Cross-validation is a technique for evaluating machine learning models by training them on different subsets of the available data and testing them on the remaining data. This approach helps to detect overfitting, which occurs when a model performs well on training data but fails to generalize to new, unseen data.

Simple Example of Cross-Validation

Let's say you have a dataset with 100 records, and you want to evaluate a model using cross-validation. Here's a straightforward way to understand how it works.

1. Divide the 100 records into 5 equal parts, each with 20 records.
2. For each of the 5 iterations:

- Use 4 parts (80 records) to train the model.
- Test the model on the remaining part (20 records).

3. Do this 5 times, so each of the 5 parts gets a chance to be the test set once.

4. After completing all 5 iterations, you average the performance metrics (like accuracy) across the 5 test sets.

By using this method, you ensure that every record in your dataset is used for both training and testing, providing a more reliable estimate of how well your model performs on unseen data.

In cross-validation, we do something similar with our data

1. **Split Data** — We split our dataset into several smaller parts.
2. **Train and Test** — We train our model on some of these parts (training data) and test it on the remaining parts (testing data).
3. **Repeat** — We repeat this process several times, each time using different parts of the data for training and testing.
4. **Average Performance** — Finally, we average the results from all the tests to get a more reliable estimate of how well our model performs.

Why Cross-Validation Helps

Cross-validation is essential for several key reasons

1. Accurate Performance Estimation

Cross-validation offers a more reliable estimate of how a model will perform on new data. By testing the model on multiple data subsets, you avoid overestimating its performance based on just one train-test split.Testing a model once might give misleading results, but

averaging performance across multiple folds provides a clearer picture.

2. Reduces Overfitting

It helps prevent overfitting by ensuring the model is validated on data it hasn't seen during training. This means the model learns to generalize better rather than memorizing the training data.Training and testing on different data subsets ensures the model isn't just memorizing but learning general patterns.

3. **Efficient Data Use**

Cross-validation makes the most out of available data. Each data point is used for both training and testing, maximizing the dataset's utility.Instead of setting aside a large portion of data as a test set, cross-validation uses every data point for both training and evaluation.

4. **Improves Model Selection**

It helps in choosing the best model or configuration by comparing performance across different parts of data. This ensures the model that consistently performs well is selected.Comparing models using cross-validation helps in identifying which model has the most reliable performance.

5. **Detects Performance Variability**

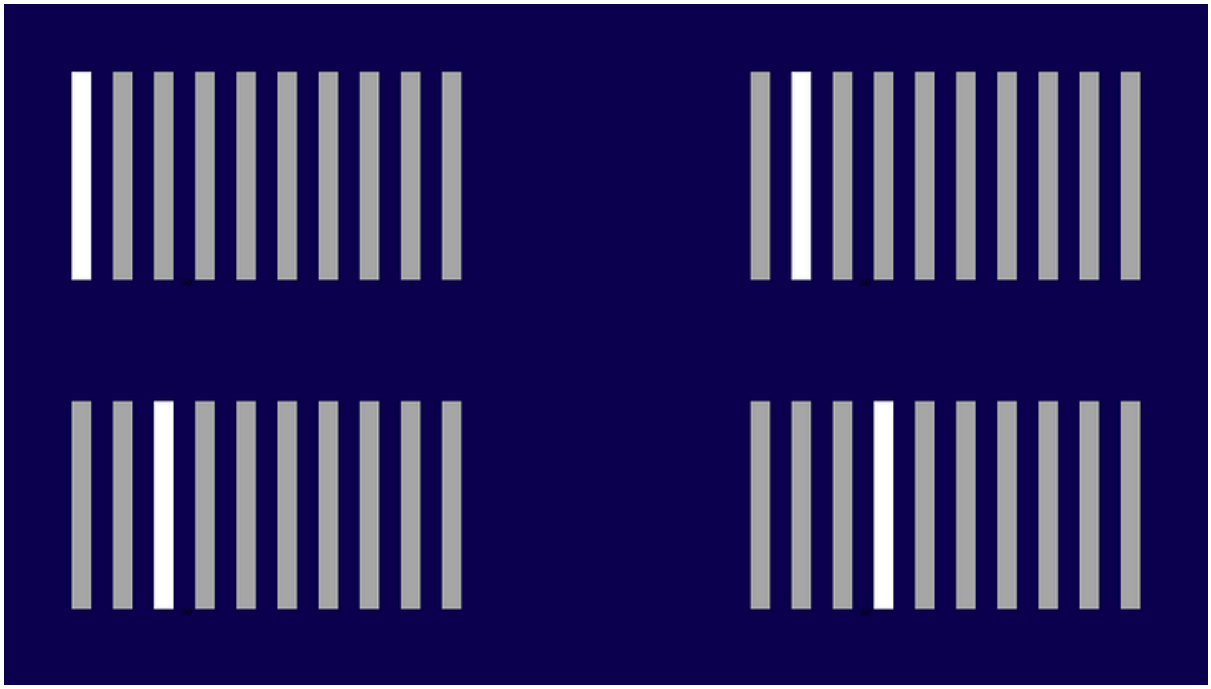
Cross-validation reveals how a model's performance varies with different data subsets, indicating its stability and robustness.A model with consistent performance across folds is more reliable, while significant variability may signal overfitting or instability.

Types of Cross-Validation

Leave-One-Out Cross-Validation (LOOCV)

It involves using each individual data point as a test set once while using the remaining data points to train the model. The main advantage of LOOCV is that it provides a thorough evaluation of the model's performance, as every data point is used for testing. However, it can be computationally expensive, especially with larger datasets, because the model needs to be trained as many times as there are data points.

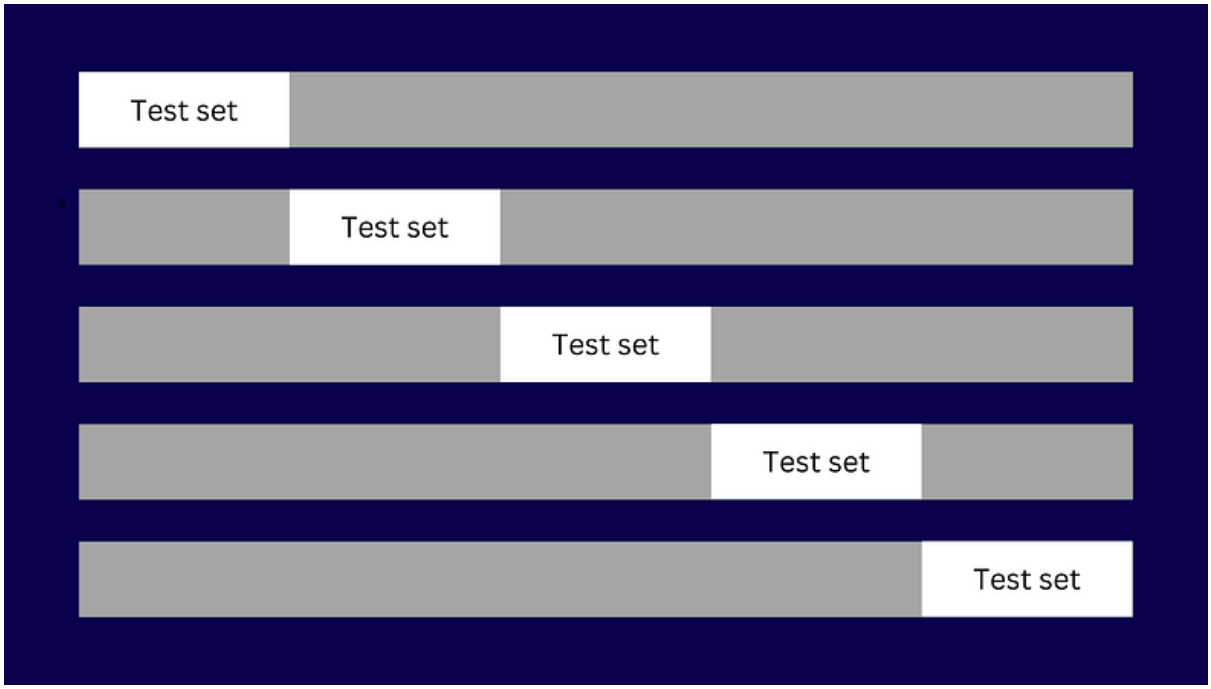
Assume you have 10 data records. You take 1 data point (let's take the first data point) as the test data and use the remaining 9 records for training the model. Then, you use the second data point as the test data and the remaining 9 records as the training data. This process continues 10 times, with each data point being used as the test data once. The below image shows only the first 4 instances of the process.



K-Fold Cross-Validation

It divides the dataset into K equally-sized parts, known as folds. The model is trained on K-1 of these folds and tested on the remaining fold (1 part of 5 set). This process is repeated K times, with each of the K folds being used as the test set once. K-Fold Cross-Validation strikes a balance between computational efficiency and a reliable performance estimate. The choice of K can affect the performance estimate, with common choices being 5 or 10.

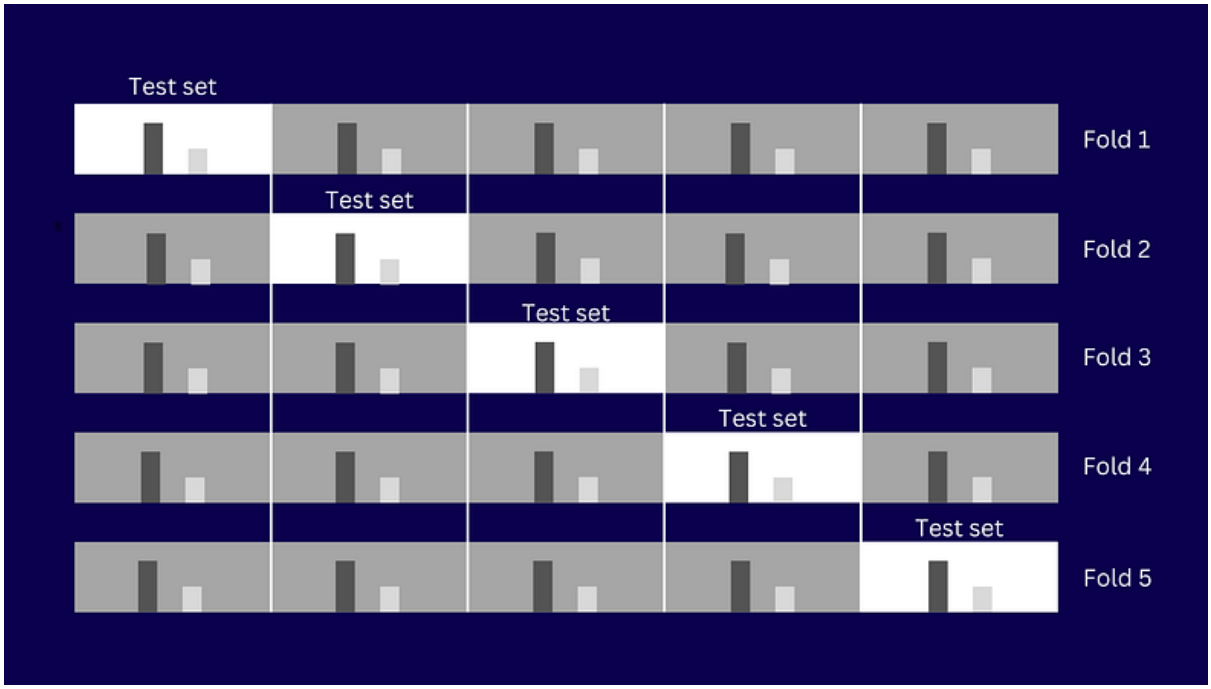
For instance, with 5-fold cross-validation, the data is split into 5 parts. The model is trained and evaluated 5 times, each time using a different fold as the test set.



Stratified Cross-Validation

It is similar to K-Fold Cross-Validation but is specifically designed to handle imbalanced datasets. In this method, each fold is created so that it reflects the overall distribution of classes in the dataset. This approach ensures that each fold is representative of the whole dataset, which is particularly important for datasets where some classes are underrepresented. Stratified Cross-Validation provides a more consistent and fair evaluation of the model's performance on imbalanced data.

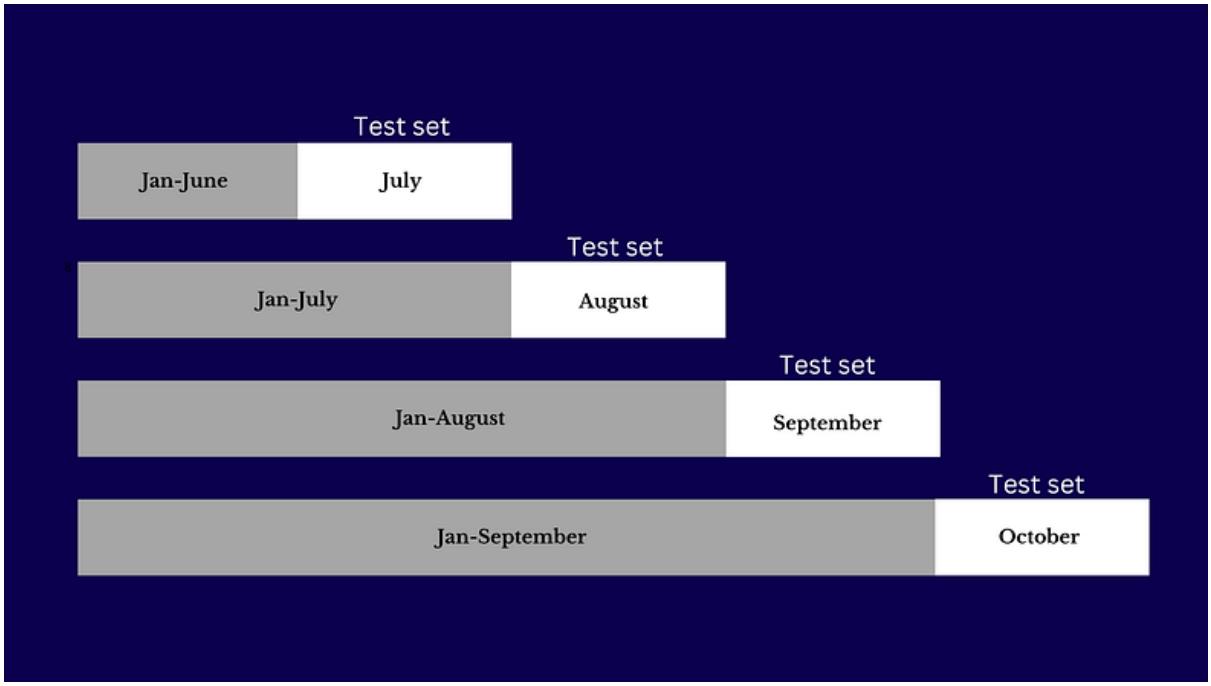
For instance, if 10% of your data belongs to one class and 90% to another, each fold will maintain this ratio. In the example image, you can see that each fold contains both classes and preserves the original ratio of the dataset for each class.



Time Series Cross-Validation

It is used for datasets where the order of data points is crucial, such as in time series forecasting. This method involves splitting the data in a way that respects the chronological order. The model is trained on past data and tested on future data, which reflects real-world scenarios where predictions are made based on historical information. Time Series Cross-Validation ensures that the model is evaluated in a manner that aligns with the temporal nature of the data.

For instance, you might train the model on data from January to June and test it on data from July. Then, you might train on data from January to July and test on August data.



Each of these cross-validation techniques helps ensure that the model's performance is accurately assessed and that it can generalize well to new data. The choice of method depends on the characteristics of the dataset and the specific requirements of the model evaluation.

Cross-validation is a powerful technique for evaluating machine learning models, ensuring that they perform well across different subsets of data and generalize effectively to new, unseen data. By dividing your data into multiple folds and iteratively training and testing your model, you gain a more reliable assessment of its performance and reduce the risk of overfitting. Whether using k-fold, leave-one-out, or other methods, cross-validation helps you make informed decisions about model selection and improvement, ultimately leading to more robust and accurate predictive models.

In the previous article, I explained cross-validation. Now, I'm going to show you K-fold cross-validation in action. This is one of the most common techniques in cross-validation. Remember, if your dataset is imbalanced, you should first address this issue before using any cross-validation technique.

K-fold cross-validation involves splitting your dataset into K parts (folds). Then, the technique uses one fold to test the model and the remaining K-1 folds to train the model. Each fold acts as a test set in turn, during K iterations.

You can download the notebook here.

In this example, I'm using the California Housing dataset from scikit-learn. Since we need to predict housing prices, this is a regression problem. Therefore, I'll use the `RandomForestRegressor` as the model (estimator).

Let's dive into the code now

Import necessary tools

```
from sklearn.datasets import fetch_california_housing
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split,cross_val_score
from sklearn.metrics import
r2_score,mean_absolute_error,mean_squared_error
import numpy as np
import pandas as pd
```

Fetch data

```
housing = fetch_california_housing()
housing

housing_df = pd.DataFrame(housing['data'],columns =
housing['feature_names'])
housing_df['target'] = housing['target']
housing_df.head()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	target
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422

This code segment fetches the California Housing dataset and stores it in a variable called `housing`. It then creates a pandas DataFrame from the dataset, using the feature names as column labels for the data. Next, it adds a new column to the DataFrame for the target values (housing prices). Finally, the code displays the first few rows of the DataFrame to provide a preview of the data.

```
np.random.seed(42)

X = housing_df.drop('target',axis=1)
y = housing_df['target']

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


mdl = RandomForestRegressor()
mdl.fit(X_train,y_train)
```

This code sets a random seed for reproducibility, then prepares the data by separating features (`X`) from the target values (`y`). It splits the data into training and testing sets, with 20% of the data reserved for

testing. Finally, it initializes a Random Forest Regressor model and trains it using the training data.

Regression Model Evaluation Metrics

The ones we are going to cover are:

1. R^2 (Pronounced r-squared) or coefficient of determination
2. Mean absolute error (MAE)
3. Mean squared error (MSE)

Now, Let's consider these values without cross-validation

R2 Score

The R^2 score measures how well a model's predictions match the actual data. It compares the model's predictions to the average of the actual target values. The score ranges from negative infinity to 1: a score of 1 means perfect predictions, while a score of 0 means the model's predictions are no better than simply using the average of the target values.

```
y_preds =  mdl.predict(X_test)
r2_score(y_true=y_test,y_pred=y_preds)

# output => 0.8066196804802649
```

Mean Absolute Error (MAE)

The Mean Absolute Error (MAE) measures the average of the absolute differences between the predicted values and the actual values. It tells you how far off your model's predictions are from the true values, providing a straightforward measure of prediction accuracy.

Data and MAE are on **same scale.**

```
# MAE

y_preds = mdl.predict(X_test)
mae = mean_absolute_error(y_test,y_preds)

mae

# output => 0.3265721842781009
```

Mean Squared Error (MSE)

The Mean Squared Error (MSE) calculates the average of the squared differences between the predicted values and the actual values. Because MSE squares the errors, it penalizes larger errors more heavily. To get back to the original scale of the data, you can use the Root Mean Squared Error (RMSE), which is the square root of the MSE.

```
# MSE

y_preds = mdl.predict(X_test)
mse = mean_squared_error(y_test,y_preds)
mse

# output => 0.2534678520824551
```

Now, we are going to calculate the same metrics using cross-validation

Refer to this table for choosing scoring parameters . According to the problem you are solving (Classification, Regression, or Clustering), the scoring parameters are different. Go through the link and find those.

R2 score (using cross-validation)

```
np.random.seed(42)

cv_score =  cross_val_score(mdl,X,y,cv=5,scoring=None), # If scoring is
# None,Model's default scoring evaluation metric is used. (in our case, it
is r2  # score)

# Take the mean
np.mean(cv_score)

# output => 0.6521420895559876
```

In this case, we used `scoring='None'`, which means the default metric used is the R^2 score. We can verify this by comparing it with the code segment below, as both segments will yield the same values.

```
np.random.seed(42)
r2 =  cross_val_score(mdl,X,y,cv=5,scoring='r2')
r2
# output => (array([0.51682354, 0.70280719, 0.74200859, 0.61659773,
0.68247339]),)

np.mean(r2)

# output => 0.6521420895559876
```

The `cross_val_score` function performs 5-fold cross-validation on the regression model (`mdl`) using the R^2 score as the metric (`scoring='r2'`). This means the dataset is split into 5 parts (`cv=5`), with each part serving as the test set once while the remaining 4 parts are used for training.

The function `cross_val_score` returns an array of R^2 scores, one for each fold. In this example, the R^2 scores for the 5 folds are approximately `[0.5168, 0.7028, 0.7420, 0.6166, 0.6825]`.
The `np.mean(r2)` calculates the average of these scores, which is about `0.6521`.

This average R^2 score gives a more comprehensive view of the model's performance across different subsets of the data compared to a single R^2 score obtained from one test/train split.

MAE (using cross-validation)

```
np.random.seed(42)
mae =  cross_val_score(mdl,X,y,cv=5,scoring='neg_mean_absolute_error')
mae

# output => array([-0.54255936, -0.40903449, -0.43716367, -0.46911343, -
0.47319069])

np.mean(mae)

# output => -0.4662123287693799
```

In this code, the `cross_val_score` function performs 5-fold cross-validation (`cv=5`) on the model using Mean Absolute Error (MAE) as the scoring metric (`scoring='neg_mean_absolute_error'`).

Since `scoring='neg_mean_absolute_error'` returns negative values you may be wondering. Let's clarify it.

MAE measures the average absolute difference between predicted and actual values. Lower MAE values indicate better model performance because fewer errors mean better predictions.

Since `cross_val_score` is designed to maximize scores, it's set up to prefer higher values. To adapt MAE, which benefits from being lower, the function returns the negative of the MAE. For example, if the MAE is 5, it returns -5; if the MAE is 9, it returns -9.

This sign flip allows the function to follow its maximization convention. A higher negative value (less negative) corresponds to a smaller MAE, which indicates better performance. Thus, -5 is better than -9 because -5 (less negative) represents a lower MAE, reflecting fewer errors and better predictions.

I hope you got the idea. Let's continue from where we left.

The `np.mean(mae)` function calculates the average of these negative MAE scores, resulting in approximately `-0.4662`. This average provides an overall measure of the model's prediction accuracy across different data subsets. To get the actual MAE, you would take the negative of this value.

MSE (using cross-validation)

```
np.random.seed(42)
mse =  cross_val_score(mdl,X,y,cv=5,scoring='neg_mean_squared_error')
mse

# output => array([-0.51906307, -0.34788294, -0.37112854, -0.44980156, -
0.4626866 ])

np.mean(mse)

# output => -0.43011254261460774
```

In this code, `cross_val_score` evaluates the regression model's performance using 5-fold cross-validation (`cv=5`) with the Mean Squared Error (MSE) as the metric.
The `scoring='neg_mean_squared_error'` parameter indicates that the function returns negative MSE values, as minimizing MSE is equivalent to maximizing the negative MSE.

The output array contains negative MSE values for each fold: `[-0.5191, -0.3479, -0.3711, -0.4498, -0.4627]`. The average of these negative MSE values, calculated as `np.mean(mse)`, is approximately `-0.4301`. This average negative MSE represents the typical squared error of the model's predictions across different data subsets, with the actual MSE being around `0.4301`.

Cross-validation, especially with k-fold (like 5-fold), provides a more reliable evaluation of a model's performance compared to a single train-test split. It helps in understanding how well the model generalizes to different subsets of the data. When using a model like `RandomForestRegressor`, cross-validation helps mitigate the risk of overfitting and provides a more robust estimate of performance.

This approach ensures that the evaluation is less dependent on the particular train-test split and gives a more comprehensive view of the model's effectiveness across various data splits.