Code explanaion.

*import* pandas *as* pd

*from* sklearn.model\_selection *import* train\_test\_split

*from* sklearn.ensemble *import* RandomForestClassifier

*from* sklearn.metrics *import* accuracy\_score, classification\_report

*from* sklearn.preprocessing *import* LabelEncoder

 **import pandas as pd**: Imports the Pandas library, which is used for data manipulation and analysis. Here, it reads your CSV file into a DataFrame (df).

 **from sklearn.model\_selection import train\_test\_split**: Imports a function to split your dataset into training and testing sets, ensuring your model can be evaluated on unseen data.

 **from sklearn.ensemble import RandomForestClassifier**: Imports the RandomForestClassifier, a machine learning algorithm that builds multiple decision trees and combines their outputs for robust predictions.

 **from sklearn.metrics import accuracy\_score, classification\_report**: Imports tools to evaluate your model’s performance. accuracy\_score measures the percentage of correct predictions, while classification\_report provides detailed metrics like precision, recall, and F1-score.

 **from sklearn.preprocessing import LabelEncoder**: Imports a utility to convert categorical labels (e.g., "Yes"/"No" or text outcomes) into numerical values, which machine learning models require.

file\_path = 'cases\_clean.csv'

df = pd.read\_csv(file\_path)

print("First few rows of the dataset:")

print(df.head())

 **file\_path = '/Users/khasanyusupkhuja/Downloads/cases\_clean.csv'**: Specifies the path to your dataset, a CSV file named cases\_clean.csv stored locally.

 **df = pd.read\_csv(file\_path)**: Loads the CSV file into a Pandas DataFrame (df), making the data ready for processing.

 **print("First few rows of the dataset:")**: Outputs a simple message to the console, labeling the upcoming data preview for clarity during presentation or debugging.

 **print(df.head())**: Displays the first five rows of your DataFrame (df) using the head() method from Pandas. This is a quick way to verify that the dataset loaded correctly and to get a sense of its structure, including column names and sample values.

columns\_to\_drop = ['Case ID', 'Date of Decision', 'Awarded Principal Amount',

'Awarded Penalty Amount', 'Awarded Fine', 'Awarded Unjustly Withheld Funds',

'Supporting Documents Presented', 'Category of a Claim']

df = df.drop(*columns*=columns\_to\_drop)

binary\_cols = ['Bankruptcy Proceedings Initiated', 'Bankruptcy Declared', 'Plaintiff Represented',

'Defendant Represented', 'Presence of Reconciliation Act', 'Presence of Contract',

'Presence of Invoice', 'Penalty Conditions', 'Partial Payment Made', 'Deadline Specified',

'Presence of Demand Letter']

*for* col *in* binary\_cols:

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

 **columns\_to\_drop = ['Case ID', 'Date of Decision', 'Awarded Principal Amount', 'Awarded Penalty Amount', 'Awarded Fine', 'Awarded Unjustly Withheld Funds', 'Supporting Documents Presented', 'Category of a Claim']**: Defines a list of columns to remove from the DataFrame. These columns (e.g., identifiers, dates, or monetary amounts) are likely irrelevant to predicting the target variable or could introduce noise.

 **df = df.drop(columns=columns\_to\_drop)**: Drops the specified columns from the DataFrame (df), simplifying the dataset to focus on predictive features.

 **binary\_cols = ['Bankruptcy Proceedings Initiated', 'Bankruptcy Declared', 'Plaintiff Represented', 'Defendant Represented', 'Presence of Reconciliation Act', 'Presence of Contract', 'Presence of Invoice', 'Penalty Conditions', 'Partial Payment Made', 'Deadline Specified', 'Presence of Demand Letter']**: Lists columns that contain binary (Yes/No) data, representing conditions or events in your cases.

 **for col in binary\_cols: df[col] = df[col].map({'Yes': 1, 'No': 0})**: Loops through each binary column and converts "Yes" to 1 and "No" to 0 using the map() function. This transforms categorical data into a numerical format suitable for machine learning algorithms.

X = df.drop(*columns*=['Outcome'])

y = df['Outcome']

le = LabelEncoder()

y = le.fit\_transform(y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, *test\_size*=0.2, *random\_state*=42, *stratify*=y)

* **X = df.drop(columns=['Outcome'])**: Creates the feature set (X) by removing the "Outcome" column from the DataFrame (df). This leaves all other columns as input features for the model.
* **y = df['Outcome']**: Defines the target variable (y) as the "Outcome" column, which the model will predict (e.g., case won/lost or another categorical outcome).
* **le = LabelEncoder()**: Initializes a LabelEncoder object from scikit-learn to convert the categorical "Outcome" values (e.g., "Win"/"Lose") into numerical labels (e.g., 0/1).
* **y = le.fit\_transform(y)**: Fits the encoder to the "Outcome" data and transforms it into numeric form, replacing the original text labels with integers.
* **X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)**: Splits the dataset into training and testing sets:
  + X\_train and y\_train: 80% of the data for training the model.
  + X\_test and y\_test: 20% for testing (controlled by test\_size=0.2).
  + random\_state=42: Ensures reproducibility of the split.
  + stratify=y: Maintains the proportion of each "Outcome" class in both training and testing sets, preventing imbalance.

*from* sklearn.model\_selection *import* GridSearchCV

param\_grid = {

'n\_estimators': [100, 200, 300],

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 4],

'max\_features': ['sqrt', 'log2']

}

grid\_search = GridSearchCV(RandomForestClassifier(*random\_state*=42),

param\_grid,

*cv*=5,

*n\_jobs*=-1,

*scoring*='accuracy')

grid\_search.fit(X\_train, y\_train)

print("Best parameters:", grid\_search.best\_params\_)

print("Best cross-validation score:", grid\_search.best\_score\_)

* **from sklearn.model\_selection import GridSearchCV**: Imports GridSearchCV, a tool for systematically testing combinations of hyperparameters to find the best-performing model.
* **param\_grid = {...}**: Defines a dictionary of hyperparameters to test:
  + 'n\_estimators': [100, 200, 300]: Number of trees in the forest (more trees can improve performance but increase computation time).
  + 'max\_depth': [None, 10, 20, 30]: Maximum depth of each tree (controls overfitting; None means unlimited depth).
  + 'min\_samples\_split': [2, 5, 10]: Minimum samples required to split a node (higher values reduce overfitting).
  + 'min\_samples\_leaf': [1, 2, 4]: Minimum samples per leaf node (increases robustness).
  + 'max\_features': ['sqrt', 'log2']: Number of features considered for each split (affects diversity of trees).
* **grid\_search = GridSearchCV(RandomForestClassifier(random\_state=42), param\_grid, cv=5, n\_jobs=-1, scoring='accuracy')**:
  + Initializes GridSearchCV with a RandomForestClassifier (random\_state=42 for reproducibility).
  + Uses param\_grid to define the search space.
  + cv=5: Performs 5-fold cross-validation, splitting the training data into 5 parts to validate performance.
  + n\_jobs=-1: Uses all available CPU cores for faster computation.
  + scoring='accuracy': Evaluates models based on accuracy.
* **grid\_search.fit(X\_train, y\_train)**: Fits the grid search to the training data, testing all combinations of hyperparameters to find the best model.
* **print("Best parameters:", grid\_search.best\_params\_)**: Outputs the optimal hyperparameter values found.
* **print("Best cross-validation score:", grid\_search.best\_score\_)**: Shows the highest accuracy achieved during cross-validation with the best parameters.

best\_rf = grid\_search.best\_estimator\_

y\_pred = best\_rf.predict(X\_test)

print("Accuracy:", accuracy\_score(y\_test, y\_pred))

print(classification\_report(y\_test, y\_pred, *target\_names*=le.classes\_))

 **best\_rf = grid\_search.best\_estimator\_**: Extracts the best-performing model (with the optimal hyperparameters) from the GridSearchCV results. This is the tuned RandomForestClassifier ready for evaluation.

 **y\_pred = best\_rf.predict(X\_test)**: Uses the best model (best\_rf) to predict outcomes on the test features (X\_test), generating predicted labels (y\_pred) for comparison with the actual test labels (y\_test).

 **print("Accuracy:", accuracy\_score(y\_test, y\_pred))**: Calculates and displays the accuracy of the model on the test set, which is the proportion of correct predictions (e.g., 0.85 means 85% accuracy).

 **print(classification\_report(y\_test, y\_pred, target\_names=le.classes\_))**: Prints a detailed report including:

* Precision: Proportion of positive predictions that were correct.
* Recall: Proportion of actual positives correctly identified.
* F1-score: Harmonic mean of precision and recall.
* Support: Number of samples per class.
* target\_names=le.classes\_: Maps numeric labels back to their original categorical names (e.g., "Win"/"Lose") for readability.

importances = best\_rf.feature\_importances\_

feature\_importance\_df = pd.DataFrame({'Feature': X.columns, 'Importance': importances})

print(feature\_importance\_df.sort\_values(*by*='Importance', *ascending*=False))

 **importances = best\_rf.feature\_importances\_**: Extracts the feature importance scores from the tuned RandomForestClassifier (best\_rf). These scores indicate how much each feature contributes to the model’s decision-making, based on how often and effectively it’s used across the trees.

 **feature\_importance\_df = pd.DataFrame({'Feature': X.columns, 'Importance': importances})**: Creates a Pandas DataFrame pairing each feature name (from X.columns) with its importance score. This organizes the data for easy analysis.

 **print(feature\_importance\_df.sort\_values(by='Importance', ascending=False))**: Sorts the DataFrame by importance in descending order and prints it. This shows the most influential features at the top (e.g., "Presence of Contract" might have a higher score than "Deadline Specified").

### What is a Random Forest?

Imagine you’re trying to decide if a case will win or lose, and you ask a group of friends for advice. Each friend looks at different clues—like whether there’s a contract or an invoice—and gives you their opinion. Then, you take a vote: the majority wins. That’s basically a **Random Forest**.

* **Technically**: It’s a machine learning model made of many decision trees (like flowcharts). Each tree looks at the data a bit differently, using random subsets of features and rows. They all “vote” on the outcome (e.g., "Win" or "Lose"), and the most popular answer wins.
* **Why it’s great**: One tree might be wrong, but hundreds together are usually smarter. It’s good at handling messy data and avoiding overfitting (memorizing the training data too well).

### What is GridSearchCV?

Now, imagine your Random Forest is a recipe. You can tweak things like how many trees to use or how deep each tree grows. But how do you find the best mix? That’s where **GridSearchCV** comes in—it’s like a super-organized taste test.

* **Technically**: It tries every combination of settings (hyperparameters) you give it—like 100 vs. 300 trees or a max depth of 10 vs. 20. CV means "cross-validation": it splits your training data into 5 chunks (in your case, cv=5), trains on 4, tests on 1, and repeats 5 times to get a reliable score. Then it picks the combo with the best accuracy.
* **Why it’s great**: Instead of guessing, it systematically finds the best version of your model. In your code, param\_grid lists the options, and grid\_search.best\_params\_ tells you the winner.

