**Data Preprocessing**

* **Loading the data:**

df = pd.read\_csv("path for the .csv file to be uploaded")

df = pd.read\_excel("path for the .xlsx file")

* **Checking the data:**

df.shape - To check the number of rows and columns

df.columns.values - What are the column names?, Sometimes import doesn’t consider column names while importing

df.head(n) - First n observations of data

df.tail(n) - Last n observations of the data

df.dtypes - Data types of all variables

df.describe() - Summary of all variables

df[‘custId’].describe() - Summary of a variable

df.columnname.value\_counts() - Get frequency table for a given variable

table(df$columnnanme) - Get frequency tables for categorical variable

sum(df.columnname.isnull()) - Missing value count in a variable

df.sample(n=10) - Take a random sample of size 10

* **Missing values:**

isnull().sum() - count missing values per column.

dropna() - drop rows with missing values.

dropna(axis=1) - drop columns that has atleast one NaN.

dropna(how="all") - drop rows where all rows are NaN.

dropna(thresh=4) - drop rows that have atleast 4 NaN.

* **Duplicate values:**

1. **full data**

df=df.duplicated()

sum(df) - some int value

df\_uniq=df.drop\_duplicates()

1. **Column wise**

df=df.column\_name.duplicated()

sum(df) - some int value

df\_uniq=df.drop\_duplicates(['column\_name'])

* **Subsetting:**

We can subset our data based on the column names,values and other attributes based on the requirement.

Example -

* Select first 1000 rows only

bank\_data1 = bank\_data.head(1000)

* Select only four columns “Cust\_num” “age” “default” and “balance”

bank\_data2 = bank\_data[["Cust\_num", "age","default","balance"]]

* Select 20,000 to 40,000 observations along with four variables “Cust\_num” “job” “marital” and “education”

bank\_data3 = bank\_data[["Cust\_num", "job","marital","education"]][20000:40000]

* Select 5000 to 6000 observations drop “poutcome“ and “y”

bank\_data4=bank\_data.drop(['poutcome','y'], axis=1)[5000:6000]

* bank\_subset1=bank\_data[(bank\_data['age']>40) & (bank\_data['loan']=="no")]
* bank\_subset2=bank\_data[(bank\_data['age']>40) | (bank\_data['loan']=="no"
* bank\_subset3= bank\_data[(bank\_data['age']>40) & (bank\_data['loan']=="no") | (bank\_data['marital']=="single" )]
* **Sorting :**

df=df.sort('column\_name',ascending=False)

df=df.sort\_index(axis=1, ascending=True)

df.sort\_values(by='column\_name',ascending=False)

* **Joining or Merging:**
* **INNER JOIN**

inner\_df = pd.merge(Table1, Table2, on='column\_name in both tables', how='inner')

* **OUTER JOIN**

outer\_df = pd.merge(Table1, Table2, on='column\_name in both tables', how='outer')

* **LEFT JOIN**

left\_df = pd.merge(Table1, Table2, on='column\_name in both tables', how='left')

* **RIGHT JOIN**

right\_df = pd.merge(Table1, Table2, on='column\_name in both tables', how='right')

* **Splitting into training, testing and validation sets:**

X=df\_data.iloc[:,1:].values (All columns and rows except first one)

Y=df\_data.iloc[:,0].values (First column and all rows)

X-train,X-test,Y\_train,Y\_test

= train\_test\_split(X,Y,test\_size=0.3,random\_state=0)

* **Categorical data:**

If you are familiar with machine learning, you will probably have encountered categorical features in many datasets. These generally include different categories or levels associated with the observation, which are non-numerical and thus need to be converted so the computer can process them.

**i) One-Hot Encoding:**

The input to this transformer should be an array-like of integers or strings, denoting the values taken on by categorical (discrete) features. The features are encoded using a one-hot (aka ‘one-of-K’ or ‘dummy’) encoding scheme. This creates a binary column for each category and returns a sparse matrix or dense array.

By default, the encoder derives the categories based on the unique values in each feature. Alternatively, you can also specify the categories manually. The OneHotEncoder previously assumed that the input features take on values in the range [0, max(values)). This behaviour is deprecated.

This encoding is needed for feeding categorical data to many scikit-learn estimators, notably linear models and SVMs with the standard kernels.

**Syntax:**

from sklearn.preprocessing import OneHotEncoder

enc = OneHotEncoder(n\_values=None, categorical\_features=None, categories=None, drop=None, sparse=True, dtype=<class ‘numpy.float64’>, handle\_unknown=’error’)

X = categories along with their numbers

enc.fit(X)

**Parameters:**

* **categories :** ‘auto’ or a list of lists/arrays of values, default=’auto’. - Categories (unique values) per feature:
* **auto :** Determine categories automatically from the training data.
* **list :** categories[i] holds the categories expected in the ith column. The passed categories should not mix strings and numeric values within a single feature, and should be sorted in case of numeric values.
* **drop : ‘**first’ or a list/array of shape (n\_features,), default=None. - Specifies a methodology to use to drop one of the categories per feature. This is useful in situations where perfectly collinear features cause problems, such as when feeding the resulting data into a neural network or an unregularized regression.
* **None :** retain all features (the default).
* **first :** drop the first category in each feature. If only one category is present, the feature will be dropped entirely.
* **array :** drop[i] is the category in feature X[:, i] that should be dropped**.**
* **sparse :** boolean, default=True - Will return sparse matrix if set True else will return an array.
* **dtype :** number type, default=np.float - Desired dtype of output.
* **handle\_unknown :** ‘error’ or ‘ignore’, default=’error’. - Whether to raise an error or ignore if an unknown categorical feature is present during transform (default is to raise). When this parameter is set to ‘ignore’ and an unknown category is encountered during transform, the resulting one-hot encoded columns for this feature will be all zeros. In the inverse transform, an unknown category will be denoted as None.
* **n\_values :** ‘auto’, int or array of ints, default=’auto’- Number of values per feature.
* **auto:** determine value range from training data.
* **int** : number of categorical values per feature.Each feature value should be in range(n\_values)
* **array :** n\_values[i] is the number of categorical values in X[:, i]. Each feature value should be in range(n\_values[i])
* **categorical\_features :** ‘all’ or array of indices or mask, default=’all’ - Specify what features are treated as categorical.
* **all:** All features are treated as categorical.
* **array of indices:** Array of categorical feature indices.
* **mask:** Array of length n\_features and with dtype=bool.
* Non-categorical features are always stacked to the right of the matrix.

**Attributes:**

* **categories\_ :** list of arrays - The categories of each feature determined during fitting (in order of the features in X and corresponding with the output of transform). This includes the category specified in drop (if any).
* **drop\_idx\_ :** array of shape (n\_features,) - drop\_idx\_[i] is the index in categories\_[i] of the category to be dropped for each feature. None if all the transformed features will be retained.
* **active\_features\_ :** array - Indices for active features, meaning values that actually occur in the training set. Only available when n\_values is 'auto'.
* **feature\_indices\_ :** array of shape (n\_features,) - Indices to feature ranges. Feature i in the original data is mapped to features from feature\_indices\_[i] to feature\_indices\_[i+1] (and then potentially masked by active\_features\_ afterwards)
* **n\_values\_ :** array of shape (n\_features,) - aximum number of values per feature.

**Methods:**

|  |  |
| --- | --- |
| [**fit**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder.fit)(self, X[, y]) | Fit OneHotEncoder to X. |
| [**fit\_transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder.fit_transform)(self, X[, y]) | Fit OneHotEncoder to X, then transform X. |
| [**get\_feature\_names**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder.get_feature_names)(self[, input\_features]) | Return feature names for output features. |
| [**get\_params**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder.get_params)(self[, deep]) | Get parameters for this estimator. |
| [**inverse\_transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder.inverse_transform)(self, X) | Convert the back data to the original representation. |
| [**set\_params**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder.set_params)(self, \\*\\*params) | Set the parameters of this estimator. |
| [**transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder.transform)(self, X) | Transform X using one-hot encoding. |

**Example:**

from sklearn.preprocessing import OneHotEncoder

enc = OneHotEncoder(handle\_unknown='ignore')

X = [['Male', 1], ['Female', 3], ['Female', 2]]

enc.fit(X)

OneHotEncoder(categorical\_features=None, categories=None, drop=None,dtype=<... 'numpy.float64'>, handle\_unknown='ignore',n\_values=None, sparse=True)

enc.categories\_

[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]

enc.transform([['Female', 1], ['Male', 4]]).toarray()

array([[1., 0., 1., 0., 0.],[0., 1., 0., 0., 0.]])

enc.inverse\_transform([[0, 1, 1, 0, 0], [0, 0, 0, 1, 0]])

array([['Male', 1],[None, 2]], dtype=object)

enc.get\_feature\_names()

array(['x0\_Female', 'x0\_Male', 'x1\_1', 'x1\_2', 'x1\_3'], dtype=object)

drop\_enc = OneHotEncoder(drop='first').fit(X)

drop\_enc.categories\_

[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]

drop\_enc.transform([['Female', 1], ['Male', 2]]).toarray()

array([[0., 0., 0.],[1., 1., 0.]])

**ii) Label Encoding:**

Label encoder will helps us in encoding the categorical variables with value between 0 and n\_classes-1.

**Syntax:**

from sklearn.preprocessing import LabelEncoder

label\_encoder=LabelEncoder()

input\_classes=categorical variables

label\_encoder.fit(input\_classes)

label\_encoder.classes\_

**Attributes:**

* **classes\_ :** array of shape (n\_class,) - Holds the label for each class.

**Methods:**

|  |  |
| --- | --- |
| [**fit**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.fit)(self, y) | Fit label encoder |
| [**fit\_transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.fit_transform)(self, y) | Fit label encoder and return encoded labels |
| [**get\_params**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.get_params)(self[, deep]) | Get parameters for this estimator. |
| [**inverse\_transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.inverse_transform)(self, y) | Transform labels back to original encoding. |
| [**set\_params**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.set_params)(self, \\*\\*params) | Set the parameters of this estimator. |
| [**transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.transform)(self, y) | Transform labels to normalized encoding. |

**Examples:**

from sklearn.preprocessing import LabelEncoder

label\_encoder=LabelEncoder()

input\_classes=['Havells','Philips','Syska','Eveready','Lloyd']

label\_encoder.fit(input\_classes)

for i,item in enumerate(label\_encoder.classes\_):

print(item,'-->',i)

Eveready --> 0

Havells --> 1

Lloyd --> 2

Philips --> 3

Syska --> 4

**Differences:**

* LabelEncoder can turn [dog,cat,dog,mouse,cat] into [1,2,1,3,2], but then the imposed ordinality means that the average of dog and mouse is cat. Still there are algorithms like decision trees and random forests that can work with categorical variables just fine and LabelEncoder can be used to store values using less disk space.
* One-Hot-Encoding has the advantage that the result is binary rather than ordinal and that everything sits in an orthogonal vector space. The disadvantage is that for high cardinality, the feature space can really blow up quickly and you start fighting with the curse of dimensionality. In these cases, I typically employ one-hot-encoding followed by PCA for dimensionality reduction. I find that the judicious combination of one-hot plus PCA can seldom be beat by other encoding schemes. PCA finds the linear overlap, so will naturally tend to group similar features into the same feature.
* **Feature Scaling:**

It is a step of Data Pre Processing which is applied to independent variables or features of data. It basically helps to normalize the data within a particular range. Sometimes, it also helps in speeding up the calculations in an algorithm.

There are two different approaches to bring the different features onto the same scale:

i)Normalization

ii)Standardization

**Normalization:**

Normalization refers to rescaling real valued numeric attributes into the range 0 and 1.This is also called as min-max scaling.

[https://www.statisticshowto.datasciencecentral.com/wp-content/uploads/2015/11/normalize-data.png](https://www.statisticshowto.datasciencecentral.com/wp-content/uploads/2015/11/normalize-data.png)

**Syntax:**

from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()

X\_train\_norm = mms.fit\_transform(X\_train)

X\_test\_norm = mms.transform(X\_test)

**Parameters:**

* **feature\_range :** tuple (min, max), default=(0, 1) - Desired range of transformed data.
* **copy :** boolean, optional, default True - Set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array).

`

**Attributes:**

* **min\_ :** ndarray, shape (n\_features,) - Per feature adjustment for minimum. Equivalent to min - X.min(axis=0) \* self.scale\_
* **scale\_ :** ndarray, shape (n\_features,) - Per feature relative scaling of the data. Equivalent to (max - min) / (X.max(axis=0) - X.min(axis=0))
* **data\_min\_** : ndarray, shape (n\_features,) - Per feature minimum seen in the data
* **data\_max\_ :** ndarray, shape (n\_features,) - Per feature maximum seen in the data
* **data\_range\_ :** ndarray, shape (n\_features,) - Per feature range (data\_max\_ - data\_min\_) seen in the data

**Examples:**

from sklearn.preprocessing import MinMaxScaler

data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]

scaler = MinMaxScaler()

print(scaler.fit(data))

MinMaxScaler(copy=True, feature\_range=(0, 1))

print(scaler.data\_max\_)

[ 1. 18.]

print(scaler.transform(data))

[[0. 0. ]

[0.25 0.25]

[0.5 0.5 ]

[1. 1. ]]

print(scaler.transform([[2, 2]]))

[[1.5 0. ]]

**Methods:**

|  |  |
| --- | --- |
| [**fit**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler.fit)(self, X[, y]) | Compute the minimum and maximum to be used for later scaling. |
| [**fit\_transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler.fit_transform)(self, X[, y]) | Fit to data, then transform it. |
| [**get\_params**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler.get_params)(self[, deep]) | Get parameters for this estimator. |
| [**inverse\_transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler.inverse_transform)(self, X) | Undo the scaling of X according to feature\_range. |
| [**partial\_fit**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler.partial_fit)(self, X[, y]) | Online computation of min and max on X for later scaling. |
| [**set\_params**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler.set_params)(self, \\*\\*params) | Set the parameters of this estimator. |
| [**transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler.transform)(self, X) | Scaling features of X according to feature\_range. |

**Standardization:**

Standardization refers to shifting the distribution of each attribute to have a mean of zero and a standard deviation of one (unit variance).

http://ci.columbia.edu/ci/premba_test/c0331/images/s6/5836240103.gif

x = the value that is being standardized  
m = the mean of the distribution  
s = standard deviation of the distribution

**Syntax:**

from sklearn.preprocessing import StandardScaler

x=array[:,0:8] input variables

scaler=StandardScaler().fit(x)

rescaledX=scaler.transform(x)

rescaledX[0:5,:]

**Parameters:**

* **copy :** boolean, optional, default True - If False, try to avoid a copy and do inplace scaling instead. This is not guaranteed to always work inplace; e.g. if the data is not a NumPy array or scipy.sparse CSR matrix, a copy may still be returned.
* **with\_mean** : boolean, True by default - If True, center the data before scaling. This does not work (and will raise an exception) when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory.
* **with\_std** : boolean, True by default - If True, scale the data to unit variance (or equivalently, unit standard deviation).

**Attributes:**

* **cale\_ :** ndarray or None, shape (n\_features,) - Per feature relative scaling of the data. This is calculated using np.sqrt(var\_). Equal to None when with\_std=False.
* **mean\_ :** ndarray or None, shape (n\_features,) - The mean value for each feature in the training set. Equal to None when with\_mean=False.
* **var\_ :** ndarray or None, shape (n\_features,) - The variance for each feature in the training set. Used to compute scale\_. Equal to None when with\_std=False**.**
* **n\_samples\_seen\_ :** int or array, shape (n\_features,)- The number of samples processed by the estimator for each feature. If there are not missing samples, the n\_samples\_seen will be an integer, otherwise it will be an array. Will be reset on new calls to fit, but increments across partial\_fit calls.

**Methods:**

|  |  |
| --- | --- |
| [**fit**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler.fit)(self, X[, y]) | Compute the mean and std to be used for later scaling. |
| [**fit\_transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler.fit_transform)(self, X[, y]) | Fit to data, then transform it. |
| [**get\_params**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler.get_params)(self[, deep]) | Get parameters for this estimator. |
| [**inverse\_transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler.inverse_transform)(self, X[, copy]) | Scale back the data to the original representation |
| [**partial\_fit**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler.partial_fit)(self, X[, y]) | Online computation of mean and std on X for later scaling. |
| [**set\_params**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler.set_params)(self, \\*\\*params) | Set the parameters of this estimator. |
| [**transform**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler.transform)(self, X[, copy]) | Perform standardization by centering and scaling |

**Example:**

from sklearn.preprocessing import StandardScaler

data = [[0, 0], [0, 0], [1, 1], [1, 1]]

scaler = StandardScaler()

print(scaler.fit(data))

StandardScaler(copy=True, with\_mean=True, with\_std=True)

print(scaler.mean\_)

[0.5 0.5]

print(scaler.transform(data))

[[-1. -1.]

[-1. -1.]

[ 1. 1.]

[ 1. 1.]]

print(scaler.transform([[2, 2]]))

[[3. 3.]]

**Examples of Algorithms where Feature Scaling matters:**

1. K-Means uses the Euclidean distance measure here feature scaling matters.

2. K-Nearest-Neighbours also require feature scaling.

3. Principal Component Analysis (PCA): Tries to get the feature with maximum variance, here too feature scaling is required.

4. Gradient Descent: Calculation speed increase as Theta calculation becomes faster after feature scaling.

**Note:** Naive Bayes, Linear Discriminant Analysis, and Tree-Based models are not affected by feature scaling. In Short, any Algorithm which is Not Distance based is Not affected by Feature Scaling.