Machine Learning with Python(M1)

Created	@December 9, 2025 10:50 AM
Module Code	IBMM01: Introduction to Machine Learning

Module 5: Evaluating and Validating Machine Learning Models

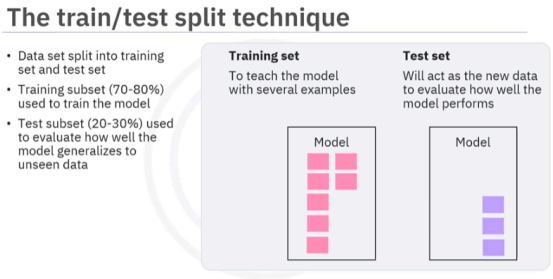
1. Supervised Learning Evaluation

- Purpose: Measures how well a machine learning model predicts outcomes on **unseen data**.
- Importance: Helps understand model effectiveness during both training and testing phases.
- Method: Compare **model predictions** to **ground truth labels**.

---

2. Train-Test-Split Technique

- Splits the dataset into:
  - **Training set** (70–80%): Used to train the model.
  - **Test set** (20–30%): Used to evaluate generalization to new data.
- Ensures the model is not overfitting to all available data.



---

3. Key Classification Metrics

1. **Accuracy**
   - Formula:

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

- Example: 70% accuracy if 7 out of 10 predictions are correct.

2. **Confusion Matrix**
   - A table comparing **true labels** vs **predicted labels**.
   - Components:
     - **True Positive (TP):** Predicted positive, actually positive.
     - **True Negative (TN):** Predicted negative, actually negative.
     - **False Positive (FP):** Predicted positive, actually negative.
     - **False Negative (FN):** Predicted negative, actually positive.
3. **Precision**
   - Measures the accuracy of positive predictions.
   - Formula:

$$\text{Precision} = \frac{TP}{TP + FP}$$

   - Important when **false positives are costly**, e.g., movie recommendations.
4. **Recall**
   - Measures coverage of actual positives.
   - Formula:

$$\text{Recall} = \frac{TP}{TP + FN}$$

   - Important when **false negatives are costly**, e.g., medical diagnoses.
5. **F1 Score**
   - Harmonic mean of precision and recall:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

   - Useful when **both precision and recall are important**, e.g., healthcare scenarios.

---

4. Practical Example

- **Pass/Fail Test Predictions:**
  - Correct predictions = green squares.
  - Incorrect predictions = grey squares.
  - Accuracy = Correct / Total.
  - Precision & recall calculated specifically for the "pass" class.
- **Iris Flower Classification with KNN:**
  - Accuracy: 93%.
  - Confusion matrix visualized with a heatmap.
  - Setosa class achieved perfect precision, recall, and F1 score.

---

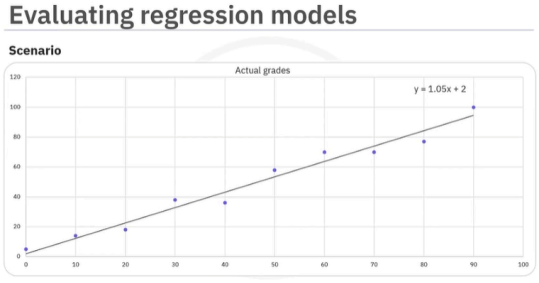
5. Summary Table

Metric	Purpose	Formula	Use Case Example
Accuracy	Overall correctness	(TP + TN) / Total	General model performance
Precision	Correctness of positive predictions	TP / (TP + FP)	Movie recommendations
Recall	Coverage of actual positives	TP / (TP + FN)	Medical diagnosis
F1 Score	Balance between precision and recall	2 * (Precision * Recall) / (P + R)	Health predictions, critical tasks

Regression Metrics and Evaluation Techniques

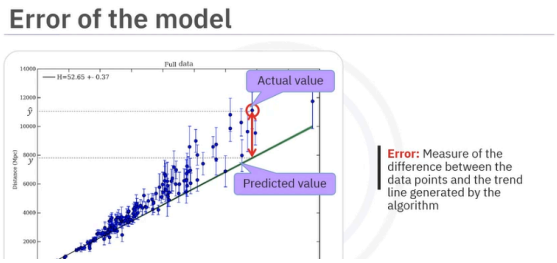
1. Purpose of Regression Evaluation

- Regression models predict **continuous numerical values** (e.g., exam scores, house prices).
- Evaluation is needed because models **can make errors**, and measuring these errors shows how well a model predicts unseen data.



Error = Difference between the predicted value ($\hat{y}_i$) and the actual value (y_i):

$$\text{Error}_i = \hat{y}_i - y_i$$



2. Key Regression Metrics

1. **Mean Absolute Error (MAE)**
   - Average of absolute differences between predicted and actual values.
   - Formula:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

   - Advantages: Easy to interpret, same units as the target variable.
2. **Mean Squared Error (MSE)**
   - Average of **squared differences** between predicted and actual values.
   - Formula:

$$\text{MSE} = \frac{1}{n-p}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

   - Penalizes larger errors more heavily.
3. **Root Mean Squared Error (RMSE)**
   - Square root of MSE.
   - Formula:

$$\text{RMSE} = \sqrt{\text{MSE}}$$

   - Popular because it has the **same units** as the target variable, making it easy to interpret.
4. **R-squared (R2R^2R2)**
   - Proportion of variance in the target variable explained by the model.

- Formula:

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

$$R^2 = 1 - \frac{\text{Unexplained Variance}}{\text{Total Variance}}$$

- Range:
  - 1 → Perfect fit
  - 0 → Model explains nothing beyond the mean
  - <0 → Model performs worse than predicting the mean
- Note: Assumes a **linear relationship**; can be misleading for nonlinear models.

---

3. Explained Variance

3. Explained Variance

- Measures how much of the **total variability in the target** is captured by the model.
- Perfect predictor: Explained variance = Total variance, Unexplained variance = 0 → $R^2 = 1$
- Simplistic predictor (predicts mean for all points): Explained variance = 0 → $R^2 = 0$

Explained variance and R^2

$$\text{Explained Variance} = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2$$

$$\text{Unexplained Variance} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \text{n} * \text{MSE}$$

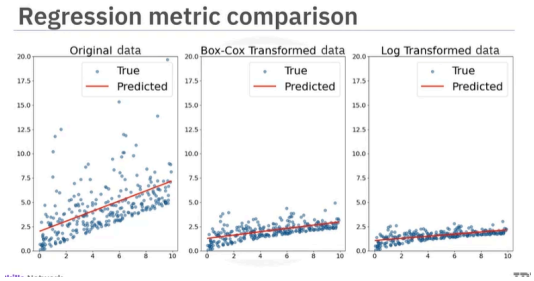
$$\text{Total Variance} = \sum_{i=1}^{n}(y_i - \bar{y})^2 = \sigma^2$$

$$R^2 = \frac{\sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} = 1 - \frac{\text{Unexplained Variance}}{\text{Total Variance}}$$

4. Model Transformation Examples

- For skewed data (e.g., log-normal distribution):
  - Transformations like **Box-Cox** or **log transformation** can improve linear model fit.
  - Metrics after transformation show:
    - Higher R2R^2R2
    - Lower MAE, MSE, RMSE

- Visual inspection confirms better alignment with the trend line.



5. Key Takeaways

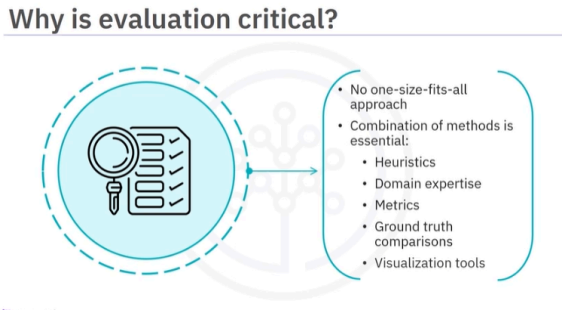
- **Error metrics (MAE, MSE, RMSE)** quantify the magnitude of prediction errors.
- **R-squared** indicates how well variance in the target is explained by the model.
- **Visualizing predictions vs actuals** is important for understanding model fit.
- Transformations can significantly improve regression performance, especially with skewed data.

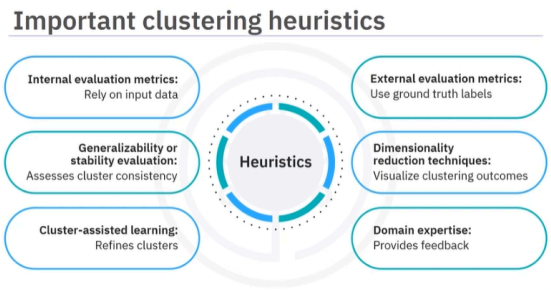
Evaluating Unsupervised Learning Models: Heuristics and Techniques

1. Challenges in Evaluating Unsupervised Learning

- **No predefined labels**: Unlike supervised learning, there's no ground truth to directly compare predictions.
- **Subjectivity**: Evaluating patterns or clusters often requires domain expertise and careful interpretation.
- **Stability**: Ensures consistent results across different subsets or perturbations of the data.

2. Evaluation Approaches



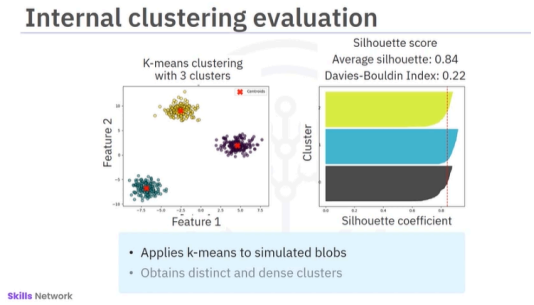


1. **Internal Metrics** – Use only the input data to assess clustering quality:

   - **Silhouette Score**:

   Measures cohesion vs separation, ranges -1 to 1; higher = better-defined clusters.

   - **Davies-Bouldin Index**: Ratio of intra-cluster compactness to inter-cluster separation; lower = better clusters.
   - **Inertia (K-means)**: Sum of squared distances within clusters; lower = more compact clusters, but tradeoff exists with more clusters.



2. **External Metrics** – Require ground truth labels (if available):

   - **Adjusted Rand Index (ARI)**: Compares clustering to true labels, -1 to 1; 1 = perfect alignment.
   - **Normalized Mutual Information (NMI)**: Measures shared information between predicted and true labels; 0 to 1.
   - **Fowlkes-Mallows Index (FMI)**: Geometric mean of precision and recall between cluster and label assignments; higher = better.

3. **Generalizability / Stability**

   - Evaluate cluster consistency across variations of the dataset.
   - Stable models produce similar clusters under small perturbations.

4. **Dimensionality Reduction Evaluation**

   - **Explained Variance Ratio (PCA)**: Amount of variance captured by components.
   - **Reconstruction Error**: How accurately original data is reconstructed from reduced representation; lower = better.

- **Neighborhood Preservation**: How well high-dimensional relationships are maintained in lower dimensions (important for t-SNE, UMAP).

5. **Visualization**
   - Scatter plots, dendrograms, and projections (PCA, t-SNE, UMAP) are essential to interpret results.
   - Example: PCA on the Iris dataset shows PC1 and PC2 effectively separating species clusters.

---

3. Practical Examples

- **K-means on Simulated Blobs**:
  - Distinct, dense clusters: Silhouette = 0.84, Davies-Bouldin = 0.22 → excellent clustering.
  - Somewhat dispersed clusters: Silhouette = 0.58, Davies-Bouldin = 0.6 → reasonable clustering.
- **Dimensionality Reduction**:
  - First two principal components capture most variance in the Iris dataset, allowing visualization in 2D.

---

4. Key Takeaways

- Unsupervised evaluation relies on **a combination of metrics, heuristics, domain expertise, and visualization**.
- **Internal metrics** assess cluster compactness and separation.
- **External metrics** compare clusters to true labels if available.
- **Dimensionality reduction** evaluation ensures important information is preserved.
- Stability and consistency are critical to trust the results of unsupervised models.

Cross-Validation and Advanced Model Validation Techniques

1. What is Model Validation?

- Goal: Make sure your model **works well on unseen data** and isn't just memorizing the training data (avoiding overfitting).
- Overfitting happens when a model performs well on the training data but poorly on new data.
- **Data snooping / leakage**: Testing your model on the test set while tuning it is **cheating**, because the model indirectly "sees" the test data. This gives overly optimistic results.

---

2. How to properly validate a model

- Split your data into **three parts**:

1. **Training set:** Used to train the model and adjust hyperparameters.
2. **Validation set:** Used to evaluate different hyperparameter choices **during training**.
3. **Test set: Held out** until the very end to check the final model's performance.

---

3. Cross-Validation

- Standard train-validation split can be biased if you happen to pick a bad validation set.
- **K-Fold Cross-Validation** helps:
  1. Split your training data into **K folds** (e.g., 5 or 10).
  2. For each fold:
     - Train on the other K−1K-1K−1 folds
     - Validate on the current fold
  3. Repeat for all folds and **average the results**.
- Benefits:
  - Every data point gets used for both training and validation.
  - Reduces overfitting to a specific validation set.
  - Gives a **more reliable estimate** of generalization performance.

---

4. Stratified Cross-Validation

- Useful when your data is **imbalanced** (e.g., more examples of one class than another).
- Ensures **class proportions are preserved** in each fold.
- Prevents biased evaluation.

---

5. Handling Skewed Targets (Regression)

- Some regression models assume a **normally distributed target**.
- If the target is skewed, use **transformations** like:
  - **Log transform**
  - **Box-Cox transform**
- These reduce skewness, helping the model fit better.

---

6. Summary

- **Model validation prevents overfitting**.
- **Data snooping** = testing too early → avoid it.

- **Validation strategy**: training set + validation set + test set.
- **K-Fold cross-validation**: more robust, uses all data points for training and validation.
- **Stratified CV**: preserves class distributions for classification problems.
- **Target transformations**: handle skewed data for better regression results.

Regularization in Regression and Classification

Regularization is used to **prevent overfitting** by adding a penalty to the size of the model's coefficients.

- constrains the model during training, discouraging it from overfitting to the training data.

It modifies the cost function (formula That tells how wrong its prediction is):

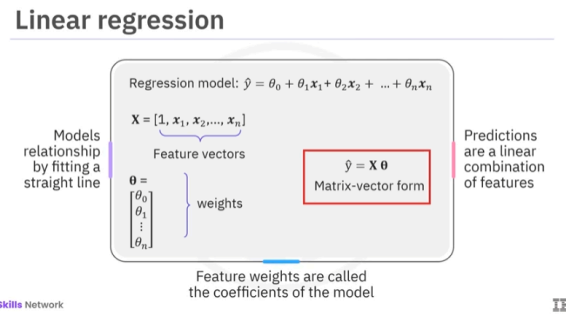
$$\text{Cost} = \text{MSE} + \lambda \times \text{Penalty}$$

where lambda: regularization hyperparameter

- Lambda is literally the knob that controls how much you punish large weights

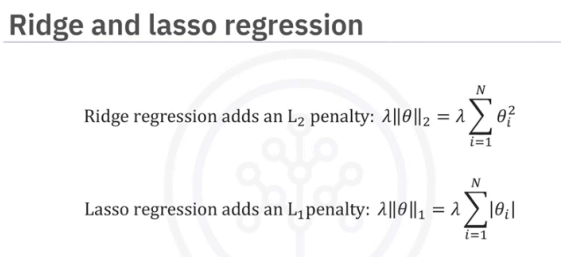
penalty: Ridge, Lasso and other methods

Two main types:



Ridge Regression (L2 penalty)

- Penalty = sum of squares of coefficients
- Shrinks coefficients **but never to zero**
- Useful when **all features are important**
- Good in noisy datasets



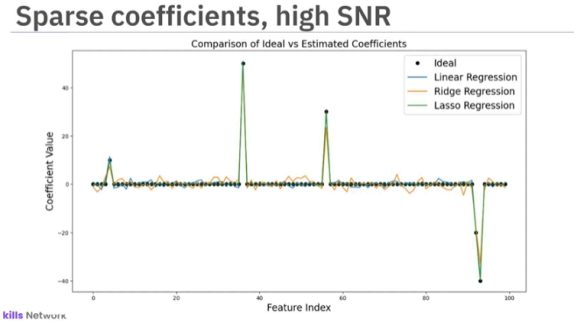
Lasso Regression (L1 penalty)

- Penalty = sum of absolute values
- Shrinks some coefficients **exactly to zero**

  ➡️ Acts as **feature selection**
- Best when **only a few features matter (sparse)**
- Handles noise better than linear regression

Linear Regression (no penalty)

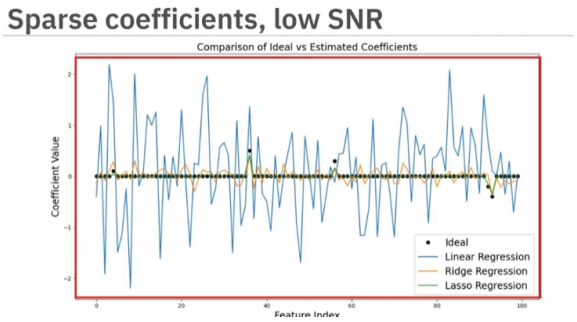
- Performs well only when **data is clean (high SNR)**
- Fails badly when data is noisy
- Very sensitive to outliers



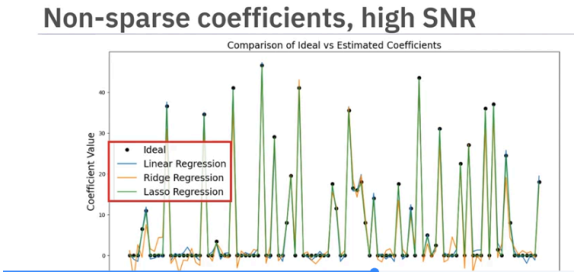
Understanding SNR (Signal-to-Noise Ratio)

- **High SNR** → data is clean (less noise)
- **Low SNR** → data is noisy and messy

In low SNR:



- Linear regression performs **very poorly**
- Ridge and Lasso are much better



🎯 Main Takeaways

- Regularization helps control complexity and reduce overfitting.
- Lasso is best for sparse data and selecting features.
- Ridge is best when all features matter but need control.
- Linear regression only works well with clean, low-noise data.

Data Leakage and Other Pitfalls

1. What is Data Leakage?

Data leakage happens when your model accidentally gets **information during training that it will not have in real life** after deployment.

Example:

You create a feature using the *average house price of the entire dataset*.

But in real life, when predicting a new house, you won't know this global average from the future data → so the model performs unrealistically well.

This makes training/test accuracy look high, but in real-world use, the model fails.

---

2. Why is Leakage Dangerous?

- Your model **cheats** by learning from information it shouldn't know.
- Test set also contains leaked info → test results look good.
- When deployed, it performs badly because this leaked info won't exist.

---

3. What Causes Data Leakage?

- Using **future information** to predict the present (e.g., using tomorrow's stock price).
- Feature engineering using the **entire dataset** (e.g., global averages).
- Poor train-test splitting (mixing test info into training).
- Incorrect cross-validation where preprocessing is done before splitting.

---

4. How to Avoid Leakage?

- Keep strict separation between **train**, **validation**, and **test** sets.
- Avoid using features that depend on **future data**.
- In pipelines, ensure preprocessing happens **inside** the CV loop for every fold.
- For time-series data, use **TimeSeriesSplit** (not random splitting).

---

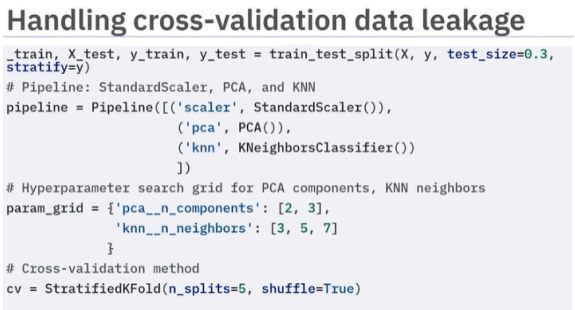
5. Example Pipeline

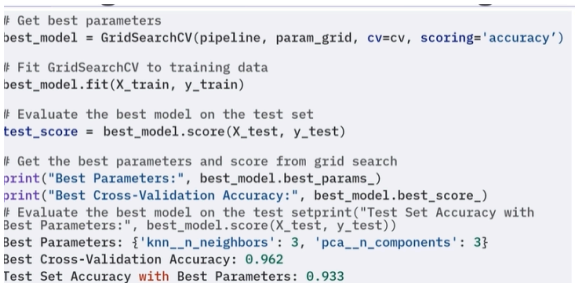
A correct pipeline:

- Split data → define pipeline (scaler → PCA → KNN) → grid search → fit on train folds only → evaluate on held-out test set.

Key point:

Pipeline is inside GridSearchCV → ensures no leakage across folds.





```
# Get best parameters
best_model = GridSearchCV(pipeline, param_grid, cv=cv, scoring='accuracy')

# Fit GridSearchCV to training data
best_model.fit(X_train, y_train)

# Evaluate the best model on the test set
test_score = best_model.score(X_test, y_test)

# Get the best parameters and score from grid search
print("Best Parameters:", best_model.best_params_)
print("Best Cross-Validation Accuracy:", best_model.best_score_)
# Evaluate the best model on the test setprint("Test Set Accuracy with
Best Parameters:", best_model.score(X_test, y_test))
Best Parameters: {'knn__n_neighbors': 3, 'pca__n_components': 3}
Best Cross-Validation Accuracy: 0.962
Test Set Accuracy with Best Parameters: 0.933
```

6. Time-Series Data

For time-dependent data:

- You cannot randomly shuffle.
- Must split sequentially (train → older data, test → newer data).
- Use **TimeSeriesSplit** for cross-validation.

7. Feature Importance Pitfalls

Understanding feature importance is tricky. Problems include:

- **Redundant features share importance** → each looks less important.
- **Feature scale** affects models like linear regression.
- Importance ≠ causation → important feature does NOT mean it *causes* the outcome.
- Some models ignore **interactions** between features.

Example:

Two features individually weak, but their product is very powerful → linear regression fails, random forest succeeds.

8. Other Modeling Pitfalls

- Using raw data without cleaning/feature engineering.
- Wrong evaluation metric.
- Ignoring class imbalance.
- Blind trust in AutoML tools.
- Doing “what-if” predictions when the model has **no causal features** → results meaningless.

📌 Short Summary (Very Simple)

- **Data leakage** = model sees info during training that it won't see in real life.
- Leakage → model looks great in training/test but fails in production.
- Avoid leakage by:

    - Proper train/validation/test split
    - Avoid using future data
    - Use pipelines inside cross-validation
    - Use **TimeSeriesSplit** for time-based data
- Feature importance pitfalls:
    - Redundant features, scaling issues, correlation ≠ causation, ignored interactions.
- Modeling pitfalls:
    - Wrong metrics, ignoring imbalance, using raw data, trusting AutoML blindly.

Model evaluation metrics and methods

Method Name	Description	Code Syntax
classification_report	Generates a report with precision, recall, F1-score, and support for each class in classification problems. Useful for model evaluation.**Hyperparameters**:target_names: List of labels to include in the report.**Pros**: Provides a comprehensive evaluation of classification models.**Limitations**: May not provide enough insight for imbalanced datasets.	1. from sklearn.metrics import classification_report 2. # y_true: True labels 3. # y_pred: Predicted labels 4. # target_names: List of target class names 5. report = classification_report(y_true, y_pred, target_names=["class1", "class2"])Copied!Wrap Toggled!
confusion_matrix	Computes a confusion matrix to evaluate the classification performance, showing counts of true positives, false positives, true negatives, and false negatives.**Hyperparameters**:labels: List of class labels to include.**Pros**: Essential for understanding classification errors.**Limitations**: Doesn't give insights into prediction probabilities.	1. from sklearn.metrics import confusion_matrix 2. # y_true: True labels 3. # y_pred: Predicted labels 4. conf_matrix = confusion_matrix(y_true, y_pred)Copied!Wrap Toggled!
mean_squared_error	Calculates the mean squared error (MSE), a common metric for regression models. Lower values indicate better performance.**Hyperparameters**:sample_weight: Weights to apply to each sample.**Pros**: Simple and widely used metric.**Limitations**: Sensitive to outliers, as large errors are squared.	1. from sklearn.metrics import mean_squared_error 2. # y_true: True values 3. # y_pred: Predicted values 4. # sample_weight: Optional, array of sample weights 5. mse = mean_squared_error(y_true, y_pred)Copied!Wrap Toggled!
root_mean_squared_error	Calculates the root mean squared error (RMSE), which is the square root of the MSE. RMSE gives more interpretable results as it is in the same units as the target.**Hyperparameters**:sample_weight: Weights to apply to each sample.**Pros**: More	1. from sklearn.metrics import root_mean_squared_error 2. # y_true: True values 3. # y_pred: Predicted values 4. # sample_weight: Optional, array of sample weights 5. rmse =

Method Name	Description	Code Syntax
	interpretable than MSE.**Limitations**: Like MSE, it can be sensitive to large errors and outliers.	root_mean_squared_error(y_true, y_pred)Copied!Wrap Toggled!
mean_absolute_error	Measures the average magnitude of errors in predictions, without considering their direction. Useful for understanding the average error size.**Hyperparameters**:sample_weight: Optional sample weights.**Pros**: Less sensitive to outliers compared to MSE.**Limitations**: Does not penalize large errors as much as MSE or RMSE.	1. from sklearn.metrics import mean_absolute_error 2. # y_true: True values 3. # y_pred: Predicted values 4. mae = mean_absolute_error(y_true, y_pred)Copied!Wrap Toggled!
r2_score	Computes the coefficient of determination (R^2), which represents the proportion of variance explained by the model. A higher value indicates a better fit.**Pros**: Provides a clear indication of model performance.**Limitations**: Doesn't always represent model quality, especially for non-linear models.	1. from sklearn.metrics import r2_score 2. # y_true: True values 3. # y_pred: Predicted values 4. r2 = r2_score(y_true, y_pred)Copied!Wrap Toggled!
silhouette_score	Measures the quality of clustering by assessing the cohesion within clusters and separation between clusters. Higher scores indicate better clustering.**Hyperparameters**:metric: Distance metric to use.**Pros**: Useful for validating clustering performance.**Limitations**: Sensitive to outliers and choice of distance metric.	1. from sklearn.metrics import silhouette_score 2. # X: Data used in clustering 3. # labels: Cluster labels for each sample 4. score = silhouette_score(X, labels, metric='euclidean')Copied!Wrap Toggled!
silhouette_samples	Provides silhouette scores for each individual sample, indicating how well it fits its assigned cluster.**Hyperparameters**:metric: Distance metric to use.**Pros**: Offers granular insight into each sample's clustering quality.**Limitations**: Same as silhouette_score; sensitive to outliers and distance metric.	1. from sklearn.metrics import silhouette_samples 2. # X: Data used in clustering 3. # labels: Cluster labels for each sample 4. samples = silhouette_samples(X, labels, metric='euclidean')Copied!Wrap Toggled!
davies_bouldin_score	Measures the average similarity ratio of each cluster with the most similar cluster. Lower values indicate better clustering.**Pros**: Provides a simple, effective clustering evaluation.**Limitations**: May not work well with highly imbalanced clusters.	1. from sklearn.metrics import davies_bouldin_score 2. # X: Data used in clustering 3. # labels: Cluster labels for each sample 4. db_score = davies_bouldin_score(X, labels)Copied!Wrap Toggled!
Voronoi	Computes the Voronoi diagram, which partitions space based on the nearest neighbor.**Pros**: Useful for spatial analysis and clustering.**Limitations**: Limited to use cases that involve spatial partitioning of data.	1. from scipy.spatial import Voronoi 2. # points: Coordinates for Voronoi diagram 3. vor =

Method Name	Description	Code Syntax
		Voronoi(points)Copied!Wrap Toggled!
voronoi_plot_2d	Plots the Voronoi diagram in 2D for visualizing clustering results.**Hyperparameters**:show_vertices: Whether to display the vertices.**Pros**: Great for visualizing spatial clustering.**Limitations**: Limited to 2D spaces and large datasets may cause performance issues.	1. from scipy.spatial import voronoi_plot_2d 2. # vor: Voronoi diagram object 3. voronoi_plot_2d(vor, show_vertices=True)Copied!Wrap Toggled!
matplotlib.patches.Patch	Creates custom shapes such as rectangles, circles, or ellipses for adding to plots.**Hyperparameters**:color: Fills color of the shape.**Pros**: Versatile for visual customization.**Limitations**: May not support all shapes or complex customizations.	1. import matplotlib.patches as patches 2. # Create a rectangle with specified width, height, and position 3. rectangle = patches.Rectangle((0, 0), 1, 1, color='blue')Copied!Wrap Toggled!
explained_variance_score	Measures the proportion of variance explained by the model's predictions. A higher score indicates better performance.**Pros**: Helps in assessing the fit of regression models.**Limitations**: Not suitable for classification tasks.	1. from sklearn.metrics import explained_variance_score 2. # y_true: True values 3. # y_pred: Predicted values 4. ev_score = explained_variance_score(y_true, y_pred)Copied!Wrap Toggled!
Ridge regression	Performs ridge regression (L2 regularization) to avoid overfitting by penalizing large coefficients.**Hyperparameters**:alpha: Regularization strength.**Pros**: Helps reduce overfitting in regression models.**Limitations**: May not work well with sparse data.	1. from sklearn.linear_model import Ridge 2. # alpha: Regularization strength (larger values indicate stronger regularization) 3. ridge = Ridge(alpha=1.0)Copied!Wrap Toggled!
Lasso regression	Performs lasso regression (L1 regularization), which encourages sparsity by penalizing the absolute value of coefficients.**Hyperparameters**:alpha: Regularization strength.**Pros**: Encourages sparse solutions, useful for feature selection.**Limitations**: May struggle with multicollinearity.	1. from sklearn.linear_model import Lasso 2. # alpha: Regularization strength (larger values indicate stronger regularization) 3. lasso = Lasso(alpha=0.1)Copied!Wrap Toggled!
Pipeline	Chains multiple steps of preprocessing and modeling into a single object, ensuring efficient workflow.**Pros**: Simplifies code, ensures reproducibility.**Limitations**: May not work well with complex pipelines requiring dynamic configurations.	1. from sklearn.pipeline import Pipeline 2. # steps: List of tuples with name and estimator/transformer 3. pipeline = Pipeline(steps=[('scaler', StandardScaler()), ('model',

Method Name	Description	Code Syntax
		Ridge(alpha=1.0))])Copied!Wrap Toggled!
GridSearchCV	Performs exhaustive search over a specified parameter grid to find the best model configuration.**Hyperparameters**:param_grid: Dictionary of parameter grids.**Pros**: Ensures optimal model parameters.**Limitations**: Computationally expensive for large grids.	1. from sklearn.model_selection import GridSearchCV 2. # estimator: Model to be tuned 3. # param_grid: Dictionary with parameters to search over 4. grid_search = GridSearchCV(estimator=Ridge(), param_grid={'alpha': [0.1, 1.0, 10.0]})Copied!Wrap Toggled!

Visualization strategies for k-means evaluation

Process Name	Brief Description	Code Snippet
Multiple runs of k-means	Executes KMeans clustering multiple times with different random initializations to assess variability in cluster assignments.**Advantage:** Helps visualize consistency.**Limitation:** Computationally costly for large datasets.	1. # Number of runs for KMeans with different random states 2. n_runs = 4 3. inertia_values = [] 4. 5. plt.figure(figsize=(12, 12)) 6. 7. # Run K-Means multiple times with different random states 8. for i in range(n_runs): 9. kmeans = KMeans(n_clusters=4, random_state=None) # Use the default `n_init` 10. kmeans.fit(X) 11. inertia_values.append(kmeans.inertia_) 12. 13. # Plot the clustering result 14. plt.subplot(2, 2, i + 1) 15. plt.scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap='tab10', alpha=0.6, edgecolor='k') 16. plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], c='red', s=200, marker='x', label='Centroids') 17. plt.title(f'K-Means Clustering Run {i + 1}') 18. plt.xlabel('Feature 1') 19. plt.ylabel('Feature 2') 20. plt.legend() 21. 22. plt.tight_layout() 23. plt.show() 24. 25. # Print inertia values 26. for i, inertia in enumerate(inertia_values, start=1): 27. print(f'Run {i}: Inertia={inertia:.2f}')
Elbow method	Evaluates the optimal number of clusters by plotting inertia (within-cluster sum of squares) for different **k** values.**Advantage:** Easy to	1. # Range of k values to test 2. k_values = range(2, 11) 3. 4. # Store performance metrics

Process Name	Brief Description	Code Snippet
	interpret.**Limitation:** Subjective elbow point.	5. inertia_values = [] 6. for k in k_values: 7. kmeans = KMeans(n_clusters=k, random_state=42) 8. y_kmeans = kmeans.fit_predict(X) 9. 10. # Calculate and store metrics 11. inertia_values.append(kmeans.inertia_) 12. 13. # Plot the inertia values (Elbow Method) 14. plt.figure(figsize=(18, 6)) 15. plt.subplot(1, 3, 1) 16. plt.plot(k_values, inertia_values, marker='o') 17. plt.title('Elbow Method: Inertia vs. k') 18. plt.xlabel('Number of Clusters (k)') 19. plt.ylabel('Inertia')Copied!Wrap Toggled!
Silhouette method	Determines the optimal number of clusters by evaluating Silhouette Scores for different **k** values.**Advantage:** Considers both cohesion and separation.**Limitation:** High computation for large datasets.	1. # Range of k values to test 2. k_values = range(2, 11) 3. 4. # Store performance metrics 5. silhouette_scores = [] 6. for k in k_values: 7. kmeans = KMeans(n_clusters=k, random_state=42) 8. y_kmeans = kmeans.fit_predict(X) 9. silhouette_scores.append(silhouette_score(X, y_kmeans)) 10. 11. # Plot the Silhouette Scores 12. plt.figure(figsize=(18, 6)) 13. plt.subplot(1, 3, 2) 14. plt.plot(k_values, silhouette_scores, marker='o') 15. plt.title('Silhouette Score vs. k') 16. plt.xlabel('Number of Clusters (k)') 17. plt.ylabel('Silhouette Score')Copied!Wrap Toggled!
Davies-Bouldin Index	Evaluates clustering performance by calculating DBI for different **k** values.**Advantage:** Quantifies compactness and separation.**Limitation:** Sensitive to cluster shapes and density.	1. # Range of k values to test 2. k_values = range(2, 11) 3. 4. # Store performance metrics 5. davies_bouldin_indices = [] 6. for k in k_values: 7. kmeans = KMeans(n_clusters=k, random_state=42) 8. y_kmeans = kmeans.fit_predict(X) 9. davies_bouldin_indices.append(davies_bouldin_score(X, y_kmeans)) 10. 11. # Plot the Davies-Bouldin Index 12. plt.figure(figsize=(18, 6)) 13. plt.subplot(1, 3, 3) 14. plt.plot(k_values, davies_bouldin_indices, marker='o') 15. plt.title('Davies-Bouldin Index vs. k') 16. plt.xlabel('Number of Clusters (k)') 17. plt.ylabel('Davies-Bouldin Index')