03 - Machine Learning Modeling with Stacking Classifier

In this notebook, we will:

- Load the cleaned HR dataset.
- Prepare and encode features.
- Split the data into training and testing sets.
- Build and evaluate a stacking classifier ensemble to predict employee attrition.

A stacking classifier combines multiple base learners (e.g., logistic regression, random forest, SVC) and uses a meta-model (here, logistic regression) to improve predictive performance.

Import Libraries & Load Data

We start by importing necessary libraries and loading the cleaned dataset (saved as "hr_data_cleaned.csv").

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

# Machine learning libraries
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier,
StackingClassifier
from sklearn.svm import SVC


# Set plotting style
sns.set(style="whitegrid")
plt.rcParams["figure.figsize"] = (12, 8)

# Load the cleaned dataset
df = pd.read_csv("hr_data_cleaned.csv")
print("DataFrame shape:", df.shape)
df.head()
```

```
DataFrame shape: (1470, 36)
```

```
   Age Attrition     BusinessTravel  DailyRate              Department  \
0   41       Yes      Travel_Rarely       1102                   Sales
```

```
1   49        No  Travel_Frequently        279  Research & Development

2   37       Yes      Travel_Rarely       1373  Research & Development

3   33        No  Travel_Frequently       1392  Research & Development

4   27        No      Travel_Rarely        591  Research & Development


   DistanceFromHome  Education EducationField  EmployeeCount
EmployeeNumber  \
0                 1          2  Life Sciences              1
1
1                 8          1  Life Sciences              1
2
2                 2          2          Other              1
4
3                 3          4  Life Sciences              1
5
4                 2          1        Medical              1
7

   ...  StandardHours StockOptionLevel  TotalWorkingYears  \
0  ...             80                0                  8
1  ...             80                1                 10
2  ...             80                0                  7
3  ...             80                0                  8
4  ...             80                1                  6

   TrainingTimesLastYear  WorkLifeBalance YearsAtCompany
YearsInCurrentRole  \
0                      0                1              6
4
1                      3                3             10
7
2                      3                3              0
0
3                      3                3              8
7
4                      3                3              2
2

  YearsSinceLastPromotion  YearsWithCurrManager  TenureBucket
0                       0                     5           3-6
1                       1                     7          6-10
2                       0                     0           NaN
3                       3                     0          6-10
4                       2                     2            <3

[5 rows x 36 columns]
```

Feature Encoding & Train-Test Split

Our target variable is **Attrition**. We convert 'Yes' to 1 and 'No' to 0. For features, we select a few predictors (e.g., Age, DistanceFromHome, MonthlyIncome, OverTime, BusinessTravel) and then use one-hot encoding for the categorical columns.

```
# Create a binary target column for attrition
df["AttritionFlag"] = df["Attrition"].map({"Yes": 1, "No": 0})

# Define the feature list (modify as needed)
features = ["Age", "DistanceFromHome", "MonthlyIncome", "OverTime",
"BusinessTravel"]

# One-hot encode categorical columns in our feature set
df_encoded = pd.get_dummies(df[features], drop_first=True)

# Prepare feature matrix X and target vector y
X = df_encoded.values
y = df["AttritionFlag"].values

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=42)

print("Training set shape:", X_train.shape)
print("Test set shape:", X_test.shape)
```

```
Training set shape: (1176, 6)
Test set shape: (294, 6)
```

Build & Train the Stacking Classifier

We define a stacking classifier that combines three base learners:

- Logistic Regression
- Random Forest
- Support Vector Classifier (with probability estimates enabled)

A logistic regression model is used as the final estimator. We then train the stacking classifier on the training data.

```
# Define base learners
estimators = [
    ('lr', LogisticRegression(max_iter=1000)),
    ('rf', RandomForestClassifier(n_estimators=100, random_state=42)),
    ('svc', SVC(probability=True))
]
```

```
# Create the stacking classifier with logistic regression as the meta-model
stacking_clf = StackingClassifier(
    estimators=estimators,
    final_estimator=LogisticRegression(max_iter=1000),
    cv=5
)

# Train the stacking classifier on the training data
stacking_clf.fit(X_train, y_train)
```

```
StackingClassifier(cv=5,
                   estimators=[('lr',
LogisticRegression(max_iter=1000)),
                               ('rf',
RandomForestClassifier(random_state=42)),
                               ('svc', SVC(probability=True))],
                   final_estimator=LogisticRegression(max_iter=1000))
```

Evaluate the Stacking Classifier

After training, we made predictions on the test set and evaluated the model using accuracy and a detailed classification report.

```
# Make predictions on the test set using the stacking classifier
y_pred_stack = stacking_clf.predict(X_test)

# Evaluate the performance
stacking_accuracy = accuracy_score(y_test, y_pred_stack)
print(f"Stacking Classifier Accuracy: {stacking_accuracy:.3f}")

print("\nClassification Report (Stacking Classifier):")
print(classification_report(y_test, y_pred_stack))
```

```
Stacking Classifier Accuracy: 0.857

Classification Report (Stacking Classifier):
              precision    recall  f1-score   support

           0       0.87      0.98      0.92       255
           1       0.29      0.05      0.09        39

    accuracy                           0.86       294
   macro avg       0.58      0.52      0.50       294
weighted avg       0.79      0.86      0.81       294
```

Addressing Class Imbalance

After observing that our HR dataset was imbalanced—meaning we had many more "No Attrition" cases than "Yes Attrition"—we added `class_weight='balanced'` to each of our base learners and the final estimator in the stacking classifier. This modification helps the model pay closer attention to the minority class (employees who leave), thereby improving metrics like **recall** and **F1-score** for attrition. While overall accuracy might remain similar or decrease slightly, detecting those at risk of leaving is a higher priority in many HR scenarios, making this trade-off worthwhile.

```
# Define base learners with class weighting to handle class imbalance
estimators = [
    ('lr', LogisticRegression(max_iter=1000,
class_weight='balanced')),
    ('rf', RandomForestClassifier(n_estimators=100, random_state=42,
class_weight='balanced')),
    ('svc', SVC(probability=True, class_weight='balanced'))
]

# Create the stacking classifier with logistic regression as the meta-
model, also using balanced class weights
stacking_clf = StackingClassifier(
    estimators=estimators,
    final_estimator=LogisticRegression(max_iter=1000,
class_weight='balanced'),
    cv=5
)

# Train the stacking classifier on the training data
stacking_clf.fit(X_train, y_train)

StackingClassifier(cv=5,
                   estimators=[('lr',

LogisticRegression(class_weight='balanced',
                                                   max_iter=1000)),
                               ('rf',

RandomForestClassifier(class_weight='balanced',

random_state=42)),
                               ('svc',
                                SVC(class_weight='balanced',
                                    probability=True))],

final_estimator=LogisticRegression(class_weight='balanced',
                                                      max_iter=1000))

# Make predictions on the test set using the stacking classifier
y_pred_stack = stacking_clf.predict(X_test)
```

```python
# Evaluate the performance
stacking_accuracy = accuracy_score(y_test, y_pred_stack)
print(f"Stacking Classifier Accuracy: {stacking_accuracy:.3f}")

print("\nClassification Report (Stacking Classifier):")
print(classification_report(y_test, y_pred_stack))
```

```
Stacking Classifier Accuracy: 0.759

Classification Report (Stacking Classifier):
              precision    recall  f1-score   support

           0       0.92      0.79      0.85       255
           1       0.28      0.54      0.37        39

    accuracy                           0.76       294
   macro avg       0.60      0.67      0.61       294
weighted avg       0.83      0.76      0.79       294
```

With `class_weight='balanced'` enabled for each model in the stacking ensemble, we observe the following key changes:

1. **Overall Accuracy Decrease (from ~0.86 to ~0.76):**
   - The model now makes more errors on the majority class (No Attrition), causing a drop in overall accuracy. This is a tradeoff we are willing to take to better identify attrition rates.
2. **Significant Recall Improvement for Class 1 (Attrition):**
   - Recall jumped from a very low ~0.05 to 0.54, meaning the model now correctly identifies over half of the employees who actually leave.

   - The F1-score for Class 1 also increased from ~0.09 to 0.37, reflecting a better balance between precision and recall for the minority class.
3. **Class 0 (No Attrition) Performance:**
   - Precision remains high at 0.92, but recall dropped to 0.79 from ~0.98 previously. This indicates the model is now more likely to classify some “No Attrition” employees as “Attrition,” increasing false positives.

Overall Takeaway

By incorporating `class_weight='balanced'`, we prioritized correctly identifying the minority class (Attrition) at the expense of some accuracy on the majority class. For our analysis, this tradeoff is worth it as we want to prioritise when there is attrition in a company.

```python
# Select 5 random samples from the test set
sample_indices = np.random.choice(X_test.shape[0], size=5,
replace=False)
sample_features = X_test[sample_indices]
```

```python
sample_true = y_test[sample_indices]

# Predict probabilities and predictions for these samples
sample_probs = stacking_clf.predict_proba(sample_features)
sample_preds = stacking_clf.predict(sample_features)

# Create a DataFrame to display sample predictions
sample_df = pd.DataFrame(sample_features, columns=df_encoded.columns)
sample_df["TrueAttrition"] = sample_true
sample_df["PredictedAttrition"] = sample_preds
sample_df["Probability_No"] = sample_probs[:, 0]
sample_df["Probability_Yes"] = sample_probs[:, 1]
sample_df
```

```
  Age DistanceFromHome MonthlyIncome OverTime_Yes  \
0  35                1          2977        False
1  37               10          4680        False
2  46                9         10096        False
3  55                1         19045         True
4  42                8         18430        False

  BusinessTravel_Travel_Frequently BusinessTravel_Travel_Rarely  \
0                            False                         True
1                            False                         True
2                            False                         True
3                            False                         True
4                             True                        False

   TrueAttrition  PredictedAttrition  Probability_No  Probability_Yes

0              0                   0        0.663633         0.336367

1              0                   0        0.526459         0.473541

2              1                   1        0.455523         0.544477

3              0                   0        0.670094         0.329906

4              0                   0        0.750414         0.249586
```

From our small sample of 5, our model manages to correctly identify the true attrition, but the confidence level is still quite low indicating that the model's confidence in these predictions is still quite uncertain. We also experimented with SMOTE to oversample the minority class, aiming to provide the model with more examples of attrition. Unfortunately, when combined with other tuning techniques, the resulting F1 score was worse compared to using balanced class weights alone. This suggests that the synthetic samples generated by SMOTE may not have perfectly captured the true distribution of the minority class or maybe it introduced noise that made the f1 score lower.