REGRESSION

Complete Guide to Regression

1. Overview of Regression

- **Regression** is a type of supervised machine learning algorithm used for predicting a continuous outcome (target variable) based on one or more input variables (features or predictors). Unlike classification, which predicts discrete class labels, regression models predict a numerical value, such as house prices, temperature, stock prices, or sales figures. Predicting house prices.

---

2. Types of Regression Models

1. Linear Regression

What It Is

- A basic regression model that assumes a linear relationship between the independent variables (X) and the dependent variable (y).
- **Formula: y=** β0+β1x1+β2x2+…+βnxn+ϵ Where:
  - y: Target variable.
  - βbeta: Intercept.
  - β1,β2,…, : Coefficients (slopes).
  - x1,x2,…,xnx_1, x_2, \ldots, x_nx1,x2,…,xn: Predictors (features).
  - **ϵ\epsilonϵ: Error term.**

Use Cases

- Predicting house prices based on square footage, number of bedrooms, etc.
- Estimating sales revenue based on advertising spend.

Strengths

- Easy to interpret.
- Computationally efficient.
- Works well for datasets with linear relationships**.**

Weaknesses

- Performs poorly if the relationship is non-linear.
- Sensitive to outliers.

Code Example

```
from sklearn.linear_model import LinearRegression

# Initialize and train the model
model = LinearRegression()
model.fit(X_train, y_train)

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

---

2. Polynomial Regression

What It Is

- A variation of linear regression that models non-linear relationships by adding polynomial terms to the predictors.
- Formula: $y= \beta_0+\beta_1 x+\beta_2 x^2+\ldots+\beta_n x^n$

Use Cases

- Modeling curves, such as stock price movements or growth patterns.
- Capturing non-linear trends in sales or revenue over time.

Strengths

- Extends linear regression to non-linear relationships.
- Retains simplicity for moderately complex relationships.

Weaknesses

- Prone to overfitting for high-degree polynomials.
- Harder to interpret compared to linear regression.

Code Example

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

# Create a polynomial regression model
poly_model = make_pipeline(PolynomialFeatures(degree=3), LinearRegression())
poly_model.fit(X_train, y_train)

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

---

3. Ridge Regression

What It Is

- A regularized version of linear regression that adds an L2 penalty to the cost function:
- **Loss Function**:

$$L==RSS+\lambda\sum_{j=1}^{n}\beta_j^2$$

Where:

- ***RSS***: Residual Sum of Squares
  $RSS=\sum(y_i-\hat{y}_i)^2$
    - y_i: Actual observed value.
    - $\hat{y}_i$: Predicted value.
    - m: Number of observations.
- $\boldsymbol{\lambda}$:
    - Regularization parameter that controls the strength of the penalty.
    - Higher λλ values increase regularization and shrink the coefficients more strongly.
- $\boldsymbol{\beta_j}$:
    - Coefficients of the independent variables in the regression model.
    - The L1 norm, $\sum_{j=1}^{n}|\beta_j|$, enforces sparsity by driving some coefficients to zero.
- ***n***:
    - Number of features or predictors in the model.
    - **Where**:
        - λ: Regularization parameter
        - RSS: Residual sum of squares

Use Cases

- When features are correlated (multicollinearity).
- To prevent overfitting in linear regression.

Strengths

- Reduces model complexity.
- Handles multicollinearity better than standard linear regression.

Weaknesses

- Does not perform feature selection (all features contribute).

Code Example

```
from sklearn.linear_model import Ridge

ridge = Ridge(alpha=1.0)
ridge.fit(X_train, y_train)
y_pred = ridge.predict(X_test)
```

---

4. Lasso Regression

What It Is

- Similar to Ridge Regression but uses L1 regularization
- **Loss Function**:
- $L=\text{RSS}+\lambda\sum_{j=1}^{n}|\beta_j|$

    - Lasso shrinks some coefficients to exactly zero, effectively performing feature selection.

Use Cases

- When there are many irrelevant or redundant features.

Strengths

- Performs feature selection by shrinking irrelevant coefficients to zero.
- Prevents overfitting.

Weaknesses

- May underfit when α\alphaα is too high.

Code Example

```
from sklearn.linear_model import Lasso

lasso = Lasso(alpha=0.01)
lasso.fit(X_train, y_train)
y_pred = lasso.predict(X_test)
```

---

5. ElasticNet Regression

What It Is

Combines Ridge (L2) and Lasso (L1) penalties:

Loss Function:

$$L=\text{RSS}+\lambda_1\sum_{j=1}^{n}|\beta_j|+\lambda_2\sum_{j=1}^{n}\beta_j^2$$

- **Where**:
    - λ_1: L1 regularization parameter
    - λ_2: L2 regularization parameter
-
    - λ\lambda: Balances L1 and L2 penalties.

Use Cases

- When the dataset has many correlated and irrelevant features.

Strengths

- Combines the strengths of Ridge and Lasso regression.

Weaknesses

- Requires tuning two hyperparameters (α\alphaα and λ\lambdaλ**).**

Code Example

```
from sklearn.linear_model import ElasticNet

elastic_net = ElasticNet(alpha=0.01, l1_ratio=0.7)
elastic_net.fit(X_train, y_train)
y_pred = elastic_net.predict(X_test)
```

---

6. Decision Tree Regression

What It Is

- A non-linear model that splits the data into regions by creating decision boundaries.

Use Cases

- Predicting outcomes where feature interactions are important.
- Handling non-linear relationships, e.g., predicting sales based on product categories.

Strengths

- Handles both numerical and categorical data.
- Captures non-linear relationships**.**

Weaknesses

- Prone to overfitting (mitigated by pruning or limiting tree depth).

Code Example

```
from sklearn.tree import DecisionTreeRegressor

tree = DecisionTreeRegressor(max_depth=5)
tree.fit(X_train, y_train)
y_pred = tree.predict(X_test)
```

---

7. Random Forest Regression

What It Is

- An ensemble model that averages predictions from multiple decision trees.

Use Cases

- Handling large datasets with complex interactions.
- Predicting outcomes with high variance or noise.

Strengths

- Reduces overfitting compared to a single decision tree.
- Works well for both linear and non-linear relationships**.**

Weaknesses

- Computationally intensive.
- Less interpretable than individual trees.

Code Example

```
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(n_estimators=100)
rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
```

---

8. Gradient Boosting Regression

What It Is

- An ensemble model that builds trees sequentially, improving residual errors at each step.

Use Cases

- Predicting complex, non-linear relationships.
- Applications in structured data problems (e.g., finance, healthcare).

Strengths

- High accuracy.
- Handles non-linear data.

Weaknesses

- Sensitive to hyperparameter tuning.
- Slower training than Random Forests.

Code Example

```
from sklearn.ensemble import GradientBoostingRegressor

gbr = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=3)
gbr.fit(X_train, y_train)
y_pred = gbr.predict(X_test)
```

---

9. Support Vector Regression (SVR)
What It Is

- Fits a hyperplane within a "margin" that captures the majority of data points.

Use Cases

- Medium-sized datasets with noise.
- Predicting outcomes where feature scaling is critical.

Strengths

- Effective in high-dimensional spaces.
- Handles non-linear relationships with kernels.

Weaknesses

- Computationally expensive for large datasets.
- Requires feature scaling.

Code Example

```
from sklearn.svm import SVR

svr = SVR(kernel='rbf')
svr.fit(X_train, y_train)
y_pred = svr.predict(X_test)
```

---

10. Neural Networks for Regression
What It Is

- A highly flexible, non-linear model using layers of neurons to learn complex patterns.

Use Cases

- Predicting outcomes with high complexity (e.g., weather forecasting, image-based predictions).

Strengths

- Handles very complex and non-linear relationships.

Weaknesses

- Requires large datasets and significant computational resources.
- Requires tuning multiple hyperparameters.

Code Example

```
from sklearn.neural_network import MLPRegressor

nn = MLPRegressor(hidden_layer_sizes=(100, 50), max_iter=500)
nn.fit(X_train, y_train)
y_pred = nn.predict(X_test)
```

---

3. Regression Workflow

Step 1: Data Preprocessing

- Handle missing values:

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='mean')
X = imputer.fit_transform(X)
```

- Encode categorical variables:

```
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder()
encoded_features = encoder.fit_transform(X[['category_column']])
```

- Scale features:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Step 2: Train-Test Split

Split data for training and evaluation:

```
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, random_state=42)
```

Step 3: Model Selection

Choose a model that aligns with the data complexity:

- Start with simple models (e.g., Linear Regression).
- Experiment with more complex models (e.g., Random Forests).

Step 4: Model Training

Train the chosen model on the training set:

```
model.fit(X_train, y_train)
```

Step 5: Making Predictions

Predict the target variable for the test set:

```
y_pred = model.predict(X_test)
```

Step 6: Model Evaluation

Evaluate Performance

Evaluation metrics are critical to understand how well your regression model predicts the target variable. Different metrics provide insights into errors and how much variance the model explains. Here's a deep dive into key metrics:

1. Mean Absolute Error (MAE)

Definition: The average of absolute differences between actual and predicted values.

Formula: $MAE=1/n\sum|y_i-\hat{y}_i|$

-
- **Interpretation:**
  - MAE provides a straightforward measure of average error.

    - The smaller the MAE, the better the model performs.
- **Range:**
    - Always non-negative (0 is the best score, indicating perfect predictions).
    - Same unit as the target variable.

from sklearn.metrics import mean_absolute_error
mae = mean_absolute_error(y_test, y_pred)
print(f"MAE: {mae}")

2. Mean Squared Error (MSE)

- **Definition:** The average of squared differences between actual and predicted values.
- **Formula:**
- $MSE=1/n\sum(y_i-\hat{y}_i)^2$
- **Where**:
    - y_i: Actual value
    - $\hat{y}_i$: Predicted value
    - n: Number of observations
-
- **Interpretation:**
    - Heavily penalizes large errors due to squaring.
    - Useful for detecting significant prediction issues.
- **Range:** Non-negative (lower is better).

from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, y_pred)
print(f"MSE: {mse}")

3. Root Mean Squared Error (RMSE)

- **Definition:** Square root of the MSE.
- **Formula:** $RMSE=\sqrt{\frac{1}{n}\Sigma(y_i-\hat{y}_i\ )2}$
- **Interpretation:**
    - Similar to MSE but in the same unit as the target variable.
    - A smaller RMSE indicates better performance.
- **Range:** Non-negative (lower is better).

rmse = mse ** 0.5
print(f"RMSE: {rmse}")

4. R² Score (Coefficient of Determination)

- **Definition:** Proportion of the variance in the target variable explained by the model.
- **Formula:** $R^2=1-\left(\sum(y_i-\hat{y}_i)^2/\sum(y_i-\bar{y})2\ \right)$
- $\bar{y}$: Mean of actual values
- **Interpretation:**
    - R2=1: Perfect predictions.

        - $R2=0$: Model performs no better than the mean.
        - $R2<0$: Model performs worse than the mean.
- **Range:** $-\infty$ to 1.

```
from sklearn.metrics import r2_score
r2 = r2_score(y_test, y_pred)
print(f"R²: {r2}")
```

Step 7:Hyperparameter Tuning

Hyperparameter tuning is essential to optimize a model's performance. Unlike parameters learned during training, hyperparameters are set before training and control the behavior of the model.

Key Hyperparameters in Common Regression Models:

1. **Linear Regression**:
    - Regularization strength for Ridge or Lasso regression (alpha):
        - Controls the penalty for large coefficients.
        - Higher alpha values increase regularization, which can reduce overfitting but may underfit.

```
from sklearn.linear_model import Ridge
model = Ridge(alpha=1.0)
```

2. **Tree-Based Models**:
    - **max_depth:** Maximum depth of the tree.
        - Limits overfitting by controlling tree size.
        - A deeper tree captures more complexity but risks overfitting.
    - **n_estimators:** Number of trees in ensembles (Random Forest, Gradient Boosting).
        - More trees improve accuracy but increase computation time.
    - **min_samples_split:** Minimum number of samples to split a node.
        - Larger values prevent splitting small samples and reduce overfitting.

```
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor(n_estimators=100, max_depth=10)
```

3. **Support Vector Regression (SVR)**:
    - **C:** Regularization parameter.
        - Smaller values specify stronger regularization, reducing overfitting.
    - **kernel:** Type of kernel function (linear, polynomial, RBF, etc.).
    - **epsilon:** Tolerance for error in predictions.

```
from sklearn.svm import SVR
model = SVR(kernel='rbf', C=1.0, epsilon=0.1)
```

Methods for Hyperparameter Tuning:

1. **Grid Search**
    - **What it does:** Exhaustively searches through a manually specified subset of hyperparameters.

   - **Advantages:**
     - Guarantees the best combination within the grid.
   - **Disadvantages:**
     - Computationally expensive.

```
from sklearn.model_selection import GridSearchCV

param_grid = {'alpha': [0.01, 0.1, 1, 10]}
grid_search = GridSearchCV(Ridge(), param_grid, cv=5, scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)

print(f"Best Parameters: {grid_search.best_params_}")
```

2. **Randomized Search**
   - **What it does:** Randomly samples hyperparameter values within defined ranges.
   - **Advantages:**
     - Faster than Grid Search.
     - Effective for large search spaces.
   - **Disadvantages:**
     - May not find the absolute best combination.

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform

param_distributions = {'alpha': uniform(0.01, 10)}
random_search = RandomizedSearchCV(Ridge(), param_distributions, n_iter=50, cv=5)
random_search.fit(X_train, y_train)

print(f"Best Parameters: {random_search.best_params_}")
```

3. **Bayesian Optimization**
   - **What it does:** Models the hyperparameter space as a probabilistic function and iteratively updates it to find optimal values.
   - **Advantages:**
     - More efficient than Grid Search or Randomized Search.
   - **Disadvantages:**
     - Requires additional libraries (e.g., Optuna, Hyperopt).

```
import optuna

def objective(trial):
    alpha = trial.suggest_loguniform('alpha', 0.01, 10)
    model = Ridge(alpha=alpha)
    model.fit(X_train, y_train)
    return -mean_squared_error(y_test, model.predict(X_test))

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=50)
```

```
print(f"Best Parameters: {study.best_params}")
```

Step 8: Residual Analysis

Residual analysis is a crucial part of evaluating the performance of a regression model. It helps identify whether the assumptions of the regression model are met and can reveal if any patterns or issues remain in the model's predictions.

What Are Residuals?

Residuals are the differences between the observed values and the predicted values. In other words, they show how far off your model's predictions are from the actual data.

Formula:

$$Residual = y_i - \hat{y}_i$$

Where:

- y_i: Actual observed value for the ii-th data point.
- $\hat{y}_i$: Predicted value for the ii-th data point.
-

Why Residual Analysis is Important

- **Assumption Checking**: In regression, several assumptions must hold for the model to be valid. Residuals can be used to verify these assumptions:
  1. **Linearity**: The relationship between predictors and the target is linear.
  2. **Independence**: The residuals are independent of each other (no autocorrelation).
  3. **Homoscedasticity**: Constant variance of residuals across the range of fitted values.
  4. **Normality**: Residuals should be normally distributed.
- **Model Improvement**: Analyzing residuals helps identify:
  - Bias: Whether the model is systematically underestimating or overestimating predictions.
  - Outliers: Extreme values that can affect the model’s performance.
  - Heteroscedasticity: Changing variance of residuals, which violates assumptions.
  - Non-linearity: If the model misses certain patterns in the data.

How to Perform Residual Analysis

1. Plotting Residuals vs. Fitted Values

- The residuals should be randomly scattered around the horizontal axis if the model is well-fitted. Any patterns (e.g., funnel shapes, curvatures) suggest that the model is not capturing some aspect of the data (e.g., non-linearity, heteroscedasticity).

Code Example

```
import matplotlib.pyplot as plt
import seaborn as sns

# Calculate residuals
residuals = y_test - y_pred

# Plot residuals vs. predicted values
```

```
plt.scatter(y_pred, residuals)
plt.axhline(y=0, color='red', linestyle='--')
plt.xlabel('Fitted Values (Predicted)')
plt.ylabel('Residuals')
plt.title('Residuals vs. Fitted Values')
plt.show()
```

Interpretation

- **No Pattern**: If the plot shows a random scatter of points, the model is likely well-specified and the assumptions of linearity, independence, and homoscedasticity may hold.
- **Patterned Residuals**: If there is a discernible pattern (like a funnel or curve), it indicates the model is not capturing some aspect of the data (e.g., a non-linear relationship between predictors and the target).

2. Histogram of Residuals

- Plotting a histogram of residuals helps check for **normality**. In regression models, we assume that the residuals are normally distributed (especially important for significance testing and confidence intervals).

Code Example

```
# Histogram of residuals
sns.histplot(residuals, kde=True)
plt.title('Histogram of Residuals')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.show()
```

Interpretation

- **Normal Distribution**: If the histogram follows a bell-shaped curve, the residuals are approximately normally distributed.
- **Non-normality**: If the histogram is skewed or has heavy tails, this could indicate that the residuals are not normally distributed, which violates one of the regression assumptions.

3. Q-Q Plot (Quantile-Quantile Plot)

- A Q-Q plot compares the quantiles of the residuals to the quantiles of a normal distribution. It’s a more formal test for normality.

Code Example

```
import scipy.stats as stats

# Q-Q plot
stats.probplot(residuals, dist="norm", plot=plt)
plt.title('Q-Q Plot of Residuals')
plt.show()
```

Interpretation

- **Straight Line**: If the points fall along the straight line, the residuals are normally distributed.
- **Deviations from Line**: If the points deviate significantly from the line, it suggests that the residuals are not normally distributed, indicating possible issues with the model.

4. Scale-Location Plot (Spread-Location Plot)

- This plot shows the residuals' spread against fitted values and helps check for **heteroscedasticity** (variance of residuals changing with fitted values). A good model should have a uniform spread of residuals.

Code Example

```
# Plot scale-location
plt.scatter(y_pred, np.sqrt(np.abs(residuals)))
plt.axhline(y=0, color='red', linestyle='--')
plt.xlabel('Fitted Values')
plt.ylabel('Square Root of |Residuals|')
plt.title('Scale-Location Plot')
plt.show()
```

Interpretation

- **Uniform Spread**: If the plot shows a horizontal line with equally spread points, it suggests that the residuals have constant variance (homoscedasticity).
- **Funnel Shape**: If the spread of residuals increases or decreases with fitted values, it indicates **heteroscedasticity**, meaning the model's variance is not constant and the regression assumptions are violated.

5. Leverage and Influence

- **Leverage** refers to how far away the independent variable values are from their mean. High leverage points have the potential to significantly influence the model's predictions.
- **Influence** refers to how much a data point impacts the regression model. High influence points can pull the regression line toward them, resulting in biased coefficients.

Code Example for Leverage

```
from sklearn.metrics import mean_squared_error

# Plot leverage
leverage = (X_train - X_train.mean(axis=0)) / X_train.std(axis=0)
plt.scatter(leverage, residuals)
plt.xlabel('Leverage')
plt.ylabel('Residuals')
plt.title('Leverage vs. Residuals')
plt.show()
```

Interpretation

- **High Leverage Points**: Identifying points with high leverage can help detect influential outliers. These points should be closely monitored, as they can heavily influence the model's results.

What to Look for in Residual Analysis

- **Linearity**: If you see patterns like curves or bends in the residual plots, it suggests that the relationship between the predictors and the target may not be linear, and you may need to use non-linear models.
- **Homoscedasticity**: If the spread of residuals increases or decreases with fitted values, this violates the homoscedasticity assumption. In such cases, you can try transforming

the target variable or using models that are robust to heteroscedasticity (e.g., weighted least squares regression).

- **Normality**: If residuals deviate from normality, you may need to transform the target or predictors or explore non-linear models.
- **Outliers**: Identifying large residuals that are far from zero can highlight outliers or influential points that may need to be removed or handled differently.

Summary of Residual Analysis

- **Residual Analysis** helps assess whether the assumptions of your regression model hold true and identify areas for improvement.
- **Key plots**:
    1. **Residuals vs. Fitted Values**: Check for linearity and homoscedasticity.
    2. **Histogram and Q-Q Plot**: Assess normality of residuals.
    3. **Scale-Location Plot**: Check for heteroscedasticity.
    4. **Leverage and Influence**: Identify outliers or influential data points.

By performing residual analysis, you can improve the robustness and reliability of your regression model, ensuring it generalizes well to unseen data.

---

4. Common Problems and Solutions

Problem: Multicollinearity

- **Solution:** Remove correlated features using VIF.

```
from statsmodels.stats.outliers_influence import variance_inflation_factor

vif = pd.DataFrame()
vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
```

Problem: Outliers

- **Solution:** Detect and handle extreme values.

```
from scipy.stats import zscore
data['z_score'] = zscore(data['feature'])
data = data[data['z_score'] < 3]
```

Problem: Overfitting

- **Solution:** Regularize models (e.g., Ridge, Lasso).

Problem: Non-Linear Patterns

- **Solution:** Use polynomial or tree-based models.

Problem: Missing Values

- **Solution:** Impute missing data.

---

5. Advanced Techniques

Cross-Validation

Evaluate model stability using multiple splits:

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, X, y, cv=5, scoring='neg_mean_squared_error')
print(f"Mean CV Score: {-scores.mean()}")
```

Feature Importance
Identify significant predictors for interpretability:

```
importances = rf_model.feature_importances_
plt.bar(range(len(importances)), importances)
plt.show()
```