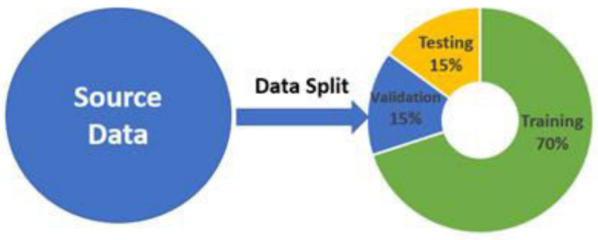
Session - 1

DATA SPLITING



One of the key aspects of supervised machine learning is model evaluation and validation. When we evaluate the predictive performance of wer model, it's essential that the process be unbiased. Using **train_test_split**() from the data science library scikit-learn, we can split wer dataset into subsets that minimize the potential for bias in wer evaluation and validation process.

What students learn?

- Why we need to **split wer dataset** in supervised machine learning
- Which **subsets** of the dataset we need for an unbiased evaluation of wer model
- How to use **train_test_split()** to split wer data
- How to combine train_test_split() with **prediction methods**

6.1 The Importance of Data Splitting

Supervised machine learning is about creating models that precisely map the given inputs (independent variables, or **predictors**) to the given outputs (dependent variables, or **responses**).

How we measure the precision of model depends on the type of a problem we're trying to solve. In regression analysis, we typically use the coefficient of determination, root-mean-square error, mean absolute error, or similar quantities. For classification problems, we often apply accuracy, precision, recall, F1 score, and related indicators.The acceptable numeric values that measure precision vary from field to field.

What's most important to understand is that the need to **unbiased evaluation** of properly use these measures, assess the predictive performance of wer model, and validate the model.

This means that one can't evaluate the predictive performance of a model with the same data that is used for training. This needs to evaluate the model with **fresh data** that hasn't been seen by the model before. Hence this can accomplish that by splitting the dataset before using it.

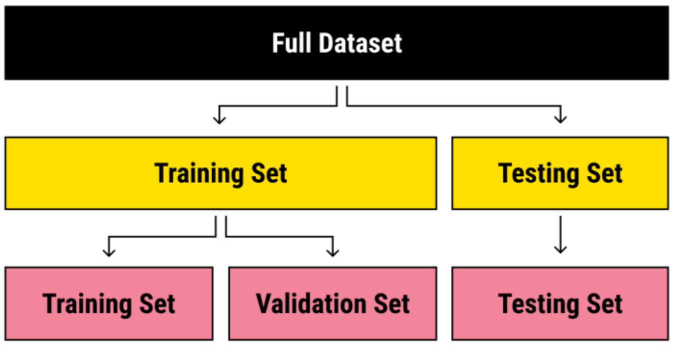
6.2. What is Data Splitting?

Data splitting is when data is divided into two or more subsets. Typically, with a two-part split, one part is used to evaluate or test the data and the other to train the model.

Data splitting is an important aspect of data science, particularly for creating models based on data. This technique helps ensure the creation of data models and processes that use data models -- such as machine learning -- are accurate.

6.3 Training, Validation, and Test Sets

Splitting wer dataset is essential for an unbiased evaluation of prediction performance. In most cases, it's enough to split wer dataset randomly into three subsets:



1. **The training set** is applied to train, or **fit**, in the model. For example, we use the training set to find the optimal weights, or coefficients, for linear regression, logistic regression, or neural networks.

2. **The validation set** is used for unbiased model evaluation during hyperparameter tuning. For example, when we want to find the optimal number of neurons in a neural network or the best kernel for a support vector machine, we experiment with different values. For each considered setting of hyperparameters, we fit the model with the training set and assess its performance with the validation set.
3. **The test set** is needed for an unbiased evaluation of the final model. We shouldn't use it for fitting or validation.

In less complex cases, when we don't have to tune hyperparameters, it's okay to work with only the training and test sets.

6.4 Underfitting and Overfitting

Splitting a dataset might also be important for detecting if wer model suffers from one of two very common problems, called underfitting and overfitting:

1. **Underfitting** is usually the consequence of a model being unable to encapsulate the relations among data. For example, this can happen when trying to represent nonlinear relations with a linear model. Underfitted models will likely have poor performance with both training and test sets.
2. **Overfitting** usually takes place when a model has an excessively complex structure and learns both the existing relations among data and noise. Such models often have bad generalization capabilities. Although they work well with training data, they usually yield poor performance with unseen (test) data.

6.5 Practice: split training and testing data sets in Python using train_test_split() of sci-kit learn.

Prerequisites for Using train_test_split()

First install version 0.23.1 of **scikit-learn**, or **sklearn**. It has many packages for data science and machine learning, but for this tutorial we willl focus on the **model_selection** package, specifically on the function **train_test_split()**.

```
$ python -m pip install -U "scikit-learn==0.23.1"
```

In Anaconda, this package is already installed. However, if we want to use a fresh environment, ensure that we have the specified version, or use Miniconda, then we can install sklearn from Anaconda Cloud with conda install:

```
$ conda install -c anaconda scikit-learn=0.23
```

Application of train_test_split()

we need to import train_test_split() and NumPy before we can use them, so we can start with the import statements:

```
Python

>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
```

Now that we have both imported, we can use them to split data into training sets and test sets. We'll split inputs and outputs at the same time, with a single function call.

With train_test_split(), we need to provide the sequences that we want to split as well as any optional arguments. It returns a list of NumPy arrays, other sequences, or SciPy sparse matrices if appropriate:

```
Python

sklearn.model_selection.train_test_split(*arrays, **options) -> list
```

arrays is the sequence of lists, NumPy arrays, pandas DataFrames, or similar array-like objects that hold the data we want to split. All these objects together make up the dataset and must be of the same length.

In supervised machine learning applications, we'll typically work with two such sequences:

1. A two-dimensional array with the inputs (x)
2. A one-dimensional array with the outputs (y)

options are the optional keyword arguments that we can use to get desired behavior:

- **train_size** is the number that defines the size of the training set. If we provide a float, then it must be between 0.0 and 1.0 and will define the share of the dataset used for testing. If we provide an int, then it will represent the total number of the training samples. The default value is None.
- **test_size** is the number that defines the size of the test set. It's very similar to train_size. We should provide either train_size or test_size. If neither is given, then the default share of the dataset that will be used for testing is 0.25, or 25 percent.
- **random_state** is the object that controls randomization during splitting. It can be either an int or an instance of RandomState. The default value is None.
- **shuffle** is the Boolean object (True by default) that determines whether to shuffle the dataset before applying the split.
- **stratify** is an array-like object that, if not None, determines how to use a stratified split.

Now it's time to try data splitting! We'll start by creating a simple dataset to work with. The dataset will contain the inputs in the two-dimensional array x and outputs in the one-dimensional array y:

Python

```
>>> x = np.arange(1, 25).reshape(12, 2)
>>> y = np.array([0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0])
>>> x
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [11, 12],
       [13, 14],
       [15, 16],
       [17, 18],
       [19, 20],
       [21, 22],
       [23, 24]])
>>> y
array([0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0])
```

To get wer data, we use arange(), which is very convenient for generating arrays based on numerical ranges. We also use .reshape() to modify the shape of the array returned by arange() and get a two-dimensional data structure.

We can split both input and output datasets with a single function call:

Python

```
>>> x_train, x_test, y_train, y_test = train_test_split(x, y)
>>> x_train
array([[15, 16],
       [21, 22],
       [11, 12],
       [17, 18],
       [13, 14],
       [ 9, 10],
       [ 1,  2],
       [ 3,  4],
       [19, 20]])
>>> x_test
array([[ 5,  6],
       [ 7,  8],
       [23, 24]])
>>> y_train
array([1, 1, 0, 1, 0, 1, 0, 1, 0])
>>> y_test
array([1, 0, 0])
```

Given two sequences, like x and y here, train_test_split() performs the split and returns four sequences (in this case NumPy arrays) in this order:

1. **x_train:** The training part of the first sequence (x)
2. **x_test:** The test part of the first sequence (x)
3. **y_train:** The training part of the second sequence (y)
4. **y_test:** The test part of the second sequence (y)

We probably got different results from what we see here. This is because dataset splitting is random by default. The result differs each time we run the function. However, this often isn’t what we want.

Sometimes, to make wer tests reproducible, we need a random split with the same output for each function call. We can do that with the parameter random_state. The value of random_state isn't important—it can be any non-negative integer. We could use an instance of numpy.random. RandomState instead, but that is a more complex approach.

In the previous example, we used a dataset with twelve observations (rows) and got a training sample with nine rows and a test sample with three rows. That's because we didn't specify the desired size of the training and test sets. By default, 25 percent of samples are assigned to the test set. This ratio is generally fine for many applications, but it's not always what we need.

Typically, we'll want to define the size of the test (or training) set explicitly, and sometimes we'll even want to experiment with different values. We can do that with the parameters train_size or test_size.

Modify the code so we can choose the size of the test set and get a reproducible result:

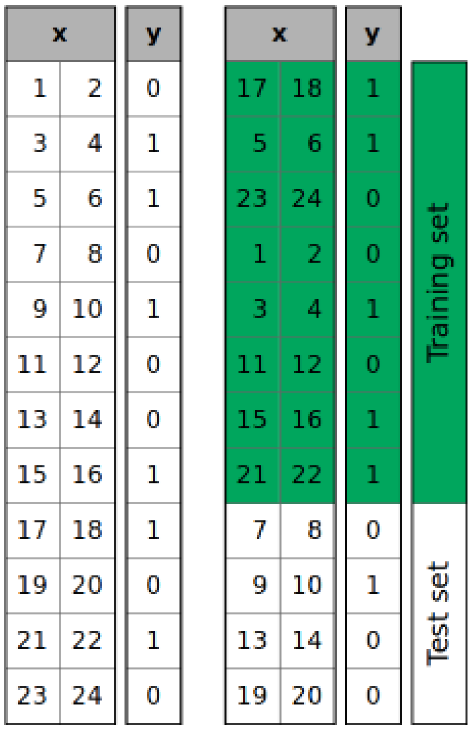
```
Python

>>> x_train, x_test, y_train, y_test = train_test_split(
...     x, y, test_size=4, random_state=4
... )
>>> x_train
array([[17, 18],
       [ 5,  6],
       [23, 24],
       [ 1,  2],
       [ 3,  4],
       [11, 12],
       [15, 16],
       [21, 22]])
>>> x_test
array([[ 7,  8],
       [ 9, 10],
       [13, 14],
       [19, 20]])
>>> y_train
array([1, 1, 0, 0, 1, 0, 1, 1])
>>> y_test
array([0, 1, 0, 0])
```

With this change, we get a different result from before. Earlier, we had a training set with nine items and test set with three items. Now, thanks to the argument test_size=4, the training set has eight items and the test set has four items. We'd get the same result with test_size=0.33 because 33 percent of twelve is approximately four.

There's one more very important difference between the last two examples: We now get the same result each time we run the function. This is because we've fixed the random number generator with random_state=4.

The figure below shows what's going on when we call train_test_split():



x		y
1	2	0
3	4	1
5	6	1
7	8	0
9	10	1
11	12	0
13	14	0
15	16	1
17	18	1
19	20	0
21	22	1
23	24	0

x		y	
17	18	1	Training set
5	6	1	
23	24	0	
1	2	0	
3	4	1	
11	12	0	
15	16	1	
21	22	1	
7	8	0	Test set
9	10	1	
13	14	0	
19	20	0	

The samples of the dataset are shuffled randomly and then split into the training and test sets according to the size we defined.

We can see that y has six zeros and six ones. However, the test set has three zeros out of four items. If we want to (approximately) keep the proportion of y values through the training and test sets, then pass stratify=y. This will enable stratified splitting:

Python

```
>>> x_train, x_test, y_train, y_test = train_test_split(
...     x, y, test_size=0.33, random_state=4, stratify=y
... )
>>> x_train
array([[21, 22],
       [ 1,  2],
       [15, 16],
       [13, 14],
       [17, 18],
       [19, 20],
       [23, 24],
       [ 3,  4]])
>>> x_test
array([[11, 12],
       [ 7,  8],
       [ 5,  6],
       [ 9, 10]])
>>> y_train
array([1, 0, 1, 0, 1, 0, 0, 1])
>>> y_test
array([0, 0, 1, 1])
```

Now y_train and y_test have the same ratio of zeros and ones as the original y array.

Stratified splits are desirable in some cases, like when we're classifying an **imbalanced dataset**, a dataset with a significant difference in the number of samples that belong to distinct classes. Finally, we can turn off data shuffling and random split with shuffle=False:

Python

```
>>> x_train, x_test, y_train, y_test = train_test_split(
...     x, y, test_size=0.33, shuffle=False
... )
>>> x_train
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [11, 12],
       [13, 14],
       [15, 16]])
>>> x_test
array([[17, 18],
       [19, 20],
       [21, 22],
       [23, 24]])
>>> y_train
array([0, 1, 1, 0, 1, 0, 0, 1])
>>> y_test
array([1, 0, 1, 0])
```

Now we have a split in which the first two-thirds of samples in the original x and y arrays are assigned to the training set and the last third to the test set. No shuffling. No randomness.