Linear Regression Tutorial By Neuraldemy - Amritesh Kumar

This tutorial is part of Neuraldemy tutorial on Linear Regression and is only permitted to be used by students who have the ownership. Distribution, selling, etc. are not permitted.

Visit https://neuraldemy.com/ for more.

First of all, I will give you an overview and then explain the important concepts. Here is how you can implemenet Linear Regression in practice.

R Vs R²

R, also known as the correlation coefficient, measures the strength and direction of a linear relationship between two variables. Useful for understanding the direction and strength of the linear relationship between two variables.Suitable for bivariate analysis (two variables). R², also called the coefficient of determination, measures the proportion of the variance in the dependent variable that is predictable from the independent variable(s). It provides insight into how well the regression model explains the variability in the dependent variable. R is related to R² in that the sign of R (positive or negative) will be reflected in the sign of the regression coefficient in a simple linear regression model. R² is the square of the correlation coefficient (R). This means that if you know R, you can square it to get R².

Simple Linear Regression

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
```

```
In [4]: # load the data and check its shape
        df = pd.read_csv("/kaggle/input/student-scores-data/student scores.csv")
        df.shape

Out[4]: (25, 2)
```

```
In [5]: df.head()
```

Out[5]:

	Hours	Scores
0	2.5	21
1	5.1	47
2	3.2	27
3	8.5	75
4	3.5	30

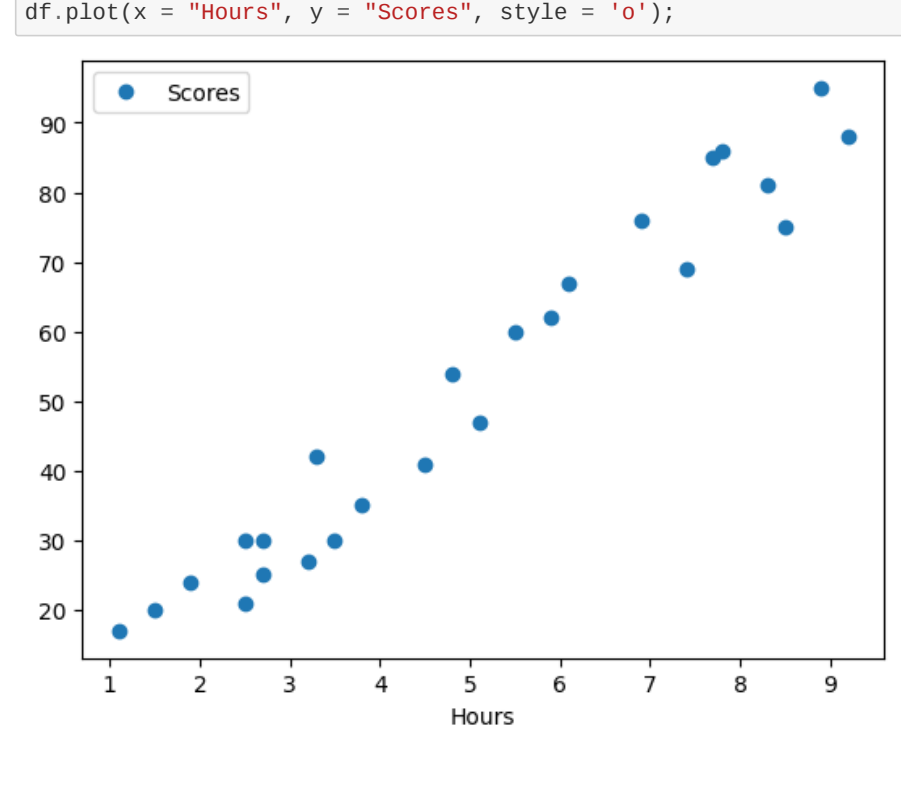
Here scores is what we are trying to predict based on number of hours a student has studied.

```
In [6]: df.describe()
```

Out[6]:

	Hours	Scores
count	25.000000	25.000000
mean	5.012000	51.480000
std	2.525094	25.286887
min	1.100000	17.000000
25%	2.700000	30.000000
50%	4.800000	47.000000
75%	7.400000	75.000000
max	9.200000	95.000000

```
In [8]: # let's plot the data
        df.plot(x = "Hours", y = "Scores", style = 'o');
```



We have to find the best fit line for this data. We have one varibale here that is hours. So, it is simple linear regression

```
In [9]: # create our X and y dataset
        X = df.iloc[:, :-1].values
        y = df.iloc[:, 1].values
        X, y

Out[9]: (array([[2.5],
                [5.1],
                [3.2],
                [8.5],
                [3.5],
                [1.5],
                [9.2],
                [5.5],
                [8.3],
                [2.7],
                [7.7],
                [5.9],
                [4.5],
                [3.3],
                [1.1],
                [8.9],
                [2.5],
                [1.9],
                [6.1],
                [7.4],
                [2.7],
                [4.8],
                [3.8],
                [6.9],
                [7.8]]),
         array([21, 47, 27, 75, 30, 20, 88, 60, 81, 25, 85, 62, 41, 42, 17, 95, 30,
                24, 67, 69, 30, 54, 35, 76, 86]))
```

```
In [10]: # fit the model
         from sklearn.linear_model import LinearRegression
         reg = LinearRegression()
         reg.fit(X, y)

Out[10]: ▸ LinearRegression
         LinearRegression()
```

```
In [11]: reg.intercept_

Out[11]: 2.48367340537321
```

```
In [12]: reg.coef_

Out[12]: array([9.77580339])
```

```
In [13]: # our model will be Y` = 2.48367340537321 + (9.77580339 * X`)
         y_pred = reg.predict(X)
```

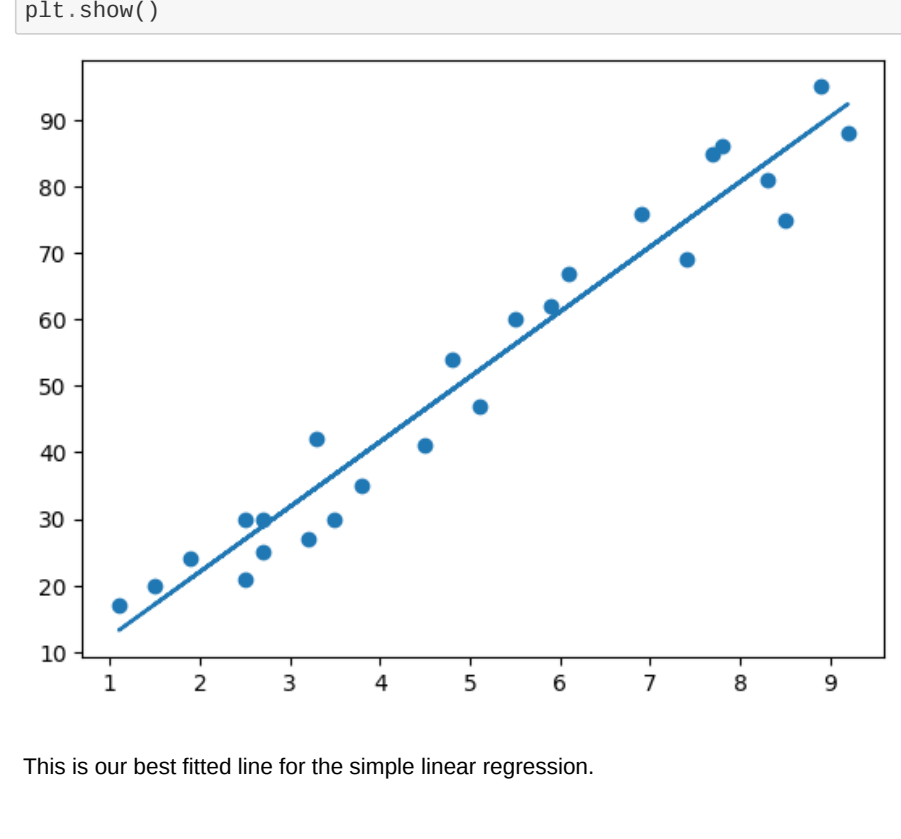
```
In [14]: df_2 = pd.DataFrame({"Actual": y, "Predicted Y`": y_pred})
         df_2
```

Out[14]:

	Actual	Predicted Y`
0	21	26.923182
1	47	52.340271
2	27	33.766244
3	75	85.578002
4	30	36.698985
5	20	17.147378
6	88	92.421065
7	60	56.250592
8	81	83.622842
9	25	28.878343
10	85	77.757360
11	62	60.160913
12	41	46.474789
13	42	34.743825
14	17	13.237057
15	95	89.488324
16	30	26.923182
17	24	21.057700
18	67	62.116074
19	69	74.824618
20	30	28.878343
21	54	49.407530
22	35	39.631726
23	76	69.936717
24	86	78.734940

```
In [15]: # let's find the error (residual)
         from sklearn.metrics import mean_squared_error, r2_score
         print("MSE:", mean_squared_error(y, y_pred))
         print("R2 Score:", r2_score(y, y_pred))

         MSE: 28.882730509245466
         R2 Score: 0.9529481969048356
```

```
In [17]: plt.scatter(X, y)
         plt.plot(X, y_pred)
         plt.show()
```



This is our best fitted line for the simple linear regression.

Multiple Linear Regression

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import pandas as pd

        df = pd.read_csv("/kaggle/input/startup-dataset/startup.csv")
        df.head()
```

Out[1]:

	R&D Spend	Administration	Marketing Spend	Profit
0	165349.20	136897.80	471784.10	192261.83
1	162597.70	151377.59	443898.53	191792.06
2	153441.51	101145.55	407934.54	191050.39
3	144372.41	118671.85	383199.62	182901.99
4	142107.34	91391.77	366168.42	166187.94

We will predict profit in this case. Since we have multiple features it is multiple linear regression

```
In [2]: df.shape

Out[2]: (50, 4)
```

```
In [3]: # divide the datatset into features and target
        X = df.drop("Profit", axis = 1)
        y = df["Profit"]
        X.shape, y.shape

Out[3]: ((50, 3), (50,))
```

```
In [7]: # divide our features and label into train and test set
        from sklearn.linear_model import LinearRegression
        from sklearn.model_selection import train_test_split

        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)
        reg = LinearRegression()
        reg.fit(X_train, y_train)

Out[7]: ▸ LinearRegression
        LinearRegression()
```

```
In [8]: y_pred = reg.predict(X_test)
        y_pred

Out[8]: array([126703.02716461,  84894.75081556,  98893.41815974,  46501.70815036,
               129128.39734381,  50992.69486261, 109016.5536578 , 100878.4644154 ,
                97700.59638629, 113106.15292226])
```

```
In [12]: df_3 = pd.DataFrame({"Actual": y_test, "Predicted Y`": y_pred})
         df_3
```

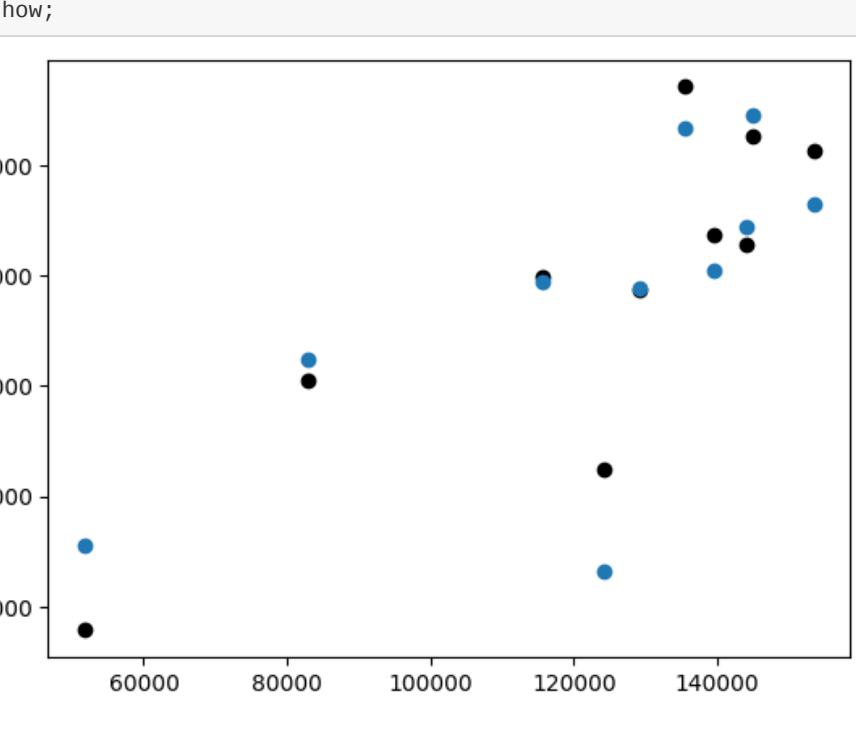
Out[12]:

	Actual	Predicted Y`
13	134307.35	126703.027165
39	81005.76	84894.750816
30	99937.59	98893.418160
45	64926.08	46501.708150
17	125370.37	129128.397344
48	35673.41	50992.694863
26	105733.54	109016.553658
25	107404.34	100878.464145
32	97427.84	97700.596386
19	122776.86	113106.152922

```
In [14]: reg.coef_

Out[14]: array([ 0.80377928, -0.06792917,  0.03124155])
```

```
In [16]: from sklearn.metrics import r2_score, mean_squared_error
         print(mean_squared_error(y_test, y_pred))
         print(r2_score(y_test, y_pred))

         80926321.22295158
         0.9000653083037321
```

```
In [24]: plt.scatter(X_test["Administration"], y_test, color = "black")
         plt.scatter(X_test["Administration"], y_pred)
         plt.show;
```

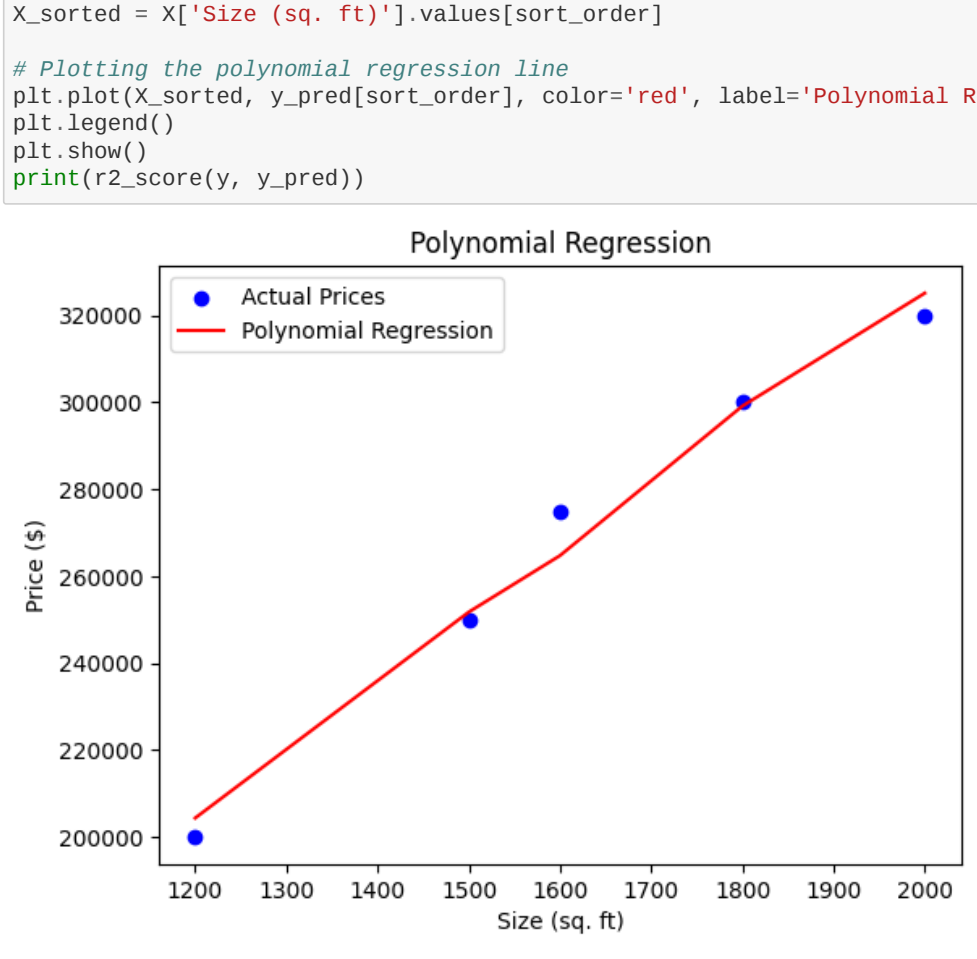


Blue ones are predicted values

Polynomial Regression

We will no implement polynomial regression our example dataset from the tutorial

```
In [32]: import numpy as np
         import matplotlib.pyplot as plt
         from sklearn.linear_model import LinearRegression
         from sklearn.preprocessing import PolynomialFeatures
         import pandas as pd

         # Example dataset
         data = {
             'Size (sq. ft)': [1500, 2000, 1200, 1600, 1800],
             'Number of Bedrooms': [3, 4, 2, 3, 4],
             'Price ($)': [250000, 320000, 200000, 275000, 300000]
         }

         # Creating a DataFrame
         df = pd.DataFrame(data)

         # Extracting features and labels
         X = df[['Size (sq. ft)', 'Number of Bedrooms']]
         y = df['Price ($)']

         # Transforming features to include polynomial terms up to degree 2
         poly = PolynomialFeatures(degree=1)
         X_poly = poly.fit_transform(X)

         # Creating a polynomial regression model
         poly_model = LinearRegression()
         poly_model.fit(X_poly, y)

         # Plotting the results
         plt.scatter(X['Size (sq. ft)'], y, color='blue', label='Actual Prices')
         plt.xlabel('Size (sq. ft)')
         plt.ylabel('Price ($)')
         plt.title('Polynomial Regression')

         # Predicting values with the model
         y_pred = poly_model.predict(X_poly)

         # Sorting X values for better visualization
         sort_order = np.argsort(X['Size (sq. ft)'])
         X_sorted = X['Size (sq. ft)'].values[sort_order]

         # Plotting the polynomial regression line
         plt.plot(X_sorted, y_pred[sort_order], color='red', label='Polynomial Regression')
         plt.legend()
         plt.show()
         print(r2_score(y, y_pred))
```



```
0.9823066841415464
```

As you can see some of our points are not on the curve if you increase the degree of polynomial it will fit perfectly, which is nothing but overfitting when it has perfect 1 score.

Bias-Variance Tradeoff:

In machine learning, when we build a model, we are aiming for it to perform well on unseen data. The predictive performance of a model is often measured by its error on new, unseen examples. This error can be broken down into three components: Bias, Variance, and Irreducible Error.

- Bias: Bias is the error introduced by approximating a real-world problem, which may be very complex, by a much simpler model. High bias can lead the model to underfit the data, meaning it fails to capture the underlying patterns.
- Variance: Variance is the error introduced by having a model that is too complex. A high-variance model is very sensitive to the training data and may capture noise instead of the underlying patterns, leading to overfitting.
- Irreducible Error: This is the inherent noise in the data that cannot be reduced. It's the minimum error that any model will face.

The Bias-Variance Tradeoff is the balance between bias and variance to achieve the best overall predictive performance.

- High Bias, Low Variance: Simple models with high bias may not fit the training data well, but they generalize better to new, unseen data.
- Low Bias, High Variance: Complex models with low bias may fit the training data very well but can perform poorly on new data because they capture noise as if it were a real pattern.

Derivation of Bias-Variance Decomposition

We have a dataset with input-output pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. We are trying to fit a model $\hat{f}(x)$ to predict y given x. The true relationship between x and y is $f(x)$.

Y is the true outcome variable. $\hat{f}(x)$ is our predicted outcome variable.

The Mean Squared Error (MSE) for a single data point is given by:

$MSE = \mathrm{E}[(Y - \hat{f}(x))^2]$

Now, let's expand this expression:

$$\begin{aligned} \mathrm{MSE} &= \mathrm{E}[(Y - \hat{f}(x) + \mathrm{E}[\hat{f}(x)] - \mathrm{E}[\hat{f}(x)])^2] \\ &= \mathrm{E}[(Y - \mathrm{E}[\hat{f}(x)])^2 + (\mathrm{E}[\hat{f}(x)] - \hat{f}(x))^2 + 2(Y - \mathrm{E}[\hat{f}(x)])(\mathrm{E}[\hat{f}(x)] - \hat{f}(x))] \end{aligned}$$

Now, let's break this down into three components: Bias, Variance, and Covariance.

Bias Term (Squared)

$Bias^2 = \mathrm{E}[(Y - \mathrm{E}[\hat{f}(x)])^2]$

This is essentially the squared difference between the true values and the expected predicted values.

Variance Term:

$Variance = \mathrm{E}[(\mathrm{E}[\hat{f}(x)] - \hat{f}(x))^2]$

This is the expected squared difference between each prediction and the expected prediction.

Covariance Term:

$Covariance = 2\mathrm{E}[(Y - \mathrm{E}[\hat{f}(x)])(\mathrm{E}[\hat{f}(x)] - \hat{f}(x))]$

This term represents the covariance between the true values and the expected predicted values.

Now, the MSE can be expressed as:

$MSE = \mathrm{Bias}^2 + \mathrm{Variance} + \mathrm{Covariance}$

But, typically, in machine learning, the covariance term is often neglected because it's considered as part of the irreducible error.

Therefore, the Bias-Variance Decomposition simplifies to:

$MSE = \mathrm{Bias}^2 + \mathrm{Variance} + \mathrm{Irreducible\ Error}$

And that's the Bias-Variance Decomposition! It helps us understand the different sources of errors in our model and guides us in finding the right balance between bias and variance for optimal model performance.

Overfitting:

Definition: Overfitting occurs when a model is too complex, capturing not only the underlying patterns in the training data but also the noise.

Relation to Bias-Variance Decomposition:

Bias (Training Error): Overfitting often leads to low bias on the training data because the model fits the training data very well.

Variance (Test Error): However, it results in high variance on new, unseen data because the model is too sensitive to the specificities of the training data, failing to generalize to new examples.

In our derivation, overfitting corresponds to a situation where the model has low bias but high variance, leading to poor performance on new data due to its sensitivity to noise.

Underfitting:

Definition: Underfitting occurs when a model is too simple, failing to capture the underlying patterns in the training data.

Relation to Bias-Variance Decomposition:

Bias (Training Error): Underfitting leads to high bias on the training data because the model is too simplistic to capture the true patterns.

Variance (Test Error): It also results in high variance on new, unseen data because the model fails to generalize well due to its oversimplified nature.

In our derivation, underfitting corresponds to a situation where the model has high bias and high variance, leading to poor performance both on the training data and new data.

Sklearn Explaination

Linear regression makes several assumptions about the underlying data and the relationship between the variables. These assumptions are important for the interpretation and reliability of the results. Here are the key assumptions of simple linear regression:

1. Linearity:

Assumption: The relationship between the independent variable (X) and the dependent variable (Y) is linear. Implication: The best-fit line is a straight line, and the change in Y for a one-unit change in X is constant.

2. Independence of Residuals:

Assumption: The residuals (errors) are independent of each other. Implication: The error in predicting one data point does not provide information about the error in predicting another data point.

3. Homoscedasticity (Constant Variance of Residuals):

Assumption: The variance of the residuals is constant across all levels of the independent variable. Implication: The spread of the residuals should be roughly constant as the values of X change.

4. Normality of Residuals:

Assumption: The residuals are normally distributed. Implication: The distribution of the residuals should resemble a bell curve, which helps with statistical inference.

5. No Perfect Multicollinearity:

Assumption: In the case of multiple linear regression, there should not be perfect linear relationships among the independent variables. Implication: Each independent variable provides unique information, and no variable is a perfect linear combination of others.

6. No Perfect Collinearity with the Independent Variable:

Assumption: There is no perfect correlation (r=±1) between any two independent variables. Implication: Each independent variable contributes unique information to the regression model.

7. Linearity of Residuals:

Assumption: The relationship between the independent variable and the residuals is linear. Implication: The residuals should not show a systematic pattern when plotted against the independent variable.

These assumptions are crucial for the validity of the statistical inferences drawn from the regression analysis. If these assumptions are violated, it may affect the accuracy and reliability of the estimated coefficients, confidence intervals, and hypothesis tests.

Issue of multicollinearity in the context of linear regression

Multicollinearity occurs when two or more independent variables in a regression model are highly correlated. This high correlation can lead to instability in the coefficient estimates, making them sensitive to small changes in the data. Multicollinearity often results in a design matrix that is close to being singular, meaning that some columns can be expressed as linear combinations of others. This singularity makes it difficult for the linear regression algorithm to distinguish the individual effects of correlated features. In the presence of multicollinearity, the coefficient estimates become unstable, and small changes in the data can lead to significant changes in the estimated coefficients. This sensitivity to random errors in the target variable increases the variance of the coefficient estimates.Multicollinearity can arise in situations where data are collected without a carefully designed experiment. For example, if two variables are highly correlated because they measure similar aspects of a phenomenon, it can lead to multicollinearity. The consequences of multicollinearity include challenges in interpreting the individual contributions of correlated variables, increased standard errors of coefficient estimates, and potential instability in the model's performance. Collecting data in a well-designed experiment can help reduce multicollinearity by ensuring that the independent variables are orthogonal or minimally correlated.

Sklearn allows you to perform Linear Regression using its class `LineaRegression`. It has some basic parameters, attributes and methods:

Paramters

1. fit_intercept (boolean, default=True): Description: Determines whether to calculate the intercept for the linear regression model. If True: The model will calculate and use an intercept in the calculations. If False: Assumes that the data is centered, and no intercept will be used in calculations.
2. copy_X (boolean, default=True): Description: Decides whether to make a copy of the input features (X) or overwrite the input. If True: The algorithm will make a copy of the input features before performing calculations. If False: The algorithm may overwrite the input features. This can be efficient if memory is a concern and the original data doesn't need to be preserved. n_jobs (integer, default=None):
3. Description: Specifies the number of parallel jobs to use for computation. If None: The algorithm will use 1 job unless in a parallel processing context. If -1: The algorithm will use all available processors. If a positive integer: The algorithm will use the specified number of jobs if the problem is large enough and meets specific conditions (e.g., multiple targets or sparse data).
4. positive (boolean, default=False): Description: When set to True, enforces that the coefficients of the linear regression model must be positive. Note: This option is only supported for dense arrays (i.e., not sparse matrices).

Attributes

1. coef_: Estimated coefficients for the linear regression problem.
2. intercept_: Independent term in the linear model.

There are other attributes as well. You should check the sklearn documentation

You can calculate the coefficient of determination of the prediction using `sklearn.metrics.r2_score` and MSE using `sklearn.metrics.mean_squared_error`

In theory notes, we also learnt about Ridge Regression And Lasso Regression. Ridge and Lasso regression are regularization techniques used in linear regression to address potential issues like overfitting and multicollinearity. They add a penalty term to the standard linear regression objective function to prevent the model from becoming too complex or over-relying on certain features. Let's see them in practice.

Ridge Regression

Ridge regression aims to find the best-fitting line while penalizing the model for having large coefficients. It adds a penalty term based on the squared values of the coefficients to the linear regression objective function. The α term controls the strength of the penalty, and higher α values lead to smaller coefficients.

In scikit-learn, the Ridge regression model has several parameters that can be adjusted to customize its behavior:

alpha (float, default=1.0):

Description: Regularization strength. It controls the amount of shrinkage applied to the coefficients. Higher values of alpha lead to more regularization. Effect: Larger alpha values result in more emphasis on regularization, potentially leading to smaller coefficients and a simpler model.

fit_intercept (bool, default=True):

Description: Whether to calculate the intercept for the model. Effect: If set to False, the model will not include an intercept term, assuming that the data is already centered.

normalize (bool, default=False):

Description: If set to True, the input features (X) will be normalized before regression by subtracting the mean and dividing by the l2-norm. Effect: Normalization can be useful when features are on different scales, ensuring that each feature contributes equally to the regularization.

copy_X (bool, default=True):

Description: Whether to create a copy of the input features. If set to False, the input features may be overwritten. Effect: If memory efficiency is a concern, setting copy_X to False can save memory by modifying the input features in-place.

max_iter (int, default=None):

Description: Maximum number of iterations for optimization. If not specified, the solver determines the number of iterations. Effect: Useful for controlling the maximum number of iterations during the optimization process.

tol (float, default=1e-3):

Description: Tolerance for the optimization. If the change in the coefficients is less than tol, the optimization process stops. Effect: A smaller tol value may lead to more precise solutions but may require more iterations.

solver ({'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga'}, default='auto'):

Description: Solver to use for optimization. Effect: Different solvers have different characteristics, and the choice may affect speed and memory usage. 'auto' chooses the solver automatically based on the input data.

random_state (int, RandomState instance, or None, default=None):

Description: If specified, it controls the random number generator for shuffling the data when solver='sag' or solver='saga'. Effect: Helps achieve reproducibility when using stochastic gradient solvers

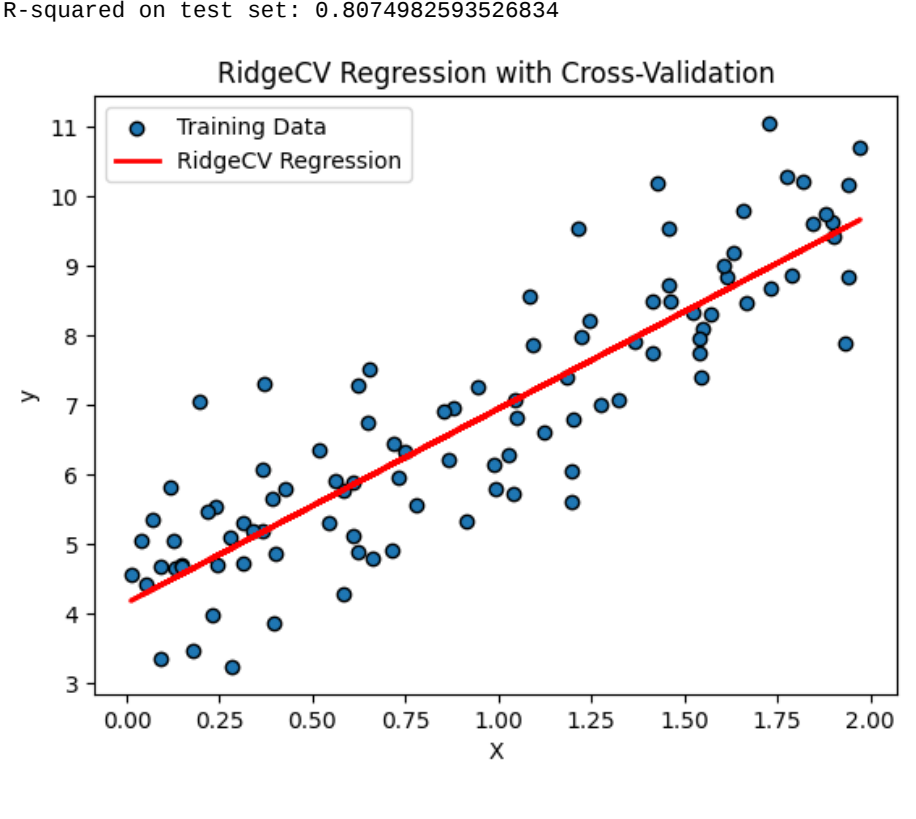
```
In [37]: # this is how you can implement Ridge Regression
         from sklearn.linear_model import Ridge
         reg = Ridge(alpha=.5)

         # fit the example data
         reg.fit(X_train, y_train)
         print(reg.coef_)
         print(reg.intercept_)

         [ 0.80377928 -0.06792917  0.03124155]
         54071.87574528911
```

There is another class in sklearn,called `RidgeCV` that used cross-validation. Cross-validation is a technique used in machine learning to assess the performance of a model and to ensure that the model generalizes well to new, unseen data. It involves splitting the dataset into multiple subsets, training the model on some of these subsets, and evaluating its performance on the remaining data. The key idea is to simulate how well the model is likely to perform on new data that it hasn't seen before. When building a machine learning model, we want to know how well it will perform on data it has never seen. Simply training and testing a model on the same data might not give a true reflection of its generalization ability. Cross-validation helps address this by providing a more robust estimate of a model's performance.

Imagine you have a dataset. Instead of using the entire dataset for training and testing, you divide it into multiple parts, often called "folds." Divide the dataset into 'k' equal-sized folds (subsets).For each iteration, use 'k-1' folds for training and the remaining fold for testing. Repeat this process 'k' times (each fold gets a chance to be the test set). Calculate the performance metric (e.g., accuracy, mean squared error) for each iteration. Average the performance metrics to get a single, more robust estimate. Cross-validation provides a more accurate estimate of a model's performance by considering multiple train-test splits. It helps identify models that may be overfitting the training data but do not generalize well to new data. Performing 'k' train-test splits increases the computational cost. For large datasets, a simple train-test split might be sufficient.

RidgeCV (Ridge Cross-Validation) in scikit-learn is an extension of the Ridge regression model that incorporates cross-validation to automatically select the optimal regularization strength (alpha) from a predefined range. This helps to find the best trade-off between fitting the training data well and preventing overfitting. The main parameter in RidgeCV is alphas, which specifies the range of alpha values to be considered during cross-validation.

```
In [38]: from sklearn.linear_model import RidgeCV

         # data
         np.random.seed(42)
         X = 2 * np.random.rand(100, 1)
         y = 4 + 3 * X + np.random.randn(100, 1)

         # 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_state=42)

         # Create a RidgeCV model with a range of alpha values
         alphas = np.logspace(-6, 6, 13)  # range of alpha values
         ridge_cv_model = RidgeCV(alphas=alphas, store_cv_values=True)

         # Fit the model to the training data
         ridge_cv_model.fit(X_train, y_train)

         # Get the optimal alpha from cross-validation
         optimal_alpha = ridge_cv_model.alpha_
         print("Optimal alpha:", optimal_alpha)

         # Evaluate the model on the test set
         test_score = ridge_cv_model.score(X_test, y_test)
         print("R-squared on test set:", test_score)

         # Plot the results
         plt.scatter(X, y, label='Training Data', marker='o', edgecolors='black', linewidth=1)
         plt.plot(X, ridge_cv_model.predict(X), label='RidgeCV Regression', linewidth=2, color='red')
         plt.title('RidgeCV Regression with Cross-Validation')
         plt.xlabel('X')
         plt.ylabel('y')
         plt.legend()
         plt.show()

         Optimal alpha: 0.1
         R-squared on test set: 0.8074982593526834
```



Lasso Regression

Lasso regression is a type of linear regression that helps build a simple and effective model by preventing it from becoming too complex. It's particularly useful when you have many features, and some of them may not be essential. In linear regression, we try to find the best-fitting line to predict an outcome based on various features. However, with many features, the model might become overly complex, leading to overfitting. Lasso introduces a technique called regularization, which penalizes the model for having too many features or for large values of coefficients. Lasso adds a penalty term to the linear regression objective function, based on the absolute values of the coefficients. This penalty discourages the model from giving too much importance to any one feature, helping to simplify the model. Lasso has a special property - it can drive some of the coefficients to exactly zero. This means that Lasso can be used for feature selection, automatically identifying and ignoring less important features. Lasso minimizes the sum of squared differences between predicted and actual values, plus a penalty term based on the sum of the absolute values of the coefficients. This leads to a balance between fitting the data well and keeping the model simple. Suppose you're predicting house prices based on various features like size, number of bedrooms, and location. Lasso might help in automatically selecting the most relevant features and preventing over-reliance on any one factor. Higher values of α lead to stronger regularization, potentially resulting in fewer features with non-zero coefficients. Lasso helps create a simpler and more interpretable model, especially when dealing with high-dimensional data.

Most of the parameters are same as previously explained. You should check out the documentation for more:

`alpha` (float, default=1.0):

- *Description:* Regularization strength. It controls the amount of shrinkage applied to the coefficients. Higher values of alpha lead to more regularization.
- *Effect:* The range of alpha values determines how much regularization is applied to the model. Smaller alpha values allow the model to be more flexible, while larger alpha values lead to more regularization.

`fit_intercept` (bool, default=True):

- *Description:* Whether to calculate the intercept for the model.
- *Effect:* If set to False, the model will not include an intercept term, assuming that the data is already centered.

`normalize` (bool, default=False):

- *Description:* If set to True, the input features (X) will be normalized before regression by subtracting the mean and dividing by the l2-norm.
- *Effect:* Normalization can be useful when features are on different scales, ensuring that each feature contributes equally to the regularization.

`precompute` ({'auto', True, False}, default=False):

- *Description:* Whether to use a precomputed Gram matrix to speed up calculations.
- *Effect:* It can be set to True if you have already precomputed the Gram matrix, which can speed up the fitting process.

`max_iter` (int, default=1000):

- *Description:* Maximum number of iterations for optimization.
- *Effect:* Useful for controlling the maximum number of iterations during the optimization process.

`tol` (float, default=1e-4):

- *Description:* Tolerance for the optimization. If the change in the coefficients is less than tol, the optimization process stops.
- *Effect:* A smaller tol value may lead to more precise solutions but may require more iterations.

`warm_start` (bool, default=False):

- *Description:* If set to True, reuse the solution of the previous call to fit as initialization for the current call.
- *Effect:* Useful for incremental fitting, especially when you want to fit the model on additional data.

```
In [39]: from sklearn import linear_model
         clf = linear_model.Lasso(alpha=0.1)
         clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
         print(clf.coef_)
         print(clf.intercept_)

         [0.85 0.  ]
         0.15000000000000002
```

In sklearn, you will find various implementation of lasso using different algorithms. The basic one uses coordinate descent. You don't need to remmeber all of them. More of you practice and re-read the sklearn documentation you will learn more. Don't get overwhelmed.

1. Lasso Regression - `linear_model.Lasso`

Concept:

- Lasso (Least Absolute Shrinkage and Selection Operator) is a type of linear regression that adds a penalty term to the ordinary least squares (OLS) objective function.
- The penalty is the sum of the absolute values of the coefficients multiplied by a regularization parameter (alpha).
- It tends to produce sparse models by encouraging some coefficients to become exactly zero, effectively performing feature selection.

Usage:

- It's suitable when you suspect that not all features are relevant, and you want the model to automatically select the most important ones.

---

2. LassoCV - `linear_model.LassoCV`

Concept:

- LassoCV is Lasso regression with built-in cross-validation to find the optimal regularization parameter (alpha).
- Cross-validation involves splitting the dataset into multiple parts, training the model on different subsets, and evaluating performance to find the best alpha.

Usage:

- Helpful when you're unsure about the appropriate regularization strength and want the algorithm to find it for you.

---

3. LassoLarsCV - `linear_model.LassoLarsCV`

Concept:

- LassoLarsCV is Lasso regression using the LARS (Least Angle Regression) algorithm with cross-validation.
- LARS is an algorithm that gradually introduces features into the model while keeping track of the correlations between the features.

Usage:

- Useful when dealing with a large number of features where traditional methods might be computationally expensive.

---

4. LassoLarsIC - `linear_model.LassoLarsIC`

Concept:

- LassoLarsIC is Lasso regression with model selection based on information criteria (AIC or BIC).
- Information criteria help in selecting the best model by considering both model fit and complexity.

Usage:

- Useful when you want an automated way to choose the best model based on certain information criteria.

---

5. Lasso Path - `linear_model.lasso_path`

Concept:

- Lasso path computes the regularization path for Lasso regression, showing how the coefficients change for different alpha values.
- It allows you to visualize the impact of regularization on each feature.

Usage:

- Useful for understanding how the strength of regularization affects the sparsity of the model.

---

6. Multi-task Lasso - `linear_model.MultiTaskLasso`

Concept:

- Multi-task Lasso is an extension of Lasso regression that allows for joint feature selection across multiple tasks.
- Tasks refer to different prediction problems that share common features.
- It applies L1 regularization to encourage sparsity both within each task and across tasks.

Usage:

- Useful when you have multiple related prediction tasks, and you want to select a common set of relevant features for all tasks simultaneously.

Example Code:

```
from sklearn.linear_model import MultiTaskLasso

# Create a Multi-task Lasso model
multi_task_lasso_model = MultiTaskLasso(alpha=1.0)

# Fit the model to the training data for multiple tasks
multi_task_lasso_model.fit(X_train, y_train)

# Make predictions for multiple tasks
predictions = multi_task_lasso_model.predict(X_test)
```

In Multi-task Lasso, the regularization term promotes sparsity not only within each task but also enforces similarity in the feature selection across different tasks. This can be particularly beneficial when tasks share common underlying features.

Elastic Net

Elastic Net is a regression method that combines both L1 (Lasso) and L2 (Ridge) regularization terms in the objective function.It provides a balance between the sparsity-inducing property of Lasso and the smoothing effect of Ridge. The regularization term includes both the sum of squenared coefficits (L2) and the sum of the absolute values of coefficients (L1). It introduces two hyperparameters, alpha (for the overall regularization strength) and l1_ratio (for the ratio between L1 and L2 regularization). It is useful when dealing with a dataset where many features are potentially irrelevant or redundant.

```
In [40]: from sklearn.linear_model import ElasticNet

         # Create an Elastic Net model
         elastic_net_model = ElasticNet(alpha=1.0, l1_ratio=0.5)

         # Fit the model to the training data
         elastic_net_model.fit(X_train, y_train)

         # Make predictions
         predictions = elastic_net_model.predict(X_test)
```

```
In [41]: predictions

Out[41]: array([6.33008716, 7.22095644, 7.08958494, 6.97199558, 6.53930578,
                6.73368233, 6.57506263, 7.18695797, 6.28403217, 6.66336568,
                6.7248864 , 6.91307638, 7.13590014, 7.27889049, 6.39014072,
                6.42917667, 7.08854065, 6.34132531, 7.15409887, 6.44472216])
```

Multi-task Elastic Net

Multi-task Elastic Net extends Elastic Net to handle multiple related prediction tasks. It applies both L1 and L2 regularization across tasks, allowing for joint feature selection and similarity in coefficient values. Similar to Elastic Net, it introduces hyperparameters alpha and l1_ratio for overall and L1/L2 regularization ratios. Useful when you have multiple tasks and want to simultaneously select relevant features and encourage similarity in coefficient values across tasks. In both Elastic Net and Multi-task Elastic Net, the balance between L1 and L2 regularization is controlled by the l1_ratio parameter. A value of 1.0 corresponds to pure Lasso (L1), 0.0 corresponds to pure Ridge (L2), and values in between allow for a combination of both regularization types. These methods are powerful when dealing with datasets with potentially correlated features and multiple prediction tasks.

Here you may have multiple related tasks, each task typically has its own target or outcome variable that you are trying to predict. The features or input variables used for prediction can be shared across tasks, but the targets are distinct.

For example, consider a scenario where you want to predict both the price and the number of bedrooms for houses. In this case:

Multiple Tasks:

Task 1: Predict the price of a house.

Task 2: Predict the number of bedrooms in a house.

Shared Features: Features such as the size of the house, location, and age may be used for predicting both the price and the number of bedrooms.

Distinct Targets: The target variable for Task 1 is the house price, and for Task 2, it is the number of bedrooms.

```
In [42]: from sklearn.linear_model import MultiTaskElasticNet

         np.random.seed(42)
         X = np.random.rand(100, 3)  # 100 samples, 3 features
         y_price = 50 * X[:, 0] + 30 * X[:, 1] + 20 * X[:, 2] + 10 * np.random.randn(100)  # Price task
         y_bedrooms = 2 * X[:, 0] + 3 * X[:, 1] + 1 * X[:, 2] + 2 * np.random.randn(100)  # Bedrooms task

         # Split the data into training and testing sets
         X_train, X_test, y_price_train, y_price_test, y_bedrooms_train, y_bedrooms_test = train_test_split(
             X, y_price, y_bedrooms, test_size=0.2, random_state=42
         )

         # Create a Multi-task Elastic Net model
         multi_task_elastic_net_model = MultiTaskElasticNet(alpha=1.0, l1_ratio=0.5)

         # Fit the model to the training data for both tasks
         multi_task_elastic_net_model.fit(X_train, np.array([y_price_train, y_bedrooms_train]).T)

         # Make predictions for both tasks
         predictions_price, predictions_bedrooms = multi_task_elastic_net_model.predict(X_test).T

         # Evaluate the model
         mse_price = mean_squared_error(y_price_test, predictions_price)
         mse_bedrooms = mean_squared_error(y_bedrooms_test, predictions_bedrooms)

         print(f'Mean Squared Error (Price): {mse_price:.2f}')
         print(f'Mean Squared Error (Bedrooms): {mse_bedrooms:.2f}')

         Mean Squared Error (Price): 524.57
         Mean Squared Error (Bedrooms): 1.93
```

Conceptual Questions

1. **Which Linear Regression training algorithm can you use if you have a training set with millions of features?** Ans: If you have a training set with millions of features, you might consider using a Stochastic Gradient Descent (SGD) variant or a mini-batch variant of Gradient Descent for efficient training. you cannot use the Normal Equa-tion or the SVD approach because the computational complexity grows quickly (more than quadratically) with the number of features. Sklearn LinearRegression class is trained using SVD approach. We will study about SGD in upcoming tutorials.
2. **Suppose the features in your training set have very different scales. Which algorithms might suffer from this, and how? What can you do about it?** e Gradient Descent algorithms will takea long time to converge. To solve this you should scale the data before training the model Note that the Normal Equation or SVD approach will work just fine without scaling. Moreover, regularized models may converge to a suboptimal solution if the features are not scaled: since regularization penalizes large weights,features with smaller values will tend to be ignored compared to features with larger values
3. **Why would you want to use:** Ridge Regression instead of plain Linear Regression (i.e., without any regularization)? Lasso instead of Ridge Regression? Elastic Net instead of Lasso?

- Ridge Regression is preferred over plain Linear Regression when multicollinearity (high correlation between features) is present, as it adds a regularization term that helps stabilize and improve the performance of the model.
- Lasso is favored over Ridge Regression when feature selection is desired, as it tends to shrink some coefficients to exactly zero, effectively excluding certain features from the model.
- Elastic Net is used instead of Lasso when there are multiple correlated features and you want to combine the advantages of both L1 (Lasso) and L2 (Ridge) regularization, offering a compromise between sparsity and handling multicollinearity.

Questions From The Book: Hands On Machine Learning

I highly recommend reading these two pages here before you move ahead.

1. https://scikit-learn.org/stable/auto_examples/inspection/plot_linear_model_coefficient_interpretation.html
2. https://scikit-learn.org/stable/modules/linear_model.html