Data Splitting

Training Material

Table of Contents

Chapter 1: Data Splitting and Types

Data splitting is a common task in data science and machine learning. It typically refers to dividing data into different subsets, often for training and testing models. In Python, data splitting can be done in several ways depending on the context. Here's a breakdown of different approaches:

1. **String Splitting:** The split() method is used to split strings into a list based on a specified delimiter (by default, it's a space).

python

```
text = "Python is great"
words = text.split()  # Splits based on space
print(words)  # Output: ['Python', 'is', 'great']
```

You can specify a custom delimiter:

python

```
data = "apple,banana,cherry"
fruits = data.split(",")  # Splits based on the comma
print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

2. **Splitting Lists:** Lists can be split into multiple parts using slicing. For example, splitting a list into two halves.

python

```
numbers = [1, 2, 3, 4, 5, 6]
mid = len(numbers) // 2
first_half = numbers[:mid]
second_half = numbers[mid:]
```

```
print(first_half)   # Output: [1, 2, 3]
print(second_half)  # Output: [4, 5, 6]
```

3. **Splitting Data for Machine Learning:** For machine learning tasks, data is often split into training and testing sets. A popular library for this is scikit-learn. The train_test_split function can be used to randomly split datasets.

python

```
from sklearn.model_selection import train_test_split

X = [[1], [2], [3], [4], [5], [6]]  # Features
y = [0, 1, 0, 1, 0, 1]  # Labels

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

print(X_train)  # Output: training features
print(X_test)   # Output: testing features
```

4. **Splitting Data in Pandas:** Pandas, a popular data manipulation library, allows splitting data into groups using the groupby() method. This is often used to group data based on one or more columns.

python

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Score': [85, 90, 78, 92],
        'Class': ['A', 'B', 'A', 'B']}
```

```
df = pd.DataFrame(data)


grouped = df.groupby('Class')
for class_name, group in grouped:
    print(f"Class: {class_name}")
    print(group)
```

Output:

vbnet

```
Class: A
     Name  Score Class
0    Alice     85     A
2  Charlie     78     A


Class: B
   Name  Score Class
1   Bob     90     B
3 David     92     B
```

Data Types in Python

Python has several built-in data types that are categorized based on how they store and represent data. Here are the most common ones:

1. **Numeric Types:**

   int: Represents whole numbers (e.g., 5, -10, 0).

   float: Represents decimal numbers (e.g., 3.14, -2.7, 0.0).

   complex: Represents complex numbers (e.g., 2+3j).

Example:

python

```
a = 5        # int
b = 3.14     # float
c = 2 + 3j   # complex
```

2. **Sequence Types:**

   list: An ordered collection of items, mutable (can be changed).

python

```
my_list = [1, 2, 3, 4, 5]
```

   tuple: An ordered collection of items, immutable (cannot be changed).

python

```
my_tuple = (1, 2, 3, 4, 5)
```

   str: A sequence of characters (string).

python

```
my_string = "Hello, World!"
```

3. **Mapping Types:**

   dict: A collection of key-value pairs, unordered, mutable.

python

```
my_dict = {'name': 'Alice', 'age': 25}
```

4. **Set Types:**

   set: An unordered collection of unique elements.

python

```
my_set = {1, 2, 3, 4}
```

   frozenset: An immutable version of a set.

python

```
my_frozenset = frozenset([1, 2, 3, 4])
```

5. **Boolean Type:**

   bool: Represents True or False.

python

```
is_valid = True
```

6. **None Type:**

   NoneType: Represents the absence of a value.

python

```
result = None
```

Type Conversion

You can convert between different types using built-in functions:

- int(): Converts to an integer.
- float(): Converts to a float.
- str(): Converts to a string.
- list(): Converts to a list.
- tuple(): Converts to a tuple.
- set(): Converts to a set.

Example:

python

```
x = "123"       # String
y = int(x)      # Convert to integer
z = float(x)    # Convert to float
```

```
print(type(x))  # <class 'str'>
print(type(y))  # <class 'int'>
print(type(z))  # <class 'float'>
```

This covers basic data splitting techniques and common data types in Python.

Chapter 2: Steps-Training and Testing Split

The process of splitting data into training and testing sets is fundamental in machine learning. It allows us to train a model on one part of the data (training set) and then test its performance on unseen data (testing set). This helps to ensure that the model generalizes well to new data, rather than just memorizing the training data.

Here is a detailed step-by-step explanation of how you can split data for training and testing in Python:

Step 1: Import Necessary Libraries

To split data, you'll typically use the train_test_split() function from the scikit-learn library. You first need to install the library (if not already installed) and import the necessary module.

bash

pip install scikit-learn

Then, import the train_test_split function:

python

from sklearn.model_selection import train_test_split

Step 2: Prepare the Data

The dataset typically consists of two components:

Features (X): These are the input variables that the model will use to learn.

Labels (y): These are the target values that the model will try to predict.

For demonstration, let’s use a simple dataset:

python

Example dataset

X = [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]] # Features (input data)

y = [0, 0, 0, 0, 1, 1, 1, 1, 1, 1] # Labels (target data)

Here, X is a list of numerical inputs, and y is the list of corresponding labels. For simplicity, we assume these are binary labels (0s and 1s), though they could be anything, such as multi-class labels or continuous values in regression tasks.

Step 3: Perform the Split

Using the train_test_split() function, you can split your dataset into training and testing subsets. This function randomly splits the data based on the proportion you define.

The main parameters of train_test_split() are:

X: Feature data

y: Labels

test_size: Proportion of data to be set aside for testing (e.g., 0.2 for 20%)

random_state: Ensures reproducibility by controlling the randomness of the split

shuffle: Shuffles the data before splitting (default is True)

Here's an example where 30% of the data is reserved for testing, and the remaining 70% for training:

python

Split the dataset

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

print("Training Features (X_train):", X_train)

print("Testing Features (X_test):", X_test)

print("Training Labels (y_train):", y_train)

print("Testing Labels (y_test):", y_test)

Step 4: Examine the Result

The train_test_split() function returns four variables:

X_train: The training set features.

X_test: The testing set features.

y_train: The training set labels.

y_test: The testing set labels.

Example output:

python

Training Features (X_train): [[6], [10], [3], [1], [7], [9], [4]]

Testing Features (X_test): [[8], [5], [2]]

Training Labels (y_train): [1, 1, 0, 0, 1, 1, 0]

Testing Labels (y_test): [1, 1, 0]

Here:

Training Set: Contains 7 samples for training the model.

Testing Set: Contains 3 samples for evaluating the model after training.

Step 5: Use the Split Data for Model Training and Testing

Once the data is split, you can proceed to train a machine learning model on the training data (X_train, y_train) and then test its performance on the testing data (X_test, y_test). Here's an example using a logistic regression model:

python

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Initialize a Logistic Regression model
model = LogisticRegression()
```

```
# Train the model on the training data
model.fit(X_train, y_train)


# Predict on the test data
y_pred = model.predict(X_test)


# Evaluate the accuracy of the model on the test set
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy * 100:.2f}%")
```

Step 6: Understanding Data Splitting Best Practices

Random State: Using the random_state parameter ensures that every time you run the code, you get the same split. This is important for reproducibility, especially when sharing your work with others or debugging.

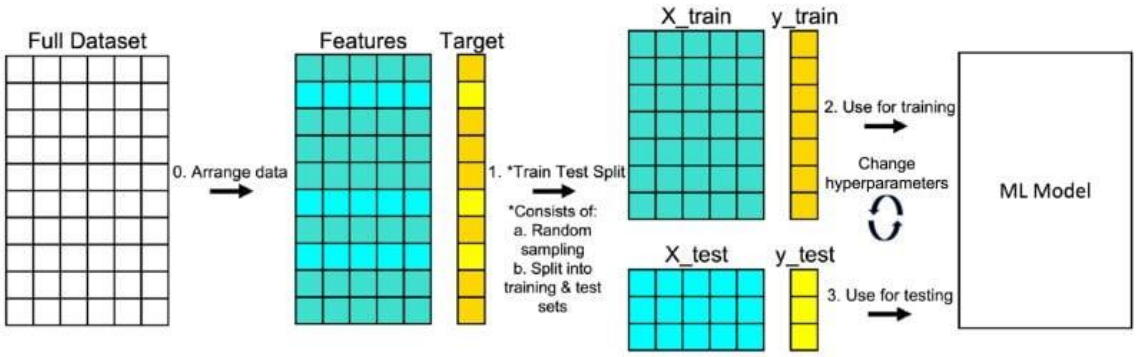
Stratified Splitting: In classification tasks, when the data has an imbalanced distribution of classes (for example, more instances of class 0 than class 1), a simple random split may result in unequal class distributions in training and test sets. In such cases, it's a good practice to use stratified splitting to ensure that both sets have approximately the same proportion of classes.

Example with stratification:

python

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

Different Splitting Ratios: While the common practice is to use 70% of the data for training and 30% for testing, other ratios such as 80/20 or 90/10 may also be used, depending on the dataset size and the nature of the problem.



Conclusion

The **train-test split** process ensures that the model generalizes well to unseen data. Following the above steps ensures a clean and effective workflow for splitting your dataset in Python:

1. **Import** the necessary functions.
2. **Prepare** your dataset.
3. **Perform** the split.
4. **Use** the split data for training and testing your model.

This process helps avoid overfitting and ensures that the model's performance is evaluated on data that was not seen during training.

Chapter 3: Cross Validation and Types

Cross-validation is a technique used in machine learning to assess how well a model generalizes to unseen data. Instead of splitting the data into just one training and testing set (like in a simple train-test split), cross-validation divides the data into multiple subsets, trains the model on different combinations of these subsets, and tests it on the remaining portions. This ensures that the model's performance is evaluated more reliably across various splits of the data.

Why Use Cross-Validation?

More Reliable Evaluation: Instead of relying on a single train-test split, which could be biased depending on how the data was split, cross-validation provides a better estimate of the model's performance by averaging results across multiple iterations.

Better Generalization: Cross-validation helps detect overfitting by showing how the model performs on different subsets of data, not just one specific test set.

Types of Cross-Validation

1. K-Fold Cross-Validation
2. Stratified K-Fold Cross-Validation
3. Leave-One-Out Cross-Validation (LOOCV)
4. Leave-P-Out Cross-Validation
5. Time Series Split

1. K-Fold Cross-Validation

K-Fold Cross-Validation splits the data into K equally sized subsets (folds). The model is trained on K-1 folds and tested on the remaining fold. This process is repeated K times, with each fold used as the test set once. The final performance metric is the average of the metrics across all K iterations.

Steps for K-Fold Cross-Validation:

Divide the data into K folds.

For each iteration:

Use Case K-1 fold for testing

Use the remaining fold for testing

Average the results over all iterations.

Example in Python:

python

Copy code

```
from sklearn.model_selection import KFold
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import numpy as np

# Sample data
X = np.array([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]])
y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

# Initialize KFold with 5 splits
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Initialize a logistic regression model
model = LogisticRegression()

accuracies = []

# Perform K-Fold cross-validation
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

```
    # Train the model
    model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)

# Calculate the mean accuracy
print(f"Average Accuracy: {np.mean(accuracies):.2f}")
```

2. Stratified K-Fold Cross-Validation

Stratified K-Fold Cross-Validation is similar to K-Fold, but it ensures that the class distribution (proportion of classes) is maintained in each fold. This is particularly important when you have an imbalanced dataset (e.g., a binary classification problem where one class is significantly more frequent than the other).

Example in Python:

python

```
from sklearn.model_selection import StratifiedKFold

# Initialize StratifiedKFold with 5 splits
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

accuracies = []
```

```
# Perform Stratified K-Fold cross-validation
for train_index, test_index in skf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Train the model
    model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)

print(f"Average Accuracy: {np.mean(accuracies):.2f}")
```

3. Leave-One-Out Cross-Validation (LOOCV)

Leave-One-Out Cross-Validation (LOOCV) is a special case of K-Fold cross-validation where K = N (i.e., the number of data points). In each iteration, the model is trained on N-1 data points and tested on the single remaining data point. This is repeated for every data point in the dataset.

LOOCV can provide a very accurate estimate of model performance but is computationally expensive, especially with large datasets.

Example in Python:

python

```
from sklearn.model_selection import LeaveOneOut

# Initialize LeaveOneOut cross-validator
```

```
loo = LeaveOneOut()

accuracies = []

# Perform LOOCV
for train_index, test_index in loo.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Train the model
    model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)

print(f"Average Accuracy: {np.mean(accuracies):.2f}")
```

4. Leave-P-Out Cross-Validation

Leave-P-Out Cross-Validation is a generalization of LOOCV. Instead of leaving out one data point, P data points are left out for testing, and the remaining data points are used for training. The process is repeated for all possible combinations of P data points.

This method is computationally expensive and typically used for small datasets.

Example in Python:

python

```
from sklearn.model_selection import LeavePOut

# Initialize LeavePOut with P=2 (leave 2 data points out for testing)
lpo = LeavePOut(p=2)

accuracies = []

# Perform Leave-P-Out Cross-Validation
for train_index, test_index in lpo.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Train the model
    model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)

print(f"Average Accuracy: {np.mean(accuracies):.2f}")
```

5. Time Series Cross-Validation (Time Series Split)

When working with time series data, the order of the data points is important, so traditional K-Fold cross-validation cannot be used because it would shuffle the data. Time Series Split ensures that the training set always contains earlier

data points, and the test set contains later data points, mimicking how real-world forecasting works.

Example in Python:

python

```
from sklearn.model_selection import TimeSeriesSplit

# Initialize TimeSeriesSplit with 3 splits
tscv = TimeSeriesSplit(n_splits=3)

accuracies = []

# Perform Time Series Split cross-validation
for train_index, test_index in tscv.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Train the model
    model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)

print(f"Average Accuracy: {np.mean(accuracies):.2f}")
```

Conclusion

K-Fold Cross-Validation is the most commonly used method, providing a good balance between bias and variance in the performance estimates.

Stratified K-Fold Cross-Validation is crucial for classification tasks with imbalanced data.

LOOCV gives the most accurate estimate but is computationally expensive for large datasets.

Leave-P-Out is a generalization of LOOCV but is also resource-intensive.

Time Series Split is essential for time-dependent data, ensuring temporal order is respected.

Cross-validation allows better assessment of a model's performance, ensuring that it generalizes well to unseen data and doesn't overfit to a single train-test split.

Advantages of Cross Validation:

Cross-validation is a crucial technique in machine learning and model evaluation. It helps in assessing how well a model generalizes to unseen data and provides more reliable estimates of model performance. Here are the key **advantages of using cross-validation** in Python:

1. Better Estimate of Model Performance

Cross-validation, by training and testing the model on multiple different subsets of the data, provides a better estimate of model performance than a simple train-test split. Instead of testing on just one train-test partition, cross-validation gives a more generalized result by averaging the performance across multiple splits.

In **K-Fold Cross-Validation**, for example, the model is trained on K-1 folds and tested on the remaining fold in each of the K iterations. This reduces the bias that can occur when evaluating the model on only one test set.

2. Reduces Overfitting Risk

One of the biggest advantages of cross-validation is its ability to **reduce overfitting**. In cases of overfitting, the model performs exceptionally well on the training data but poorly on unseen (testing) data. Cross-validation prevents

over-reliance on a single train-test split and ensures that the model is not "memorizing" the training data.

> It does this by forcing the model to be trained on different subsets of data and tested on unseen subsets during each iteration. If the model performs consistently well across all folds, it's a good indicator that it will generalize well to new data.

3. Uses the Entire Dataset for Both Training and Testing

In simple train-test splits, a portion of the data is reserved exclusively for testing, which reduces the amount of data available for training. With cross-validation, every observation in the dataset is used for both training and testing. This is particularly beneficial when you have a **small dataset**.

> In **K-Fold Cross-Validation**, for example, each data point is used in both the training set (K-1 times) and the test set (once). This makes efficient use of limited data.

4. Provides a More Robust Model Selection Process

When choosing between multiple models or tuning hyperparameters, cross-validation offers a more robust process by testing model performance on multiple data splits. Instead of relying on a single test set to evaluate different models, cross-validation averages the results across multiple test sets, leading to more informed decision-making.

> In **Hyperparameter Tuning**, cross-validation is often used in techniques like **Grid Search** or **Random Search** to find the best combination of hyperparameters that yields the highest accuracy or another performance metric.

5. Stratified Cross-Validation for Handling Imbalanced Datasets

In classification tasks where you have **imbalanced data** (e.g., one class is much more frequent than the other), a random train-test split might result in uneven distribution of classes in the training or test sets. **Stratified Cross-Validation** ensures that the class proportions are preserved in each fold, giving more representative and balanced training and testing sets.

> This is crucial when you have rare classes that need to be represented fairly in both the training and testing sets to avoid biased results.

6. More Reliable Performance Estimates

Cross-validation tends to give a **more reliable estimate** of how well a model will perform on unseen data because it tests the model across different train-test splits. By averaging the results, it smooths out potential irregularities caused by unlucky splits that might skew performance in a simple train-test split.

> For example, if the test set in a simple train-test split contains outliers or a biased distribution, it could negatively affect the performance estimate. Cross-validation mitigates this by using multiple test sets and averaging the results.

7. Works with Various Model Types

Cross-validation can be used with a variety of models, including:

> **Classification Models**: Models like Decision Trees, SVM, Logistic Regression, etc.
>
> **Regression Models**: Models like Linear Regression, Ridge, Lasso, etc.
>
> **Time Series Models**: Special cross-validation techniques like **Time Series Split** can be used for time-dependent data to account for the temporal ordering of data points.

8. Avoids Dependency on Random Splits

A simple train-test split might give misleading results if it depends on random or unrepresentative splits of the data. Cross-validation avoids this by training and testing the model multiple times on different subsets, giving a more balanced view of the model's performance.

> Using a **random state** in cross-validation (like in KFold(n_splits=5, shuffle=True, random_state=42)) ensures that the results are reproducible while still using different subsets of the data.

9. Helps in Detecting Data Issues

Cross-validation can help reveal **data issues** that might not be apparent with a simple train-test split. For example, if the model performs well on some folds and poorly on others, it may indicate that the dataset is not well-represented, has noise, or contains outliers.

> This allows you to take further steps, such as improving data cleaning or feature selection.

10. Supports Model Tuning and Feature Selection

Cross-validation is commonly used in combination with techniques such as **feature selection** and **model tuning** (hyperparameter tuning) to find the best model configuration. By evaluating different models or different features through cross-validation, you can find the optimal model that performs consistently well across various data splits.

Tools like **Grid Search** or **Random Search** in combination with cross-validation allow you to explore various hyperparameters or feature sets and select the ones that yield the best average performance across folds.

Disadvantages of Cross Validation:

While cross-validation is a powerful technique for model evaluation, it comes with some disadvantages and limitations. Here are the key disadvantages of using cross-validation in Python:

1. Computationally Expensive

Cross-validation involves training and evaluating the model multiple times (one for each fold or split). For instance, in K-Fold Cross-Validation with 10 folds, the model is trained and tested 10 times, which can significantly increase the computation time, especially with large datasets or complex models.

Disadvantage: For large datasets or models that take a long time to train (e.g., deep learning models), cross-validation can be very slow, requiring substantial computational resources (e.g., CPU, GPU, memory).

Example: Training a neural network 10 times with 10-fold cross-validation will take 10 times longer than a simple train-test split**.**

2. Not Suitable for Time Series Data (Without Modifications)

Disadvantage: Regular K-Fold cross-validation can lead to data leakage in time series Traditional cross-validation techniques like K-Fold or Stratified K-Fold assume that the data points are independent and identically distributed (i.i.d.). However, with time series data, this assumption is invalid because the data points are temporally dependent. A model trained on future data points and tested on past data points would be unrealistic in time series problems.

Solution: Use Time Series Split for cross-validation with time-dependent data to ensure the model is trained on past data and tested on future data.

3. May Lead to Higher Variance in Results

In some cases, the variance between folds in cross-validation can be high, especially when the dataset is small or the splits are not representative. This can lead to inconsistent model evaluation results**.**

> **Disadvantage:** High variance between the performance on different folds can make it hard to get a clear understanding of the model's performance. A model might perform very well on some folds and poorly on others, leading to confusing evaluation metrics.
>
> **Cause**: This often happens when the data is not homogeneous or has noisy features, outliers, or imbalanced class distributions that affect the folds differently.

4. Overhead in Hyperparameter Tuning

Cross-validation is commonly used with hyperparameter tuning methods like Grid Search or Random Search to find the best set of hyperparameters. However, this can introduce a significant amount of overhead.

> **Disadvantage:** Nested cross-validation, which combines cross-validation with hyperparameter tuning, can be computationally expensive. For instance, if a grid search tests 10 hyperparameter combinations and each is evaluated using 10-fold cross-validation, this results in 100 model evaluations, making it very slow.

5. Bias When Data is Too Small

When the dataset is very small, cross-validation may not provide reliable results. This is because the model might be evaluated on test sets that are too small to be representative, leading to high variance or unreliable performance estimates.

> **Disadvantage:** Cross-validation does not work well with very small datasets because the test set in each fold may not be large enough to evaluate the model fairly. In extreme cases, Leave-One-Out Cross-Validation (LOOCV) might overestimate performance because each test set has only one data point.

6. May Be Redundant for Large Datasets

For very large datasets, a simple train-test split may suffice. In such cases, cross-validation might be overkill, as the performance of the model on a large, well-represented test set will likely provide an accurate estimate of the model's generalization ability.

Disadvantage: Cross-validation might become unnecessary or redundant for very large datasets, and the computational cost might not be justified. A single train-test split may give results that are just as reliable without the added complexity and time.

7. May Not Be Helpful for Imbalanced Datasets (Without Stratification)

If you are dealing with imbalanced datasets (e.g., many more instances of one class than the other), using regular cross-validation without proper stratification can lead to folds where some classes are underrepresented or even missing. This can result in misleading performance estimates.

Disadvantage: Without Stratified Cross-Validation, the model might be trained or tested on data that is not representative of the actual class distribution, leading to biased results and inaccurate metrics.

Solution: Use Stratified K-Fold Cross-Validation to ensure the class proportions are preserved in each fold**.**

8. Difficult to Parallelize for Complex Models

For some complex machine learning models, parallelizing cross-validation can be challenging. Although libraries like scikit-learn support parallel processing (n_jobs=-1), certain models or custom pipelines might not be easy to parallelize, limiting the computational efficiency.

Disadvantage: If you can't easily parallelize cross-validation, the evaluation process becomes slower and less efficient, especially when dealing with large datasets or computationally expensive models.

9. Potential for Overfitting in Nested Cross-Validation

Nested cross-validation, often used for model selection and hyperparameter tuning, can still suffer from overfitting if the hyperparameters are tuned too aggressively to the validation set during cross-validation. This can result in a model that is highly tuned to the specific folds of cross-validation but may not generalize well to truly unseen data**.**

Disadvantage: Even though cross-validation helps reduce overfitting, nested cross-validation can sometimes overfit to the cross-validation folds, particularly when there are many hyperparameters to tune.

10. Difficult Interpretation in Ensemble Learning

In some advanced models like ensembles (e.g., stacking, bagging), interpreting cross-validation results can become more complicated, especially

when multiple models are combined and cross-validation is used multiple times within each model pipeline.

Disadvantage: The complexity of ensemble learning and nested cross-validation can make it hard to interpret individual model performances and how they contribute to the overall prediction quality.

Chapter-4 Random and Stratified Splitting

In machine learning, splitting a dataset into training and testing sets is a crucial step to evaluate a model's performance. There are two common techniques for splitting data in Python: **random splitting** and **stratified splitting**. Let's elaborate on both methods, their differences, and when to use each approach.

1. Random Splitting

Random splitting means randomly dividing the dataset into training and testing subsets without considering the distribution of target labels or any specific structure in the data.

Example:

You randomly select a percentage (e.g., 80%) of the data to use for training, and the rest (e.g., 20%) for testing. This approach is common when you have a sufficiently large and balanced dataset.

How to Perform Random Splitting in Python:

In Python, you can perform random splitting using the train_test_split function from the **scikit-learn** library.

python

```
from sklearn.model_selection import train_test_split

# X is your features, y is your labels
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Here's what's happening:

X: Feature matrix (independent variables).

y: Target labels (dependent variable).

test_size=0.2: Specifies that 20% of the data should be used for testing.

random_state=42: This is used to ensure that the random split is reproducible (every time you run the code with the same seed, you get the same result).

Pros of Random Splitting:

1. **Simple**: Easy to implement.
2. **Fast**: Quick to execute, especially for large datasets.
3. **Reproducible**: Using a random_state ensures you get the same split every time for comparison purposes.

Cons of Random Splitting:

1. **Class Imbalance**: If your dataset has imbalanced classes (e.g., more instances of one class than another), random splitting can lead to uneven distribution of classes in the training and test sets.
2. **Unrepresentative Split**: Random splitting may result in unrepresentative training or testing sets, especially in small datasets. For example, it might randomly select more high-valued examples for testing and low-valued ones for training.

When to Use Random Splitting:

When the dataset is large and representative of the problem.

When the dataset has a balanced class distribution.

When class proportions are not a concern.

2. Stratified Splitting

Stratified splitting ensures that the distribution of target labels (or some other feature of interest) is consistent across both training and test sets. This is particularly useful for classification tasks with **imbalanced classes**.

Example:

In a classification problem where you have 90% of instances in class 0 and 10% in class 1, random splitting might result in an uneven distribution of classes between the training and test sets. **Stratified splitting** ensures that both training and testing sets maintain the same ratio of class 0 to class 1 as in the original dataset.

How to Perform Stratified Splitting in Python:

Stratified splitting is done using the train_test_split function, with the stratify parameter set to the target variable (y).

```
python

from sklearn.model_selection import train_test_split


# X is your features, y is your labels

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
```

Here's what's happening:

stratify=y: Ensures that the train and test sets will have the same proportion of target labels as the original dataset.

Pros of Stratified Splitting:

1. **Preserves Class Distribution**: Ensures that both the training and test sets have the same distribution of labels as the original dataset.
2. **Handles Imbalanced Datasets**: Helps avoid skewed splits, especially in classification problems with imbalanced datasets.
3. **More Representative Splits**: By ensuring equal distribution, it provides a more accurate evaluation of the model's generalization performance.

Cons of Stratified Splitting:

1. **Slightly Slower**: Adds some overhead compared to simple random splitting because it needs to calculate and preserve the distribution.
2. **Only Works on Certain Targets**: Stratification works on categorical or discrete targets (e.g., classification problems). It is less effective or less meaningful for continuous variables in regression tasks.

When to Use Stratified Splitting:

When your dataset is **imbalanced** (e.g., many more examples of one class than another).

When maintaining the **class distribution** in both training and test sets is important.

In **classification tasks** where you want the test set to be representative of the full data's label distribution.

Comparing Random and Stratified Splitting

Feature	Random Splitting	Stratified Splitting
Class Distribution	May not preserve class distribution	Preserves the class distribution
Best for	Balanced datasets	Imbalanced datasets
Use in Classification Problems	Can lead to uneven class distribution	Ensures class distribution is maintained
Use in Regression Problems	Works fine for most cases	Less applicable, since stratification is based on classes
Speed	Faster	Slightly slower due to stratification

Practical Example: Random vs. Stratified Splitting

1. Random Splitting Example:

python

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Create a dataset with imbalanced classes
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.9, 0.1], random_state=42)

# Perform random splitting
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Check the class distribution in the training and test sets
```

```
print(f"Training set class distribution: {sum(y_train == 1) / len(y_train):.2%}")

print(f"Test set class distribution: {sum(y_test == 1) / len(y_test):.2%}")
```

2. Stratified Splitting Example:

python

```
# Perform stratified splitting

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)


# Check the class distribution in the training and test sets

print(f"Training set class distribution (stratified): {sum(y_train == 1) / len(y_train):.2%}")

print(f"Test set class distribution (stratified): {sum(y_test == 1) / len(y_test):.2%}")
```

Output of Random Splitting:

Training set class distribution: 8.88%

Test set class distribution: 13.50%

Output of Stratified Splitting:

Training set class distribution (stratified): 10.00%

Test set class distribution (stratified): 10.00%

In the stratified example, the training and test sets maintain the original distribution of the target classes (10%), whereas random splitting results in an uneven split, with a higher percentage of the minority class in the test set.

Chapter 5: Group Based Splitting

Group-based splitting (or grouped splitting) in Python is a technique used when you have data organized in groups or

clusters, and you want to ensure that all data points from a specific group appear in either the training set or the test set, but not in both. This is particularly useful when you have grouped data (e.g., customers, patients, subjects) and you want to avoid information leakage by keeping the groups consistent across splits.

Why Use Group-Based Splitting?

When the data points within a group are highly correlated or not independent, splitting them randomly could lead to data leakage. For example, if some data points from the same group appear in both the training and test sets, the model may perform better in testing because it has already seen similar data during training.

Group-based splitting is crucial in cases such as:

Medical studies where each patient has multiple measurements.

Repeated experiments on the same subject.

Sales data where transactions are grouped by customer.

Example Scenarios

1. **Medical Data:** If you have patient data where each patient has multiple visits, you should split by patient so that a patient's visits either all appear in the training set or all in the test set.
2. **Customer Transactions:** If a dataset contains transactions from various customers, you might want to ensure that all transactions from a given customer are grouped together (either in training or testing).

How to Perform Group-Based Splitting in Python

In Python, you can perform group-based splitting using the GroupKFold or GroupShuffleSplit classes from scikit-learn.

1. GroupKFold Cross-Validation

GroupKFold is used for performing cross-validation where the splits are made based on a group identifier (e.g., patient ID, customer ID). It ensures that no data points from the same group are used in both the training and testing sets in any split**.**

Example:

python

```
from sklearn.model_selection import GroupKFold
import numpy as np

# Example dataset
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
y = np.array([0, 0, 1, 1, 0, 1])  # Labels
groups = np.array([1, 1, 2, 2, 3, 3])  # Group identifiers (e.g., patient IDs)

# Initialize GroupKFold
gkf = GroupKFold(n_splits=3)

# Perform the split
for train_index, test_index in gkf.split(X, y, groups):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    print(f"X_train:\n{X_train}\nX_test:\n{X_test}\n")
```

In this example:

X: Features.

y: Target labels.

groups: Group identifiers (e.g., patient IDs or customer IDs).

The data is split into 3 folds, and each fold ensures that all data points belonging to the same group are either in the training set or in the test set, but not both.

Output:

lua

```
TRAIN: [2 3 4 5] TEST: [0 1]
X_train:
[[ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]
X_test:
[[1 2]
 [3 4]]

TRAIN: [0 1 4 5] TEST: [2 3]
X_train:
[[ 1  2]
 [ 3  4]
 [ 9 10]
 [11 12]]
X_test:
[[5 6]
 [7 8]]

TRAIN: [0 1 2 3] TEST: [4 5]
X_train:
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

X_test:

[[9 10]

 [11 12]]

In this example, the data points from each group (denoted by groups) appear either in the training set or the test set, but never in both.

2. GroupShuffleSplit

If you want to perform a single train-test split with grouped data, you can use GroupShuffleSplit, which allows you to randomly shuffle groups and split them into training and testing sets while keeping the groups intact.

Example:

python

```
from sklearn.model_selection import GroupShuffleSplit

# Example dataset
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
y = np.array([0, 0, 1, 1, 0, 1])
groups = np.array([1, 1, 2, 2, 3, 3])  # Group identifiers

# Initialize GroupShuffleSplit
gss = GroupShuffleSplit(test_size=0.5, n_splits=1, random_state=42)

# Perform the split
for train_index, test_index in gss.split(X, y, groups):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

```
print(f"X_train:\n{X_train}\nX_test:\n{X_test}\n")
```

Output:

lua

```
TRAIN: [0 1 4 5] TEST: [2 3]
X_train:
[[ 1  2]
 [ 3  4]
 [ 9 10]
 [11 12]]
X_test:
[[5 6]
 [7 8]]
```

Here:

The groups with IDs [1, 3] are used in the training set.

The group with ID [2] is used in the test set.

The group splitting is random but ensures that all data points in a group are either in the training or test set, not both.

Advantages of Group-Based Splitting

1. Avoids Data Leakage: Ensures that the model does not see part of a group during training and another part during testing, preventing information leakage.
2. Respects Group Integrity: Useful when the data within a group is correlated, and splitting within a group could lead to overestimating model performance.
3. More Realistic Evaluation: For cases where entire groups will be unseen during real-world testing (e.g., new customers, new patients), group-based splitting gives a better estimate of how the model will perform on unseen data.

When to Use Group-Based Splitting

Medical Studies: If multiple samples belong to the same patient, you should ensure that all samples from a single patient are in either the training or testing set.

Customer Segmentation: If you have customer purchase data, and multiple purchases are associated with the same customer, group by customer ID to avoid having one customer's data in both the training and test sets.

Time Series with Events: If you are working with data that is organized around specific events or time blocks (e.g., transactions or measurements grouped by day or event), group-based splitting can ensure that entire events are either in training or testing sets.

Summary: Comparing Splitting Methods

Splitting Method	Description	Use Case
Random Splitting	Randomly splits data into training and testing sets.	For independent and identically distributed (i.i.d.) data.
Stratified Splitting	Ensures balanced class distribution between splits.	For imbalanced classification problems.
Group-Based Splitting	Ensures that all data points from a group appear in either the training or test set.	For grouped or clustered data (e.g., patient or customer data).

Group-based splitting ensures that your model generalizes well to new, unseen groups and prevents artificially inflated performance metrics due to group data leakage. It's particularly important for problems where group integrity matters.

Conclusion:

Data splitting is a fundamental step in machine learning that allows you to train and evaluate models using different subsets of a dataset.

By splitting the data, you can estimate how well the model generalizes to unseen data, helping to avoid overfitting and underfitting.

In Python, several splitting techniques are available, each designed for specific scenarios, and choosing the right method depends on the nature of the dataset and the problem at hand.

Key Data Splitting Techniques:

1. **Random Splitting:**

   Description: This is the most basic form of data splitting, where the data is randomly divided into training and testing sets.

   Usage: It works well for large datasets where the classes or outcomes are evenly distributed**.**

   Limitations: Random splitting might lead to unbalanced class distributions in small or imbalanced datasets, which can cause biased performance evaluations**.**

2. **Stratified Splitting:**

   Description: Ensures that the proportion of classes (target labels) is the same in both the training and test sets.

   Usage: Primarily used in classification problems, especially when the dataset is imbalanced, ensuring that minority classes are properly represented in both training and testing data.

   Benefits: Prevents skewed splits and gives a better representation of model performance, particularly for imbalanced data.

   Limitations: Only useful for classification problems; not ideal for regression or continuous targets.

3. **Cross-Validation:**

   Description: A more advanced technique where the data is split into multiple folds, and the model is trained and tested on each fold.

   Usage: Provides a more robust evaluation of model performance by ensuring that every data point is used for both training and testing across different iterations.

Benefits: Helps mitigate variance in model performance due to a single train-test split, especially useful for small datasets.

Limitations: Computationally expensive, especially for large datasets.

4. **Group-Based Splitting:**

   Description: Ensures that data points belonging to the same group (e.g., patient, customer, or event) are either entirely in the training set or entirely in the testing set**.**

   Usage: Critical for grouped or correlated data (e.g., medical studies, customer behavior). Prevents leakage where information from a group might appear in both training and testing sets, which could artificially inflate performance.

   Benefits: Maintains group integrity and gives a realistic evaluation for grouped or clustered data.

   Limitations: Adds complexity and can be slightly slower to implement due to the need for managing group assignments.

Choosing the Right Splitting Method

The choice of a splitting technique depends on various factors, including the size of the dataset, class distribution, and group structure. Here's a general guide:

1. **Large and Balanced Datasets:** Random splitting works well as long as the dataset is balanced, and there's no risk of data leakage**.**
2. **Imbalanced Classification Problems:** Stratified splitting ensures the class proportions are maintained, preventing biased model performance.
3. **Small Datasets or Need for Robust Evaluation**: Cross-validation is ideal for ensuring that every data point is used for both training and testing, reducing the likelihood of overfitting or underfitting.
4. **Grouped Data (e.g., Customers, Patients):** Group-based splitting is essential for data where there are natural groupings or correlations to ensure accurate performance estimation.

Best Practices for Data Splitting

Test Set Proportion: A typical split is 70%-80% for training and 20%-30% for testing. However, this ratio can change depending on dataset size and complexity**.**

Validation Set: Besides the training and testing sets, it's common to have a separate validation set for tuning model hyperparameters before final evaluation.

Cross-Validation for Small Datasets: For smaller datasets, cross-validation can help evaluate models on more data points, preventing overfitting to a single split.

Avoid Data Leakage: Whether through random splitting, cross-validation, or group-based splitting, ensure that information from the test set does not leak into the training process. This is particularly crucial for time series data and group-based data, where future or related data could be mistakenly included in training.

Common Pitfalls to Avoid

1. **Data Leakage:** Ensure that your test set contains entirely unseen data to prevent artificially inflated performance**.**
2. **Class Imbalance Ignorance:** When dealing with imbalanced datasets, always use stratified splitting or resampling methods to avoid misleading performance results.
3. **Overfitting to Test Set**: Avoid using the test set multiple times during model development, as this may lead to overfitting the test set, resulting in overly optimistic evaluations.
4. **Ignoring Temporal or Group Structures:** If your data is grouped (e.g., by patient, customer, or time), ignoring this structure can lead to poor generalization in real-world applications.

Final Thoughts:

Data splitting in Python is not just about dividing data; it’s about making informed decisions to ensure that your model evaluation is accurate and generalizable. Choosing the right splitting technique (random, stratified, cross-validation, or group-based) depends on the problem you're solving, the characteristics of your data, and the risks of data leakage or bias. Applying

these best practices ensures that you build machine learning models that perform well on unseen data, making them robust and reliable in production environments.