Understanding Bias and Variance and Implement L1 L2

Bias and variance are two key concepts in machine learning that influence a model's performance. They are crucial to understanding the trade-off between underfitting and overfitting.

Bias

- **Definition:** Bias refers to the error introduced by a model's inability to capture the underlying relationship between the features and the target variable.
- **High Bias:** A model with high bias is underfit, meaning it's too simple to capture the complexities of the data. It performs poorly on both training and testing data.
- **Example:** A linear regression model trying to fit a non-linear relationship would have high bias.

Variance

- **Definition:** Variance measures how sensitive a model is to small changes in the training data. It's a measure of a model's consistency.
- **High Variance:** A model with high variance is overfit, meaning it's too complex and learns the training data too well, but struggles to generalize to new, unseen data.
- **Example:** A decision tree with a large number of leaves might overfit to the training data and have high variance.

The Bias-Variance Trade-off

- **Balance:** The goal is to find a balance between bias and variance to achieve optimal model performance.
- **Underfitting:** A model with high bias and low variance is underfit.
- **Overfitting:** A model with low bias and high variance is overfit.
- **Best Scenario:** The ideal scenario is a model with both low bias and low variance.

Example: Predicting House Prices

- **Underfitting:** A simple linear regression model might underfit the data if the relationship between features (e.g., size, location) and price is complex.
- **Overfitting:** A complex decision tree with many branches might overfit the training data, performing well on it but poorly on new houses.
- **Optimal Model:** A model that strikes a balance, such as a random forest or a neural network with appropriate regularization techniques, might be the best choice.

By understanding bias and variance, you can choose appropriate models and techniques to prevent underfitting and overfitting, leading to more accurate and reliable machine learning models.

Here's a concise explanation of **regularization** along with different techniques and examples for your notes:

Regularization:

Regularization is a technique used in machine learning to prevent **overfitting** by adding a penalty to the loss function of a model. This penalty discourages the model from fitting the training data too closely, leading to better generalization on unseen data.

When a model is too complex (e.g., many parameters in a high-degree polynomial regression or deep decision trees), it can fit the noise in the data, which leads to overfitting. Regularization techniques help reduce the model's complexity while maintaining performance.

Why Regularization?

- **Prevent Overfitting:** Regularization helps control the flexibility of a model to avoid overfitting by penalizing large coefficients in models like linear regression.
- **Improve Generalization:** It improves the model's ability to generalize to new, unseen data by simplifying the model.

Types of Regularization Techniques:

1. L1 Regularization (Lasso Regression):

- **Penalty:** Adds the absolute value of the magnitude of coefficients as a penalty term to the loss function.
- **Effect:** It can shrink some coefficients to **exactly zero**, effectively selecting a simpler model and performing feature selection.
- **Equation:**
  Cost Function = Loss Function + lambda \sum |w_i|

  where lambda is the regularization parameter and w_i are the model coefficients.
- **Example:**
  In **Lasso Regression**, suppose we have a regression model:

  y = w_1.x_1 + w_2.x_2 + w_3.x_3 + epsilon

  If we apply Lasso, some of the coefficients (like w_2) might be shrunk to zero, leading to a simplified model:

y = w_1.x_1 + w_3.x_3 + epsilon

2. **L2 Regularization (Ridge Regression):**

- **Penalty:** Adds the square of the magnitude of coefficients as a penalty term to the loss function.
- **Effect:** It **shrinks the coefficients** but does not eliminate them completely. The model retains all features but with smaller weights.
- **Equation:**

  Cost Function = Loss Function + lambda \sum w_i^2
- **Example:**
  In **Ridge Regression**, given the same model:

  y = w_1.x_1 + w_2.x_2 + w_3.x_3 + epsilon

  Ridge regression will reduce the magnitude of all coefficients without forcing any of them to zero:

  y = \tilde{w}_1.x_1 + \tilde{w}_2.x_2 + \tilde{w}_3.x_3 + epsilon

  where tilde{w}_i are the shrunk coefficients.

3. **Elastic Net:**

- **Combination of L1 and L2 Regularization:**
  Elastic Net combines both L1 (Lasso) and L2 (Ridge) penalties. It is particularly useful when there are multiple correlated features or when feature selection and model complexity reduction are both desired.
- **Effect:** It provides a balance between L1 and L2, allowing the model to select important features (like Lasso) while reducing the magnitude of coefficients (like Ridge).
- **Equation:**

  Cost Function = Loss Function + \lambda_1 \sum |w_i| + \lambda_2 \sum w_i^2
- **Example:**
  In **Elastic Net**, you would adjust the regularization parameters lambda_1 and lambda_2 to determine the balance between L1 and L2 regularization. It would reduce some coefficients to zero while shrinking others.

4. **Dropout (For Neural Networks):**

- **Penalty:** Dropout randomly sets a fraction of the input units (neurons) to zero at each update during training time, which prevents neurons from co-adapting too much.

- **Effect:** Reduces overfitting by preventing the network from relying too heavily on specific neurons.
- **Example:**
  In a neural network, suppose we have a layer with 100 neurons. Dropout with a rate of 0.5 would randomly "drop out" (deactivate) 50 neurons during each training step, forcing the remaining neurons to learn better representations and generalize well.

Summary of Regularization Techniques:

- **L1 (Lasso):** Drives some coefficients to zero, performing feature selection.
- **L2 (Ridge):** Shrinks all coefficients but does not remove any features.
- **Elastic Net:** Combines L1 and L2, balancing between feature selection and coefficient shrinkage.
- **Dropout:** Used in neural networks to randomly deactivate neurons during training, preventing co-dependency.

Real-World Example:

Let's say you're building a model to predict housing prices.

- **Without regularization**, the model might memorize noise in the training data, leading to overfitting.
- **With L1 regularization (Lasso)**, the model could automatically remove irrelevant features like "house color."
- **With L2 regularization (Ridge)**, the model would maintain all features but assign smaller weights to less important ones.

Lets Implement L1(Lasso) and L2(Ridge)

Lets take our house price prediction example

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

In [2]:
```
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
```

In [3]:
```
dataset = pd.DataFrame(housing.data, columns=housing.feature_names)
```

In [4]:
```
dataset['Price'] = housing.target
```

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

Out[5]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Lo
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	

In [6]:
```
## Split the data into independent and dependent features
X = dataset.iloc[:,:-1]
y = dataset.iloc[:,-1]
```

In [7]:
```
## Split the data into train and test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
```

In [8]:
```
## Normalization of the given data points
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_norm = scaler.fit_transform(X_train)
```

In [9]:
```
X_test_norm = scaler.transform(X_test)
X_test_norm[0]
```

Out[9]:
```
array([-1.1526893 , -0.28346293, -0.50781822, -0.16927816, -0.03151006,
        0.06127763,  0.19166399,  0.28664112])
```

In [10]:
```
from sklearn.linear_model import LinearRegression
regression = LinearRegression()
regression.fit(X_train_norm, y_train)
```

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

In [11]:
```
print(regression.coef_)
print(regression.intercept_)
```

```
[ 8.49221760e-01  1.22119309e-01 -2.99558449e-01  3.48409673e-01
 -8.84488134e-04 -4.16980388e-02 -8.93855649e-01 -8.68616688e-01]
2.0692396089424165
```

In [12]:
```
reg_pred = regression.predict(X_test_norm)
reg_pred
```

Out[12]:
```
array([0.72604907, 1.76743383, 2.71092161, ..., 2.07465531, 1.57371395,
       1.82744133])
```

In [13]:
```
## lower error value - MSE and MAE
## higher value - r2 score and adjusted r2 score
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_s
print(mean_squared_error(y_test, reg_pred))
print(mean_absolute_error(y_test, reg_pred))
```

```
print(r2_score(y_test, reg_pred))
print(np.sqrt(mean_squared_error(y_test, reg_pred)))
```

```
0.5305677824766755
0.5272474538305956
0.5957702326061662
0.7284008391515454
```

Regularization: to avoid the overfitting in the model

In [14]:
```
from sklearn.linear_model import Lasso, Ridge
lasso_regression = Lasso(alpha=1.0)
lasso_regression.fit(X_train_norm, y_train)

ridge_regression = Ridge(alpha=1.0)
ridge_regression.fit(X_train_norm, y_train)
```

Out[14]:
```
Ridge()
```

In [15]:
```
lasso_pred = lasso_regression.predict(X_test_norm)
lasso_pred
```

Out[15]:
```
array([2.06923961, 2.06923961, 2.06923961, ..., 2.06923961, 2.06923961,
       2.06923961])
```

In [16]:
```
ridge_pred = ridge_regression.predict(X_test_norm)
ridge_pred
```

Out[16]:
```
array([0.72643939, 1.76724055, 2.71051564, ..., 2.07445267, 1.57387691,
       1.82728481])
```

In [17]:
```
print(np.sqrt(mean_squared_error(y_test, lasso_pred)))
```

```
1.1456636798696462
```

In [18]:
```
print(np.sqrt(mean_squared_error(y_test, ridge_pred)))
```

```
0.7283843311159477
```

The expected value of the square root of mean square error (RMSE) is the expected value of the square root of the average squared difference between the predicted values and the actual values.

Mathematically:

$$E[\sqrt{MSE}] = E[\sqrt{(1/n) * \Sigma(yi - \hat{y}i)^2}]$$

where:

- E is the expected value operator
- MSE is the mean square error
- n is the number of data points
- yi is the actual value for the ith data point
- $\hat{y}i$ is the predicted value for the ith data point

Interpretation:

- The expected value of RMSE provides an estimate of the average error that we can expect to see when using the model to make predictions on new data.
- A lower expected value of RMSE indicates that the model is more accurate, while a higher value suggests that the model is less accurate.

Note:

- The exact value of the expected value of RMSE will depend on the specific model and dataset being used.
- It is often difficult to calculate the exact expected value of RMSE analytically, so it is typically estimated using simulation or other numerical methods.

In [19]:
```
print("Just Regression", regression.predict([X_test_norm[0]]))
print("Lasso Regression", lasso_regression.predict([X_test_norm[0]]))
print("Ridge Regression", ridge_regression.predict([X_test_norm[0]]))
```

```
Just Regression [0.72604907]
Lasso Regression [2.06923961]
Ridge Regression [0.72643939]
```