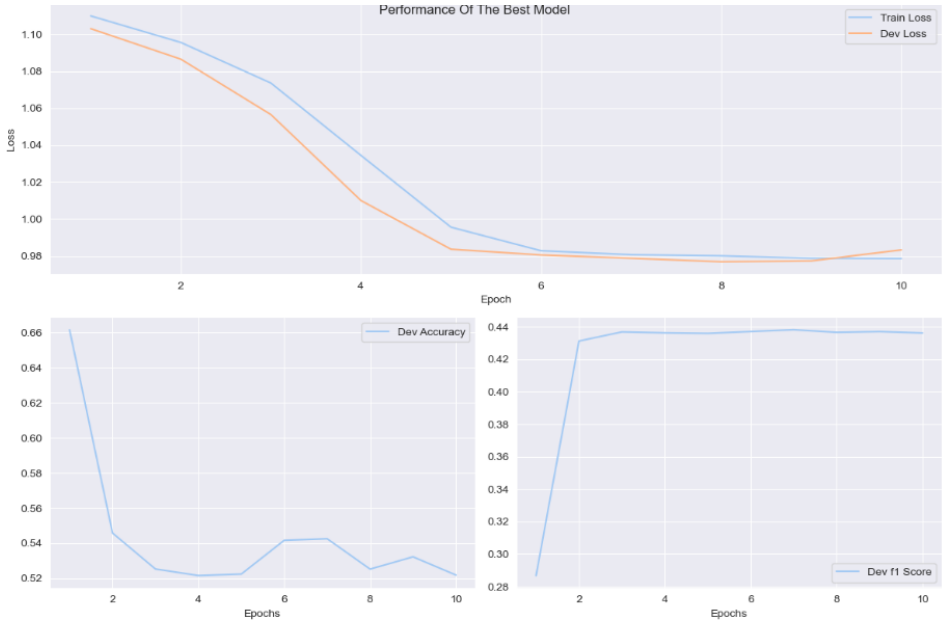
To assess the model's generalizability to real-world scenarios, we utilized a completely unseen test set. The classification report presented below details how effectively the best-performing model performs on this independent dataset, providing insights into its practical applicability.

	precision	recall	f1-score	support
0.0	0.95	0.43	0.59	17642
1.0	0.29	0.91	0.44	4464
accuracy			0.53	22106
macro avg	0.62	0.67	0.51	22106
weighted avg	0.82	0.53	0.56	22106

Comparing:

- **SGD Optimizer**

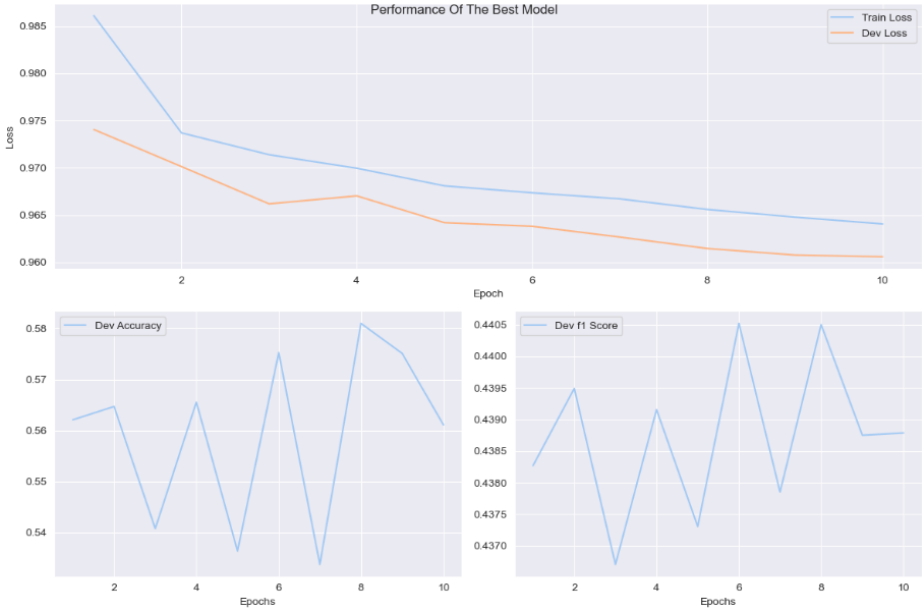
Leveraging the architecture of the best model identified during the grid search, we re-trained it using the SDG optimizer instead of Adam, as see, it converges smoother and final f1 score is 0.93, which is slightly better than performance training model when using Adam.



	precision	recall	f1-score	support
0.0	0.96	0.41	0.58	17642
1.0	0.29	0.93	0.44	4464
accuracy			0.52	22106
macro avg	0.62	0.67	0.51	22106
weighted avg	0.82	0.52	0.55	22106

- **Sigmoid Activation Function:**

Retained the best model's architecture from the grid search but replaced the ReLU activation functions in its hidden layers with Sigmoid functions. This modification resulted in a slower convergence pattern during training compared to the ReLU-based model. Additionally, after running 10 epochs, the final F1-score achieved with Sigmoid (0.86) fell short of the performance obtained using ReLU (0.91).



	precision	recall	f1-score	support
0.0	0.93	0.48	0.64	17642
1.0	0.30	0.86	0.44	4464
accuracy			0.56	22106
macro avg	0.62	0.67	0.54	22106
weighted avg	0.80	0.56	0.60	22106