Data Splitting and Model Training Methods

REDA OUZIDANE

April 3, 2025

1 Introduction

In machine learning, splitting the dataset into training and testing subsets is crucial for evaluating a model's performance. Several methods are available for splitting data, including:

- Train/Test Split
- K-Fold Cross-Validation
- Stratified Shuffle Split
- Hyperparameter Tuning with Cross-Validation (GridSearchCV/RandomizedSearchCV)

2 Train/Test Split

The simplest method for splitting data is the **Train/Test Split**, where the data is divided into two parts: one for training the model and the other for testing its performance. The method works well for large datasets and when a quick evaluation is needed.

2.1 Python Implementation

```
from sklearn.model_selection import train_test_split

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=9)
```

3 K-Fold Cross-Validation

K-Fold Cross-Validation divides the dataset into k subsets or "folds". For each fold, the model is trained on k-1 folds and tested on the remaining fold. This process is repeated for each fold. It is more reliable than the train/test split, especially for smaller datasets.

3.1 Python Implementation

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

# Define the model
model = RandomForestClassifier(random_state=9)

# Perform cross-validation (e.g., 5-fold cross-validation)
cv_scores = cross_val_score(model, X, y, cv=5)

# Output the cross-validation scores
print("Cross-validation scores:", cv_scores)
print("Mean cross-validation score:", cv_scores.mean())
```

4 Stratified Shuffle Split

Stratified Shuffle Split ensures that each split preserves the distribution of target classes. This method is particularly useful when dealing with imbalanced datasets.

4.1 Python Implementation

```
from sklearn.model_selection import StratifiedShuffleSplit

# Define the stratified shuffle split (e.g., 80% training, 20% testing)
sss = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=9)

for train_index, test_index in sss.split(X, y):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
```

5 Hyperparameter Tuning with Cross-Validation

GridSearchCV and **RandomizedSearchCV** combine cross-validation with hyperparameter tuning to find the optimal parameters for the model. These methods are more computationally expensive but can lead to better model performance.

5.1 Python Implementation

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

# Define the model
```

```
model = RandomForestClassifier(random_state=9)

# Define the parameter grid to search through
param_grid = {
    'n_estimators': [10, 50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
}

# Use GridSearchCV to search for the best parameters (with cross-validation)
grid_search = GridSearchCV(model, param_grid, cv=5, scoring='accuracy')

# Fit the model on the training data
grid_search.fit(X_train, y_train)

# Get the best parameters and best score
print("Best parameters found: ", grid_search.best_params_)
print("Best cross-validation score: ", grid_search.best_score_)
```

6 Conclusion

Each method has its own strengths and use cases:

- **Train/Test Split**: Best for large datasets and quick evaluation.
- **K-Fold Cross-Validation**: Best for smaller datasets or when a more robust evaluation is needed.
- **Stratified Shuffle Split**: Ideal for imbalanced datasets.
- **GridSearchCV/RandomizedSearchCV**: Best for hyperparameter tuning combined with cross-validation.