

# Configuration Manual

MSc Research Project  
Data Analytics

Sreejith Sridhar  
Student ID: X22242376

School of Computing  
National College of Ireland

Supervisor: Prof. Christian Horn

**National College of Ireland**  
**MSc Project Submission Sheet**



**School of Computing**

<b>Student Name:</b>	Sreejith Sridhar		
<b>Student ID:</b>	X22242376		
<b>Programme:</b>	MSc. Data Analytics	<b>Year:</b>	2024
<b>Module:</b>	MSc. Research Project		
<b>Supervisor:</b>	Prof. Christian Horn		
<b>Submission Due Date:</b>	12/08/2024		
<b>Project Title:</b>	Configuration Manual		
<b>Word Count:</b>	911	<b>Page Count:</b>	9

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

<b>Signature:</b>	Sreejith Sridhar
<b>Date:</b>	12 <sup>TH</sup> August 2024

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST**

Attach a completed copy of this sheet to each project (including multiple copies)	<input type="checkbox"/>
<b>Attach a Moodle submission receipt of the online project submission,</b> to each project (including multiple copies).	<input type="checkbox"/>
<b>You must ensure that you retain a HARD COPY of the project,</b> both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

<b>Office Use Only</b>	
Signature:	
Date:	
Penalty Applied (if applicable):	

# Configuration Manual

Sreejith Sridhar  
X22242376

## 1 Introduction

In order predict employee attrition, this study uses a wide range of classification techniques, such as Random Forest, K-Nearest Neighbours (KNN), Decision Tree, Logistic Regression, and Support Vector Machine (SVM). To improve prediction accuracy, the research also incorporates an ensemble technique using a Voting Classifier, merging numerous models. This configuration manual provides extra code snippets when appropriate and describes the workflow from environment setup to model assessment, outlining the steps required to reproduce the project.

## 2 System Configuration

The project was executed on a computer operating Microsoft Windows 11 Pro, using an Intel(R) Core(TM) i5-8365U CPU and 16 GB of RAM. The development environment utilised was Jupyter Notebook, a component of the Anaconda Distribution, operating on Python version 3.11.5.

## 3 Data Collection

The dataset utilised in this study is obtained from Kaggle, first created by IBM data scientists. The dataset encompasses a range of characteristics about employees, such as their demographics, job positions, satisfaction levels, and other qualities relevant to their employment. These factors are essential for accurately predicting attrition. The dataset is available for access and download via the provided link<sup>1</sup>.

## 4 Environment Setup

### 4.1 Libraries and Packages Requires

The project uses many Python tools and packages for data processing, model training, assessment, and visualisation. Pandas and numpy for data processing, scikit-learn for machine learning methods and model assessment, seaborn and matplotlib for visualisation,

---

<sup>1</sup> <https://www.kaggle.com/datasets/pavansubhasht/ibm-hr-analytics-attrition-dataset/data>

and joblib for model storing and loading are essential. Ensemble approaches and SMOTE for unbalanced datasets are also installed as shown in Figure 1.

```
#Importing Libraries and Packages
import numpy as np
import pandas as pd
import seaborn as sns
import math
import os
import graphviz
import joblib
from matplotlib import pyplot as plt
%matplotlib inline

from sklearn import tree
from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from imblearn.over_sampling import SMOTE
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.manifold import TSNE
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import VarianceThreshold
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
from sklearn.ensemble import IsolationForest

from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import MinMaxScaler
import warnings
warnings.filterwarnings('ignore')

from sklearn.svm import SVC
from sklearn import svm

from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import AdaBoostClassifier

from sklearn.ensemble import VotingClassifier

from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
```

Figure 1: Importing Python Library and Packages

## 5 Data Preprocessing

### 5.1 Handling Categorical Features

The categorical characteristics were encoded using LabelEncoder to convert them into a numerical representation that is appropriate for machine learning models. Every category characteristic was encoded separately, and the encoders were saved for future use.

```
categorical_features = ['Attrition', 'BusinessTravel', 'Department', 'Education', 'EducationField',
                        'EnvironmentSatisfaction', 'Gender', 'Over18', 'JobInvolvement', 'JobLevel',
                        'JobRole', 'JobSatisfaction', 'MaritalStatus', 'NumCompaniesWorked',
                        'OverTime', 'PerformanceRating', 'RelationshipSatisfaction',
                        'StockOptionLevel', 'TrainingTimesLastYear', 'WorkLifeBalance']

# Create a dictionary to store LabelEncoders
label_encoders = {}

# Apply LabelEncoder to each categorical feature and store the encoder
for column in categorical_features:
    le = LabelEncoder()
    data[column] = le.fit_transform(data[column])
    label_encoders[column] = le

# Display the first few rows of the encoded dataset
data
```

Figure 2: Encoding Categorical features using LabelEncoder

## 5.2 Identifying Unique Values

The unique values of each attribute were examined. This stage identifies categorical variables with few values to influence feature encoding and preprocessing selections.

```
# Investigate all the elements within each feature
for column in data.columns:
    unique_vals = data[column].unique()
    length = len(unique_vals)
    if length < 18:
        print("The number of values for feature {}: {} ---> {}".format(column, length, unique_vals))
    else:
        print("The number of values for feature {}: {}".format(column, length))
```

Figure 3: Finding Unique Values

## 5.3 Converting Boolean Columns to Integer

The dataset's Boolean columns are transformed into integer type to guarantee compatibility with machine learning methods, which often need numerical input.

```
# Selecting all columns with boolean data types and converting them to integer type

bool_columns = new_data.select_dtypes(include=['bool']).columns
new_data[bool_columns] = new_data[bool_columns].astype(int)
```

Figure 4: Converting Boolean Columns to Integer

## 5.4 Feature Scaling

The numerical characteristics were normalised using the MinMaxScaler to scale the data within the range of 0 to 1. This is crucial for algorithms such as K-Nearest Neighbours and SVM.

```
# Defining numerical columns that require scaling to ensure features are on the same scale
cols_to_scale = ['Age', 'DailyRate', 'DistanceFromHome', 'HourlyRate', 'MonthlyIncome', 'MonthlyRate', 'PercentSalaryHike',
                 'TotalWorkingYears', 'YearsAtCompany', 'YearsInCurrentRole', 'YearsWithCurrManager',
                 'YearsSinceLastPromotion']

# Initializing the MinMaxScaler to scale features to a range of 0 to 1
scaler = MinMaxScaler()

# Fitting the scaler to the training data and transforming both training and testing data
# This ensures that both datasets are scaled consistently, based on the training data
scaler.fit(X_train[cols_to_scale])
X_train[cols_to_scale] = scaler.transform(X_train[cols_to_scale])
X_test[cols_to_scale] = scaler.transform(X_test[cols_to_scale])
```

Figure 5: MinMaxScaler to scale features

## 5.5 Handling Feature Correlation

Highly correlated characteristics are found and eliminated to address multicollinearity and enhance the interpretability and performance of the model.

```
corr_features = correlation(X_train, 0.85)
print(len(corr_features))
print(corr_features)
```

**Figure 6: Finding Highly Correlated Features**

## 5.6 Handling Imbalance data

SMOTE is used to balance the dataset by generating synthetic samples for the minority class (Attrition – No).

```
#Balancing the data using SMOTE
smote = SMOTE(random_state=42)
X_train, y_train = smote.fit_resample(X_train, y_train)
```

**Figure 7: Implementing SMOTE**

# 6 Model Training and Evaluation

## 6.1 Hyperparameter Tuning

Hyperparameter optimisation was conducted to enhance the performance of the models. The scikit-learn library's GridSearchCV techniques were employed to identify the optimal hyperparameters for each model.

```
dt = DecisionTreeClassifier(random_state=42)

# Create the parameter grid based on the results of random search
params = {
    'max_depth': [2],
    'min_samples_leaf': [5, 10, 20, 50, 100],
    'criterion': ["gini", "entropy"]
}

# Instantiate the grid search model
grid_search = GridSearchCV(estimator=dt,
                           param_grid=params,
                           cv=4, n_jobs=-1, verbose=1, scoring = "accuracy")

grid_search.fit(X_train, y_train)
```

**Figure 8.1: Decision tree**

```
svm_clf = svm.SVC()
# defining parameter range
param_grid = {'C': [0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['linear', 'poly', 'sigmoid'],
              'probability': [True, False]}

svm_grid = GridSearchCV(SVC(), param_grid, cv=10, n_jobs=-1, verbose=2)

# fitting the model for grid search
svm_grid.fit(X_train, y_train)

svm_grid.best_estimator_
```

**Figure 8.2: Support Vector Machine**

```

rf_randomcv.best_params_

param_grid = {
    'criterion': [rf_randomcv.best_params_['criterion']],
    'max_depth': [rf_randomcv.best_params_['max_depth']],
    'max_features': [rf_randomcv.best_params_['max_features']],
    'min_samples_leaf': [rf_randomcv.best_params_['min_samples_leaf'],
                        rf_randomcv.best_params_['min_samples_leaf']+2,
                        rf_randomcv.best_params_['min_samples_leaf'] + 4],
    'min_samples_split': [rf_randomcv.best_params_['min_samples_split'] - 2,
                        rf_randomcv.best_params_['min_samples_split'] - 1,
                        rf_randomcv.best_params_['min_samples_split'],
                        rf_randomcv.best_params_['min_samples_split'] + 1,
                        rf_randomcv.best_params_['min_samples_split'] + 2],
    'n_estimators': [rf_randomcv.best_params_['n_estimators'] - 200, rf_randomcv.best_params_['n_estimators'] - 100,
                    rf_randomcv.best_params_['n_estimators'],
                    rf_randomcv.best_params_['n_estimators'] + 100, rf_randomcv.best_params_['n_estimators'] + 200]
}

param_grid

# Fit the grid search to the data
rf=RandomForestClassifier()
rf_gridsearch=GridSearchCV(estimator=rf,param_grid=param_grid,cv=10,n_jobs=-1,verbose=2)
rf_gridsearch.fit(X_train,y_train)

```

**Figure 8.3: Random Forest**

```

knn = KNeighborsClassifier()
from sklearn.model_selection import GridSearchCV

grid_params = { 'n_neighbors' : [5,6,7,9,11,13,15],
                'weights' : ['uniform','distance'],
                'metric' : ['minkowski','euclidean','manhattan']}

knn_grid = GridSearchCV(
    KNeighborsClassifier(),
    grid_params,
    verbose = 1,
    cv = 3,
    n_jobs = -1
)

knn_grid.fit(X_train, y_train)

knn_grid.best_estimator_

```

**Figure 8.4: K-Nearest Neighbor**

```

#Logistic Regression GridSearch

param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],
    'penalty': ['l1', 'l2', 'elasticnet', 'none'], # Note: 'elasticnet' works only with 'saga' solver
    'solver': ['newton-cg', 'lbfgs', 'liblinear', 'saga'] # Note: 'liblinear' doesn't support 'elasticnet'
}
# Perform grid search
logreg_grid = GridSearchCV(
    LogisticRegression(random_state=1, max_iter=1000),
    param_grid,
    cv=3,
    verbose=1,
    n_jobs=-1
)
logreg_grid.fit(X_train, y_train)
# Print best parameters found by grid search for Logistic Regression
print("Best Parameters found by Grid Search:")
logreg_grid.best_params_

```

**Figure 8.5: Logistic Regression**

## 6.2 Model Training

A Voting Classifier was trained using an ensemble of Logistic Regression, Decision Tree, Support Vector Machine (SVM), Random Forest, and K-Nearest Neighbours (KNN) models. The decision to use this ensemble approach with soft voting was taken to enhance the accuracy of predictions.

```

# List of best estimators
estimators = []
estimators.append(('DTC', DecisionTreeClassifier(max_depth=2, min_samples_leaf=5, random_state=42)))
estimators.append(('RFG', RandomForestClassifier(max_depth=340, max_features='log2', n_estimators=2000)))
estimators.append(('SVC', SVC(C=10, gamma=1, kernel='linear', probability=True)))
estimators.append(('KNN', KNeighborsClassifier(metric='manhattan', n_neighbors=6)))
estimators.append(('LR', LogisticRegression(C=0.1, penalty='l2', solver='newton-cg', random_state=42)))

# Voting Classifier with soft voting
vot_soft = VotingClassifier(estimators=estimators, voting='soft')
vot_soft.fit(X_train, y_train)

```

**Figure 9: Estimators to ensemble Voting Classifier**

## 6.3 Model Evaluation

The model had been evaluated using several measures, such as the confusion matrix, accuracy, and ROC AUC score. A confusion matrix was utilised to enhance knowledge of the model's performance.

```

# Evaluate the model
y_pred = vot_soft.predict(X_test)
print("\033[1mVoting Classifier (Soft Voting)\033[0m\n")

# Print confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n")
print(pd.DataFrame(conf_matrix, index=['Actual No', 'Actual Yes'], columns=['Predicted No', 'Predicted Yes']))

# Print classification report
print("\nClassification Report:\n")
print(classification_report(y_test, y_pred))

# Calculate and print accuracy
score = accuracy_score(y_test, y_pred)
print("\nSoft Voting Accuracy Score: {:.4f}".format(score))

```

**Figure 10: Evaluating the Voting Classifier Model**

## 6.4 ROC AUC Score Calculation for used models

This code computes the ROC AUC scores for several machine learning models, such as Logistic Regression, Random Forest , KNN, SVM, Decision Tree, Voting Classifier, Hybrid Model (DT + RF), and AdaBoost.

```

# Plot ROC Curve
plt.figure(figsize=(10, 5))
for result in result_table:
    plt.plot(result['fpr'], result['tpr'], label='{0} (area = {1:.3f})'.format(result['model_name'], result['auc']))

plt.plot([0, 1], [0, 1], 'k--') # Random guessing line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()

```

**Figure 11: Plotting ROC Curve**

## 6.5 Feature Importance

Feature importance was estimated using models that have the capability to do so, and the most important features were visually represented.

```
# Normalize the importances
feature_importances /= len([est for est in vot_soft.named_estimators_.values() if hasattr(est, "feature_importances_")])

# Function to plot top 15 feature importances
def plot_top_feature_importances(importances, feature_names, title, top_n=15):
    indices = np.argsort(importances)[::-1][:top_n]
    plt.figure(figsize=(8,4))
    sns.barplot(x=importances[indices], y=np.array(feature_names)[indices], palette="viridis")
    plt.title(title, fontsize=12)
    plt.xlabel('Importance', fontsize=10)
    plt.ylabel('Features', fontsize=10)
    plt.grid(True, linestyle='--', alpha=0.1)
    plt.tight_layout()
    plt.show()

# Plot the top 15 feature importances
plot_top_feature_importances(feature_importances, feature_names, "Top 15 Feature Importances from Voting Classifier", top_n=15)
```

Figure 12: Plotting Feature Importance

## 7 Model Deployment

### 7.1 Saving the Voting Classifier Model

The Voting Classifier model, MinMax Scaler, and Label Encoders are stored using the joblib library. This enables convenient reloading of the models and tools without the need to undergo retraining.

```
# Save the model for Streamlit
joblib.dump(vot_soft, 'voting_classifier_model.pkl')
# Save the scaler
joblib.dump(scaler, 'minmax_scaler.pkl')
# Save the label encoders
joblib.dump(label_encoders, 'label_encoders.pkl')
```

Figure 13: Saving the Model and Preprocessing Tools

### 7.2 Streamlit App Code

To utilise the stored model and preprocessing tools for making predictions on new information, they may be reloaded by employing the joblib.load function.

```
# Load the trained model and pre-fitted scaler, and label encoders
model = joblib.load('voting_classifier_model.pkl')
scaler = joblib.load('minmax_scaler.pkl') # Load your fitted MinMaxScaler
label_encoders = joblib.load('label_encoders.pkl') # Load your fitted LabelEncoders
```

Figure 14: Loading the Model and Preprocessing Tools

The system loads a pre-trained voting classifier model, as well as essential data preprocessors such as scalers and label encoders. The application gathers user input via the

sidebar, does preprocessing on the data, and subsequently estimates the probability of an employee departing from the company. The software presents the predicted outcome along with a probability value. To execute this code, save it as a Python file and run it in a Python environment by using the specified command.

```
# Function to preprocess user input
def preprocess_input(input_data):
    # Apply necessary preprocessing steps, e.g., scaling, encoding
    return input_data

# Function to get user input from the sidebar
def get_user_input():
    # Add the code to gather user input
    return {}

# Main function to run the Streamlit app
def main():
    st.title("Employee Attrition Prediction")
    st.write("Enter the employee details to predict attrition using the sidebar.")

    # Get user input
    user_input = get_user_input()

    # Button to make prediction
    if st.button('Predict'):
        # Preprocess the user input
        input_data = preprocess_input(user_input)

        # Convert input_data to a numpy array and reshape for prediction
        input_array = np.array(input_data).reshape(1, -1)

        # Make prediction
        prediction = model.predict(input_array)
        prediction_proba = model.predict_proba(input_array)

        # Show only the probability of attrition
        attrition_probability = prediction_proba[0][1]

        # Determine the color based on the probability
        color = "red" if attrition_probability > 0.5 else "green"

        # Display the prediction result with color
        st.markdown(f"<h3 style='color:{color}';>Probability of Attrition: {attrition_probability:.2%}</h3>", unsafe_allow_html=True)

        if prediction[0] == 1:
            st.error("The employee is likely to leave the company.")
        else:
            st.success("The employee is likely to stay in the company.")

if __name__ == '__main__':
    main()
```

**Figure 15: Main Function of the Streamlit App**

### 7.3 Running the Streamlit App file

```
bash

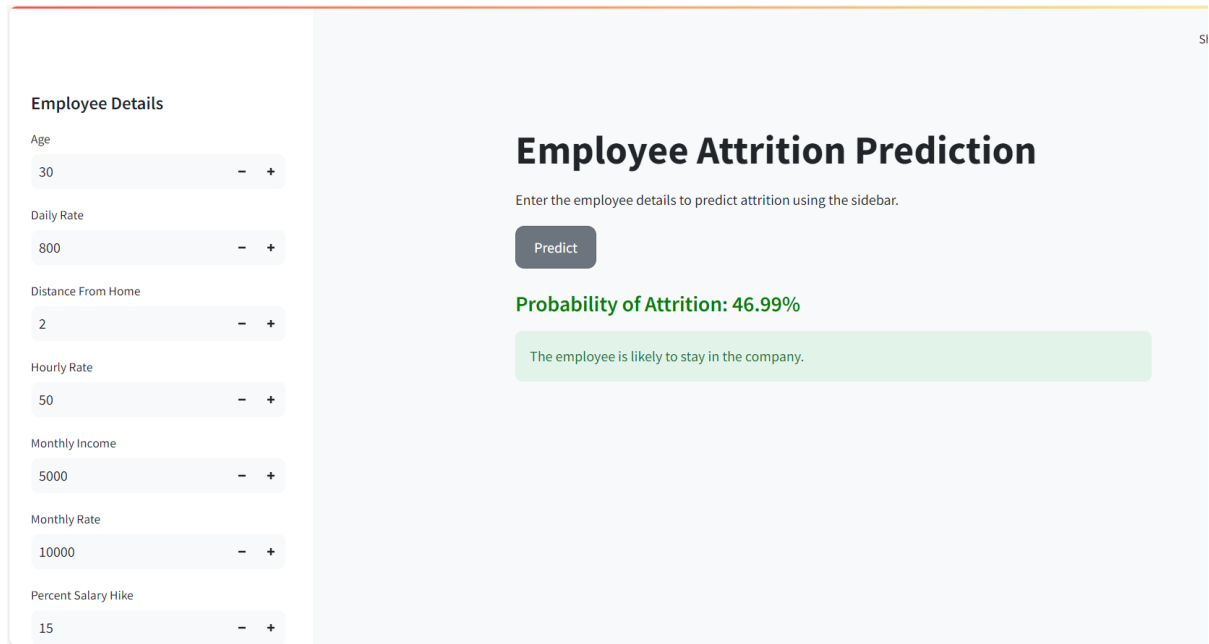
pip install streamlit numpy joblib

streamlit run employee_attrition_app.py
```

**Figure 16: Running the file in Python Environment**

## 8 Streamlit Application for Employee Attrition Prediction

This is the user interface of the Streamlit program created to predict employee attrition. This application can be accessed through the following link<sup>2</sup>.



The screenshot displays the user interface of a Streamlit application for predicting employee attrition. On the left, a sidebar titled "Employee Details" contains eight input fields, each with a minus and plus button for adjustment: Age (30), Daily Rate (800), Distance From Home (2), Hourly Rate (50), Monthly Income (5000), Monthly Rate (10000), Monthly Salary Hike (15), and Percent Salary Hike (15). The main area on the right features the title "Employee Attrition Prediction" and a subtext "Enter the employee details to predict attrition using the sidebar." Below this is a "Predict" button. The prediction result is shown as "Probability of Attrition: 46.99%" in green text, followed by a green box containing the message "The employee is likely to stay in the company."

**Figure 17: Streamlit Interface**

---

<sup>2</sup> <https://x22242376employeeattrition.streamlit.app/>