Regression

Created	@September 29, 2024 6:34 PM
Class	Supervised Learning with scikit-learn

Introduction to Regression in Supervised Learning

Prerequisites

- We'll be using `diabetes` dataset.

Key Details

- Focus: Regression for predicting continuous target variables

Regression Overview

Definition

- Predicts continuous target variables (e.g., GDP, house prices, blood glucose levels)

Data Preparation

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Load and prepare data
df = pd.read_csv('womens_health_data.csv')
X = df.drop('blood_glucose', axis=1).values
y = df['blood_glucose'].values

# Using single feature (BMI) for simplicity
X_bmi = X[:, 3].reshape(-1, 1)
```

Linear Regression Model

Model Fitting

```
reg = LinearRegression()
reg.fit(X_bmi, y)
```

```
# Generate predictions
predictions = reg.predict(X_bmi)
```

Visualization

```
plt.scatter(X_bmi, y)
plt.plot(X_bmi, predictions, color='black')
plt.xlabel('Body Mass Index')
plt.ylabel('Blood Glucose Level')
plt.show()
```

Mathematical Formulation of Linear Regression

Linear regression models the relationship between a dependent variable y and one or more independent variables X:

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n + \epsilon$$

Where:

- y is the dependent variable (target)
- $X_1, X_2, ..., X_n$ are independent variables (features)
- β_0 is the y-intercept
- $\beta_1, \beta_2, ..., \beta_n$ are the coefficients
- ϵ is the error term

For simple linear regression (one feature):

$$y = \beta_0 + \beta_1 X + \epsilon$$

The goal is to find β_0 and β_1 that minimize the sum of squared residuals:

$$\min_{\beta_0, \beta_1} \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_i))^2$$

Key Takeaways

- Regression predicts continuous target variables

- Linear regression fits a straight line to the data
- Data preparation involves separating features (X) and target (y)
- Scikit-learn requires 2D array for features, even with single feature
- Visualizing data and model predictions helps in understanding relationships
- Linear regression shows a weak-to-moderate positive correlation between BMI and blood glucose levels in this example

Understanding Linear Regression

Prerequisites

- We'll be using `advertising_and_sales` dataset.

Key Details

- Focus: Mechanics of linear regression and model evaluation
- Concepts: Simple and multiple linear regression, loss functions, model performance metrics

Linear Regression Fundamentals

Simple Linear Regression

- Equation: $y = ax + b$
  - y: target variable
  - x feature
  - a: slope (coefficient)
  - b: intercept

Error Function (Loss/Cost Function)

- Goal: Minimize vertical distance between fit line and data points
- Residual: Vertical distance between observation and fitted line

- Residual Sum of Squares (RSS): $\sum_{i=1}^{n}(y_i - (ax_i + b))^2$
- Method: Ordinary Least Squares (OLS)

Multiple Linear Regression

- Equation: $y = a_1x_1 + a_2x_2 + ... + a_nx_n + b$
  - $x_1, x_2, ..., x_n$: features
  - $a_1, a_2, ..., a_n$: coefficients for each feature

Implementation with scikit-learn

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.3, random_state=42)

# Create and fit model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)
```

Model Evaluation Metrics

R-squared (Coefficient of Determination)

- Measures the proportion of variance in the target variable explained by the features
- Range: 0 to 1 (1 indicates perfect fit)
- Calculation:

```
r_squared = model.score(X_test, y_test)
print(f"R-squared: {r_squared:.2f}")
```

Mean Squared Error (MSE) and Root Mean Squared Error (RMSE)

- MSE: Average of squared residuals
- RMSE: Square root of MSE (in original units of target variable)
- Calculation:

```
from sklearn.metrics import mean_squared_error

rmse = mean_squared_error(y_test, y_pred, squared=False)
print(f"RMSE: {rmse:.2f}")
```

Mathematical Formulation

Ordinary Least Squares (OLS)

Minimize:

$\sum_{i=1}^{n}(y_i - (a_1x_{1i} + a_2x_{2i} + ... + a_nx_{ni} + b))^2$

R-squared

$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}i)^2}{\sum i=1^{n}(y_i - \bar{y})^2}$

Where $\hat{y}_i$ are predicted values and $\bar{y}$ is the mean of observed values.

RMSE

$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$

Key Takeaways

- Linear regression fits a line to data by minimizing the sum of squared residuals

- Simple linear regression uses one feature, multiple linear regression uses two or more
- R-squared measures the proportion of variance explained by the model
- RMSE provides an interpretable measure of model error in the original units of the target variable
- In the example, the model explains about 35% of blood glucose level variance
- The average prediction error is around 24 mg/dL for blood glucose levels

Cross-Validation in Machine Learning

Prerequisites

- We'll be using `advertising_and_sales` dataset.

Key Details

- Cross-validation mitigates dependence on random train-test splits
- Provides a more robust estimate of model performance
- Assesses model's ability to generalize to unseen data

The Problem with Simple Train-Test Splits

- R-squared on test set depends on specific data points in that split
- May not be representative of model's true generalization ability

Cross-Validation Technique

1. Split dataset into k groups or folds (e.g., 5 folds)
2. Iterative process:
    - Set aside one fold as test set
    - Fit model on remaining folds
    - Predict on test set

    - Compute performance metric (e.g., R-squared)
3. Repeat for each fold as test set
4. Result: k performance metric values

Types of Cross-Validation

- 5-fold CV: dataset split into 5 folds
- 10-fold CV: dataset split into 10 folds
- k-fold CV: dataset split into k folds

Mathematical Representation

Let D be the full dataset, and D_i be the i^{th} fold of the data.

For k-fold cross-validation:

$D = \bigcup_{i=1}^{k} D_i$

Where $D_i \cap D_j = \emptyset \text{ for } i \neq j$

For each iteration i:

- Test set: D_i
- Training set: $D \setminus D_i$

The cross-validation estimate of the performance metric is:

$CV_k = \frac{1}{k} \sum_{i=1}^{k} M_i$

Where M_i is the performance metric (e.g., R-squared) for the i^{th} fold.

R-squared Calculation

R-squared for each fold is calculated as:

$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}i)^2}{\sum i=1^{n} (y_i - \bar{y})^2}$

Where:

- y_i are the true values
- $\hat{y}_i$ are the predicted values

- $\bar{y}$ is the mean of the true values

Confidence Interval Calculation

95% Confidence Interval:

$CI_{95\%} = [\bar{M} - 1.96 \cdot \frac{s}{\sqrt{k}}, \bar{M} + 1.96 \cdot \frac{s}{\sqrt{k}}]$

Where:

- $\bar{M}$ is the mean of the k performance metrics
- s is the standard deviation of the k performance metrics
- k is the number of folds

Trade-offs

- More folds: More computationally expensive
- Reason: More fitting and predicting iterations

Implementing k-fold CV in scikit-learn

```
from sklearn.model_selection import cross_val_score, KFold
from sklearn.linear_model import LinearRegression
import numpy as np

# Create KFold object
kf = KFold(n_splits=6, shuffle=True, random_state=42)

# Instantiate model
model = LinearRegression()

# Perform cross-validation
cv_results = cross_val_score(model, X, y, cv=kf)

# Print results
print(cv_results)
```

```
# Calculate statistics
mean_score = np.mean(cv_results)
std_score = np.std(cv_results)
confidence_interval = np.quantile(cv_results, [0.025, 0.975])

print(f"Mean R-squared: {mean_score}")
print(f"Standard deviation: {std_score}")
print(f"95% Confidence Interval: {confidence_interval}")
```

Key Takeaways

- Cross-validation provides multiple performance metrics
- Allows calculation of mean, median, and confidence intervals
- More robust than single train-test split
- Helps assess model's ability to generalize
- Tradeoff between number of folds and computational cost
- Scikit-learn provides easy implementation with `cross_val_score` and `KFold`

Regularization in Regression

Prerequisites

- We'll be using `diabetes` dataset.

Key Details

- **Regularization** is a technique used to prevent overfitting by penalizing large coefficients in regression models.
- Overfitting occurs when the model fits the training data too well, capturing noise instead of general patterns.

Loss Function in Regression

- Fitting a linear regression model typically involves minimizing a **loss function**:

$$\text{Loss} = \sum (y_{\text{true}} - y_{\text{pred}})^2$$

- In regularized regression, the loss function is altered to include a penalty for large coefficients.

Ridge Regression

Key Concepts

- Ridge regression adds a **penalty term** based on the sum of the squared values of the coefficients:

$$\text{Ridge Loss} = \sum (y_{\text{true}} - y_{\text{pred}})^2 + \alpha \sum \beta_i^2$$

  - α is a hyperparameter controlling the strength of regularization.
  - β_i represents the coefficients of the regression model.

Impact of Alpha α:

- When $\alpha = 0$, ridge regression behaves like ordinary least squares (**OLS**) regression, leading to possible **overfitting**.
- A high α penalizes large coefficients, which can result in **underfitting**.

Code Example: Ridge Regression

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression

# Create synthetic data
X, y = make_regression(n_samples=100, n_features=2, noise=0.
```

```
1, random_state=42)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.2, random_state=42)

# Initialize Ridge regression with alpha=1.0
ridge_model = Ridge(alpha=1.0)
ridge_model.fit(X_train, y_train)

# Predict and display coefficients
y_pred = ridge_model.predict(X_test)
print("Ridge coefficients:", ridge_model.coef_)
```

Choosing Alpha:

- α is a hyperparameter that balances model complexity.
  - A small α leads to minimal regularization (closer to OLS).
  - A large α increases regularization, reducing overfitting but risking underfitting.

Lasso Regression

Key Concepts

- **Lasso** (Least Absolute Shrinkage and Selection Operator) regression modifies the loss function by adding the **absolute values** of the coefficients:

$$\text{Lasso Loss} = \sum (y_{\text{true}} - y_{\text{pred}})^2 + \alpha \sum |\beta_i|$$

  - Lasso can shrink some coefficients to zero, effectively performing **feature selection**.

Impact of Alpha α:

- Similar to Ridge, α controls the strength of regularization.
  - A large α can result in **underfitting** by shrinking too many coefficients to zero.

Feature Selection:

- Lasso is useful for identifying **important features** in a dataset by zeroing out less important coefficients.

Code Example: Lasso Regression

```
from sklearn.linear_model import Lasso

# Initialize Lasso with alpha=0.1
lasso_model = Lasso(alpha=0.1)

# Fit model to the entire dataset
lasso_model.fit(X, y)

# Extract coefficients
lasso_coef = lasso_model.coef_
print("Lasso coefficients:", lasso_coef)

# Plotting coefficients
import matplotlib.pyplot as plt
plt.bar(range(len(lasso_coef)), lasso_coef)
plt.show()
```

Feature Importance:

- In the example, Lasso regression helps determine the most important predictor for blood glucose levels. The most important feature is whether an individual has **diabetes**, which aligns with expectations.

ElasticNet Regression

Key Concepts

- **ElasticNet** combines the penalties of both Ridge and Lasso regression:

$$\text{ElasticNet Loss} = \sum(y_{\text{true}} - y_{\text{pred}})^2 + \alpha_1 \sum \beta_i^2 + \alpha_2 \sum |\beta_i|$$

  - ElasticNet strikes a balance between Ridge's squared penalty and Lasso's absolute value penalty.

Code Example: ElasticNet

```
from sklearn.linear_model import ElasticNet

# Initialize ElasticNet with specific alpha parameters
elasticnet_model = ElasticNet(alpha=0.1, l1_ratio=0.5)

# Fit the model and make predictions
elasticnet_model.fit(X_train, y_train)
y_pred = elasticnet_model.predict(X_test)
```

Summary of Regularization Techniques

Ridge Regression:

- Penalizes the **squared values** of the coefficients.
- Effective for datasets where **all features are relevant**, but overfitting needs control.

Lasso Regression:

- Penalizes the **absolute values** of the coefficients.
- Performs **feature selection** by shrinking irrelevant coefficients to zero.

ElasticNet Regression:

- Combines both Ridge and Lasso penalties.
- Useful for cases where **some features** are important, but **feature selection** is needed.

Practical Considerations

- Regularization techniques are essential when dealing with high-dimensional data where overfitting is a concern.
- Tuning the \(\alpha\) parameter is critical to finding the right balance between underfitting and overfitting.