Data Preprocessing

Definition:

Data preprocessing is a data mining technique that involves transforming raw data into an understandable format. Real-world data is often incomplete, inconsistent, and/or lacking in certain behaviors or trends, and is likely to contain many errors. Data preprocessing is a proven method of resolving such issues. Data preprocessing prepares raw data for further processing.

Data goes through a series of steps during preprocessing:

- **Data Cleaning**: Data is cleansed through processes such as filling in missing values, smoothing the noisy data, or resolving the inconsistencies in the data.
- **Data Integration**: Data with different representations are put together and conflicts within the data are resolved.
- **Data Transformation**: Data is normalized, aggregated and generalized.
- **Data Reduction**: This step aims to present a reduced representation of the data in a data warehouse.
- **Data Discretization:** Involves the reduction of a number of values of a continuous attribute by dividing the range of attribute intervals.

Here are some of the steps in python to preprocess the data.

- ## 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:**

i) full data

df=df.duplicated()
sum(df) - some int value
df_uniq=df.drop_duplicates()

ii) 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(self, X[, y])	Fit OneHotEncoder to X.
fit_transform(self, X[, y])	Fit OneHotEncoder to X, then transform X.
get_feature_names(self[, input_features])	Return feature names for output features.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Convert the back data to the original representation.
set_params(self, **params)	Set the parameters of this estimator.
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(self, y)	Fit label encoder
fit_transform(self, y)	Fit label encoder and return encoded labels
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, y)	Transform labels back to original encoding.
set_params(self, **params)	Set the parameters of this estimator.
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.

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

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(self, X[, y])	Compute the minimum and maximum to be used for later scaling.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Undo the scaling of X according to feature_range.
partial_fit(self, X[, y])	Online computation of min and max on X for later scaling.
set_params(self, **params)	Set the parameters of this estimator.
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).

$$Z = \frac{X - \mu}{\sigma}$$

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(self, X[, y])	Compute the mean and std to be used for later scaling.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X[, copy])	Scale back the data to the original representation
partial_fit(self, X[, y])	Online computation of mean and std on X for later scaling.
set_params(self, **params)	Set the parameters of this estimator.
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-Neighbors 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.

- ## Dimensionality Reduction:

In machine learning classification problems, there are often too many factors on the basis of which the final classification is done. These factors are basically variables called features. The higher the number of features, the harder it gets to visualize the training set and then work on it. Sometimes, most of these features are correlated, and hence redundant. This is where dimensionality reduction algorithms come into play. Dimensionality reduction is the process of reducing the number of random variables under consideration, by obtaining a set of principal variables.

It can be divided into

1) **Feature Selection**
2) **Feature Extraction.**

Feature selection: In this, we try to find a subset of the original set of variables, or features, to get a smaller subset which can be used to model the problem. It usually involves three ways:

1) Filter

2) Wrapper

3) Embedded

Filter:

Filter methods are generally used as a preprocessing step. The selection of features is independent of any machine learning algorithms. Instead, features are selected on the basis of their scores in various statistical tests for their correlation with the outcome variable. The correlation is a subjective term here. For basic guidance, you can refer to the following table for defining correlation co-efficient.

Feature\Response	Continuous	Categorical
Continuous	Pearson's Correlation	LDA
Categorical	Anova	Chi-Square

Pearson's Correlation: It is used as a measure for quantifying linear dependence between two continuous variables X and Y. Its value varies from -1 to +1. Pearson's correlation is given as:

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y}$$

Syntax:

```
from scipy.stats.stats import pearsonr
X,Y= continous variables of same length
pearsonr(X, Y)
```

LDA (Linear Discriminant Analysis): A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule. The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix. The fitted model can also be used to reduce the dimensionality of the input by projecting it to the most discriminative directions

Syntax:

```
import numpy as np
from sklearn.lda import LDA
lda = LDA(n_components=None, priors=None, shrinkage=None,
solver='svd',store_covariance=False, tol=0.0001)
X_train_lda,Y_train_lda=()lda.fit_transform(X_train,Y_train)
```

Parameters:

- **solver :** string, optional - Solver to use, possible values:
  - ✓ **svd':** Singular value decomposition (default). Does not compute thecovariance matrix, therefore this solver is recommended for data with a large number of features.
  - ✓ **'lsqr'**: Least squares solution, can be combined with shrinkage.
  - ✓ **'eigen':** Eigenvalue decomposition, can be combined with shrinkage**.**

- **shrinkage :** string or float, optional - Shrinkage parameter, possible values:
  - ✓ **None:** no shrinkage (default).
  - ✓ **'auto':** automatic shrinkage using the Ledoit-Wolf lemma.

- ✓ **float between 0 and 1**: fixed shrinkage parameter.
- ✓ Note that shrinkage works only with 'lsqr' and 'eigen' solvers.

- **priors :** array, optional, shape (n_classes,) - Class priors.

- **n_components :** int, optional - Number of components (< n_classes - 1) for dimensionality reduction.

- **store_covariance :** bool, optional - Additionally compute class covariance matrix (default False).

Attributes:

- **coef_** : array, shape (n_features,) or (n_classes, n_features) - Weight vector(s).

- **intercept_** : array, shape (n_features,) - Intercept term.

- **covariance_** : array-like, shape (n_features, n_features) - Covariance matrix (shared by all classes).

- **means_** : array-like, shape (n_classes, n_features) - Class means.

- **priors_** : array-like, shape (n_classes,) - Class priors (sum to 1).

- **scalings_** : array-like, shape (rank, n_classes - 1) - Scaling of the features in the space spanned by the class centroids.

- **xbar_** : array-like, shape (n_features,) - Overall mean.

- **classes_** : array-like, shape (n_classes,) - Unique class labels.

Methods:

decision_function(X)	Predict confidence scores for samples.
fit(X, y[, store_covariance, tol])	Fit LDA model according to the given training data and parameters.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict class labels for samples in X.
predict_log_proba(X)	Estimate log probability.
predict_proba(X)	Estimate probability.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Project data to maximize class separation.

Example:

```
import numpy as np
from sklearn.lda import LDA
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
y = np.array([1, 1, 1, 2, 2, 2])
clf = LDA()
clf.fit(X, y)
LDA(n_components=None, priors=None, shrinkage=None,
solver='svd',store_covariance=False, tol=0.0001)
print(clf.predict([[-0.8, -1]]))
```

ANOVA:

ANOVA stands for Analysis of variance. It is similar to LDA except for the fact that it is operated using one or more **categorical independent features** and one

continuous dependent feature. It provides a statistical test of whether the means of several groups are equal or not.

Syntax:

from sklearn.feature_selection import f_classif

X= The set of regressors that will be tested sequentially.

Y= The data matrix.

classif(X, Y)

Attributes:

- **F :** array, shape = [n_features,] - The set of F values.
- **pval :** array, shape = [n_features,] - The set of p-values.

Chi-Square:

It is a statistical test applied to the groups of categorical features to evaluate the likelihood of correlation or association between them using their frequency distribution.

Syntax:

from sklearn.feature_selection import chi2

X= Sample vectors.

Y= arget vector (class labels).

chi2(X, Y)

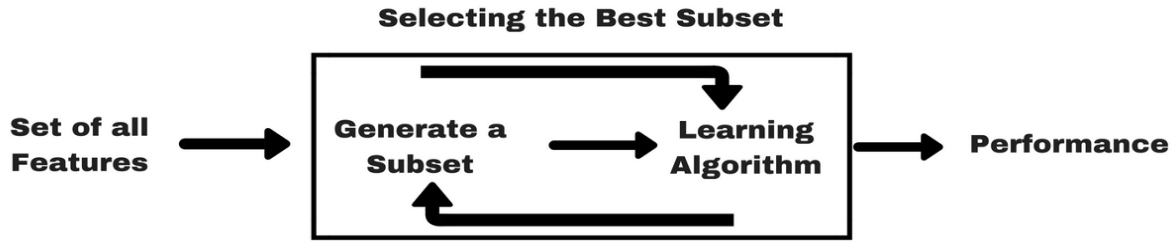
Attributes:

- **chi2 :** array, shape = (n_features,) - chi2 statistics of each feature.
- **pval :** array, shape = (n_features,) - p-values of each feature.

NOTE: One thing that should be kept in mind is that filter methods do not remove multicollinearity. So, you must deal with multicollinearity of features as well before training models for your data.

Wrapper:

In wrapper methods, we try to use a subset of features and train a model using them. Based on the inferences that we draw from the previous model, we decide to add or remove features from your subset. The problem is essentially reduced to a search problem. These methods are computationally very expensive. Some common examples of wrapper methods are **forward feature selection, backward feature elimination, recursive feature elimination**, etc.



Forward Selection:

Forward selection is an iterative method in which we start with having no feature in the model. In each iteration, we keep adding the feature which best improves our model till an addition of a new variable does not improve the performance of the model.

Backward Elimination:

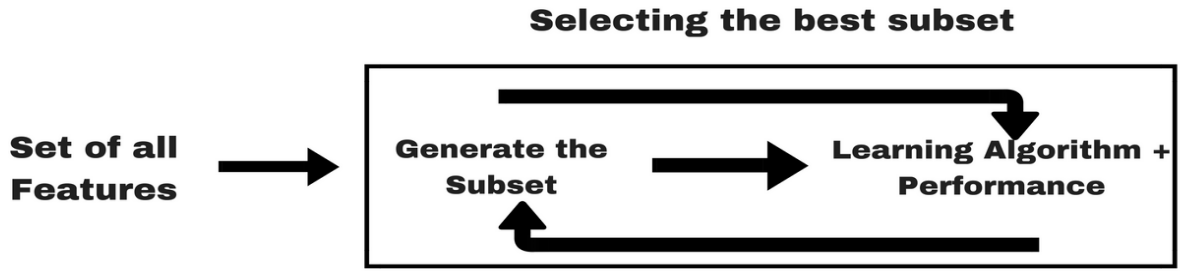
In backward elimination, we start with all the features and removes the least significant feature at each iteration which improves the performance of the model. We repeat this until no improvement is observed on removal of features.

Recursive Feature elimination:

It is a greedy optimization algorithm which aims to find the best performing feature subset. It repeatedly creates models and keeps aside the best or the worst performing feature at each iteration. It constructs the next model with the left features until all the features are exhausted. It then ranks the features based on the order of their elimination.

Embedded:

Embedded methods combine the qualities' of filter and wrapper methods. It's implemented by algorithms that have their own built-in feature selection methods. Some of the most popular examples of these methods are **LASSO** and **RIDGE** regression which have inbuilt penalization functions to reduce overfitting.



Lasso or L1 Regression:

Lasso regression uses the L1 penalty term and stands for **Least Absolute Shrinkage and Selection Operator**. The penalty applied for L2 is equal to the absolute value of the magnitude of the coefficients:

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Again, if *lambda* is zero then we will get back OLS whereas very large value will make coefficients zero hence it will under-fit.

Ridge or L2 Regression:

Ridge regression adds "squared magnitude" of coefficient as penalty term to the loss function. Here the *h*ighlighte*d* part represents L2 regularization element.

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

Here, if *lambda* is zero then you can imagine we get back OLS. However, if *lambda* is very large then it will add too much weight and it will lead to under-fitting. Having said that it's important how *lambda* is chosen. This technique works very well to avoid over-fitting issue.

Differences:

- The **key difference** between these techniques is that Lasso shrinks the less important feature's coefficient to zero thus, removing some feature altogether. So, this works well for **feature selection** in case we have a huge number of features.

L2 loss function	L1 loss function
Not very robust	Robust
Stable solution	Unstable solution
Always one solution	Possibly multiple solutions

Feature Extraction:

Feature extraction is a dimensionality reduction process, where an initial set of raw variables is reduced to more manageable groups (features) for processing, while still accurately and completely describing the original data set.

Principal Component Analysis (PCA):

Principal Component Analysis is a method that uses simple matrix operations from linear algebra and statistics to calculate a projection of the original data into the same number or fewer dimensions.

Principal Component Analysis, or PCA for short, is a method for reducing the dimensionality of data. It can be thought of as a projection method where data with m-columns (features) is projected into a subspace with m or fewer columns, whilst

retaining the essence of the original data. The PCA method can be described and implemented using the tools of linear algebra. PCA is an operation applied to a dataset, represented by an n x m matrix A that results in a projection of A which we will call B.

Syntax:

```
import numpy as np
from sklearn.decomposition import PCA
X = numpy array of inputs
pca = PCA(n_components=2)
pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=2,
random_state=None,svd_solver='auto', tol=0.0, whiten=False)
```

Parameters:

- **n_components :** int, float, None or string - Number of components to keep. if n_components is not set all components are kept:
  - ✓ n_components == min(n_samples, n_features)
  - ✓ If n_components == 'mle' and svd_solver == 'full', Minka's MLE is used to guess the dimension. Use of n_components == 'mle' will interpret svd_solver == 'auto' as svd_solver == 'full'.
  - ✓ If 0 < n_components < 1 and svd_solver == 'full', select the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by n_components.
  - ✓ If svd_solver == 'arpack', the number of components must be strictly less than the minimum of n_features and n_samples.
  - ✓ Hence, the None case results in: n_components == min(n_samples, n_features) - 1
- **copy :** bool (default True) - If False, data passed to fit are overwritten and running fit(X).transform(X) will not yield the expected results, use fit_transform(X) instead.

- **whiten :** bool, optional (default False)
    - ✓ When True (False by default) the components_ vectors are multiplied by the square root of n_samples and then divided by the singular values to ensure uncorrelated outputs with unit component-wise variances.
    - ✓ Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making their data respect some hard-wired assumptions.

- **svd_solver :** string {'auto', 'full', 'arpack', 'randomized'}
    - ✓ **auto :**the solver is selected by a default policy based on X.shape and n_components: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient 'randomized' method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.
    - ✓ **full :** run exact full SVD calling the standard LAPACK solver via scipy.linalg.svd and select the components by postprocessing
    - ✓ **arpack :**run SVD truncated to n_components calling ARPACK solver via scipy.sparse.linalg.svds. It requires strictly 0 < n_components < min(X.shape)
    - ✓ **randomized :**run randomized SVD by the method of Halko et al.

- **iterated_power :** int >= 0, or 'auto', (default 'auto') - Number of iterations for the power method computed by svd_solver == 'randomized'.

Attributes:

- **components_ :** array, shape (n_components, n_features) - Principal axes in feature space, representing the directions of maximum variance in the data. The components are sorted by explained_variance_.

- **explained_variance_ :** array, shape (n_components,) - The amount of variance explained by each of the selected components**.**

- **explained_variance_ratio_ :** array, shape (n_components,) - Percentage of variance explained by each of the selected components.If n_components is not set then all components are stored and the sum of the ratios is equal to 1.0.

- **singular_values_ :** array, shape (n_components,) - The singular values corresponding to each of the selected components. The singular values are equal to the 2-norms of the n_components variables in the lower-dimensional space.

- **mean_ :** array, shape (n_features,) - Per-feature empirical mean, estimated from the training set.

- **n_components_ :** int - The estimated number of components. When n_components is set to 'mle' or a number between 0 and 1 (with svd_solver == 'full') this number is estimated from input data. Otherwise it equals the parameter n_components, or the lesser value of n_features and n_samples if n_components is None.

- **noise_variance_ :** float - The estimated noise covariance following the Probabilistic PCA model from Tipping and Bishop 1999.

Methods:

fit(self, X[, y])	Fit the model with X.
fit_transform(self, X[, y])	Fit the model with X and apply the dimensionality reduction on X.
get_covariance(self)	Compute data covariance with the generative model.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Compute data precision matrix with the generative model.
inverse_transform(self, X)	Transform data back to its original space.
score(self, X[, y])	Return the average log-likelihood of all samples.

score_samples(self, X)	Return the log-likelihood of each sample.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply dimensionality reduction to X.

Example:

```
import numpy as np
from sklearn.decomposition import PCA
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
pca = PCA(n_components=2)
pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=2,
random_state=None,svd_solver='auto', tol=0.0, whiten=False)
print(pca.explained_variance_ratio_)
[0.9924... 0.0075...]
print(pca.singular_values_)
[6.30061... 0.54980...]
```