**Cross-Validation**

Cross-validation is a statistical method used to estimate the skill of machine learning models.

It is commonly used in applied machine learning to compare and select a model for a given predictive modeling problem because it is easy to understand, easy to implement, and results in skill estimates that generally have a lower bias than other methods.

Regression and classification Machine Learning models aim to predict a value or class from the variables contained in the data. Each model has its own algorithm to try to identify the patterns contained in the data that allow an accurate prediction to be made.

The models, in addition to being accurate, must be generalist, with the ability to interpret data never seen before and reach an adequate result. One way of evaluating this generalization capacity of the models is to apply Cross-Validation.

Cross-validation is a technique used as a way of obtaining an estimate of the overall performance of the model. There are several Cross-Validation techniques, but they basically consist of separating the data into training and testing subsets.

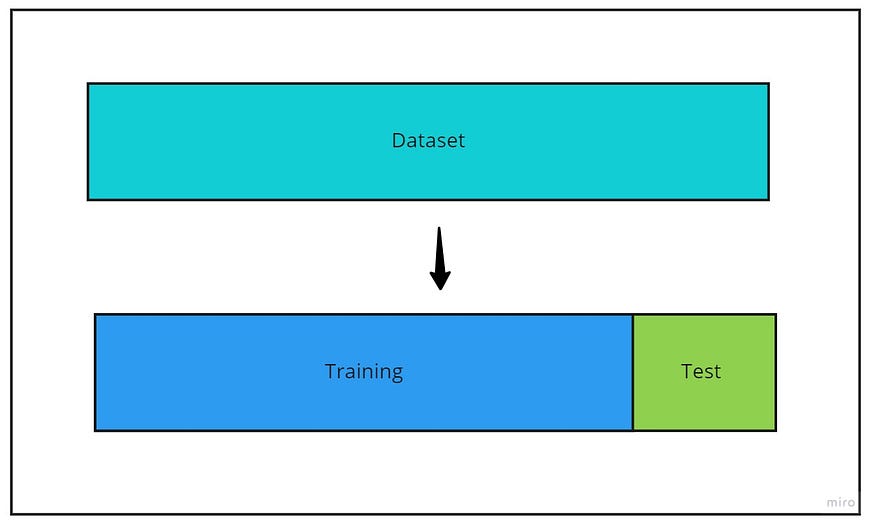
The training subset, as the name implies, will be used during the training process to calculate the hyperparameters of the model. To calculate the generalization capacity of the model, after the training stage, the test model is used.

The performance metrics of the model such as Accuracy (classification) and Root Mean Absolute Error (regression) are calculated using the true labels from the test dataset and the predictions made by the trained model on the test data.

There are many types of Cross-Validation techniques, and in this post I will talk about three of them: **Holdout**, **K-Fold** and **Leave-One-Out.**

probably the most famous type of Cross-Validation technique is the Holdout. This technique consists in separating the whole dataset into two groups, without overlap: training and testing sets. This separation can be made shuffling the data or maintaining its sorting, depends on the project.

It is common to see a 70/30 split in projects and studies, with 70% of the data being used to train the model and the remaining 30% being used to test and evaluate it. However, this ratio is not a rule and it may vary depending on the specificity of the project.



Example of the Holdout Cross-Validation applied to a dataset — image by author

In Python, the Holdout Cross-Validation is easily done using the train\_test\_split [function](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)from the scikit-learn library.

**Why cross-validation?**

*CV provides the ability to estimate model performance on unseen data not used while training.*

Data scientists rely on several reasons for using cross-validation during their building process of Machine Learning (ML) models. For instance, tuning the model hyperparameters, testing different properties of the overall datasets, and iterate the training process. Also, in cases where your training dataset is small, and the ability to split them into training, validation, and testing will significantly affect training accuracy. The following main points can summarize the reason we use a CV, but they overlap. Hence, the list is presented here in a simplified way:

**(1) Testing on unseen data**

One of the critical pillars of validating a learning model before putting them in production is making accurate predictions on unseen data. The unseen data is all types of data that a model has never learned before. Ideally, the testing data is supposed to flow directly to the model in many testing iterations. However, in reality, access to such data is limited or not yet available in a new environment.

The typical 80–20 rule of splitting data into training and testing can still be vulnerable to accidentally ending up in a perfect split that boosts the model accuracy while limiting it from performing the same in a real environment. Sometimes, the accuracy calculated this way is mostly a matter of luck! The 80–20 is not an actual rule per se, and you will find alternative ratios that range between 25~30% for testing and 70~75% for training.

**(2) Tuning model hyperparameter**

Finding the best combination of model parameters is a common step to tune an algorithm toward learning the dataset’s hidden patterns. But, doing this step on a simple training-testing split is typically not recommended. The model performance is usually very sensitive to such parameters, and adjusting those based on a predefined dataset split should be avoided. It can cause the model to overfit and reduce its ability to generalize.

**(3) The third split is not achievable**

For parameter tuning and avoid model overfitting, you might find some recommendations around splitting the dataset into three partitions: training, testing, and validation. For instance, 70% of the data is used for training, 20% for validation, and the remaining 10% is used for testing. In cases where the actual dataset is small, we might need to invest in using the maximum amount of data to train the model. In other instances, splitting into such three partitions could create bias in the training process where some significant examples are kept in training or validation splits. Hence, CV can become handy to resolve this issue.

**(4) Avoid instability of sampling**

Sometimes the splits of training-testing data can be very tricky. The properties of the testing data are not similar to the properties of the training. Although randomness ensures that each sample can have the same chance to be selected in the testing set, the process of a single split can still bring instability when the experiment is repeated with a new division.

**How does it work?**

Cross-Validation has two main steps: splitting the data into subsets (called folds) and rotating the training and validation among them. The splitting technique commonly has the following properties:

* Each **fold** has approximately the same size.
* Data can be **randomly** selected in each fold or **stratified**.
* All folds are used to train the model except one, which is used for validation. That validation fold should be rotated until all folds have become a validation fold once and only once.
* Each example is recommended to be contained in one and only one fold.

**K-fold** and **CV** are two terms that are used interchangeably. K-fold is just describing how many folds you want to split your dataset into. Many libraries use k=10 as a default value representing 90% going to training and 10% going to the validation set. The next figure describes the process of iterating over the picked ten folds of the dataset.



Figure 2: A 10-fold representation of how each fold is used in the cross-validation process.

Figure 2 shows how one fold in each step iteratively is held out for testing while the remaining folds are used to build the model. Each step calculates the prediction error. Upon the completion of the final step, a list of computed error are produced of which we can take the mean.

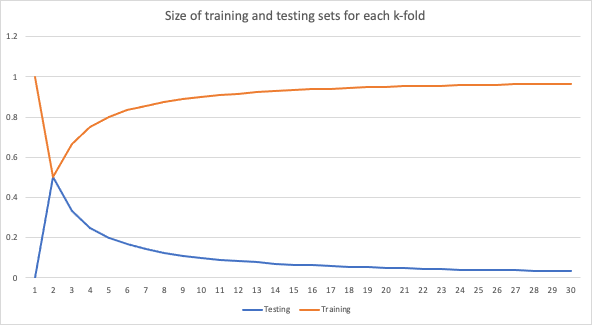


Figure 3: A graph representation showing training vs. testing size in each value picked for k.

Figure 3 shows the change in the training and validation sets’ size when using different values for k. The training set size increases whenever we increase the number of folds while the validation set decreases. Typically, the smaller the validation set, the likelihood that the randomness rises with a high noise variation. Increasing the training set size will increase such randomness and bring more reliable performance metrics.

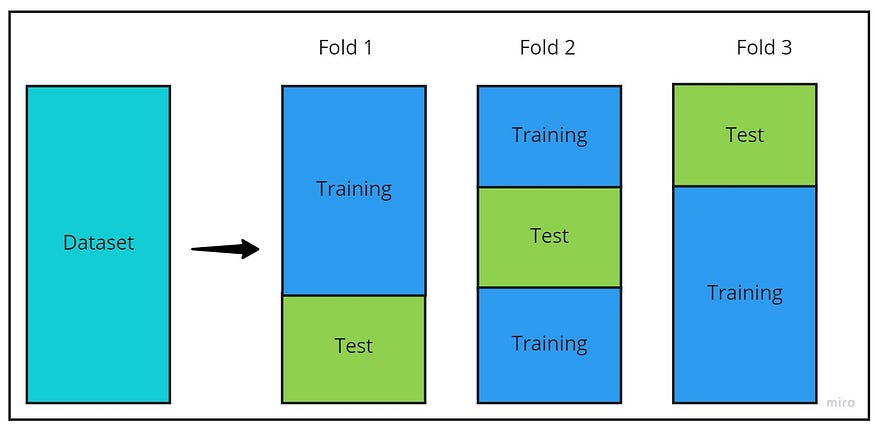
**Analyzing the results**

One of the advantages of CV is observing the model predictions against all the instances in the dataset. It ensures that the model has been tested on the full data without testing them simultaneously. Variations are expected in each step of the validation; therefore, computing the mean and standard deviation can reduce the information into a few comparing values.

**K-Fold Cross-Validation**

Before separating the data into training and testing sets, the K-Fold Cross-Validation separates the whole data into K separated subsets with approximate size. Only then, each subset is divided into training and testing sets.

Each subset is used to train and test the model. In practice, this technique generates K different models with K different results. The final result of the K-Fold Cross-Validation is the average of the individual metrics of each subset.



Example of a 3-Fold Cross-Validation applied to a dataset — image by author

It is important to notice that since the K-Fold divides the original data into smaller subsets, the size of the dataset and the K number of subsets must be taken into account. If the dataset is small or the number of K is too big, the resulting subsets may become very small.

This may result in just a few data to be used to train the models, resulting in a poor performance since the algorithm couldn’t understand and learn the patterns in the data due to lack of information.

Python also has a easy way to perform the K-Fold split using the Kfold from the [scikit-learn library](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html).

from sklearn.model\_selection import KFold

print('Shape of the original data:', X.shape)

# Output:

# Shape of the data: (569, 30)

# Creating the K-Fold ojbect

kf = KFold(n\_splits = 3)

# Performing the K-Fold Cross-Validation

# Each split represents one subset

for train\_index, test\_index in kf.split(X):

# Obtaining the index for the training and test sets for the subset

print("First five train index:", train\_index[0:5], "First five test index:", test\_index[0:5])

print('Shape of the Training set:', train\_index.shape, 'Shape of the Test set:', test\_index.shape, '\n')

# Output:

# First five train index: [190 191 192 193 194] First five test index: [0 1 2 3 4]

# Shape of the Training set: (379,) Shape of the Test set: (190,)

#

#First five train index: [0 1 2 3 4] First five test index: [190 191 192 193 194]

#Shape of the Training set: (379,) Shape of the Test set: (190,)

#

#First five train index: [0 1 2 3 4] First five test index: [380 381 382 383 384]

#Shape of the Training set: (380,) Shape of the Test set: (189,)

**Fundamentally, the Holdout Cross-Validation is the same as a 1-Fold Cross-Validation.**

**Other techniques for cross-validation**

There are other techniques on how to implement cross-validation. Let’s jump into some of those:

**(1) Leave-one-out cross-validation (LOOCV)**

LOOCV is the an exhaustive holdout splitting approach that k-fold enhances. It has one additional step of building k models tested with each example. This approach is quite expensive and requires each holdout example to be tested using a model. It also might increase the overall error rate and becomes computationally costly if the dataset size is large. Usually, it is recommended when the dataset size is small. Figure 4 demonstrates the overall process of a simple dataset containing ten examples.

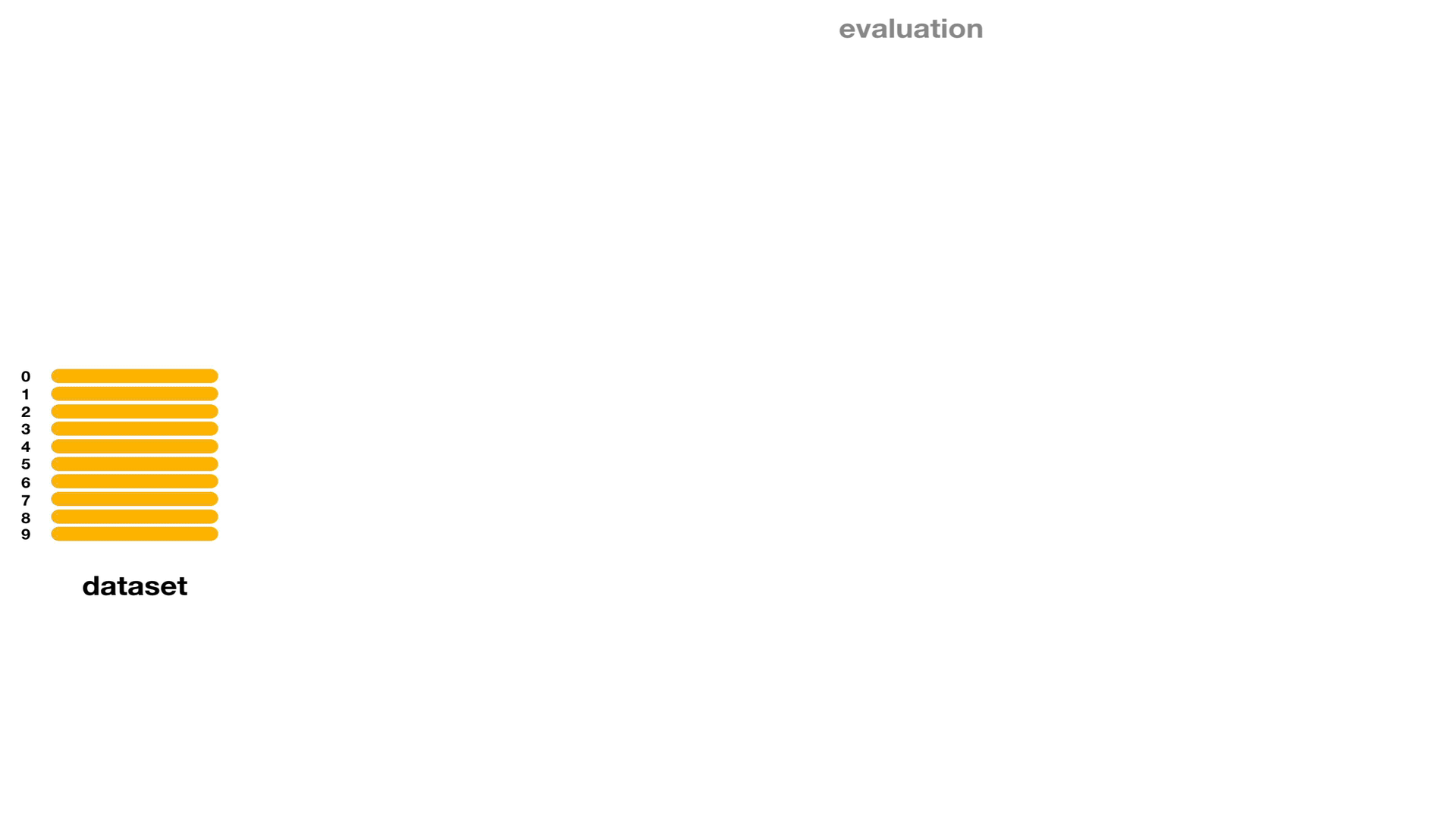


Figure 4: A demonstration of how the dataset is split into partitions using LOOCV.

**(2) Leave-pair-out cross-validation (LPOCV)**

It is another exhaustive technique for performing CV. We can define in each iteration how many examples shall be used for testing the model and how many shall be left for training. The process is repeated for all possible combinations of pairs in the dataset. Note that LOOCV is a LOPCV technique where p =1.

**Example**

Let’s refresh our minds on how to split the data using the Sklearn library. The following code divides the dataset into two splits: training and testing. We defined here that 1/3 of the dataset should be used for testing.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=1)

We will then build a function to show how k-fold works compared to the single split in the previous code.

def k\_Fold\_Split(data, n\_splits, shuffle, random\_state, verbose=False):  
 # Creating k-fold splitting   
 kfold = KFold(n\_splits, shuffle, random\_state)  
 sizes = [0 , 0]  
 for train, test in kfold.split(data):  
 if verbose:  
 print('train: indexes %s, val: indexes %s, size (training) %d, (val) %d' % (train, test, train.shape[0], test.shape[0]))  
 sizes[0] += train.shape[0]  
 sizes[1] += test.shape[0]  
   
 return int(sizes[0]/n\_splits), int(sizes[1]/n\_splits)

The `k\_fold\_split` function shows the indexes and the size of each partition. It enables you to track how much data is dedicated for each fold and the chosen indexes.

**LOOCV**

Using LOOCV as a splitting strategy is pretty straight forward. We will use again Sklearn library to perform the cross-validation.

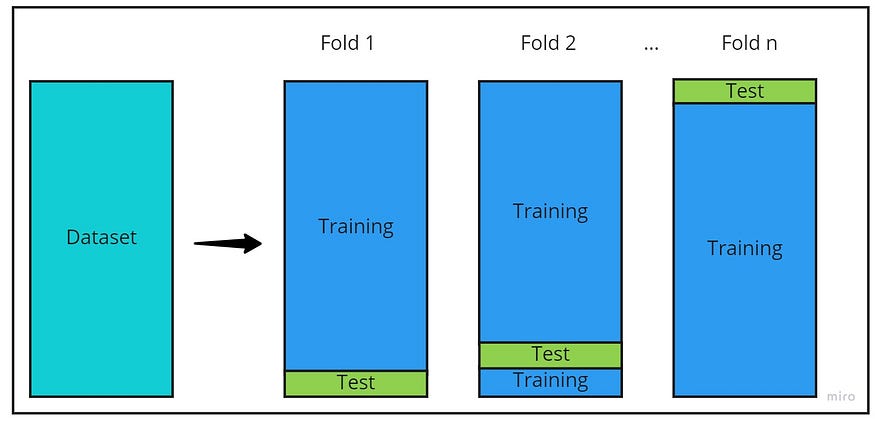
from sklearn.model\_selection import LeaveOneOut  
cv\_strategy = LeaveOneOut()  
# cross\_val\_score will evaluate the model   
scores = cross\_val\_score(estimator, X, y, scoring='accuracy', cv=cv\_strategy, n\_jobs=-1)

**Leave-One-Out Cross-Validation**

The Leave-One-Out Cross-Validation consists in creating multiple training and test sets, where the test set contains only one sample of the original data and the training set consists in all the other samples of the original data. This process repeats for all the samples in the original dataset.

This type of validation usually is very consuming because if the data used contains *n* samples, the algorithm will have to train (using *n-1* samples) and evaluate the model *n*times.

On the positive side, this technique, of all seen in this post, is the one in which the models used have the largest amount of samples used for training, and this may result in better models developed. Also, there is no need to shuffle the data, since all possible combinations of train/test sets will be generated.



from sklearn.model\_selection import LeaveOneOut

print('Shape of the original data:', X.shape)

# Output:

# Shape of the data: (569, 30)

# Creating the Leave-One-Out ojbect

loo = LeaveOneOut()

# Auxiliar variable to stop the prints

break\_aux = 0

# Performing the Leave-One-Out Cross-Validation

for train\_index, test\_index in loo.split(X):

# Stop printing after the third iteraction

if break\_aux < 3:

print("First five train index:", train\_index[0:5], "Test index:", test\_index)

break\_aux = break\_aux + 1

# Output

# First five train index: [1 2 3 4 5] Test index: [0]

# First five train index: [0 2 3 4 5] Test index: [1]

# First five train index: [0 1 3 4 5] Test index: [2]

Similar to the Holdout, the Leave-One-Out Cross-Validation is also a special type of K-Fold, where the value of K is equal to the number of samples of the dataset.

**Careful Considerations**

**Time-series dataset**

Cross-validation is a great way to ensure the training dataset does not have an implicit type of ordering. However, some cases require the order to be preserved, such as time-series use cases. We can still use cross-validation for time-series datasets using some other technique such as time-based folds.

**Unbalanced dataset**

Dealing with cross-validation in an unbalanced dataset can be tricky as well. If oversampling is used, the leaking of the oversampled examples can mislead the CV results. One way to consider is the use of stratified sampling instead of splitting randomly. Here is a fantastic blog article discussing how to handle this [situation](https://www.marcoaltini.com/blog/dealing-with-imbalanced-data-undersampling-oversampling-and-proper-cross-validation).

**Nested cross-validation**

CV measures the generalization of the model, but it cannot avoid bias altogether. Nested cross-validation focuses on ensuring the model’s hyperparameters are not overfitting the dataset. The nested keyword comes to hint at the use of double cross-validation on each fold. The hyperparameter tuning validation is achieved using another k-fold splits on the folds used to train the model.

**Overfitting**

As mentioned earlier, CV is used to measure if a learning model can **generalize** on unseen data. We rely on using a validation dataset for early stopping and parameter tuning different from the testing set during the building of models. There are a couple of research papers recently suggesting the use of CV to measure overfitting as well. In this case, the direct application would be the use of CV as a validation set for a learning model.

**Summary**

Cross-validation is a procedure to evaluate the performance of learning models. Datasets are typically split in a random or stratified strategy. The splitting technique can be varied and chosen based on the data’s size and the ultimate objective. Also, the computational cost plays a role in implementing the CV technique. Regression and classification problems can use CV, but careful consideration must be paid for some types of datasets such as time-series.

[A Gentle Introduction to k-fold Cross-Validation - MachineLearningMastery.com](https://machinelearningmastery.com/k-fold-cross-validation/)

[3.1. Cross-validation: evaluating estimator performance — scikit-learn 1.5.1 documentation](https://scikit-learn.org/stable/modules/cross_validation.html)

[What is Cross-Validation? Testing your machine learning models… | by Mohammed Alhamid | Towards Data Science](https://towardsdatascience.com/what-is-cross-validation-60c01f9d9e75)

[Cross Validation in Machine Learning - GeeksforGeeks](https://www.geeksforgeeks.org/cross-validation-machine-learning/)

[Cross Validation - What, Why and How | Machine Learning | by Ashwin Prasad | Analytics Vidhya | Medium](https://medium.com/analytics-vidhya/cross-validation-what-why-and-how-machine-learning-f8a1159ce5ff)

[medium/cross\_validation/cross\_validation\_examples.ipynb at main · alerlemos/medium (github.com)](https://github.com/alerlemos/medium/blob/main/cross_validation/cross_validation_examples.ipynb)