Machine Learning

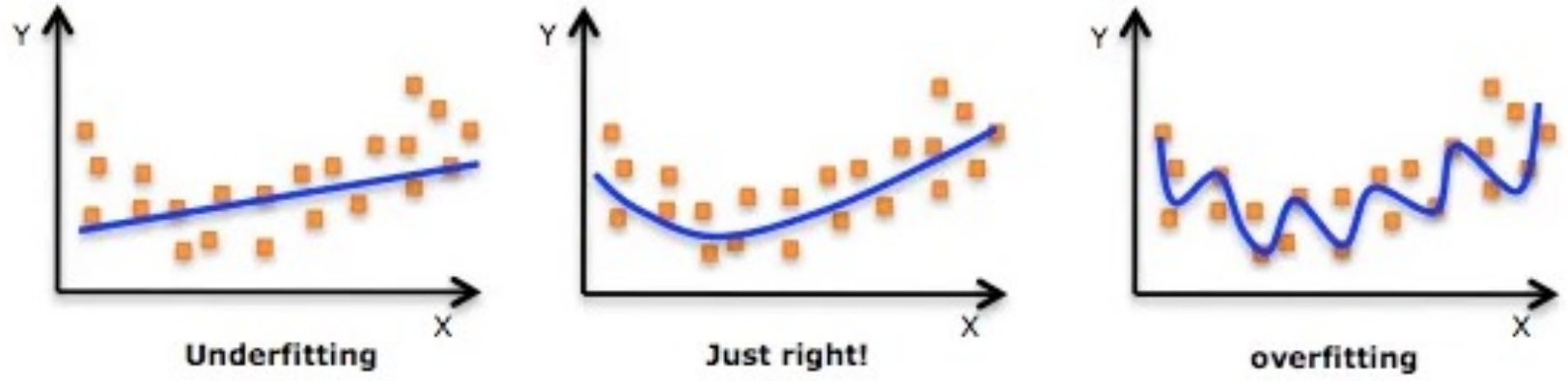
06048203

Overfitting

Overfitting means that model we trained has trained “too well” and is now, well, fit too closely to the training dataset. This usually happens when the model is too complex (i.e. too many features/variables compared to the number of observations). This model will be very accurate on the training data but will probably be very not accurate on untrained or new data

Underfitting

In contrast to overfitting, when a model is underfitted, it means that the model does not fit the training data and therefore misses the trends in the data. It also means the model cannot be generalized to new data.



Underfitting in machine learning occurs when a model is too simple and doesn't capture the underlying structure or complexity of the data, resulting in poor performance. Here are several strategies to address underfitting:

Increase Model Complexity: If your model is too simple, consider making it more complex. For example, if you're using a linear model, you might consider adding interaction terms or using a non-linear model. For a neural network, consider adding more layers or nodes.

Feature Engineering: You can add more relevant features to help the model better understand the underlying patterns in the data. These could be interactions, transformations, or entirely new data. Domain knowledge can be particularly helpful in creating new features.

Increase the Amount of Training Data: Sometimes, the model might be underfitting because there isn't enough data for it to learn effectively. If possible, collect more data.

Balance Your Dataset: If your dataset is imbalanced, the model might be underfitting because it's biased towards the majority class. Techniques like oversampling the minority class or undersampling the majority class can help.

Data Transformation: Sometimes, linear models can underfit if the data isn't linearly separable. In such cases, applying transformations like logarithmic or polynomial transformations could potentially help the model fit better.

Decrease Regularization: If you're using a regularization technique (like L1 or L2) to avoid overfitting, reducing the regularization parameter might allow your model to fit the data more closely.

Change the Model: Some models might be inherently unsuitable for a given task. If underfitting persists despite your efforts, consider using a different machine learning algorithm.

Use Ensemble Methods: Methods like Bagging or Boosting can help to reduce bias and variance, and hence could potentially reduce underfitting.

Overfitting in machine learning occurs when a model learns the training data too well, including its noise and outliers, and thus performs poorly on unseen data. This is often a result of a model that is too complex. Here are several strategies to tackle overfitting:

Collect More Data: The more data you have, the harder it is for the model to learn noise and outliers by heart. This strategy can help, especially when your dataset is small.

Data Augmentation: For certain types of data like images, you can create new training samples via augmentation, such as flipping, rotating, or cropping images in different ways. This can also help to make your model more robust.

Simplify the Model: If your model is too complex (like a deep neural network with many layers), you might want to simplify it. This could mean using a less complex model, reducing the number of layers in a deep learning model, or reducing the number of features in your dataset.

Regularization: Regularization techniques like L1 (Lasso), L2 (Ridge), or Elastic Net add a penalty to the loss function, which discourages the model from learning a too complex model and hence helps to avoid overfitting.

Cross-Validation: Cross-validation, such as k-fold cross-validation, provides a robust estimate of the model's performance on unseen data, which can help in identifying if your model is overfitting.

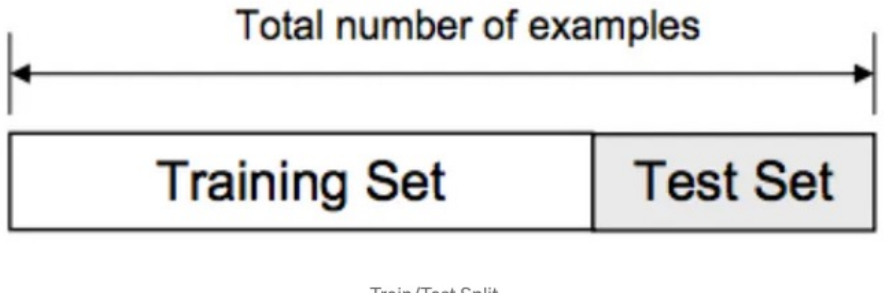
Early Stopping: When training a neural network, you can use a technique called early stopping. During the training process, keep track of the model's performance on a validation set. When the performance on the validation set starts to degrade (i.e., error starts to increase or accuracy starts to decrease), stop the training process. This prevents the model from learning the training data too well.

Dropout: In neural networks, dropout is a regularization technique where during training, randomly selected neurons are ignored or "dropped out". This helps to prevent overfitting by making the network more robust and less prone to rely heavily on particular neurons.

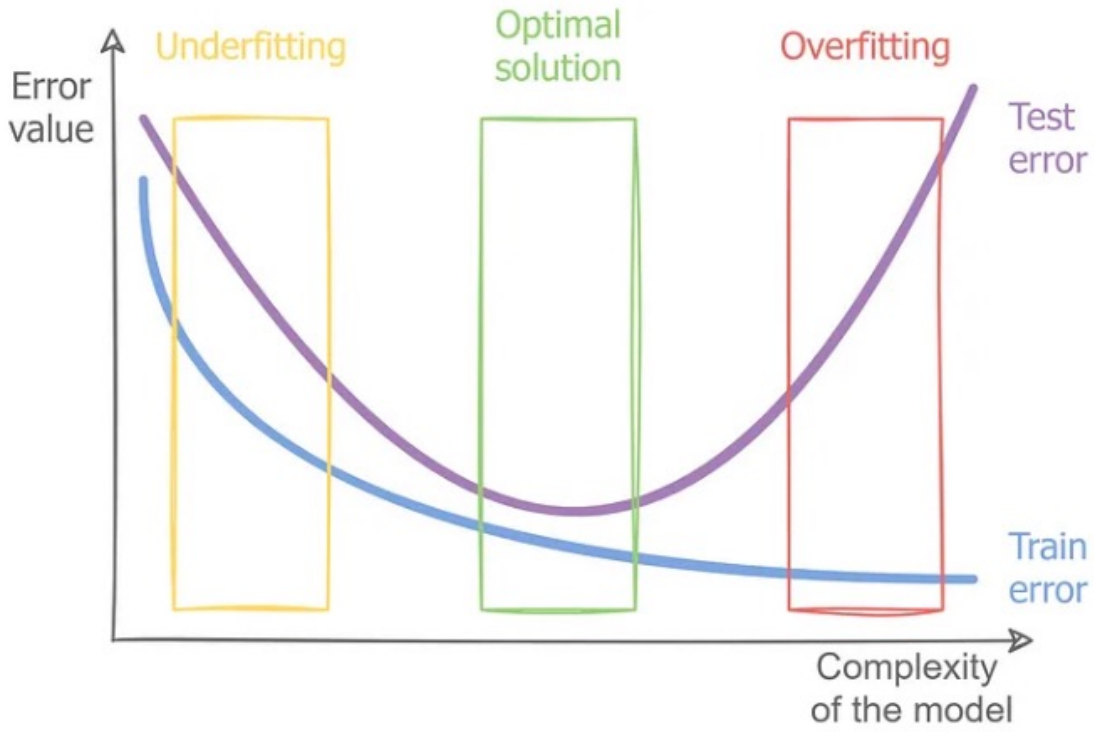
Ensemble Methods: These are meta-algorithms that combine several machine learning techniques into one predictive model to decrease variance (bagging), bias (boosting), or improve predictions (stacking). Ensemble methods can help in avoiding overfitting by averaging the results of different models, thus reducing the chance that the final model will learn the noise in the training data.

Train/Test Split

- the data we use is usually split into training data and test data. The training set contains a known output and the model learns on this data in order to be generalized to other data later on. We have the test dataset (or subset) in order to test our model's prediction on this subset.



How to Detect Underfitting and Overfitting

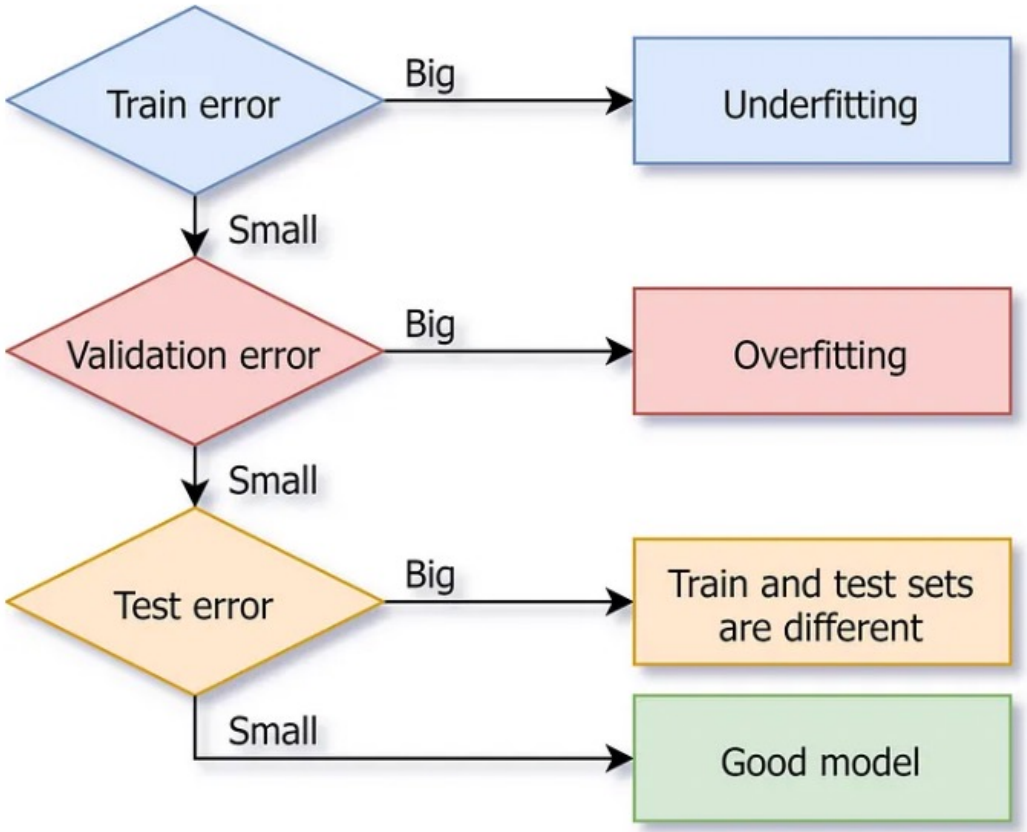


1.Underfitting means that your model makes accurate, but initially incorrect predictions. In this case, train error is large and val/test error is large too.

2.Overfitting means that your model makes not accurate predictions. In this case, **train error is very small** and **val/test error is large**.

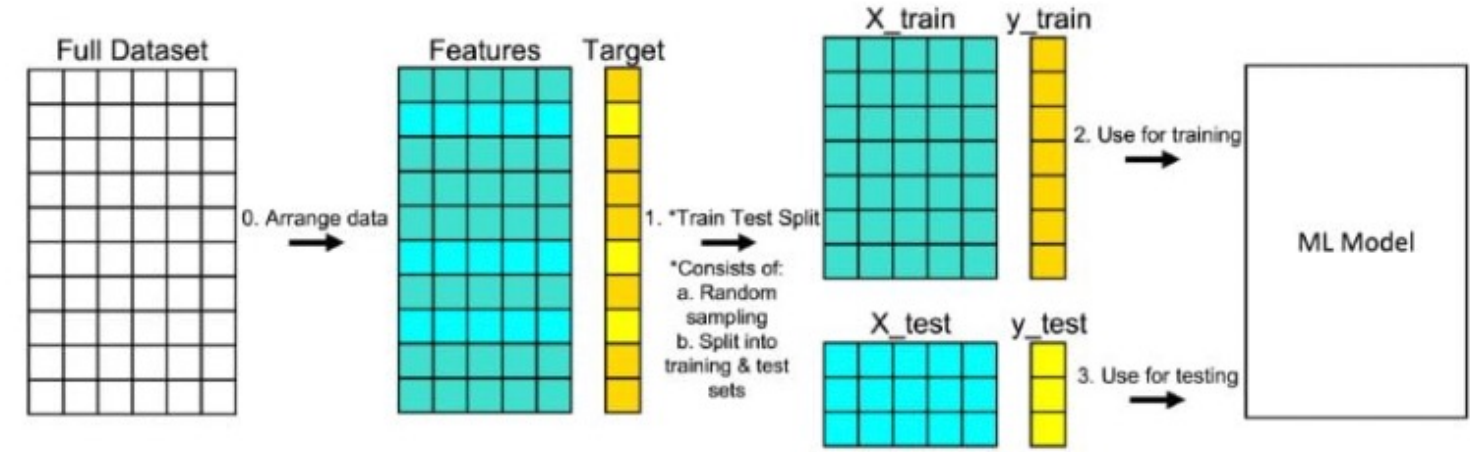
3. **Good model**, **train error is small** (but larger than in the case of overfitting), and **val/test error is small** too

If validation and test error are very different, then you need to get more data similar to test data and make sure that you split the data correctly.



- **What Is the Train Test Split Procedure?**

Train test split is a model validation procedure that allows you to simulate how a model would perform on new/unseen data. Here is how the procedure works:

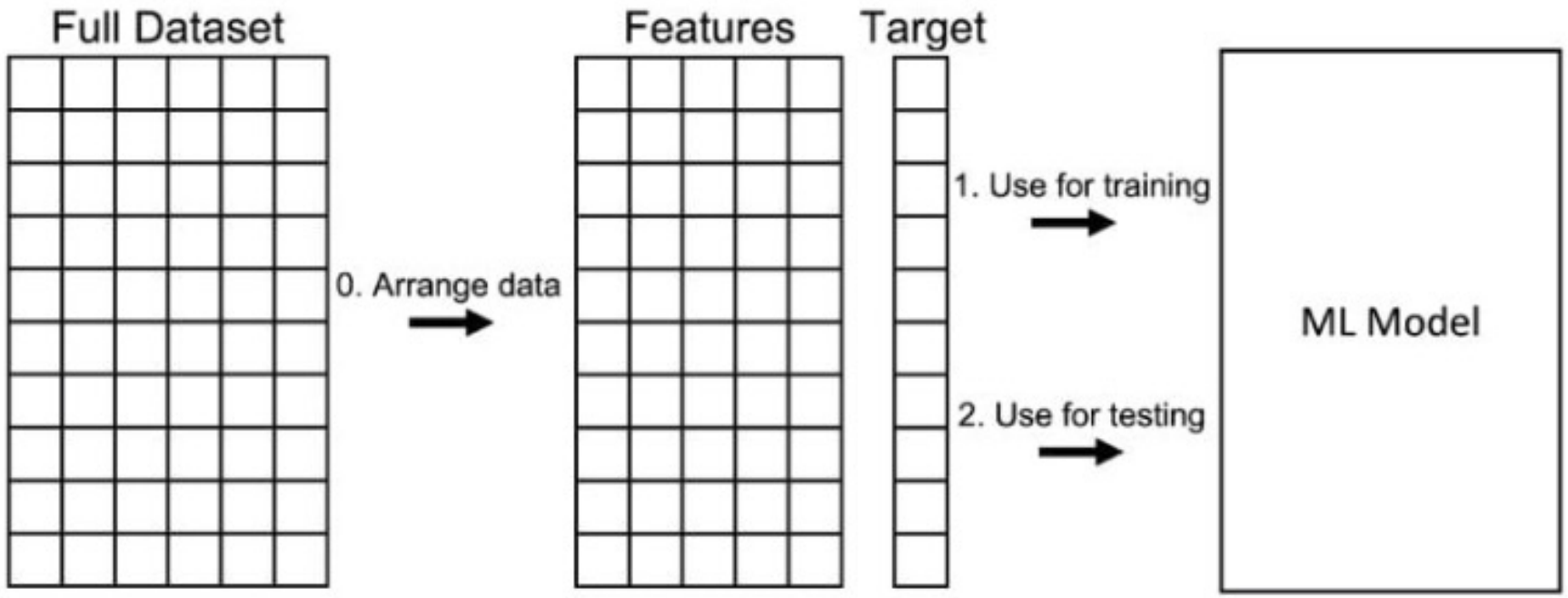


Train test split procedure. | Image: Michael Galarnyk

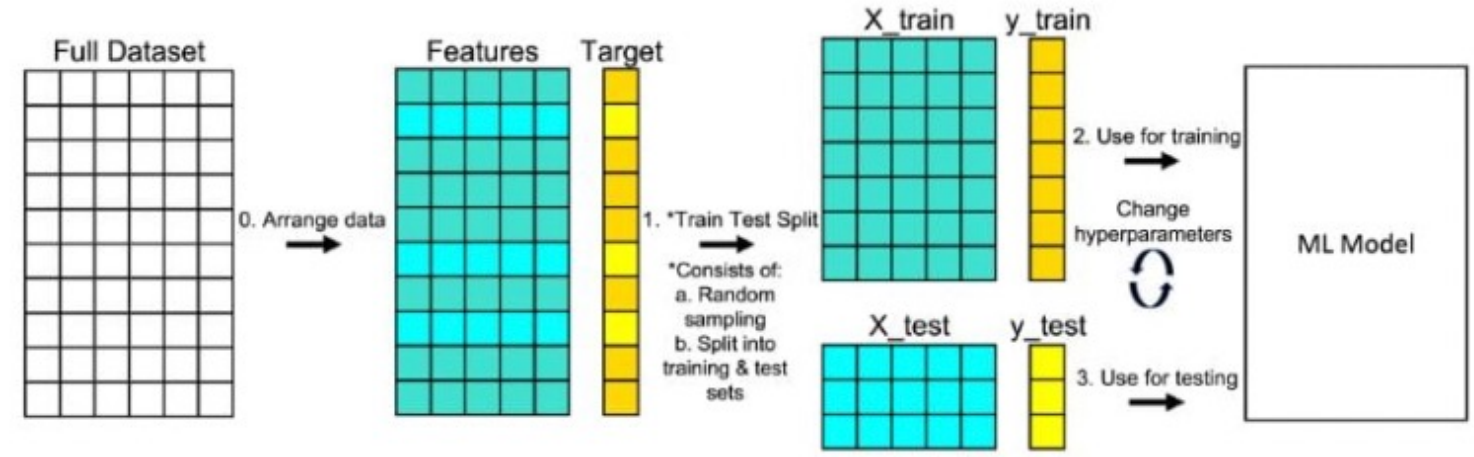
https://builtin.com/data-science/train-test-split

Consequences of Not Using Train Test Split

- You could try not using train test split and instead train and test the model on the same data. However, I don't recommend this approach as it doesn't simulate how a model would perform on new data. It also tends to reward overly complex models that overfit on the data set.



Using Train Test Split In Python



"Change hyperparameters" in this image is also known as hyperparameter tuning. | Image: Michael Galarnyk.

Python has a lot of libraries that help you accomplish your data science goals including scikit-learn, pandas, and NumPy, which the code below imports.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import tree
from sklearn.model_selection import train_test_split
```

LOAD THE DATA SET

```
df.info()
```

✓ 0.0s

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 6 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   bedrooms     21613 non-null  int64
 1   bathrooms    21613 non-null  float64
 2   sqft_living  21613 non-null  int64
 3   sqft_lot     21613 non-null  int64
 4   floors       21613 non-null  float64
 5   price        21613 non-null  float64
dtypes: float64(3), int64(3)
memory usage: 1013.2 KB
```

```
url = 'https://raw.githubusercontent.com/mGalarnyk/Tutorial_Data/master/King_County/
df = pd.read_csv(url)
# Selecting columns I am interested in
columns = ['bedrooms','bathrooms','sqft_living','sqft_lot','floors','price']
df = df.loc[:, columns]
df.head(10)
```

✓ 0.7s

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	price
0	3	1.00	1180	5650	1.0	221900.0
1	3	2.25	2570	7242	2.0	538000.0
2	2	1.00	770	10000	1.0	180000.0
3	4	3.00	1960	5000	1.0	604000.0
4	3	2.00	1680	8080	1.0	510000.0
5	4	4.50	5420	101930	1.0	1225000.0
6	3	2.25	1715	6819	2.0	257500.0
7	3	1.50	1060	9711	1.0	291850.0
8	3	1.00	1780	7470	1.0	229500.0
9	3	2.50	1890	6560	2.0	323000.0

ARRANGE DATA INTO FEATURES AND TARGET

Scikit-learn's train_test_split expects data in the form of features and target. In scikit-learn, a features matrix is a two-dimensional grid of data where rows represent samples and columns represent features. A target is what you want to predict from the data. This tutorial uses price as a target.

```
features = ['bedrooms','bathrooms','sqft_living','sqft_lot','floors']

X = df.loc[:, features]

y = df.loc[:, ['price']]
```

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	price
0	3	1.000000	1180	5650	1.000000	221900.000000
1	3	2.250000	2570	7242	2.000000	538000.000000
2	2	1.000000	770	10000	1.000000	180000.000000
3	4	3.000000	1960	5000	1.000000	604000.000000
4	3	2.000000	1680	8080	1.000000	510000.000000
5	4	4.500000	5420	101930	1.000000	1225000.000000
6	3	2.250000	1715	6819	2.000000	257500.000000
7	3	1.500000	1060	9711	1.000000	291850.000000
8	3	1.000000	1780	7470	1.000000	229500.000000
9	3	2.500000	1890	6560	2.000000	323000.000000

SPLIT DATA INTO TRAINING AND TESTING SETS

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	price
0	3	1.000000	1180	5650	1.000000	221900.000000
1	3	2.250000	2570	7242	2.000000	538000.000000
2	2	1.000000	770	10000	1.000000	180000.000000
3	4	3.000000	1960	5000	1.000000	604000.000000
4	3	2.000000	1680	8080	1.000000	510000.000000
5	4	4.500000	5420	101930	1.000000	1225000.000000
6	3	2.250000	1715	6819	2.000000	257500.000000
7	3	1.500000	1060	9711	1.000000	291850.000000
8	3	1.000000	1780	7470	1.000000	229500.000000
9	3	2.500000	1890	6560	2.000000	323000.000000

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0, train_size = .75)
```

In the code below, train_test_split splits the data and returns a list which contains four NumPy arrays, while train_size = 0.75 puts 75 percent of the data into a training set and the remaining 25 percent into a testing set.

SPLIT DATA INTO TRAINING AND TESTING SETS

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0, train_size = .75)
```

The image below shows the number of rows and columns the variables contain using the shape attribute before and after the `train_test_split`.

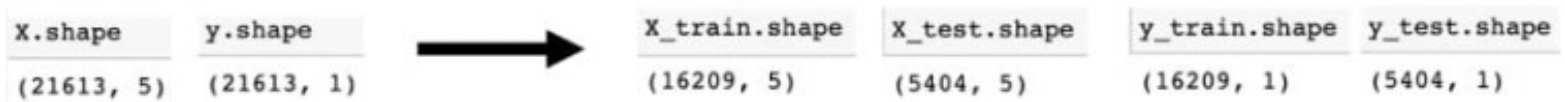
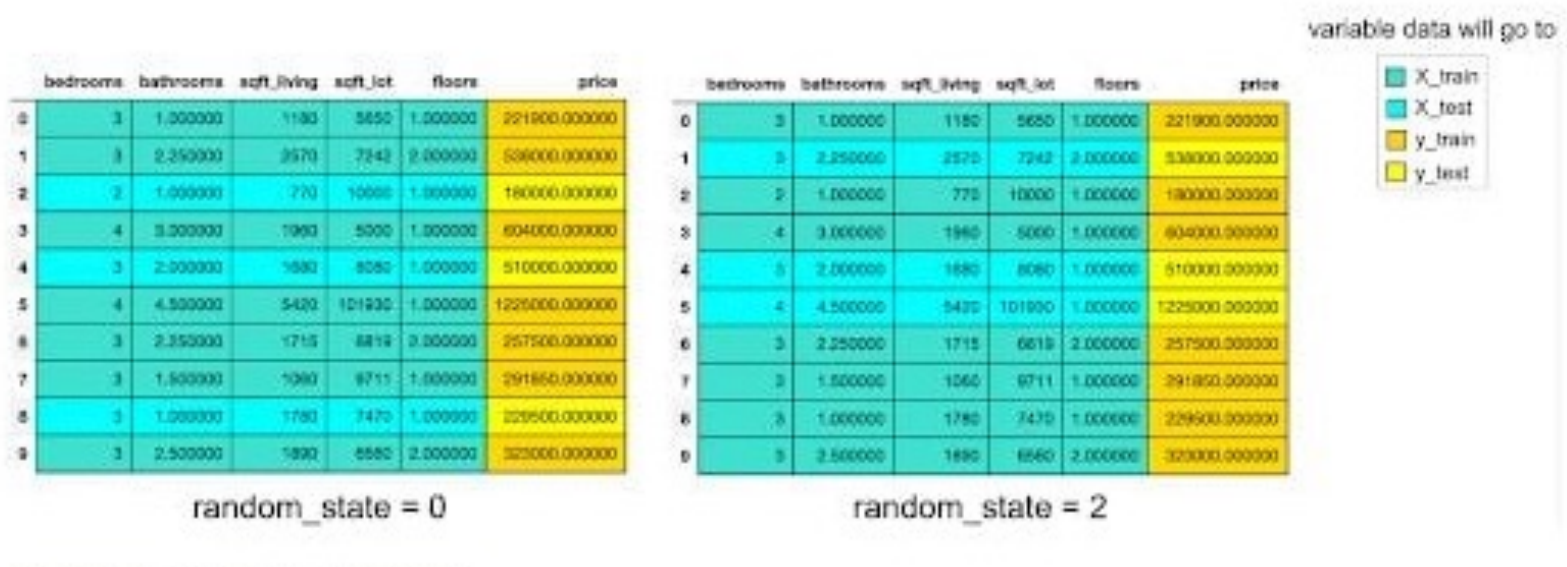


Image: Michael Galarnyk.

Shape before and after `train_test_split. 75` percent of the rows went to the training set `16209/ 21613 = .75` and 25 percent went to the test set `5404 / 21613 = .25`.

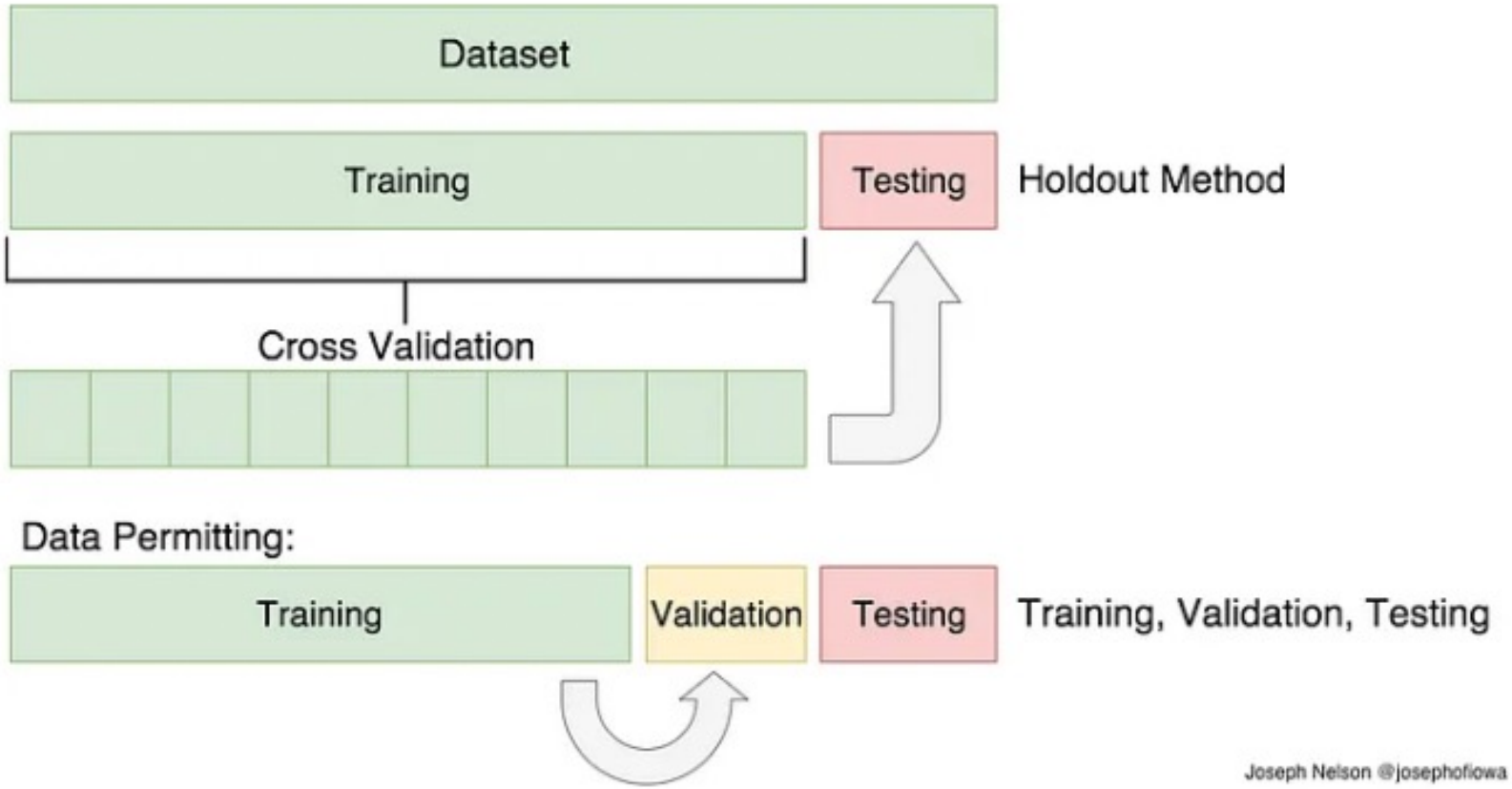
WHAT IS ‘RANDOM_STATE’ IN TRAIN TEST SPLIT?



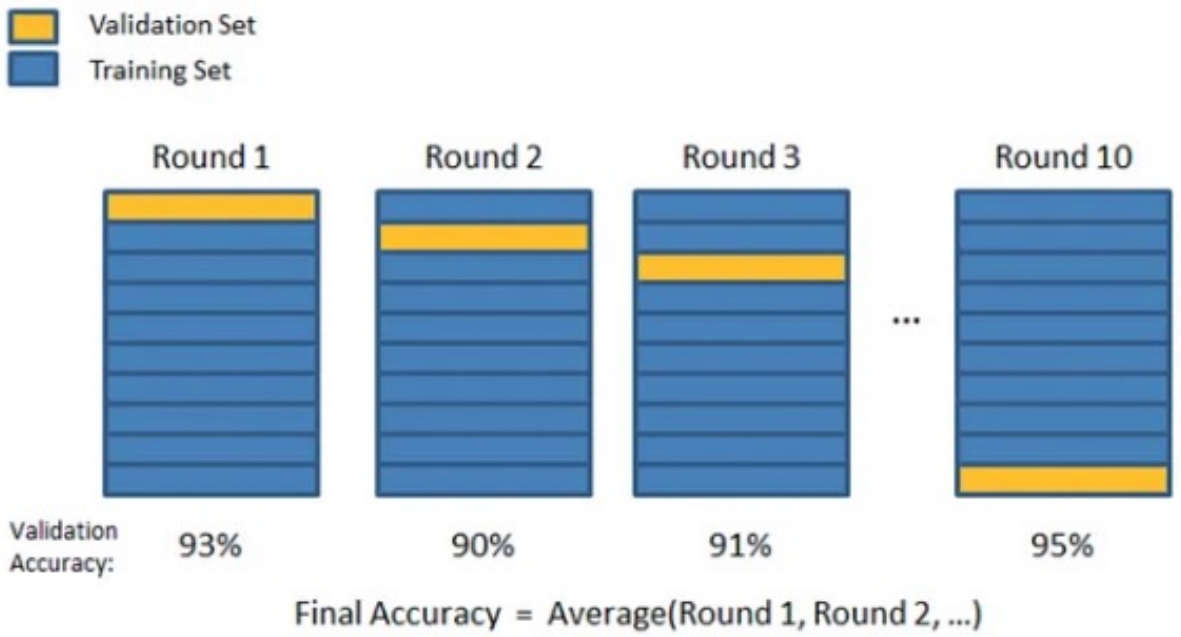
There are a number of reasons why people use random_state, including software testing, tutorials like this one and talks. However, it is recommended you remove it if you are trying to see how well a model generalizes to new data.

Cross Validation

- It’s very similar to train/test split, but it’s applied to more subsets. Meaning, we split our data into k subsets, and train on k-1 one of those subset. What we do is to hold the last subset for test. We’re able to do it for each of the subsets.



K-Folds Cross Validation



In K-Folds Cross Validation we split our data into k different subsets (or folds). We use k-1 subsets to train our data and leave the last subset (or the last fold) as test data. We then average the model against each of the folds and then finalize our model. After that we test it against the test set.

https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6

```python
from sklearn.model_selection import KFold # import KFold
X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]]) # create an array
y = np.array([1, 2, 3, 4]) # Create another array

kf = KFold(n_splits=2, random_state=None, shuffle=False)
```

```python
for train_index, test_index in kf.split(X):
    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]
```

✓ 0.0s

```
TRAIN: [2 3] TEST: [0 1]
TRAIN: [0 1] TEST: [2 3]
```