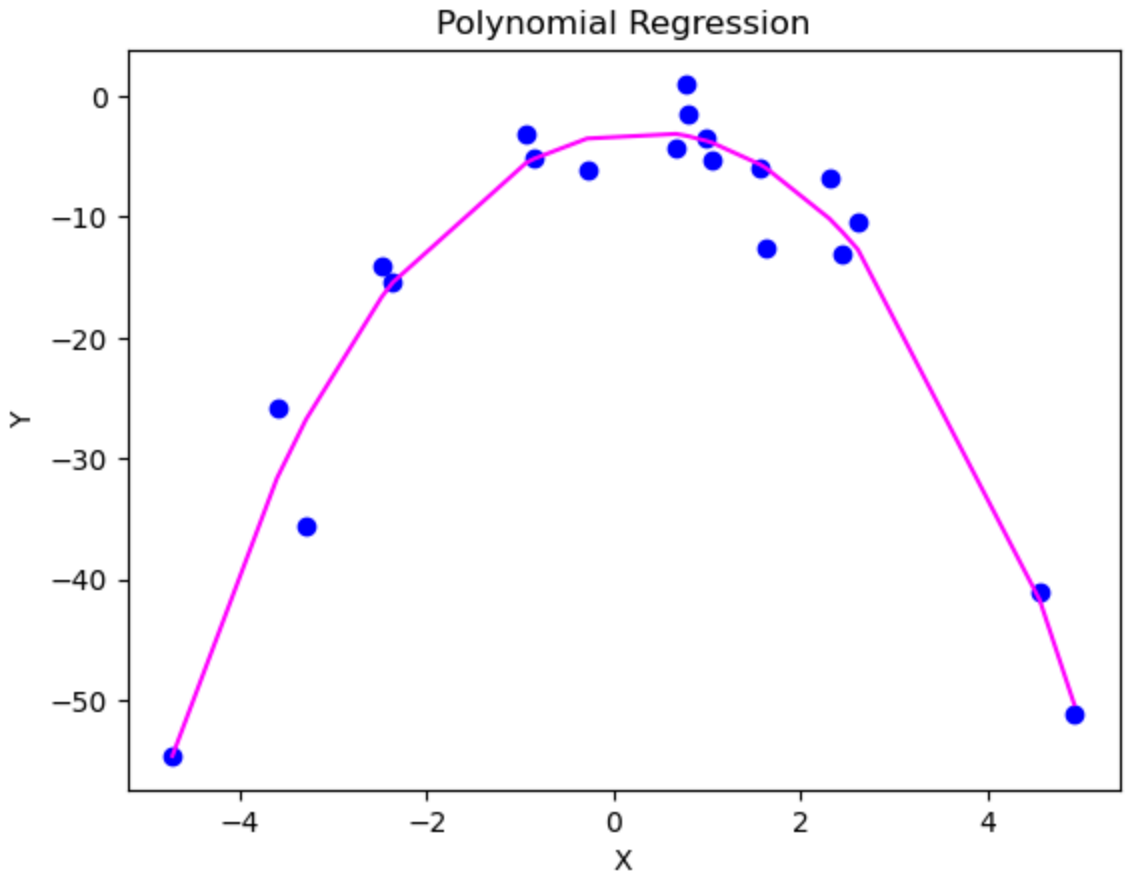
1:- Polynomial Regression

Polynomial Regression is a form of linear regression in which the relationship between the independent variable x and dependent variable y is modeled as an nth degree polynomial. Polynomial regression fits a nonlinear relationship between the value of x and the corresponding conditional mean of y, denoted E(y |x), and has been used to describe nonlinear phenomena such as the growth rate of tissues, the distribution of carbon isotopes in lake sediments, and the progression of disease epidemics.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
import operator

# Step 1: Create Dummy Data
np.random.seed(0)
x = 2 - 3 * np.random.normal(0, 1, 20)
y = x - 2 * (x ** 2) + np.random.normal(-3, 3, 20)

# Reshaping for the model
x = x[:, np.newaxis]
y = y[:, np.newaxis]

# Step 2: Polynomial Transformation
degree = 4  # Degree of the polynomial
poly_features = PolynomialFeatures(degree=degree)
x_poly = poly_features.fit_transform(x)

# Step 3: Train Linear Regression Model
model = LinearRegression()
model.fit(x_poly, y)
y_poly_pred = model.predict(x_poly)

# Step 4: Plotting the results
plt.scatter(x, y, color='blue')
sorted_axis = operator.itemgetter(0)
sorted_zip = sorted(zip(x, y_poly_pred), key=sorted_axis)
x, y_poly_pred = zip(*sorted_zip)
```

```
plt.plot(x, y_poly_pred, color='magenta')
plt.title('Polynomial Regression')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



2:- Ridge Regression

Ridge Regression is a regularized version of Linear Regression: a regularization term equal to is added to the cost function. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to evaluate the model's performance using the unregularized performance measure.

Regularization is a technique used in machine learning and statistics to prevent overfitting of models on training data. Overfitting occurs when a model learns the training data too well, including its noise and outliers, leading to poor generalization to new, unseen data. Regularization helps to solve this problem by adding a penalty to the model's complexity.

Ridge regression, also known as Tikhonov regularization, is a type of linear regression that includes a regularization term. The key idea behind ridge regression is to find a new line that doesn't fit the training data as well as ordinary least squares regression, in order to achieve better generalization to new data. This is particularly useful when dealing with multicollinearity (independent variables are highly correlated) or when the number of predictors (features) exceeds the number of observations.

Key Concept:

Regularization: Ridge regression adds a penalty equal to the square of the magnitude of coefficients. This penalty term (squared L2 norm) shrinks the coefficients towards zero, but it doesn't make them exactly zero.

Mathematical Representation:

The ridge regression modifies the least squares objective function by adding a penalty term:

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

where:

y is the response value for the ith observation.

X is the value of the jth predictor for the ith observation.

Beta is the regression coefficient for the jth predictor.

lemda is the tuning parameter that controls the strength of the penalty; lemda>=0

In this code, alpha is the regularization strength (\lambda). Adjusting alpha changes the strength of the regularization penalty. A larger alpha enforces stronger regularization (leading to smaller coefficients), and a smaller alpha tends towards a model similar to linear regression.

Key Points:

Choosing Alpha: Selecting the right value of alpha is crucial. It can be done using cross-validation techniques like RidgeCV.

Standardization: It's often recommended to standardize the predictors before applying ridge regression.

Bias-Variance Tradeoff: Ridge regression balances the bias-variance tradeoff in model training.

In [2]:
```
from sklearn.linear_model import Ridge
import numpy as np

# Example data
X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
# Target values
y = np.dot(X, np.array([1, 2])) + 3

# Ridge Regression Model
ridge_reg = Ridge(alpha=1.0)  # alpha is the equivalent of lambda in the formula
ridge_reg.fit(X, y)

# Coefficients
print("Coefficients:", ridge_reg.coef_)
# Intercept
print("Intercept:", ridge_reg.intercept_)
```

```
Coefficients: [0.8 1.4]
Intercept: 4.5
```

Comparing Simple Linear Regression vs. Ridge Regression

Import Libraries and load the data

In [3]:
```
import seaborn as sns
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error, mean_absolute_percentage_error
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
import numpy as np

# Load Titanic dataset
df = sns.load_dataset('titanic')
```

Pre Process the data

In [4]:
```
# Selecting a subset of columns for simplicity
columns_to_use = ['survived', 'pclass', 'sex', 'age', 'fare']
df = df[columns_to_use]

# Handling missing values
df['age'].fillna(df['age'].median(), inplace=True)

# Define feature and target variable
X = df.drop('survived', axis=1)
y = df['survived']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

```
C:\Users\ustb\AppData\Local\Temp\ipykernel_3436\2343086075.py:6: FutureWarning: A value is trying to be set on a copy
of a DataFrame or Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which
we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or d
f[col] = df[col].method(value) instead, to perform the operation inplace on the original object.


  df['age'].fillna(df['age'].median(), inplace=True)
```

Creating a pipeline

In [5]:
```
# Define a pipeline for OneHotEncoding and model
categorical_features = ['sex']
numerical_features = ['pclass', 'age', 'fare']

# Preprocessor
preprocessor = ColumnTransformer(
    transformers=[
        ('num', 'passthrough', numerical_features),
        ('cat', OneHotEncoder(), categorical_features)])

# Linear Regression Pipeline
lr_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('regressor', LinearRegression())])

# Ridge Regression Pipeline
ridge_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                                 ('regressor', Ridge(alpha=1.0))])
```

train and Evaluate the Models

In [6]:
```
# Train and evaluate Linear Regression
lr_pipeline.fit(X_train, y_train)
lr_pred = lr_pipeline.predict(X_test)
lr_mse = mean_squared_error(y_test, lr_pred)
lr_r2 = r2_score(y_test, lr_pred)
```

```
lr_mae = mean_absolute_error(y_test, lr_pred)
lr_mape = mean_absolute_percentage_error(y_test, lr_pred)
lr_rmse = np.sqrt(lr_mse)

# Train and evaluate Ridge Regression
ridge_pipeline.fit(X_train, y_train)
ridge_pred = ridge_pipeline.predict(X_test)
ridge_mse = mean_squared_error(y_test, ridge_pred)
ridhe_r2 = r2_score(y_test, ridge_pred)
ridge_mae = mean_absolute_error(y_test, ridge_pred)
ridge_mape = mean_absolute_percentage_error(y_test, ridge_pred)
ridge_rmse = np.sqrt(ridge_mse)

print("Linear Regression MSE:", lr_mse)
print("Ridge Regression MSE:", ridge_mse)
print("::::::::::::::::::::::::::::::")
print("Linear Regression R2:", lr_r2)
print("Ridge Regression R2:", ridhe_r2)
print("::::::::::::::::::::::::::::::")
print("Linear Regression MAE:", lr_mae)
print("Ridge Regression MAE:", ridge_mae)
print("::::::::::::::::::::::::::::::")
print("Linear Regression MAPE:", lr_mape)
print("Ridge Regression MAPE:", ridge_mape)
print("::::::::::::::::::::::::::::::")
print("Linear Regression RMSE:", lr_rmse)
print("Ridge Regression RMSE:", ridge_rmse)
```

```
Linear Regression MSE: 0.13684268526287455
Ridge Regression MSE: 0.13686022744784476
::::::::::::::::::::::::::::::
Linear Regression R2: 0.4223219395905451
Ridge Regression R2: 0.42224788568426963
::::::::::::::::::::::::::::::
Linear Regression MAE: 0.2888229558416338
Ridge Regression MAE: 0.28923126730713655
::::::::::::::::::::::::::::::
Linear Regression MAPE: 697272156502681.8
Ridge Regression MAPE: 698032476179649.6
::::::::::::::::::::::::::::::
Linear Regression RMSE: 0.3699225395442599
Ridge Regression RMSE: 0.3699462494036732
```

Lasso Regression

Lasso Regression, which stands for Least Absolute Shrinkage and Selection Operator, is a type of linear regression that uses shrinkage. Shrinkage here means that the data values are shrunk towards a central point, like the mean. The lasso technique encourages simple, sparse models (i.e., models with fewer parameters). This particular type of regression is well-suited for models showing high levels of multicollinearity or when you want to automate certain parts of model selection, like variable selection/parameter elimination.

Key Features of Lasso Regression:

1: "Regularization Term:" The key characteristic of Lasso Regression is that it adds an L1 penalty to the regression model, which is the absolute value of the magnitude of the coefficients. The cost function for Lasso regression is:

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

where lemda is the regularization parameter

2: "Feature Selection:" One of the advantages of lasso regression over ridge regression is that it can result in sparse models with few coefficients; some coefficients can become exactly zero and be eliminated from the model. This property is called automatic feature selection and is a form of embedded method.

3: "Parameter Tuning:" The strength of the L1 penalty is determined by a parameter, typically denoted as alpha or lambda. Selecting a good value for this parameter is crucial and is typically done using cross-validation.

4: "Bias-Variance Tradeoff:" Similar to ridge regression, lasso also manages the bias-variance tradeoff in model training. Increasing the regularization strength increases bias but decreases variance, potentially leading to better generalization on unseen data.

5: "Scaling:" Before applying lasso, it is recommended to scale/normalize the data as lasso is sensitive to the scale of input features.

Implementation in Scikit-Learn:

Lasso regression can be implemented using the Lasso class from Scikit-Learn's linear_model module. Here's a basic example:

In [7]:
```
from sklearn.linear_model import Lasso, Ridge
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Generate some regression data
X, y = make_regression(n_samples=1000, n_features=15, noise=0.1, random_state=42)

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

# Create a Lasso regression object
lasso = Lasso(alpha=1.0)
ridge = Ridge(alpha=1.0)

# Fit the model
lasso.fit(X_train, y_train)
ridge.fit(X_train, y_train)

# Make predictions
y_pred_lasso = lasso.predict(X_test)
y_pred_ridge = ridge.predict(X_test)
# Evaluate the model
print("MSE of Lasso:", mean_squared_error(y_test, y_pred_lasso))
print("MSE of Ridge:", mean_squared_error(y_test, y_pred_ridge))
```

```
MSE of Lasso: 9.387744740461265
MSE of Ridge: 0.05090866185224575
```

In this example, alpha is the parameter that controls the amount of L1 regularization applied to the model. Fine-tuning alpha through techniques like cross-validation is a common practice to find the best model.

In [8]:
```
# Fine tune alpha value using cv
from sklearn.model_selection import GridSearchCV
import numpy as np

# Create a Lasso regression object
lasso = Lasso()

# Create a dictionary for the grid search key and values
param_grid = {'alpha': np.arange(1, 10, 0.1)}

# Use grid search to find the best value for alpha
lasso_cv = GridSearchCV(lasso, param_grid, cv=10)

# Fit the model
lasso_cv.fit(X, y)

# Print the tuned parameters and score
print("Tuned Lasso Regression Parameters: {}".format(lasso_cv.best_params_))
print(".....................................")
print("Best score is {}".format(lasso_cv.best_score_))

print(".....................................")
# Create a Ridge regression object
ridge = Ridge()

# Create a dictionary for the grid search key and values
param_grid = {'alpha': np.arange(1, 10, 0.1)}

# Use grid search to find the best value for alpha
ridge_cv = GridSearchCV(ridge, param_grid, cv=10)

# Fit the model
ridge_cv.fit(X, y)

# Print the tuned parameters and score
print("Tuned Ridge Regression Parameters: {}".format(ridge_cv.best_params_))
print(".....................................")
print("Best score is {}".format(ridge_cv.best_score_))
```

```
Tuned Lasso Regression Parameters: {'alpha': 1.0}
.....................................
Best score is 0.9995685234915115
.....................................
Tuned Ridge Regression Parameters: {'alpha': 1.0}
.....................................
Best score is 0.9999981195099323
```

In [ ]: