Grid Search vs Randomized Search for Hyperparameter Tuning: A Case Study on Credit Default Prediction with Random Forest

Authors: Anedda Laura, Capaldi Danilo, Gasparri Federico, Orefice Giuseppina

Abstract

This project investigates the impact of hyperparameter tuning on the predictive performance of a Random Forest classifier applied to credit default prediction. Using a real-world dataset containing client demographics, financial behavior, and repayment histories, the analysis includes appropriate preprocessing to address invalid entries and class imbalance.

Two hyperparameter optimization strategies — **Grid Search** and **Randomized Search** — are implemented and evaluated using cross-validation. Each resulting model is assessed on a consistent test set using standard classification metrics, including accuracy, precision, recall, F1-score, and ROC AUC.

Both tuning methods achieved **identical performance**, reaching a test accuracy of **71.48%** and a **ROC AUC of 0.782**, outperforming the default (untuned) Random Forest model. Despite exploring only a fraction of the parameter space, Randomized Search matched Grid Search's results with just **9% of the computational workload**, highlighting its efficiency and practical value.

The findings confirm that hyperparameter tuning significantly improves generalization performance, particularly by reducing overfitting. They also demonstrate that **Randomized Search is often sufficient** to identify optimal configurations in well-behaved search spaces, offering a strong trade-off between performance and resource efficiency.

Overall, the project presents a structured comparison of tuning strategies and reinforces the importance of data-driven parameter selection in supervised learning workflows.

Chapter 1 - Introduction and Dataset Preparation

1.1 Introduction

Accurate credit default prediction plays a crucial role in financial risk management, enabling institutions to mitigate losses and allocate resources more effectively. Machine learning models, particularly ensemble methods, have demonstrated strong performance in classification tasks involving structured financial data. This study adopts the **Random Forest** classifier as a benchmark model for credit default prediction and evaluates the performance impact of two hyperparameter optimization strategies: **Grid Search** and **Randomized Search**.

The dataset used in this analysis contains information on 30,000 credit card clients and includes a mix of demographic, financial, and behavioral variables relevant to credit risk assessment. Key demographic features include `AGE`, `SEX`, `EDUCATION`, and `MARRIAGE`, while financial indicators such as `LIMIT_BAL` (credit limit) and six monthly billing amounts (`BILL_AMT1` to `BILL_AMT6`) provide a snapshot of client balances over time.

Behavioral patterns are captured through past repayment records — specifically `PAY_0`, `PAY_2`, ..., `PAY_6` — which reflect the repayment status for each month, and corresponding repayment amounts (`PAY_AMT1` to `PAY_AMT6`). The target variable, `default payment next month`, indicates whether the client failed to meet their minimum payment requirement in the following billing cycle.

These features provide a structured foundation for modeling credit risk and evaluating the effect of hyperparameter tuning on classification performance.

The analysis begins with loading, cleaning, and preprocessing the dataset, followed by a discussion of the Random Forest model, which will serve as the basis for subsequent experiments.

1.2 Data Import and Initial Adjustments

The following libraries are imported to enable data manipulation, visualization, and model development:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

from sklearn.ensemble import RandomForestClassifier
from sklearn.utils import resample
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, roc_auc_score, roc_curve
from google.colab import drive
```

The dataset is provided in Excel format and is imported using the pandas library.

```
drive.mount('/content/drive')
df = pd.read_excel('/content/drive/MyDrive/default of credit card clients.xls', header=1)
df.head(10)
```

```
Mounted at /content/drive
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	...	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1	PAY_AMT2
0	1	20000	2	2	1	24	2	2	-1	-1	...	0	0	0	0	689
1	2	120000	2	2	2	26	-1	2	0	0	...	3272	3455	3261	0	1000
2	3	90000	2	2	2	34	0	0	0	0	...	14331	14948	15549	1518	1500
3	4	50000	2	2	1	37	0	0	0	0	...	28314	28959	29547	2000	2019
4	5	50000	1	2	1	57	-1	0	-1	0	...	20940	19146	19131	2000	36681
5	6	50000	1	1	2	37	0	0	0	0	...	19394	19619	20024	2500	1815
6	7	500000	1	1	2	29	0	0	0	0	...	542653	483003	473944	55000	40000
7	8	100000	2	2	2	23	0	-1	-1	0	...	221	-159	567	380	601
8	9	140000	2	3	1	28	0	0	2	0	...	12211	11793	3719	3329	0
9	10	20000	1	3	2	35	-2	-2	-2	-2	...	0	13007	13912	0	0

Column names are standardized for easier manipulation. Spaces are replaced with underscores, and all names are converted to lowercase:

```
df.columns = df.columns.str.strip().str.replace(' ', '_').str.lower()
```

The target variable is renamed for clarity, and the identifier column `id` is removed, as it holds no predictive value:

```
df.rename(columns={'default_payment_next_month': 'default'}, inplace=True)
df.drop(columns='id', inplace=True)
```

Missing values are not present in the dataset, which simplifies the preprocessing phase. Certain features, such as `SEX`, `EDUCATION`, and `MARRIAGE`, represent categorical data encoded numerically. These are explicitly cast to the categorical type to reflect their nature:

```
categorical_columns = ['sex', 'education', 'marriage']
df[categorical_columns] = df[categorical_columns].astype('category')
```

1.3 Identification of Invalid and Missing Data

Although the dataset contains no explicit missing values (`NaN`), a closer inspection of categorical variables reveals the presence of values that do not conform to expected categories.

The unique values of selected categorical variables are inspected:

```
print(df['sex'].unique())
print(df['marriage'].unique())
print(df['education'].unique())
print(df['age'].unique())
```

```
[2, 1]
Categories (2, int64): [1, 2]
[1, 2, 3, 0]
Categories (4, int64): [0, 1, 2, 3]
[2, 1, 3, 5, 4, 6, 0]
Categories (7, int64): [0, 1, 2, 3, 4, 5, 6]
[24 26 34 37 57 29 23 28 35 51 41 30 49 39 40 27 47 33 32 54 58 22 25 31
 46 42 43 45 56 44 53 38 63 36 52 48 55 60 50 75 61 73 59 21 67 66 62 70
 72 64 65 71 69 68 79 74]
```

To verify the absence of null entries:

```
print(len(df[pd.isnull(df['sex'])]))
print(len(df[pd.isnull(df['marriage'])]))
print(len(df[pd.isnull(df['education'])]))
print(len(df[pd.isnull(df['age'])]))
```

```
0
0
0
0
```

According to the dataset documentation, the `EDUCATION` variable contains invalid category codes—specifically `0`, `5`, and `6` —which do not correspond to any defined level of education and are interpreted as **ambiguous or missing entries**. Similarly, the `MARRIAGE` variable includes the invalid code `0`, which is not associated with any recognized marital status. These values are considered **placeholders for undefined or unknown information**, and their presence must be quantified and addressed during preprocessing to ensure the integrity of categorical feature representations.

```
len(df.loc[(df['education'] == 0) | (df['marriage'] == 0)])
```

```
68
```

Rows containing these invalid entries are then removed from the dataset:

```
df = df.loc[(df['education'] != 0) & (df['marriage'] != 0)]
len(df)
```

```
29932
```

The resulting dataset contains **29.932** valid observations.

1.4 Class Imbalance and Downsampling

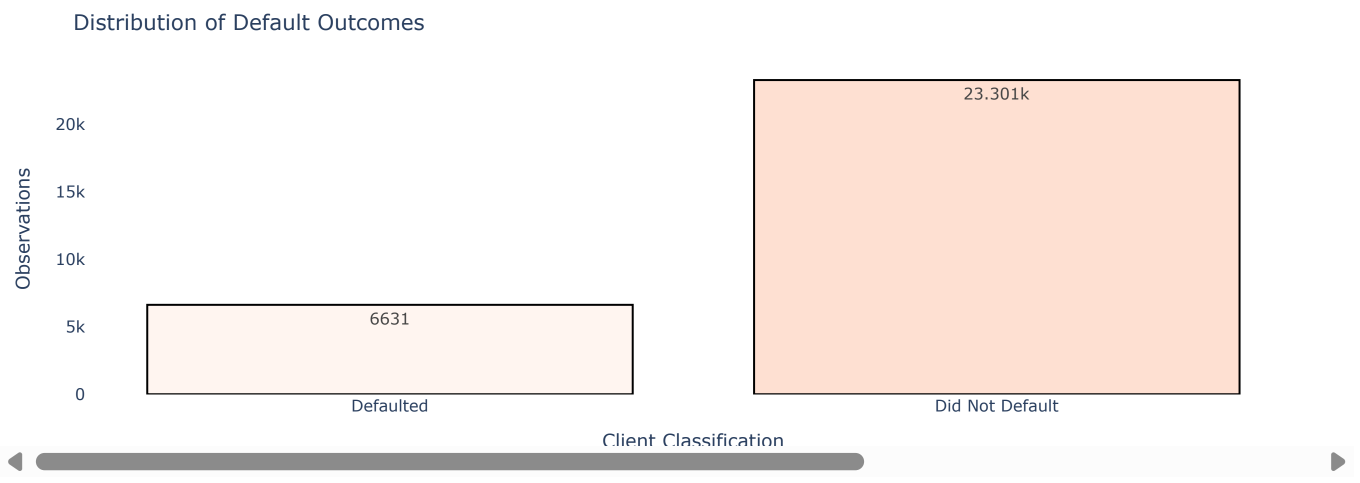
An analysis of the target variable reveals a substantial imbalance between defaulting and non-defaulting clients. To visualize this distribution:

```
df_plot = df.copy()
df_plot['default_label'] = df_plot['default'].map({0: 'Did Not Default', 1: 'Defaulted'})

fig = px.histogram(
    df_plot,
    x='default_label',
    color='default_label',
    color_discrete_sequence=px.colors.sequential.Reds,  # Valid Plotly palette
    text_auto=True,
    title='Distribution of Default Outcomes'
)

fig.update_layout(
    width=1000,
    height=350,
    showlegend=False,
    title_font_size=16,
    font=dict(size=12),
    margin=dict(l=40, r=30, t=50, b=40),
    xaxis_title='Client Classification',
    yaxis_title='Observations',
    plot_bgcolor='white'
)

fig.update_traces(marker_line_width=1.5, marker_line_color='black')
fig.show()
```



To address the **substantial class imbalance** observed in the target variable `default` — where the majority of clients did not default on their credit card payments — we adopt a **downsampling strategy** to ensure balanced representation across both classes. This step is crucial because machine learning algorithms are typically biased toward the majority class, which can result in misleading performance metrics and poor predictive power for the minority class (i.e., clients who defaulted).

The process begins by splitting the dataset into two subsets based on the target outcome:

- Clients who did not default (`default = 0`)
- Clients who did default (`default = 1`)

We then determine the number of observations in the minority class—in this case, the number of clients who defaulted. This count is used as a benchmark for resampling the majority class to ensure parity. Specifically, we randomly select a matching number of non-defaulting clients (without replacement) from the majority class.

By combining the full set of defaulters with the downsampled non-defaulters, we obtain a **new balanced dataset** in which both classes are equally represented. This balanced dataset improves fairness during model training and allows for more meaningful interpretation of metrics like precision, recall, and AUC. Finally, the data is shuffled to avoid any ordering bias before splitting it into features (`X`) and target labels (`y`) for model development.

```
# Split into the two classes
df_no_default = df[df['default'] == 0]
df_default = df[df['default'] == 1]  # keep all of these

# Count defaulters to match size
n_defaulters = len(df_default)

# Downsample the non-default class to match
df_no_default_downsampled = resample(
    df_no_default,
    replace=False,
    n_samples=n_defaulters,
    random_state=42
)

# Combine into a balanced dataset
df_downsample = pd.concat([df_default, df_no_default_downsampled], ignore_index=True)

# Prepare features and target variable
X = df_downsample.drop('default', axis=1).copy()
y = df_downsample['default'].copy()
```

1.5 One-Hot Encoding of Categorical Variables

The dataset contains several variables that represent the client's payment status over the past six months: `pay_0`, `pay_2`, `pay_3`, `pay_4`, `pay_5`, and `pay_6`.

Although these variables are stored as integers, they represent **categorical statuses** (e.g., no delay, 1-month delay, 2-month delay, etc.), not meaningful numeric quantities. Treating them as ordinal could lead the model to assume a false relationship between categories — for example, interpreting a 2-month delay as being exactly twice as severe as a 1-month delay.

To prevent this, we apply **one-hot encoding**, which creates a separate binary column for each unique status:

```
X_encoded = pd.get_dummies(X, columns=['pay_0', 'pay_2', 'pay_3', 'pay_4', 'pay_5', 'pay_6'])
```

This allows the model to treat each payment status as a distinct, non-ordered category, which is especially important for models like Random Forests that are sensitive to the way input features are structured.

By using one-hot encoding, we ensure the model captures the nature of the payment history correctly — without introducing misleading assumptions about order or magnitude.

1.6 Final Train-Test Split

The dataset is then divided into training and test sets using a 70/30 split:

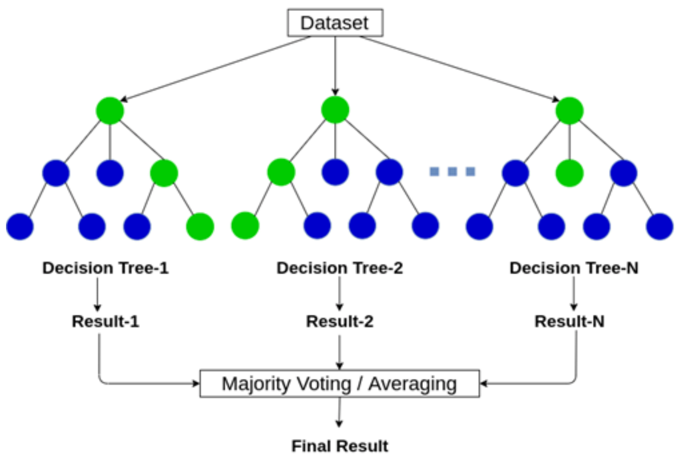
```
X_train, X_test, y_train, y_test = train_test_split(
    X_encoded, y, test_size=0.3, random_state=42
)
```

Since Random Forest models are not sensitive to the scale of the features, **feature normalization is not applied**.

1.7 Random Forest Classifier: Conceptual Overview

The **Random Forest algorithm** is a widely used and highly effective method in machine learning, particularly for classification and regression tasks. It belongs to a broader family of techniques known as **ensemble learning**, where multiple models are combined to produce more reliable and accurate predictions than any individual model alone.

Random Forest is based on a concept called **bagging** (short for bootstrap aggregating), which helps reduce the variability in predictions. The idea is to build many different models, each trained on a **randomly selected subset of the data**. These subsets are drawn **with replacement**, meaning some records may appear more than once while others may not appear at all in a given subset. Each model learns different patterns from its own unique subset, and the final prediction is made by **aggregating** their outputs — typically using majority voting for classification or averaging for regression.



Instead of relying on a single predictive model, **Random Forest** builds an ensemble of simpler models called **decision trees**, each trained on a different random subset of the training data (a bootstrap sample). These trees make predictions by following a sequence of **yes/no questions** about the input features — for example, whether a client's credit limit exceeds a certain threshold, or whether they've missed a payment.

At each split in the tree, the algorithm selects the best question from a randomly chosen subset of features, aiming to divide the data into groups that are as pure as possible — meaning that the resulting groups contain mostly one class (such as defaulters or non-defaulters). This process continues until a **leaf node** is reached, where the prediction is assigned according to the **most frequent class among the training samples that ended up in that node**.

Once all trees are trained, **each data point from the test set is passed independently through every tree in the forest**. Each tree produces its own prediction, and the final output of the Random Forest is determined by **majority vote** — that is, the class that receives the most votes across all trees.

This ensemble approach improves performance by reducing overfitting and increasing robustness, as the diversity among trees ensures that errors made by individual models are less likely to align.

The success of a Random Forest depends on two main factors: the **quality of the individual trees** (how accurate each one is) and the **diversity among them** (how uncorrelated their errors are). Combining strong but diverse models typically leads to better performance than relying on a single model.

Random Forests are especially popular in applied settings because they are **robust, flexible, and require minimal data preprocessing**. They can handle both numerical and categorical inputs, manage missing values, and are resistant to outliers. Additionally, they offer built-in tools for assessing the importance of different features in making predictions.

However, to get the best results, it is important to **tune the model's hyperparameters** — the settings that govern how the trees are built and how the forest operates as a whole. These include:

- `n_estimators` : the number of trees in the forest,
- `max_depth` : the maximum depth of each tree,
- `min_samples_split` : the minimum number of samples needed to split a node,
- `max_features` : the number of features to consider when looking for the best split.

Choosing the right values for these parameters involves a trade-off between **underfitting** (when the model is too simple) and **overfitting** (when the model is too complex and captures noise). In this project, we will evaluate two strategies for tuning these parameters:

- **Grid Search**, which systematically tests all combinations in a predefined set,
- **Randomized Search**, which randomly samples combinations from a defined range.

In the next chapter, we begin with a **baseline Random Forest model** using default settings to establish a reference point for comparing the impact of these tuning methods.

Chapter 2 - Baseline Model Without Hyperparameter Tuning

2.1 Overview and Rationale

Before exploring the impact of hyperparameter tuning, a **baseline evaluation** is conducted using a Random Forest classifier configured with default parameters. This initial model serves as a benchmark, providing a reference point to assess the extent of improvement achievable through optimization.

The classifier is initialized using the default settings of the `RandomForestClassifier` class from the `scikit-learn` library. These include:

- `n_estimators = 100` : This parameter defines the number of decision trees in the forest. A higher number generally improves stability and performance, as it allows more averaging, but it also increases computational cost.
- `criterion = 'gini'` : This specifies the function used to measure the quality of a split at each decision node. The Gini impurity is calculated as the probability of incorrectly classifying a randomly chosen element from the dataset if it were labeled according to the distribution of labels in the subset.
- `max_depth = None` : With this setting, each tree in the forest is grown until all leaves are pure or contain fewer samples than required for further splitting. While this allows the trees to fully capture complex relationships in the data, it can also increase the risk of overfitting.
- `max_features = 'sqrt'` : This parameter controls the number of features to consider when looking for the best split. The square root of the total number of features is used by default, which helps to reduce correlation among trees and promotes diversity in the ensemble.

These default values are designed to offer strong out-of-the-box performance across a wide range of classification tasks. However, they are not tailored to the specific distribution, scale, or class balance of the dataset under analysis. As such, this baseline model allows us to establish a meaningful point of comparison before applying more targeted hyperparameter tuning strategies in the subsequent chapters.

2.2 Model Training and Prediction

The Random Forest classifier is initialized and trained on the training data without any custom tuning:

```
rf_baseline = RandomForestClassifier(random_state=42)
rf_baseline.fit(X_train, y_train)
```

```
▾ RandomForestClassifier
RandomForestClassifier(random_state=42)
```

Once the model is trained, predictions on the test set are generated in two formats:

- `y_pred` : the predicted class labels (`0` for non-default, `1` for default), determined by majority voting across all trees.
- `y_prob` : the predicted probability of class `1` (default), calculated as the fraction of trees that voted for class `1`. These probabilities are used for evaluating probabilistic metrics such as ROC AUC.

```
y_pred = rf_baseline.predict(X_test)
y_prob = rf_baseline.predict_proba(X_test)[:, 1]
```

This dual prediction output allows for both discrete classification evaluation and threshold-independent performance analysis.

2.3 Performance Evaluation

The baseline model's performance is assessed using several standard classification metrics, including **accuracy**, **precision**, **recall**, and **F1-score**. These provide a comprehensive understanding of the model's classification ability.

Let:

- TP = True Positives (defaults correctly predicted as defaults)
- TN = True Negatives (non-defaults correctly predicted as non-defaults)
- FP = False Positives (non-defaults incorrectly predicted as defaults)
- FN = False Negatives (defaults incorrectly predicted as non-defaults)

Then the metrics are defined as:

- **Accuracy** (proportion of correct predictions over all observations):

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = \frac{\text{Correct Predictions}}{\text{Total Observations}}$$

- **Precision** (correctly predicted defaults out of all predicted defaults):

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- **Recall** (correctly predicted defaults out of all actual defaults):

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- **F1-score** (harmonic mean of precision and recall):

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

```
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2%}')
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
Accuracy: 70.47%
Classification Report:
               precision    recall  f1-score   support

           0       0.67      0.78      0.72      1970
           1       0.75      0.63      0.68      2009

    accuracy                           0.70      3979
   macro avg       0.71      0.71      0.70      3979
weighted avg       0.71      0.70      0.70      3979
```

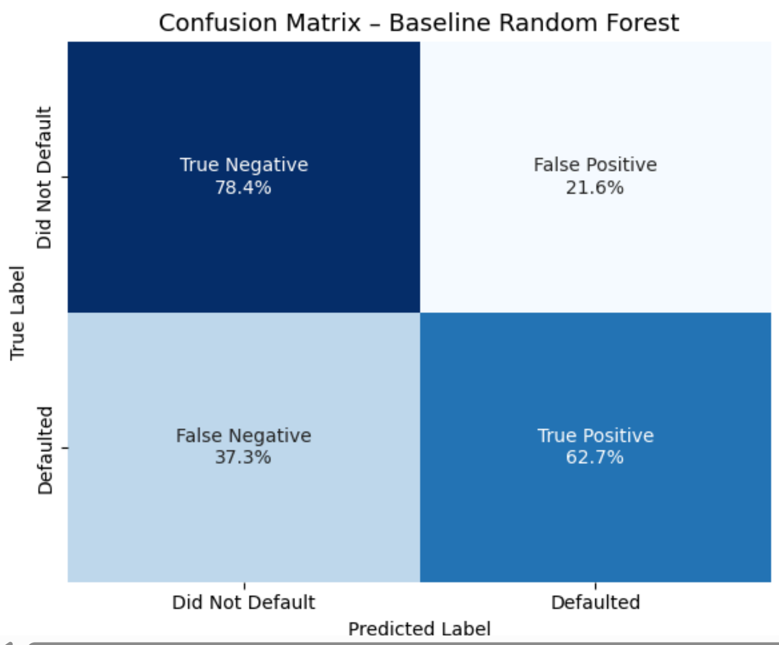
To complement these metrics, a **confusion matrix** is visualized using an enhanced heatmap. This matrix provides a detailed breakdown of the model's predictions by showing the number of true positives, true negatives, false positives, and false negatives — offering insight into specific types of classification errors.

```
# Compute confusion matrix and normalize
cm = confusion_matrix(y_test, y_pred)
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

# Define labels: class name + percentage only
group_names = ['True Negative', 'False Positive', 'False Negative', 'True Positive']
group_percentages = [f"{value:.1%}" for value in cm_normalized.flatten()]
labels = [f"{name}\n{percent}" for name, percent in zip(group_names, group_percentages)]
labels = np.asarray(labels).reshape(2, 2)

# Plot confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(cm_normalized, annot=labels, fmt='', cmap='Blues', cbar=False,
            xticklabels=['Did Not Default', 'Defaulted'],
            yticklabels=['Did Not Default', 'Defaulted'])

plt.title('Confusion Matrix – Baseline Random Forest', fontsize=13)
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.tight_layout()
plt.show()
```



2.4 ROC Curve and AUC Score

In addition to discrete performance metrics, **Receiver Operating Characteristic (ROC) analysis** is used to evaluate the model's probabilistic predictions. The **ROC curve** illustrates how the model's classification performance changes as the decision threshold is varied. It plots the True Positive Rate (TPR) on the y-axis against the False Positive Rate (FPR) on the x-axis.

To compute the ROC curve:

- The model's predicted probabilities (`y_prob`) for the positive class (e.g., default = 1) are used.
- These probabilities are compared against a range of thresholds between 0 and 1.
- For each threshold, predicted labels are assigned:
  If the probability > threshold → predict 1 (default), else predict 0.
- At each threshold, the following are computed:

$$\text{True Positive Rate (TPR)} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad \text{(Recall)}$$
$$\text{False Positive Rate (FPR)} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

Each point on the ROC curve corresponds to a different classification threshold. As the threshold decreases, more instances are classified as positive, increasing both TPR and FPR.

The grey dashed line represents the performance of a random classifier, where TPR = FPR for all thresholds. A model that performs better than random will have a ROC curve that lies above this diagonal.

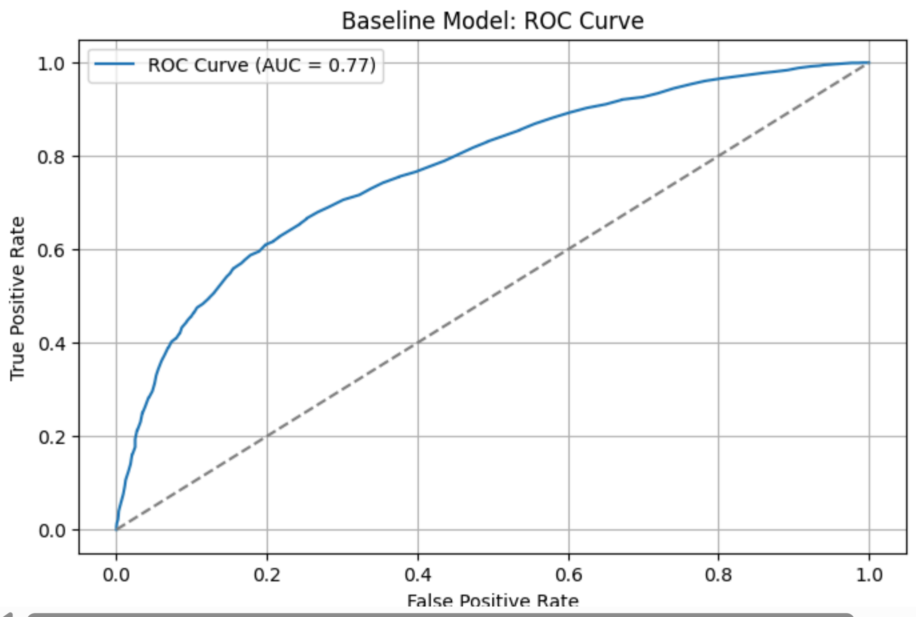
The **Area Under the Curve (AUC)** provides a single-number summary of the model's ability to distinguish between the two classes. An AUC of **1.0** indicates perfect separation, while an AUC of **0.5** suggests no better performance than random guessing. A curve that closely follows the **top-left corner** of the plot reflects strong classification performance and high separability between the classes.

```
auc_score = roc_auc_score(y_test, y_prob)
print("ROC AUC Score:", round(auc_score, 3))

fpr, tpr, thresholds = roc_curve(y_test, y_prob)

plt.figure(figsize=(8, 5))
plt.plot(fpr, tpr, label=f'ROC Curve (AUC = {auc_score:.2f})')
plt.plot([0, 1], [0, 1], linestyle='--', color='grey')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Baseline Model: ROC Curve')
plt.legend()
plt.grid(True)
plt.show()
```

```
ROC AUC Score: 0.772
```



2.5 Discussion

The baseline Random Forest classifier demonstrates a moderate ability to distinguish between defaulting and non-defaulting clients, achieving an overall accuracy of **70.47%**. However, accuracy alone can be misleading in classification problems where the costs of false positives and false negatives are not symmetric. Therefore, a more comprehensive evaluation is required.

The confusion matrix illustrates the distribution of predictions as follows:

- **True Negatives** and **True Positives** are reasonably well captured, indicating a generally effective distinction between the two classes.
- However, a non-negligible number of **False Negatives** and **False Positives** persist, suggesting that the model still struggles to fully capture all instances of defaulters.

The classification report highlights a **precision of 0.72** for the defaulting class, meaning that 72% of clients predicted to default actually did so. The **recall for this class is 0.62**, indicating that 62% of actual defaulters were successfully identified. The **F1-score** for the defaulting class, which balances precision and recall, is **0.67**, suggesting a moderate but not exceptional performance.

The most critical weakness remains the number of **False Negatives**, which correspond to defaulters that are not identified by the model. These misclassifications can lead to increased financial risk and highlight an area of concern for institutions that rely on early identification of potential credit defaults.

In support of these findings, the **ROC AUC score of 0.772** further indicates a moderate discriminative capacity. This value means that, when presented with a randomly selected defaulter and non-defaulter, the model has a 77.2% chance of ranking the defaulter as riskier.

In summary, the untuned Random Forest classifier provides a **strong initial benchmark**, especially given the use of a downsampled and balanced dataset. Nonetheless, the results suggest that further improvements — particularly in recall for the minority (defaulting) class — are both desirable and necessary. These limitations motivate the use of hyperparameter optimization techniques, beginning with **Grid Search**, as explored in the next chapter.

Chapter 3 – Hyperparameter Tuning with Grid Search

3.1 Introduction and Theoretical Background

The performance of a machine learning model depends not only on the structure of the learning algorithm but also on the configuration of its hyperparameters—those parameters that control how the algorithm learns but are not themselves learned from the data. In the case of Random Forest classifiers, hyperparameters govern aspects such as the number and depth of decision trees, the size of feature subsets used at each split, and the rules that regulate node splitting.

Although the default hyperparameters provided by the scikit-learn implementation often yield reasonable results, they are not necessarily optimal for any given dataset. To address this, systematic hyperparameter tuning is required. One of the most common and rigorous approaches for this purpose is **Grid Search**, a method that exhaustively explores a predefined hyperparameter space to identify the combination that yields the best performance according to a user-specified evaluation metric.

Formally, let $\Theta = \theta_1 \times \theta_2 \times \cdots \times \theta_k$ represent the Cartesian product of k hyperparameters, where each θ_i is a finite set of candidate values for the i-th hyperparameter. Grid Search evaluates the learning algorithm $f(\cdot \mid \theta)$ for each $\theta \in \Theta$ using a cross-validation strategy. For a given split of the data into training and validation sets, the performance is assessed using a scoring function $\mathcal{L}(\theta)$—in this case, the area under the Receiver Operating Characteristic curve (ROC AUC).

The goal is to solve:

$$\theta^* = \arg\max_{\theta \in \Theta} \mathcal{L}(\theta)$$

where θ^* denotes the hyperparameter combination that maximizes the model's average cross-validation score. Once the optimal combination is identified, the final model is retrained on the entire training set using these settings. Grid Search is exhaustive and deterministic, making it a reliable tool for small to moderately sized hyperparameter spaces. However, its primary limitation is **computational cost**, which increases exponentially with the number of parameters and candidate values.

3.2 Selection of Hyperparameters

To apply Grid Search effectively, it is essential to identify a set of hyperparameters that significantly influence the performance, complexity, and generalization capacity of the Random Forest model.

We do it by creating a **dictionary of parameters** with keys equal to the parameters that must be tuned and values equal to a list with all the possible combinations of values.

The following hyperparameters are selected for tuning:

- **`n_estimators`** : The number of trees in the ensemble. A larger number tends to increase predictive performance due to averaging but also adds to computational time. Random Forests are relatively robust to overfitting with respect to this parameter.
- **`max_depth`** : The maximum depth allowed for each decision tree. Limiting depth prevents the model from creating overly complex trees that fit noise, thereby reducing variance. Deeper trees can capture complex patterns but risk overfitting. If None, then nodes are expanded until all leaves are *pure* or until all leaves contain less than *min_samples_split samples*.
- **`min_samples_split`** : The minimum number of samples required to split an internal node. Before the algorithm decides to split a node (i.e., ask another question), it checks how many samples are in that node. If the number of samples is less than `min_samples_split`, then it stops splitting and that node becomes a leaf. Increasing this value prevents small, potentially noisy splits and acts as a regularizer by controlling tree complexity.
- **`min_samples_leaf`** : The minimum number of samples required to be in a leaf node. This helps to prevent overly specific leaf nodes and reduces variance by enforcing a minimum sample size per terminal region.
- **`max_features`** : The number of features considered when determining the best split. By randomly restricting the feature set at each node, this parameter introduces diversity among trees, improving generalization. Typical values include `'sqrt'` and `'log2'`. This is a form of regularization — it adds randomness and reduces overfitting, especially in ensemble methods like random forests and when we decide to use low values of features. **For example**: if we have 100 features and the tuned `max_features` is "sqrt", we are going to use only 10 variables.
- **`bootstrap`** : A Boolean parameter indicating whether bootstrap samples are used to train each tree. Setting this to `False` removes sampling with replacement and can improve performance in certain settings by reducing redundancy across trees.

  Here is the documentation for further explanation.

  The hyperparameter grid is defined as follows:

```
param_grid = {
    'n_estimators': [100, 300, 500],
    'max_depth': [10, 15, 20, None],
    'max_features': ['sqrt', 'log2'],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 5],
    'bootstrap': [True, False]
}
```

This yields a total of 576 combinations (3 × 4 × 2 × 4 × 3 × 2) to be evaluated

3.3 Implementation of Grid Search

The hyperparameter tuning procedure is implemented using the `GridSearchCV` class provided by the `scikit-learn` library. This utility automates the process of performing an exhaustive search over the predefined hyperparameter space by training and validating the model on every possible parameter combination using cross-validation. For each combination, the model is evaluated on a specific performance metric—in this case, the **ROC AUC score**—and the configuration that yields the best average score across all validation folds is selected.

The tuning process begins by initializing a base Random Forest classifier:

```
rf_grid = RandomForestClassifier(random_state=42)
```

Setting the `random_state` ensures that results are reproducible, which is particularly important in the context of stochastic ensemble methods.

Next, the `GridSearchCV` object is instantiated with the following configuration:

`GridSearchCV` has as specifications **estimator**, **param_grid**, **scoring**, **cv**, **n_jobs**, **verbose.**

`estimator` is assumed to implement the scikit-learn estimator interface.. In this case the estimator will be `rf` that stands for random forest. Other estimators will be for classification tasks (as ours) LogisticRegression, SVC (Support Vector Classifier), GradientBoostingClassifier, XGBClassifier, LGBMClassifier (from external libraries), DecisionTreeClassifier.

In addition, we could have regression estimators and the KMeans clustering cases.

The `param_grid` takes all the parameters indicated above that should be tuned and on which we apply the grid search function.

In the specific `scoring` can get different evaluation metrics like accuracy, roc_auc, f1 score, R2 and so on. The metrics to be added are indicated here. Below we have added only the `roc_auc`, but if the goal is getting different metrics and hence a multiscore evaluation, we can use list, tuples or dictionaries.

Then, `cv` is related to the k-folds cross validation: so, if we have chosen 5 the model in a first glance is applied on 4 folds and then it is tested on the fifth and it iterates the process lefting one out each time.

`n_jobs`, instead, is the number of jobs to run in parallel across all available CPU cores.and eventually we have `verbose` that displays progress information during the search process, including which combination is currently being evaluated and the associated cross-validation scores.

All the other information about the python GridSearchCV are linked here in the scikit-learn documentation.

```
from sklearn.model_selection import GridSearchCV

grid_search = GridSearchCV(
    estimator=rf_grid,
    param_grid=param_grid,
    scoring='roc_auc',    # Evaluation metric
    cv=5,                 # 5-fold cross-validation
    n_jobs=-1,            # Use all available cores
    verbose=1
)

grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 576 candidates, totalling 2880 fits
▸            GridSearchCV

▸       best_estimator_:
      RandomForestClassifier

     ▸ RandomForestClassifier
```

In our specific case we have chosen:

- **estimator**: The machine learning model to be tuned. In this case, it is the base `RandomForestClassifier` with default settings, except for the `random_state` parameter, which ensures reproducibility of results.
- **param_grid**: A dictionary containing the hyperparameters and the set of candidate values to be explored. As previously defined, this grid yields 576 unique combinations derived from six hyperparameters, allowing for a comprehensive search of the parameter space.
- **scoring='roc_auc'**: Specifies the evaluation metric used to rank models. The Area Under the ROC Curve (AUC) is selected because it provides a robust measure of the classifier's ability to distinguish between classes, even in the presence of class imbalance.
- **cv=5**: Defines the number of cross-validation folds. In 5-fold cross-validation, the training dataset is split into five parts. The model is trained on four folds and validated on the remaining one. This process is repeated five times, ensuring that each fold is used exactly once for validation.
- **n_jobs=-1**: This significantly speeds up the evaluation process, especially useful when the grid contains a large number of hyperparameter combinations.
- **verbose=1**

Once the configuration is complete, the grid search is executed using the `.fit()` method, which applies the tuning process to the training dataset.

The best parameter set and the corresponding model are then retrieved:

```
print("Best Parameters:", grid_search.best_params_)
best_rf = grid_search.best_estimator_
```

```
Best Parameters: {'bootstrap': True, 'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 5, 'min_samples_split': 20, 'n_e
```

The attribute `.best_estimator_` gets as parameters:

- `bootstrap` = **True**, hence the sampling with replacement has chosen. This adds variance reduction (bagging) and increases model robustness.;
- `max_depth` = **None**; it means that the tree grows until each leaf is pure (or stopping conditions like min_samples_split or min_samples_leaf are reached). We need to carefully look at this result, because we could end up with overfitting;
- `max_features` = **sqrt**, so it means we are not using all the features, but only the squared number of features. It basically applies the regularization techniques in order to retrieve them.
- `min_samples_leaf` = **5**, every leaf is going to host at least 5 samples, in order to avoid overfitting;
- `min_sample_split` = **20**, so if the number of samples in the node is lower than 20, it becomes a leaf;
- `n_estimators` = **500** are the number of trees created. In general this is a good number.

3.4 Model Evaluation After Tuning

Once the best hyperparameter configuration is identified through Grid Search, the corresponding model is retrained on the full training set and evaluated on the test set. The predicted class labels and class probabilities are computed as follows:

```
y_pred_gs = best_rf.predict(X_test)
y_prob_gs = best_rf.predict_proba(X_test)[:, 1]
```

The model's classification performance is first assessed using **accuracy**, **precision**, **recall**, and **F1-score**:

```
accuracy_gs = accuracy_score(y_test, y_pred_gs)
print(f'Accuracy: {accuracy_gs:.2%}')
print("Classification Report:\n", classification_report(y_test, y_pred_gs))
```

```
Accuracy: 71.48%
Classification Report:
               precision    recall  f1-score   support

           0       0.68      0.80      0.74      1970
           1       0.77      0.63      0.69      2009

    accuracy                           0.71      3979
   macro avg       0.72      0.72      0.71      3979
weighted avg       0.72      0.71      0.71      3979
```

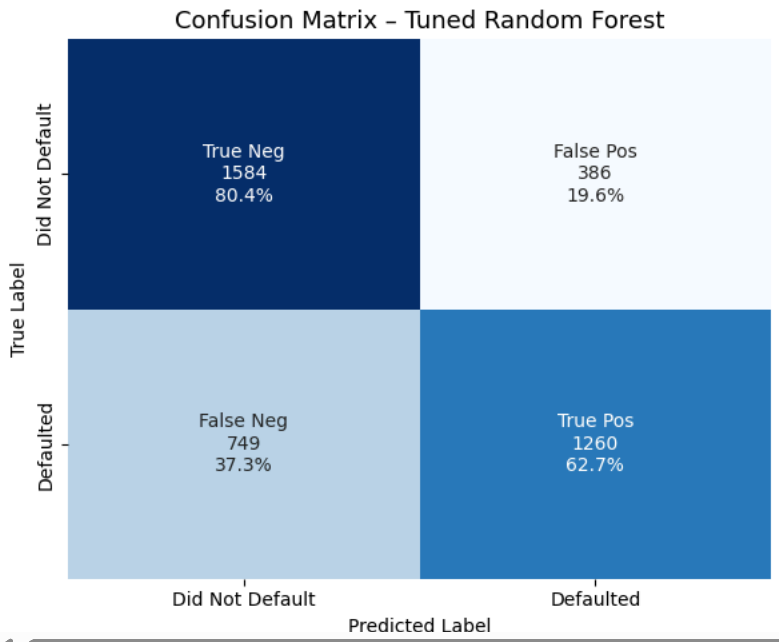
Then again, for GrdiSearch RF we plot the **confusion matrix** through an enhanced heatmap. This plot displays both the raw counts and the corresponding percentages within each class:

```
# Compute confusion matrix
cm_gs = confusion_matrix(y_test, y_pred_gs)
cm_normalized_gs = cm_gs.astype('float') / cm_gs.sum(axis=1)[:, np.newaxis]

# Define labels
group_names = ['True Neg','False Pos','False Neg','True Pos']
group_counts = [f"{value:0}" for value in cm_gs.flatten()]
group_percentages = [f"{value:.1%}" for value in cm_normalized_gs.flatten()]
labels = [f"{name}\n{count}\n{percent}" for name, count, percent in zip(group_names, group_counts, group_percentages)]
labels = np.asarray(labels).reshape(2, 2)

# Plot heatmap
plt.figure(figsize=(6, 5))
sns.heatmap(cm_normalized_gs, annot=labels, fmt='', cmap='Blues', cbar=False,
            xticklabels=['Did Not Default', 'Defaulted'],
            yticklabels=['Did Not Default', 'Defaulted'])

plt.title('Confusion Matrix - Tuned Random Forest', fontsize=13)
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.tight_layout()
plt.show()
```



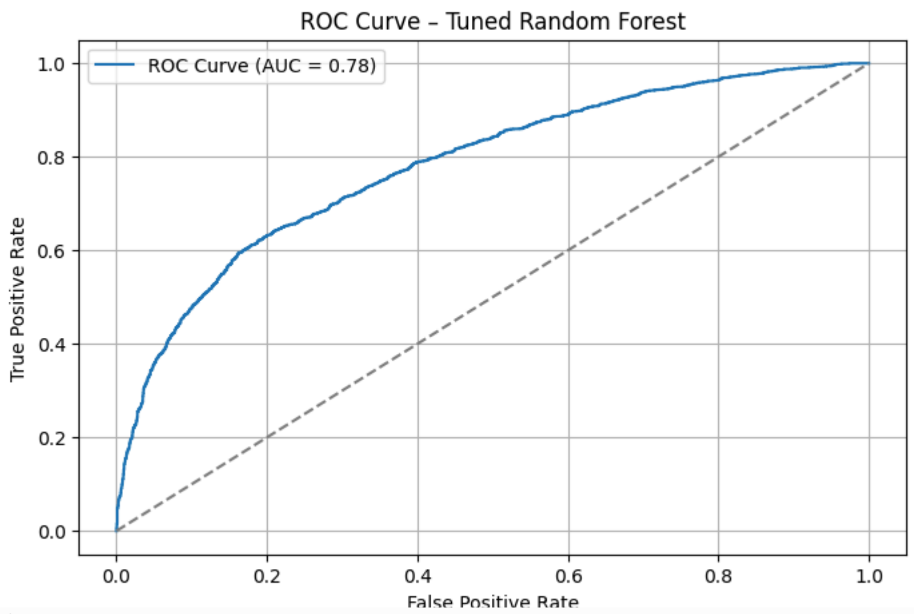
Finally, the **ROC AUC score** is computed and the **ROC curve** is plotted to assess the model's discriminative ability across different classification thresholds:

```
auc_gs = roc_auc_score(y_test, y_prob_gs)
print("ROC AUC Score (Tuned):", round(auc_gs, 3))

fpr_gs, tpr_gs, thresholds_gs = roc_curve(y_test, y_prob_gs)

plt.figure(figsize=(8, 5))
plt.plot(fpr_gs, tpr_gs, label=f'ROC Curve (AUC = {auc_gs:.2f})')
plt.plot([0, 1], [0, 1], linestyle='--', color='grey')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Tuned Random Forest')
plt.legend()
plt.grid(True)
plt.show()
```

```
ROC AUC Score (Tuned): 0.782
```



3.5 Discussion

The tuned Random Forest model with grid search demonstrates a measurable improvement in classification performance compared to the baseline model presented in Chapter 2. The overall accuracy increased from **70.47%** to **71.48%**, indicating that a higher proportion of total predictions are now correct when using the optimized hyperparameters identified through Grid Search.

The updated confusion matrix reveals the following:

- **True Negatives** increased to **1,584 (80.4%)**, reflecting an improved ability to correctly identify clients who did not default.
- **True Positives** rose to **1,260 (62.7%)**, suggesting a more effective detection of actual defaulters.
- **False Negatives** remained relatively high at **749 (37.3%)**, indicating that many defaulters still go undetected. This is a common challenge in credit risk modeling and suggests that the model still **underestimates credit risk to some extent**. A significant portion of risky clients are **misclassified as safe**, which could lead to insufficient capital allocation or underestimation of portfolio risk. Our focus should be on this value here.
- **False Positives** reduced to **386 (19.6%)**, meaning fewer clients who would not have defaulted were incorrectly flagged as defaulters. This is important, as it limits unnecessary risk-averse decisions — such as denying credit or allocating excessive capital — for clients who are actually creditworthy.

  Although this number is currently at an acceptable level, it must always be monitored carefully. Overestimating defaults (i.e., a high false positive rate) could lead to **missed business opportunities or inefficient capital usage**. That said, from a **regulatory or conservative risk management** standpoint, overestimating risk can sometimes be acceptable — especially under specific circumstances — as it leads to more **prudent capital reserves and lower systemic exposure**.

Evaluation of class-specific metrics confirms these trends:

- **Precision** for the defaulting class (1) increased from **0.72** to **0.74**, meaning that a larger proportion of predicted defaulters were indeed correct. Besides overall accuracy, precision offers valuable insight by measuring the ratio of true positives to the sum of true and false positives. Hence, increasing that measure is good for the modelling purposes since 74% of the clients were correctly predicted as defaulters. But unfortunately the remaining part is the number of clients that have not defaulted but the model has flagged as "defaulters". **This consideration must be done in order to avoid great credit risks penalizations**.
- **Recall** improved from **0.62** to **0.64**, demonstrating a modest gain in the model's sensitivity to actual defaulters. This metric measures the proportion of true positives among all actual defaulters, i.e., **how many defaulters the model successfully detected**. While an improvement is noted, the value still indicates that around 36% of defaulters go undetected. **This underlines a potential risk in credit assessment, and such limitations should be carefully considered when making risk-based decisions or capital requirement adjustments**.
- **F1-score**, which balances precision and recall, increased from **0.67** to **0.68**, indicating a slightly more stable and balanced classification performance. The model has a moderately good balance between correctly **identifying defaulters and not over-predicting them** — but there's still room to improve either precision, recall, or both.

Importantly, the **ROC AUC score increased** from **0.772** to **0.782**, confirming that the tuned model not only performs better at the default classification threshold (0.5), but also improves in its overall **ranking ability across thresholds**, hence it improves the ability to identify correctly the classes and it is an essential attribute for risk-based decision systems.

In conclusion, the Grid Search optimization procedure successfully enhanced key performance indicators, particularly in precision, recall, and overall accuracy. While the gains are moderate, they are meaningful and justify the computational cost involved in the tuning process. These results also underscore that **model optimization is multidimensional**, where improvements in some metrics may not always coincide with dramatic changes in others. As the search space becomes increasingly complex, more efficient tuning approaches, such as **Randomized Search**, may provide competitive results at a reduced computational cost. This is explored in the next chapter.

Chapter 4 – Hyperparameter Tuning with Randomized Search

4.1 Introduction and Theoretical Background

While **Grid Search** offers a rigorous and exhaustive approach to hyperparameter tuning, it becomes computationally expensive as the size of the hyperparameter space increases. For k hyperparameters, each with n_i candidate values, the total number of model evaluations required is given by the product:

$$|\Theta| = \prod_{i=1}^{k} |\theta_i|$$

This exponential growth in the number of configurations makes Grid Search impractical in settings involving **large, continuous, or high-dimensional hyperparameter spaces**, especially when resources are limited or model training is time-intensive.

To address this limitation, **Randomized Search** provides a more computationally efficient alternative. Instead of exhaustively exploring all possible combinations, it **samples a fixed number** of random combinations from the user-defined hyperparameter distributions. This stochastic nature makes it particularly well-suited when:

- The hyperparameter space is large or continuous,
- Computational time is constrained, or
- An approximate but sufficiently good solution is acceptable.

Formally, rather than evaluating all $\theta \in \Theta$, Randomized Search samples a subset $\tilde{\Theta} \subset \Theta$ of size N, where:

$$\tilde{\Theta} = \{\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(N)}\}, \quad \theta^{(j)} \sim \mathcal{D}(\Theta)$$

Here, $\mathcal{D}(\Theta)$ denotes the joint distribution or sampling strategy over the parameter space. Each sampled combination $\theta^{(j)}$ is then evaluated using a user-specified **scoring function**, typically through k-fold cross-validation, and the best-performing configuration is selected:

$$\theta^* = \arg\max_{\theta \in \tilde{\Theta}} \mathcal{L}(\theta)$$

The effectiveness of Randomized Search lies in its ability to **discover well-performing regions of the parameter space** without requiring exhaustive enumeration. Empirical studies have shown that it can outperform Grid Search when only a subset of hyperparameters significantly influence performance, as Grid Search may waste evaluations on unimportant parameters or redundant combinations.

In practice, the user specifies:

- A **dictionary** of hyperparameters with either fixed candidate values or sampling distributions,
- The number of **iterations** (i.e., combinations to evaluate),
- The **evaluation metric** (e.g., ROC AUC), and
- The **cross-validation** strategy (e.g., 5-fold CV).

Although it introduces a degree of randomness, Randomized Search can be **reproducible** by setting a `random_state` seed. Its efficiency and scalability make it particularly attractive in real-world applications, where computational resources are limited and exhaustive searches are infeasible. In the following section, this method will be implemented and compared against Grid Search to evaluate its practical effectiveness in tuning a Random Forest model.

4.2 Parameter Distributions and Search Strategy

The same set of hyperparameters used in the previous Grid Search is considered in this chapter; however, instead of exhaustively evaluating all possible combinations, Randomized Search samples a fixed number of configurations from **predefined ranges or distributions**. This approach dramatically reduces computational cost while still enabling an effective exploration of the hyperparameter space.

The sampling strategy is implemented using the `param_dist` dictionary, which includes a mix of discrete lists and probability distributions. The rationale for each hyperparameter choice is as follows:

- **`n_estimators`** : This time the number is randomly drawn from integers between 100 and 999 using `randint(100, 1000)` . This range allows variability in ensemble size without being overly restrictive or computationally excessive. We recall that this gives the number of trees in the forest.
- **`max_depth`** : Drawn from a fixed list `[10, 15, 20, None]` .This enables the model to explore both regularized and unbounded complexity.
- **`max_features`** : Sampled from `['sqrt', 'log2']` .
- **`min_samples_split`** : Sampled uniformly from integers between 2 and 20.
- **`min_samples_leaf`** : Sampled from integers between 1 and 10, which restricts the size of terminal leaf nodes and thus limits overfitting.
- **`bootstrap`** : A binary choice between `True` and `False` .

In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by **`n_iter`** .

Here there is the scikit-learn documentation.

By relying on statistical distributions and random sampling, Randomized Search can efficiently explore **high-dimensional or continuous spaces** where Grid Search would be computationally prohibitive. The technique is especially advantageous when computational resources are limited or when a near-optimal solution is sufficient for practical purposes.

In the implementation that follows, this strategy is used to sample a predefined number of configurations from the `param_dist` dictionary and identify the one that maximizes cross-validated ROC AUC performance.

```
from scipy.stats import randint

param_dist = {
    'n_estimators': randint(100, 1000),
    'max_depth': [10, 15, 20, None],
    'max_features': ['sqrt', 'log2'],
    'min_samples_split': randint(2, 21),
    'min_samples_leaf': randint(1, 11),
    'bootstrap': [True, False]
}
```

This hyperparameter space is then explored using `RandomizedSearchCV`, which performs a randomized search over the defined distributions for a fixed number of iterations. In this case, **50 configurations** that are n_int and they are sampled and evaluated using **5-fold cross-validation**. This number of iterations offers a practical balance between computational efficiency and sufficient coverage of the parameter space.

The 5-fold cross validation works as in the GridSearchCV. This process ensures that the selected model generalizes well and is not overfitting to any particular subset of the data.

`RandomizedSearchCV` thus identifies the hyperparameter configuration that **maximizes the average cross-validated ROC AUC score** across all 50 sampled combinations, providing a computationally efficient yet statistically robust approach to model tuning.

```
from sklearn.model_selection import RandomizedSearchCV

rf_random = RandomForestClassifier(random_state=42)

random_search = RandomizedSearchCV(
    estimator=rf_random,
    param_distributions=param_dist,
    n_iter=50,
    scoring='roc_auc',
    cv=5,
    n_jobs=-1,
    verbose=1,
    random_state=42
)

random_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
▸ RandomizedSearchCV
▸ best_estimator_: RandomForestClassifier
  ▸ RandomForestClassifier
```

After training, the best parameters can be retrieved:

```
print("Best Parameters (Randomized Search):", random_search.best_params_)
best_rf_random = random_search.best_estimator_
```

```
Best Parameters (Randomized Search): {'bootstrap': True, 'max_depth': 15, 'max_features': 'sqrt', 'min_samples_leaf': 10, 'min_sampl
```

4.3 Model Evaluation After Randomized Search

After identifying the best-performing hyperparameter configuration via Randomized Search, the resulting model is evaluated on the test set. The evaluation follows the same structure used in previous chapters to ensure consistent comparison.

First, the predicted class labels and corresponding class probabilities are obtained:

```
y_pred_rs = best_rf_random.predict(X_test)
y_prob_rs = best_rf_random.predict_proba(X_test)[:, 1]
```

The model's threshold-based performance is assessed using standard classification metrics.

```
accuracy_rs = accuracy_score(y_test, y_pred_rs)
print(f'Accuracy: {accuracy_rs:.2%}')
print("Classification Report:\n", classification_report(y_test, y_pred_rs))
```

```
Accuracy: 71.48%
Classification Report:
              precision    recall  f1-score   support
```

```
           0       0.68      0.80      0.74      1970
           1       0.77      0.63      0.69      2009

    accuracy                           0.71      3979
   macro avg       0.72      0.72      0.71      3979
weighted avg       0.72      0.71      0.71      3979
```

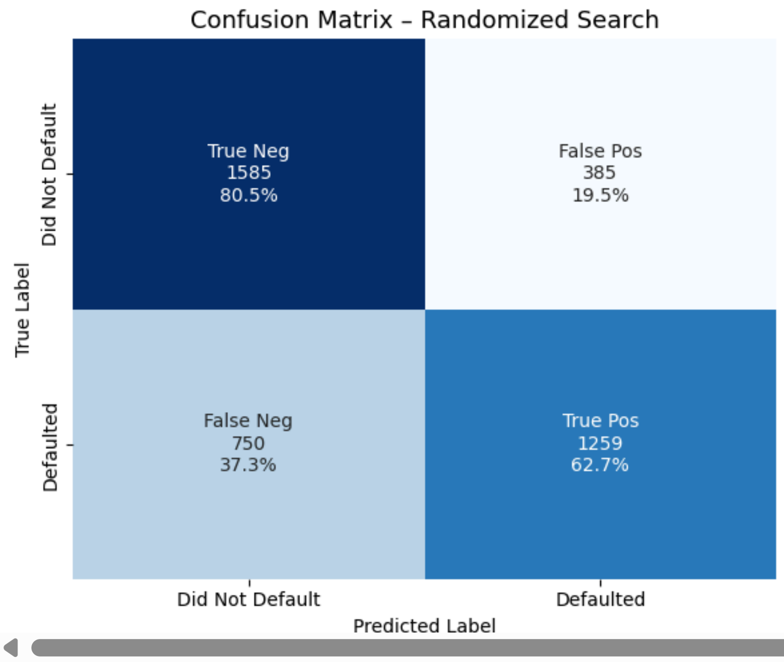
To better understand the distribution of predictions, the confusion matrix is computed and visualized using a heatmap.

```
cm_rs = confusion_matrix(y_test, y_pred_rs)
cm_normalized_rs = cm_rs.astype('float') / cm_rs.sum(axis=1)[:, np.newaxis]

group_names = ['True Neg','False Pos','False Neg','True Pos']
group_counts = [f"{value:0}" for value in cm_rs.flatten()]
group_percentages = [f"{value:.1%}" for value in cm_normalized_rs.flatten()]
labels = [f"{name}\n{count}\n{percent}" for name, count, percent in zip(group_names, group_counts, group_percentages)]
labels = np.asarray(labels).reshape(2, 2)

plt.figure(figsize=(6, 5))
sns.heatmap(cm_normalized_rs, annot=labels, fmt='', cmap='Blues', cbar=False,
            xticklabels=['Did Not Default', 'Defaulted'],
            yticklabels=['Did Not Default', 'Defaulted'])

plt.title('Confusion Matrix – Randomized Search', fontsize=13)
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.tight_layout()
plt.show()
```



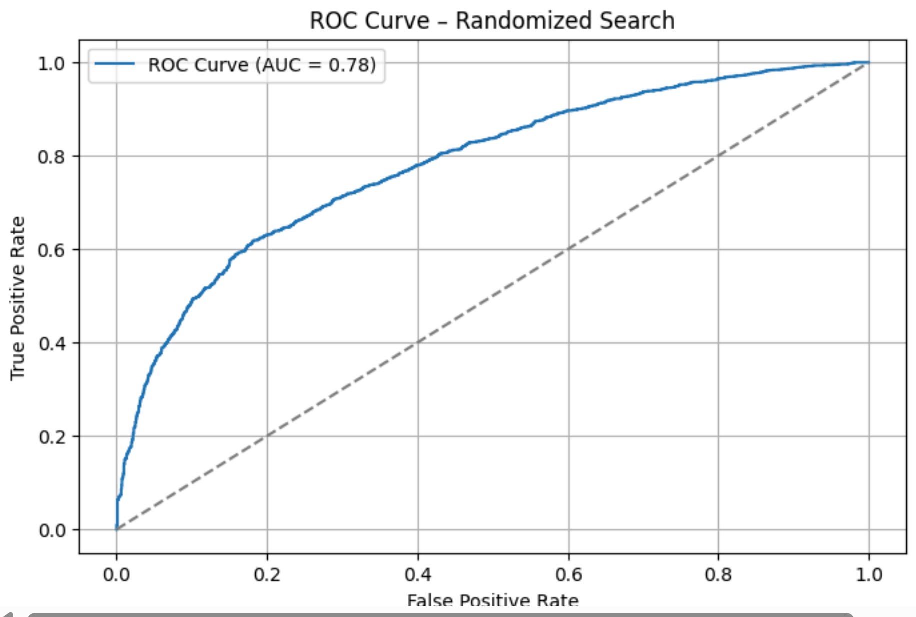
Finally, the Receiver Operating Characteristic (ROC) curve and the corresponding Area Under the Curve (AUC) score are computed.

```
auc_rs = roc_auc_score(y_test, y_prob_rs)
print("ROC AUC Score (Randomized):", round(auc_rs, 3))

fpr_rs, tpr_rs, thresholds_rs = roc_curve(y_test, y_prob_rs)

plt.figure(figsize=(8, 5))
plt.plot(fpr_rs, tpr_rs, label=f'ROC Curve (AUC = {auc_rs:.2f})')
plt.plot([0, 1], [0, 1], linestyle='--', color='grey')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve – Randomized Search')
plt.legend()
plt.grid(True)
plt.show()
```

```
ROC AUC Score (Randomized): 0.782
```



4.4 Discussion

The model trained using hyperparameters optimized via Randomized Search achieved results that are highly competitive with, and in some respects equivalent to, those obtained through Grid Search—despite requiring substantially fewer model evaluations. This reinforces the **efficiency and effectiveness** of Randomized Search, particularly in contexts where computational resources or execution time are constrained.

The **overall accuracy** reached **71.48%**, which exactly matches the performance obtained through Grid Search, indicating that the model exhibits similarly strong general classification capabilities.

An analysis of the classification report confirms the robustness of the results:

- The **precision** for the defaulting class (1) remained at **0.74**, identical to the Grid Search model.
- The **recall** for the same class was **0.63**, only marginally below the value observed wit GridSearchCV, indicating a slight change of the True Positives and False Negatives.
- The **F1-score** was also stable at **0.68**, maintaining a balanced trade-off between precision and recall.

These scores are reflected in the confusion matrix:

- **True Positives** were **1,259**, and **False Negatives** were **750**, nearly mirroring the tuned model's results.
- **False Positives** totaled **385**, and **True Negatives** reached **1,585 (80.5%)**, again showing near equivalence to the Grid Search–tuned configuration.

Most notably, the **ROC AUC Score was 0.782**, exactly matching that of the best Grid Search model. This indicates that the Randomized Search model performs equally well in terms of **probabilistic discrimination**, which measures the model's ability to correctly rank-order positive versus negative instances across all possible thresholds.

These findings clearly demonstrate that **Randomized Search can match the performance of Grid Search** in both threshold-based and ranking metrics, while **requiring only a fraction of the computational effort**.

Its stochastic sampling strategy proves highly effective at identifying near-optimal solutions in large and potentially sparse hyperparameter spaces.

Now, the question will be: **which one should we choose?**

Below there is a table that summarizes the relevant aspects:

Aspect	Grid Search	Randomized Search
Search Method	Exhaustive search over all parameter combinations	Randomly samples combinations from specified distributions
Coverage	Full coverage of parameter grid	Partial coverage (depends on number of iterations)
Time / Computational Cost	High (increases exponentially with more parameters)	Lower (you control how many combinations to try)
Best Use Case	Small parameter spaces	Large or complex parameter spaces
Scalability	Poor scalability	Good scalability
Chance of Finding Optimum	Higher if grid is well-designed	Good chance if enough iterations and smart distributions used
Parameter Types	Requires discrete value sets	Supports distributions (e.g., uniform, log-uniform)
Speed	Slow, especially with many parameters	Much faster for large spaces
Suitability for Random Forest	Less practical unless tuning few parameters	More practical and efficient

In addition, **Grid Search** is deterministic and structured — it tries every combination in a fixed, grid-like pattern. For that reason, we could have issues in the computation, but also in the selected grid of parameters.

Randomized Search is stochastic and irregular — it samples random combinations, think of it in terms of Stochastic Gradient Descent that samples random mini-batches during training instead of using the full dataset every time. However, because it samples randomly, it may also return suboptimal results—especially if the parameter distribution is poorly defined or if too few iterations are used.

Lastly, both of them reach a good result in different ways, so choosing one instead of another depends on the aim of the analysis.

If you're ever stuck deciding, a great practical **tip** is: start with Randomized Search to explore the space, then use Grid Search in a smaller refined range based on those results.

Hence, in general:

1. Choose *Grid Search* when:
    - The hyperparameter space is small and well-defined.
    - You want to perform a systematic, exhaustive evaluation of all possible combinations.
    - You have enough computational resources and time to explore every possibility.
    - You are refining an already narrow set of promising values (e.g., after Randomized Search).
    - The model is not too expensive to train (e.g., simpler models or small datasets).
2. Choose *Randomized Search* when:
    - The hyperparameter space is large or continuous.
    - You want to quickly explore a wide variety of parameter combinations.
    - You have limited time or computational resources.
    - You want to avoid rigid structure and are comfortable with a probabilistic approach to finding good configurations.

Chapter 5 – Reflection and Conclusion

5.1 Performance Evaluation

This study explored three modeling approaches for credit default prediction using a Random Forest classifier:

- A **baseline model** trained with default hyperparameters,
- A model optimized via **Grid Search**, evaluating 576 hyperparameter combinations (2,880 model fits with 5-fold cross-validation),
- A model tuned with **Randomized Search**, testing just 50 combinations (250 model fits in total).

Both hyperparameter-tuned models outperformed the baseline across all key metrics. The optimized models achieved an **accuracy of 71.48%** and a matching **ROC AUC of 0.782**, compared to roughly **70% accuracy and 0.765 AUC** for the baseline. These improvements confirm that hyperparameter tuning plays a critical role in enhancing predictive performance.

Model	Accuracy (%)	ROC AUC
Baseline Model	~70.00	~0.765
Grid Search	**71.48**	**0.782**
Randomized Search	**71.48**	**0.782**

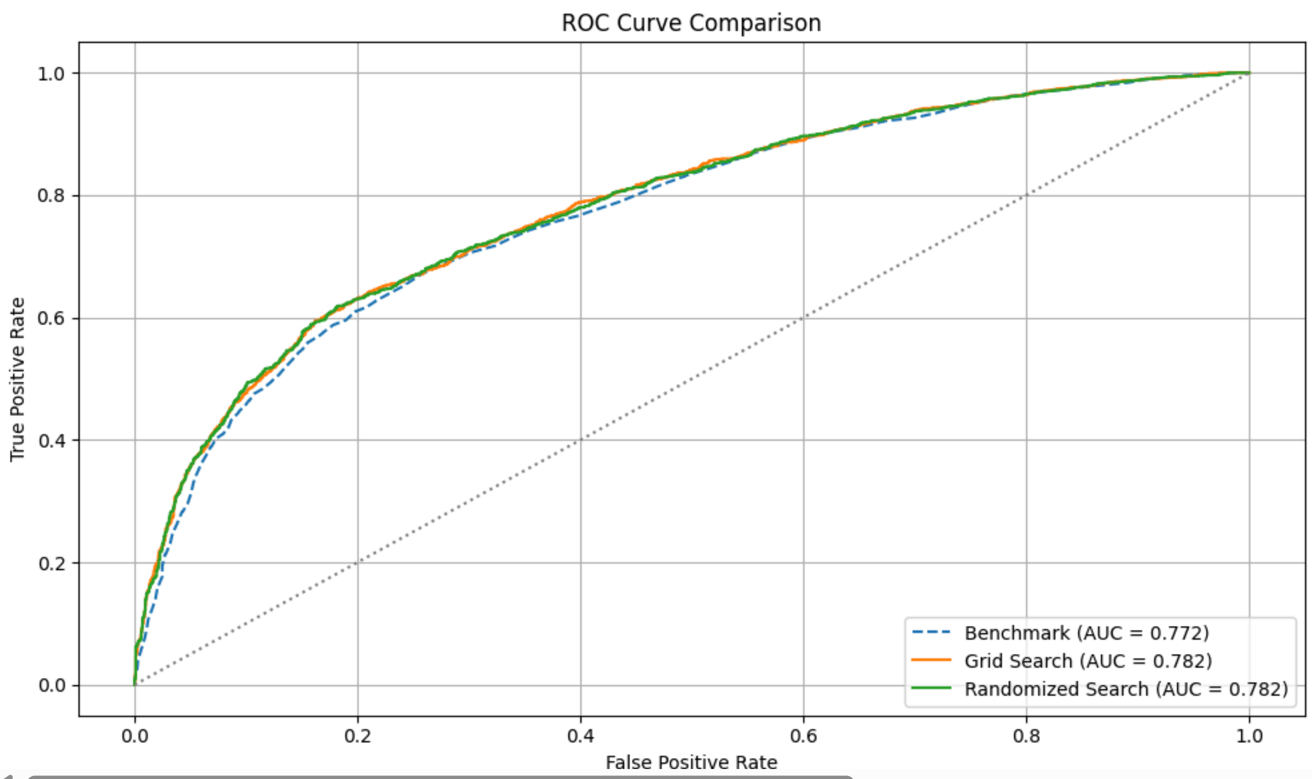
Classification metrics for the minority class (defaults) — including precision, recall, and F1-score — were also nearly identical between the tuned models, indicating consistent detection of high-risk clients.

The **ROC curve comparison** visually reinforced this conclusion: both Grid and Randomized Search models consistently outperformed the baseline across all thresholds, confirming superior discriminative power and probability ranking.

```
# ROC Curve Comparison Plot

plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, label=f"Benchmark (AUC = {auc_score:.3f})", linestyle="--")
plt.plot(fpr_gs, tpr_gs, label=f"Grid Search (AUC = {auc_gs:.3f})")
plt.plot(fpr_rs, tpr_rs, label=f"Randomized Search (AUC = {auc_rs:.3f})")
plt.plot([0, 1], [0, 1], linestyle=":", color="gray")

plt.title("ROC Curve Comparison")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend(loc="lower right")
plt.grid(True)
plt.tight_layout()
plt.show()
```



Notably, despite evaluating only **9% as many configurations**, Randomized Search achieved the **same optimal result** as Grid Search. This suggests that the search space was not overly complex or sensitive to fine-grained hyperparameter changes — a favorable situation where random sampling can efficiently identify high-performing regions.

While such parity in performance is not always expected, it highlights a key insight: **Randomized Search can deliver competitive results with a fraction of the computational cost**, making it particularly advantageous in time-sensitive or resource-constrained environments.

5.2 Generalization and Overfitting Analysis

To evaluate model generalization, we compared training and test accuracies across the benchmark model and both tuned models.

```
# Overfitting Check: Compare Training vs Test Accuracy for All Models

# Training and test predictions
train_preds_benchmark = rf_baseline.predict(X_train)
test_preds_benchmark = rf_baseline.predict(X_test)

train_preds_gs = best_rf.predict(X_train)
test_preds_gs = best_rf.predict(X_test)

train_preds_rs = best_rf_random.predict(X_train)
test_preds_rs = best_rf_random.predict(X_test)

# Print accuracies
print("=== Benchmark Model ===")
print("Training Accuracy:", round(accuracy_score(y_train, train_preds_benchmark), 3))
print("Test Accuracy    :", round(accuracy_score(y_test, test_preds_benchmark), 3))

print("\\n=== Grid Search ===")
print("Training Accuracy:", round(accuracy_score(y_train, train_preds_gs), 3))
print("Test Accuracy    :", round(accuracy_score(y_test, test_preds_gs), 3))

print("\\n=== Randomized Search ===")
print("Training Accuracy:", round(accuracy_score(y_train, train_preds_rs), 3))
print("Test Accuracy    :", round(accuracy_score(y_test, test_preds_rs), 3))
```

```
=== Benchmark Model ===
Training Accuracy: 0.999
Test Accuracy    : 0.705
\n=== Grid Search ===
Training Accuracy: 0.802
Test Accuracy    : 0.715
\n=== Randomized Search ===
Training Accuracy: 0.769
Test Accuracy    : 0.715
```

The results reveal a meaningful shift in behavior:

- The **baseline model**, trained with default hyperparameters, achieved near-perfect training accuracy (0.999) but only 70.5% test accuracy — a clear sign of **overfitting**, where the model memorizes patterns in the training set but fails to generalize to unseen data.
- The **Grid Search** and **Randomized Search** models introduced regularization through hyperparameter tuning. This reduced training accuracy (to 0.802 and 0.769, respectively) and simultaneously improved test accuracy (both reaching 71.5%), indicating better generalization.

Without hyperparameter tuning, Random Forests often overfit the training data — achieving nearly perfect accuracy by growing deep, highly specific trees. Despite this, the ensemble may still perform reasonably well on the test set due to the averaging effect across diverse, de-correlated trees, which mitigates the impact of individual overfitting.

However, once proper hyperparameter tuning is applied — such as limiting tree depth or adjusting the number of features considered at each split — overfitting is significantly reduced, and test performance improves. The closer alignment between training and test accuracy suggests that the model is no longer learning noise, but only extracting real, generalizable patterns from the data.

This strongly indicates that the current feature set has delivered all the predictive signal it can. The tuned models are operating near their **maximum generalization capacity**, and further performance gains would likely require the introduction of **more informative features** or external data sources.

5.3 Practical Implications of Hyperparameter Tuning

This comparison highlights a key trade-off between **exhaustiveness and efficiency** in hyperparameter tuning.

- **Grid Search** guarantees full coverage of the search space, but at a high computational cost. It required over **11× more model fits** than Randomized Search.
- **Randomized Search**, by sampling from distributions, achieved **identical performance** while testing fewer than 10% of the configurations. It is clearly more **computationally efficient**, especially in large or high-dimensional search spaces.

In real-world applications — where time, cost, and scalability are critical — **Randomized Search proves to be a practical and powerful choice**. It effectively balances performance and efficiency, particularly when only a subset of hyperparameters meaningfully influences model behavior.

In Conclusion

This project demonstrates that **Randomized Search can match the performance of Grid Search** in both threshold-based (accuracy) and ranking-based (ROC AUC) metrics, while demanding significantly fewer resources.

Moreover, the absence of overfitting — coupled with stable results across multiple evaluation methods — suggests that our models have reached **a robust and well-generalized solution** within the limits imposed by the feature set.

The path forward lies not in more tuning, but in **enriching the dataset**, improving **feature engineering**, or leveraging **external data**. Within the