Chapter 1

**Data Preprocessing**

**1.1 Python and R installation**

1. Install python with Anaconda(IDE).
2. Install R with R-studio(IDE).

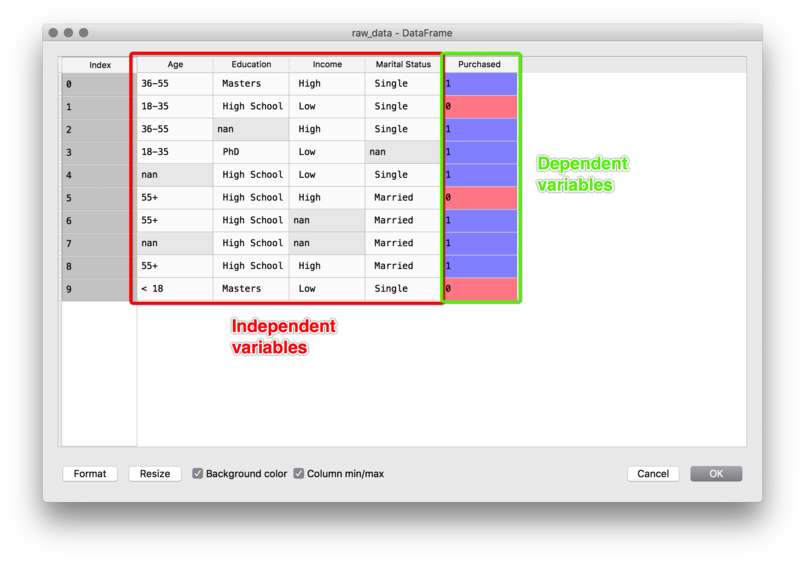
**1.2 Extract Given data sets and Codes and file tree**

1. Extract All examples.
2. Extract All codes and Datasets.
3. Extract the given file tree.

**1.3 INDEPENDENT and DEPENDENT Variables in ML**

***Independent*** variables (also referred to as Features) are the input for a process that is being analyzes. ***Dependent*** variables are the output of the process.

* For example, in the below data set, the ***independent*** variables are the input of the purchasing process being analyzed. The result (whether a user purchased or not) is the ***dependent*** variable.
* Using ***Independent*** variables, we PREDICT ***Dependent*** variables.



**1.4 Importing the Libraries**

* Three essential libraries: Use this kind of command to install the libraries: Eh: to install *matplotlib*,

*pip install matplotlib*

#*Importing the libraries*

**import** numpy **as** np

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

1. NumPy: Contains mathematical tools. Since of course machine learning models are based on mathematics then we will absolutely need ***NumPy*** from time to time to do them but *not all the time* you'll see.
2. matplotlib: *Matplotlib* is a comprehensive library for creating static, animated, and interactive visualizations in Python. We're gonna use the module ***pyplot*** of ***matplotlib*** library.

* module matplotlib.pyplot: ***matplotlib.pyplot*** is a state-based interface to ***matplotlib***. It provides a ***MATLAB-like*** way of plotting.
* *pyplot* is mainly intended for *interactive plots* and simple cases of *programmatic plot generation*:
* The object-oriented API is recommended for more complex plots.

**import** numpy **as** np

**import** matplotlib**.**pyplot **as** plt

x = np**.arange**(0, 5, 0.1)

y = np**.sin**(x)

plt**.plot**(x, y)

plt**.show**()

1. pandas: Used for import and manage datasets. pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

**1.5 Importing the Dataset**

* Use the ide Spider: ***Spider*** in ***Anaconda***. Enable panes: Editor, ipython console, Variable explorer, object inspector (aka Help).
* ***Ctrl+enter*** to execute.
* Pandas intro: Recall 100-days-of-code.
* ***DataFrame.iloc***

Purely integer-location based indexing for selection by position. ***.iloc[]*** is primarily integer position based (from ***0*** to ***length-1*** of the axis), but may also be used with a boolean array.

Indexing just the rows:

With a scalar integer: df.iloc[0]

With a list of integers: df.iloc[[0]]

df.iloc[[0, 1]]

With a slice object: df.iloc[:3]

Indexing both axes: You can mix the indexer types for the index and columns. Use ***:*** to select the ***entire*** ***axis***.

With scalar integers: df.iloc[0, 1]

With lists of integers. df.iloc[[0, 2], [1, 3]]

With slice objects: df.iloc[1:3, 0:3]

* Matrix of features and dependent variable vector:
* Matrix of features: ***Matrix*** of independent variables. "**:**" is for slicing.

x = dataset.**iloc**[:, :-1]

* "**:-1**" is to all columns excluding last column
* "**:**" to select all rows
* Dependent variable vector: ***Vector*** of last column as dependent variable.

y = dataset.**iloc**[:, 3]

#*Importing the libraries*

**import** numpy **as** np

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

#*x = np.arange(0, 5, 0.1)*

#*y = np.sin(x)*

#*plt.plot(x, y)*

#*plt.show()*

#*importing the dataset*

dataset = pd**.read\_csv**("Data.csv")

**print**(dataset)

#*":-1" is to all columns excluding last column*

#*":" to select all rows*

x = dataset**.**iloc[:, :-1]

**print**(x)

y = dataset**.**iloc[:, 3]

**print**(y)

#*python prcts\_dt\_prep.py*

**1.6 Missing Data**

Install library ***sklearn***. Import ***imputer*** from ***preprocessing*** module.

*pip install -U scikit-learn*

* Use Mean: Eliminate ***NaN***.

class sklearn.preprocessing.Imputer(missing\_values='NaN', strategy='mean', axis=0, verbose=0, copy=True)

#*----------- For older version -------------*

#*from sklearn.preprocessing import Imputer*

#*imptr = Imputer()*

* `***SimpleImputer***` replaces the previous `***sklearn.preprocessing.Imputer***` estimator which is now removed.

**class** SimpleImputer(**self**, missing\_values=np**.**nan, strategy="mean", fill\_value=**None**, verbose=0, copy=**True**, add\_indicator=**False**)

Parameters

***missing\_values :*** number, string, np.nan (default) or None

The placeholder for the missing values. All occurrences of `missing\_values` will be imputed. For pandas' dataframes with nullable integer dtypes with missing values, `missing\_values` should be set to `np.nan`, since `pd.NA` will be converted to `np.nan`.

***strategy :*** string, default='mean'

The imputation strategy.   
  
- If "mean", then replace missing values using the mean along each column. Can only be used with numeric data.   
- If "median", then replace missing values using the median along each column. Can only be used with numeric data.   
- If "most\_frequent", then replace missing using the most frequent value along each column. Can be used with strings or numeric data.   
- If "constant", then replace missing values with fill\_value. Can be used with strings or numeric data.   
  
***strategy="constant"*** for fixed value imputation.

***fill\_value :*** string or numerical value, default=None

When strategy == "constant", fill\_value is used to replace all occurrences of missing\_values.   
If left to the default, fill\_value will be 0 when imputing numerical data and "missing\_value" for strings or object data types.

#*------------------ Data Preprocessing --------------------*

#*Importing the libraries*

**import** numpy **as** np

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

#*x = np.arange(0, 5, 0.1)*

#*y = np.sin(x)*

#*plt.plot(x, y)*

#*plt.show()*

#*importing the dataset*

dataset = pd**.read\_csv**("Data.csv")

**print**(dataset)

#*":-1" is to all columns exluding last column*

#*":" to select all rows*

old\_x = dataset**.**iloc[:, :-1]

old\_y = dataset**.**iloc[:, 3]

**print**("Before\n",old\_x)

#*Taking care of missing data*

#*import sklearn*

#*print(sklearn.\_\_version\_\_)*

**from** sklearn**.**impute **import** SimpleImputer

imptr = **SimpleImputer**(missing\_values= np**.**nan, strategy="mean") #*"NaN" connot be used. Use np.nan*

#*fix the 2nd and 3rd column*

impt = imptr**.fit**(old\_x**.**iloc[:, 1:3])

**print**(impt) #*it is a method, acts on feature matrix*

old\_x**.**iloc[:, 1:3] = impt**.transform**(old\_x**.**iloc[:, 1:3])

**print**("After\n",old\_x)

#*----------- For older version -------------*

#*from sklearn.preprocessing import Imputer*

#*imptr = Imputer(missing\_values='NaN', strategy='mean', axis=0, verbose=0, copy=True)*

* Note: Slice list in python: Index ***-1*** represents the ***last*** element and ***-n*** represents the ***first*** element of the list(considering n as the length of the list). Lists can also be manipulated using negative indexes also.

|  |  |
| --- | --- |
| Lst[ Initial : End : IndexJump ]  #*List slicing*  #*Initialize list*  Lst = [50, 70, 30, 20, 90, 10, 50]  #*Display list*  **print**(Lst[::])  #*Negative indexing, reverse*  **print**(Lst[-7::1])  #*Display list portion*  **print**(Lst[1:5]) | D:\100_days_comer_soln\Screenshot584.png |

* NOTE: Use power shell **cd** with "" to consider it as a *string* with *spaces*.

CD "L:\2\_Code\_source\_1\ML\_a2z\_cods\_dtsts\Part 1 - Data Preprocessing\Section 2 -------------------- Part 1 - Data Preprocessing --------------------\Python"

**1.7 Categorical Data : Encode texts into numbers**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Country | Age | Salary | Purchased | |
| France |  | 72000 | No |  |
| Spain | 27 | 48000 | Yes |  |
| Germany | 30 | 54000 | No |  |
| Spain | 38 | 61000 | No |  |
| Germany | 40 |  | Yes |  |
| France | 35 | 58000 | Yes |  |
| Spain |  | 52000 | No |  |
| France | 48 | 79000 | Yes |  |
| Germany | 50 | 83000 | No |  |
| France | 37 | 67000 | Yes |  |
|  |  |  |  |  |

* In this dataset we can see that we have two categorical variables, ***country*** and ***purchase***. These two variables are categorical variables because they contain categories.
* Here the ***country*** contains three categories. It's France Spain and Germany. And the ***purchase*** variable contains two categories. ***Yes*** and ***No*** that's why they're called categorical variables.
* But ML models works with numbers, so we need to encode these categories into numerical values. Since machine learning models are based on mathematical equations you can intuitively understand that it would cause some problem if we keep the text here and the categorical variables in the equations because we would only want numbers in the equations.
* So that's why we need to encode the categorical variables. That is to encode the text that we have here into numbers.

**from** sklearn**.**preprocessing **import** LabelEncoder, OneHotEncoder

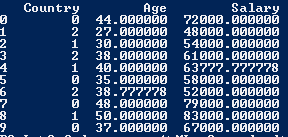
label\_encode\_x = **LabelEncoder**()

old\_x**.**iloc[:, 0] = label\_encode\_x**.fit\_transform**(old\_x**.**iloc[:, 0])

#*one\_hot\_encode = OneHotEncoder(categories= old\_x[0])*

#*old\_x = one\_hot\_encode.fit\_transform(old\_x).toarray()*

**print**(old\_x)



* Labelencoder: If only two categories occurs, like yes-no or on-off we then use "labelencoder" give 0 and 1.

#*Encode dependent vector Column*

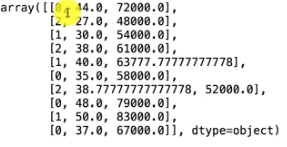
**from** sklearn**.**preprocessing **import** LabelEncoder

label\_encode = **LabelEncoder**()

old\_y = label\_encode**.fit\_transform**(old\_y)

**print**(old\_y)

* But for more than two categories, " **LabelEncoder** " returns 0, 1, 2, 3, . . . . This can cause error. For example ***country*** is conceded to 0, 1, 2.
* Problem: The problem is *there is unwanted numerical-orders*. France is 0, Spain is 2 and Germany is 1. But there is no relationship order between these three countries France Germany and Spain.
* So we want to avoid the model to have such an interpretation because that could cause some misinterpreted correlations between the ***features*** and the ***outcome*** which we want to ***predict***.



* DUMMY-Encoding (use **OneHotEncoder**) Dummy variables: Don't use LabelEncoder to transform categorical into numerical, just use ***OneHotEncoder***.
* Using LabelEncoder introduces a problem of data ordering, when your categorical data gets encoded into let's say 1,2,3 etc. and for model this is a clear ordering 1 < 2 < 3, however in reality with categorical features this is not true. For this reason we introduce three dummy variables for these counties, 1 represents the occurrence of the county in a row.
* So Country column into three columns, because there are actually three different classes in this country column you know three different categories (countries).
* If there were for example five countries here we would turn this column into five columns and ***OneHotEncoder*** consists of creating binary vectors for each of the countries.
* Legacy codes not supported anymore:

Legacy Code

#*Encoding the Independent Variable*

**from** sklearn**.**preprocessing **import** LabelEncoder, OneHotEncoder

labelencoder\_X = **LabelEncoder**()

X[:, 0] = labelencoder\_X**.fit\_transform**(X[:, 0])

onehotencoder = **OneHotEncoder**(categorical\_features = [0])

X = onehotencoder**.fit\_transform**(X)**.toarray**()

For Newer Version

**from** sklearn**.**preprocessing **import** LabelEncoder, OneHotEncoder

**from** sklearn**.**compose **import** ColumnTransformer

#*Encode Country Column*

ct = **ColumnTransformer**(transformers = [("encoding", **OneHotEncoder**(), [0])], remainder = 'passthrough')

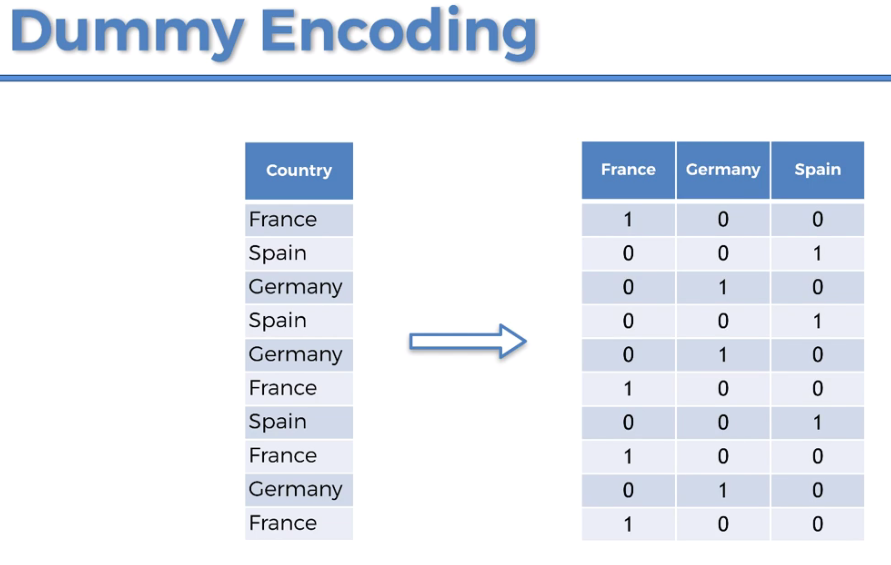
#*remainder = 'passthrough' for remaining columns to be unchanged*

old\_x = ct**.fit\_transform**(old\_x) #*convert this output to NumPy array*

**print**(old\_x)

old\_x = np**.array**(old\_x)

**print**(old\_x)



|  |  |
| --- | --- |
|  |  |

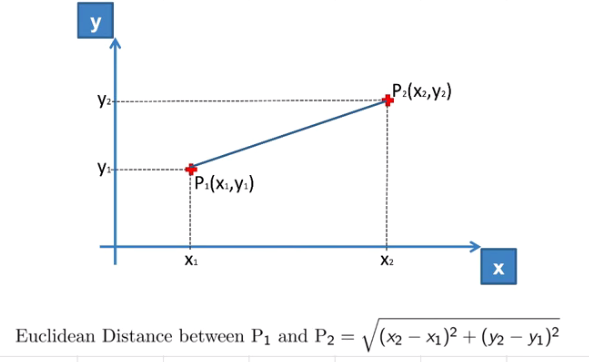
**1.8 Feature Scaling**

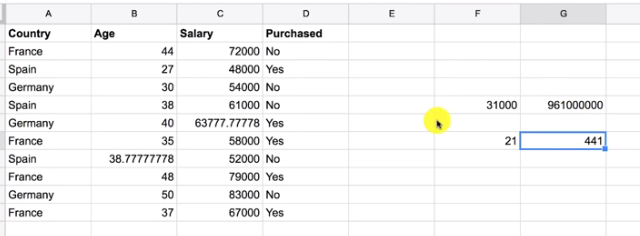
Why Feature Scaling ?

Let's just focus on the age and the salary. You notice that the variables are not on the same scale because the age are going from 27 to 50.

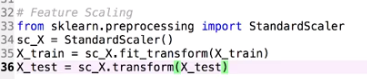
And the salaries going from 40 K to like 90 K. Hence this age variable in the salary variable don't have the same scale.

* This will cause some issues in your machinery models. Because a lot of ***ML model*** are based on the Euclidean distance. The Euclidean distance between two points is the square root of the sum of the square coordinates.





* Now in our dataset consider the 9th and the third rows. Salary difference is 31000 and age difference is 21. Their squares are 96100000 and 441.
* So you can see very clearly how this square difference 96100000 dominates this square difference 441.
* And that's because these two variables are not on the same scale.
* So that's why we absolutely need to put the variables in the same scale. And going to have values in the same range.
* fit and transform for train-data only transform test-data: We're calling the method fit and transform for train-data but we will only transform the test set because it will automatically fitted during train-data (it's already fitted to the training set).



* Do we need to fit and transform the DUMMY VARIABLES: It depends on the context. It depends on how much you want to keep interpretation in your models. Because if we scale this it will be good because everything will be on the same scale we will be happy with that and it will be good for our predictions but who will lose the interpretation of knowing which observations belongs to which country etc..
* So as you want it won't break your model if you don't scale the dummy variables because there will be actually on the same scale as the future scales.
* Do we need to apply features scaling to the DEPENDENT VARIABLE VECTOR: In our case it is a categorical variable because it's taking only two values 0 and one.

We don't need to do it because this is a classification problem with a category called dependent variable.

But you will see that for regression when the dependent variable will take a huge range of values. We will need to apply feature scaling to the dependent variable as well.

* Feature Scaling for other models: Even if sometimes machine models are not based on ***Euclidean*** ***distances*** we will still need to do ***features scaling*** because the ALGORITHM will converge much faster. That will be the case for decision trees. Decision trees are *not based on Euclidean distances* but you will see that we will need to do feature scaling because if we don't do it they will run for a very long time.

#*---------------------- Feature Scaling ------------------------*

**from** sklearn**.**preprocessing **import** StandardScaler

sc = **StandardScaler**()

old\_x = sc**.fit\_transform**(old\_x)

**print**(old\_x)

* Feature Scaling: Feature Scaling is a technique to *standardize* the *independent* *features* present in the data in a fixed range. It is performed during the *data pre-processing* to handle *highly* *varying* *magnitudes* or values or units. If feature scaling is not done, then a machine learning algorithm tends to *weigh* *greater* values, *higher* and consider *smaller* *values* as the *lower* values, regardless of the unit of the values.
* Example: If an algorithm is not using the feature scaling method then it can consider the value 3000 meters to be greater than 5 km but that’s actually not true and in this case, the algorithm will give wrong predictions. So, we use Feature Scaling to bring all values to the same magnitudes and thus, tackle this issue.

|  |  |
| --- | --- |
|  |  |

* What is Feature Scaling and why do we have to use it for: Feature Scaling is a technique that will put all your features in the same range.
* If we have a look again at our data well we can clearly see that the values of the ***age*** feature are not in the same range as the values of the ***salary*** feature.
* The ***ages*** from like ***0*** to ***100*** and ***salary*** is from ***0*** to ***100,000***. So these two features are not in the same range and feature scaling consist of putting the values of these two features in the same range.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Country*** | ***Age*** | ***Salary*** | ***Purchased*** | |
| France |  | 72000 | No |  |
| Spain | 27 | 48000 | Yes |  |
| Germany | 30 | 54000 | No |  |
| Spain | 38 | 61000 | No |  |
| Germany | 40 |  | Yes |  |
| France | 35 | 58000 | Yes |  |
| Spain |  | 52000 | No |  |
| France | 48 | 79000 | Yes |  |
| Germany | 50 | 83000 | No |  |
| France | 37 | 67000 | Yes |  |

* Why we have to apply FEATURE SCALING: Well that's because when training our ML models (some of them not all of them), if you have some features that are taking much higher values than other features this can create a bias in the *correlations computations*.
* In other words the ***features*** that have ***high*** ***values*** compared to the other ones will ***dominate*** the other ***features*** so that the ***other*** ***features*** might not be ***considered***.
* However the other features might be the ones having the largest impact on the dependent variable.
* You know the ones that actually contribute the most to predicting the outcome.
* The dependent variable and that's why it's very very important to put all the features in the same scale.

|  |  |
| --- | --- |
|  | is the mean and  is the standard deviation. |

* Not all ML models needs feature scaling: You will see when we start building ML models, sometimes we will apply ***feature*** ***scaling*** and sometimes not. That's because some ML models models actually ***automatically*** ***understand*** that some ***features*** have indeed values lower than the others and they will actually fix that automatically by playing with the coefficients.
* For example consider linear regression. ***Linear*** ***regression*** has some ***coefficients*** for each of the ***features*** and for the features having ***super*** ***high*** ***values*** it will just compensate with a ***very*** ***low*** ***coefficient***.
* So it's fine that we won't have to apply Features Scaling for ***linear*** ***regression***
* But you will see that for other models like Logistic Regression and SVR we have to apply Feature Scaling.
* Bounding Range: Normalization has a bounding range in (0, 1). Unlike normalization, standardization does not have a bounding range, however in many cases it may be in between (-3, 3).
* The Big Question – NORMALIZE or STANDARDIZE?

Normalization vs. standardization is an eternal question among machine learning newcomers. Let me elaborate on the answer in this section.

* ***Normalization*** is good to use when you know that the distribution of your data does not follow a Gaussiandistribution. This can be useful in *algorithms* that *do not assume any distribution* of the data like ***K-Nearest Neighbors*** and ***Neural*** ***Networks***.
* ***Standardization***, on the other hand, can be helpful in cases where the data follows a Gaussian distribution. However, this does not have to be ***necessarily*** ***true***. Also, unlike normalization, standardization does not have a Bounding Range. So, even if you have outliers in your data, they will not be ***affected*** by standardization.

However, at the end of the day, the choice of using normalization or standardization will depend on your problem and the machine learning algorithm you are using. There is no hard and fast rule to tell you when to normalize or standardize your data. You can always start by fitting your model to raw, normalized and standardized data and compare the performance for best results.

* How to apply:

**from** sklearn**.**preprocessing **import** StandardScaler

sc = **StandardScaler**()

Will automatically apply .

old\_x = sc**.fit\_transform**(old\_x)

Will scale our features.

**1.9 Splitting the Dataset into the Training-set and Test-set**

In machine your algorithm or model is going to learn from your data to make predictions or other machine learning goals. So Machine Learning Model is going to learn to do something on your data set by *understanding* some *correlations* that there is in your *dataset* and *imagine* your *machine* *learning* *model* is learning too much on the data set like it's learning *too much to correlations* then I'm not sure it's performance would be great.

We are going to build our *machine learning models* on a data set but then we have to test it on a new set which is going to be slightly different from the dataset on which we build the machine model.

So we have to make *two different sets of* TrainingSets on which we build the ***ML model*** and a Test Set on which we *test* the *performance* of this ***ML model*** and the performance on the test set *shouldn't be much different* from the performance on the training sets because this would mean that the ***ML model*** understood well the correlations and didn't learn them by heart so that you can *adapt* to *new* *sets* and *new* *situations*.

So that's the idea about splitting the dataset into a ***training*** ***set*** and a ***test*** ***set***.

Learn from train\_data (80%) and apply to test\_data (20%): We are building our ***ML model*** on this Test Set by establishing some correlations between the *independent* *variables* here and the *dependent* *variable* of the Test Set.

And then once the machine learning model understands the correlations between *independent* *variables* and the *dependent* *variable* we will test ***ML model*** that it can apply the correlations based on the *training* *set* on the *test* *set*.

That means that we will see that ***ML model*** can predict about the *dependent* *variable* on the *teat* *dataset* (which is 20% in our case).

If ML model learn from 100% from the data it didn't understand quite well the logic and won't be able to make good predictions. That's called overfitting we will be talking about that in further details in the regression section and we will learn how to use REGULARISATION techniques to prevent this.

We need to make two different datasets. The training sets on which the machine learning model learns and the *test* *set* on which we test if the ***ML model*** *learned* *correctly* the *correlations*.

#*------------ Splitting the Dataset: Train and Test ----------------*

**from** sklearn**.**model\_selection **import** train\_test\_split

x\_train, x\_test, y\_train, y\_test = **train\_test\_split**(old\_x, old\_y, test\_size= 0.2, random\_state = 0)

**print**(x\_train)

**print**(y\_train)

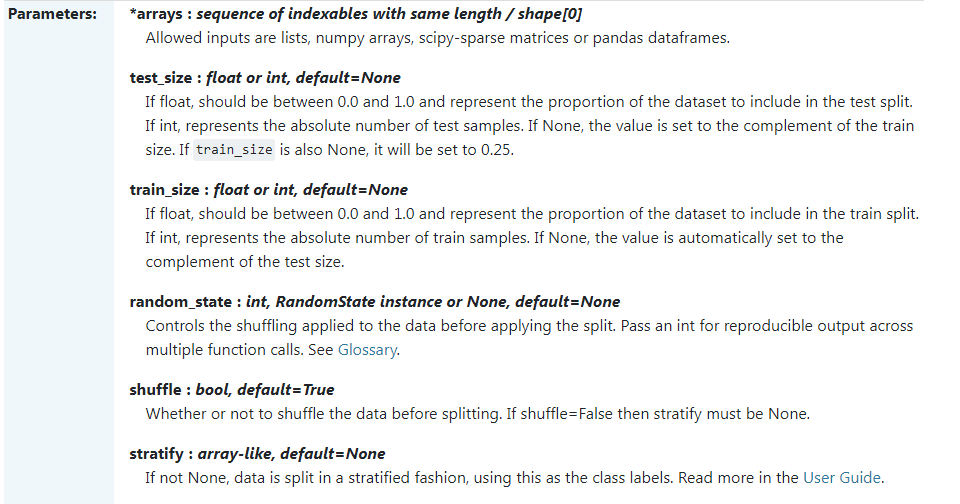
**print**(x\_test)

**print**(y\_test)

* What is Splitting Dataset: We will split our dataset into two separate dataset, Training-set and Test-set.
* Training-set will contain 70% or 80% data to train our ML models.
* Other 30% or 20% data is the Test-set, which will be used to test our models how well it will predict.
* Why Splitting Dataset: We will always keep test-set to avoid the Overfitting. Overfitting is when ML model learn too much from ***train-set*** but ***bad*** at ***prediction***
* Examples of Overfitting: Let’s say we want to predict if a student will land a job interview based on her resume. Now, assume we train a model from a dataset of 10,000 resumes and their outcomes. Next, we try the model out on the original dataset, and it predicts outcomes with 99% accuracy… wow!
* But now comes the bad news. When we run the model on a ***new (“unseen”)*** dataset of resumes, we only get 50% accuracy… uh-oh!
* Our model doesn’t generalize well from our *training* *data* to *unseen* *data*.
* This is known as overfitting, and it’s a common problem in machine learning and data science.
* How to apply:

**from** **sklearn.model\_selection** **import** **train\_test\_split**

x\_train, x\_test, y\_train, y\_test = **train\_test\_split**(old\_x, old\_y, test\_size= 0.2, random\_state = 0)



* ***x\_train, x\_test, y\_train, y\_test*** is used because **train\_test\_split**() returns four matrices. And we grab them in these four variables ***x\_train, x\_test, y\_train, y\_test***. The order must be maintained.
* test\_size= 0.2 means test size is *20%*. It should be 20% or 25% or 30-33% and less than 40% (very rare). train\_size Not needed, it would automatically set to ***1- test\_size***.
* random\_state = 0 Means *no shuffling*. Actually *shuffling* is *good* option 0 is used here to generate same results. Use 42 generally.

**1.10 Data Preprocessing Template**

#*Data Preprocessing Template*

#*Importing the libraries*

**import** numpy **as** np

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

#*Importing the dataset*

dataset = pd**.read\_csv**('Data.csv')

X = dataset**.**iloc[:, :-1]**.**values

y = dataset**.**iloc[:, -1]**.**values

#*Splitting the dataset into the Training set and Test set*

**from** sklearn**.**model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size = 0.2, random\_state = 0)

#*Feature Scaling*

"""from sklearn.preprocessing import StandardScaler

sc\_X = StandardScaler()

X\_train = sc\_X.fit\_transform(X\_train)

X\_test = sc\_X.transform(X\_test)

sc\_y = StandardScaler()

y\_train = sc\_y.fit\_transform(y\_train.reshape(-1,1))"""

* ***Missing data***, ***Categorical*** ***data***, ***Feature*** ***scaling*** are optional. Hence not included here.

**1.11 Standard Deviation**

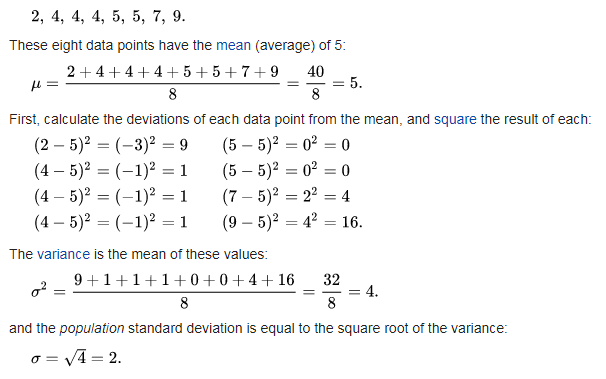
In [statistics](https://en.wikipedia.org/wiki/Statistics), the **standard deviation** is a measure of the amount of variation or [dispersion](https://en.wikipedia.org/wiki/Statistical_dispersion) of a set of values.[[1]](https://en.wikipedia.org/wiki/Standard_deviation#cite_note-StatNotes-1) A low standard deviation indicates that the values tend to be close to the [mean](https://en.wikipedia.org/wiki/Mean) (also called the [expected value](https://en.wikipedia.org/wiki/Expected_value)) of the set, while a high standard deviation indicates that the values are spread out over a wider range.

Standard deviation may be abbreviated **SD**, and is most commonly represented in mathematical texts and equations by the lower case [Greek letter](https://en.wikipedia.org/wiki/Greek_alphabet) sigma [**σ**](https://en.wikipedia.org/wiki/Sigma) or **σ2**, for the population standard deviation, or the [Latin](https://en.wikipedia.org/wiki/Latin_alphabet) letter [s](https://en.wikipedia.org/wiki/S), for the sample standard deviation.

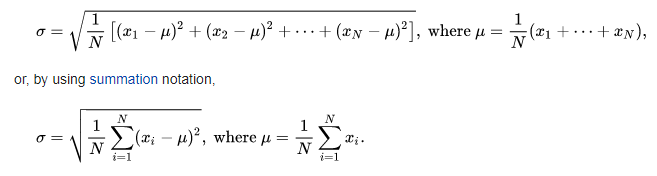
The standard deviation of a [random variable](https://en.wikipedia.org/wiki/Random_variable), [sample](https://en.wikipedia.org/wiki/Sample_(statistics)), [statistical population](https://en.wikipedia.org/wiki/Statistical_population), [data set](https://en.wikipedia.org/wiki/Data_set), or [probability distribution](https://en.wikipedia.org/wiki/Probability_distribution) is the [square root](https://en.wikipedia.org/wiki/Square_root) of its [variance](https://en.wikipedia.org/wiki/Variance). It is [algebraically](https://en.wikipedia.org/wiki/Algebra) simpler, though in practice less [robust](https://en.wikipedia.org/wiki/Robust_statistics), than the [average absolute deviation](https://en.wikipedia.org/wiki/Average_absolute_deviation).[[2]](https://en.wikipedia.org/wiki/Standard_deviation#cite_note-2)[[3]](https://en.wikipedia.org/wiki/Standard_deviation#cite_note-3) A useful property of the standard deviation is that, unlike the variance, it is expressed in the same unit as the data.

* Example:

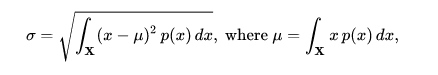
Suppose that the entire population of interest is eight students in a particular class. For a finite set of numbers, the population standard deviation is found by taking the [square root](https://en.wikipedia.org/wiki/Square_root) of the [average](https://en.wikipedia.org/wiki/Average) of the squared deviations of the values subtracted from their average value. The marks of a class of eight students (that is, a [statistical population](https://en.wikipedia.org/wiki/Statistical_population)) are the following eight values:



* In the case where *X* takes random values from a finite data set *x*1, *x*2, …, *xN*, with each value having the same probability, the standard deviation is



* If, instead of having equal probabilities, the values have different probabilities, let *x*1 have probability *p*1, *x*2 have probability *p*2, …, *xN* have probability *pN*. In this case, the standard deviation will be



Practiced version code

#*------------------ Data Preprocessing --------------------*

#*------------------- Importing the libraries ----------------------*

**import** numpy **as** np

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

#*x = np.arange(0, 5, 0.1)*

#*y = np.sin(x)*

#*plt.plot(x, y)*

#*plt.show()*

#*-------------------- importing the dataset --------------------------*

dataset = pd**.read\_csv**("Data.csv")

**print**(dataset)

#*":-1" is to all columns exluding last column*

#*":" to select all rows*

old\_x = dataset**.**iloc[:, :-1]

old\_y = dataset**.**iloc[:, 3]

**print**("Before\n",old\_x)

#*-----------------------  Taking care of missing data ----------------------*

#*import sklearn*

#*print(sklearn.\_\_version\_\_)*

**from** sklearn**.**impute **import** SimpleImputer

imptr = **SimpleImputer**(missing\_values= np**.**nan, strategy="mean") #*"NaN" connot be used. Use np.nan*

#*fix the 2nd and 3rd column*

impt = imptr**.fit**(old\_x**.**iloc[:, 1:3])

**print**(impt) #*it is a method, acts on feature matrix*

old\_x**.**iloc[:, 1:3] = impt**.transform**(old\_x**.**iloc[:, 1:3])

**print**("After\n",old\_x)

#*ooooooooo For older version oooooooooo*

#*from sklearn.preprocessing import Imputer*

#*imptr = Imputer(missing\_values='NaN', strategy='mean', axis=0, verbose=0, copy=True)*

#*-------------- Categorical data --------------*

#*from sklearn.preprocessing import LabelEncoder, OneHotEncoder*

#*label\_encode\_x = LabelEncoder()*

#*old\_x.iloc[:, 0] = label\_encode\_x.fit\_transform(old\_x.iloc[:, 0])*

#*print(old\_x)*

**from** sklearn**.**preprocessing **import** LabelEncoder, OneHotEncoder

**from** sklearn**.**compose **import** ColumnTransformer

#*Encode Country Column*

ct = **ColumnTransformer**(transformers = [("encoding", **OneHotEncoder**(), [0])], remainder = 'passthrough')

#*remainder = 'passthrough' for remaining columns to be unchanged*

old\_x = ct**.fit\_transform**(old\_x) #*convert this output to NumPy array*

**print**(old\_x)

old\_x = np**.array**(old\_x)

**print**(old\_x)

#*Encode dependent vector Column*

label\_encode = **LabelEncoder**()

old\_y = label\_encode**.fit\_transform**(old\_y)

**print**(old\_y)

#*---------------------- Feature Scaling ------------------------*

**from** sklearn**.**preprocessing **import** StandardScaler

sc = **StandardScaler**()

old\_x = sc**.fit\_transform**(old\_x)

**print**(old\_x)

#*------------ Splitting the Dataset: Train and Test ----------------*

**from** sklearn**.**model\_selection **import** train\_test\_split

x\_train, x\_test, y\_train, y\_test = **train\_test\_split**(old\_x, old\_y, test\_size= 0.2, random\_state = 0)

**print**(x\_train)

**print**(y\_train)

**print**(x\_test)

**print**(y\_test)

#*# ^^^^^^^^^^ List slicing ^^^^^^^^^^^*

#*# Initialize list*

#*Lst = [50, 70, 30, 20, 90, 10, 50]*

#*# Display list*

#*print(Lst[::])*

#*# Index -1 represents the last element and -n represents the first element of the list(considering n as the length of the list). Lists can also be manipulated using negative indexes also.*

#*# Initialize list*

#*Lst = [50, 70, 30, 20, 90, 10, 50]*

#*# Display list*

#*print(Lst[-7::1])*

#*# Display list*

#*print(Lst[1:5])*

#*python prcts\_dt\_prep.py*

Current codes

#*Data Preprocessing Tools*

#*Importing the libraries*

**import** numpy **as** np

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

#*Importing the dataset*

dataset = pd**.read\_csv**('Data.csv')

X = dataset**.**iloc[:, :-1]**.**values

y = dataset**.**iloc[:, -1]**.**values

**print**(X)

**print**(y)

#*Taking care of missing data*

**from** sklearn**.**impute **import** SimpleImputer

imputer = **SimpleImputer**(missing\_values=np**.**nan, strategy='mean')

imputer**.fit**(X[:, 1:3])

X[:, 1:3] = imputer**.transform**(X[:, 1:3])

**print**(X)

#*Encoding categorical data*

#*Encoding the Independent Variable*

**from** sklearn**.**compose **import** ColumnTransformer

**from** sklearn**.**preprocessing **import** OneHotEncoder

ct = **ColumnTransformer**(transformers=[('encoder', **OneHotEncoder**(), [0])], remainder='passthrough')

X = np**.array**(ct**.fit\_transform**(X))

**print**(X)

#*Encoding the Dependent Variable*

**from** sklearn**.**preprocessing **import** LabelEncoder

le = **LabelEncoder**()

y = le**.fit\_transform**(y)

**print**(y)

#*Feature Scaling*

**from** sklearn**.**preprocessing **import** StandardScaler

sc = **StandardScaler**()

X = sc**.fit\_transform**(X)

**print**(X)

#*Splitting the dataset into the Training set and Test set*

**from** sklearn**.**model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size = 0.2, random\_state = 0)

**print**(X\_train)

**print**(X\_test)

**print**(y\_train)

**print**(y\_test)

Legacy (older) codes. Cause ERR.

***missing\_data.py*** and ***categorical\_data.py***

#*Data Preprocessing*

#*Importing the libraries*

**import** numpy **as** np

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

#*Importing the dataset*

dataset = pd**.read\_csv**('Data.csv')

X = dataset**.**iloc[:, :-1]**.**values

y = dataset**.**iloc[:, 3]**.**values

#*Taking care of missing data*

**from** sklearn**.**preprocessing **import** Imputer

imputer = **Imputer**(missing\_values = 'NaN', strategy = 'mean', axis = 0)

imputer = imputer**.fit**(X[:, 1:3])

X[:, 1:3] = imputer**.transform**(X[:, 1:3])

#*Encoding categorical data*

#*Encoding the Independent Variable*

**from** sklearn**.**preprocessing **import** LabelEncoder, OneHotEncoder

labelencoder\_X = **LabelEncoder**()

X[:, 0] = labelencoder\_X**.fit\_transform**(X[:, 0])

onehotencoder = **OneHotEncoder**(categorical\_features = [0])

X = onehotencoder**.fit\_transform**(X)**.toarray**()

#*Encoding the Dependent Variable*

labelencoder\_y = **LabelEncoder**()

y = labelencoder\_y**.fit\_transform**(y)