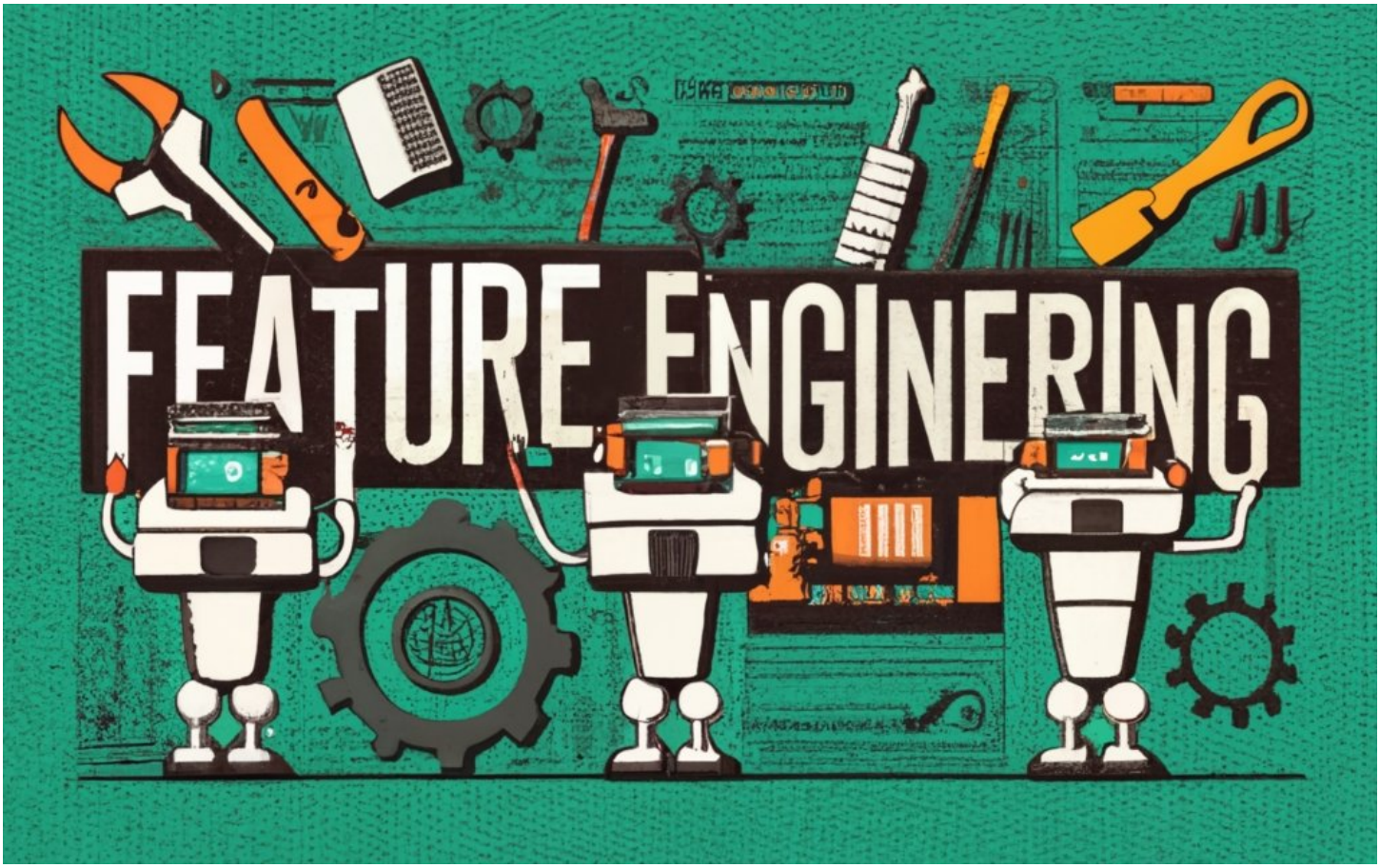


Feature Engineering Part 2: Exploring Advanced Tecnhiques

Cont.Table Of Contents

5.Feature Transformation

```
- Undestanding QQPlot and PP-Plot
- logarithmic transformation
- reciprocal transformation
- square root transformation
- exponential transformation
- boxcox transformation
```

6.Using Column Transformer to speed up FE

7.Using Pipelines to automate the FE

- What are Pipelines
- Accessing individual steps in pipeline
- Accessing Parameters in Pipeline
- Performing Grid Search with Pipeline
- Combining Transformers and Pipeline
- Visualizing the Pipeline

4. Feature Selection Techniques

4.1 Why Feature Selection Matters?

In the real data, not all features are of equal importance. Some might be irrelevant, redundant, or noisy. Feature selection helps in choosing the most relevant features, which can lead to faster training times, improved model performance, reduce models, and better generalization.

However, it's essential to balance feature selection with the risk of losing potentially valuable information. Removing important features can lead to a loss of crucial insights and might result in a less accurate or less robust model. Therefore, it's crucial to carefully evaluate the trade-offs when performing feature selection in machine learning tasks.

Scikit Learn provides `sklearn.feature_selection` API to accomplish this task efficiently.

4.2 Types of Feature Selection

Feature selection can be approached in several ways:

Filter Methods: These methods rely on statistical measures to assign a score to each feature. Common techniques include correlation coefficient scores, chi-squared test, and mutual information. Features are selected based on predefined criteria, such as a certain threshold score.

Wrapper Methods: These methods involve the use of a specific machine learning algorithm to evaluate the performance of a model with different subsets of features. It's an iterative process where different combinations of features are used to train models, and the performance is evaluated to select the best feature subset.

Embedded Methods: These methods incorporate feature selection as part of the model training process. Some machine learning algorithms, like Lasso (L1 regularization) and tree-based algorithms, inherently perform feature selection by assigning zero weights to irrelevant features during model training.

4.3 Filter Methods

Variance Threshold

AS the name suggests, Variance Threshold is a simple technique that removes all features whose variance doesn't meet a certain threshold. This method operates on numerical features and is particularly useful for datasets where low-variance features are considered uninformative or noisy.

We can use `VarianceThreshold` method from `sklearn.feature_selection` module to accomplish this task, and by default it removes a feature which has same value, that is zero variance.

In [1]:
```
import pandas as pd
import numpy as np

from sklearn.datasets import load_breast_cancer

# Load the breast cancer dataset from scikit-learn
data = load_breast_cancer()
X, y = data.data, data.target

# Convert the data to a DataFrame
df = pd.DataFrame(data.data, columns=data.feature_names)

df.head()
```

Out[1]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	dim
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	(
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	(
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	(
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	(
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	(

5 rows × 30 columns

In [2]:
```
from sklearn.feature_selection import VarianceThreshold

# Define the variance threshold value
threshold_value = 0.1

# Initialize the Variance Threshold object
selector = VarianceThreshold(threshold=threshold_value)

# Fit the selector to the data
X_filtered = selector.fit_transform(X)

# Print the shape of the original and filtered data
print("Original Data Shape:", X.shape)
print("Filtered Data Shape:", X_filtered.shape)
```

```
Original Data Shape: (569, 30)
Filtered Data Shape: (569, 11)
```

By applying the Variance Threshold method to the breast cancer dataset, you can observe how it filters out low-variance (less than 0.1) features and brings down the features from 30 to 11, however we have to do further investigation as to observe if we lost any important features.

Univariate Feature Selection

Univariate feature selection is a type of feature selection method that selects the best features based on univariate statistical tests. It works by selecting the features that have the strongest relationship with the target variable. This approach assesses each feature individually and independently to determine its strength in relation to the target variable, without considering the interaction between features.

Univariate feature selection is based on the following key steps:

Scoring Features: It involves using statistical tests like chi-squared test, ANOVA F-test, or mutual information to score the features individually.

Ranking Features: Based on the scores obtained from the statistical tests, the features are ranked in order of their significance or importance.

Selecting Features: The top-k ranked features are then selected for further analysis or model building.

Scikit-learn Implementation: Scikit-learn provides several methods for univariate feature selection, including `SelectKBest` , `SelectPercentile` , and `GenericUnivariateSelect` .

- Each API need a scoring function to score each feature, we can choose from 3 classes of scoring functions such as `Mutual Information(MI)` , `chi-square` , and `F-statistics` .
- MI And F-statistics can be used in both classification and regression problems, by using the methods `mutual_info_regression` , `mutual_info_classif` , `f_regression` , and `f_classif` .
- Chi-square can be used only in classification problems, by using `chi2`
- sklearn provides one more class of univariate feature selection methods that work on common univariate statistical tests for each feature such as `SelectFpr` , `SelectFdr` , and `SelectFwe`
- SelectFpr selects features based on a false positive rate test.
- SelectFdr selects features based on an estimated false discovery rate.
- SelectFwe selects features based on family-wise error rate

SelectKBest

SelectKBest is a univariate feature selection method that selects the top K features based on their scores. It allows you to choose a specific number of features that exhibit the strongest relationship with the target variable.

In [3]:
```
from sklearn.datasets import load_breast_cancer

# Load the breast cancer dataset
data = load_breast_cancer()
X = data.data
y = data.target
```

In [4]:
```
X.shape
```

Out[4]:
```
(569, 30)
```

30 features.

In [5]:
```
data.feature_names
```

Out[5]:
```
array(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal dimension',
       'radius error', 'texture error', 'perimeter error', 'area error',
       'smoothness error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error',
       'fractal dimension error', 'worst radius', 'worst texture',
       'worst perimeter', 'worst area', 'worst smoothness',
       'worst compactness', 'worst concavity', 'worst concave points',
       'worst symmetry', 'worst fractal dimension'], dtype='<U23')
```

In [6]:
```
from sklearn.feature_selection import SelectKBest, chi2

# Initialize SelectKBest with the desired configuration
selector = SelectKBest(score_func=chi2, k=20)

# Fit the selector to the data
X_new = selector.fit_transform(X, y)

# Get the support mask, which indicated the selected features.
support = selector.get_support()

# Display the selected columns
selected_columns = [column for column, is_selected in zip(data.feature_names, sup
port) if is_selected]
print("Selected Columns:")
print(selected_columns)
```

```
Selected Columns:
['mean radius', 'mean texture', 'mean perimeter', 'mean area', 'mean compactnes
s', 'mean concavity', 'mean concave points', 'radius error', 'perimeter error',
'area error', 'compactness error', 'concavity error', 'worst radius', 'worst tex
ture', 'worst perimeter', 'worst area', 'worst compactness', 'worst concavity',
'worst concave points', 'worst symmetry']
```

In [7]:
```
# Another appraoch to get the same, if the above is confusing.
from sklearn.feature_selection import SelectKBest, chi2

### Apply SelectKBest Algorithm
selector = SelectKBest(score_func=chi2,k=20)
selector.fit(X,y)

dfscores=pd.DataFrame(selector.scores_,columns=["Score"])
dfcolumns=pd.DataFrame(data.feature_names)

features_rank=pd.concat([dfcolumns,dfscores],axis=1)

features_rank.columns=['Features','Score']
features_rank.nlargest(20,'Score')['Features'].values
```

Out[7]:
```
array(['worst area', 'mean area', 'area error', 'worst perimeter',
       'mean perimeter', 'worst radius', 'mean radius', 'perimeter error',
       'worst texture', 'mean texture', 'worst concavity', 'radius error',
       'mean concavity', 'worst compactness', 'worst concave points',
       'mean concave points', 'mean compactness', 'worst symmetry',
       'concavity error', 'compactness error'], dtype=object)
```

Now we got the top 20 important features using SelectKBest

SelectPercentile

SelectPercentile is similar to SelectKBest but selects the top features based on a specified percentage of the highest scores. This is helpful when you want to maintain a certain proportion of the most relevant features. we can implement in the same way as SelectKBest, just that you need to use the percentile parameter to specify the percentile threshold, `selector = SelectKBest(score_func=chi2, k=no of features to select)`

GenericUnivariateSelect

This method is a versatile feature selection tool that allows you to perform univariate feature selection with configurable strategies. It provides various options for setting the mode of operation, which enables you to customize the feature selection process based on your specific requirements. We can implement it same way as SelectKBest, by using `GenericUnivariateSelect(score_func=f_classif, mode='percentile', param=no of features to select)` . As for mode, default is percentile, you have various other options as `k_best` , `fpr` , `fdr` , and `fwe` .

Note: Make sure to not use regression feature scoring function with a classification problem, It will lead to useless results.

4.4 Wrapper Based Filter Selection

These methods involve the use of a specific machine learning algorithm to evaluate the performance of a model with different subsets of features.

Recursive Feature Elimination (RFE)

Recursive Feature Elimination (RFE) is a feature selection technique that works by recursively considering smaller and smaller sets of features.

Model Training: RFE begins by training a model on the entire set of features and ranks the features based on their importance derived from the model.

Feature Elimination: It then eliminates the least important feature(s) and retrains the model on the remaining features.

Recursive Process: This process is repeated recursively, with the least important features continuously being eliminated until the desired number of features is reached.

Optimization: RFE aims to find the optimal feature subset that maximizes model performance based on a specified metric.

Scikit-learn Implementation: Scikit-learn provides the `RFE` method that can be used with various estimators to perform recursive feature elimination.

In [8]:
```
from sklearn.datasets import load_breast_cancer

# Load the breast cancer dataset
data = load_breast_cancer()
X, y = data.data, data.target
feature_names = data.feature_names
```

In [9]:
```
X.shape
```

Out[9]:
```
(569, 30)
```

In [10]:
```
from sklearn.feature_selection import RFE
from sklearn.svm import SVC

# Initialize the SVM classifier
estimator = SVC(kernel="linear")

# Initialize the RFE with the estimator and desired number of features
selector = RFE(estimator, n_features_to_select=10, step=1)

# Fit the selector to the data
selector = selector.fit(X, y)

# Display the ranking of the features
print("Feature Ranking: ", selector.ranking_)
```

```
Feature Ranking:  [ 1 12  7 21  4  3  1  1  2 18 16  1  5 13 14 10  9 11 19 17
1  6 15 20
  1  1  1  1  1  8]
```

So this is the ranking for the columns based on the svm classifier. we will be selecting the features corresponding to the top 10 ranks. When step is set to 1, it means that one feature will be eliminated at each iteration until the specified stopping criterion is met.

In [11]:
```
# Get the support mask
support = selector.support_

# Display the selected column names
selected_columns = [feature_names[i] for i in range(len(feature_names)) if suppor
t[i]]
print("Selected Columns:")
print(selected_columns)
```

```
Selected Columns:
['mean radius', 'mean concavity', 'mean concave points', 'texture error', 'worst
radius', 'worst smoothness', 'worst compactness', 'worst concavity', 'worst conc
ave points', 'worst symmetry']
```

In this example:

- We initialize the SVC (Support Vector Classifier) estimator.
- Then initialize the RFE with the SVC estimator and specify the desired number of features to select.
- And fit the selector to the dataset X and the target y.
- Finally, we display the ranking of the features, indicating their importance based on the RFE process.

RFE can be effective when dealing with complex feature spaces where the relationship between features and the target variable is not apparent, as it can systematically eliminate less important features to reveal the most relevant ones.

Recursive Feature Elimination with Cross-Validation (RFECV)

- It can get difficult to choose the correct number of features, So use RFECV if we do not want to specify the desired number of features in RFE.
- RFECV Performs RFE in a cross validation loop to find the optimal number of features.

In [12]:
```
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV
from sklearn.svm import SVC

# Initialize the SVM classifier
estimator = SVC(kernel="linear")

# Initialize the RFECV with the estimator and cross-validation generator
rfecv = RFECV(estimator, step=1, cv=StratifiedKFold(5), scoring='accuracy')

# Fit the RFECV to the data
rfecv.fit(X, y)

# Get the support mask
support = rfecv.support_

# Get the selected column names
selected_columns = [data.feature_names[i] for i in range(len(data.feature_names))
if support[i]]
print(f"Selected {len(selected_columns)} Columns:")
print(selected_columns)
```

```
Selected 14 Columns:
['mean radius', 'mean smoothness', 'mean compactness', 'mean concavity', 'mean c
oncave points', 'mean symmetry', 'texture error', 'perimeter error', 'worst radi
us', 'worst smoothness', 'worst compactness', 'worst concavity', 'worst concave
points', 'worst symmetry']
```

Until now we have been selecitng 20 columns, but through RFECV we got to know 14 columns would be a wise choice. We can do further invesigation to confirm on this.

Select From Model

SelectFromModel Selects desired number of important features above certain threshold of feature importances as obtained from the trained estimator.

Model Training: First, you train a model on the entire set of features.

Feature Importance: After training the model, you can extract feature importances, coefficients, or weights from the model, depending on the type of model used.

Thresholding: The threshold can be specified either numerically or through string argument based on built in heuristics such as `mean` , `median` .

Selecting Features: Features with importance scores above the threshold are selected, while the rest are discarded.

Scikit-learn Implementation: Scikit-learn provides the SelectFromModel class that can be used with various models, including linear models, tree-based models, and others.

In [13]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectFromModel
from sklearn.svm import LinearSVC

# Load the breast cancer dataset
data = load_breast_cancer()
X, y = data.data, data.target

# Initialize the Linear SVM classifier
estimator = LinearSVC(C=0.01, penalty="l1", dual=False)

# Initialize SelectFromModel with the estimator and the threshold
selector = SelectFromModel(estimator, threshold='mean')

# Fit the selector to the data
selector = selector.fit(X, y)

# Get the support mask
support = selector.get_support()

# Display the selected column names
selected_columns = [data.feature_names[i] for i in range(len(data.feature_names))
if support[i]]
print("Selected Columns:")
print(selected_columns)
```

```
Selected Columns:
['mean perimeter', 'area error', 'worst texture', 'worst area']

C:\Users\raviteja\anaconda3\lib\site-packages\sklearn\svm\_base.py:1208: Converg
enceWarning: Liblinear failed to converge, increase the number of iterations.
  ConvergenceWarning,
```

you may need to adjust the threshold based on the feature importances provided by the specific model you are using.

Sequential Feature Selection

- Performs feature selection by selecting or deselecting features one by one in a greedy manner. It uses two approaches.
- **Forward Selection:** This approach starts with an empty feature set and at each iteration adds the feature that maximizes some criterion. It stops when the desired number of features is reached.
- **Backward Selection:** This approach starts with the full feature set and at each iteration removes the feature that contributes the least to the criterion. It stops when the desired number of features is reached.
- Use sfs by importing `SequentialFeatureSelector` , you can use the `direction` parameter to control whether forward or backward SFS to be used. And they don't give the equivalent results.
- SFS may be slower than RFE and SelectFromModel as it needs to evaluate more models compared to the other two approaches.

Pricipal Component Analysis(PCA)

Principal Component Analysis (PCA) is a popular dimensionality reduction technique used to transform high-dimensional datasets into a lower-dimensional subspace. It achieves this by identifying the directions (principal components) that capture the most variance in the data, thus allowing the data to be represented with fewer dimensions while preserving the most important information.

In [14]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Load the breast cancer dataset
data = load_breast_cancer()
X, y = data.data, data.target

# Standardize the feature matrix
X = StandardScaler().fit_transform(X)

# Initialize the PCA with no of features you want
pca = PCA(n_components=10)

# Apply PCA to the standardized feature matrix
principalComponents = pca.fit_transform(X)

# Visualize the variance explained by each principal component
explained_variance = pca.explained_variance_ratio_

# Create a DataFrame for the reduced feature matrix
principalDf = pd.DataFrame(data=principalComponents)

# Concatenate with the target variable to visualize the reduced data
finalDf = pd.concat([principalDf, pd.Series(data.target, name='target')], axis=1)

finalDf
```

Out[14]:

	0	1	2	3	4	5	6	7	8
0	9.192837	1.948583	-1.123166	3.633731	-1.195110	1.411424	2.159370	-0.398409	-0.157112
1	2.387802	-3.768172	-0.529293	1.118264	0.621775	0.028656	0.013359	0.240986	-0.711909
2	5.733896	-1.075174	-0.551748	0.912083	-0.177086	0.541452	-0.668167	0.097376	0.024068
3	7.122953	10.275589	-3.232790	0.152547	-2.960878	3.053422	1.429910	1.059570	-1.405439
4	3.935302	-1.948072	1.389767	2.940639	0.546747	-1.226495	-0.936213	0.636376	-0.263806
...	...	...	...	...	...	...	...	...	...
564	6.439315	-3.576817	2.459487	1.177314	-0.074824	-2.375193	-0.596130	-0.035467	0.987924
565	3.793382	-3.584048	2.088476	-2.506028	-0.510723	-0.246710	-0.716327	-1.113356	-0.105210
566	1.256179	-1.902297	0.562731	-2.089227	1.809991	-0.534447	-0.192759	0.341889	0.393915
567	10.374794	1.672010	-1.877029	-2.356031	-0.033742	0.567936	0.223082	-0.280241	-0.542032
568	-5.475243	-0.670637	1.490443	-2.299157	-0.184703	1.617837	1.698952	1.046349	0.374105

569 rows × 11 columns

5. Feature Transformation

When we plan to transform data, we assume a distribution and then we fit the distribution to our data, then we have statistical tests like chi-square and f-tests to get the goodness of fit, however there are also probability plots such as **QQ-Plot** and **PP-Plot** to observe how well the data is fit to the distribution.

QQ-Plot

- Graph of the qi-quantile of a fitted distribution versus the qi-quantile of the sample distribution.
- If both quantile matches, then we will get a nice 45 degrees line, how far the points away the points from the line will tell us how good of a fit the plot is.
- If the points are close to the line, then we can say that they are of a good fit, if the points are too far away then it means that the distribution doesn't match!!

PP-Plot

- Probability Probability Plot, A graph of model probability againist the sample probability.
- The interpretation is pretty much same as the qqplot.

Note:

1. qqplot checks if the distribution is fitting well on the tail regions.
2. ppplot checks if the distribution is fitting well on the center regions.

In [15]:
```
#Importing the required statistical packages to plot qqplot and ppplot
import scipy.stats as stats
import statsmodels.api as sm
```

In [16]:
```
df=pd.read_csv('titanic.csv',usecols=['Age','Fare','Survived'])
df.head()
```

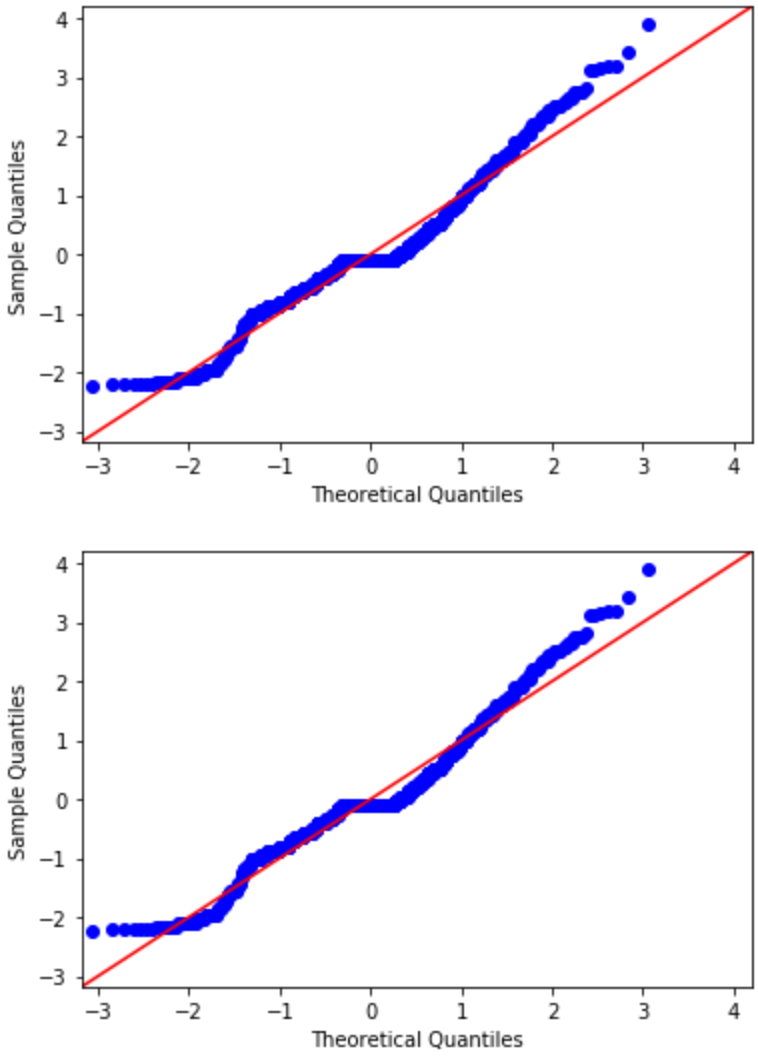
Out[16]:

	Survived	Age	Fare
0	0	22.0	7.2500
1	1	38.0	71.2833
2	1	26.0	7.9250
3	1	35.0	53.1000
4	0	35.0	8.0500

In [17]:
```
### fillnan
df['Age']=df['Age'].fillna(df['Age'].median())
```

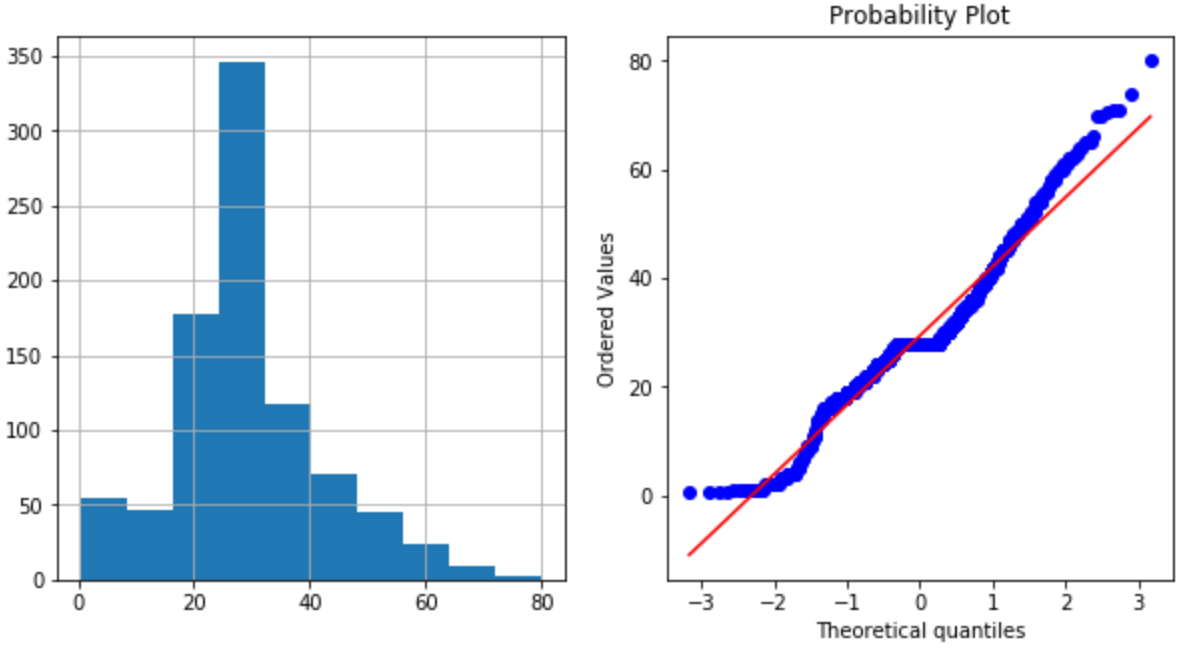
In [18]:
```
sm.qqplot(df['Age'], stats.norm, fit=True, line='45')
```

Out[18]:



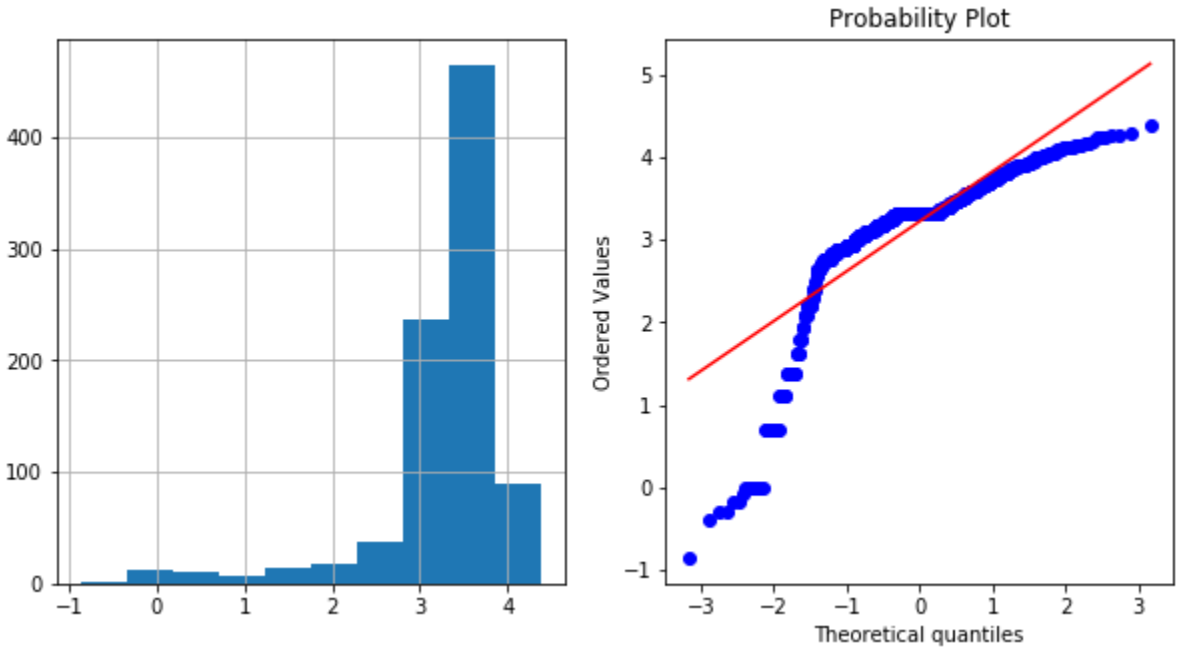
In [19]:
```
#### Histplot, QQplot and pp-plot
import matplotlib.pyplot as plt
import pylab
def plot_data(df,feature):
    plt.figure(figsize=(10,5))
    plt.subplot(1,2,1)
    df[feature].hist()
    plt.subplot(1,2,2)
    stats.probplot(df[feature],dist='norm',plot=pylab)
    plt.show()
```

In [20]:
```
plot_data(df,'Age')
```



Logarithmic Transformation

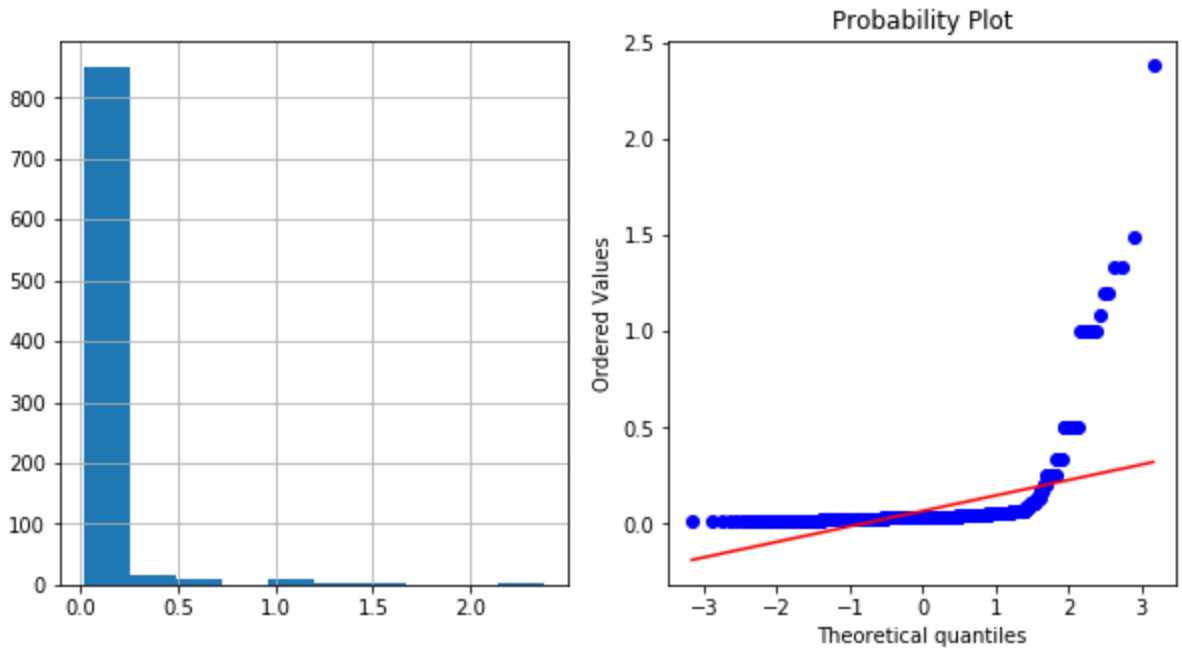
In [21]:
```
import numpy as np
df['Age_log']=np.log(df['Age'])
plot_data(df,'Age_log')
```



We can see that the a lot of points are not close to the line, which signifies this is not of log distribution.

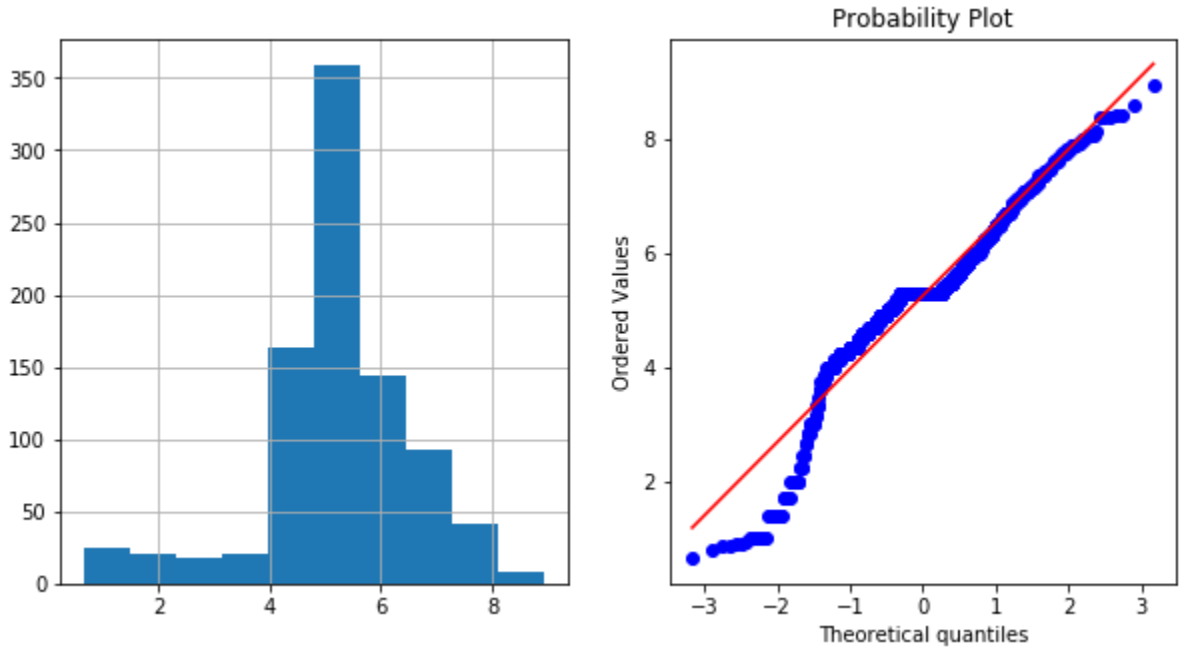
Reciprocal Trnasformation

In [22]:
```
df['Age_reciprocal']=1/df.Age
plot_data(df,'Age_reciprocal')
```



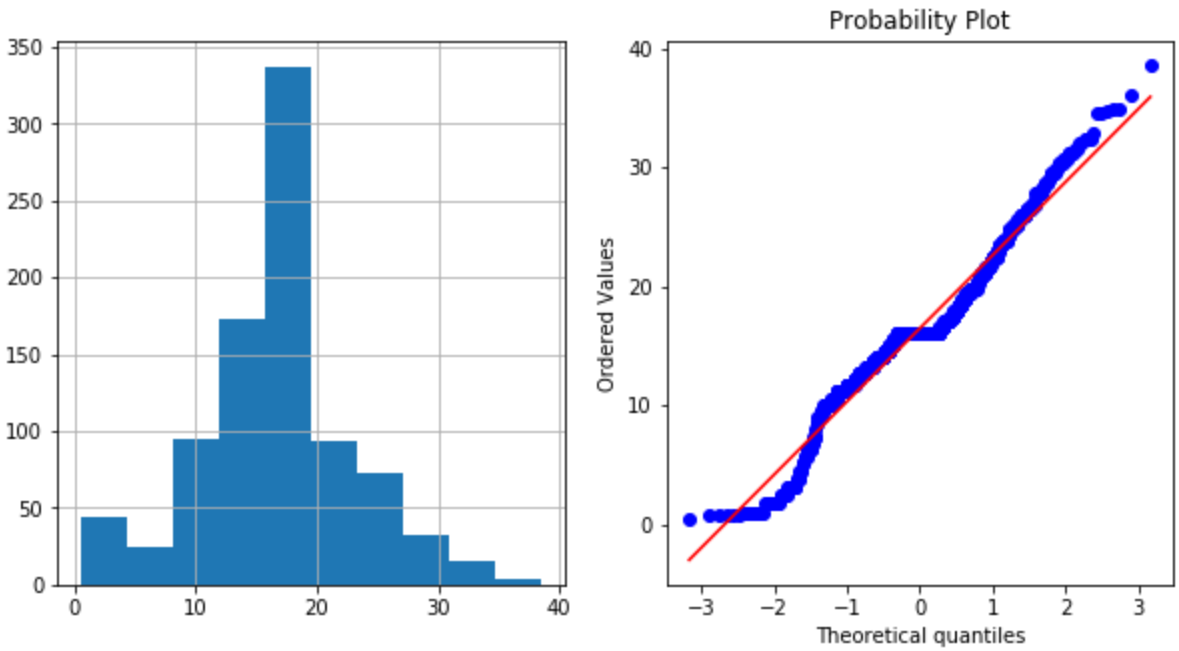
Square Root Transformation

In [23]:
```
df['Age_sqaure']=df.Age**(1/2)
plot_data(df,'Age_sqaure')
```



Exponential Transdormation

In [24]:
```
df['Age_exponential']=df.Age**(1/1.2)
plot_data(df,'Age_exponential')
```



We can observe that the points are very close to the line comparing to any other distribution. so we can say, this can be of exponential distribution.

BoxCOx Transformation

The Box-Cox transformation is defined as:

$T(Y)=(Y\ \exp(\lambda)-1)/\lambda$

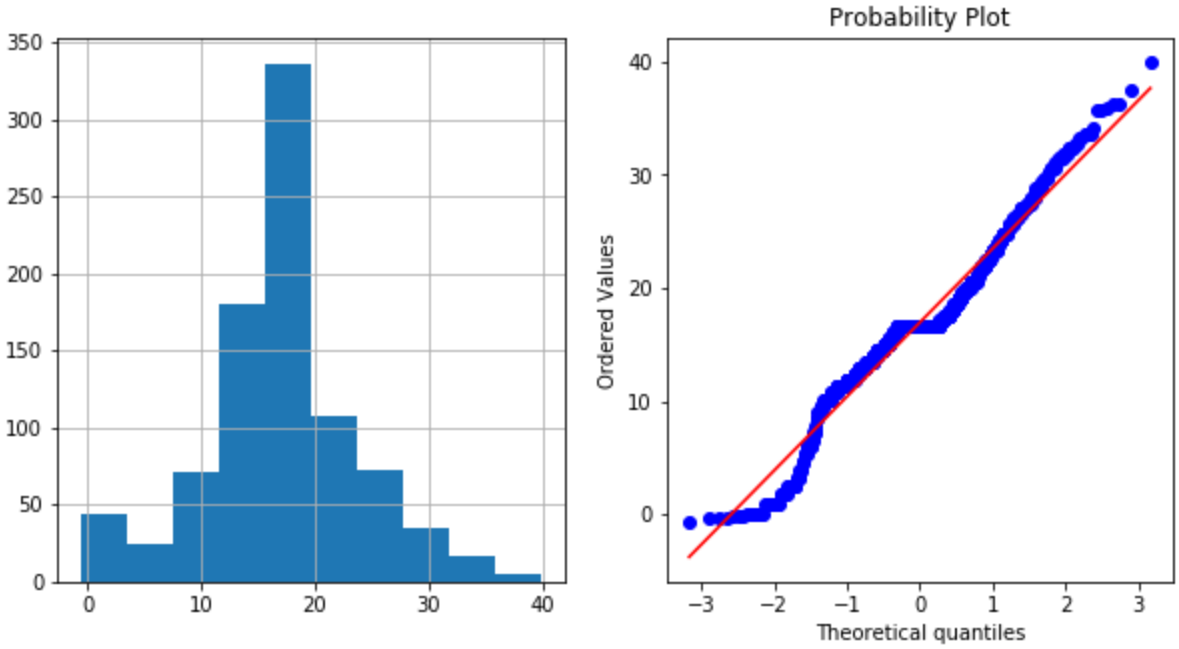
where Y is the response variable and λ is the transformation parameter. λ varies from -5 to 5. In the transformation, all values of λ are considered and the optimal value for a given variable is selected.

In [25]:
```
df['Age_Boxcox'],parameters=stats.boxcox(df['Age'])
```

In [26]:
```
print(parameters)
```

```
0.7964531473656952
```

```
In [27]: plot_data(df,'Age_Boxcox')
```



6. Using Composite Transformers to speed up FE

- Transformers are algorithms or tools that are used to preprocess, modify, or create new representations of data. The ones we have seen so far are called transformers, like the StandardScaler, OneHotEncoding etc. Transformers can perform a variety of operations, including scaling, encoding, imputing missing values, and generating new features.
- We know that generally training data contains diverse features such as numeric and categorical. And different feature types are processed with different transformers.
- And **to combine those individual transformers**, we use Composite Transformers.

Column Transformer

- Imagine you have a table with different types of information like numbers, categories, and so on. A ColumnTransformer is like a smart worker who can handle each type of information in the table differently.
- For instance, let's say you have columns for people's ages, their jobs, and their education levels. You might want to do different things to each of these types of data. Maybe you want to scale the ages so they all have a similar impact. You might also want to turn the different jobs and education levels into numbers so the computer can understand them better.
- A ColumnTransformer is like a helpful friend who can take care of each of these tasks automatically. You tell the ColumnTransformer what to do with each column, and it does the job for you. So you don't have to worry about doing each task separately.
- To implement this we can use `ColumnTransformer` API from sklearn.compose module

In [28]:
```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# Create a sample dataset
data = {'age': [30, 40, 35, 25, 32],
        'gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
        'income': [50000, 75000, 100000, 40000, 60000]}

df = pd.DataFrame(data)

df
```

Out[28]:

	age	gender	income
0	30	Male	50000
1	40	Female	75000
2	35	Male	100000
3	25	Female	40000
4	32	Male	60000

In [29]:
```
# Define the transformations for different columns
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), ['age','income']),
        ('cat', OneHotEncoder(), ['gender'])
    ])

# Apply the transformations to the dataset
transformed_df = preprocessor.fit_transform(df)
transformed_df
```

Out[29]:
```
array([[-0.47961646, -0.71509694,  0.        ,  1.        ],
       [ 1.51878546,  0.47673129,  1.        ,  0.        ],
       [ 0.5195845 ,  1.66855953,  0.        ,  1.        ],
       [-1.47881742, -1.19182824,  1.        ,  0.        ],
       [-0.07993608, -0.23836565,  0.        ,  1.        ]])
```

- So if you observe here, age and income are scaled with standard scaler and gender is scaled with onehot encoder all with one single transformer that we created.
- So, for the testing data, we don't have to apply each and every transformer seperately, we can just apply this one alone.

7. Using Pipelines to Automate FE

The `sklearn.pipeline` module provides utilities to build a composite estimator, as a chain of transformers and estimators.

- **Pipeline:** Constructs a chain of multiple transformers to execute a fixed sequence of steps in data preprocessing and modelling.
- **FeatureUnion:** Combines output from several transformer objects by creating a new transformer from them.

Pipelines

- The purpose of the pipeline is to automate the process of sequentially applying a list of transformers and a final estimator, allowing for a more efficient and organized workflow in machine learning tasks.
- So usually it is used like intermediate steps as transformers, and the final step as estimator.
- Pipeline() method will take a list of (estimator_name,estimator()) tuples.

In [30]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load the Iris dataset
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_s
tate=42)

# Data Preprocessing
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

# Model Training
clf = SVC()
clf.fit(X_train_pca, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test_pca)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 0.9333333333333333
```

So, IF you observe above, we had to do few data processing steps as

- Create standard scaler and fit and transform it with training data, and apply it for test data.
- Create PCA to reduce Dimensions and fit and transform it with training data, and apply it for test data.
- Create Model and fit and predict with the model.

Now, all these steps can be combined and used seamlessly with pipelines as follows:

In [31]:
```
from sklearn.pipeline import Pipeline

# Define the pipeline
pipe = Pipeline([
    ('scaler', StandardScaler()),  # Data preprocessing - Feature scaling
    ('pca', PCA(n_components=2)),  # Data preprocessing - Dimensionality reductio
n
    ('classifier', SVC())  # Model training - Support Vector Classifier
])

# Fit the pipeline on the training data
pipe.fit(X_train, y_train)

# Make predictions on the test data
y_pred = pipe.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 0.9333333333333333
```

- Here, Just by giving Pipe.fit it applies all the things as scaling, dimensionality reduction and model fitting.
- Then during pipe.predict, it wil apply these to test data, very simple, right?
- Similarly, you can add many more transformer as you want.

Access Individual steps in pipeline

- YOu can also access the individual steps easily in various ways, say if i want to access pca estimator from above pipeline.

1. pipe.named_steps.pca
2. pipe.steps[1]
3. pipe[1]
4. pipe['pca']

Accessing parameters of each step in Pipeline

Parameters of the estimators in the pipeline can be accessed using the `estimator__parametername` . You can set all the parameter values at once using `.set_params()`

In [32]:
```
# Define the pipeline
pipe = Pipeline([
    ('scaler', StandardScaler()),  # Data preprocessing - Feature scaling
    ('pca', PCA()),  # Data preprocessing - Dimensionality reduction
    ('classifier', SVC())  # Model training - Support Vector Classifier
])

pipe.set_params(pca__n_components = 2)

# Fit the pipeline on the training data
pipe.fit(X_train, y_train)

# Make predictions on the test data
y_pred = pipe.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 0.9333333333333333
```

Performing Grid Search with Pipeline

By using naming convention of nested parameters, grid search can implemented

In [33]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.svm import SVC
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score

# Load the Iris dataset
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_s
tate=42)

# Create a pipeline
pipe = Pipeline([
    ('scaler', StandardScaler()),  # Data preprocessing - Feature scaling
    ('pca', PCA()),  # Data preprocessing - Dimensionality reduction
    ('classifier', SVC())  # Model training - Support Vector Classifier
])

# Define the hyperparameter grid
param_grid = {
    'pca__n_components': [2, 3, 4],
    'classifier__C': [0.1, 1, 10, 100],
    'classifier__gamma': [0.1, 0.01, 0.001],
}

# Use GridSearchCV to find the best hyperparameters
grid_search = GridSearchCV(pipe, param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Make predictions on the test data
y_pred = grid_search.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Best parameters: {grid_search.best_params_}')
print(f'Accuracy: {accuracy}')
```

```
Best parameters: {'classifier__C': 1, 'classifier__gamma': 0.1, 'pca__n_componen
ts': 3}
Accuracy: 1.0
```

So, the given parameters each combination will be checked, and the best accuracy one will be fit to the model.

you can get the best parameters by using `.best_params_` method for your created grid object.

Combining Transformers and Pipelines

- sklearn.pipeline.FeatureUnion concatenates results of multiple transformer objects.
- Applies a list of transformer objects in parallel, and their outputs are concatenated side-by-side into a larger matrix.

In [34]:
```
import pandas as pd
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load the data (using a sample dataset for demonstration)
X = pd.read_csv('titanic.csv',usecols=['Fare','Age','Sex','Embarked'])
y = pd.read_csv('titanic.csv',usecols=['Survived'])

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_s
tate=42)

# Numeric Pipeline
numeric_pipeline = ColumnTransformer([
    ('imputer', SimpleImputer(strategy='mean'),['Age']),
    ('scaler', StandardScaler(),['Age','Fare'])
])

# Column Transformer for Categorical Features
categorical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Feature Union
full_pipeline = FeatureUnion([
    ('numeric_pipeline', numeric_pipeline),
    ('categorical_transformer', categorical_transformer)
])

# Fit the pipeline on the training data
full_pipeline.fit(X_train, y_train)
```

Out[34]:
```
FeatureUnion(transformer_list=[('numeric_pipeline',
                                ColumnTransformer(transformers=[('imputer',
                                                                 SimpleImputer
(),
                                                                 ['Age']),
                                                                ('scaler',
                                                                 StandardScaler
(),
                                                                 ['Age',
                                                                  'Fare'])])),
                               ('categorical_transformer',
                                Pipeline(steps=[('imputer',
                                                 SimpleImputer(strategy='most_fr
equent')),
                                                ('onehot',
                                                 OneHotEncoder(handle_unknown='i
gnore'))]))])
```

1. The numeric_pipeline has two steps:
   - SimpleImputer: This step imputes missing values in the numeric features using the mean value.
   - StandardScaler: This step scales the numeric features to have a mean of 0 and a standard deviation of 1.

1. The categorical_transformer has two steps:

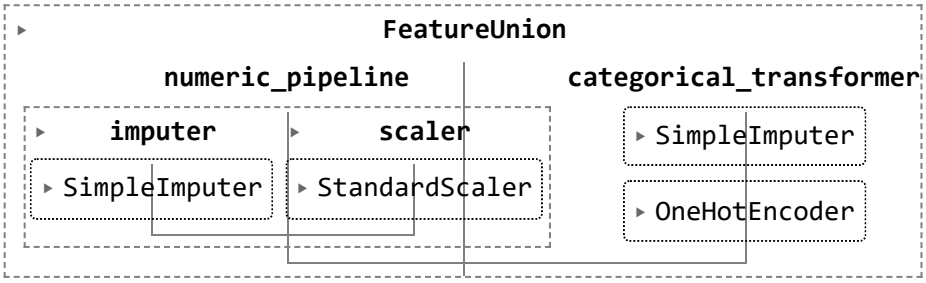
   - SimpleImputer: This step imputes missing values in the categorical features using the most frequent value.
   - OneHotEncoder: This step one-hot encodes the categorical features, which means that it creates a new binary feature for each unique category in each categorical feature.

1. The FeatureUnion pipeline then combines the output of the numeric_pipeline and the categorical_transformer into a single feature matrix. This feature matrix can then be used to train a machine learning model.

Visualizing the Pipeline

In [35]:
```
from sklearn import set_config
set_config(display='diagram')
full_pipeline
```

Out[35]:



This is my Entire Feature Engineering Learnings. Hope this Helps! Happy Learning :)