Detailed Guide on Data Splitting for Training ML Models

Type	@datasciencebrain

Data Splitting in Machine Learning

Data splitting is a critical procedure in Machine Learning (ML) used to divide the available dataset into subsets for training, validating, and evaluating machine learning models. The main purpose of splitting data is to effectively assess the model's generalization capability on unseen data.

The choice of splitting strategy significantly impacts model reliability and accuracy. Improper splitting may lead to biased evaluation, overly optimistic or pessimistic performance estimates, or poor generalization in production environments.

Different Approaches to Data Splitting

The method for splitting data depends on the problem complexity, data volume, and evaluation requirements. Common splitting methods include:

1. Two-way Split (Train-Test Split)

This is the simplest and most common splitting approach.

Purpose:

- To train the model using one subset and evaluate it using another independent subset.

Typical proportions:

- Training Set: Typically 70%–80% of data
- Testing Set: Typically 20%–30% of data

Advantages:

- Simple and easy to implement.
- Quick model evaluation.

Disadvantages:

- No dedicated validation set for hyperparameter tuning or model selection.
- Risk of biased evaluation if random splitting is not properly done.

```
from sklearn.model_selection import train_test_split
import pandas as pd

# Sample dataset (replace this with your actual data)
data = {'feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        'feature2': [11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
        'target': [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]}

# Convert the data into a DataFrame
df = pd.DataFrame(data)

# Split the data into features (X) and target (y)
X = df.drop('target', axis=1)
y = df['target']

# Perform a 2-way train-test split (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

# Display the train and test sets
print("Training Features (X_train):\n", X_train)
print("\nTesting Features (X_test):\n", X_test)
print("\nTraining Target (y_train):\n", y_train)
print("\nTesting Target (y_test):\n", y_test)
```

Explanation:

1. `train_test_split`: Splits the dataset into training and testing sets.

- `X`: Features (independent variables).
- `y`: Target (dependent variable).
- `test_size=0.2`: 20% of the data will be used for testing, and 80% will be used for training.
- `random_state=42`: Ensures reproducibility of the random split.

---

2. Three-way Split (Train-Validation-Test Split)

This method introduces a validation set.

Purpose:

- Training set: used for model training.
- Validation set: used for hyperparameter tuning, model selection, and preventing overfitting during training.
- Testing set: used for the final unbiased evaluation of model performance.

Typical proportions:

- Training Set: Approximately 60%–70%
- Validation Set: Approximately 10%–20%
- Testing Set: Approximately 15%–25%

Advantages:

- Allows rigorous model tuning and hyperparameter optimization.
- Provides an unbiased final performance estimate.

Disadvantages:

- Reduces the amount of data available for training, which might be problematic for smaller datasets.

Here's a Python code example for performing a three-way split (train-validation-test split) using the `train_test_split` function twice from `sklearn.model_selection`:

```
from sklearn.model_selection import train_test_split
import pandas as pd
```

```
# Sample dataset (replace this with your actual data)
data = {'feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        'feature2': [11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
        'target': [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]}

# Convert the data into a DataFrame
df = pd.DataFrame(data)

# Split the data into features (X) and target (y)
X = df.drop('target', axis=1)
y = df['target']

# Perform a first split (80% for training, 20% for testing)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2, random_s
tate=42)

# Perform a second split (from the remaining 20%, split it into 10% for validati
on and 10% for testing)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, ran
dom_state=42)

# Display the train, validation, and test sets
print("Training Features (X_train):\n", X_train)
print("\nValidation Features (X_val):\n", X_val)
print("\nTesting Features (X_test):\n", X_test)
print("\nTraining Target (y_train):\n", y_train)
print("\nValidation Target (y_val):\n", y_val)
print("\nTesting Target (y_test):\n", y_test)
```

Explanation:

1. **First Split (80-20)**: The data is first split into 80% for training and 20% for a temporary split (validation and testing).

2. **Second Split (50-50 from 20%)**: The temporary 20% is further split into 10% for validation and 10% for testing. This ensures you have distinct training, validation, and testing datasets.

Final proportions:

- **Training Set**: 80% of the data
- **Validation Set**: 10% of the data
- **Testing Set**: 10% of the data

---

3. Four-way Split (Train, Train-Test, Validation, Test Split)

This more advanced method is used for extensive experimentation.

Purpose:

- Training set: used for initial model training.
- Train-Test set (also called development set): used for preliminary testing, experimentation, and selecting promising models or approaches.
- Validation set: used strictly for hyperparameter optimization and model selection.
- Test set: reserved exclusively for final model evaluation.

Typical proportions:

- Training Set: Approximately 50%–60%
- Train-Test Set: Approximately 10%–15%
- Validation Set: Approximately 10%–15%
- Test Set: Approximately 10%–20%

Advantages:

- Allows detailed experimentation without influencing the unbiased testing stage.
- Helps avoid multiple-testing biases and overly optimistic estimates.

Disadvantages:

- Requires larger datasets due to data partitioning.
- Complexity increases with multiple data subsets.

Here's a Python code example for performing a four-way split (train, train-test, validation, test split) using `train_test_split` multiple times from `sklearn.model_selection`:

```
from sklearn.model_selection import train_test_split
import pandas as pd

# Sample dataset (replace this with your actual data)
data = {'feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        'feature2': [11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
        'target': [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]}

# Convert the data into a DataFrame
df = pd.DataFrame(data)

# Split the data into features (X) and target (y)
X = df.drop('target', axis=1)
y = df['target']

# Perform the first split (60% for training, 40% for the remaining set)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_s
tate=42)

# Perform the second split (50% for train-test, 50% for validation and testing f
rom the remaining 40%)
X_train_test, X_val_test, y_train_test, y_val_test = train_test_split(X_temp, y_tem
p, test_size=0.5, random_state=42)

# Perform the third split (50% for validation, 50% for testing from the remainin
g 50%)
X_val, X_test, y_val, y_test = train_test_split(X_val_test, y_val_test, test_size=0.5,
random_state=42)
```

```
# Display the sets
print("Training Features (X_train):\n", X_train)
print("\nTrain-Test Features (X_train_test):\n", X_train_test)
print("\nValidation Features (X_val):\n", X_val)
print("\nTesting Features (X_test):\n", X_test)
print("\nTraining Target (y_train):\n", y_train)
print("\nTrain-Test Target (y_train_test):\n", y_train_test)
print("\nValidation Target (y_val):\n", y_val)
print("\nTesting Target (y_test):\n", y_test)
```

Explanation:

1. **First Split (60-40)**: The data is first split into 60% for training and 40% for the remaining subset.
2. **Second Split (50-50 from 40%)**: The 40% is further split into 50% for a "train-test" set (used for experimentation) and 50% for a combined validation and test set.
3. **Third Split (50-50 from 50%)**: The remaining 50% (validation and testing) is split into 50% for validation and 50% for testing.

Final proportions:

- **Training Set**: 60% of the data
- **Train-Test Set**: 20% of the data (used for model testing during experimentation)
- **Validation Set**: 10% of the data
- **Test Set**: 10% of the data

This approach allows for extensive experimentation while keeping the final model evaluation unbiased.

Techniques for Data Splitting

1. Random Splitting

- Data is randomly partitioned into subsets.
- Suitable when the dataset is sufficiently large and well-balanced.
- Risk of imbalance if dataset is small or highly skewed.

Here's an example of performing **random splitting** of a dataset using `train_test_split` from `sklearn.model_selection`:

```
from sklearn.model_selection import train_test_split
import pandas as pd

# Sample dataset (replace this with your actual data)
data = {'feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        'feature2': [11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
        'target': [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]}

# Convert the data into a DataFrame
df = pd.DataFrame(data)

# Split the data into features (X) and target (y)
X = df.drop('target', axis=1)
y = df['target']

# Perform random splitting (70% for training, 30% for testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_stat
e=42, shuffle=True)

# Display the train and test sets
print("Training Features (X_train):\n", X_train)
print("\nTesting Features (X_test):\n", X_test)
print("\nTraining Target (y_train):\n", y_train)
print("\nTesting Target (y_test):\n", y_test)
```

Explanation:

1. `train_test_split` **with** `shuffle=True`: This performs random splitting, shuffling the data before splitting it into training and testing sets.
2. `test_size=0.3`: Specifies that 30% of the data will be used for testing, leaving the remaining 70% for training.
3. `random_state=42`: This ensures that the split is reproducible, so you get the same result every time the code is run.

Advantages:

- Simple to implement and works well when the dataset is large and balanced.
- Useful for creating random splits without introducing any bias.

Disadvantages:

- If the dataset is small or imbalanced, random splitting may lead to an uneven distribution of classes between the train and test sets. This can affect model performance, particularly in classification tasks.

2. Stratified Splitting

- Ensures each subset maintains a similar distribution of a target variable or key categorical features.
- Particularly important for classification problems with imbalanced classes.
- Significantly improves representativeness of subsets.

Here's an example of **stratified splitting** using the `train_test_split` function from `sklearn.model_selection`, ensuring that each subset maintains a similar distribution of the target variable:

```
from sklearn.model_selection import train_test_split
import pandas as pd

# Sample dataset (replace this with your actual data)
data = {'feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        'feature2': [11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
        'target': [0, 1, 0, 1, 0, 0, 0, 1, 1, 0]}  # Imbalanced target variable
```

```
# Convert the data into a DataFrame
df = pd.DataFrame(data)

# Split the data into features (X) and target (y)
X = df.drop('target', axis=1)
y = df['target']

# Perform stratified splitting (70% for training, 30% for testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_stat
e=42, stratify=y)

# Display the train and test sets
print("Training Features (X_train):\n", X_train)
print("\nTesting Features (X_test):\n", X_test)
print("\nTraining Target (y_train):\n", y_train)
print("\nTesting Target (y_test):\n", y_test)

# Check the distribution of the target variable in both sets
print("\nTarget Distribution in Training Set:\n", y_train.value_counts())
print("\nTarget Distribution in Testing Set:\n", y_test.value_counts())
```

Explanation:

1. `stratify=y`: Ensures that the target variable `y` is evenly distributed in both the training and testing sets. This is particularly important in classification tasks where the target variable may be imbalanced (e.g., more 0s than 1s).
2. `train_test_split`: The function splits the data, and the `stratify=y` argument ensures the split maintains the same proportion of classes in both the training and testing sets.

Output:

You should see that both the training and testing sets have a similar distribution of the target variable (e.g., if your target has 70% class 0 and 30% class 1, the split will maintain these proportions in both sets).

Advantages:

- **Improved representativeness**: Each subset will have a similar distribution of the target variable, reducing the bias introduced by an imbalanced dataset.
- **Better performance**: Stratified splitting is crucial for classification tasks where class imbalance can significantly affect the model's ability to learn from minority classes.

Disadvantages:

- **Less effective with small datasets**: Even with stratification, if the dataset is very small, the subsets may still not be sufficiently representative.

3. Time-based Splitting (Temporal Splitting)

- Data is split chronologically.
- Necessary for time series problems or scenarios where future data must be predicted using historical data.
- Prevents data leakage and ensures realistic evaluation.

Here's an example of **time-based splitting** (temporal splitting) where the data is split chronologically, typically used for time series problems:

```
import pandas as pd

# Sample dataset (replace this with your actual time-series data)
data = {'timestamp': pd.date_range(start='2023-01-01', periods=10, freq='D'),
        'feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        'target': [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]}

# Convert the data into a DataFrame
df = pd.DataFrame(data)

# Set the timestamp column as the index
df.set_index('timestamp', inplace=True)

# Perform time-based splitting (80% for training, 20% for testing)
```

```
train_size = int(len(df) * 0.8)
train_data = df[:train_size]  # First 80% for training
test_data = df[train_size:]   # Remaining 20% for testing

# Display the training and testing sets
print("Training Data:\n", train_data)
print("\nTesting Data:\n", test_data)
```

Explanation:

1. **Time-based Splitting**: The data is split by taking the first 80% of the data for training and the remaining 20% for testing. This ensures that training data contains only past information, and the model is tested on future data, which is the correct approach for time series forecasting.
2. `timestamp` **as index**: The data is indexed by the timestamp column to simulate time-based splitting. The dataset is sorted in chronological order to prevent future data from being used in the training set.

Key Points:

- **Training Set**: Contains past data (up to a certain point in time).
- **Testing Set**: Contains future data, which simulates the real-world scenario where the model is used for future predictions.

Advantages:

- **Prevents data leakage**: By splitting chronologically, we ensure that no future data is used in training, which is crucial for preventing unrealistic evaluations.
- **Realistic evaluation**: It mirrors how real-world systems are deployed, where predictions are made on future data.

Disadvantages:

- **Limited flexibility**: This method is specific to time series data and may not be applicable for all types of datasets.

- **Not suitable for random cross-validation**: Since the data is not randomly shuffled, the model might miss certain patterns that could have been captured with a random split.

Time-based splitting is essential in time series forecasting, where historical data is used to predict future values, and it ensures that the evaluation process mirrors how the model would behave in a real-world setting.

4. Group-based Splitting

- Data is split by groups or categories, ensuring members of the same group do not appear across multiple subsets.
- Critical when data points within a group are correlated or closely related (e.g., user interactions, repeated measurements from a single individual).

Here's an example of **group-based splitting**, where data is split based on a group identifier, ensuring that all data points within the same group remain in the same subset (i.e., no data leakage between training and test sets):

```
from sklearn.model_selection import GroupShuffleSplit
import pandas as pd

# Sample dataset (replace this with your actual data)
data = {'feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        'feature2': [11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
        'target': [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        'group': ['A', 'A', 'B', 'B', 'C', 'C', 'D', 'D', 'E', 'E']}  # Group identifier

# Convert the data into a DataFrame
df = pd.DataFrame(data)

# Define features (X) and target (y)
X = df.drop(['target', 'group'], axis=1)
y = df['target']
groups = df['group']  # Group identifiers

# Perform group-based splitting (80% for training, 20% for testing)
```

```
group_splitter = GroupShuffleSplit(test_size=0.2, random_state=42)
train_indices, test_indices = next(group_splitter.split(X, y, groups=groups))

# Create training and testing datasets
X_train, X_test = X.iloc[train_indices], X.iloc[test_indices]
y_train, y_test = y.iloc[train_indices], y.iloc[test_indices]

# Display the train and test sets
print("Training Features (X_train):\n", X_train)
print("\nTesting Features (X_test):\n", X_test)
print("\nTraining Target (y_train):\n", y_train)
print("\nTesting Target (y_test):\n", y_test)

# Check the groups in training and testing sets
print("\nGroups in Training Set:", groups.iloc[train_indices].unique())
print("\nGroups in Testing Set:", groups.iloc[test_indices].unique())
```

Explanation:

1. `GroupShuffleSplit`: This method splits the data into training and test sets based on group identifiers, ensuring that no group is split between the training and testing sets.
2. **Groups**: The `group` column identifies the different groups (e.g., `A`, `B`, `C`, etc.). The `group_splitter.split()` function ensures that all data points from the same group are either in the training set or in the test set, but not both.
3. **Train and Test Sets**: The training and testing sets are created based on the indices returned by the `GroupShuffleSplit`.

Key Points:

- **Grouping**: It ensures that all data points from the same group (e.g., a user, patient, or time series measurement from a single entity) are kept together in either the training or test set.
- **Preventing data leakage**: Ensures that the model doesn't learn from overlapping data points or correlations that might exist within the same group,

which could lead to unrealistic evaluation results.

Advantages:

- **Ensures realistic training and testing**: Group-based splitting is crucial when data points within a group are correlated (e.g., repeated measurements for a single individual), as splitting data by random indices could lead to data leakage and unrealistic evaluation.
- **Avoids overfitting**: Helps ensure that the model does not overfit to particular groups.

Disadvantages:

- **May reduce training data**: Depending on the group sizes and the number of groups, you may end up with fewer training data points, as entire groups are excluded from the test set.
- **Group size imbalance**: If the groups vary significantly in size, it can lead to unbalanced training or test sets.

This method is essential when dealing with data where the observations within a group are not independent, such as repeated measures from the same individual, user interactions, or time series data where points from the same entity must not appear in both the training and test sets.

Critical Factors and Considerations in Data Splitting

When splitting data, it's important to maintain the integrity and representativeness of each subset. Below are essential considerations:

1. Ensuring Feature Representation (Stratification)

- Stratification ensures each split has proportional representation of different classes or categories of a critical feature.
- For example, in a classification dataset with highly imbalanced classes (e.g., 95% class A and 5% class B), random splitting might create subsets with poor

representation of class B. Stratified splitting helps address this issue.

2. Preventing Data Leakage

- Data leakage occurs when information from the test or validation sets inadvertently influences model training.
- For example, in temporal datasets, random splitting can create scenarios where future data (from the test set) indirectly influences the training, leading to unrealistic model performance estimates.
- Proper splitting techniques (e.g., temporal or group-based splits) avoid this issue.

3. Dataset Size and Allocation Proportions

- The proportion of data used for training, validation, and testing may vary depending on dataset size:
  - **Small datasets:** Use cross-validation or fewer splits to maximize training data.
  - **Large datasets:** Multiple splits can provide more rigorous validation and testing.

4. Cross-Validation as an Alternative

- When datasets are limited in size, k-fold cross-validation is recommended as an alternative approach.
- Cross-validation partitions data into k subsets (folds), training and testing iteratively, which helps achieve stable performance estimates without losing substantial data for evaluation.

Here's an example of **k-fold cross-validation** using `KFold` from `sklearn.model_selection`:

```
from sklearn.model_selection import KFold
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

```
# Sample dataset (replace this with your actual data)
data = {'feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        'feature2': [11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
        'target': [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]}  # Binary target variable

# Convert the data into a DataFrame
df = pd.DataFrame(data)

# Define features (X) and target (y)
X = df.drop('target', axis=1)
y = df['target']

# Initialize the k-fold cross-validation object (5 folds in this case)
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Initialize the model
model = LogisticRegression()

# List to store accuracy scores for each fold
accuracy_scores = []

# Perform k-fold cross-validation
for train_index, test_index in kf.split(X):
    # Split the data into training and testing sets
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

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

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

    # Calculate accuracy for the current fold
    accuracy = accuracy_score(y_test, y_pred)
    accuracy_scores.append(accuracy)
```

```
# Calculate the average accuracy across all folds
average_accuracy = sum(accuracy_scores) / len(accuracy_scores)

# Display the results
print("Accuracy scores for each fold:", accuracy_scores)
print("Average accuracy:", average_accuracy)
```

Explanation:

1. `KFold`: The `KFold` class splits the dataset into `k` subsets (or folds). In this example, we use `k=5`, meaning the data is split into 5 folds. Each fold is used once as a test set, while the remaining `k-1` folds are used for training.
2. `kf.split(X)`: This generates indices for each fold that can be used to split the data into training and testing sets iteratively.
3. **Model Training and Evaluation**: For each fold, the model is trained on the training set (`X_train`, `y_train`) and evaluated on the test set (`X_test`, `y_test`). Accuracy is computed using the `accuracy_score` metric.
4. `average_accuracy`: The average accuracy across all the folds is calculated and displayed.

Key Points:

- **No data loss**: Each data point gets to be in both the training and test set exactly once, making this method particularly useful when data is limited.
- **Multiple splits**: The model is trained and tested multiple times, providing a more robust estimate of its performance.

Advantages:

- **Improved performance estimates**: Since the model is tested on multiple different splits, cross-validation gives a more reliable estimate of model performance, especially on smaller datasets.
- **Reduced variance**: The performance metrics from each fold can be averaged, helping reduce the impact of outliers or variations in a single training-test split.

- **Uses all data**: Each data point is used for both training and testing, allowing the model to learn from all available data without losing any for evaluation.

Disadvantages:

- **Computational cost**: Cross-validation requires training and testing the model multiple times (once per fold), which can be computationally expensive, especially for large datasets or complex models.
- **Time-consuming**: Since the model is trained multiple times, cross-validation can take longer to run compared to a simple train-test split.

When to Use:

- **Small datasets**: Cross-validation is especially helpful when you have a limited amount of data, as it maximizes the usage of available data for both training and testing.
- **Reliable model evaluation**: When you need a stable and reliable performance estimate, especially if you plan to compare different models or fine-tune hyperparameters.

Best Practices for Effective Data Splitting

1. **Always shuffle data (unless it is temporal):**
    - Shuffling ensures randomized and unbiased partitioning.
    - Exception: time-series datasets, where temporal order matters.
2. **Use stratification in classification tasks:**
    - Helps maintain class distribution consistency across splits.
3. **Maintain independence between subsets:**
    - Prevent data leakage by clearly separating groups, users, or temporal boundaries.
4. **Choose splitting strategy based on problem domain:**
    - Temporal data → Time-based splitting.

- Imbalanced classification → Stratified splitting.
- Correlated groups (users, patients, sensors) → Group-based splitting.

5. **Perform careful validation of splits:**
   - Verify distributions (mean, variance, category proportions) across subsets.
   - Ensure no subset deviates significantly from others.
6. **Ensure reproducibility:**
   - Fix random seeds during splitting to enable reproducible experimentation and debugging.

Data Splitting Tips and Tricks

1. Shuffling is Key (But Not Always)

- **When to shuffle**: When you have random and independent observations (non-sequential data), shuffling helps ensure a fair split between training and testing sets.
- **When NOT to shuffle**: For time-series data or cases where temporal relationships or groupings matter (e.g., medical or financial data), **never shuffle**. Shuffling can cause **data leakage**, where future data points unintentionally influence past data.

2. Stratification for Class Imbalance

- **Use stratified splits** when your dataset has imbalanced classes, especially in classification tasks. Stratification ensures each class is represented in the training and test sets in the same proportion as it appears in the original dataset.
- **For Example**: In a dataset where 95% of data points belong to class A and 5% to class B, without stratification, your training set could end up with too few samples of class B, leading to biased models.

Code Example:

```
from sklearn.model_selection import train_test_split

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

3. Use K-Fold Cross-Validation for Small Datasets

- When working with small datasets, **k-fold cross-validation** can be a powerful alternative to the traditional train-test split. It helps ensure that the model is trained and tested multiple times on different parts of the dataset, resulting in more reliable performance metrics.
- **Tip**: Choose `k` wisely. A larger `k` (like 10-fold) works well for smaller datasets, but in large datasets, using 5-fold might be more efficient and faster.

Code Example:

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

model = RandomForestClassifier(random_state=42)
cv_scores = cross_val_score(model, X, y, cv=5)
print(f"Cross-validation scores: {cv_scores}")
```

4. Maintain a Separate Validation Set for Hyperparameter Tuning

- **Never use the test set** for hyperparameter tuning, as this can lead to data leakage. Instead, use a **validation set** to fine-tune model parameters. The test set should remain untouched until the final evaluation.
- **Best Practice**: If you split the data into **train-validation-test**, avoid using the validation set for any other purpose (e.g., model selection or additional training).

5. For Large Datasets, Experiment with Proportional Splitting

- When you have a **huge amount of data** (e.g., millions of data points), it's often unnecessary to use a very high proportion for training (like 80%-90% of the

total data).

- **Reasoning**: With large datasets, even small amounts of data (e.g., 10%-20%) for testing can still provide robust performance metrics.
- **Tip**: For extremely large datasets, using smaller test sets (like 5%-10%) can provide a good balance between training and evaluation data. You may even use **cross-validation** with fewer folds or randomly sample subsets of the data for model training and testing.

Code Example (using a smaller test size for large datasets):

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.05, random_state=42)
```

Reasoning: As the dataset size grows, even a small fraction of data for testing can yield highly reliable results. The larger training dataset allows the model to capture more patterns, and the smaller test set is still representative enough for validation.

Why the Ratio of Splits Changes with Large Datasets

When working with huge datasets (e.g., millions of data points), the ratio of train-test or train-validation-test splits can be adjusted for better efficiency and accuracy. Here's why:

1. Large Amounts of Data Make Small Test Sets Sufficient

- **Reason**: With large datasets, the model can still be trained effectively on a smaller subset of the data while still being evaluated reliably with a smaller test set. For example, with a dataset of millions of points, using **90% of data for training** and **10% for testing** is still sufficient for an unbiased performance evaluation.
- **Effect**: With smaller datasets, more data is needed for testing to ensure reliable results. However, with larger datasets, the model can generalize well even with a smaller test set.

2. Training Models Becomes Expensive and Time-Consuming

- **Reason**: With larger datasets, training models becomes computationally expensive. Using too large a training set may increase training time unnecessarily, especially for complex models or large deep learning networks.
- **Effect**: Smaller training data (e.g., 60%–70%) might still be enough for training, with additional cross-validation helping to fine-tune the model.

3. Overfitting Concerns Are Less with Larger Datasets

- **Reason**: Overfitting is more common when the training data is small and the model memorizes the data instead of learning generalizable patterns. In large datasets, there is a greater diversity of examples, and the risk of overfitting reduces.
- **Effect**: Using a smaller proportion for training and a larger proportion for testing or validation may still yield valid results without significant risk of overfitting.

4. Cross-Validation Helps Maximize Training Data

- **Reason**: With large datasets, the model can be trained more effectively using multiple folds in **cross-validation**, allowing it to train on different parts of the data. In cross-validation, even a small percentage of data used for testing in each fold is enough to validate performance.
- **Effect**: Cross-validation allows you to get a robust estimate of model performance using less testing data while ensuring that all training data is used.

5. Distributed Training with Large Datasets

- **Reason**: For extremely large datasets, you might also consider **distributed training**, where the model training is done on multiple machines (e.g., cloud-based infrastructure). With distributed systems, the training set can be large while still achieving reasonable training times.
- **Effect**: In such cases, you can afford to train with larger subsets of the data without running into performance bottlenecks.

Best Practices for Data Splitting in Large Datasets

1. **Use Sampling**: Instead of using the entire dataset for training, you can use a **representative sample** of the data. For example, in image recognition or natural language processing tasks, you might sample data to make training more efficient while maintaining model accuracy.
2. **Reduce Testing Set Proportion**: When dealing with massive datasets, you can reduce the testing set proportion to 5%–10% (or even less) while still obtaining reliable performance metrics.
3. **Employ Cross-Validation**: For large datasets, **cross-validation** can help utilize data efficiently, reducing the need for a large fixed test set. This provides more robust estimates while maximizing training data usage.
4. **Monitor Data Distribution**: For large datasets, always check that your splits maintain the **distribution of key features** (e.g., stratify categorical features or maintain temporal continuity in time-series data).