## Feature Engineering Techniques

## ****Terminologies in Statistics - Statistics for Data Science****

* ****The population**** is the set of sources from which data has to be collected.
* A ****Sample**** is a subset of the Population
* A ****Variable**** is any characteristics, number, or quantity that can be measured or counted. A variable may also be called a data item.
* Also known as a statistical model, A statistical ****Parameter**** or population parameter is a quantity that indexes a family of probability distributions. For example, the mean, median, etc of a population.

****Types of Analysis****

1. An analysis of any event can be done in one of two wa****Quantitative Analysis:**** Quantitative Analysis or Statistical Analysis is the science of collecting and interpreting data with numbers and graphs to identify patterns and trends.
2. ****Qualitative Analysis:**** Qualitative or Non-Statistical Analysis gives generic information and uses text, sound and other forms of media to do so.

## ****Categories in Statistics****

There are two main categories in Statistics, namely:

1. Descriptive Statistics
2. Inferential Statistics

****Descriptive Statistics****

**Descriptive Statistics uses the data to provide descriptions of the population, either through numerical calculations or graphs or tables.**

****Inferential Statistics****

**Inferential Statistics makes inferences and predictions about a population based on a sample of data taken from the population in question.**

### ****Measures Of The Center****

1. ****Mean:**** Measure of average of all the values in a sample is called Mean.
2. ****Median:****Measure of the central value of the sample set is called Median.
3. ****Mode:****The value most recurrent in the sample set is known as Mode.

****Measures of the Spread****

Just like the measure of center, we also have measures of the spread, which comprises of the following measures:

1. ****Range:**** It is the given measure of how spread apart the values in a data set are.
2. ****Inter Quartile Range (IQR):**** It is the measure of variability, based on dividing a data set into quartiles.
3. ****Variance:**** It describes how much a random variable differs from its expected value. It entails computing squares of deviations.
   1. ****Deviation**** is the difference between each element from the mean.
   2. ****Population Variance**** is the average of squared deviations
   3. ****Sample Variance**** is the average of squared differences from the mean
4. ****Standard Deviation:**** It is the measure of the dispersion of a set of data from its mean.

# **What is Feature Engineering?**

Feature engineering is the process of using data domain knowledge to create features or variables that make machine learning algorithms work more efficiently. It’s a fundamental task for improving machine learning model performance and prediction accuracy.

**Feature engineering includes a number of processes, like**

* filling missing values within a variable
* encoding categorical variables into numbers
* variable transformation
* creating or extracting new features from the ones available in your dataset

# **Feature Engineering vs. Feature Selection**

Feature engineering allows us to ****create new features**** from the ones we already have in order to help the machine learning model make more effective and accurate predictions.

[Feature selection](https://heartbeat.fritz.ai/hands-on-with-feature-selection-techniques-an-introduction-1d8dc6d86c16" \t "https://heartbeat.fritz.ai/_blank), on the other hand, allows us to ****select features**** from the feature pool (including any newly-engineered ones) that will help machine learning models make predictions on target variables more efficiently.

# **Series Sections**

This series is divided into sections—here are brief descriptions of each of the sections:

* ****Feature Types:****Or variables types—we'll learn about continuous, discrete, and categorical variables (which can be nominal or ordinal), alongside time-date and mixed variables.
* ****Common Issues:****This post will discuss different issues you’ll see in real-world datasets like missing data, data imputations, variable distribution, outliers, and others.
* ****Dealing with Missing Values:****We’ll examine the major techniques used to fill the missing data in your dataset.
* ****Categorical Encoding:****This post will discuss the different techniques to transform categorical variables into numbers—frequency encoding, one-hot encoding, and more.
* ****Feature Transformation:****We’ll explore the mathematical transformations you can apply to alter the distribution of numerical variables, like logarithmic or reciprocal transformations.
* ****Variable Discretization:****This article looks at the procedures to discretize variables, like equal-width, equal-frequency, discrete decision using decision trees, clustering and more.
* ****Dealing with Outliers:****This post will show how toidentify outliers and remove them from your dataset.
* ****Feature Scaling:**** We’ll cover several techniques to scale features, like standardization, scaling to the minimum and maximum, scaling to the unit length of the vector, and more.
* ****Handling Time-Date and Mixed Variables:****Various waysto create new features from date, time, and mixed variables.

# **What is a Variable?**

A variable is any characteristic, number, or quantity that can be measured or counted. We call them ****variables**** because the values they take may vary, and usually do.

The following are examples of variables:

* Age (23, 52, 60, …).
* Gender (male, female)
* Country of birth (Algeria, USA, Japan, …)
* Eye color (green, brown, blue, and purple maybe)
* Vehicle Brand (Buggati, Ferrari)

We classify variables in a dataset into one of these major types:

* ****Numerical variables****
* ****Categorical variables****
* ****Datetime variables****
* ****Mixed variables****

# **Numerical Variables**

The values of numerical variables are (predictably) numbers. For example, total rainfall measured in inches, heart rate, the number of cheeseburgers consumed in an hour—all numerical values.

We can further classify them into:

* Continuous variables
* Discrete variables

## **@ Continuous Variable**

A ****continuous variable**** is one that can take on an uncountable set of values. It may contain any value within a given range.

* For example****,**** the total amount paid by a customer in a supermarket is continuous. The customer can pay $20, $16.50, $150, and so on.

**To visualize such a variable we have a range of options, including:**

* Density plot
* Histogram
* Box plot
* Scatter plot.

## **Discrete Variable**

A discrete variable is a variable that can only take on a finite number of values, and these values are integers ( — which means numbers that are not a fraction), they are counts.

* For example, the number of things bought by a customer in a supermarket is discrete. The customer can buy 2, 20, or 150 things, but not 10.4 items. It is always a round number.

**To visualize such discrete variables, you can use the following type of plots:**

* Count plot
* Pie chart

# **Categorical Variables**

The values of a categorical variable are selected from a group of ****categories****, also called ****labels****. Examples are gender (male or female) and marital status (never married, married, divorced, or widowed).

Other examples of categorical variables include:

* Gender (male, female)
* Mobile network provider (Mobilis, Vodafone, Orange)
* City name (Tiaret, Algiers, Texas, Dubai)

# **Dates and Times**

A particular type of categorical variable are those that take dates or time as values.

For example:

* Date of birth (16–04–1997, 12–01–2012)
* Date of application (2020-Jan, 2022-Feb)

# **Mixed Variables**

Finally, mixed variables are those whose values can contain both numbers and labels.

For a variety of reasons, mixing variables can occur in a given dataset, especially when filling its values.

# Common Issues in Datasets.

Here are the general issues we can encounter in a given dataset:

* Missing data
* Categorical variable — cardinality
* Categorical variable — rare labels
* Linear model assumptions
* Variable distribution
* Outliers
* Feature magnitude

<https://heartbeat.fritz.ai/hands-on-with-feature-engineering-techniques-common-issues-in-datasets-a0c2cf97b1a5>

# Imputing Missing Values.

# **Data Imputation**

Data imputation is the act of replacing missing data with statistical estimates of the missing values.

# **Missing Data Imputation Techniques**

We’re going to dive into techniques that apply to numerical and categorical variables, and also some methods that apply to both:

## **Numerical Variables**

* Mean or median imputation
* Arbitrary value imputation
* End of tail imputation

## **Categorical Variables**

* Frequent category imputation
* Add a ****missing**** category

# **Mean or Median Imputation**

Mean or median imputation consists of replacing all occurrences of missing values (NA) within a variable with the ****mean or median**** of that variable.

*This method is suitable for numerical variables.*

Here are some points to consider when using this method:

* If the variable follows a normal distribution, the mean and median are approximately the same.
* If the variable has a skewed distribution, then the median is a better representation.

# **Missing Category Imputation**

This method consists of treating missing data as an ****additional****label or category of the variable. Thus, we create a new label or category by filling the missing observations with a ****Missing****category.

CODE for IMPUTER of MISSING VALUES

|  |
| --- |
|  |
|  | from sklearn.impute import SimpleImputer |
|  | # create the imputer, the strategy can be mean and median. |
|  | imputer = SimpleImputer(missing\_values=np.nan, strategy='mean') |
|  |  |
|  | # fit the imputer to the train data |
|  | imputer.fit(train) |
|  |  |
|  | # apply the transformation to the train and test |
|  | train = imputer.transform(train) |
|  | test = imputer.transform(test) |
|  |  |

# **# create the imputer, the strategy can be mean and median.**

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

# **# create the imputer, with fill value 999 as the arbitraty value**

# **imputer = SimpleImputer(missing\_values=np.nan, strategy='constant', fill\_value=999)**

# **# create the imputer (End Tail Impouter)**

# **imputer = EndTailImputer(distribution='gaussian', tail='right')**

# **# create the imputer, with most frequent as strategy to fill missing value.**

# **imputer = SimpleImputer(missing\_values=np.nan, strategy='most\_frequent')**

# **# create the imputer, with most frequent as strategy to fill missing value.**

# **imputer = SimpleImputer(missing\_values=np.nan, strategy='constant', fill\_value="Missing")**

# **Arbitrary Value Imputation**

Arbitrary value imputation consists of replacing all occurrences of missing values (NA) within a variable with an arbitrary value. The arbitrary value should be different from the mean or median and not within the****normal values of the variable.****

We can use arbitrary values such as 0, 999, -999 (or other combinations of 9s) or -1 (if the distribution is positive).

# **End of Tail Imputation**

End of tail imputation is roughly equivalent to arbitrary value imputation, but it automatically selects the arbitrary values at the end of the variable distributions.

*This method is suitable for numerical variables.*

Here are ways to select arbitrary values:

* If the variable follows a normal distribution, we can use the mean plus or minus 3 times the standard deviation.
* If the variable is skewed, we can use the IQR proximity rule.

*The values to replace missing data should be calculated only on the train set.*

# **Frequent Category Imputation**

Frequent category imputation—or ****mode imputation****—consists of replacing all occurrences of missing values (NA) within a variable with the mode, or the most frequent value.

*This method is suitable for numerical and categorical variables, but in practice, we use this technique with categorical variables.*

# Encoding Categorical Variable.

# **Categorical Encoding**

Most machine learning algorithms and deep neural networks require numerical inputs. This means that if we have categorical data, we must first encode it to numbers in order to build models that actually work.

## **Traditional techniques**

* One-hot encoding
* Count or frequency encoding
* Ordinal or label encoding

## **Monotonic relationship**

* Ordered label encoding
* Mean encoding
* Probability ratio encoding
* Weight of evidence

## **Alternative techniques**

* Rare labels encoding
* Binary encoding

# **One-hot encoding**

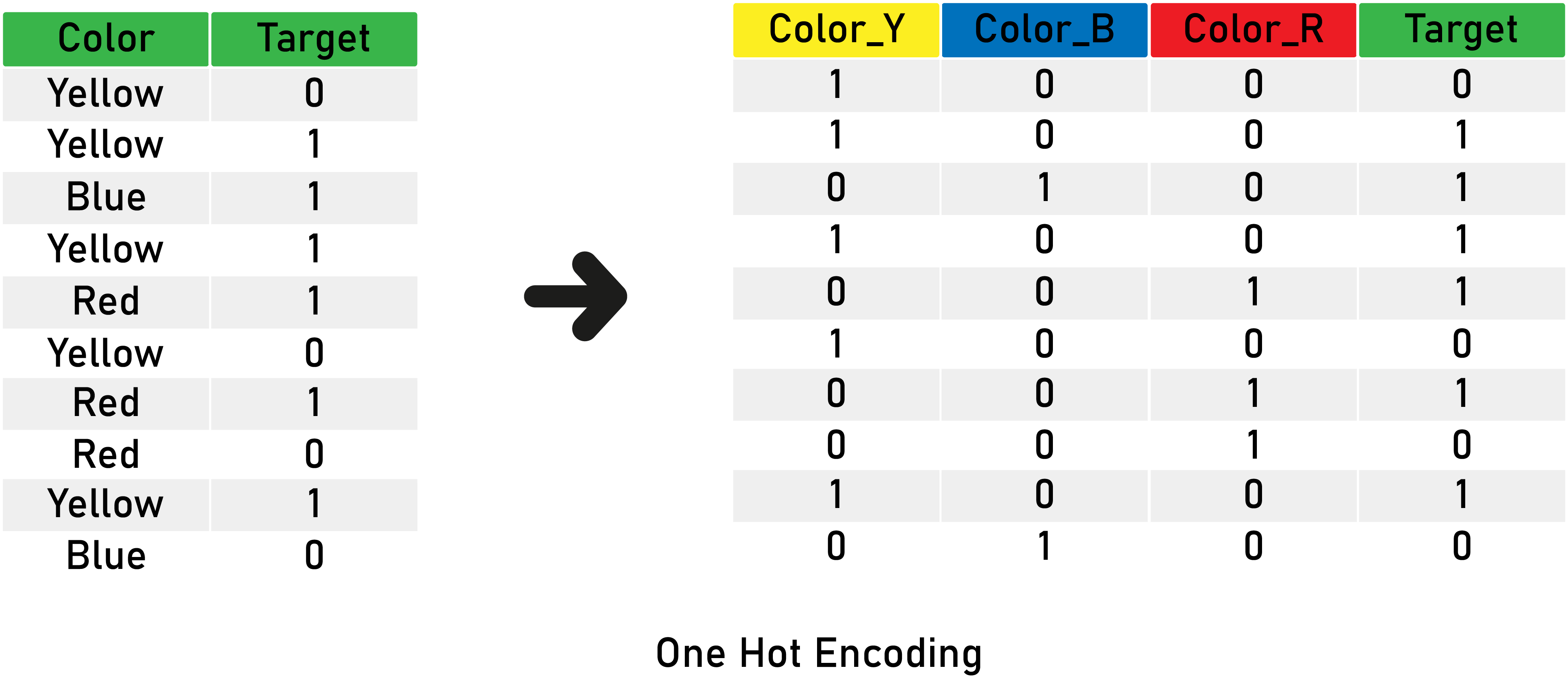
One-hot encoding consists of encoding each categorical variable with a set of boolean variables that take values of ****0**** or ****1.****This value then serves to indicate if a category is present for each observation.

## **One-hot encoding into k variables**

There are a few occasions when it’s better to encode variables into ****k**** variables:

* When building tree-based algorithms.
* When making feature selection with recursive algorithms.
* When interested in determining the importance of every single category.

Here is an illustration of one-hot encoding into ***k*** variables:

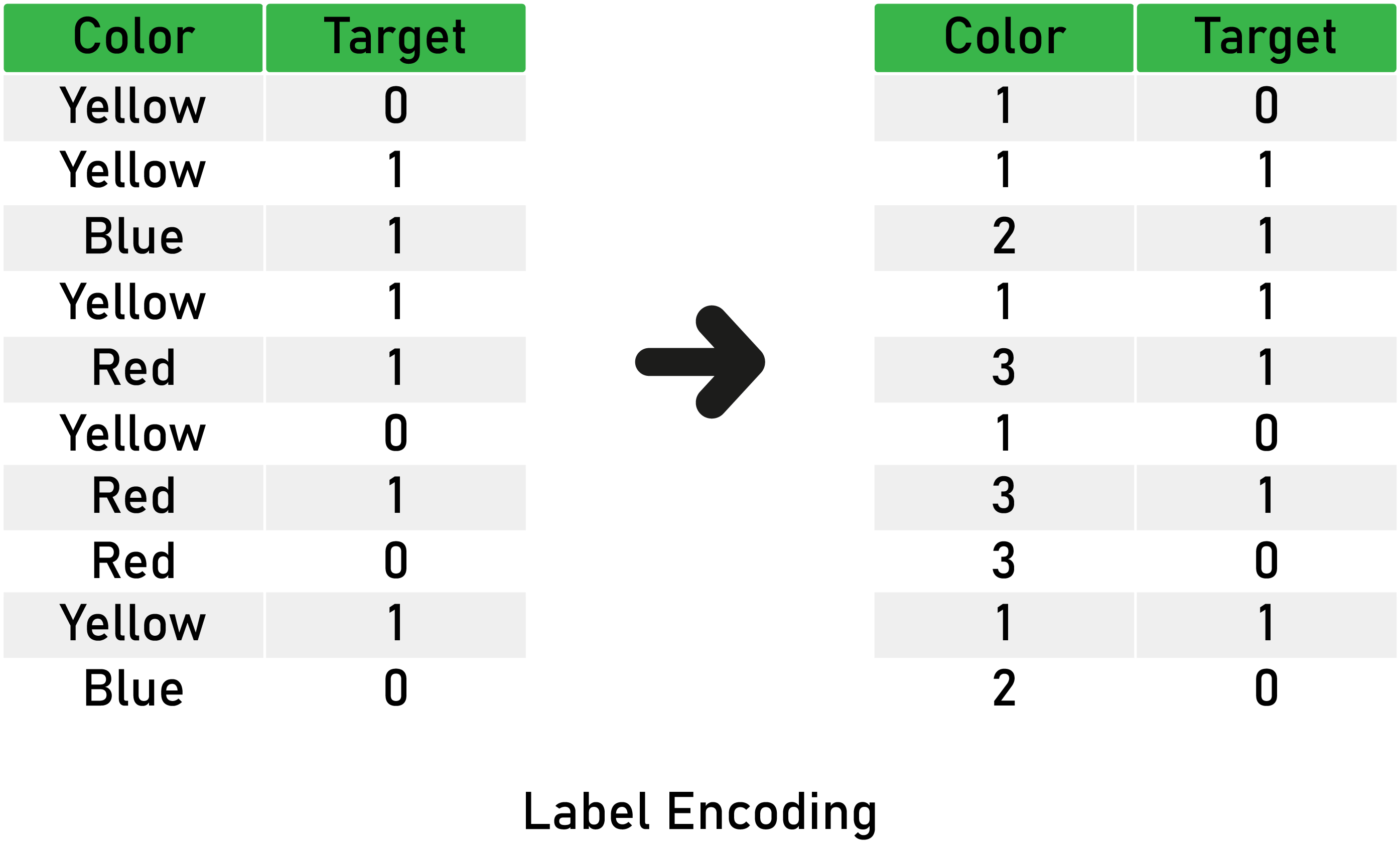


Finally, here is a code snippet with the Pandas library to achieve ***k*** and***k-1*** one-hot encoding:

|  |
| --- |
|  |
|  | import pandas as pd |
|  |  |
|  | # read your dataset |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | # perform one hot encoding with k |
|  | data\_with\_k = pd.get\_dummies(data) |
|  |  |
|  | # perform one hot encoding with k - 1, it automatically drop the first. |
|  | data\_with\_k\_one = pd.get\_dummies(data, drop\_first = true) |

# **Integer (Label) Encoding**

Integer encoding (also known as label encoding) includes replacing the categories with digits from ****1**** to ****n**** (or ****0**** to ****n-1****, depending on the implementation), where ****n**** is the number of the variable’s distinct categories (the ****cardinality****), and these numbers are assigned arbitrarily.



## **Advantages of integer (label) encoding**

* Straightforward to implement.
* Does not expand the feature space.
* Can work well enough with tree-based algorithms.
* Allows agile benchmarking of machine learning models.

# Import label encoder

**from** sklearn **import** preprocessing

# label\_encoder object knows how to understand word labels.

label\_encoder **=** preprocessing.LabelEncoder()

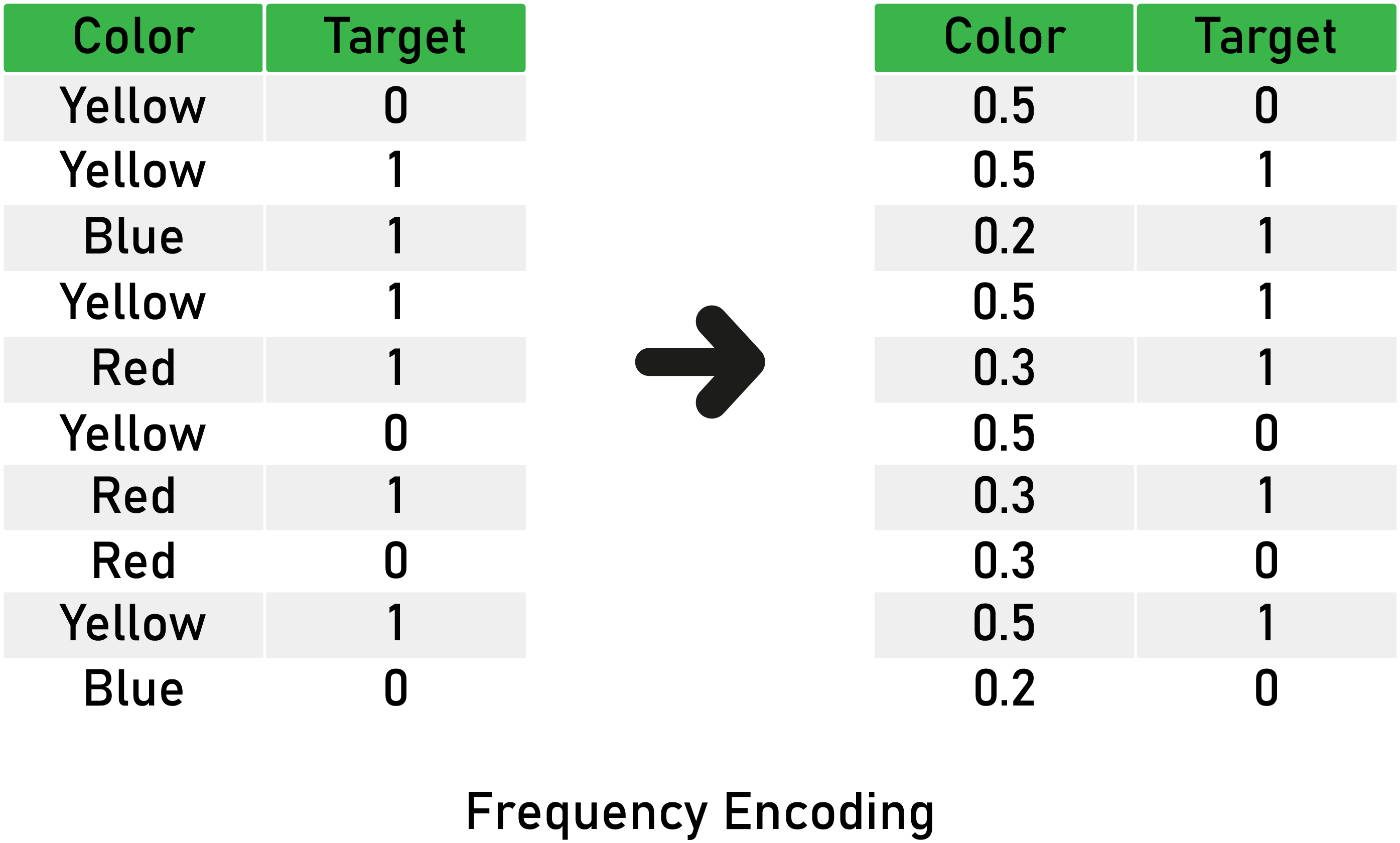
# Encode labels in column 'species'.

df['species']**=** label\_encoder.fit\_transform(df['species'])

df['species'].unique()

# **Count or Frequency Encoding**

Here, we replace categories with the ****count****or ****percentage****of observations that show each category in the dataset. Thus, we capture the representation of each label in the dataset.



## **Advantages of Count or Frequency encoding**

* Straightforward to implement.
* Does not expand the feature space.
* Can work well with tree-based algorithms.

|  |
| --- |
|  |
|  | import pandas as pd |
|  |  |
|  | # get you data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | # loop to find the different count of categories in a dict and apply them to the variable |
|  | # in train and test set. |
|  | for variable in train.columns: |
|  | count\_dict = train[variable].value\_counts().to\_dict() |
|  | train[variable].map(count\_map) |
|  | test[variable].map(count\_map) |

# **Mean (Target) Encoding**

Mean encoding means replacing the category with the mean target value for that category.

mean\_ordinal=df.groupby(['Cabin'])['Survived'].mean().to\_dict()

|  |
| --- |
|  |
|  | import pandas as pd |
|  |  |
|  | # get you data. |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | # get your target variable name. |
|  | target = "your target variable name" |
|  |  |
|  | # loop over the categorical columns to apply the encoding. |
|  | for variable in train.columns: |
|  | # create dictionary of category:mean values. |
|  | dict = train.groupby([variable])[target].mean().to\_dict() |
|  | # apply the encoding to the train and test sets. |
|  | train[variable] = train[variable].map(dict) |
|  | test[variable] = test[variable].map(dict) |

# **Weight of Evidence Encoding**

Weight of evidence (WOE) is a technique used to encode categorical variables for classification.

The rule is simple; WOE is the ****natural logarithm (***ln***)****of the probability that the target equals ****1**** divided by the probability of the target equals ****0****.

Here is a mathematic formula : ****WOE = ln (p(1) / p(0))****.

* Where ****p(1)**** is the probability of the target being 1, and ****p(0)**** is the probability of the target being 0.

***If the result is negative, you can change the sign of the output.***



# **Probability Ratio Encoding**

This encoding is suitable for classification problems only, where the target is ****binary****.

It’s similar to WOE, but we don’t apply the natural logarithm. Instead, for each category, we calculate the mean of the target = ****1****, which is the probability of the target being 1 (****P(1)****), and the probability of the target being 0 (****P(0)****).

Finally, we calculate the ratio = ****P(1)/P(0)****, and we replace the categories by that ratio.



|  |
| --- |
|  |
|  | import pandas as pd |
|  | import numpy as np |
|  |  |
|  | # get you data. |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | # get your target variable name. |
|  | target = "your target variable name" |
|  |  |
|  | # we loop over all the categorical variables |
|  | for variable in train.columns: |
|  |  |
|  | # calculating the mean of target for each category. (probability of events or p(1)) |
|  | dataframe = pd.DataFrame(train.groupby([variable])[target].mean()) |
|  |  |
|  | # calculating the non target probability. (probability of non-events or p(0)) |
|  | dataframe['non-target'] = 1 - dataframe[target] |
|  |  |
|  | # calculating the ration. |
|  | dataframe['ratio'] = dataframe[target] / dataframe['non-target'] |
|  | ratio\_mapping = dataframe['ratio'].to\_dict() |
|  |  |
|  | # applying the probability ration encoding. |
|  | train[variable] = train[variable].map(ratio\_mapping) |
|  | test[variable] = test[variable].map(ratio\_mapping) |

# **Binary encoding**

The binary encoding uses (as its name indicates) binary code; the process is relatively easy and straightforward. As a first step, we convert each integer to binary code. Then each binary digit gets one column in the dataset.

If there are ****n**** unique categories, then binary encoding results in the only ****log****base 2 of ****n****features, which is = ****ln(n) / ln(2)****

## **Advantages of Binary encoding**

* Straightforward to implement.
* Does not expand the feature space too much.



|  |
| --- |
|  |
|  | import pandas as pd |
|  | from category\_encoders import BinaryEncoder |
|  |  |
|  | # get your data. |
|  | data = # your data |
|  |  |
|  | # split into X and y. |
|  | x\_train = data.drop('target', axis = 1) |
|  | y\_train = data['target'] |
|  |  |
|  | # create an encoder object - it will apply on all strings columns. |
|  | binary = BinaryEncoder() |
|  |  |
|  | # fit and transform to get encoded data. |
|  | binary.fit\_transform(x\_train, y\_train) |

# Transforming Variable

# **Why Transformations?**

Some machine learning models, like linear and logistic regression, assume that the variables follow a normal distribution. More likely, variables in real datasets will follow more a skewed distribution.

By applying a number of transformations to these variables, and mapping their skewed distribution to a normal distribution, we can increase the performance of our models.

|  |
| --- |
|  |
|  | import matplotlib.pyplot as plt |
|  | import scipy.stats as stats |
|  | import pandas as pd |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("some\_data.csv") |
|  |  |
|  | ## create and show the plot. |
|  | stats.probplot(data["your\_variable"], dist="norm", plot=plt) |
|  | plt.show() |

# **How can we transform variables?**

The most commonly-used methods to transform variables are the following:

* Logarithmic transformation
* Square root transformation
* Reciprocal transformation
* Exponential or power transformation
* Box-Cox transformation
* Yeo-Johnson transformation

# **Logarithmic transformation**

IMG_257

This is the simplest and most popular among the different types of transformations and involves a substantial transformation that significantly affects distribution shape.

|  |
| --- |
|  |
|  | from sklearn.preprocessing import FunctionTransformer |
|  | columns = ['col\_1','col\_2','col\_n'] |
|  | # create the function transformer object with logarithm transformation |
|  | logarithm\_transformer = FunctionTransformer(np.log, validate=True) |
|  |  |
|  | # apply the transformation to your data |
|  | data\_new = logarithm\_transformer.transform(data[cols]) |

# **Square root transformation**

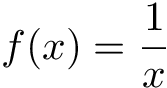
IMG_257

Another simple transformation, this one has an average effect on distribution shape: it’s weaker than logarithmic transformation, and it’s also used for reducing right-skewed distributions.

One particular advantage of square root transformation is that you can apply it to zero values.

|  |
| --- |
|  |
|  | from sklearn.preprocessing import FunctionTransformer |
|  | columns = ['col\_1','col\_2','col\_n'] |
|  | # create the function transformer object with square root transformation |
|  | sqrt\_transformer = FunctionTransformer(np.sqrt, validate=True) |
|  |  |
|  | # apply the transformation to your data |
|  | data\_new = sqrt\_transformer.transform(data[cols]) |

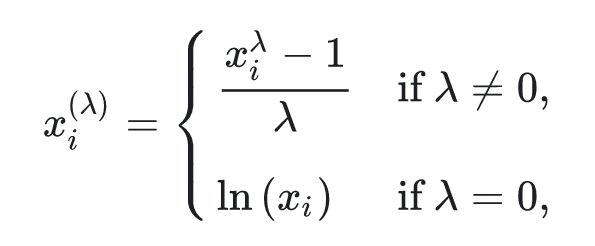
# **Reciprocal transformation**



The reciprocal transformation is a powerful transformation with a radical effect. The reciprocal ****reverses the order**** among values of the same sign, so large values become smaller. The negative reciprocal ****preserves the order****among values of the same sign.

|  |
| --- |
|  |
|  | reciprocal\_transformer = FunctionTransformer(np.reciprocal, validate=True) |
|  |  |
|  | # apply the transformation to your data |
|  | data\_new = reciprocal\_transformer.transform(data[cols]) |

# **Box-Cox Transformation**



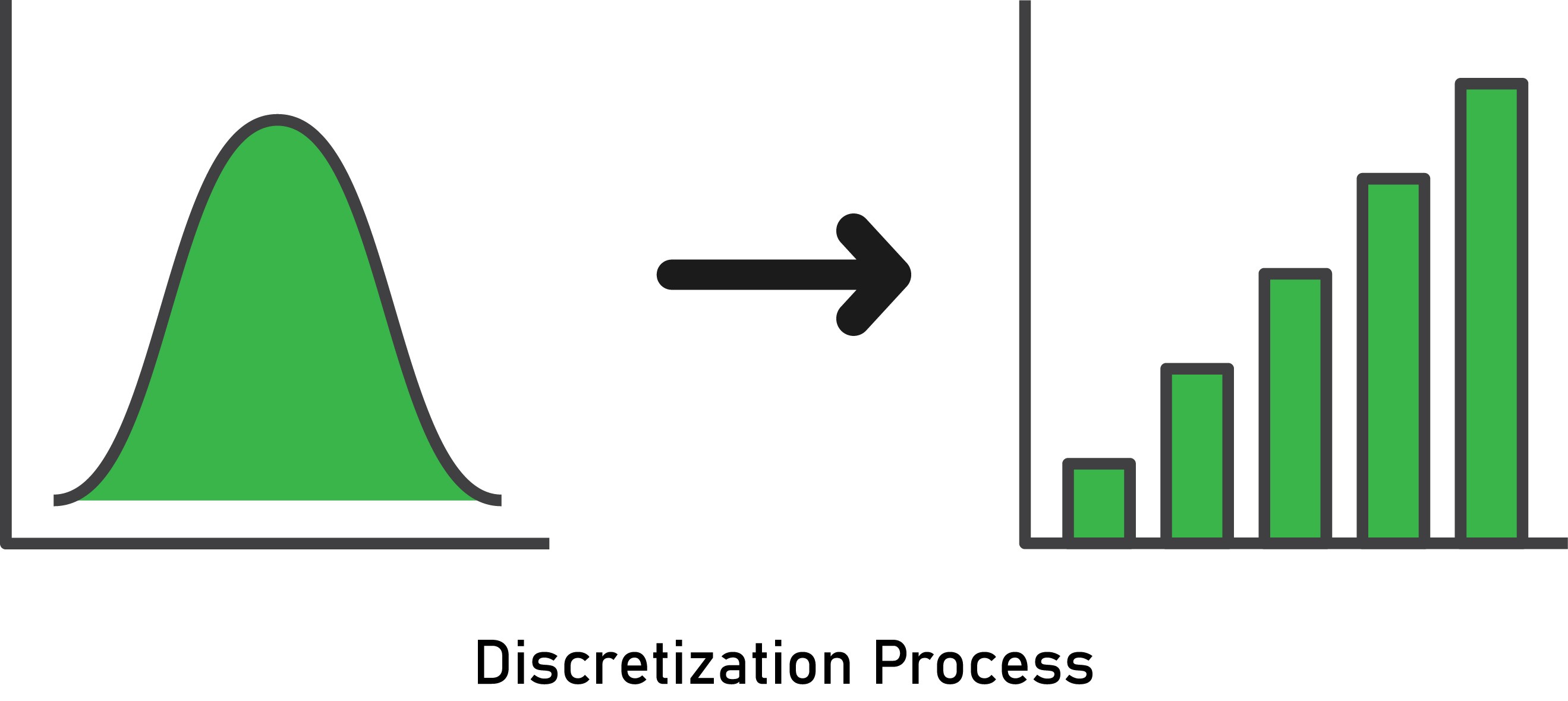
Box-Cox is one of the most successful transformations; it’s an evolution of the exponential transformation, which looks through various exponents instead of trying them manually.

By doing Box-Cox transformation, we’re searching and evaluating all the other transformations and choosing the best one.

from sklearn.preprocessing import PowerTransformer

|  |
| --- |
|  |
|  | boxcox\_transformer = PowerTransformer(method='box-cox', standardize=False) |
|  |  |
|  | # apply the transformation to your data |
|  | data\_new = boxcox\_transformer.transform(data[cols]) |

# Variable Discretization



This article is all about variable discretization, which is the process of ****transforming a continuous variable into a discrete one.**** It essentially creates a set of contiguous intervals that span the variable’s value range.

# **Discretization Approaches**

There are multiple approaches to achieve this discretization. In this guide, we’ll explore the following methods:

## **Supervised Approach**

* Discretization with decision trees

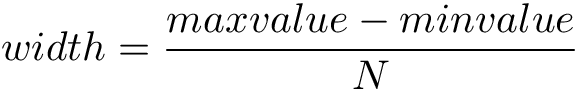
## **Unsupervised Approaches**

* Equal-width discretization
* Equal-frequency discretization
* K-means discretization

# **Equal-width Discretization**

This is the most simple form of discretization—it divides the range of possible values into ****N**** bins of the ****same width****.

The width of intervals is determined by the following formula:



*where******N******is the number of bins or intervals, this parameter is something to determine experimentally—there’s no rule of thumb here.*

For example, if the variable interval is [100, 200], and we want to create 5 bins, that means 200-100 / 5 = 20, so each bin’s width is 20, andthe intervals will be [100, 120], [120, 140],…,[180, 200].

****When working with equal-width discretization, there are some points to consider:****

* Equal-width discretization does not improve the values spread.
* This method handles outliers.
* It creates a discrete variable (obviously).
* It’s useful when combined with with categorical encodings

|  |
| --- |
|  |
|  | import pandas as pd |
|  | from sklearn.preprocessing import KBinsDiscretizer |
|  |  |
|  | # load your data |
|  | data = pd.read\_csv('yourData.csv') |
|  |  |
|  | # create the discretizer object with strategy uniform and 8 bins |
|  | discretizer = KBinsDiscretizer(n\_bins=8, encode='ordinal', strategy='uniform') |
|  |  |
|  | # fit the discretizer to the train set |
|  | discretizer.fit(train) |
|  |  |
|  | # apply the discretisation |
|  | train = discretizer.transform(train) |
|  | test = discretizer.transform(test) |

# **K-Means Discretization**

This discretization method consists of applying k-means clustering to the continuous variable—then each cluster is considered as a ****bin****.

First, let’s quickly review the k-means algorithm:

1. We create ****K**** random points. These points will be the center of clusters.
2. We associate every data point with the closest center (using some distance metric like Euclidean distance).
3. Finally, we re-compute each center position in the center of its associated points.

|  |
| --- |
|  |
|  | discretizer = KBinsDiscretizer(n\_bins=6, encode='ordinal', strategy='kmeans') |

# NOTE :- Rest Code is same of equal width

# **Outliers**

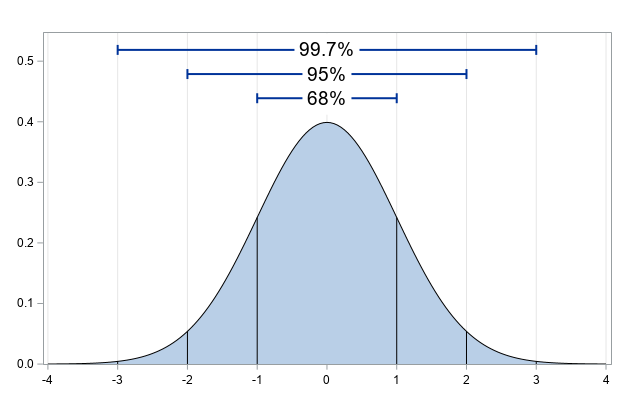
[An outlier](https://heartbeat.fritz.ai/how-to-make-your-machine-learning-models-robust-to-outliers-44d404067d07" \t "https://heartbeat.fritz.ai/_blank) is a data point that’s ****significantly different**** from the remaining data.

Another way of saying this is that an outlier is an observation that deviates so much from the other observations, it arouses suspicion that a different mechanism produced it.

# **Detecting Outliers**

We can detect and find outliers using various techniques. Some of them include:

* ****Using visualization plots like boxplot and scatterplot:****
* ****Using a normal distribution (mean and std):****



# In a normal distribution, about 99.7% of the data lie within three standard deviations of the mean. Consequently, if any observation is more than three times the standard deviation, it’s possible that it’s an outlier.

## ****Inter-quantal range proximity rule:****

The concept of the ****interquartile range**** (****IQR****) is used to build boxplot graphs. The idea is simple—we divide our data into four parts, and each part is a quartile.

IQR is the difference between the third quartile Q3 ( 75 percent) and the first quartile or Q1 (25 percent).

With IQR, outliers are defined as the observations that are:

* Below Q1 − 1.5 × IQR.
* Above Q3 + 1.5 × IQR.

# **Handling Outliers**

Now that we understand how to detect outliers in a better way, it’s time to engineer them. We’re going to explore a few different techniques and methods to achieve that:

* ****Trimming:**** Simply removing the outliers from our dataset.
* ****Imputing:****We treat outliers as missing data, and we apply missing data imputation techniques.
* ****Discretization:**** We place outliers in edge bins with higher or lower values of the distribution.
* ****Censoring:**** Capping the variable distribution at the maximum and minimum values.

# **Trimming**

Trimming (or truncation) merely means removing outliers from the dataset; what we need here is just to decide on a metric to determine outliers.

Here are some points to consider when working with the trimming method:

* This method is fast.
* It can remove a significant amount of data ( — so be careful).

## **Inter-quantal range proximity rule**

In this rule, the boundaries are determined using IQR proximity rules:

|  |
| --- |
|  |
|  | import pandas as pd |
|  | import numpy as np |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | for variable in data.columns: |
|  | #calculate the IQR |
|  | IQR = data[variable].quantile(0.75) - data[variable].quantile(0.25) |
|  |  |
|  | #calculate the boundries |
|  | lower = data[variable].quantile(0.25) - (IQR \* 1.5) |
|  | upper = data[variable].quantile(0.75) + (IQR \* 1.5) |
|  |  |
|  | # find the outliers |
|  | outliers = np.where(data[variable] > upper, True, np.where(data[variable] < lower, True, False)) |
|  |  |
|  | # remove outliers from data. |
|  | data = data.loc[~(outliers, ]  **OR**  # replacing the outliers data[variable] = np.where(data[variable] > upper, upper, np.where(data[variable] < lower, lower, data[variable])) |

# **Censoring**

Censoring (or capping) means setting the maximum ****and/or**** the minimum of the distribution at an arbitrary value.

In other words, values bigger or smaller than the arbitrarily chosen value are ****replaced by this value.****

When doing capping, remember that:

* It does not remove data.
* It distorts the distributions of the variables.

## **Arbitrarily**

You can choose values to replace outliers arbitrarily; this can be based on the requirements of your use case. Here’s a code snippet:

|  |  |
| --- | --- |
|  | # import the needed packages |
|  | import pandas as pd |
|  | import numpy as np |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | for variable in data.columns: |
|  |  |
|  | # create boundries (age for example) |
|  | lower = 10 |
|  | upper = 89 |
|  |  |
|  | # replacing the outliers |
|  | data[variable] = np.where(data[variable] > upper, upper, np.where(data[variable] < lower, lower, data[variable])) |
|  |  |

## **Gaussian approximation**

Another code snippet that sets the boundaries with values according to the mean and standard deviation:

|  |  |
| --- | --- |
|  | # import the needed packages |
|  | import pandas as pd |
|  | import numpy as np |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | for variable in data.columns: |
|  |  |
|  | #calculate the boundries |
|  | lower = data[variable].mean() - 3 \* data[variable].std() |
|  | upper = data[variable].mean() + 3 \* data[variable].std() |
|  |  |
|  | # replacing the outliers |
|  | data[variable] = np.where(data[variable] > upper, upper, np.where(data[variable] < lower, lower, data[variable])) |

## **Using quantiles**

In the following code snippet, the boundaries are determined using the quantiles, through which you can specify any percentage you want:

|  |  |
| --- | --- |
|  | # import the needed packages |
|  | import pandas as pd |
|  | import numpy as np |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | for variable in data.columns: |
|  |  |
|  | #calculate the boundries |
|  | lower = data[variable].quantile(0.10) |
|  | upper = data[variable].quantile(0.90) |
|  |  |
|  | # replacing the outliers |
|  | data[variable] = np.where(data[variable] > upper, upper, np.where(data[variable] < lower, lower, data[variable])) |

# **Imputing (Already Discussed in the same sheet)**

Another technique used to handle outliers is to treat them as missing data. We have a range of methods that we can use to replace or impute outliers. If you’d like to explore these techniques in more depth, you can do so [here](https://heartbeat.fritz.ai/hands-on-with-feature-engineering-techniques-imputing-missing-values-6c22b49d4060" \t "https://heartbeat.fritz.ai/_blank).

# **Transformation (Already Discussed in the same sheet)**

We can also apply some mathematical transformations, such as log transformation. To handle the outliers, there are a range of transformation techniques, which you can learn more about [here](https://heartbeat.fritz.ai/hands-on-with-feature-engineering-techniques-transforming-variables-acea03472e24" \t "https://heartbeat.fritz.ai/_blank).

## Feature Scaling

# **Feature Scaling**

****Feature scaling**** refers to the methods used to normalize the range of values of independent variables. In other words, the ways to set the feature value range within a similar scale.

# **Why Feature Scaling Matters**

Feature magnitude matters for several reasons:

* The scale of the variable directly influences the regression coefficient.
* Variables with a more significant magnitude dominate over the ones with a smaller magnitude range.
* Gradient descent converges faster when features are on similar scales.
* Feature scaling helps decrease the time to find support vectors for SVMs.
* Euclidean distances are sensitive to feature magnitude.

## **Algorithms sensitive to feature magnitude**

* Linear and logistic regression
* Neural networks
* Support vector machines
* KNN
* K-means clustering
* Linear discriminant analysis (LDA)
* Principal component analysis (PCA)

## **Algorithms intensive to feature magnitude**

* Classification and regression trees
* Random forests
* Gradient boosted trees

# **Scaling methods**

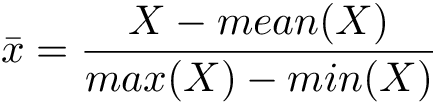
Next, let’s take a closer look at the following feature scaling methods:

* Mean normalization
* Standardization
* Robust scaling (scaling to median and IQR)
* Scale to maximum and minimum
* Scale to the absolute maximum
* Scale to unit norm

# **Mean Normalization**

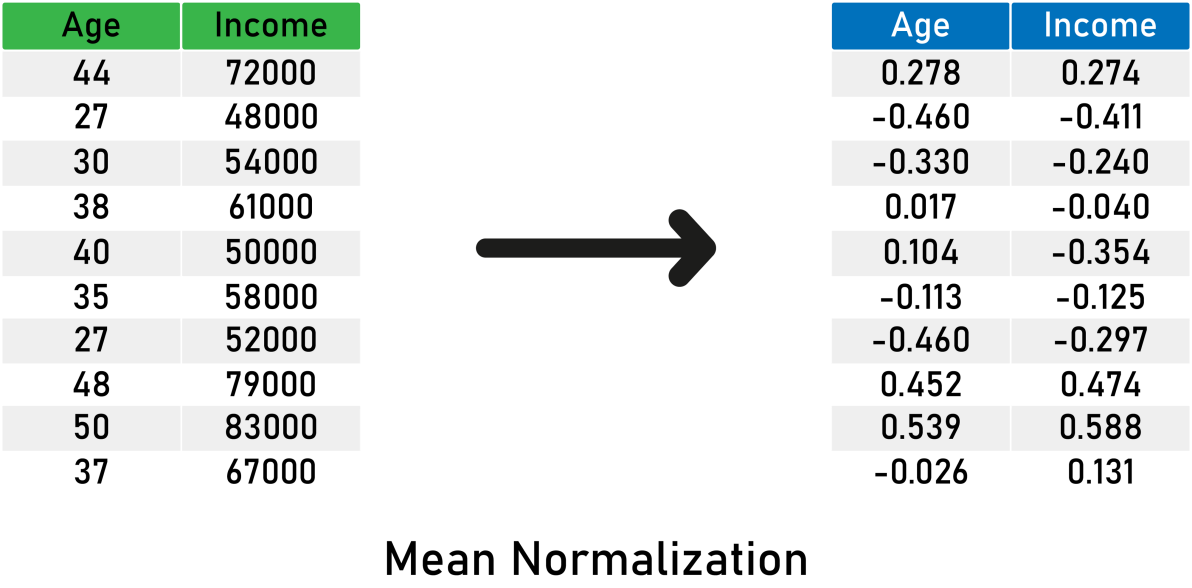
Mean normalization suggests centering the variable at 0 and re-scaling the variable’s****value range****tothe range of -1 to 1.

The method includes subtracting the mean from each variable observation and then dividing by the difference between the minimum and the maximum value of that variable:



*This technique will not******normalize the variable distribution.*****

Here’s an illustration of this method:



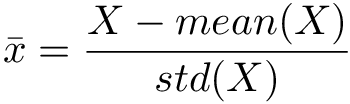
Here’s what you need to remember about mean normalization:

* It centers the mean at 0.
* The resulting variance will be different.
* It may modify the shape of the original distribution.
* It “normalizes” the minimum and maximum values within the range[-1, 1].
* It preserves outliers if they exist.

|  |
| --- |
|  |
|  | import pandas as pd |
|  | import numpy as np |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | # calculate the means |
|  | means = train.mean(axis = 0) |
|  |  |
|  | # calculate max - min |
|  | max\_min = train.max(axis = 0) - train.min(axis = 0) |
|  |  |
|  | # apply the transformation to data |
|  | train\_scaled = (train - means) / max\_min |
|  | test\_scaled = (test - means) / max\_min |

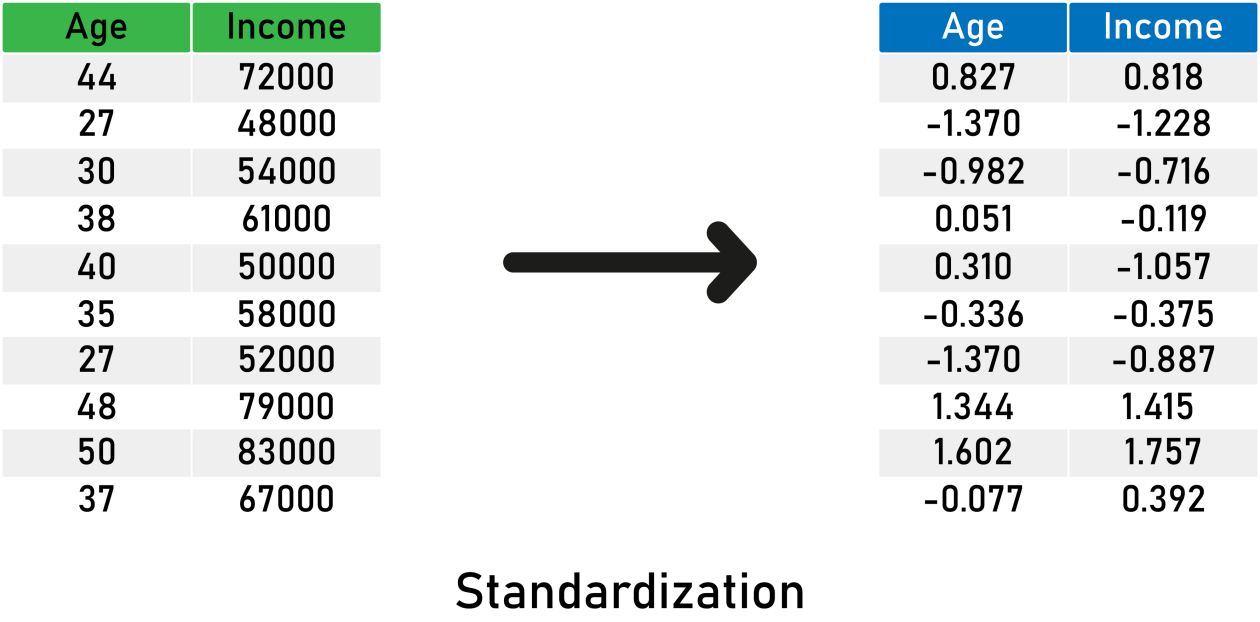
# **Standardization**

Standardization suggests centering the variable at 0 and standardizing the variance to 1. The procedure includes subtracting the mean from each variable observation and then dividing by the standard deviation:



*This technique will not******normalize the variable distribution.*****

Here’s an example on some sample data:



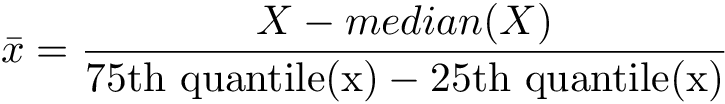
Here’s what you need to remember about standardization:

* It scales the variance at 1.
* It centers the mean at 0.
* It preserves the shape of the original distribution.
* It preserves outliers if they exist.
* Minimum and maximum values vary.

|  |
| --- |
| s |
|  | import pandas as pd |
|  | from sklearn.preprocessing import StandardScaler |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | # create the scaler object |
|  | scaler = StandardScaler() |
|  |  |
|  | # fit the scaler to the train data |
|  | scaler.fit(train) |
|  |  |
|  | # transform train and test data |
|  | train\_scaled = scaler.transform(train) |
|  | test\_scaled = scaler.transform(test) |

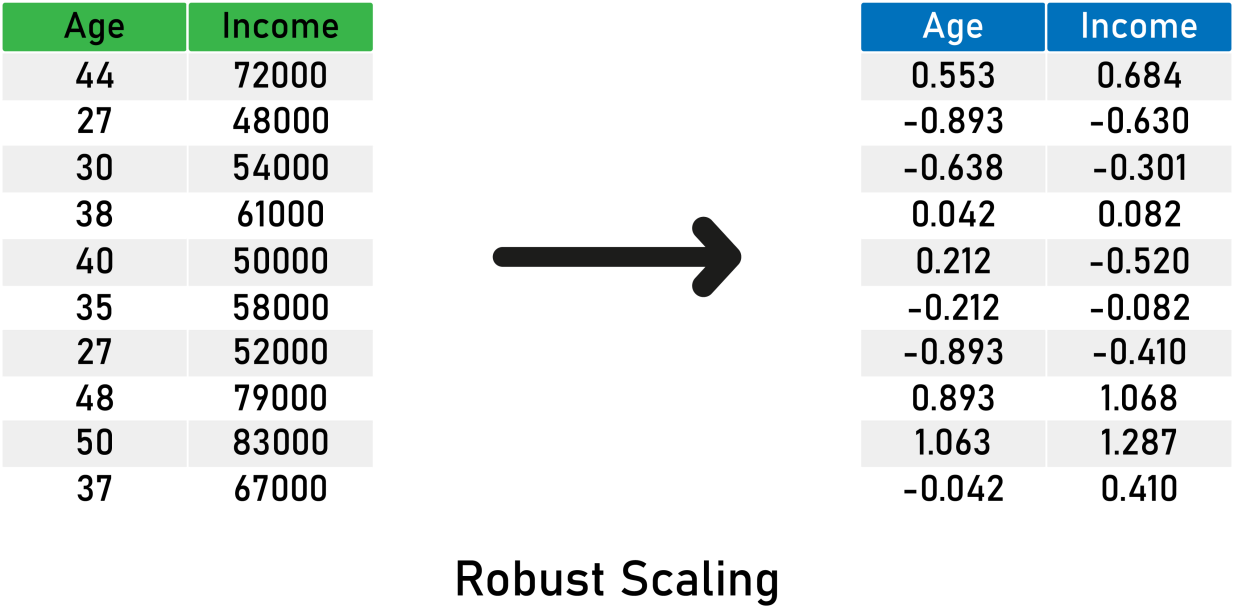
# **Robust Scaling (scaling to median and IQR)**

In this method, the median is used instead of the mean. We remove the median from the variable observations, and then we scale to the inter-quantile range (IQR).



*The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile).*

Here’s an illustration of robust scaling:



Here’s what you need to remember about robust scaling:

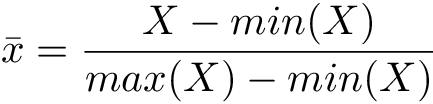
* It centers the medianat 0 .
* The resulted variance varies across variables.
* It may not preserve the shape of the original distribution.
* The minimum and maximum values vary.
* It is robust to outliers.

And a code snippet with the [RobustScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html" \l "sklearn.preprocessing.RobustScaler" \t "https://heartbeat.fritz.ai/_blank) from sklearn:

|  |
| --- |
|  |
|  | import pandas as pd |
|  | from sklearn.preprocessing import RobustScaler |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | # create the scaler object |
|  | robust = RobustScaler() |
|  |  |
|  | # fit the scaler to the train data |
|  | scaler.fit(train) |
|  |  |
|  | # transform train and test data |
|  | train\_scaled = robust.transform(train) |
|  | test\_scaled = robust.transform(test) |

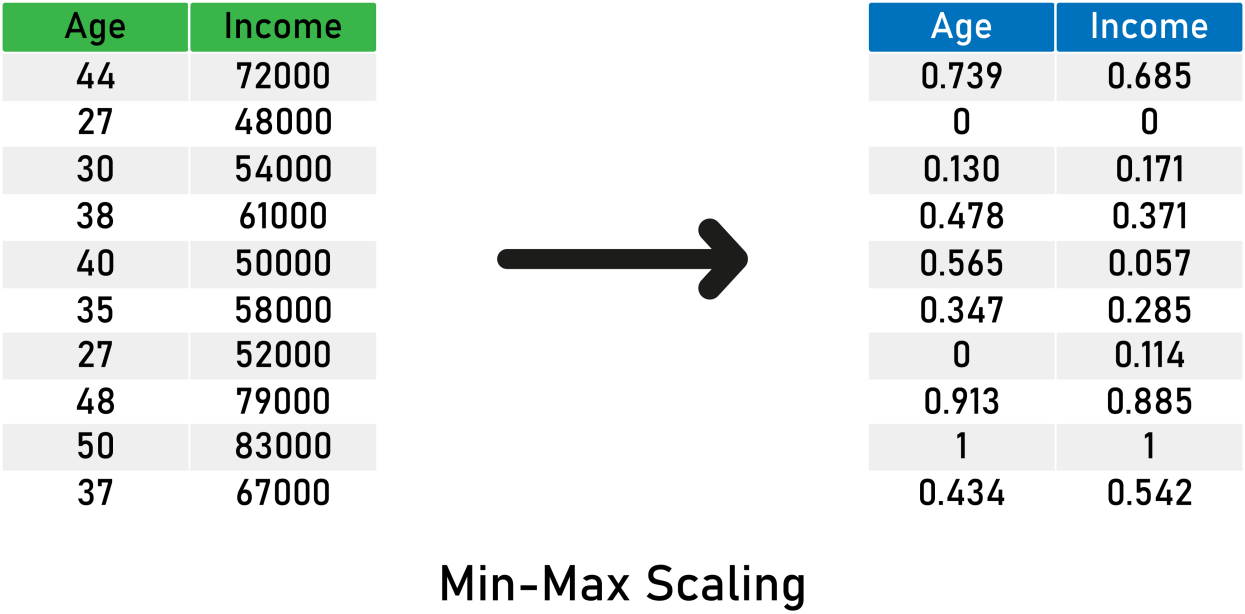
# **Min-Max Scaling**

Minimum and maximum scaling both compress the values between 0 and 1. It subtracts the minimum value from all the variable observations, and then divides it by the variable’s value range:



*This technique will not******normalize the variable distribution.*****

Here is an example using MinMax Scaling:



Here’s what you need to remember about Min-Max Scaling:

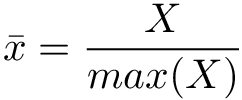
* It does not center the mean at 0.
* It makes the variance vary across variables.
* It may not maintain the shape of the original distribution.
* The minimum and maximum values are in the range of [0,1].
* This method is very sensitive to outliers.

And a code snippet with [MinMaxScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html" \l "sklearn.preprocessing.MinMaxScaler" \t "https://heartbeat.fritz.ai/_blank) from sklearn:

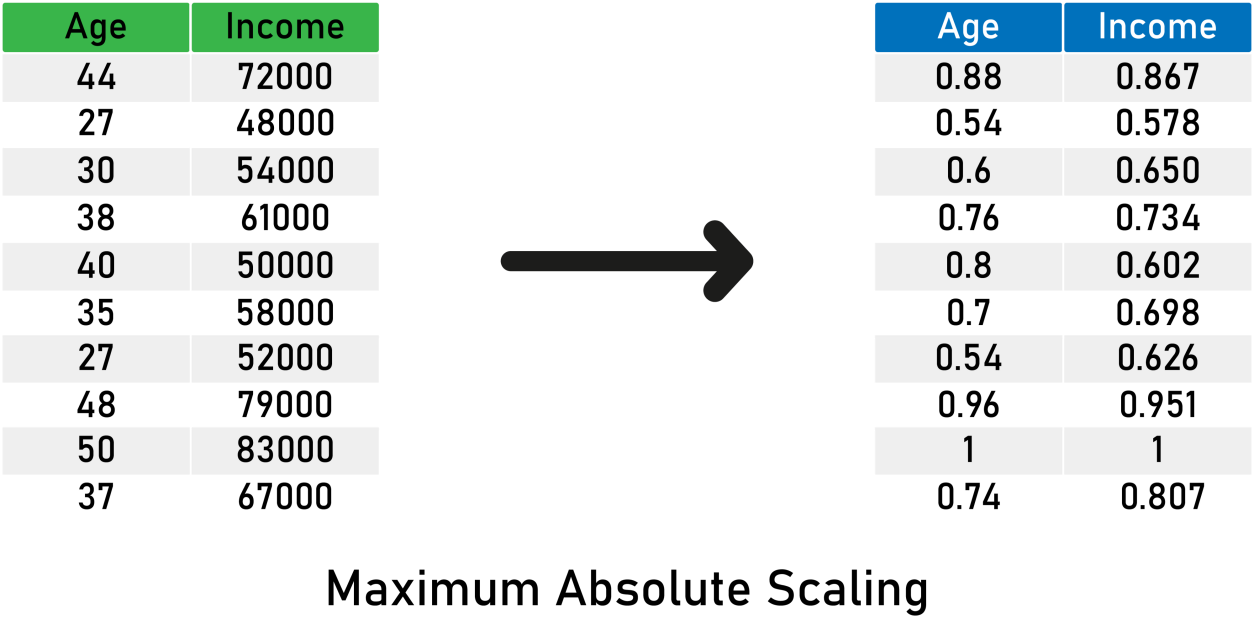
|  |
| --- |
| s |
|  | import pandas as pd |
|  | from sklearn.preprocessing import MinMaxScaler |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | # create the scaler object |
|  | min\_max = MinMaxScaler() |
|  |  |
|  | # fit the scaler to the train data |
|  | min\_max.fit(train) |
|  |  |
|  | # transform train and test data |
|  | train\_scaled = min\_max.transform(train) |
|  | test\_scaled = min\_max.transform(test) |

# **Maximum Absolute Scaling**

Maximum absolute scaling scales the variable values between -1 and 1 by dividing the data by its maximum value:



Here’s an illustration of the concept:



Here is what you need to remember about Maximum Absolute Scaling:

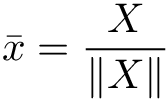
* The resulting mean is not centered.
* It doesn't scale the variance.
* It’s sensitive to outliers.

Here’s a code snippet with the [MaxAbsScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html" \l "sklearn.preprocessing.MinMaxScaler" \t "https://heartbeat.fritz.ai/_blank) from sklearn:

|  |
| --- |
|  |
|  |
|  | import pandas as pd |
|  | from sklearn.preprocessing import MaxAbsScaler |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | # create the scaler object |
|  | max\_abs = MaxAbsScaler() |
|  |  |
|  | # fit the scaler to the train data |
|  | max\_abs.fit(train) |
|  |  |
|  | # transform train and test data |
|  | train\_scaled = max\_abs.transform(train) |
|  | test\_scaled = max\_abs.transform(test) |

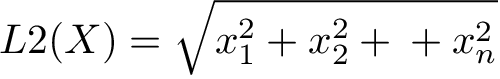
# **Scaling to vector unit norm**

In scale to the vector unit norm, we divide each feature’s vector by the distance of the vector, as shown below:



For the distance measure, you can use either:

* ****The**[Euclidean distance](https://en.wikipedia.org/wiki/Euclidean_distance" \t "https://heartbeat.fritz.ai/_blank)** ( — or ****L2**** norm) with the formula:

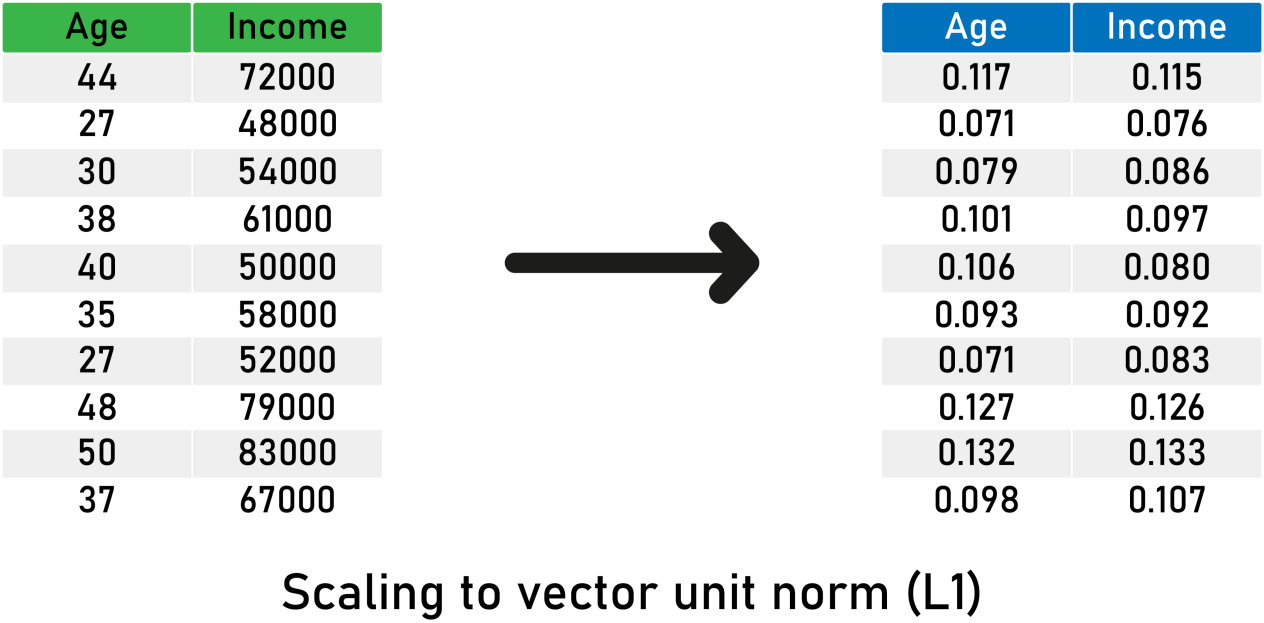


* ****The**[Manhattan distance](https://en.wikipedia.org/wiki/Taxicab_geometry" \t "https://heartbeat.fritz.ai/_blank)** ( — or ****L1**** norm) with the formula:

IMG_273

*Euclidean distance******squares******the values of the feature vector components; therefore,******outliers******will have a heavier weight. For that, if you have outliers, try to use L1 normalization instead.*

Here is an example with ****L1 Norm****:



Here’s what you need to remember about scaling to the vector unit norm:

* The length of the resulting vector is 1.
* It normalizes the ****feature**** vector and not the ****observation**** vector.
* It’s sensitive to outliers.
* Recommended for text classification and clustering.

Here’s a code snippet using the [Normalizer](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Normalizer.html" \l "sklearn.preprocessing.Normalizer" \t "https://heartbeat.fritz.ai/_blank) object from sklearn:

|  |
| --- |
|  |
|  | import pandas as pd |
|  | from sklearn.preprocessing import Normalizer |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("yourData.csv") |
|  |  |
|  | # create the scaler object with your prefered norm. |
|  | max\_abs = Normalizer(norm = 'l2') |
|  |  |
|  | # fit the scaler to the train data |
|  | max\_abs.fit(train) |
|  |  |
|  | # transform train and test data |
|  | train\_scaled = max\_abs.transform(train) |
|  | test\_scaled = max\_abs.transform(test) |

# Handling Date-time and Mixed Variables

# **Engineering Variables of Date and Time**

Date and time variables are good sources of information, with each number corresponding to a specific part of the date and the time.

In real-world datasets, we can find date-time variables in many formats like the following:

* Time of birth (19:45:57)
* Birthday date (16–08–1995, 18–04–1997)
* Invoice date (03–06–2020 19:47:29)

## **Date Variable**

When you have date variables, you can extract many features to enrich your feature’s set—here are some examples of what we can extract:

* Day of the week (Thursday, Wednesday).
* Day of the month (16,02).
* Month (January, August), or numbers (1,8).
* Year (2020, 1997, 2009).
* Week of the year (2 would represent the second week of January).
* Season of the year (Summer, Winter).
* Quarter.
* Semester.
* Is the day a weekend (True for Saturday, False for Thursday)
* Public holiday or not.
* Leap year or not.
* Difference between two dates.

|  |  |
| --- | --- |
|  | # import the libraries |
|  | import pandas as pd |
|  | import numpy as np |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("data.csv") |
|  |  |
|  | # example of 'payment' date feature |
|  | # convert the feature to datetime feature type |
|  | data['payment\_datetime'] = pd.to\_datetime(data['payment']) |
|  |  |
|  | # extracting the day in numerical format (1 to 31) |
|  | data['payment\_day'] = data['payment\_datetime'].dt.day |
|  |  |
|  | # extracting the day of the week (0 to 6) |
|  | data['payment\_dayofweek'] = data['payment\_datetime'].dt.dayofweek |
|  |  |
|  | # extracting the day of the week (Thursday, Wednesday...) |
|  | data['payment\_dayofweek\_name'] = data['payment\_datetime'].dt.weekday\_name |
|  |  |
|  | # extracting the month. (1 to 12) |
|  | data['payment\_month'] = data['payment\_datetime'].dt.month |
|  |  |
|  | # extracting the year. |
|  | data['payment\_year'] = data['payment\_datetime'].dt.year |
|  |  |
|  | # extracting the week of the year. (1 to 52) |
|  | data['payment\_week'] = data['payment\_datetime'].dt.week |
|  |  |
|  | # extracting the quarter (1 to 4). |
|  | data['payment\_quarter'] = data['payment\_datetime'].dt.quarter |
|  |  |
|  | # extracting the semester (1 or 2). |
|  | data['payment\_semester'] = np.where(data['payment\_datetime'].isin([1,2]), 1, 2) |
|  |  |
|  | # extracting if the day is weekend (True or false). |
|  | data['payment\_weekend'] = np.where(data['payment\_dayofweek\_name'].isin(['Sunday', 'Saturday']), 1,0) |
|  |  |
|  | # extracting the difference in days between two payments |
|  | data["payment\_difference"] = (data['first\_payment'] - data['second\_payment']).dt.days |

## **Time Variables**

The same thing applies to time features—you can extract many features, such as:

* Seconds
* Minutes
* Hour
* [Unix Timestamp](https://www.unixtimestamp.com/" \t "https://heartbeat.fritz.ai/_blank)
* Business hours—open or closed

You can also denote time differences in multiple formats. For example, the time between a first\_payment and a second\_payment can be expressed in hours, minutes, and seconds as well.

|  |
| --- |
|  |
|  | import pandas as pd |
|  | import numpy as np |
|  |  |
|  | # read your data |
|  | data = pd.read\_csv("data.csv") |
|  |  |
|  | # example of 'payment' date feature |
|  | # convert the feature to datetime feature type |
|  | data['payment\_datetime'] = pd.to\_datetime(data['payment']) |
|  |  |
|  | # extracting the hour(0 to 23) |
|  | data['payment\_hour'] = data['payment\_datetime'].dt.hour |
|  |  |
|  | # extracting the minutes(0 to 59) |
|  | data['payment\_minute'] = data['payment\_datetime'].dt.minute |
|  |  |
|  | # extracting the seconds(0 to 59) |
|  | data['payment\_second'] = data['payment\_datetime'].dt.second |
|  |  |
|  | # extracting the Unix time stamp (number of seconds since January 1st, 1970 at UTC) |
|  | data['payment\_unix\_timestamp'] = (data['payment\_datetime'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s') |
|  |  |
|  | # extracting the is buissness hour ( for example, from 8AM to 12AM) (1 or 0). |
|  | data['payment\_buissness'] = np.where(data['payment\_hour'].isin([8, 9, 10, 12]), 1, 0) |
|  |  |
|  | # calculating differences |
|  | payment\_difference = data['first\_payment'] - data['second\_payment'] |
|  |  |
|  | # extracting the payment differencce in seconds. |
|  | data['payment\_difference\_seconds'] = payment\_difference / np.timedelta64(1, 's') |
|  |  |
|  | # extracting the payment differencce in minutes. |
|  | data['payment\_difference\_minutes'] = payment\_difference / np.timedelta64(1, 'm') |
|  |  |
|  | # extracting the payment differencce in hours. |
|  | data['payment\_difference\_hours'] = payment\_difference / np.timedelta64(1, 'h') |
|  |  |
|  | # extracting the payment differencce in milliseconds. |
|  | data['payment\_difference\_milliseconds'] = payment\_difference / np.timedelta64(1, 'ms') |