DSKC Week 2 Report - Data Set PreProcessing and Existing Models Implementation

Group 2

Members:

Garima Singh

Nishkarsh Singhal

Shaivee Sharma

Date of Submission: 23rd June, 2025

The dataset used in this study is the publicly available IBM Employee Attrition dataset, which contains structured records of employees within an organization. It consists of 1,470 rows (employees) and 35 columns (features) representing various attributes related to employee demographics, job roles, performance metrics, and satisfaction levels.

The core objective is to develop a machine learning-based predictive system that can:

- *Accurately identify employees at risk of leaving the company*
- *Support human resource departments in proactive retention strategies*
- *Compare the effectiveness of various models under class imbalance conditions*

DATASET PREPROCESSING

1. Dataset Overview

The dataset used for this analysis pertains to employee details relevant to predicting attrition (whether an employee has left the company). It was imported using pandas and stored in a DataFrame.

```
df = pd.read_csv(".../ATTRITION DATASET.csv")
```

---

2. Initial Cleanup

The following columns were dropped as they either had constant values, redundant information, or identifiers with no predictive power:

- EmployeeCount (same value for all records)
- Over18 (constant or irrelevant for analysis)
- StandardHours (standardized hours, no variation)
- EmployeeNumber (unique ID, doesn't contribute to prediction)

```
df.drop(columns=['EmployeeCount', 'Over18', 'StandardHours',
'EmployeeNumber'], inplace=True)
```

---

3. Target Variable Conversion

The target variable Attrition was originally categorical (Yes/No). It was converted to binary format for model compatibility:

- Yes → 1
- No → 0

```
df['Attrition'] = df['Attrition'].map({'Yes': 1, 'No': 0})
```

---

4. Feature-Target Split

The dataset was split into:

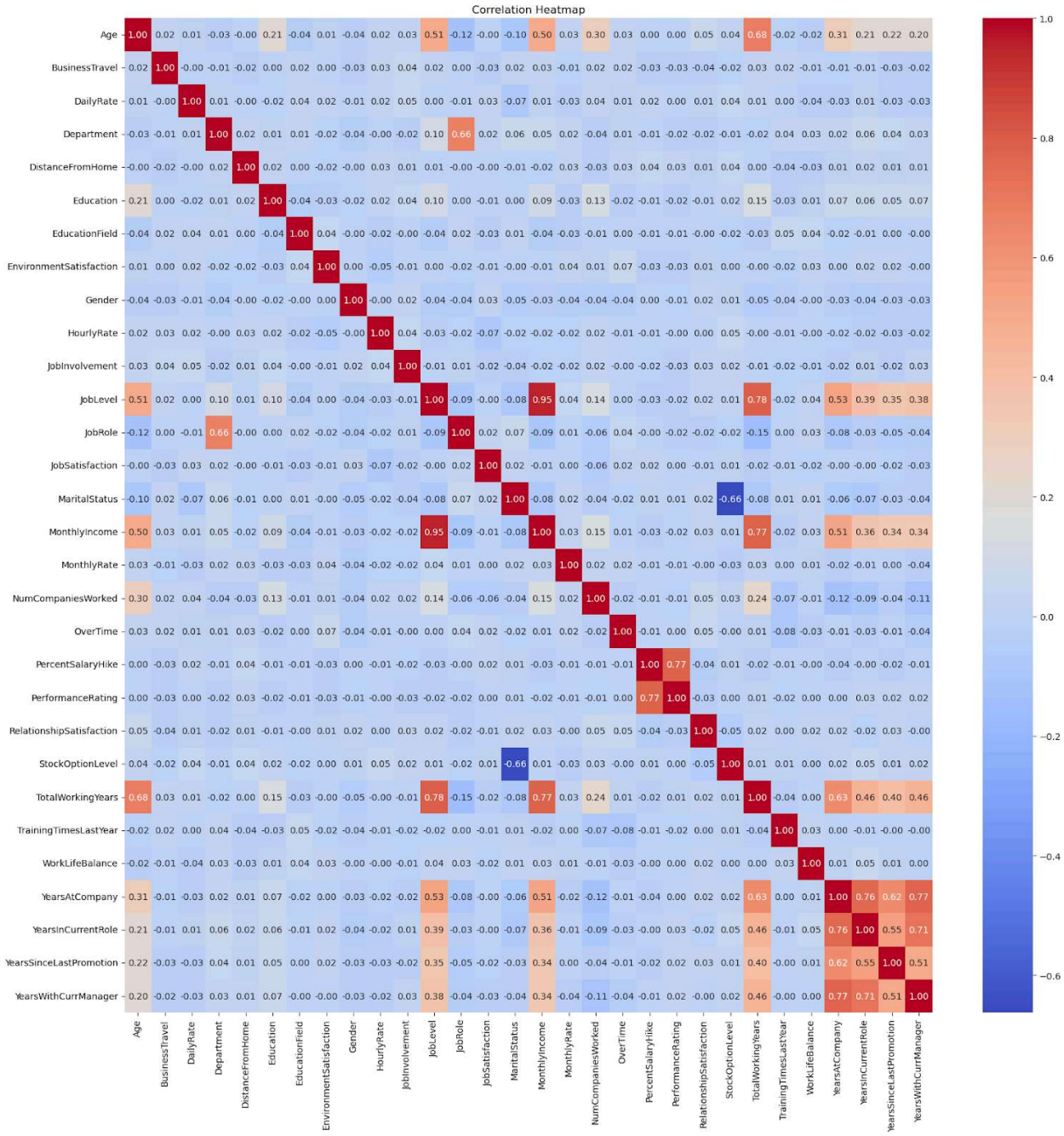
- X: Feature matrix (independent variables)
- y: Target vector (Attrition)

```
y = df['Attrition']
X = df.drop('Attrition', axis=1)
```

---

5. Correlation Analysis

```
categorical_cols = X.select_dtypes(include=['object', 'category']).columns
X_enc = X.copy()
le = LabelEncoder()
for col in categorical_cols:
    X_enc[col] = le.fit_transform(X_enc[col])
plt.figure(figsize=(20, 20))
corr_matrix = X_enc.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap')
plt.show()
```



Categorical columns were temporarily label encoded to compute correlation using Pearson's method.

A heatmap revealed a strong positive correlation between: JobLevel and MonthlyIncome. Hence, MonthlyIncome was dropped to reduce multicollinearity.

```
X.drop(columns=['MonthlyIncome'], inplace=True)
```

6. Encoding Categorical Variables

For models that are sensitive to feature magnitude or linear relationships, One-hot encoding was used.

- Categorical and numerical columns were identified separately.
- One-hot encoding was applied to the categorical columns using pd.get_dummies() with drop_first=True to avoid the dummy variable trap.
- Final feature matrix X_final was created by concatenating numerical and encoded categorical columns.

```
X_encoded = pd.get_dummies(X[categorical_cols], drop_first=True)
X_final = pd.concat([X[numerical_cols], X_encoded], axis=1)
```

For tree-based models that don't assume ordinal relationships, One-hot encoding can lead to higher dimensionality and hence is not preferred. Label encoding is less computationally expensive for high-cardinality categorical variables.

- Creates a label encoder object to convert text labels into numbers.
- Finds all columns with text (categorical) data.
- Loops through each categorical column.
- Converts unique text values in that column to integers.
- Replaces the original column with its numeric version.

```
le = LabelEncoder()
for col in X.select_dtypes(include='object').columns:
    X[col] = le.fit_transform(X[col])
```

---

7. Feature Scaling

The features were standardized using StandardScaler to normalize the feature distribution, which is essential for gradient-based algorithms.

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_final)
```

---

8. Train-Test Split

The dataset was split into training and testing sets using an 80:20 ratio with stratification to maintain class balance in both sets.

```
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, stratify=y, random_state=42
)
```

---

9. Handling Class Imbalance

To address the class imbalance in the target (Attrition is often imbalanced), SMOTE (Synthetic Minority Oversampling Technique) was applied to the training set to synthetically generate more instances of the minority class.

```
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train,
y_train)
```

THRESHOLD TUNING

- After training the XGBoost classifier, probability scores (predict_proba) were obtained for each test instance instead of hard class labels.
- Since the dataset is imbalanced, using the default threshold of 0.5 might not yield the best performance, especially in terms of the F1 Score, which balances precision and recall.
- A loop was implemented to test various threshold values ranging from 0.10 to 0.75 (step of 0.05), and for each threshold:
    - Predictions were binarized based on whether the probability exceeded the threshold.
    - Macro F1 Score was calculated, which gives equal importance to both classes regardless of their frequency.
- The goal was to identify the threshold value that results in the highest macro F1 Score, thus optimizing the classifier's ability to handle class imbalance effectively.
- The threshold achieving the maximum macro F1 score was stored and later used for making final predictions.

```
y_proba = xgb_model.predict_proba(X_test)[:, 1]
best_threshold = 0.5
best_f1 = 0
for thresh in np.arange(0.1, 0.8, 0.05):
    y_pred = (y_proba >= thresh).astype(int)
    f1 = f1_score(y_test, y_pred, average='macro')
    print(f"Threshold: {thresh:.2f} --> F1 Score: {f1:.4f}")
    if f1 > best_f1:
        best_f1 = f1
        best_threshold = thresh
```

DIFFERENT MODELS

1. XG-Boost:

What is XGBoost?

XGBoost (Extreme Gradient Boosting) is a powerful and efficient machine learning algorithm based on gradient boosting. It builds an ensemble of decision trees sequentially, where each new tree corrects the errors made by the previous ones.

Key Features:

- **Boosting-based**: Combines multiple weak models (usually decision trees) to make a strong one.
- **Gradient Descent**: Minimizes error using gradient descent on a custom loss function.
- **Regularization (L1/L2)**: Helps reduce overfitting.
- **Parallel Processing**: Very fast and optimized for performance.
- **Handles Missing Values**: Automatically learns the best direction for missing data.

How It Works:

1. Starts with an initial prediction.
2. Calculates the **residuals** (errors).
3. Builds a new tree to predict those errors.
4. Repeats this process, improving accuracy each time.
5. Final prediction = sum of all trees (weighted).

Parameter Tuning:

```
XGBClassifier(
    use_label_encoder=False,
    eval_metric='logloss',
    n_estimators=100,
    learning_rate=0.03,
    max_depth=3,
    subsample=0.6,
    colsample_bytree=0.6,
    scale_pos_weight=2,
    reg_lambda=10.0,
    reg_alpha=2.0,
    random_state=42,
    gamma=10.0
)
```

Parameter	Value	Explanation (Why it Matters)
`use_label_encoder`	False	Avoids deprecated warnings; ensures compatibility with newer versions of XGBoost.
`eval_metric`	logloss	Log loss is ideal for binary classification as it penalizes wrong confident predictions more.
`n_estimators`	100	Moderate number of trees. Paired with lower depth and learning rate for faster training with decent performance.
`learning_rate`	0.03	Slower learning for better generalization. Prevents overfitting when using many trees.
`max_depth`	3	Shallow trees help reduce overfitting. Good for controlling complexity and interpretability.
`subsample`	0.6	Only 60% of training data is used per tree → adds randomness and reduces overfitting.
`colsample_bytree`	0.6	Only 60% of features used per tree → increases model robustness and generalization.
`scale_pos_weight`	2	Gives extra weight to the minority class (e.g., attrition = 1). Helps handle class imbalance effectively.
`reg_lambda` ***(L2)***	10.0	Strong L2 regularization (Ridge). Helps shrink weights and prevent overfitting.
`reg_alpha` ***(L1)***	2.0	Adds sparsity via L1 regularization (Lasso). Pushes some weights to zero → feature selection effect.

REPORT:

Training accuracy: 87.02%

Testing Accuracy: 81.97%

Metric	Class 0	Class 1	Macro Avg	Weighted Avg
Precision	0.94	0.46	0.70	0.86
Recall	0.84	0.70	0.77	0.82
F1-Score	0.89	0.55	0.72	0.83

2. Random Forest:

What is Random Forest?

Random Forest is a bagging-based ensemble learning algorithm that builds multiple decision trees using random subsets of data and features. Instead of combining trees sequentially (like boosting), it builds them in parallel, and the final prediction is made via majority voting (for classification) or averaging (for regression).

Key Features:

- **Bagging-Based**: Builds trees independently on random bootstrapped samples and aggregates their output.
- **Random Feature Selection**: At each split, a random subset of features is considered → improves diversity among trees.
- **Reduces Overfitting**: Aggregating predictions from many trees reduces variance and improves generalization.
- **Handles Both Classification & Regression**: Versatile and widely used for structured/tabular data.
- **Robust to Outliers & Noise**: Since decisions are averaged, errors from individual trees have minimal impact.
- **No Need for Feature Scaling**: Trees aren't sensitive to magnitude of input features.

How It Works:

1. Takes multiple bootstrapped samples from the original dataset.
2. Grows a decision tree on each sample.
3. At every split, selects a random subset of features.
4. Grows trees fully (or till stopping criteria like `max_depth`).
5. Combines predictions:
    a. Classification: Majority vote across all trees.
    b. Regression: Mean of all tree outputs.

Parameter Tuning:

```
rf = RandomForestClassifier(
    n_estimators=120,
    max_depth=5,
    min_samples_split=10,
    min_samples_leaf=5,
    max_features='sqrt',
```

```
    bootstrap=True,
    class_weight='balanced_subsample',
    random_state=42
)
```

Parameter	Value	Explanation (Why it Matters)
`n_estimators`	120	Number of trees in the forest. More trees → better performance, but longer training.
`max_depth`	5	Limits depth of each tree → prevents overfitting.
`min_samples_split`	10	A node must have at least 10 samples to be split → controls tree complexity.
`min_samples_leaf`	5	A leaf must have at least 5 samples → ensures balanced trees.
`max_features`	'sqrt'	Uses √(total features) at each split → encourages diversity among trees.
`bootstrap`	True	Enables bagging (random sampling with replacement).
`class_weight`	'balanced_subsample'	Adjusts weights to handle class imbalance within each tree's sample.
`random_state`	42	Ensures reproducibility of results.

REPORT:

Training Accuracy: 91.94%

Test Accuracy: 76.53%

Metric	Class 0	Class 1	Macro Avg	Weighted Avg
Precision	0.95	0.39	0.67	0.86
Recall	0.76	0.79	0.77	0.77
F1-Score	0.84	0.52	0.68	0.79

---

3. LightGBM (LGBM):

What is LightGBM?

LightGBM (Light Gradient Boosting Machine) is an extremely fast, scalable, and efficient gradient boosting framework developed by Microsoft. It's optimized for large datasets and is known for its speed and accuracy.

Key Features:

- **Histogram-based algorithm**: Buckets continuous features into discrete bins for faster training.
- **Leaf-wise Tree Growth**: Unlike level-wise (used in XGBoost), LightGBM grows trees leaf-wise → deeper trees → better accuracy (but needs tuning to avoid overfitting).
- **Faster training speed**: Significantly faster on large datasets.
- **Low memory usage**: Uses less memory due to histogram-based splits.
- **Built-in support for categorical features** (no need for manual encoding).
- **Parallel and GPU training** supported.

How It Works:

1. Starts with a base prediction.
2. Residuals are calculated (similar to XGBoost).
3. Builds new trees in a **leaf-wise** manner (chooses the leaf with the highest loss reduction).
4. Combines results from all trees for the final prediction.

Parameter Tuning:

```
LGBMClassifier(
    subsample=0.8,
    scale_pos_weight=0.8,
    reg_lambda=1,
    reg_alpha=0.5,
    n_estimators=800,
    max_depth=10,
    learning_rate=0.01,
    colsample_bytree=0.7,
    random_state=42,
    class_weight=None
)
```

Parameter	Value	Explanation (Why it Matters)
`subsample`	0.8	Trains on 80% of data in each iteration → adds randomness, prevents overfitting.
`scale_pos_weight`	0.8	Balances the impact of minority class. Since the dataset is slightly imbalanced, this helps bias the model a bit more towards class 1.
`reg_lambda (L2)`	1	Standard L2 regularization. Adds penalty for large weights, preventing overfitting.
`reg_alpha (L1)`	0.5	Encourages sparsity in model weights. Helps in automatic feature selection.
`n_estimators`	800	High number of trees to learn complex patterns, balanced out by a low learning rate.
`max_depth`	10	Deep trees allow more feature interactions. But depth capped to control overfitting.
`learning_rate`	0.01	Small learning rate = slow learning = better generalization. Prevents aggressive weight updates.
`colsample_bytree`	0.7	Uses 70% of features for each tree → avoids reliance on same features.
`random_state`	42	Ensures reproducibility of results.
`class_weight`	None	Left default. Custom class weights handled via `scale_pos_weight` instead.

REPORT:

Training Accuracy: 95.02%

Testing Accuracy: 83.67%

Metric	Class 0	Class 1	Macro Avg	Weighted Avg
Precision	0.91	0.49	0.70	0.84
Recall	0.89	0.53	0.71	0.84
F1-Score	0.90	0.51	0.71	0.84

---

4. MLP (Multi-Layer Perceptron):

What is MLP?

Multi-Layer Perceptron (MLP) is a type of feedforward artificial neural network used in supervised learning. It consists of multiple layers of nodes (neurons), where each node is fully connected to the next layer.

Key Features:

- **Neural Network-Based**: Learns complex non-linear patterns using multiple layers and activations.
- **Backpropagation**: Uses gradient descent to minimize error through backward weight updates.
- **Flexibility**: Can learn from almost any structured data type with proper preprocessing.
- **Activation Functions**: Like ReLU help model complex relationships.
- **Optimizer (Adam)**: Combines momentum + RMSProp for fast and stable convergence.

How It Works:

1. Input features are fed into one or more hidden layers.
2. Each layer applies an activation function (like ReLU) to introduce non-linearity.
3. Predictions are generated by the output layer, and errors are calculated.
4. Weights are updated using backpropagation to minimize the loss function.

Parameter Tuning:

```
mlp = MLPClassifier(
    hidden_layer_sizes=(20,),
    activation='relu',
    solver='adam',
    alpha=0.0005,
    max_iter=100,
    random_state=42,
    early_stopping=True
)
```

Parameter	Value	Explanation (Why it Matters)
`hidden_layer_sizes`	(20,)	One hidden layer with 20 neurons. Controls the model's learning capacity.
`activation`	relu	ReLU is fast and avoids vanishing gradients. Helps learn non-linear patterns effectively.
`solver`	adam	Adaptive Moment Estimation — robust and efficient for noisy datasets.
`alpha`	0.0005	L2 regularization strength. Helps reduce overfitting by penalizing large weights.
`max_iter`	100	Limits training iterations. Prevents endless loops during convergence.
`random_state`	42	For reproducibility — same training behavior across runs.
`early_stopping`	True	Stops training early if validation loss doesn't improve — prevents overfitting.

REPORT:

Training Accuracy: 93.51%
Testing Accuracy: 86.39%

Metric	Class 0	Class 1	Macro Avg	Weighted Avg
Precision	0.92	0.58	0.75	0.86
Recall	0.92	0.55	0.74	0.86
F1-Score	0.92	0.57	0.74	0.86

---

5. Logistic Regression

What is Logistic Regression?

Logistic Regression is a linear classification algorithm used for predicting binary outcomes. It models the probability that a given input belongs to a particular class, using the logistic (sigmoid) function. Despite the name, it is a classification model, not regression.

Key Features:

- **Linear Model**: Draws a straight line (or hyperplane) to separate classes in the feature space.
- **Probabilistic Output**: Predicts the **probability** of class membership using a sigmoid curve.
- **Interpretable Coefficients**: Weights indicate the **influence of each feature** on the target.
- **Regularization (L2 by default)**: Helps control overfitting.
- **Class Weighting**: Can adjust for **imbalanced datasets** using `class_weight='balanced'`.

How It Works:

1. Computes a linear combination of input features:
   $z = w1x1 + w2x2 +\ldots + wnxn + b$
2. Applies the sigmoid function to map z into a probability:
   $\hat{y} = 1/(1 + e\text{^}(-z))$
3. Sets a threshold (usually 0.5) to decide the class: If ŷ ≥ threshold, predict class 1 , Else, predict class 0
4. Optimizes weights by minimizing log loss using a chosen solver (like `liblinear`).

Parameter Tuning:

```
lr = LogisticRegression(
    C=2.0,
    solver='liblinear',
    max_iter=2000,
    class_weight='balanced',
    random_state=42
)
```

Parameter	Value	Explanation
`C`	2.0	Inverse of regularization strength → higher value means less regularization (more freedom to fit).
`solver`	'liblinear'	Good for small datasets; supports L1/L2 penalties.
`max_iter`	2000	Ensures convergence in complex data; avoids early stopping.
`class_weight`	'balanced'	Adjusts weights to handle class imbalance by penalizing the majority class.
`random_state`	42	Ensures reproducibility.

REPORT:

Training Accuracy: 78.80%
Testing Accuracy: 86.73%

Metric	Class 0	Class 1	Macro Avg	Weighted Avg
Precision	0.91	0.61	0.76	0.86
Recall	0.94	0.49	0.71	0.87
F1-Score	0.92	0.54	0.73	0.86

---

6. Extra Trees Classifier

What is Extra Trees?

Extra Trees (Extremely Randomized Trees) is an ensemble learning method that builds multiple uncorrelated decision trees and averages their predictions. Unlike Random Forest, it adds extra randomness by choosing split thresholds randomly rather than by best split, making it faster and more diverse.

Key Features:

- **More Random Than Random Forest**: Splits are chosen using random thresholds for features → reduces variance.
- **Ensemble-Based**: Combines predictions from many trees → stable and robust.

- **Fast and Efficient**: Parallelized and optimized for speed.
- **Handles Imbalanced Data**: Using `class_weight='balanced'`.
- **Feature Importance**: Can rank features based on predictive power.

How It Works:

1. Randomly selects a subset of features for each tree.
2. At each node, instead of choosing the best split, it randomly selects a split value.
3. Trains each tree independently on bootstrapped samples.
4. Final prediction is made by majority vote (classification) or average (regression).

Parameter Tuning:

```
etc = ExtraTreesClassifier(
    n_estimators=100,
    max_depth=6,
    min_samples_split=10,
    min_samples_leaf=5,
    max_features='sqrt',
    class_weight='balanced',
    random_state=42,
    n_jobs=-1
)
```

Parameter	Value	Explanation
`n_estimators`	100	Number of trees in the forest.
`max_depth`	6	Limits the depth of each tree → prevents overfitting.
`min_samples_split`	10	Minimum number of samples required to split a node.
`min_samples_leaf`	5	Minimum number of samples in each leaf → avoids tiny leaf nodes.
`max_features`	'sqrt'	Uses √(total features) at each split → more randomness.
`class_weight`	'balanced'	Tackles class imbalance by assigning more weight to the minority class.
`random_state`	42	Ensures reproducible results.
`n_jobs`	-1	Uses all CPU cores for faster training.

REPORT:

Training Accuracy: 87.53%

Testing Accuracy : 79.25%

Metric	Class 0	Class 1	Macro Avg	Weighted Avg
Precision	0.94	0.42	0.68	0.86
Recall	0.80	0.74	0.77	0.79
F1-Score	0.87	0.53	0.70	0.81

---

7. Support Vector Machine (SVM)

What is SVM?

Support Vector Machine is a powerful classification algorithm that finds the best hyperplane that separates different classes by maximizing the margin between them. It works well in high-dimensional spaces and is robust to outliers.

Key Features:

- **Margin Maximization**: Ensures the decision boundary is as far as possible from both classes.
- **Kernel Trick**: Allows modeling of non-linear boundaries using functions like RBF (Radial Basis Function).
- **Robust to Overfitting**: Especially with proper regularization (via `C`).
- **Effective in High Dimensions**: Performs well even when number of features > number of observations.
- **Works with Class Imbalance**: Through `class_weight='balanced'`.

How It Works:

1. Maps data into high-dimensional space using kernels (e.g., RBF).
2. Finds the hyperplane that maximizes the margin between two classes.
3. Uses support vectors (critical boundary points) to define this hyperplane.
4. Makes predictions based on which side of the hyperplane a new point lies.

Parameter Tuning:

```
svm = SVC(
    kernel='rbf',
    C=0.5,
    gamma='scale',
    probability=True,
    class_weight='balanced',
    random_state=42
)
```

Parameter	Value	Explanation
`kernel`	'rbf'	Allows non-linear separation using the Gaussian kernel.
`C`	0.5	Regularization strength → smaller value allows for a softer margin → better generalization.
`gamma`	'scale'	Automatically adjusts the influence of points → prevents overfitting.
`probability`	True	Enables probability estimates using Platt scaling.
`class_weight`	'balanced'	Tackles class imbalance by adjusting weights based on label frequencies.
`random_state`	42	Reproducible results.

REPORT:

Training Accuracy: 96.04%
Testing Accuracy:80.27%

Metric	Class 0	Class 1	Macro Avg	Weighted Avg
Precision	0.91	0.41	0.66	0.83
Recall	0.85	0.53	0.69	0.80
F1-Score	0.88	0.46	0.67	0.81

---

8. Naive Bayes Classifier (GaussianNB)

What is Naive Bayes?

Naive Bayes is a probabilistic classifier based on Bayes' Theorem, assuming all features are independent given the class label (the "naive" assumption).
The `GaussianNB` version assumes the features are normally distributed (Gaussian).

Key Features:

- **Very Fast & Lightweight**: Trains almost instantly.
- **Works Well with High-Dimensional Data**: Especially text, spam detection, and binary classification.
- **Probabilistic Output**: Returns class probabilities directly.
- **Assumes Feature Independence**: Which simplifies computation but can be a limitation for correlated data.

How It Works:

1. Calculates prior probability of each class.
2. Computes likelihood of each feature assuming Gaussian distribution.
3. Uses Bayes' Theorem to compute posterior probability.
4. Class with the highest posterior probability is predicted.

Parameter Tuning:

```
nb = GaussianNB()
```

Parameter	Default	Explanation
`var_smoothing`	1e-9	Adds small value to variances to prevent division by zero. This is the only tunable parameter.
`priors`	None	Allows manual specification of class priors (rarely used).

Naive Bayes (`GaussianNB`) is a parameter-light algorithm. It typically does not require extensive parameter tuning, as it makes strong assumptions about feature independence and uses straightforward probabilistic formulas. The only tunable parameter, `var_smoothing`, usually retains its default value without significantly impacting performance.

REPORT:

Training Accuracy: 71.25%
Testing Accuracy: 76.19%

Metric	Class 0	Class 1	Macro Avg	Weighted Avg
Precision	0.91	0.35	0.63	0.82
Recall	0.80	0.57	0.69	0.76
F1-Score	0.85	0.44	0.64	0.78

---

9. K-Nearest Neighbors (KNN)

What is KNN?

K-Nearest Neighbors (KNN) is a supervised, instance-based learning algorithm that predicts a class by considering the majority class among its k nearest data points in the feature space. It doesn't build an internal model but rather stores all training data, making it a lazy learner.

Key Features:

- Non-parametric: Makes no assumptions about the data distribution.
- Distance-Based: Uses Euclidean (or other) distances to find similar instances.
- Simple & Intuitive: Works well for classification and regression.
- Sensitive to Feature Scale: Requires scaling for meaningful distance measures.
- Memory-Intensive: Slower on large datasets, since predictions require full data scanning.

How It Works:

1. Choose a value for k (number of neighbors).
2. For a new data point, calculate the distance to all training instances.
3. Identify the k nearest neighbors (e.g., based on Euclidean distance).
4. Majority vote among the neighbors determines the class.

Parameter Tuning:

```
KNeighborsClassifier(
    n_neighbors=5,
    metric='minkowski',
    p=2
)
```

Parameter	Value	Explanation
`n_neighbors`	5	Considers 5 nearest data points for majority voting.
`metric`	'minkowski'	General distance function. With `p=2`, it becomes Euclidean.
`p`	2	Euclidean distance (L2 norm). Commonly used.

REPORT:

Training Accuracy:86.51%
Testing Accuracy:75.51%

Metric	Class 0	Class 1	Macro Avg	Weighted Avg
Precision	0.91	0.34	0.62	0.82
Recall	0.79	0.57	0.68	0.76
F1-Score	0.84	0.43	0.64	0.78
