1. Dataset Selection:

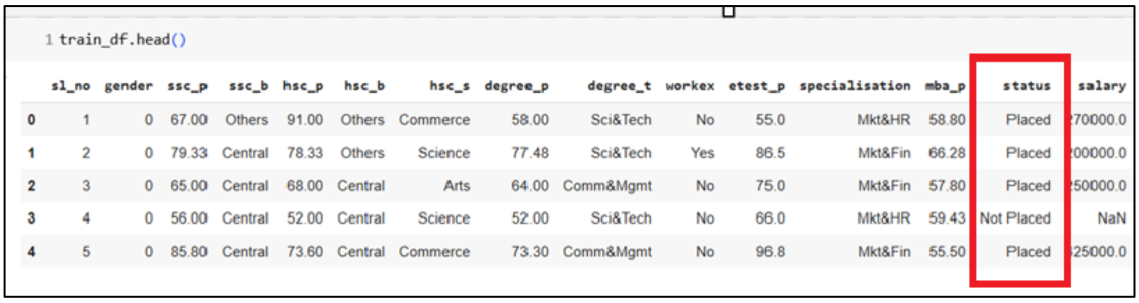
The dataset used for this project was sourced from Kaggle, a reliable and widely used platform for machine learning datasets. The main objective of this project is to build a predictive model for campus recruitment outcomes, helping to determine whether a student is likely to be placed based on their academic and personal background.

The dataset contains 215 rows and 15 columns, with a clearly defined categorical target variable: status, which indicates whether a student was successfully placed through a campus recruitment process. It includes a mix of numerical and categorical features such as gender, hsc_p, and ssc_p etc.,making it well-suited for a classification task. The data is of adequate size and complexity, allowing for meaningful model training and evaluation (see *Feature Understanding* section).

The dataset is split into two files: train.csv and test.csv. For this assignment, I used only train.csv. A complete exploratory data analysis (EDA) was performed on this file, followed by a 70/30 split into training and validation sets for modeling.

Steps done to check if the dataset (train.csv) has clear categorical target variable:

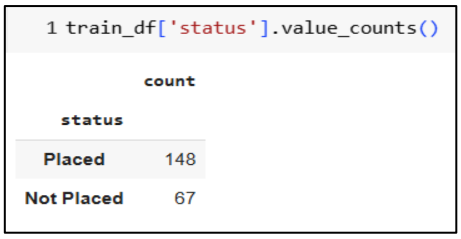
- Used train_df.head(). Based on the classification goal- Campus Recruitment Prediction, looks like the target column is ***status***. The status column has values like ***"Placed" and "Not Placed"***. This fits the requirement for a classification problem.



```
1 train_df.head()
```

	sl_no	gender	ssc_p	ssc_b	hsc_p	hsc_b	hsc_s	degree_p	degree_t	workex	etest_p	specialisation	mba_p	status	salary
0	1	0	67.00	Others	91.00	Others	Commerce	58.00	Sci&Tech	No	55.0	Mkt&HR	58.80	Placed	70000.0
1	2	0	79.33	Central	78.33	Others	Science	77.48	Sci&Tech	Yes	86.5	Mkt&Fin	66.28	Placed	00000.0
2	3	0	65.00	Central	68.00	Central	Arts	64.00	Comm&Mgmt	No	75.0	Mkt&Fin	57.80	Placed	50000.0
3	4	0	56.00	Central	52.00	Central	Science	52.00	Sci&Tech	No	66.0	Mkt&HR	59.43	Not Placed	NaN
4	5	0	85.80	Central	73.60	Central	Commerce	73.30	Comm&Mgmt	No	96.8	Mkt&Fin	55.50	Placed	25000.0

- Inspect the target distribution to confirm if the class balance is acceptable based in the rubrics. I used - train_df['status'].value_counts() :



```
1 train_df['status'].value_counts()
```

status	count
Placed	148
Not Placed	67

Observation: The target variable status has two classes: *Placed* and *Not Placed*. The class distribution is approximately **69% Placed** and **31% Not Placed**, indicating a *moderate class imbalance*.

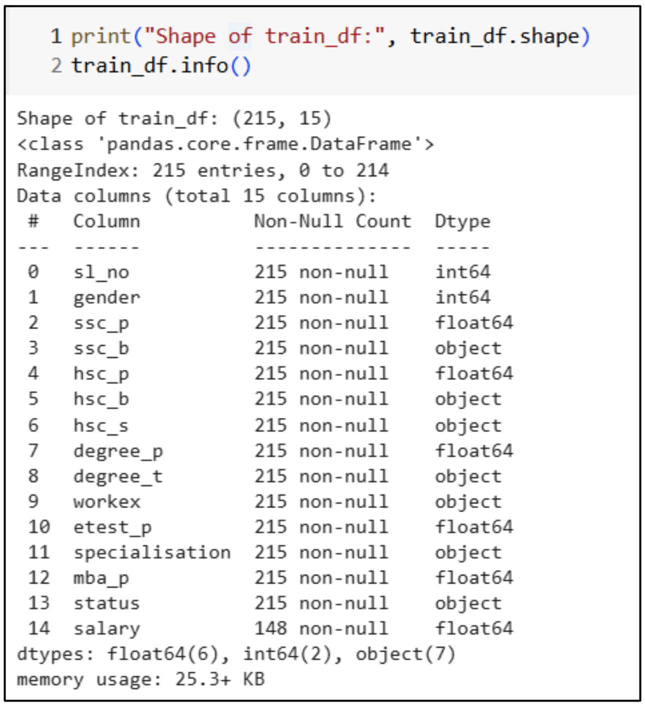
Implication: Models may slightly favour the majority class - “Placed”. Since more students in the dataset are placed, the model might end up predicting "Placed" more often, even when it's wrong. This can make the accuracy look high, but the model might be wrong more often on the "Not Placed" group.

Planned Approach: To address the class imbalance, I did the following:

- Used evaluation metrics beyond accuracy such as *precision, recall, and F1-score,* to better assess model performance.
- Applied *class balancing techniques* later in the preprocessing or modelling stages (e.g., *oversampling, under sampling, or class weights*) to help the model learn properly.

Feature Understanding :

- Checked the shape of the dataset using train_df.shape



```
print("Shape of train_df:", train_df.shape)
train_df.info()
```

```
Shape of train_df: (215, 15)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 215 entries, 0 to 214
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   sl_no           215 non-null    int64  
 1   gender          215 non-null    int64  
 2   ssc_p           215 non-null    float64
 3   ssc_b           215 non-null    object 
 4   hsc_p           215 non-null    float64
 5   hsc_b           215 non-null    object 
 6   hsc_s           215 non-null    object 
 7   degree_p        215 non-null    float64
 8   degree_t        215 non-null    object 
 9   workex          215 non-null    object 
 10  etest_p         215 non-null    float64
 11  specialisation  215 non-null    object 
 12  mba_p           215 non-null    float64
 13  status          215 non-null    object 
 14  salary          148 non-null    float64
dtypes: float64(6), int64(2), object(7)
memory usage: 25.3+ KB
```

Observations :

- Rows : 215 Columns : 15
- Only salary has missing values
- Target column (status) is non null and categorical
- Column types: mix of float64, int64, and object
- 14 features with a mix of numerical and categorical
- sl_no is just an index – this can be dropped from the modelling pipeline
- Multiple categorical columns (object dtype) (which means need encoding later)

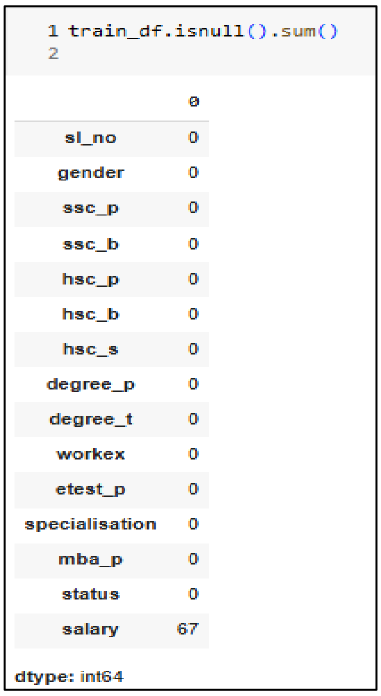
The dataset includes 14 features that describe students’ academic performance and background. Only three fields — sl_no, gender, and salary — were described on the Kaggle site. To better understand the remaining features, I researched and interpreted their probable meanings based on their names and typical usage in educational datasets.

Column Name	Likely Description
sl_no	an anonymous id unique to a given employee (from Kaggle)
Gender	The sex of employee (from Kaggle)
ssc_p	Secondary Education % score or grade (numeric)
ssc_b	Board of Education (categorical: Central/Other)
hsc_p	Higher Secondary % score or grade
hsc_b	Higher Secondary Board (categorical)
hsc_s	Stream (Science, Commerce, Art)
degree_p	Degree % (The student’s overall percentage score or grade average in their undergraduate degree)

degree_t	Type of degree (Sci & Tech, Comm & Mgmt, etc.)
workex	Work experience (Yes/No)
etest_p	Employability test score
specialisation	MBA specialization (Mkt&Fin or Mkt&HR)
mba_p	MBA (overall percentage score or GPA in their MBA program (Master of Business Administration))
status	Target variable: Placed / Not Placed
salary	may be correlated with placement;

2. Data Preprocessing:

2.1 Check for Missing Values



```
train_df.isnull().sum()

                0
         sl_no  0
        gender  0
         ssc_p  0
         ssc_b  0
         hsc_p  0
         hsc_b  0
         hsc_s  0
      degree_p  0
      degree_t  0
        workex  0
       etest_p  0
specialisation  0
         mba_p  0
        status  0
        salary 67

dtype: int64
```

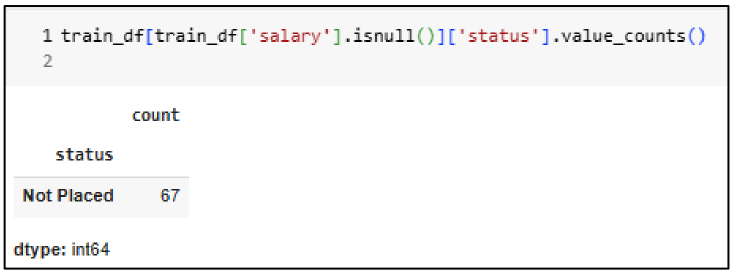
Observations:

All columns except **salary** have **0 missing values**
salary has ***67 missing values*** out of ***215***

Note *:*

The salary column is the ***only one with missing data****, and it's likely missing for students who were* ***not placed*** *(because no job = no salary)*

To check if all missing salary rows are from "Not Placed" students, I ran the following code :



```
train_df[train_df['salary'].isnull()]['status'].value_counts()

            count
    status
Not Placed     67

dtype: int64
```

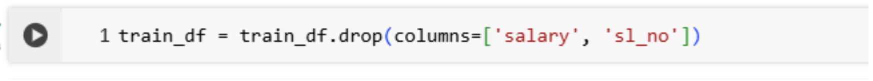
Interpretation :

- All 67 missing salary values are from students who were Not Placed
- This means that salary is only available for placed students, which makes total sense in this case.

Missing Values Handling Summary :

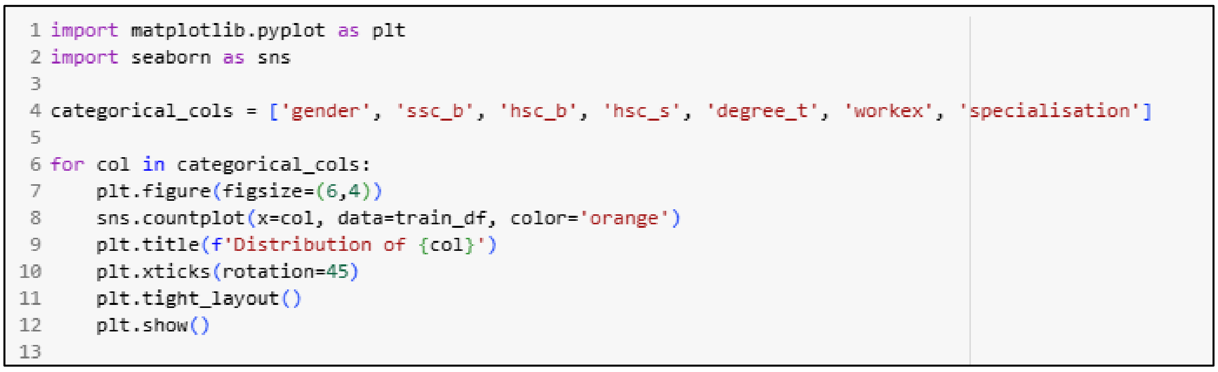
All columns were complete except for salary, which had *67 missing entries*. These were exclusively associated with students who were "not placed", which was expected. Since salary is only available for placed students, it would not be appropriate to use this feature in predictive modelling as this would "leak future information" into the model, also known as data leakage.

Therefore, the salary column was dropped to avoid introducing data leakage. No other missing values were found in the dataset.

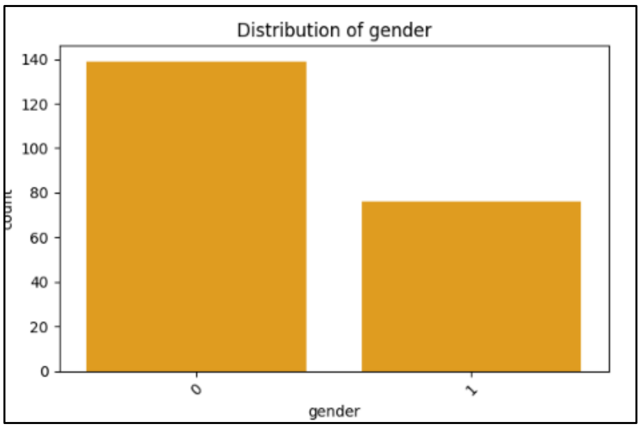


```python
train_df = train_df.drop(columns=['salary', 'sl_no'])
```

2.2 Categorical Features' Distribution. This is to check the distribution of the categorical features to find imbalances, rare classes, typos, encoding needs. I used bar plot to visualize the distribution.

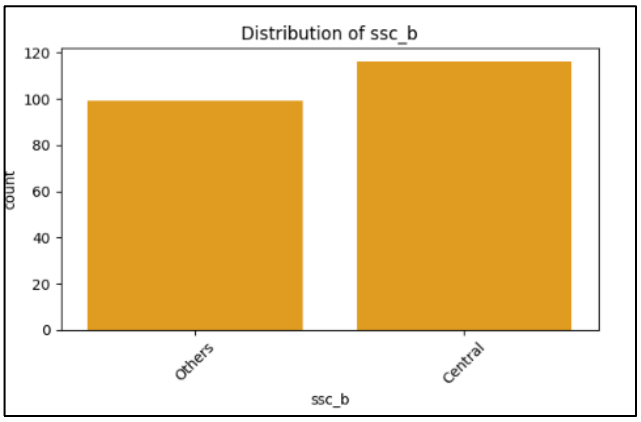


```python
import matplotlib.pyplot as plt
import seaborn as sns

categorical_cols = ['gender', 'ssc_b', 'hsc_b', 'hsc_s', 'degree_t', 'workex', 'specialisation']

for col in categorical_cols:
    plt.figure(figsize=(6,4))
    sns.countplot(x=col, data=train_df, color='orange')
    plt.title(f'Distribution of {col}')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

```

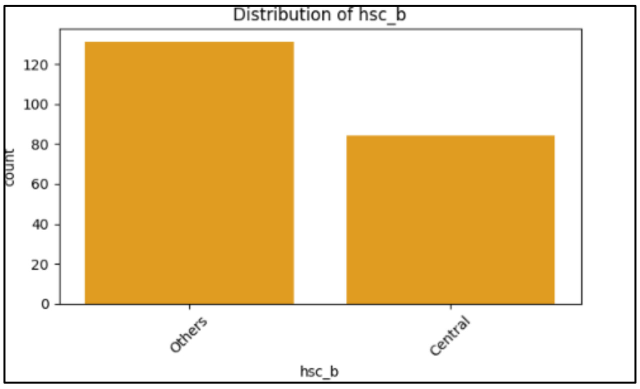


Gender distribution is *slightly imbalanced:*
139 (0) vs. 76 (1)

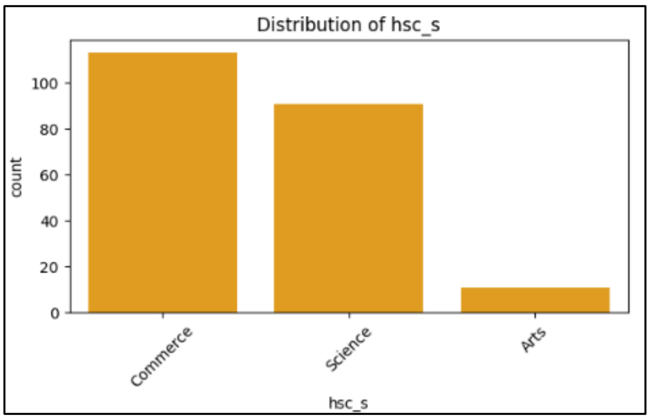
The ***gender*** variable is already encoded as 0 and 1; based on common practices, 0 is likely Male and 1 Female, but this assumption was not confirmed by dataset documentation.



ssc_b is reasonably balanced:
Central (116) vs. Others (99)

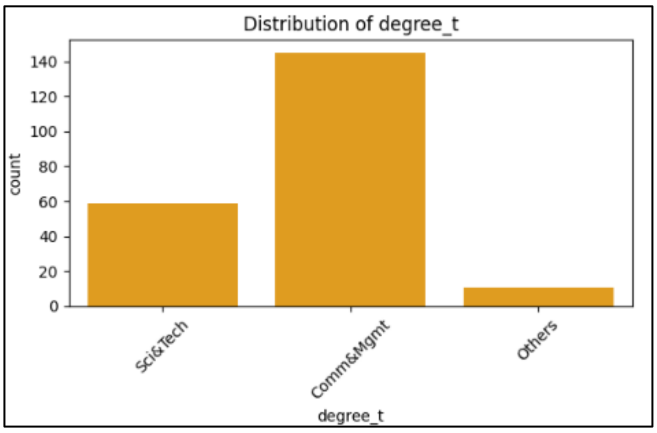


Slight imbalance:
Others (131) > Central (84)



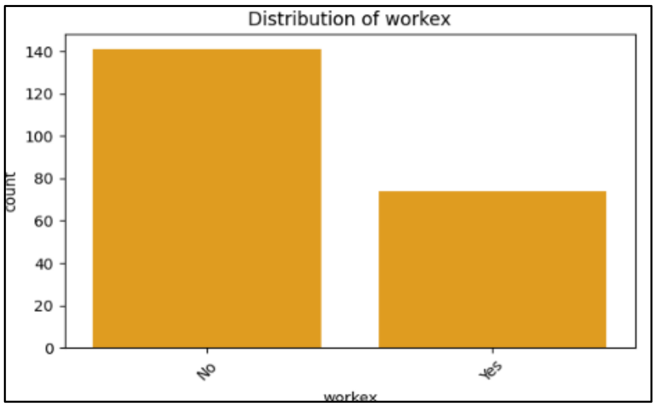
Noticeable imbalance:

Commerce (113)
Science (91)
Arts (11) ***Arts is underrepresented***

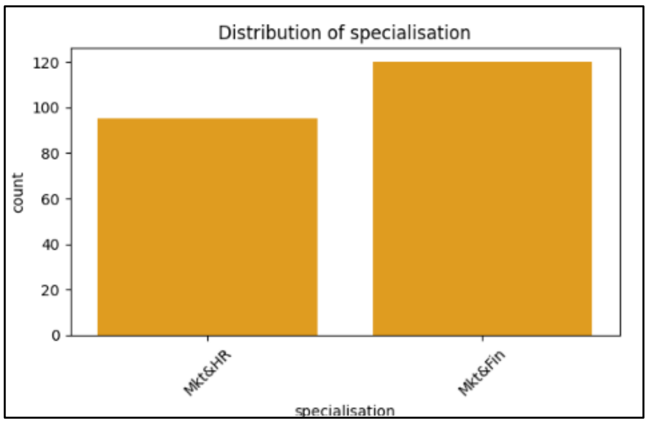


Heavily imbalanced:

Comm&Mgmt (145)
Sci&Tech (59)
Others (11) — may consider merging or special encoding for "Others"



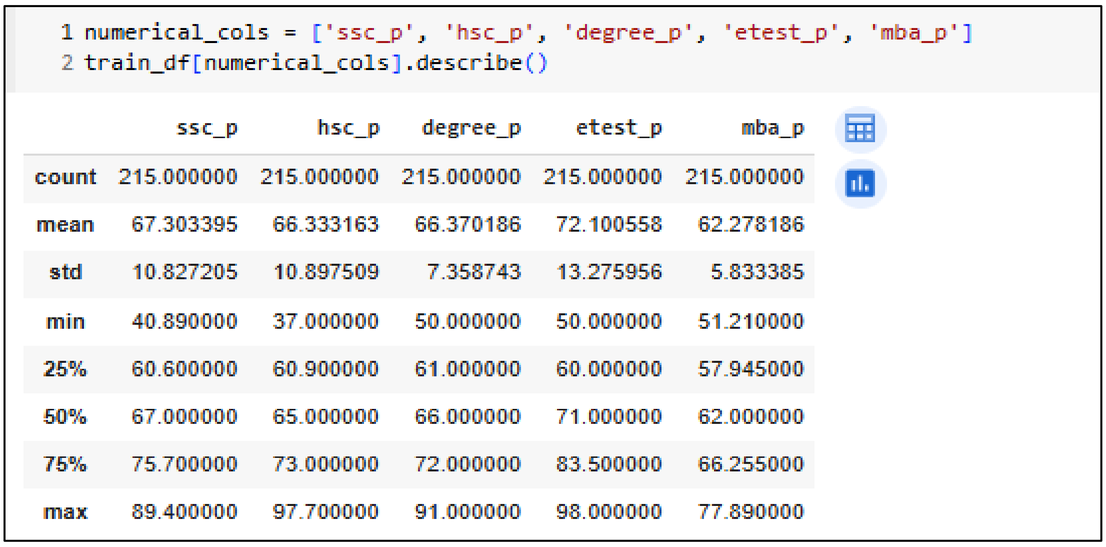
Slight imbalance: No (141), Yes (74)



Acceptable: Mkt&Fin (120), Mkt&HR (95)

2.3 Numerical Features' Distribution. This is to check if the dataset has outliers, to spot skewed distributions and to decide whether scaling or transformation is needed.

Note : I did not include salary as it has missing values



```
numerical_cols = ['ssc_p', 'hsc_p', 'degree_p', 'etest_p', 'mba_p']
train_df[numerical_cols].describe()
```

```
            ssc_p       hsc_p    degree_p     etest_p       mba_p
count  215.000000  215.000000  215.000000  215.000000  215.000000
mean    67.303395   66.333163   66.370186   72.100558   62.278186
std     10.827205   10.897509    7.358743   13.275956    5.833385
min     40.890000   37.000000   50.000000   50.000000   51.210000
25%     60.600000   60.900000   61.000000   60.000000   57.945000
50%     67.000000   65.000000   66.000000   71.000000   62.000000
75%     75.700000   73.000000   72.000000   83.500000   66.255000
max     89.400000   97.700000   91.000000   98.000000   77.890000
```

Observations :

Feature	***Range of Values (min to max)***	***Note***
ssc_p	40.9 to 89.4	Most students scored between 60 and 75. The scores are spread out well.
hsc_p	37.0 to 97.7	Most students scored between 60 and 73. The scores are also spread out well.
degree_p	50.0 to 91.0	These scores are a bit more concentrated around the 60–70 range, meaning most students got similar marks.
etest_p	50.0 to 98.0	Scores vary quite a bit.
mba_p	51.2 to 77.9	These scores are more tightly packed. Everyone scored close to each other.

2.4 Encoding Categorical Features

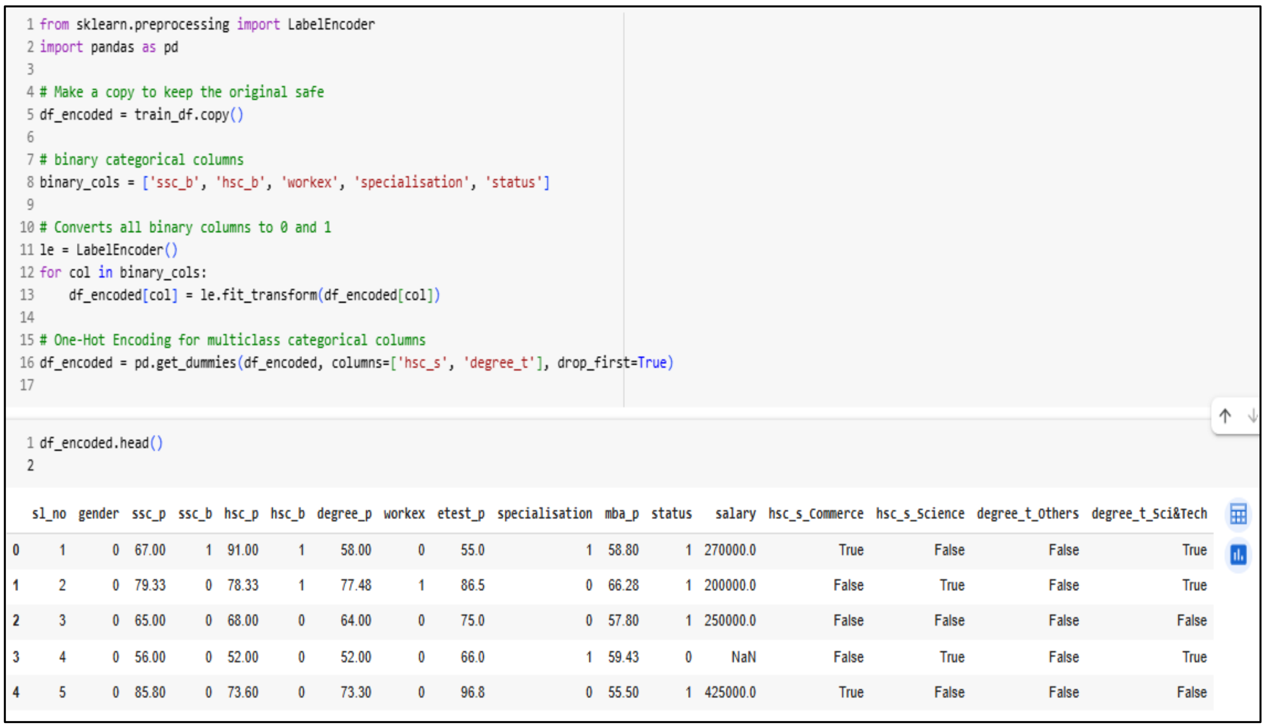
Convert all categorical (non-numeric) columns into a format that machine learning models can understand.

Encoding Method:

Feature	Type	Encoding Method
gender	Binary	Already in numeric
ssc_b	2 categories (e.g., Central/Others)	Label Encoding
hsc_b	2 categories	Label Encoding
hsc_s	3 categories (Commerce/Science/Arts)	*One-Hot Encoding*
degree_t	3 categories	*One-Hot Encoding*
workex	Yes/No	Label Encoding
specialisation	2 categories	Label Encoding
status	Target	Label Encoding (Placed = 1, Not Placed = 0)

Note:

I used label encoding for categorical features that has two categories – binary (e.g "Yes" and "No" or "Central" and "Others"). For the rest of the features with more than 2 categories (Multiclass), I used one hot encoding to avoid introducing false order.



```python
from sklearn.preprocessing import LabelEncoder
import pandas as pd

# Make a copy to keep the original safe
df_encoded = train_df.copy()

# binary categorical columns
binary_cols = ['ssc_b', 'hsc_b', 'workex', 'specialisation', 'status']

# Converts all binary columns to 0 and 1
le = LabelEncoder()
for col in binary_cols:
    df_encoded[col] = le.fit_transform(df_encoded[col])

# One-Hot Encoding for multiclass categorical columns
df_encoded = pd.get_dummies(df_encoded, columns=['hsc_s', 'degree_t'], drop_first=True)

```

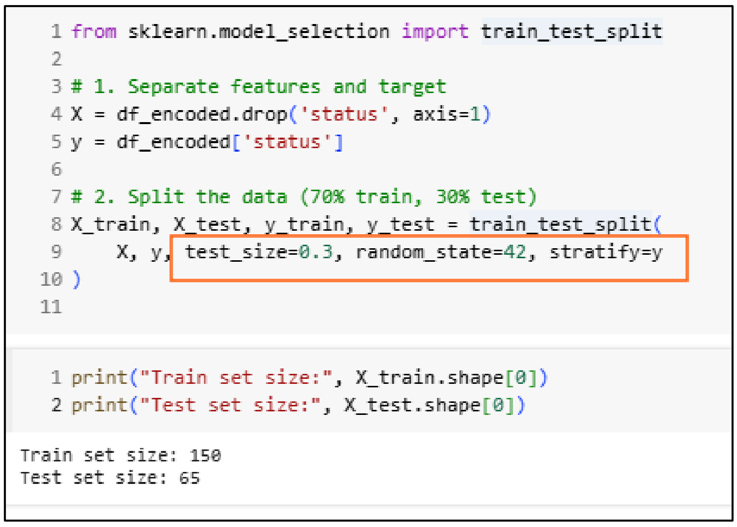
```python
df_encoded.head()

```

```
   sl_no  gender  ssc_p  ssc_b  hsc_p  hsc_b  degree_p  workex  etest_p  specialisation  mba_p  status    salary  hsc_s_Commerce  hsc_s_Science  degree_t_Others  degree_t_Sci&Tech
0      1       0  67.00      1  91.00      1     58.00       0     55.0               1  58.80       1  270000.0            True          False            False               True
1      2       0  79.33      0  78.33      1     77.48       1     86.5               0  66.28       1  200000.0           False           True            False               True
2      3       0  65.00      0  68.00      0     64.00       0     75.0               0  57.80       1  250000.0           False          False            False              False
3      4       0  56.00      0  52.00      0     52.00       0     66.0               1  59.43       0       NaN           False           True            False               True
4      5       0  85.80      0  73.60      0     73.30       0     96.8               0  55.50       1  425000.0            True          False            False              False
```

The above image shows conversion of categorical features using LabelEncoder for binary categories pd.get_dummies() for multiclass categories (hsc_s, degree_t).

The final dataset consists entirely of numeric values, ready for model training.

2.5 Train / Test Split. Split the dataset (train.csv) (70%) for training the model and test set (30%) for evaluating model performance. We will separate the features (X) from the target (y) and use train_test_split from scikit-learn.



```python
from sklearn.model_selection import train_test_split

# 1. Separate features and target
X = df_encoded.drop('status', axis=1)
y = df_encoded['status']

# 2. Split the data (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

```

```python
print("Train set size:", X_train.shape[0])
print("Test set size:", X_test.shape[0])
```

```
Train set size: 150
Test set size: 65
```

test_size=0.3 (gives a 70/30 split)

random_state=42 (ensures reproducibility**)**

stratify=y (keeps the same ratio of "Placed" and "Not Placed" in both sets (very important for imbalanced data))

***Note**: During the data selection phase, I observed a moderate class imbalance in the target variable: approximately 69% of students were labelled as "Placed" and 31% as "Not Placed". To makes sure both sets are representative of the real data. I used stratify=y during the data split. This ensures fair training, prevents model bias toward the majority class, and allows for more reliable evaluation results.*

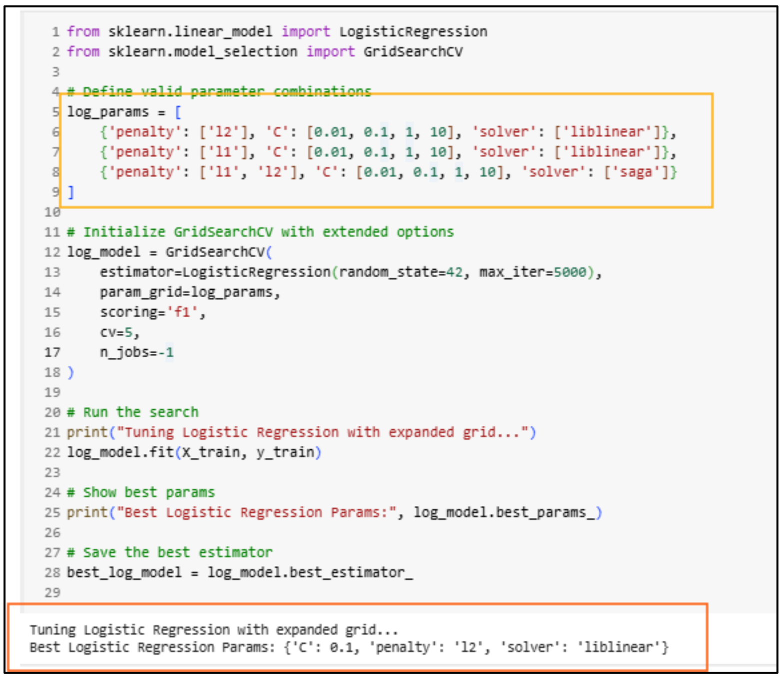
3. Model Selection

Model	Type	Why it's Suitable
Logistic Regression	Linear	Fast, simplicity and interpretability and handles binary classification well.
Random Forest Classifier	Ensemble Tree	Handles categorical & numerical features, good with imbalanced dataset which is what I have in my dataset.
Support Vector Machine (SVM)	Margin-based	Known for its ability to find optimal class boundaries, making it effective in smaller datasets with moderate imbalance

The above models provide a good mix of interpretability, performance and flexibility.

MODEL TUNING AND EVALUATION

Model Tuning for Logistic Regression:

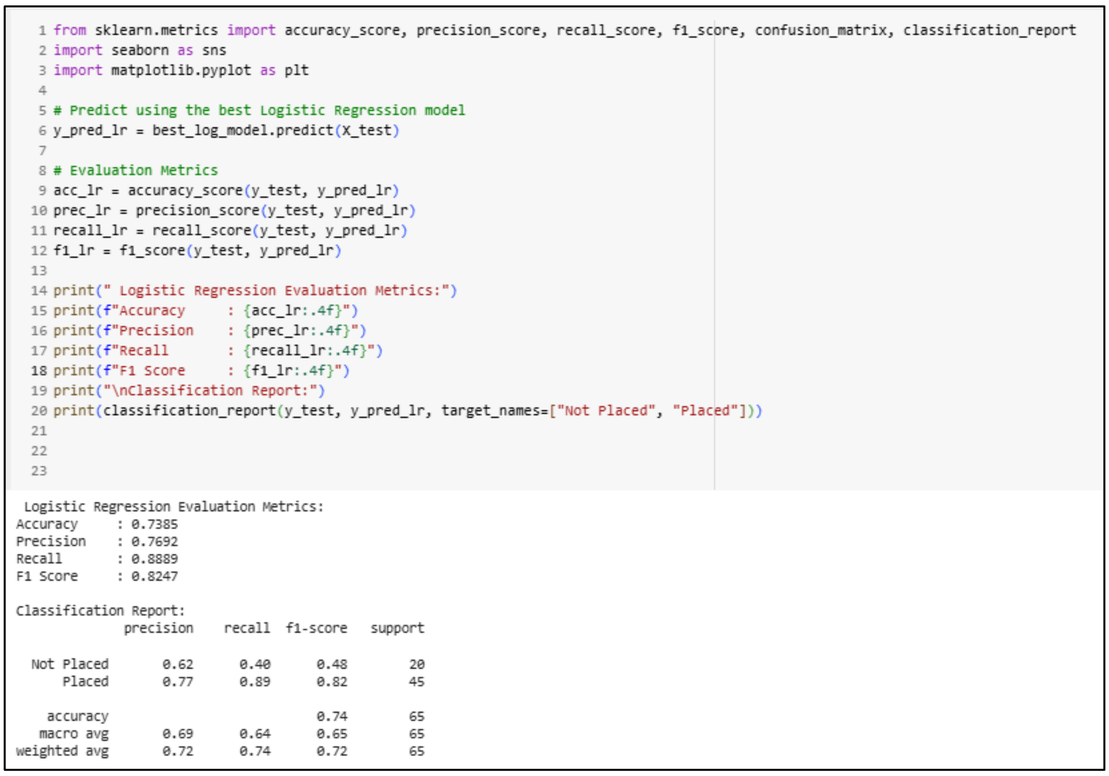


```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

# Define valid parameter combinations
log_params = [
    {'penalty': ['l2'], 'C': [0.01, 0.1, 1, 10], 'solver': ['liblinear']},
    {'penalty': ['l1'], 'C': [0.01, 0.1, 1, 10], 'solver': ['liblinear']},
    {'penalty': ['l1', 'l2'], 'C': [0.01, 0.1, 1, 10], 'solver': ['saga']}
]

# Initialize GridSearchCV with extended options
log_model = GridSearchCV(
    estimator=LogisticRegression(random_state=42, max_iter=5000),
    param_grid=log_params,
    scoring='f1',
    cv=5,
    n_jobs=-1
)

# Run the search
print("Tuning Logistic Regression with expanded grid...")
log_model.fit(X_train, y_train)

# Show best params
print("Best Logistic Regression Params:", log_model.best_params_)

# Save the best estimator
best_log_model = log_model.best_estimator_

```

```
Tuning Logistic Regression with expanded grid...
Best Logistic Regression Params: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
```

- To improve the performance of the Logistic Regression model, I used GridSearchCV to test different combinations hyperparameters.
    - I tried different values for regularization strength (C) - 'C': [0.01, 0.1, 1, 10]. Stronger regularization like 0.1 helps prevent overfitting by shrinking the model's weight. While weaker regularization - large C like 10 focuses more on fitting the training data exactly, which might cause overfitting.
    - I explored both L1 and L2 regularization during model tuning. L1 can help by removing unimportant features, while L2 reduces the impact of all features to prevent overfitting. The best performance was achieved using L2, suggesting that keeping all features (but shrinking their influence) worked better for this dataset.
    - Although there are several solvers are available in scikit-learn (such as lbfgs, newton-cg, and sag), I focused on *liblinear and saga* because they support both L1 and L2 regularization. This ensured both compatibility and meaningful hyperparameter exploration.
    - I used *F1-score for tuning to account for class imbalance* and ensure the model performs well on both "Placed" and "Not Placed" classes.
    - I used *random_state=42 for reproducibility and max_iter=5000 to ensure convergence,* especially when using the saga solver. Once the best parameters were identified, the final model was retrained on the full training set using those settings.

Model Evaluation for Logistic Regression (accuracy, precision, recall, F1-score)



```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt

# Predict using the best Logistic Regression model
y_pred_lr = best_log_model.predict(X_test)

# Evaluation Metrics
acc_lr = accuracy_score(y_test, y_pred_lr)
prec_lr = precision_score(y_test, y_pred_lr)
recall_lr = recall_score(y_test, y_pred_lr)
f1_lr = f1_score(y_test, y_pred_lr)

print(" Logistic Regression Evaluation Metrics:")
print(f"Accuracy     : {acc_lr:.4f}")
print(f"Precision    : {prec_lr:.4f}")
print(f"Recall       : {recall_lr:.4f}")
print(f"F1 Score     : {f1_lr:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred_lr, target_names=["Not Placed", "Placed"]))
```

```
 Logistic Regression Evaluation Metrics:
Accuracy     : 0.7385
Precision    : 0.7692
Recall       : 0.8889
F1 Score     : 0.8247

Classification Report:
              precision    recall  f1-score   support

  Not Placed       0.62      0.40      0.48        20
      Placed       0.77      0.89      0.82        45

    accuracy                           0.74        65
   macro avg       0.69      0.64      0.65        65
weighted avg       0.72      0.74      0.72        65
```

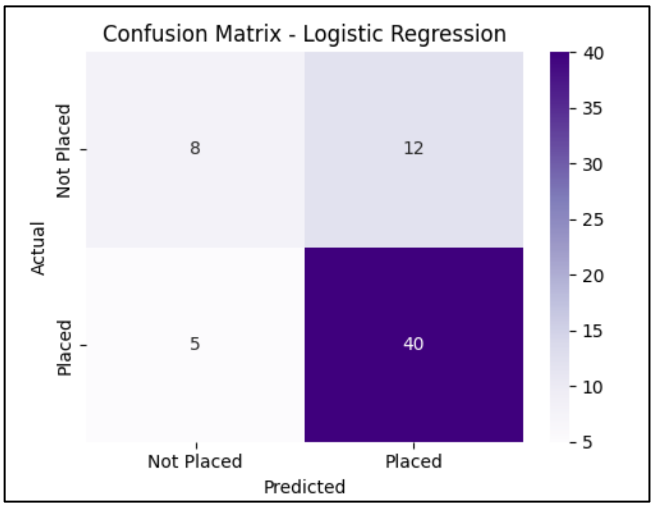
Interpretation:

- High recall (0.89) means the model is very good at catching students who got placed *(true positives)*
- Precision (0.77)— about 77% of those predicted as "Placed" were correct.
- Accuracy (~74%) reflects overall correctness but is slightly affected by class imbalance.
- Class-level recall for 'Not Placed' is low (0.40) → the model misses many "Not Placed" students, likely due to the class imbalance

Summary:

The Logistic Regression model was evaluated on the test set using multiple metrics. ***It achieved an accuracy of 73.85%, with a precision of 76.92% and a recall of 88.89%.*** The high recall indicates the model is very effective at identifying students who were placed. However, its performance on the "Not Placed" class was lower, with a ***recall of only 40%,*** *suggesting that the class imbalance may have impacted the model's ability to correctly predict unplaced students.* **The overall F1 score of 82.47% reflects a strong balance between precision and recall for the majority class.**

Confusion Matrix:



Interpretations:

- The model correctly predicted 40 placed students (TP)
- It missed *5 placed students (FN)*
- It correctly *flagged 8 students as not placed (TN)*
- But it *mistakenly predicted 12 not placed students as placed (FP)*

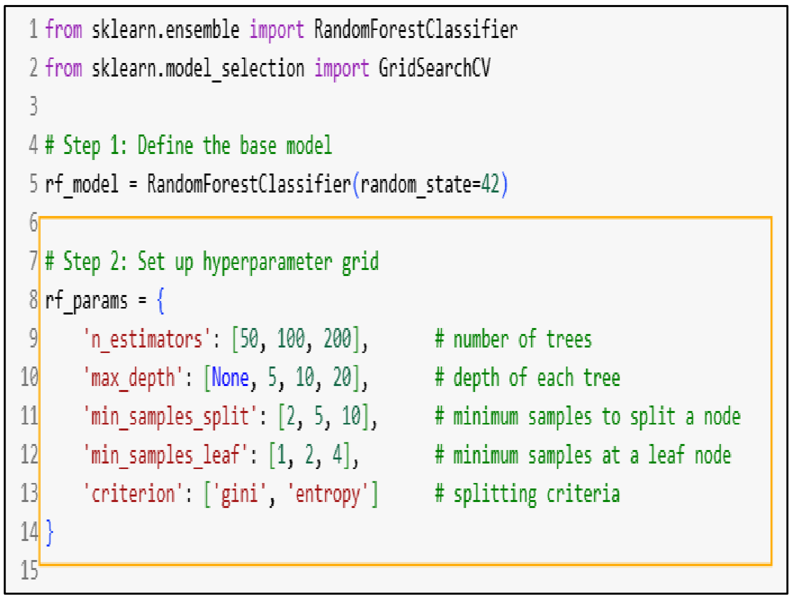
Summary:

The confusion matrix shows that the model *correctly identified 40 students who were placed and 8 who were not placed.* However, *it misclassified 12 students who were not placed as placed, and 5 placed students as not placed.* This reflects the model's tendency to favour the "Placed" class, likely due to the class imbalance. *While the model performs well overall, its ability to accurately identify students who were not placed is more limited.*

Overall, Logistic Regression performed well in terms of general placement prediction but struggled slightly with the minority class, which is expected in imbalanced datasets. The model offers good interpretability and a solid baseline for comparison against more complex classifiers.

Model Tuning for Random Forrest

Hyperparameter Grid:



```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

# Step 1: Define the base model
rf_model = RandomForestClassifier(random_state=42)

# Step 2: Set up hyperparameter grid
rf_params = {
    'n_estimators': [50, 100, 200],        # number of trees
    'max_depth': [None, 5, 10, 20],        # depth of each tree
    'min_samples_split': [2, 5, 10],       # minimum samples to split a node
    'min_samples_leaf': [1, 2, 4],         # minimum samples at a leaf node
    'criterion': ['gini', 'entropy']       # splitting criteria
}

```

n_estimators = Number of trees in the forest (more = better accuracy, slower training)

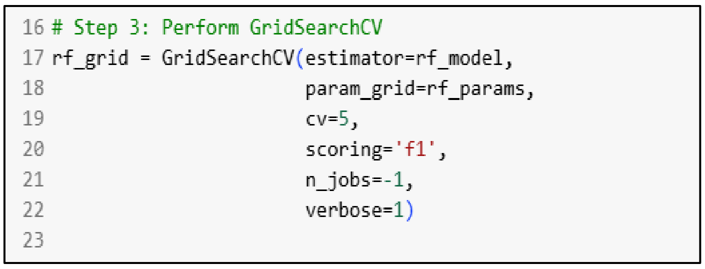
max_depth = How deep each tree can grow

min_samples_split = Minimum # of samples needed to split a node

min_samples_leaf = Minimum # of samples required at a leaf

criterion = How the model splits nodes (gini or entropy)

Perform GridSearchCV:

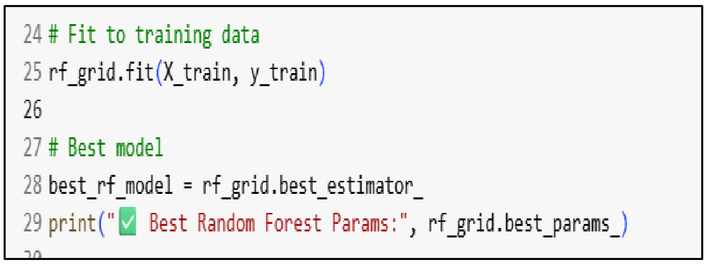


```python
# Step 3: Perform GridSearchCV
rf_grid = GridSearchCV(estimator=rf_model,
                       param_grid=rf_params,
                       cv=5,
                       scoring='f1',
                       n_jobs=-1,
                       verbose=1)

```

Summary :

- Used GridSearchCV to try all the combinations of parameters with 5-fold cross-validation.
- Used F1-score for scoring, again because of class imbalance.
- n_jobs=-1 makes it run faster by using all CPU cores.
- verbose=1 gives you feedback during training.

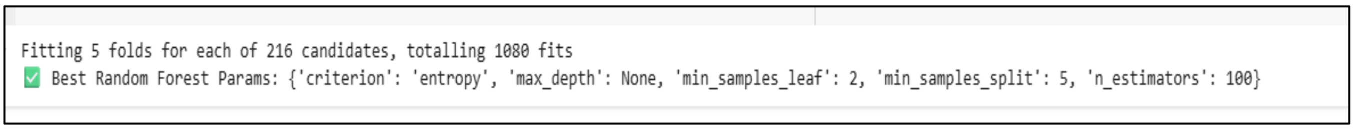
Fit the Grid Search to Training Data



```python
# Fit to training data
rf_grid.fit(X_train, y_train)

# Best model
best_rf_model = rf_grid.best_estimator_
print("✅ Best Random Forest Params:", rf_grid.best_params_)
```

- Trains many versions of the model using your training set.
- Picks the best one based on F1-score.
- Stores that in best_rf_model so you can use it later.

Output :



```
Fitting 5 folds for each of 216 candidates, totalling 1080 fits
✅ Best Random Forest Params: {'criterion': 'entropy', 'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 5, 'n_estimators': 100}
```

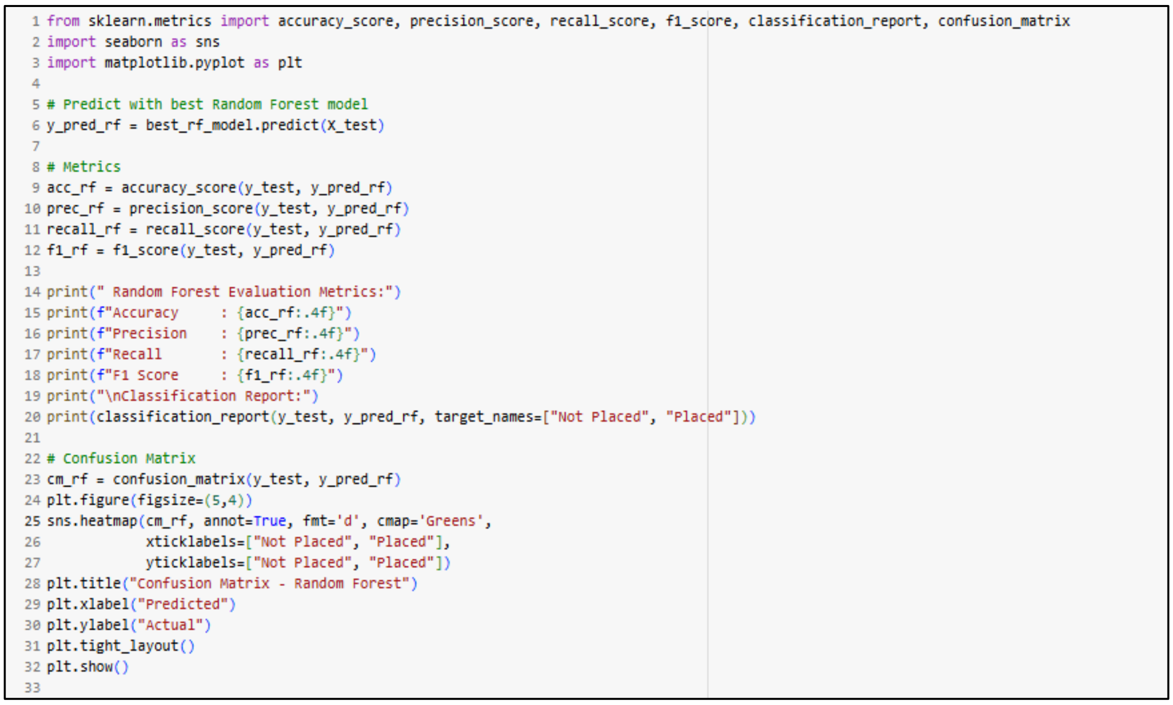
Interpretation :

Parameter	Value	Meaning
criterion	entropy	Model *uses information gain* to split nodes (instead of Gini)
max_depth	None	Trees were allowed to grow *fully* — no max limit
min_samples_leaf	2	Each leaf must have at *least 2 samples,* helps avoid overfitting
min_samples_split	5	A node must have *at least 5 samples* to be split
n_estimators	100	The forest has 100 trees, a solid balance of speed and accuracy

Summary :

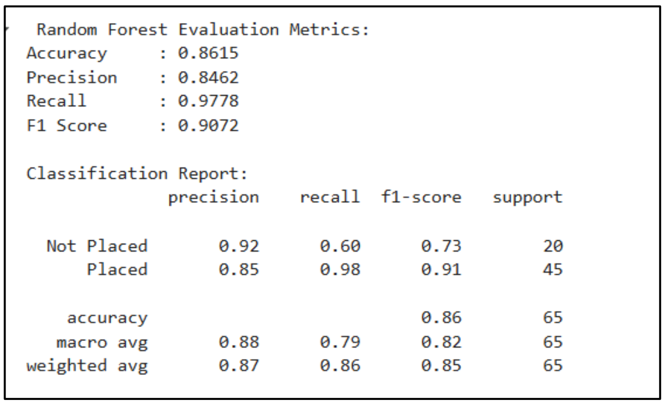
Random Forest was tuned using GridSearchCV over a wide range of parameters. The best configuration found was: n_estimators = 100, max_depth = None, min_samples_split = 5, min_samples_leaf = 2, and criterion = 'entropy'. These settings allowed the model to grow complex trees using the information gain criterion while controlling for overfitting with minimum leaf and split size constraints.

Model Evaluation for Random Forest (accuracy, precision, recall, F1-score)



```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Predict with best Random Forest model
y_pred_rf = best_rf_model.predict(X_test)

# Metrics
acc_rf = accuracy_score(y_test, y_pred_rf)
prec_rf = precision_score(y_test, y_pred_rf)
recall_rf = recall_score(y_test, y_pred_rf)
f1_rf = f1_score(y_test, y_pred_rf)

print(" Random Forest Evaluation Metrics:")
print(f"Accuracy     : {acc_rf:.4f}")
print(f"Precision    : {prec_rf:.4f}")
print(f"Recall       : {recall_rf:.4f}")
print(f"F1 Score     : {f1_rf:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred_rf, target_names=["Not Placed", "Placed"]))

# Confusion Matrix
cm_rf = confusion_matrix(y_test, y_pred_rf)
plt.figure(figsize=(5,4))
sns.heatmap(cm_rf, annot=True, fmt='d', cmap='Greens',
            xticklabels=["Not Placed", "Placed"],
            yticklabels=["Not Placed", "Placed"])
plt.title("Confusion Matrix - Random Forest")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.tight_layout()
plt.show()

```

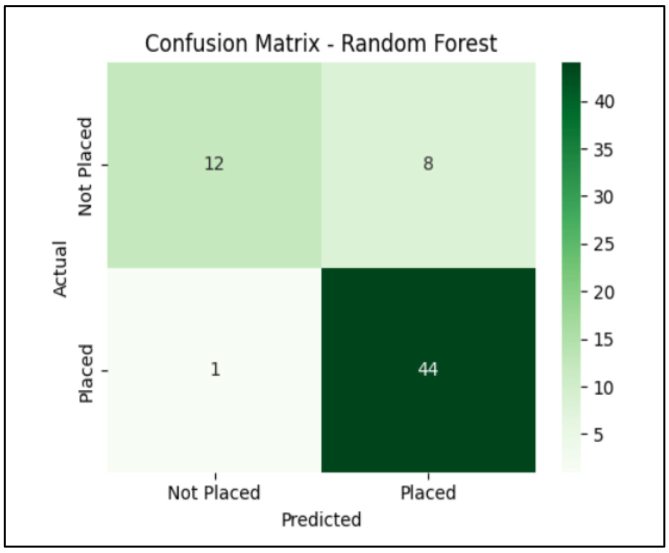
Output :



```
 Random Forest Evaluation Metrics:
Accuracy     : 0.8615
Precision    : 0.8462
Recall       : 0.9778
F1 Score     : 0.9072

Classification Report:
              precision    recall  f1-score   support

  Not Placed       0.92      0.60      0.73        20
      Placed       0.85      0.98      0.91        45

    accuracy                           0.86        65
   macro avg       0.88      0.79      0.82        65
weighted avg       0.87      0.86      0.85        65
```

Interpretation:

- *Very high recall (97.78%):* the model *is extremely good at identifying "Placed" students*.
- Precision (84.62%) is strong — most students predicted as "Placed" actually were.
- Class-level recall for "Not Placed" is 60% — better than Logistic Regression’s 40%
- Overall *F1-score is 90.72%,* the best so far.
- *Accuracy improved to 86% from Logistic Regression's 74%.*



Interpretations:

44 students who were placed were correctly predicted (True Positives).

12 students who were not placed were also *correctly* predicted (True Negatives).

8 students who were *not placed were incorrectly* predicted as placed (False Positives).

Only 1 placed student was wrongly predicted as not placed (False Negative).

Summary :

The confusion matrix shows that *the Random Forest model correctly predicted 44 placed students and 12 not placed students*. It made only *1 false negative error* (misclassifying a placed student as not placed) and *8 false positives (misclassifying not placed students as placed*). Compared to Logistic Regression, this model showed much stronger performance in both class recall and overall prediction accuracy, especially by reducing false negatives — a critical factor in real-world placement prediction tasks.

Model Tuning for Support Vector Machine:

Hyperparameter Grid:

```python
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# Step 1: Define the base SVM model
svm_model = SVC(probability=True, random_state=42)

# Step 2: Set up the hyperparameter grid
svm_params = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf'],
    'gamma': ['scale', 'auto']
}
```

Base SVM Model :

SVC() = Support Vector Classifier

probability=True: allows us to use this model in *soft voting late. This uses Platt scaling to approximate probabilities*

random_state=42 for reproducibility

- Just like in Logistic Regression, I experimented with different values for the regularization parameter C ([0.1, 1, 10]). A smaller C value like 0.1 applies stronger regularization, which helps prevent overfitting by allowing the model to focus on the broader pattern instead of fitting the training data too closely. On the other hand, a larger C like 10 applies weaker regularization, which makes the model try harder to fit the training data exactly — increasing the risk of overfitting.
- I tested two different kernel types: 'linear' and 'rbf'. The *kernel* parameter in SVM determines the shape of the decision boundary. The 'linear' kernel works well when the data is linearly separable — meaning the classes can be divided by a straight line (or hyperplane). The 'rbf' (Radial Basis Function) kernel, on

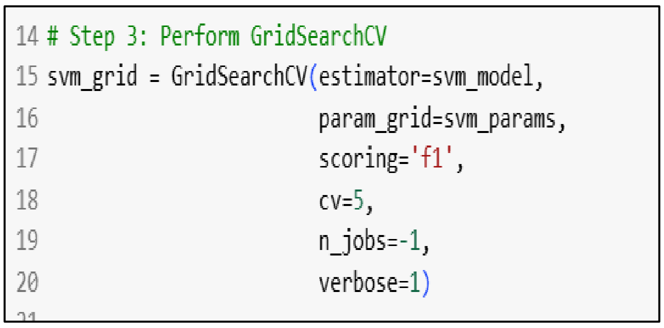
the other hand, is used *for non-linear problems.* It creates a more flexible, curved decision boundary that can better fit complex patterns in the data. Including both options allowed the model to choose between a simple and a more complex boundary based on the dataset.

- I tested two values for the gamma parameter: 'scale' and 'auto'. This parameter controls how much influence a single training example has on the decision boundary — it only affects models using the 'rbf' kernel.

  'scale' is the default and calculates gamma based on the number of features and their variance. It often gives more stable and balanced results.

  'auto' sets gamma to a constant based only on the number of features, which can sometimes lead to underfitting or overfitting, depending on the data.

Perform GridSearchCV:



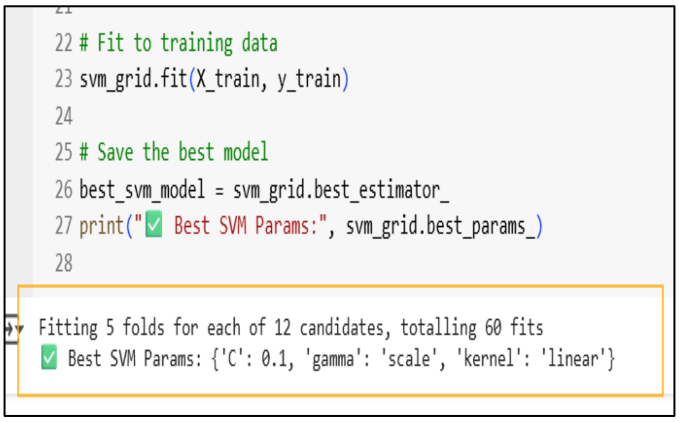
```
# Step 3: Perform GridSearchCV
svm_grid = GridSearchCV(estimator=svm_model,
                        param_grid=svm_params,
                        scoring='f1',
                        cv=5,
                        n_jobs=-1,
                        verbose=1)
```

Try **all combinations** of C, kernel, and gamma

Use **F1-score** as the evaluation metric (for class imbalance)

Perform **5-fold cross-validation** on the training data

Train the best-performing SVM model and Output:



```
# Fit to training data
svm_grid.fit(X_train, y_train)

# Save the best model
best_svm_model = svm_grid.best_estimator_
print("✅ Best SVM Params:", svm_grid.best_params_)
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
✅ Best SVM Params: {'C': 0.1, 'gamma': 'scale', 'kernel': 'linear'}
```

C = 0.1: Stronger regularization, which helps prevent overfitting.

kernel = 'linear' : The model found a **straight-line boundary** was sufficient — the data was likely linearly separable enough.

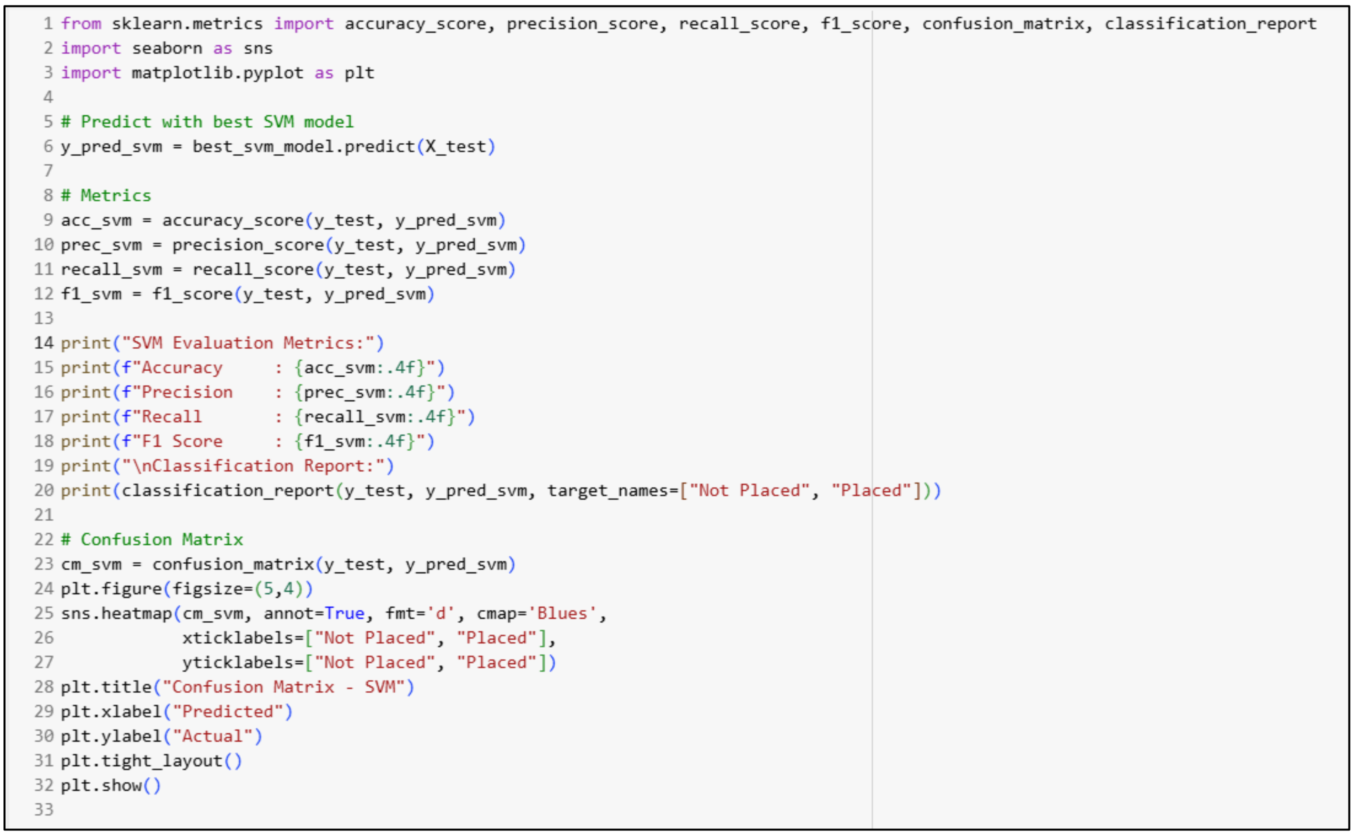
gamma = 'scale' :The default value, balancing how much influence each training example has. It's safe and generally performs well.

Interpretation:

The Support Vector Machine (SVM) model was tuned using GridSearchCV to explore combinations of regularization strength (C), kernel type (linear and rbf), and gamma (scale and auto).

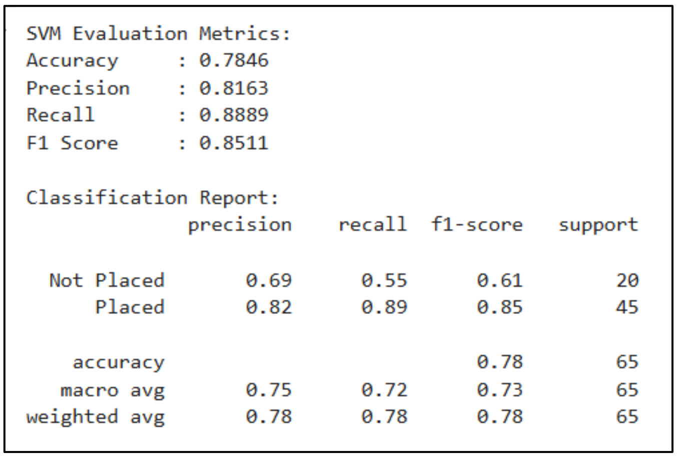
The best-performing configuration was C = 0.1, kernel = 'linear', and gamma = 'scale'. This suggests that a linear decision boundary with strong regularization provided the best balance between fitting the training data and generalizing to new data.

Model Evaluation for Support Vector Machine (accuracy, precision, recall, F1-score)



```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt

# Predict with best SVM model
y_pred_svm = best_svm_model.predict(X_test)

# Metrics
acc_svm = accuracy_score(y_test, y_pred_svm)
prec_svm = precision_score(y_test, y_pred_svm)
recall_svm = recall_score(y_test, y_pred_svm)
f1_svm = f1_score(y_test, y_pred_svm)

print("SVM Evaluation Metrics:")
print(f"Accuracy      : {acc_svm:.4f}")
print(f"Precision     : {prec_svm:.4f}")
print(f"Recall        : {recall_svm:.4f}")
print(f"F1 Score      : {f1_svm:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred_svm, target_names=["Not Placed", "Placed"]))

# Confusion Matrix
cm_svm = confusion_matrix(y_test, y_pred_svm)
plt.figure(figsize=(5,4))
sns.heatmap(cm_svm, annot=True, fmt='d', cmap='Blues',
            xticklabels=["Not Placed", "Placed"],
            yticklabels=["Not Placed", "Placed"])
plt.title("Confusion Matrix - SVM")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.tight_layout()
plt.show()

```

Output :

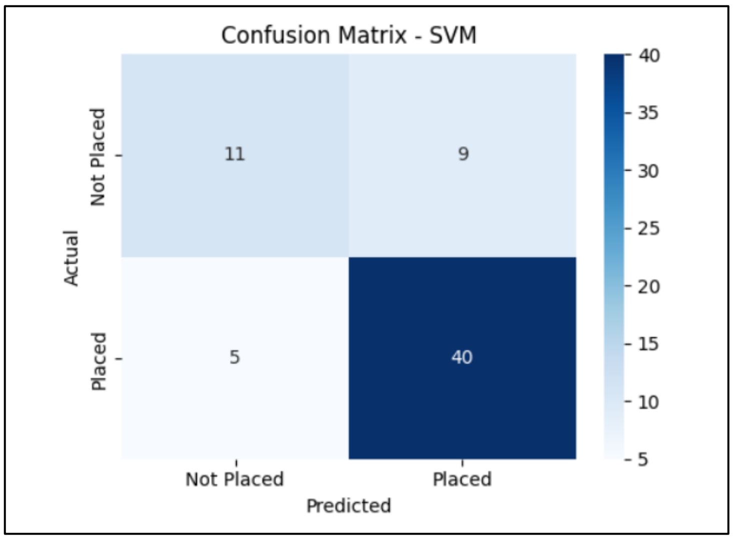


```
SVM Evaluation Metrics:
Accuracy      : 0.7846
Precision     : 0.8163
Recall        : 0.8889
F1 Score      : 0.8511

Classification Report:
              precision    recall  f1-score   support

  Not Placed       0.69      0.55      0.61        20
      Placed       0.82      0.89      0.85        45

    accuracy                           0.78        65
   macro avg       0.75      0.72      0.73        65
weighted avg       0.78      0.78      0.78        65
```

Summary :

The SVM model achieved an accuracy of **78.46%**, a precision of **81.63%**, recall of **88.89%**, and an F1 score of **85.11%**. It demonstrated strong performance in identifying placed students, like the other models. However, its ability to detect students who were not placed was slightly lower, with a recall of **55%**. This suggests that while SVM provides good overall performance, especially for the majority class, it may underperform in comparison to Random Forest when it comes to handling class imbalance.

Confusion Matrix:



The model *correctly predicted 40 placed students.*

It also correctly predicted *11 not placed students.*

It missed *5 placed students (false negatives).*

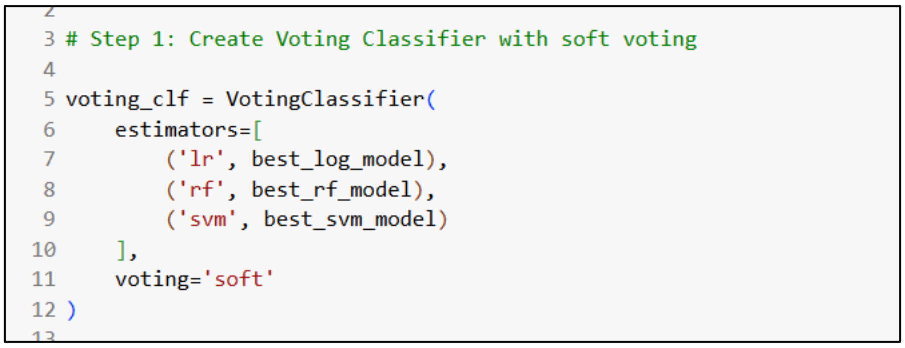
It *incorrectly predicted 9 unplaced students as placed (false positives).*

Summary :

The confusion matrix shows the SVM model correctly identified 40 placed students and 11 not placed students. However, it also misclassified 9 unplaced students as placed and missed 5 placed students. While the model handles the majority class well, its performance on the minority class ("Not Placed") still leaves room for improvement.

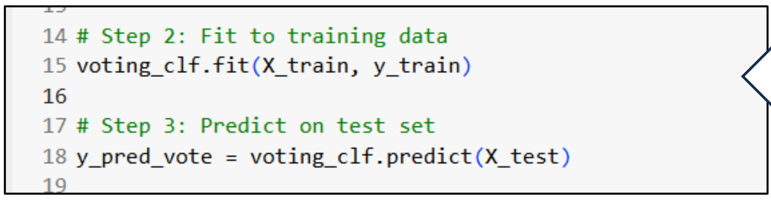
Voting Classifier

Voting Classifier combined model using the best-tuned Logistic Regression, Random Forest, and SVM. 'soft' voting uses probability outputs (how confident each model is about each class)



```
# Step 1: Create Voting Classifier with soft voting

voting_clf = VotingClassifier(
    estimators=[
        ('lr', best_log_model),
        ('rf', best_rf_model),
        ('svm', best_svm_model)
    ],
    voting='soft'
)
```

Train and Predict on Test Set

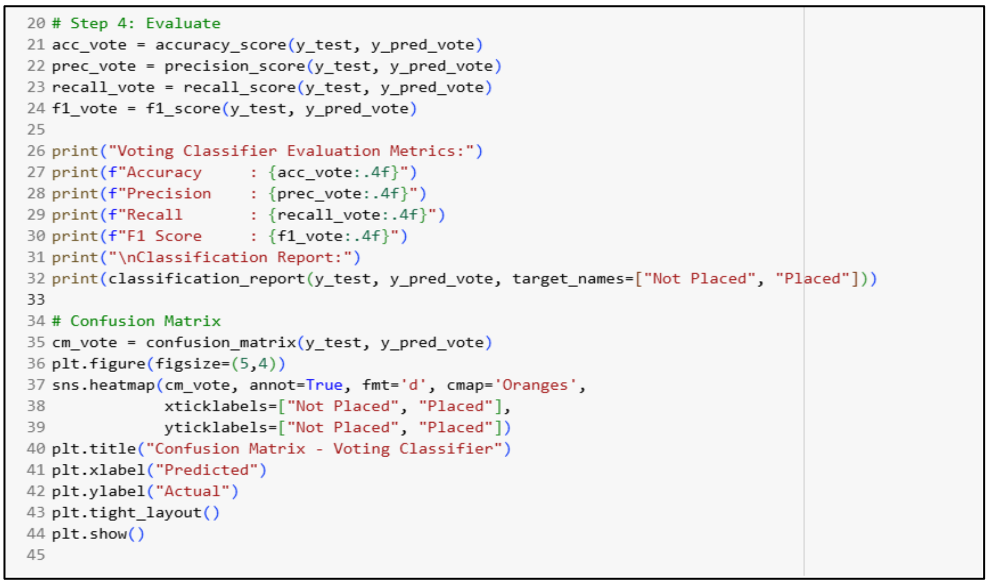


```
# Step 2: Fit to training data
voting_clf.fit(X_train, y_train)

# Step 3: Predict on test set
y_pred_vote = voting_clf.predict(X_test)
```

Step 2: This trains the voting classifier on your training set

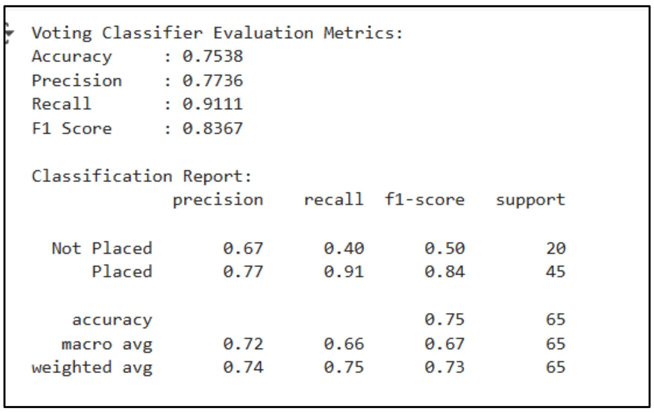
Step 3: Used all three models' outputs, combines them (soft voting), and makes a final prediction for each student in the test set.

Voting Classifier Evaluation (accuracy, precision, recall, F1-score)



```python
# Step 4: Evaluate
acc_vote = accuracy_score(y_test, y_pred_vote)
prec_vote = precision_score(y_test, y_pred_vote)
recall_vote = recall_score(y_test, y_pred_vote)
f1_vote = f1_score(y_test, y_pred_vote)

print("Voting Classifier Evaluation Metrics:")
print(f"Accuracy     : {acc_vote:.4f}")
print(f"Precision    : {prec_vote:.4f}")
print(f"Recall       : {recall_vote:.4f}")
print(f"F1 Score     : {f1_vote:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred_vote, target_names=["Not Placed", "Placed"]))

# Confusion Matrix
cm_vote = confusion_matrix(y_test, y_pred_vote)
plt.figure(figsize=(5,4))
sns.heatmap(cm_vote, annot=True, fmt='d', cmap='Oranges',
            xticklabels=["Not Placed", "Placed"],
            yticklabels=["Not Placed", "Placed"])
plt.title("Confusion Matrix - Voting Classifier")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.tight_layout()
plt.show()
```

Output:



```
Voting Classifier Evaluation Metrics:
Accuracy     : 0.7538
Precision    : 0.7736
Recall       : 0.9111
F1 Score     : 0.8367

Classification Report:
              precision    recall  f1-score   support

  Not Placed       0.67      0.40      0.50        20
      Placed       0.77      0.91      0.84        45

    accuracy                           0.75        65
   macro avg       0.72      0.66      0.67        65
weighted avg       0.74      0.75      0.73        65
```

Summary :

A soft-voting ensemble was created using the tuned Logistic Regression, Random Forest, and SVM models.

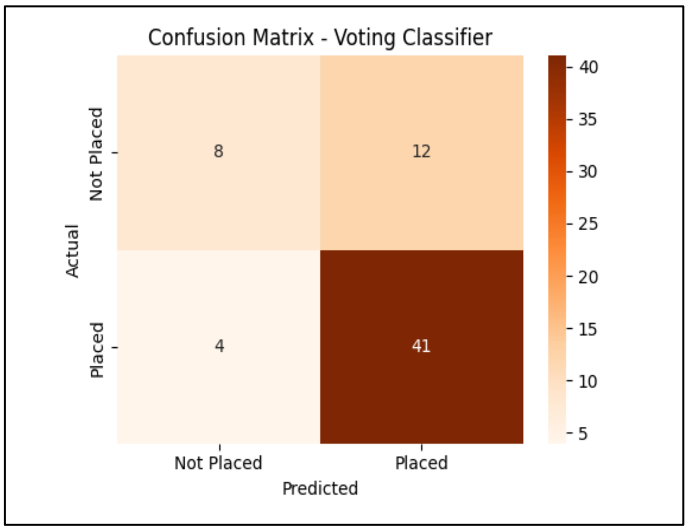
The Voting Classifier achieved *an accuracy of 75.38%, a precision of 77.36%, a recall of 91.11%, and an F1 score of 83.67%.*

While it maintained high recall for the majority class ("Placed"), performance on the minority class ("Not Placed") dropped, with a recall of only 40%.

This suggests that the ensemble leaned toward the majority class, possibly due to how the soft voting averaged the predicted probabilities.

Although ensemble methods are generally expected to improve performance, in this case the Voting Classifier did not outperform the best individual model (Random Forest).

Confusion Matrix:



The model *correctly predicted 41 placed students.*

It also *correctly predicted 8 students as not placed.*

It *misclassified 12 not placed students as placed (False Positives) — this led to the low 40% recall for "Not Placed".*

It *missed 4 placed students (False Negatives)*.

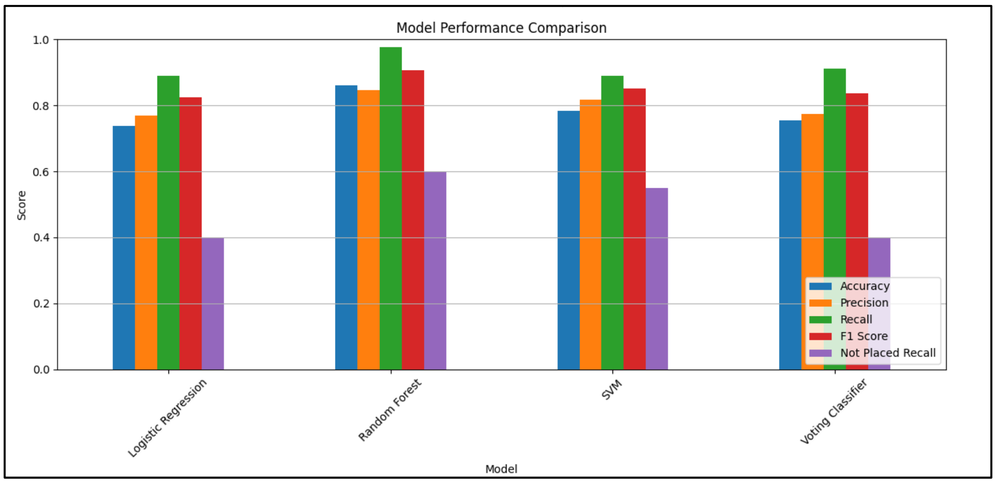
Summary :

The confusion matrix confirms that the Voting Classifier performed well in identifying placed students (41 correct out of 45), but struggled with the minority class, misclassifying 12 out of 20 not placed students. This aligns with the low recall observed for the "Not Placed" class and highlights how the ensemble leaned heavily toward the majority class.

Model Comparison Table

Model	Accuracy	Precision	Recall	F1 Score	Not Placed Recall
Logistic Regression	0.7385	0.7692	0.8889	0.8247	0.4
Random Forest	0.8615	0.8462	0.9778	0.9072	0.6
SVM	0.7846	0.8163	0.8889	0.8511	0.55
Voting Classifier	0.7538	0.7736	0.9111	0.8367	0.4

Note: "Not Placed Recall" is shown separately to highlight how well each model handled the *minority class.*



Summary :

Among the four models evaluated, **Random Forest performed the best overall**, achieving the highest accuracy (86.15%), F1-score (90.72%), and recall (97.78%) for the "Placed" class. It also had the strongest recall (60%) for the "Not Placed" class, indicating better handling of class imbalance.

While the Voting Classifier maintained high recall for the majority class, its performance on the minority class (40% recall) did not improve upon Logistic Regression or SVM. This suggests that ensemble averaging leaned toward the majority prediction.

SVM provided a good balance with strong F1 and recall, while **Logistic Regression** served as a solid baseline with respectable precision and recall, though with weaker minority class detection.