PREDICTING EMPLOYEE ATTRITION USING TABNET

[1]Evant Valery Wijaya, [2]Shinta Estri Wahyuningrum
[1,2]Program Studi Teknik Informatika Fakultas Ilmu Komputer,
Universitas Katolik Soegijapranata
[1]21k10010@student.unika.ac.id, [2]shinta@unika.ac.id

ABSTRACT

High employee turnover may threaten stability and productive working environment in a company. Not to mention, it is also more costly than retaining existing employees. The key to solve this problem is to predict employee attrition. Most of the previous researches utilized tree-based model such as Random Forest or a simple deep learning model such as Multi-layer Perceptron. This project will include training a TabNet model for the prediction of employee attrition and comparison of its performance concerning metrics such as accuracy, precision, recall, and F1 score against both a Multi-Layer Perceptron model and a Random Forest model. This study anticipated that the TabNet model would produce results comparable to other models; however, TabNet demonstrated lower performance than both the Random Forest and Multi-Layer Perceptron models. Out of all the models, the Random Forest model performs the best in all key metrics, followed closely by the Multi-Layer Perceptron model. The results indicate that the tree-based algorithms seem to be producing better outputs for predicting employee attrition in structured datasets. The findings of this research offer valuable insights for businesses aiming to improve employee retention strategies.

Keywords: Employee, Attrition, TabNet, Multi-Layer Perceptron, Random Forest

INTRODUCTION

Employee attrition refers to the number of employees leaving the organization for any reasons. High turnover threatens the stability and productivity of a company. In addition, high employee attrition is more costly than to keep the existing ones [1]. It may be due to some reasons such as heavy workload, poor salary and an unsuitable working environment, etc [2]. Businesses with the skill to identify these variables and to predict employee attrition are in a position to ensure the retention of their most valuable and experienced workers at the same time as reducing the financial losses.

With the current technology, Artificial Intelligence has allowed us to solve complex predictions, one of which is the attrition of employees [2]. Among Artificial Intelligence techniques, deep learning have the ability to learn and perform effectively on large datasets, often surpassing traditional machine learning models in specific applications [3]. However, most previous researches employed traditional machine learning models such as Random Forest [1], [4], [5], [6], [7], [8], [9], [10] and simpler deep learning models such as Multi-Layer Perceptron [1], [5], [11], but none used TabNet. Given that TabNet is a relatively new algorithm, this study aims to evaluate TabNet's effectiveness in predicting employee attrition and compare its performance to Random Forest and Multi-Layer Perceptron.

The objective of this research is to implement the TabNet algorithm to build a predictive model and compare it with the Multi-Layer Perceptron model and Random Forest model. The aim of this analysis is to assess whether TabNet is better at predicting employee attrition than the Multi-Layer Perceptron model and Random Forest model or not. Additionally, the writer employed several preprocessing steps to enhance the performance of the predictive model.

Problem Formulation

This research aims to answer the following critical questions :

1. Does the TabNet model exhibit superior predictive performance compared to the Multi-Layer Perceptron model and Random Forest model?
2. How does the TabNet model perform in terms of various metrics (such as accuracy, precision, recall, and F1-score) when predicting employee attrition?

Scope

This research uses the "Employee Analysis | Attrition Report" dataset sourced from Kaggle.com to predict employee attrition. The evaluation metrics will include accuracy, precision, recall, and F1 score. Preprocessing steps involve data cleaning, encoding of categorical features using one-hot encoder, rescaling using MinMax Scaling, and balancing the dataset using SMOTE. The models developed are TabNet, Multi-Layer Perceptron, and Random Forest, implemented with PyTorch and Scikit-learn, respectively. GridSearchCV will be used to perform hyperparameter tuning with cross-validation for model validation.

Objective

The objective of this research is to evaluate the test score including accuracy, precision, recall, and F1 score of the TabNet model. These results will then be compared with those of the Multi-Layer Perceptron (MLP) and Random Forest (RF) models, based on each model's best performance on the test set across these metrics.

LITERATURE STUDY

In 2022, Raza et al. [7] implemented a research study to find the causes of employee attrition and a learning framework to predict it. The research used IBM HR Employee Attrition dataset and the authors compared the predictive capability of four machine learning techniques. They have applied Employee Exploratory Data Analysis (EEDA) to identify major causes behind employee attrition as well. While the study is insightful about the preprocessing techniques, it still does not emerge with detailed information of how these key factors might affect attrition of employees. While not forming the foundation of my research, this study will complement my research by offering supplementary information on preprocessing steps.

Faced with a similar challenge, Jain et al. [6] also attempted constructing a model for predicting employee attrition. They believed that measuring employee appraisal and satisfaction within the company helps to reduce attrition rate. They did their research for predicting the employee attrition using Support Vector Machine, Decision Tree and Random Forest by using the

human resource management dataset. However, their study did not perform data balancing. This flaw produced classifier bias as their classifier tends to detect the majority class causing their classifier to have low sensitivity to the minority class. Therefore, this research provides a good example to show why data balancing is so crucial.

Another research on employee attrition was also done by Arqawi et al. [1]. In this study, they used a deep learning model along with the other machine learning models in their research. By using the IBM HR Employee Attrition dataset, they found that Random Forest model outperformed other machine learning models. Random Forest is an ensemble learning method designed for classification and regression tasks. It works by combining multiple decision trees to produce accurate and stable prediction. For regression tasks, the output is the mean of the predictions, while for classification tasks, the result is determined by the majority vote of the trees [9]. However, after comparing the Random Forest model with the deep learning model, the deep learning model produced superior performance compared to Random Forest. This research serves as an additional information to develop my own Random Forest and Multi-Layer Perceptron models for the purpose of comparison with the proposed TabNet model.

Another study employing the same IBM HR Employee Attrition dataset was proposed by Alsubaie and Aldoukhi [10] who also aimed to assist organizations in minimizing attrition and manage human resources by developing precise predictive models. While sharing similar research objectives, they employed different methodologies compared to previous studies. They improved their models' accuracy by using pruning method for the Decision Tree model, the cross-validation method for the Random Forest model and the stepwise method for Binary Linear Regression. They concluded that Binary Linear Regression achieved the highest accuracy after these enhancements. However, I pointed out that there was a weakness to their analysis that involved using accuracy as the sole performance measure. The basic evaluation metrics like accuracy could be complemented with other methods, for instance, precision, recall, and F1-score that would give a better insight into the model performance. Nevertheless, the studies presented provide useful additional information regarding the important variables that impact employee turnover.

In the research conducted by Darraji et al. [2], they also employed deep learning technique to predict employee attrition. In this research they used two versions of the dataset which are, original IBM HR Employee Attrition dataset and the balanced version of it. Their proposed nine layers of deep learning model, achieved impressive accuracies of 91.16% and 94.16% for the unbalanced and balanced dataset, respectively, surpassing the state-of-the-art techniques using the same dataset. However, as informative as their reasoning may have been, they fell short on providing comprehensive explanation of the main causes of attrition. Some of their work, although not directly proposed as relevant to my research, offers valuable additional sources of information.

Previous studies have used only the classification algorithms in their research work while Usha and Balaji [4] took a different approach by using KMeans alongside traditional classification algorithms. This shift from the typical practice is quite fascinating since clustering algorithms are normally not applied in this fashion. Based on their results, they discovered that Naïve Bayes was

more effective than other algorithms while on the other hand KMeans could not benchmark well compared to other classification algorithms' benchmarks in terms of performances. One limitation of this research is that it could be enhanced by finding the key factors contributing to employee attrition. This research led me to a conclusion that clustering algorithms might not be the best fit for tasks like predicting employee attrition.

Another study was also carried out to predict top employees with high retention risk numbers by Qutub et al. [9]. The authors further boosted the model accuracy by using ensemble methods, which were not covered in the previous studies. Using the IBM Attrition Dataset, the authors created just three ensemble models. Surprisingly, the results indicated that Logistic Regression outperformed other algorithms regarding performance. Though they used ensemble methods, the ensemble model's accuracy was relatively good. This, according to the authors, is due to the small size of the dataset that favors the Logistic Regression. Still, in larger unseen datasets, the ensemble methods may be more suitable. The remaining dissatisfaction is that the study examines only some limited combination of models. Also, it does not explain in detail how the selection was done for the features; that is, it does not indicate which attributes were in subsets d2-d5. Despite these shortcomings, the study still provides valuable supplementary information for my research.

Fallucchi et al. [8] set out to determine through their research how objective factors influence employee attrition. They were more interested in predicting the maximum number of potential departures by minimizing the number of false negatives. Emphasizing recall in this case, the Gaussian Naïve Bayes classifier was determined best after validation of the models and thus fulfilling the study objective. On the attrition-related features, despite offering a detailed analysis of the study, it remains inconsistent in most places. For instance, even while mentioning five essential features, it will only state four in the conclusion. Second, another inconsistency is directed toward the false positive rate mentioned in the discussion part and later referring to the false negative. Despite these inconsistencies, it still provides vital inputs into the detailed explanation of the features.

The following research completed by Mansor et al. [11] compared the predictive performance of Decision Tree (DT), Support Vector Machine (SVM), and Artificial Neural Network (ANN). To boost the level of accuracy in the models further, the authors have used several preprocessing techniques, tuning of parameters, and regularization techniques. It was only at the initial stages that the ANN model outperformed others in terms of accuracy, Root Mean Square Error (RMSE), and Receiver Operating Characteristic (ROC). The specific type of ANN used in their study was the Multi-Layer Perceptron (MLP), a non-linear predictive model designed to learn the relationship between input data (X) and target outputs (Y). During training, the MLP adjusts its parameters to make its predictions as close as possible to the actual target values. However, after parameter tuning, the authors found that the Support Vector Machine model outperformed the Artificial Neural Network. This highlights the critical role of parameter tuning in improving the model's predictive performance significantly.

Alsheref et al. [5] conducted another study using a different dataset but applied an ensemble method. In contrast to previous studies, this study applied the majority voting ensemble method, which selects the final prediction based on most of the predicted class by all base models. Conclusively, the authors found that there is no model that is perfect for each business cases. The weakness identified is that the lack of discussion regarding the implementation of the majority voting ensemble method.

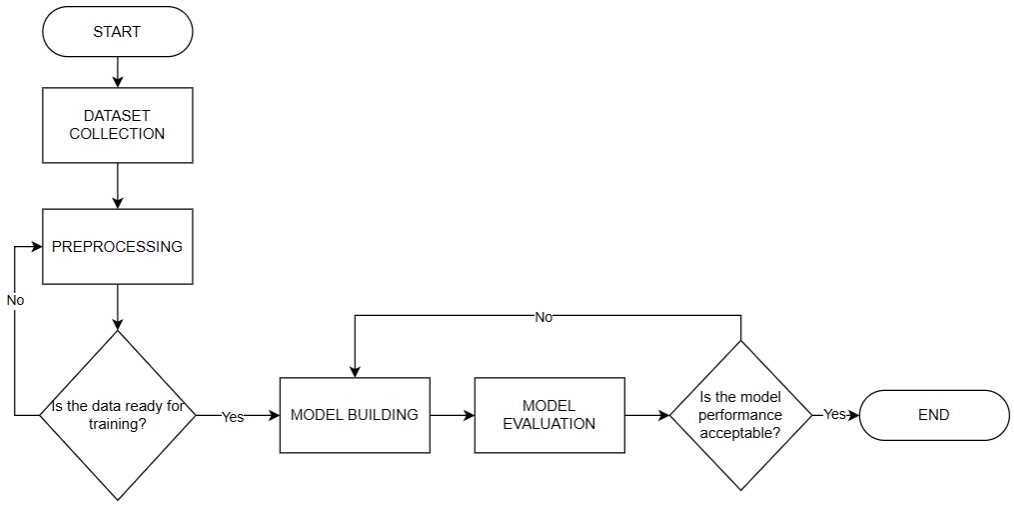
With this background known from the previous studies, this research will focus on constructing a TabNet model, a deep learning architecture specifically designed for tabular data, to predict employee attrition. TabNet was created by Arik and Pfister [12] to address the limitations of traditional Deep Neural Networks (DNNs) with tabular data. Unlike other DNNs, TabNet uses a sequential attention mechanism to select the most important features at each step, leading to interpretability and more efficient learning. This feature selection is instance-wise, meaning it focuses on the most relevant features for each individual employee, and unlike some other methods, it uses a single deep learning architecture for both feature selection and reasoning.

Arik and Pfister [12] also mention that TabNet supports two types of training: supervised training and unsupervised pre-training. This study employs only supervised training, as the dataset used is not considered heavily unlabelled. Unsupervised pre-training, typically situational, is used when a significant portion of the dataset lacks labels. Its purpose is to enhance the model's performance during subsequent supervised training by training the model to learn the relationships between columns. This is achieved by having the model predict the value of a masked column based on the remaining data.

RESEARCH METHODOLOGY

Research Methodology Overview

In this section, an overview of the methodology used in this research to achieve the stated objectives is highlighted. The study process can be summed up in the flowchart in Gambar 1. The process starts with getting the appropriate dataset. Next, the data goes through a long preprocessing steps to get it ready for analysis. Once the data is good enough to train on, models are built and tested over and over again. If the model's performance isn't good enough, changes are made to the model settings, and the process starts over. This repeated process makes sure that a strong and accurate model is made.



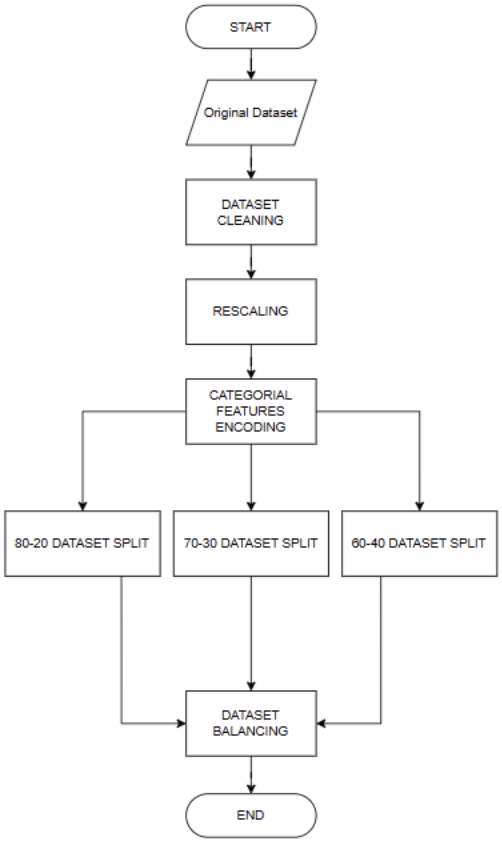
Gambar 1. Flowchart of Research Methodology

Dataset Collection

This study employed a dataset that was sourced from Kaggle specifically: "Employee Analysis | Attrition Report". The dataset contains 35 features from 1470 employees. The key variable of interest is "Attrition", which either has the value "yes" meaning an employee leaves the company or "no" meaning they end up retained. This dataset contains 18 numeric features and 17 categorical features.

Dataset Preprocessing

Preprocessing plays a crucial role in enhancing the performance of machine learning models. This study focused on employing several preprocessing steps, which included, cleaning, rescaling, categorical data encoding, dataset splitting into various ratios (80-20, 70-30, and 60-40), and dataset balancing. The entire sequence of preprocessing processes is illustrated in Gambar 2 below.



Gambar 2. Flowchart of Dataset Preprocessing

Dataset Cleaning

In the initial analysis of the dataset, it was observed that certain features such as EmployeeCount, Over18, and StandardHours, had identical values across all employees. Other than that, previous research [11] found that EmployeeNumber was not useful for the modeling and prediction process. As a result, these features were removed from the dataset. Further examination revealed that the dataset contained no null values, thus no rows were eliminated.

Data Rescaling

Machine learning models tend to perform better when dealing with small value feature rather than larger ones [1]. To solve this issue, feature rescaling technique was used. In this study, Min-Max scaling was used to rescale feature values within the range of 0 to 1 using the equation below.

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \quad (1)$$

In function (1) above, x' is the normalized value, x is the original value, x_{max} and x_{min} are the maximum and minimum value of the feature respectively.

Categorical Features Encoding

Categorical features cannot be directly used for model training. They need to be converted into a numerical form. In this study, one-hot encoder was employed to solve this problem as was done in previous research [1]. As for the target feature, Attrition, the value of yes and no were converted into 1 and 0 respectively.

Hosni [13] explained, that the one-hot encoding process begins by determining the amount of distinct categories within the categorical attribute. Then, a series of columns is made, each column representing a different category. Next, depending on whether a data point is included in a specific category or not, binary values (0 or 1) are assigned to these columns. The number of columns generated corresponds to the number of distinct categories in the attribute.

Dataset Splitting

The performance of the models in this research were assessed comprehensively throughout this study through applying and comparing the results of splitting the dataset with varying ratios from 60/40 to 80/20. This approach involved dividing the dataset into two, training set and testing set.

Balancing the Dataset

In situations when class imbalance is present in the dataset such that one class is larger than the other, appropriate balancing techniques may be employed in order to improve the performance of model. After conducting further investigation on the data used in this project, it was discovered that there are 237 employees quit their job while the remaining 1233 employees did not, which clearly shows the case of class imbalance. To counter such problem, Synthetic Minority Oversampling Technique (SMOTE) was used to adjust the distribution of the proportion of classes in the training dataset. Based on previous work [11], 200% oversampling degree with five nearest neighbors was applied. The implementation of SMOTE in this study used the Scikit-learn library.

Description of Attributes and Preprocessing Actions

This section provides the description of attributes in the dataset and the specific preprocessing actions applied. The complete description of the dataset's attributes and their preprocessing action is shown in Tabel 1below.

Tabel 1. Description of Attributes and Preprocessing Actions

No	Attribute Name	Type of Data	Data Description	Preprocessing Action
1	Age	Numerical	Employee's age	Min = 18, Max = 60 rescaled
2	Attrition	Categorical	Employee decisions to leave the company or not (Yes, No)	No = 0, Yes = 1
3	Business Travel	Categorical	Business Travel frequency (No Travel, Travel Rarely, Travel Frequently)	Applied one-hot encoding

4	Daily Rate	Numerical	Daily Salary	Min = 102, Max = 1499 rescaled
5	Department	Categorical	Employee Working Department (HR, R&D, Sales)	Applied one-hot encoding
6	DistanceFromHome	Numerical	The distance from home to office	Min = 1, Max = 29 rescaled
7	Education	Categorical	Employee's education level	Retained
8	EducationField	Categorical	Employee's field of study	Applied one-hot encoding
9	EmployeeCount	Numerical	Employee individual count	Cardinality = 1, removed
10	EmployeeNumber	Numerical	Employee ID	Cardinality = 1470, removed
11	EnvironmentSatisfaction	Categorical	Employee satisfaction with the environment	Retained
12	Gender	Categorical	Employee's gender	Applied one-hot encoding
13	HourlyRate	Numerical	Hourly salary	Min = 30, Max = 100 rescaled
14	JobInvolvement	Categorical	The level of involvement required for the employee's job	Retained
15	JobLevel	Categorical	The job level of the employee	Retained
16	JobRole	Categorical	The role of the employee in the organization	Applied one-hot encoding
17	JobSatisfaction	Categorical	Employee's satisfaction with their job	Retained
18	MaritalStatus	Categorical	Employee's marital status	Applied one-hot encoding
19	MonthlyIncome	Numerical	Employee's monthly income	Min = 1009, Max = 19999 rescaled
20	MonthlyRate	Numerical	Monthly rate of pay for the employee	Min = 2094, Max = 26999 rescaled

21	NumCompaniesWorked	Numerical	Number of companies the employee has worked for	Min = 0, Max = 9 rescaled
22	Over18	Categorical	Whether or not the employee is over 18	Cardinality = 1 removed
23	OverTime	Categorical	Whether or not the employee works overtime	Applied one-hot encoding
24	PercentSalaryHike	Numerical	The percentage of salary hike for the employee	Min = 11, Max = 25 rescaled
25	PerformanceRating	Categorical	The performance rating of the employee	Retained
26	RelationshipSatisfaction	Categorical	The employee's satisfaction with their relationships	Retained
27	StandardHours	Numerical	The standard hours of work for the employee	Cardinality = 1, removed
28	StockOptionLevel	Numerical	The stock option level of the employee	Retained
29	TotalWorkingYears	Numerical	The total number of years the employee has worked	Min = 0, Max = 40 rescaled
30	TrainingTimesLastYear	Numerical	The number of times the employee was taken for training in the last year	Retained
31	WorkLifeBalance	Categorical	The employee's perception of their work-life balance	Retained
32	YearsAtCompany	Numerical	The number of years the employee has been with the company	Min = 0, Max = 40 rescaled
33	YearsInCurrentRole	Numerical	The number of years the employee has been in their current role	Min = 0, Max = 18 rescaled
34	YearsSinceLastPromotion	Numerical	The number of years since the employee's last promotion	Min = 0, Max = 15 rescaled
35	YearsWithCurrManager	Numerical	The number of years the employee has been with their current manager	Min = 0, Max = 17 rescaled

Model Building

The TabNet model was constructed utilizing the Pytorch library. The Multi-Layer Perceptron (MLP) and Random Forest (RF) models were built using Scikit-learn library. The models' performance was optimized via hyperparameter tuning using Grid Search algorithm. All models

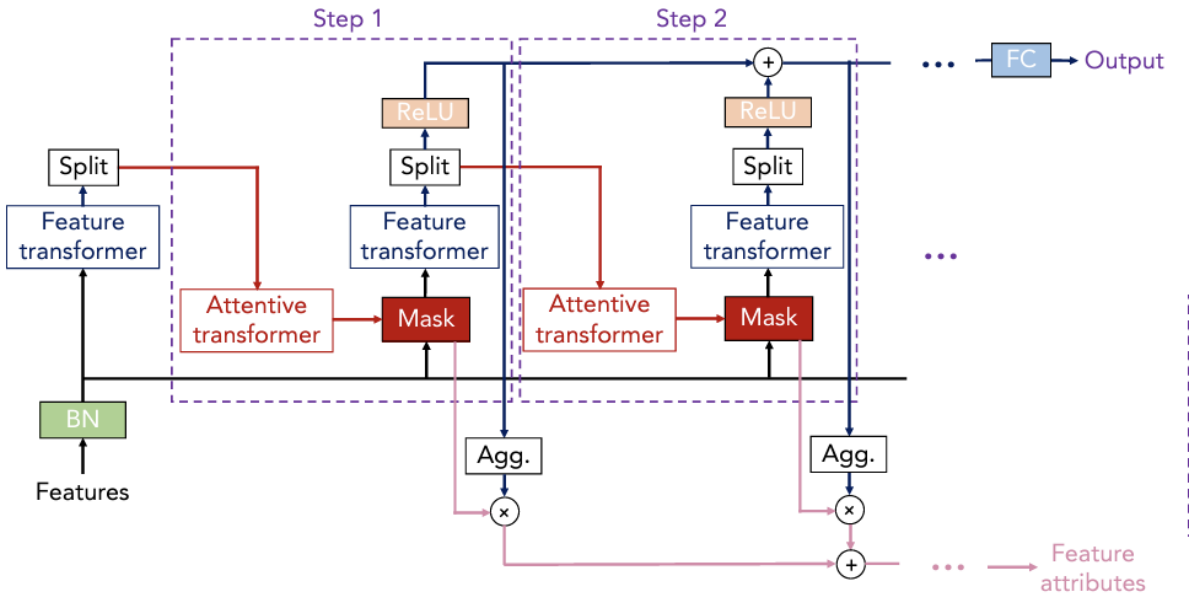
were trained on the same feature set and the same split ratio of test and train data, and all of them were validated by stratified 5-fold cross validation on the training set. This technique partitions the training data into five segments equally, while maintaining the proportion of each class in every fold. Four parts are used for training and one portion for validation. By averaging the results on these folds, the author could get a fair estimate of the model generalization ability for each split ratio.

Hyperparameter Tuning

Hyperparameter tuning is one of the important steps in order to improve the performance of machine learning models [14]. In this research, the author carried out the procedure of tuning the model's hyperparameters through the application of grid search. Grid search is a hyperparameter optimization technique that systematically traverses all parameter combinations in a specified search space. All parametric possibilities are checked and the model is trained and evaluated for each of the combinations. By using this technique, the author can identify the combination which gives the best results in terms of accuracy.

TabNet Supervised Learning

This research did not employ unsupervised pre-training, as the dataset used did not fall into the category of heavily unlabeled data. Therefore, only supervised training was implemented. TabNet's supervised learning utilizes the TabNet encoder as its main component. To better understand how TabNet functions, the architecture of the encoder is illustrated in Gambar 3 below.



(a) TabNet encoder architecture

Gambar 3. TabNet Encoder Architecture (adapted from Arik and Pfister [12])

In the supervised learning process, the dataset's features first pass through a Batch Normalization (BN) layer, where the data is normalized to accelerate and stabilize the training process. This normalization ensures that the mean of the data approaches zero and the standard deviation approaches one. Once normalized, the data is split and flows into two components: the feature transformer and the decision step. Within the feature transformer, the model learns the relationships among features, producing an output that will be divided into two parts: one becomes the output for that decision step, while the other serves as input for the attentive transformer in the subsequent step. Since the first feature transformer is not associated with any decision step, it only produces an output that will be utilized by the attentive transformer in the first decision step.

The information passed from the previous decision step is used as the primary input for the attentive transformer to produce a masking layer. The data first passes through a Fully Connected (FC) layer and a BN layer, where the model begins to identify which features are most relevant at the current step. After that, the data is modulated by its prior scale to ensure that the selected features differ across steps. The results are then processed through a sparsemax function, which generates a masking layer to filter the input for the decision step. The input from the initial BN layer passes through the masking layer ensuring sparse feature selection. This allows only a subset of features to be processed at each step, enhancing the model's efficiency without entirely discarding less relevant features. Instead, the influence of less relevant features is reduced.

The selected features are then sent back to the feature transformer, where the model continues learning feature relationships and generates outputs for the current decision step and inputs for the next. After all decision steps are completed, the outputs from each step are combined and passed through an FC layer to produce the final prediction. Additionally, the masking generated at each decision step is combined to form a global feature importance metric, which highlights the most influential features in the overall prediction process.

Model Evaluation and Comparison

In this section, the author evaluates and compares the performance of the TabNet model with the Multi-Layer Perceptron (MLP) and Random Forest (RF) models. To assess each model's performance, several evaluation metrics were used, including accuracy, precision, recall, and F1 score.

Each model was tuned and cross-validated on three data splits (60-40, 70-30, and 80-20), with the objective of identifying the best-performing configuration for each model based on cross validated training performance. For each model, the configuration with the highest average cross validation accuracy across folds within each split was initially selected. This configuration was then evaluated on the test set, and the test results determined the final model performance.

Finally, the test set results for the highest-performing configurations of TabNet, Multi-Layer Perceptron, and Random Forest were compared across all metrics to determine overall model performance.

IMPLEMENTATION AND RESULTS

Experiment Setup

This research was conducted using Python version 3 in Google Colab, with a disk usage of 36.5 GB. To build the TabNet model, the author used the PyTorch library, while the Multi- Layer Perceptron (MLP) and Random Forest (RF) models were built using the Scikit-learn library.

Implementation

This section describes the tools and libraries used to predict employee attrition using Tabnet and compares its performance to Random Forest and Multi-Layer Perceptron. The initial step of the implementation is to download and upload the "Employee Analysis | Attrition Report" dataset from Kaggle to Google Colab. Following that, we can install the libraries required for this research, including imbalanced-learn, torch, and pytorch-tabnet.

In this research, several important libraries were imported for data preprocessing, model training, and model evaluation. Pandas was used to handle and manipulate DataFrames. Scikitlearn was mostly used in this research, it provided tools for calculating metrics (f1_score, precision_score, recall_score, accuracy_score, make_scorer), as well as for preprocessing data with encoders (OneHotEncoder, LabelEncoder). Moreover, it was also used for splitting dataset (train_test_split) and perform grid search (GridSearchCV and StratifiedKFold). RandomForestClassifier and MLPClassifier were also imported from Scikit-learn for model training. The scaling in this study implemented Min-Max scaling to rescale the feature values.

Additionally, the TabNetClassifier was imported from the PyTorch library for training the TabNet model. To handle imbalanced datasets, SMOTE from the imbalanced-learn library was used for oversampling the minority class.

Dataset Cleaning

After uploading the dataset to google colab, the dataset needs to be cleaned from unused features as shown in the code below.

```
1. def clean_data(df):
2.     """Clean and preprocess the data by dropping irrelevant columns."""
3.     df = df.drop(columns=['EmployeeCount', 'StandardHours', 'Over18',
4.     'EmployeeNumber'])
5.     return df
6.
7. # Data cleaning
8. df = clean_data(df)
```

The function above cleans the "Employee Analysis | Attrition Report" dataset by dropping irrelevant features ('EmployeeCount', 'StandardHours', 'Over18', and 'EmployeeNumber') that add unnecessary complexity to the model. To clean the dataset, we can call the function and input the original DataFrame as an argument, which then returns a cleaned DataFrame stored in df variable for further use.

Feature Rescaling

After cleaning the dataset, the next step is feature rescaling by using the Min-Max scaling technique as shown in the code below.

```
def scale_features(df, columns_to_rescale):
    """Scale specified numerical features using MinMaxScaling."""
    for column in columns_to_rescale:
      min_val = df[column].min()
      max_val = df[column].max()
      df[column]= (df[column] - min_val)/(max_val - min_val)
    return df

columns_to_rescale=['Age','DailyRate','DistanceFromHome','HourlyRate','Mon
thlyIncome','MonthlyRate', 'NumCompaniesWorked', 'PercentSalaryHike',
'TotalWorkingYears', 'YearsAtCompany', 'YearsInCurrentRole',
'YearsSinceLastPromotion', 'YearsWithCurrManager']

 df = scale_features(df, columns_to_rescale)
```

The function above rescales the numerical features of the "Employee Analysis | Attrition Report" dataset. Before apply rescaling, relevant numerical columns are listed in the columns_to_rescale variable. To rescale these features, call scale_features(df, columns_to_rescale) with the cleaned DataFrame and column list as arguments. The function returns a rescaled DataFrame, stored in df for further use.

Categorical Features Encoding

After rescaling the dataset, the next step is categorical features encoding by using the OneHotEncoder and LabelEncoder function from Scikit-learn library as shown in the code below.

```
def encode_features(df, categorical_columns):
    """OneHot encode specified categorical features."""
    encoder = OneHotEncoder(sparse_output=False)
    encoded_columns = encoder.fit_transform(df[categorical_columns])
    encoded_column_names                                              =
encoder.get_feature_names_out(categorical_columns)
    encoded_df               =               pd.DataFrame(encoded_columns,
columns=encoded_column_names, index=df.index)
    df = df.drop(columns=categorical_columns)
    df = pd.concat([df, encoded_df], axis=1)

    return df
```

The above function one-hot encodes the specified categorical columns. This function uses Scikit-learn's OneHotEncoder to transform the listed categorical columns, producing a DataFrame with encoded values and appending it to the original DataFrame after dropping the original categorical columns.

```
1. def label_encode_target(df, target_column):
2.     """Label encode the target column."""
3.     label_encoder = LabelEncoder()
4.     df[target_column] = label_encoder.fit_transform(df[target_column])
5.     return df
```

The above function label-encodes the target feature, 'Attrition,' using Scikit-learn's LabelEncoder. This method converts categorical values into binary form (e.g., 'Yes' as 1 and 'No' as 0).

```
1. categorical_columns = ['BusinessTravel', 'Department',
   'EducationField','Gender', 'JobRole', 'MaritalStatus', 'OverTime']
2.
3. df = encode_features(df, categorical_columns)
4. # Label encode the target
5. df = label_encode_target(df, 'Attrition')
```

Before apply the feature encoding, relevant categorical features are listed in the categorical_columns. To encode the categorical features, call the encode_features(df, categorical_columns) function with the rescaled DataFrame and column list as arguments. The function returns an encoded DataFrame, stored in df. After that, to transform the 'Attrition' feature values into 1 and 0, call the label_encode_target(df, 'Attrition') function with the encoded DataFrame and column name as arguments. The function then return a labeled DataFrame, stored in df for further use.

Dataset Balancing

After encoding the categorical features of the dataset, the next step is to balance the dataset. Before balancing, we first split the dataset into training and testing sets. Dataset balancing is then applied only to the training set. In this research, three different split ratios are tested, each using a unique sampling_strategy for the respective split, as shown in the code below.

```
1. # SMOTE 80-20
2. smote = SMOTE(sampling_strategy=0.608, random_state=42)
3. X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
4. # SMOTE 70-30
5. smote = SMOTE(sampling_strategy=0.619, random_state=42)
6. X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
7. # SMOTE 60-40
8. smote = SMOTE(sampling_strategy=0.645, random_state=42)
9. X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
```

In the 80-20 split, the sampling_strategy is set to 0.608, meaning that the minority class is resampled to 60.8% of the majority class size. For the 70-30 split, a sampling_strategy of 0.619 increases the minority class to 61.9% of the majority class. In the 60-40 split, the sampling_strategy is set to 0.645, bringing the minority class to 64.5% of the majority class.

Results

In this study, the author tested three different train-test split ratios for three different algorithms: Random Forest, Multi-Layer Perceptron, and TabNet. To find the best hyperparameters for each split ratio, the models were trained and tuned utilizing GridSearchCV and stratified 5-fold cross-validation. The best parameter combination for each model was chosen using the model's average accuracy. The model with the best average accuracy from the tuning process was then used on the testing set to assess it's performance.

Results for Data Balancing

In this section, the data balancing was done using SMOTE with 200% oversampling degree, referenced from Mansor et al.'s research [11]. The table below shows the division of majority and minority classes of the training data before and after SMOTE was applied.

Tabel 2. Resampled Training Data Results of 80-20 Split

	Number of Instances	**Minority Class (Attrition Yes)**	**Majority Class (Attrition No)**
Before	1176	198 (16.8%)	978 (83.2%)
After	1572	594 (37.8%)	978 (62.2%)

Tabel 3. Resampled Training Data Results of 70-30 Split

	Number of Instances	**Minority Class (Attrition Yes)**	**Majority Class (Attrition No)**
Before	1029	176 (17.1%)	853 (82.9%)
After	1381	528 (38.2%)	853 (61.8%)

Tabel 4. Resampled Training Data Results of 60-40 Split

	Number of Instances	**Minority Class (Attrition Yes)**	**Majority Class (Attrition No)**
Before	882	156 (17.7%)	726 (82.3%)
After	1194	468 (39.2%)	726 (60.8%)

From Tabel 2, Tabel 3, and Tabel 4, the results reveal that SMOTE only preserves the number of the majority class while increasing the data in the minority class by 200%. This controlled rise helps prevent possible overfitting with too much synthetic data by lessening the imbalance without totally equalising the classes. This method maintains the original majority class size while moderately increasing the minority class, achieving a balance between enhanced minority class representation and the preservation of natural class proportions.

Results for Random Forest

The Random Forest model was trained using three different train-test split ratios. In each scenario, the model underwent the same hyperparameter tuning process using GridSearchCV with stratified 5-fold cross-validation. The model was initialized with class_weight='balanced' and a fixed random_state=42 to handle imbalanced data and ensure consistent results across runs. In this research there are a total of five hyperparameters tested, including n_estimator, max_feature,

min_samples_split, max_dept, and max_leaf_nodes. Below is an explanation of each hyperparameter.

1. **n_estimator** : This hyperparameter represents the number of trees in the forest. Higher values generally improve performance by reducing variance, but they also increase the computational cost. The values tested in this research are 50, 100, and 150.
2. **max_features** : This parameter controls the number of features considered for splitting at each node. The values tested in this research are None, sqrt, and log2.
3. **min_samples_split** : This parameter sets the minimum number of samples required to split an internal node. Lower values allow the model to split nodes more frequently, potentially leading to overfitting, while higher values help to control complexity. The values tested in this research are 2, 3, and 5.
4. **max_depth** : This parameter controls the maximum depth of each tree in the forest. Limiting depth prevents overfitting by reducing model complexity. The values tested in this research are None, 3, 6, and 9.
5. **max_leaf_nodes** : This parameter defines the maximum number of leaf nodes in a tree. Limiting the number of leaf nodes helps control model complexity, similar to max_depth. The values tested in this research are None, 3, 6, and 9.

From the explanation above, there are 432 unique hyperparameter combinations were evaluated during grid search. For each combination, the model was trained and validated using stratified 5-fold cross-validation. Thus, the total number of model fits performed during the grid search was 2160 fits.

Tabel 5. Best Hyperparameter Combination for Random Forest

n_estimators	max_feature	min_samples_split	max_depth	max_leaf_nodes	Avg Acc
80% - 20%					
100	log2	2	None	None	0. 911
70% - 30%					
150	log2	3	None	None	0.907
60% - 40%					
150	log2	3	None	None	0.906

After evaluating 432 unique hyperparameter combinations using GridSearchCV for Random Forest, the best combination for each split was identified based on the highest average accuracy from 5-fold cross-validation. As shown in Tabel 5, the best hyperparameter settings for the 80-20, 70-30, and 60-40 splits are substantially identical. However, the 80-20 split differs significantly: the optimal n_estimators are 100, whereas the other splits are 150. Furthermore, min_samples_split is set to 2 for the 80-20 split, as opposed to 3 for the other splits.

Tabel 6. Cross-Validation Results for the Best Random Forest Model

Cross Validation (CV)	Accuracy	F1	Precision	Recall	Training Time
80% - 20%					
CV1	0.905	0.903	0.908	0.905	0.3s
CV2	0.949	0.949	0.949	0.949	0.3s
CV3	0.901	0.899	0.905	0.901	0.3s
CV4	0.898	0.897	0.898	0.898	0.3s
CV5	0.904	0.902	0.909	0.904	0.3s
70% - 30%					
CV1	0.906	0.905	0.907	0.906	0.4s
CV2	0. 902	0.901	0.904	0.902	0.4s
CV3	0. 899	0. 896	0.904	0.899	0.4s
CV4	0.928	0.927	0.928	0.928	0.4s
CV5	0.902	0.901	0.905	0.902	0.4s
60% - 40%					
CV1	0.921	0.919	0.924	0.921	0.4s
CV2	0.908	0.907	0.908	0.908	0.4s
CV3	0.908	0.906	0.912	0.908	0.4s
CV4	0.887	0.885	0.891	0.887	0.4s
CV5	0.908	0.906	0.910	0.908	0.4s

Tabel 6 displays the cross-validation outcomes utilizing the optimal hyperparameters for each train-test division. Cross-validation prevents overfitting and ensures performance across training data subsets.

The model's cross-validation performance for the 80-20 split ranges from 0.898 to 0.949, averaging 0.911, meaning that 91% of predictions over all folds are correct. F1 scores range from 0.897 to 0.949, averaging 0.91. Precision is 0.898 to 0.949, averaging 0.914, while recall is 0.898 to 0.949, averaging 0.911. The model's computational efficiency is shown by its steady 0.3 seconds per fold training time.

The model performs consistently but somewhat lower in the 70-30 split, average 0.907 with accuracy between 0.899 to 0.928, suggesting 90% accuracy. The average F1 score is 0.906, ranging from 0.896 to 0.927. Precision ranges from 0.904 to 0.928, averaging 0.901, and recall from 0.899 to 0.928, averaging 0.907. Compared to the 80-20 split, training takes 0.4 seconds per fold longer.

Compared to the 70-30 split, the 60-40 split model has slightly poorer accuracy, ranging from 0.887 to 0.921 and averaging 0.906. The F1 score is slightly more consistent, averaging 0.905 from 0.885 to 0.919. Precision is 0.891 to 0.924, averaging 0.901, while recall is 0.887 to 0.921, averaging 0.906. Training time remains 0.4 seconds each fold, as in the preceding split.

Tabel 7. Test Result for the Best Random Forest Model

Split	Accuracy	F1	Precision	Recall
80% - 20%	0.881	0.856	0.860	0.881
70% - 30%	0.878	0.849	0.858	0.878
60% - 40%	0.881	0.853	0.865	0.881

The Random Forest model performed consistently across all splits with accuracy between 0.878 and 0.881, demonstrating high generalization to unknown data. Precision and recall remain steady, scoring between 0.858 - 0.865 and 0.878 - 0.881, demonstrating the model's accurate true positive categorization and effective false positive control. The F1 score is constant, ranging from 0.849 to 0.856, with the greatest score in the 80-20 split, indicating a strong precision-recall balance. These data show robust performance across splits, with the 80-20 split slightly better.

Results for Multi-Layer Perceptron

The Multi-Layer Perceptron model was also implemented with the same procedures as previous model with early_stopping was set to True and max_iter to 500 directly within the model initialization to control convergence and prevent overfitting. The model was also initialized with a fixed random_state=42 to ensure consistent results across runs.

In this research there are a total of four hyperparameters tested, including hidden_layer_sizes, activation, solver, and learning_rate. Below is an explanation of each hyperparameter.

1. **hidden_layer_sizes** : This parameter specifies the number of neurons and hidden layers of the Multi-Layer Perceptron. Each hidden layer helps the network learn patterns in the data, and more neurons generally increase the model's capacity to learn complex relationships. The values tested in this research are (100), (100,100), (100,100,100), and (100,100,100,100).
2. **activation** : The activation function transforms the weighted sum of inputs into the output of a neuron. It introduces non-linearity, allowing the network to learn complex relationships in the data. The values tested in this research are tanh, relu, identity, and logistic.
3. **solver** : The solver is the optimization algorithm used to update the weights during the training process. It controls how the model minimizes the loss function (a measure of the difference between the model's predictions and the actual values) and adjusts the weights accordingly. The values tested in this research are sgd, adam, and lbfgs.
4. **learning_rate** : The learning rate determines how much to adjust the model's weights with respect to the gradient of the loss function during each update step. A larger learning rate means the model updates weights more aggressively, while a smaller learning rate updates weights more cautiously. The values tested in this research are constant, adaptive, and invscaling.

From the explanation above, there are 144 unique hyperparameter combinations were evaluated during grid search. For each combination, the model was trained and validated using 5-fold cross-validation. Thus, the total number of model fits performed during the grid search was 720 fits.

Tabel 8. Best Hyperparameter Combination for Multi-Layer Perceptron

hidden_layer_sizes	activation	solver	learning_rate	Avg Acc
80% - 20%				
(100)	relu	lbfgs	constant	0.875
70% - 30%				
(100, 100, 100)	relu	adam	constant	0.871
60% - 40%				
(100, 100, 100)	relu	lbfgs	constant	0.877

After evaluating 144 unique hyperparameter combinations using GridSearchCV for Multi-Layer Perceptron, the best combination for each split was identified based on the highest average accuracy from 5-fold cross-validation.

Tabel 8 shows that the best hyperparameter configurations for the 80-20, 70-30, and 60-40 splits are similar, especially for activation function (relu) and learning rate (constant). But they differ in hidden_layer_sizes and solver. The 70-30 and 60-40 splits worked best with three hidden layers, but the 80-20 split worked best with one hidden layer. Meanwhile, the 80-20 and 60-40 splits worked best with lbfgs solver while the 70-30 worked best with adam solver.

Tabel 9. Cross-Validation Results for the Best Multi-Layer Perceptron Model

Cross Validation (CV)	Accuracy	F1	Precision	Recall	Training Time
80% - 20%					
CV1	0.902	0.902	0.903	0.902	2.0s
CV2	0.905	0.905	0.906	0.905	2.0s
CV3	0.847	0.848	0.851	0.847	1.0s
CV4	0.882	0.883	0.887	0.882	0.9s
CV5	0.841	0.843	0.850	0.841	1.0s
70% - 30%					
CV1	0.874	0.874	0.876	0.874	0.9s
CV2	0. 855	0.854	0.854	0.855	1.1s
CV3	0. 855	0. 855	0.856	0.855	1.6s
CV4	0.877	0.877	0.877	0.877	1.2s
CV5	0.895	0.894	0.895	0.895	0.9s
60% - 40%					
CV1	0.841	0.843	0.854	0.841	9.5s
CV2	0.879	0.879	0.883	0.879	5.6s
CV3	0.883	0.884	0.889	0.883	11.0s
CV4	0.895	0.895	0.895	0.895	8.9s
CV5	0.887	0.886	0.886	0.887	7.5s

For the 80-20 split, the model displayed accuracy ranging from 0.841 to 0.905, with an average of 0.875. The F1 score varied from 0.843 to 0.905, with an average of 0.876. Precision averaged 0.879, with values spanning from 0.850 to 0.906, while recall averaging 0.875 and

ranging between 0.841 and 0.905. Training times across folds varied from 0.9 to 2.0 seconds, with an average of 1.38 seconds.

For the 70-30 split, the model demonstrated stable performance, achieving accuracy scores ranging from 0.855 to 0.895, with an average accuracy of 0.871. The F1 score, ranging from 0.854 to 0.894 and averaging 0.871, highlights balanced performance between precision and recall. Precision values ranged from 0.854 to 0.895, with an average of 0.872, while recall scores were spanning from 0.855 to 0.895, with an average of 0.871. Training times across folds varied from 0.9 to 1.6 seconds, averaging 1.14 seconds, showing moderate variability in training duration.

In the 60-40 split, the model exhibited stable performance with accuracy ranged from 0.841 to a peak of 0.895 averaging 0.877. The F1 score spanning from 0.843 to 0.895 averaging 0.878. Precision scores spanned from a high of 0.889 to 0.854 averaging 0.881, while recall values ranging from 0.841 to 0.895 averaging 0.877. Training times took way longer than the previous two splits averaging 8.5 seconds, with a maximum of 11.0 seconds and a minimum of 5.6 seconds in.

Tabel 10. Test Result for the Best Multi-Layer Perceptron Model

Split	Accuracy	F1	Precision	Recall
80% - 20%	0.813	0.825	0.842	0.813
70% - 30%	0.857	0.850	0.845	0.857
60% - 40%	0.833	0.837	0.842	0.833

The Multi-Layer Perceptron model exhibits consistent performance across the splits, with accuracy values ranging from 0.813 to 0.857, indicating reliable generalization to new data. Precision demonstrates stability, with values ranging from 0.842 to 0.845. Meanwhile, recall showed minor fluctuations across splits with values ranging from 0.813 to 0.857. This indicates the model's consistent ability to identify positives but fluctuating ability to manage false positives. The F1 score exhibits minor fluctuations as well across splits, ranging from 0.825 to 0.850, with the peak value recorded in the 70-30 split, indicating a robust balance between precision and recall. The results indicate strong performance across all splits, with the 70-30 split achieving slightly higher overall metrics.

Results for TabNet

Similar to the two previous models, the TabNet model was also implemented using the same procedures. The model was initialized with optimizer_params=dict(lr=5e-3) to set the learning rate to 0.005, providing a slower and more controlled learning process compared to the default rate.

In this research there are a total of four hyperparameters tested, including n_d, n_a, n_steps, and mask_type. Below is an explanation of each hyperparameter.

1. **n_d** : This parameter controls the dimensionality of the feature transformer's output in each decision step of the TabNet model. Higher values mean more capacity for learning complex representations, while lower values limit the model's complexity. The values tested in this research are 8, 16, and 32.

2. **n_a** : This parameter controls the dimensionality of the feature transformer's output to be sent to the attentive transformer in the next decision step. The attention layer decides which features to highlight or suppress as the model processes the data. The values tested in this research are 8, 16, and 32.
3. **n_steps** : This parameter specifies the number of decision steps, or sequential passes, that the model performs over the data. Each step is a chance for the model to refine its decision by focusing on different features using the attention mechanism. More steps allow for deeper feature processing and increased decision refinement. The values tested in this research are 3, 5, and 10.
4. **mask_type** : The mask_type parameter controls how the model selects features at each decision step, influencing which features get more focus. The values tested in this research are entmax and sparsemax.

From the explanation above, there are 54 unique hyperparameter combinations were evaluated during grid search. For each combination, the model was trained and validated using 5-fold cross-validation. Thus, the total number of model fits performed during the grid search was 270 fits.

Tabel 11. Best Hyperparameter Combination for TabNet

n_d	n_a	n_steps	mask_type	Avg Acc
80% - 20%				
16	32	3	entmax	0.852
70% - 30%				
16	32	3	entmax	0.856
60% - 40%				
16	32	5	sparsemax	0.614

After evaluating 54 unique hyperparameter combinations using GridSearchCV for TabNet, the best combination for each split was identified based on the highest average accuracy from 5-fold cross-validation.

From Tabel 11, the optimal hyperparameter configurations across the 80-20, 70-30, and 60-40 splits are mostly similar, particularly in n_d (16) and n_a (32). However, the 60-40 split differ in the n_steps and mask_type parameter. using n_steps set to 5 and mask_type set to sparsemax rather than entmax.

Tabel 12. Cross-Validation Results for the Best TabNet Model

Cross Validation (CV)	Accuracy	F1	Precision	Recall	Training Time
80% - 20%					
CV1	0.867	0. 867	0. 867	0. 867	22.1s
CV2	0.857	0.858	0.859	0.857	21.8s
CV3	0.844	0.844	0.845	0.844	21.1s
CV4	0.838	0.839	0.842	0.838	21.8s

CV5	0.854	0.853	0.853	0.854	21.8s
70% - 30%					
CV1	0.848	0. 848	0. 848	0. 848	24.8s
CV2	0. 873	0. 873	0. 873	0. 873	24.1s
CV3	0. 848	0. 848	0. 848	0. 848	23.9s
CV4	0.833	0.835	0.838	0.833	24.3s
CV5	0.880	0.881	0.881	0.880	24.4s
60% - 40%					
CV1	0.615	0.480	0.635	0.615	0.3s
CV2	0.619	0.493	0.688	0.619	0.3s
CV3	0.607	0.479	0.566	0.607	0.3s
CV4	0.603	0.477	0.537	0.603	0.3s
CV5	0.626	0.517	0.653	0.626	0.3s

For the 80-20 split, the model achieves relatively consistent performance, with accuracy ranging from 0.838 to 0.867 and averaging 0.852. The F1 score varies from 0.839 to 0.867, with an average of 0.852, indicating that the model maintains a moderate balance between precision and recall. Precision averages at 0.853, with values ranging from 0.842 to 0.867, while recall averages at 0.852, with scores ranging from 0.838 to 0.867. Training times across folds remain stable, spanning from 21.1 to 22.1 seconds and averaging around 21.72 seconds.

In the 70-30 split, the model demonstrates a slight improvement in performance, with accuracy ranging from 0.833 to 0.880 and an average accuracy of 0.856. The F1 scores vary from 0.835 to 0.881, with an average score of 0.857, indicating better overall classification compared to the 80-20 split. Precision averages at 0.858, with values from 0.838 to 0.881, while recall averages at 0.856, with values ranging from 0.833 to 0.880. The training time for this split is slightly longer, averaging around 24.3 seconds per fold and spanning from 23.9 to 24.8 seconds.

For the 60-40 split, the model's performance drops significantly in this split, with accuracy values ranging from 0.603 to 0.626 and averaging at 0.614. Meanwhile, F1 score is between 0.477 and 0.517, averaging at 0.489. This suggests that the model struggles to generalize in this configuration. Precision ranges between 0.537 and 0.688, averaging 0.616, while recall values vary from as low as 0.603 to 0.626, with an average of 0.614. This indicates that the model can identify positives at times but largely fails to capture true positives consistently. Training time remains at 0.3 seconds per fold.

Tabel 13. Test Result for the Best TabNet Model

Split	Accuracy	F1	Precision	Recall
80% - 20%	0.827	0.826	0.825	0.827
70% - 30%	0.798	0.796	0.794	0.798
60% - 40%	0.776	0.790	0.809	0.776

Tabel 13 shows varying performance of the TabNet model across the splits, with accuracy ranging from 0.776 to 0.827, indicating that it generalizes reasonably well but slightly less consistently than other models. Precision and recall fluctuate, scoring between 0.794 and 0.825 for precision and 0.776 and 0.827 for recall, suggesting that the model's true positive classification

and false positive control are generally reliable but vary more with different splits. The F1 score is relatively stable, ranging from 0.790 to 0.826, with the highest score in the 80-20 split, indicating a balanced precision-recall trade-off in this configuration. Overall, these results highlight moderate robustness across splits, with the 80-20 split yielding the best overall performance.

Discussion

Tabel 14. Comparison of Model's Best Test Result

Model	Split	Accuracy	F1 Score	Precision	Recall
Random Forest	80-20	0.881	0.856	0.860	0.881
Multi-Layer Perceptron	70-30	0.857	0.850	0.845	0.857
TabNet	80-20	0.827	0.826	0.825	0.827

Based from Tabel 14, the Random Forest model demonstrated strong performance with an accuracy of 88.1%, precision of 86%, recall of 88.1%, and an F1 score of 85.6% on the 80-20 split, marking it as the highest-performing model among the three. In comparison, the Multi-Layer Perceptron (MLP) model achieved its best performance on the 70-30 split, with 85.7% accuracy, 84.5% precision, 85,7% recall, and an F1 score of 85%, which, while solid, is still slightly lower than Random Forest. Lastly, the best TabNet model, tested on the 80-20 split, attained an accuracy of 82.7%, precision of 82.5%, and both recall and F1 scored closely at 82.7% and 82.6% respectively.

These results reveal that, while each model performed well, Random Forest offered the most robust results across all metrics. This finding aligns with research by Grinsztajn et al. [15] in 2022, which highlights why tree-based models often outperform neural network approaches in tabular data tasks. According to their work, tree-based models like Random Forest are particularly effective at capturing the complex, irregular patterns that often appear in structured data. While neural network models such as Multi-Layer Perceptron and TabNet are generally designed to find smooth and generalized patterns, tree-based models divide the data into discrete segments, allowing them to pick up on local irregularities that deep learning models might miss.

Additionally, Grinsztajn et al. [15] emphasize that tree-based models are highly resilient to uninformative or irrelevant features, which can often introduce noise and reduce performance in neural networks. In our dataset, not all features equally contribute to predicting employee attrition. Tree-based models naturally deprioritize these less relevant features during the splitting process, focusing instead on the features with the strongest predictive power. Neural networks, by contrast, can be more sensitive to unimportant features if they are not carefully tuned or regularized.

Grinsztajn et al. [15] also noted another advantage of tree-based models such as Random Forest lie in how they respect the distinct meaning and structure of each feature. Each feature in tabular data has a unique context (for example, "Age" representing years or "MonthlyIncome" representing salary), which is essential for effective interpretation and accurate predictions. Unlike neural network models, which are rotationally invariant, they treat all features as equally important, making it challenging to distinguish between informative and uninformative features. This can lead to reduced performance when irrelevant features dominate the dataset.

The dataset size may also have played a role in Random Forest's success in this study. Deep learning models like Multi-Layer Perceptron and TabNet often require large datasets to fully leverage their pattern-recognition capabilities. With a limited dataset, they may struggle to generalize effectively, whereas simpler models like Random Forest perform robustly even on smaller data. Interestingly, despite being specifically designed for tabular data, TabNet performed lower than Multi-Layer Perceptron. This may be due to the tuning process, as TabNet is a complex model with many hyperparameters that can be sensitive to tuning. If not well-tuned, it might not perform to its full potential. Multi-Layer Perceptron, on the other hand, has a simpler architecture, which may make it easier to tune and produce consistent results with smaller datasets.

Thus, while Random Forest achieved the best results here, larger datasets and careful hyperparameter tuning could potentially allow TabNet model to match or even surpass Random Forest's performance in future studies.

CONCLUSION

Based on the results above, it can be concluded that the TabNet model is suitable for predicting employee attrition. While it demonstrates good overall performance, it does not outperform the Multi-Layer Perceptron (MLP) or Random Forest (RF) models in this study. Among the three models tested on the test set, Random Forest demonstrated the highest predictive performance. For the TabNet model, the model performs the best on the 80-20 split with a consistent result. The model achieves an accuracy of 82.7%, F1 score of 82.6%, precision of 82.5%, and recall at 82.7%.

While TabNet shows competitive performance with these metrics, deep learning models like TabNet and Multi-Layer Perceptron generally require longer training times compared to Random Forest due to their higher model complexity and tuning requirements. The relatively small dataset size in this study may also have limited TabNet's ability to fully utilize its deep learning architecture, which could partially explain its comparatively lower performance relative to Random Forest.

Suggestions for future research include exploring a larger dataset than the one used in this study, as deep learning models like TabNet may achieve better performance with increased data size. If future researchers choose to use the same dataset, increasing the oversampling percentage is recommended to address class imbalance more effectively. Additionally, conducting more extensive hyperparameter tuning is suggested to further enhance TabNet's performance. Exploring a wider range of hyperparameters and configurations could help the model better adapt to the dataset, potentially addressing its lower performance compared to other models.

DAFTAR PUSTAKA

[1] S. Arqawi *et al.*, "Predicting Employee Attrition and Performance Using Deep Learning," *Journal of Theoretical and Applied Information Technology*, vol. 100, no. 21, 2022, [Online]. Available: http://www.jatit.org/volumes/Vol100No21/21Vol100No21.pdf

[2] S. Al-Darraji, D. G. Honi, F. Fallucchi, A. I. Abdulsada, R. Giuliano, and H. A. Abdulmalik, "Employee Attrition Prediction Using Deep Neural Networks," *Computers*, vol. 10, no. 11, Art. no. 11, Nov. 2021, doi: 10.3390/computers10110141.

[3] C. Janiesch, P. Zschech, and K. Heinrich, "Machine learning and deep learning," Apr. 14, 2021, *arXiv*: arXiv:2104.05314. doi: 10.48550/arXiv.2104.05314.

[4] P. M. Usha and N. V. Balaji, "A comparative study on machine learning algorithms for employee attrition prediction," *IOP Conf. Ser.: Mater. Sci. Eng.*, vol. 1085, no. 1, p. 012029, Feb. 2021, doi: 10.1088/1757-899X/1085/1/012029.

[5] F. K. Alsheref, I. E. Fattoh, and W. M. Ead, "Automated Prediction of Employee Attrition Using Ensemble Model Based on Machine Learning Algorithms," *Computational Intelligence and Neuroscience*, vol. 2022, Jun. 2022, doi: 10.1155/2022/7728668.

[6] P. K. Jain, M. Jain, and R. Pamula, "Explaining and predicting employees' attrition: a machine learning approach," *SN Appl. Sci.*, vol. 2, no. 4, p. 757, Mar. 2020, doi: 10.1007/s42452-020-2519-4.

[7] A. Raza, K. Munir, M. Almutairi, F. Younas, and M. M. S. Fareed, "Predicting Employee Attrition Using Machine Learning Approaches," *Applied Sciences*, vol. 12, no. 13, Art. no. 13, Jan. 2022, doi: 10.3390/app12136424.

[8] F. Fallucchi, M. Coladangelo, R. Giuliano, and E. William De Luca, "Predicting Employee Attrition Using Machine Learning Techniques," *Computers*, vol. 9, no. 4, Art. no. 4, Dec. 2020, doi: 10.3390/computers9040086.

[9] A. Qutub, A. Al-Mehmadi, M. Al-Hssan, R. Aljohani, and H. S. Alghamdi, "Prediction of Employee Attrition Using Machine Learning and Ensemble Methods," *IJMLC*, vol. 11, no. 2, pp. 110–114, Mar. 2021, doi: 10.18178/ijmlc.2021.11.2.1022.

[10] F. Alsubaie and M. Aldoukhi, "Using machine learning algorithms with improved accuracy to analyze and predict employee attrition," *Decision Science Letters*, vol. 13, no. 1, pp. 1–18, 2024, doi: 10.5267/j.dsl.2023.12.006.

[11] N. Mansor, N. S. Sani, and M. Aliff, "Machine Learning for Predicting Employee Attrition," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 12, no. 11, Art. no. 11, Jan. 2021, doi: 10.14569/IJACSA.2021.0121149.

[12] S. O. *Arik* and T. Pfister, "TabNet: Attentive Interpretable Tabular Learning," *arXiv.org*, Aug. 2019, doi: 10.48550/arXiv.1908.07442.

[13] M. Hosni, "Encoding Techniques for Handling Categorical Data in Machine Learning-Based Software Development Effort Estimation," presented at the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, SCITEPRESS, Nov. 2023, pp. 460–467. doi: 10.5220/0012259400003598.

[14] M. R. Hossain and D. D. Timmer, "Machine Learning Model Optimization with Hyper *Parameter* Tuning Approach," *Global Journal of Computer Science and Technology*, vol. 21, no. D2, pp. 7–13, Sep. 2021.

[15] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep *learning* on tabular data?," Jul. 18, 2022, *arXiv*: arXiv:2207.08815. doi: 10.48550/arXiv.2207.08815.