import numpy as np import matplotlib.pyplot as plt from sklearn.cluster import KMeans from skimage import io # Load image image = io.imread('https://upload.wikimedia.org/wikipedia/commons/thumb/2/24/Cat03.jpg/320px-Cat03.jpg') plt.imshow(image) plt.title('Original Image') plt.axis('off') plt.show() # Reshape the image to a 2D array of pixels and 3 color values (RGB) pixels = image.reshape(-1, 3) # Apply KMeans clustering k = 4 # Number of clusters kmeans = KMeans(n_clusters=k, random_state=42) kmeans.fit(pixels) segmented_img = kmeans.cluster_centers_[kmeans.labels_].reshape(image.shape).astype(np.uint8) # Display segmented image plt.imshow(segmented_img) plt.title(f'Segmented Image with {k} colors') plt.axis('off') plt.show() convert into to algorithm

Here is a step-by-step algorithm for segmenting an image using KMeans clustering, as described by your code:

Image Segmentation Using KMeans Algorithm

Steps

1. **Import Required Libraries**
    - Prepare to use numpy, matplotlib, scikit-learn, and skimage for handling arrays, visualization, clustering, and image loading.
2. **Load the Image**
    - Read the image from the given URL (or local path) into a suitable array format. For example: `image = io.imread('URL')`.

3. **Display the Original Image**
    - Visualize the image with a plotting library for initial inspection. Remove axes to focus on content.
4. **Reshape Image Data**
    - Transform the 3D array of pixel colors (height, width, channels) into a 2D array where each row is a pixel and each column is a color channel (RGB).
        - Formula: If shape is `(h, w, 3)`, reshape to `(h*w, 3)`.
5. **Apply KMeans Clustering**
    - Choose the number of clusters $ k $ (the number of colors to segment into).
    - Fit the KMeans model to the pixel data and find cluster centers (dominant colors) and assign each pixel to a cluster.
6. **Map Pixels to Cluster Centers**
    - Replace each pixel's color with its cluster's centroid color, effectively reducing the image’s color palette to $ k $ colors.
7. **Reshape Segmented Data**
    - Convert the 2D clustered array back to the original image shape so it can be displayed properly.
8. **Display Segmented Image**
    - Use visualization to show the reduced-color, segmented version of the image for interpretation and analysis.

Pseudocode

```
Input: image_url, k

1. Import numpy, matplotlib.pyplot, sklearn.cluster.KMeans, skimage.io
2. image ← read image from image_url
3. Show image (optional)
4. pixels ← reshape image to (num_pixels, 3)
5. kmeans ← KMeans(n_clusters=k, random_state=42)
6. kmeans.fit(pixels)
7. segmented_pixels ← kmeans.cluster_centers_[kmeans.labels_]
8. segmented_img ← reshape segmented_pixels to original image shape
9. Convert segmented_img to uint8 type
10. Show segmented_img
Output: Segmented image with k distinct colors
```

Notes

- KMeans clustering is sensitive to the chosen $ k $; more clusters result in more nuanced segmentation but less compression.
- This algorithm is general: it works with any image and any reasonable number of $ k $.

This algorithm segments an image into $ k $ dominant colors using KMeans clustering on pixel RGB values, simplifying the color space and revealing basic patterns in the image.

give me the algorithm with formula

Here is the algorithm for image segmentation using KMeans clustering, explicitly including the key formulas and mathematical steps used in each stage:

Image Segmentation Using KMeans (with Formulas)

Steps and Formulas

1. **Load and Reshape Image Data**
    - Load the image into a 3D array:
    $$\text{image} \in \mathbb{R}^{h \times w \times 3}$$
    - Reshape to form a matrix of pixel color vectors:
    $$X = \text{reshape}(\text{image}, (\text{h \textbackslash times w}, 3))$$
    where each row $ x_i \in \mathbb{R}^3 $ is an RGB pixel.
2. **Initialize KMeans Clustering**
    - Choose the number of clusters $ k $.
    - Initial centroids:
    $$\mu_1, \ldots, \mu_k \in \mathbb{R}^3$$
3. **Assign Each Pixel to the Closest Cluster**
    - For each pixel vector $ x_i $:
    $$c_i = \arg\min_j \|x_i - \mu_j\|_2$$
    where $ c_i $ is the index of the centroid closest to $ x_i $, and $ | \cdot |_2 $ is the Euclidean norm.
4. **Update Cluster Centroids**
    - For each centroid $ \mu_j $:
    $$\mu_j = \frac{1}{|S_j|} \sum_{x_i \in S_j} x_i$$
    where $ S_j = \{ x_i \mid c_i = j \} $ is the set of pixels assigned to cluster $ j $.
5. **Iterate Assignment and Update Steps**
    - Repeat the pixel assignment and centroid update steps until convergence (no change in assignments or centroids).

6. **Create Segmented Image**
    - Replace each pixel value $ x_i $ by its cluster centroid $ \mu_{c_i} $:
      $$x_i^{\text{seg}} = \mu_{c_i}$$
    - Reshape the resulting matrix of pixel vectors back to the original shape $ h \times w \times 3 $.

Pseudocode (with key formulas)

```
Input: image, k

1. Convert image to matrix X of shape (n, 3), where n = h*w.
2. Randomly initialize centroids μ₁, ..., μ_k ∈ ℝ³.

3. Repeat until convergence:
   a. Assign pixels: for each x_i in X,
      c_i = argmin_j ||x_i - μ_j||₂
   b. Update centroids: for each j,
      μ_j = (1 / |S_j|) * sum_{x_i ∈ S_j} x_i

4. For each pixel x_i, set
   x_i_segmented = μ_{c_i}

5. Reshape segmented pixels to original image shape.
Output: segmented image
```

This algorithm maps every pixel to the color of its assigned cluster center, reducing the image's color palette and highlighting dominant regions. Each formula shows the mathematical operation performed during KMeans clustering and segmentation.

Step-by-Step Algorithm: PCA vs. Kernel PCA on "Moons"

Let's walk through the process of comparing **PCA** and **Kernel PCA** on a non-linear dataset (the "moons"). This guide assumes you're just starting with machine learning and data visualization.

1. Generate and Visualize the Data

- Create a dataset using the "moons" pattern (two interleaving half circles). This makes it easy to spot non-linear structure.
- Visualize the original data as a scatter plot.

Formula

- The "moons" dataset is just a collection of points X with labels y.

2. Standardize the Data

- Standardize (scale) each feature so they have mean 0 and variance 1. This helps PCA work correctly by removing feature bias.

Formula

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

where μ is mean and σ is standard deviation of each feature.

3. Apply Linear PCA

- PCA finds the directions (axes) that explain the most variance in your data ("principal components") and projects your data onto them.
- Use PCA to reduce data to 2 components (for easy visualization).

Formula

- Calculate the eigenvectors (principal components) V of the covariance matrix of X_{scaled}.
- Project data:

$$X_{\text{PCA}} = X_{\text{scaled}} \cdot V$$

4. Apply Kernel PCA (With RBF Kernel)

- Kernel PCA is like PCA, but it can "bend" the axes using a kernel function—good for non-linear structure.
- Use **RBF kernel** ("Gaussian kernel") to project non-linear patterns into a space where they look linear.

Formula

- Compute the kernel matrix with RBF:

$$K(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right)$$

- Perform PCA on this kernel matrix instead of raw data.

5. Visualize the Results

- Make three side-by-side scatter plots:
    1. The original moons data.
    2. The data projected by regular (linear) PCA.
    3. The data projected by Kernel PCA with RBF kernel.
- Compare how each method separates the two moon shapes.

Summary Table

Method	Formula/Idea	Good For
PCA	Find linear axes, project data: $X \cdot V$	Linearly separable data
Kernel PCA	Apply RBF kernel then PCA: $K(x_i, x_j)$, X_{KPCA}	Complex/nonlinear data

If you're new, try running the steps one at a time. Pause after each plot and ask: *Does the method separate the two moons better than before?* If you want more details or examples, just ask!

```
from sklearn.datasets import make_classification
from sklearn.model_selection import
train_test_split from sklearn.preprocessing import
StandardScaler from sklearn.linear_model import
LogisticRegression from sklearn.tree import
DecisionTreeClassifier from sklearn.svm import
SVC # Dataset X, y =
make_classification(n_samples=1000,
n_features=20, n_informative=15, n_redundant=5,
n_classes=3, weights=[0.5, 0.3, 0.2],
class_sep=1.5, flip_y=0.01, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.2,random_state=0) # Scale features
for models sensitive to scale scaler =
StandardScaler() X_train_scaled =
scaler.fit_transform(X_train) X_test_scaled =
scaler.transform(X_test) # Logistic Regression lr =
LogisticRegression(multi_class='multinomial',
solver='lbfgs', max_iter=1000)
lr.fit(X_train_scaled, y_train) print("Logistic
Regression Accuracy:", lr.score(X_test_scaled,
y_test))# Decision Tree tree =
DecisionTreeClassifier(max_depth=10,
random_state=42) tree.fit(X_train, y_train)
print("Decision Tree Accuracy:", tree.score(X_test,
y_test)) # SVM with different kernels svc_linear =
SVC(kernel='linear') svc_linear.fit(X_train_scaled,
y_train) print("Linear SVM Accuracy:",
svc_linear.score(X_test_scaled, y_test))svc_poly =
SVC(kernel='poly', degree=3)
svc_poly.fit(X_train_scaled, y_train)
print("Polynomial SVM Accuracy:",
```

svc_poly.score(X_test_scaled, y_test))svc_rbf = SVC(kernel='rbf') svc_rbf.fit(X_train_scaled, y_train) print("RBF SVM Accuracy:", svc_rbf.score(X_test_scaled, y_test)) give in algorithm

Here is the step-by-step algorithm for training and evaluating multiple classifiers (Logistic Regression, Decision Tree, SVM with various kernels) on a synthetic classification dataset:

Multiclass Classification Workflow

Steps

1. **Generate Synthetic Classification Dataset**
    - Create data with:
        - $n = 1000$ samples,
        - 20 features (15 informative, 5 redundant),
        - 3 classes (with weights $0.5, 0.3, 0.2$),
        - separation between classes parameter (1.5),
        - small label noise (1%).
    - Formula:
      $(X, y) = \text{make}\backslash\text{\_classification}(n\_samples, n\_features, n\_informative, n\_red$
2. **Split Into Train/Test Sets**
    - Divide data into training (80%) and test (20%) sets:
      $$(X_{\text{train}}, X_{\text{test}}, y_{\text{train}}, y_{\text{test}}) = \text{train}\backslash\text{\_test}\backslash\text{\_split}(X, y, \text{test}\backslash\text{\_size} = 0.2)$$
3. **Feature Scaling**
    - Standardize features for models sensitive to scale:
      $$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$
    - Fit scaler on training data, transform both train and test.
4. **Train and Evaluate Models**
    - **Logistic Regression:**
        - Multinomial logistic regression using L-BFGS optimizer.
        - Train on scaled features:
          $$\text{Model: } \hat{y} = \arg\max_k \frac{\exp(\beta_k^T x)}{\sum_j \exp(\beta_j^T x)}$$
        - Evaluate accuracy:

$$\text{Accuracy} = \frac{\text{number of correct predictions}}{\text{total predictions}}$$

- **Decision Tree:**
    - Train a decision tree (max depth 10) on unscaled features.
    - Splits data via criterion (e.g. Gini, entropy).
    - Use tree structure to classify test data and compute accuracy.
- **Support Vector Machine (SVM):**
    - Train SVMs with different kernels on scaled features:
        - Linear kernel: $K(x_i, x_j) = x_i^T x_j$
        - Polynomial kernel: $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d$ (degree $d = 3$)
        - RBF kernel: $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$
    - Fit each on training data, predict test labels, and report accuracy.

Pseudocode

```
Input: Parameters for dataset, models

1. Generate dataset:
   X, y ← make_classification(parameters)
2. Split dataset:
   X_train, X_test, y_train, y_test ← train_test_split(X, y)
3. Fit scaler on X_train, transform both X_train and X_test.
4. Train models:
   a. Logistic Regression on scaled features
   b. Decision Tree on unscaled features
   c. SVM (linear, poly, rbf) on scaled features
5. For each model:
   - Fit on training data.
   - Predict and calculate accuracy on test data.
Output: Accuracy scores of all models
```

This workflow enables comparison of multiple classifiers on the same dataset, quantifying their performance via test accuracy for multiclass classification. Each model uses specific feature setups and kernel functions to learn data patterns.

Step 1: Importing libraries and loding the image The first step involves importing thon libraries like Numpy,Metplotlib,skimage.io ,sklearn.cluster.kmeans . The input image is the loadedintoskimage.io.imread() and stored as a 3D Numpy array of shape wher the 3Drepresents theRGB color channels. Step2: Reshaping the image data: ▯ The 3D image array is reshaped into 2D array of shape ▯ Each row in this reshaped array coressponds to a single pixel and the columns reprssents its red, green and blue values. Step 3: Defining the number of clusters(K) The number of desired color segment is chosen .This determins how many distinct color clusters will be formed. Step 4: Clustering with k-means ▯ The k-mean algorithm is initialized with n clusters= k ▯ Initial cluster centroid are selected ▯ Each pixel is assigned to the nearest centroid based on euclidean distance in RGBspace ▯ Then the centroin is updated as the mean of all points assigned to each other ▯ This iterative process is continued until convergence is achieved Step 5: Segmenting the image ▯ The final k centroid values represent the dominant colors in the image ▯ Each oringinal pixel is replaced bt the RGB values of its assigned clusters centroid▯ The result is a simplified image where the number of unique colors is reduced tok. Step 6: Reconstructing and displaying the segmented image ▯ The transformed 2D array is reshaped back into its original 3D shape. ▯ The segmented image noe contains only k unique color values

coressponding to thecluster centroids. The image is then visualized using metplotlib.pyplot.imshow().mport Required Libraries

Prepare to use numpy, matplotlib, scikit-learn, and skimage for handling arrays, visualization, clustering, and image loading.
Load the Image
Read the image from the given URL (or local path) into a suitable array format. For example: image = io.imread('URL').
Display the Original Image
Visualize the image with a plotting library for initial inspection. Remove axes to focus on content.
Reshape Image Data
Transform the 3D array of pixel colors (height, width, channels) into a 2D array where each row is a pixel and each column is a color channel (RGB).
Formula: If shape is (h, w, 3), reshape to (h*w, 3).
Apply KMeans Clustering
Choose the number of clusters kk (the number of colors to segment into).
Fit the KMeans model to the pixel data and find cluster centers (dominant colors) and assign each pixel to a cluster.
Map Pixels to Cluster Centers
Replace each pixel's color with its cluster's centroid color, effectively reducing the image’s color palette to kk colors.
Reshape Segmented Data
Convert the 2D clustered array back to the original image shape so it can be displayed properly.
Display Segmented Image
Use visualization to show the reduced-color, segmented version of the image for interpretation and analysis. give me precise answer with formula and import statements

Here is a precise algorithm for image segmentation using KMeans clustering, including the required import statements and key formulas:

Image Segmentation Using KMeans Algorithm

Import Statements

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io
from sklearn.cluster import KMeans
```

Algorithm Steps with Formulas

1. **Load the Image**
    - Read the image as a 3D NumPy array of shape $(h, w, 3)$, where h is height, w is width, and 3 corresponds to RGB channels.

$$\text{image} = \text{io.imread}(\text{image\\_url})$$

2. **Reshape the Image Data**
    - Reshape the image into a 2D array of shape $(h \times w, 3)$, where each row represents a pixel's RGB values.

$$X = \text{reshape}(\text{image}, (h \times w, 3))$$

3. **Define Number of Clusters k**
    - Choose the number of distinct color clusters k for segmentation.
4. **Apply KMeans Clustering**
    - Initialize KMeans with k clusters:

$$\text{kmeans} = \text{KMeans}(n\_clusters = k)$$

    - Fit KMeans to pixel data X:

$$\text{kmeans.fit}(X)$$

    - Assign each pixel to the nearest centroid c_i using Euclidean distance:

$$c_i = \arg\min_j \|x_i - \mu_j\|_2$$

    - Update centroids as the mean of the pixels assigned to each cluster:

$$\mu_j = \frac{1}{|S_j|} \sum_{x_i \in S_j} x_i$$

5. **Segment the Image**
    - Replace each pixel color with its cluster centroid color:

$$X_{\text{seg}}[i] = \mu_{c_i}$$

6. **Reconstruct and Display Segmented Image**
    - Reshape X_{seg} back to original image shape $(h, w, 3)$.

$$\text{segmented\\_image} = \text{reshape}(X_{\text{seg}}, (h, w, 3))$$

    - Display using Matplotlib:

```
plt.imshow(segmented_image)
plt.axis('off')
plt.show()
```

This approach reduces the number of unique colors in the image from potentially millions to k, producing a segmented simplified image by grouping similar colors and replacing them with cluster centers. Each step highlights the transformation performed on the original image data through KMeans clustering.

This completes an exact, formula-supported, and code-ready summary for KMeans image segmentation.

⁂

import matplotlib.pyplot as plt from sklearn.datasets import make_moons from sklearn.decomposition import PCA, KernelPCA from sklearn.preprocessing import StandardScaler # Generate synthetic non-linear dataset (two moons) X, y = make_moons(n_samples=300, noise=0.05, random_state=42) # Standardize data scaler = StandardScaler() X_scaled = scaler.fit_transform(X) # Apply PCA pca = PCA(n_components=2) X_pca = pca.fit_transform(X_scaled) # Apply Kernel PCA with RBF kernel kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15) X_kpca = kpca.fit_transform(X_scaled) # Plot Original Data plt.subplot(1, 3, 1) plt.scatter(X[:,0], X[:,1], c=y, cmap='viridis') plt.title("Original Data") # Plot PCA plt.subplot(1, 3, 2) plt.scatter(X_pca[:,0], X_pca[:,1], c=y, cmap='viridis') plt.title("PCA Projection") # Plot Kernel PCA plt.subplot(1, 3, 3) plt.scatter(X_kpca[:,0], X_kpca[:,1], c=y, cmap='viridis') plt.title("Kernel PCA (RBF) Projection") plt.show() give me answer like the previous one

Here is a precise algorithm with import statements and key formulas for applying PCA and Kernel PCA on a non-linear dataset like the two moons, similar in style to the previous answer:

Dimensionality Reduction with PCA and Kernel PCA

Import Statements

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA, KernelPCA
```

Algorithm Steps with Formulas

1. **Generate Non-linear Dataset**
    - Generate two interleaving half moons dataset with noise:
    $$(X, y) = \text{make\textbackslash\_moons}(n\_samples = 300, \text{noise} = 0.05)$$
2. **Standardize the Data**
    - Center and scale features to zero mean and unit variance:
    $$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$
    where μ, σ are mean and standard deviation vectors for each feature.
3. **Apply Linear PCA**
    - Compute covariance matrix:
    $$C = \frac{1}{n} X_{\text{scaled}}^T X_{\text{scaled}}$$
    - Find eigenvalues and eigenvectors of C.
    - Project data onto the top 2 principal components:
    $$X_{\text{PCA}} = X_{\text{scaled}} V_2$$
    where V_2 are eigenvectors corresponding to the top two eigenvalues.
4. **Apply Kernel PCA with RBF Kernel**
    - Define RBF kernel function between points x_i, x_j:
    $$K(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right)$$
    - Compute kernel matrix K.
    - Center kernel matrix.
    - Perform eigen-decomposition on K.
    - Project data onto the top 2 eigenvectors of K to get:
    $$X_{\text{KPCA}} = \text{KernelPCA}(X_{\text{scaled}}, n\_components = 2, \text{kernel} =' rbf', \gamma = 15)$$
5. **Visualize**
    - Plot three scatter plots side by side:
        - Original data X colored by labels y.

- Linear PCA projection X_{PCA}.
- Kernel PCA projection X_{KPCA}.

Summary Pseudocode

```
1. Import required libraries.
2. Generate moon-shaped dataset: X, y = make_moons(...)
3. Standardize features: X_scaled = StandardScaler().fit_transform(X)
4. Fit linear PCA: X_pca = PCA(n_components=2).fit_transform(X_scaled)
5. Fit Kernel PCA with RBF kernel: X_kpca = KernelPCA(n_components=2, kernel='rbf', gamma
6. Plot:
   - Original data (X)
   - PCA projection (X_pca)
   - Kernel PCA projection (X_kpca)
```

Linear PCA projects the data onto linear subspaces capturing variance but cannot separate non-linear structures well. Kernel PCA uses a kernel trick (here, RBF) to map data into a higher-dimensional space where nonlinear patterns become linearly separable, enabling better separation of complex clusters like the two moons shape. [11] [12] [13]

⁂

```
from sklearn.datasets import make_classification
from sklearn.model_selection import
train_test_split from sklearn.preprocessing import
StandardScaler from sklearn.linear_model import
LogisticRegression from sklearn.tree import
DecisionTreeClassifier from sklearn.svm import
SVC # Dataset X, y =
make_classification(n_samples=1000,
n_features=20, n_informative=15, n_redundant=5,
n_classes=3, weights=[0.5, 0.3, 0.2],
class_sep=1.5, flip_y=0.01, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.2,random_state=0) # Scale features
for models sensitive to scale scaler =
StandardScaler() X_train_scaled =
scaler.fit_transform(X_train) X_test_scaled =
scaler.transform(X_test) # Logistic Regression lr =
LogisticRegression(multi_class='multinomial',
solver='lbfgs', max_iter=1000)
lr.fit(X_train_scaled, y_train) print("Logistic
Regression Accuracy:", lr.score(X_test_scaled,
y_test))# Decision Tree tree =
DecisionTreeClassifier(max_depth=10,
random_state=42) tree.fit(X_train, y_train)
print("Decision Tree Accuracy:", tree.score(X_test,
y_test)) # SVM with different kernels svc_linear =
SVC(kernel='linear') svc_linear.fit(X_train_scaled,
y_train) print("Linear SVM Accuracy:",
svc_linear.score(X_test_scaled, y_test))svc_poly =
SVC(kernel='poly', degree=3)
svc_poly.fit(X_train_scaled, y_train)
print("Polynomial SVM Accuracy:",
```

svc_poly.score(X_test_scaled, y_test))svc_rbf = SVC(kernel='rbf') svc_rbf.fit(X_train_scaled, y_train) print("RBF SVM Accuracy:", svc_rbf.score(X_test_scaled, y_test)) give me the answer like the last one

Here is a clear, step-by-step algorithm for classification using Logistic Regression, Decision Tree, and SVM with different kernels, including key steps and formulas:

Multiclass Classification Algorithm

Import Statements

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
```

Steps with Explanation and Formulas

1. **Generate Synthetic Dataset**
    - Create dataset with $n = 1000$ samples, 20 features (15 informative, 5 redundant), 3 classes, with class weights, separation, and noise:

    $$(X, y) = \mathrm{make\backslash\_classification}(n\_samples = 1000, n\_features = 20, n\_informati$$

2. **Split Dataset**
    - Split into train and test sets (80%-20%):

    $$(X_{\mathrm{train}}, X_{\mathrm{test}}, y_{\mathrm{train}}, y_{\mathrm{test}}) = \mathrm{train\backslash\_test\backslash\_split}(X, y, \mathrm{test\backslash\_size} = 0.2)$$

3. **Feature Scaling**
    - Scale features for scale-sensitive models (Logistic Regression, SVM):

    $$X_{\mathrm{train\backslash\_scaled}} = \frac{X_{\mathrm{train}} - \mu}{\sigma}, \quad X_{\mathrm{test\backslash\_scaled}} = \frac{X_{\mathrm{test}} - \mu}{\sigma}$$

    where μ, σ are mean and standard deviation computed on training data.

4. **Train and Evaluate Models**
    - **Logistic Regression (Multinomial)**
      Learn weights β for each class k minimizing cross-entropy loss:

      $$P(y = k \mid x) = \frac{e^{\beta_k^T x}}{\sum_j e^{\beta_j^T x}}$$

      Fit on $X_{\mathrm{train\backslash\_scaled}}, y_{\mathrm{train}}$, then compute accuracy on $X_{\mathrm{test\backslash\_scaled}}, y_{\mathrm{test}}$.

- **Decision Tree Classifier**
  Build tree with splits on features optimizing impurity (e.g., Gini) to max depth 10 using unscaled data. Predict test labels and compute accuracy.
- **Support Vector Machine (SVM)**
  Train three SVM models using $X_{\text{train\_scaled}}, y_{\text{train}}$:
  - Linear kernel:
    $$K(x_i, x_j) = x_i^T x_j$$
  - Polynomial kernel degree 3:
    $$K(x_i, x_j) = (x_i^T x_j + 1)^3$$
  - RBF kernel:
    $$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$$

  Predict test labels and compute accuracy for each kernel on scaled test data.

Summary Pseudocode

```
1. Import libraries
2. Generate dataset (X, y)
3. Split into train and test sets
4. Fit scaler on training data; scale train and test features
5. Train models:
   - Logistic Regression on scaled data
   - Decision Tree on unscaled data
   - SVM with linear, poly, and RBF kernels on scaled data
6. Evaluate and print accuracies on respective test data
```

This approach allows comparison of multiple classifiers on the same dataset, with scaling applied only where needed, and accurately reports the test performance of each model type and kernel. [21] [22]

⁂

```
import numpy as np import matplotlib.pyplot as
plt from sklearn.datasets import
make_classification from sklearn.model_selection
import train_test_split from sklearn.preprocessing
import StandardScaler, label_binarize from
sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC from sklearn.metrics
import ( accuracy_score, precision_score,
recall_score, f1_score,roc_auc_score,
average_precision_score,
roc_curve,precision_recall_curve ) # Dataset X, y =
make_classification( n_samples=3000,
n_features=50, n_informative=15,
n_redundant=10, n_classes=3, weights=[0.6, 0.3,
0.1], flip_y=0.02,random_state=42 ) X_train, X_test,
y_train, y_test = train_test_split(X,y,stratify=y,
test_size=0.2, random_state=0) # Classifiers
classifiers = { "Logistic Regression":
LogisticRegression(max_iter=2000,multi_class='
multinomial'), "Decision Tree":
DecisionTreeClassifier(max_depth=10,random_sta
te=42), "SVM": SVC(kernel='rbf', probability=True)
} # Evaluation for name, clf in classifiers.items(): if
name in ["Logistic Regression", "SVM"]: #
scalefeatures scaler = StandardScaler()
X_train_scaled =
scaler.fit_transform(X_train)X_test_scaled =
scaler.transform(X_test) clf.fit(X_train_scaled,
y_train) y_pred = clf.predict(X_test_scaled)
y_proba = clf.predict_proba(X_test_scaled) else:
clf.fit(X_train, y_train) y_pred = clf.predict(X_test)
```

```
y_proba = clf.predict_proba(X_test) acc =
accuracy_score(y_test, y_pred) prec =
precision_score(y_test, y_pred,
average='macro')rec = recall_score(y_test, y_pred,
average='macro')f1 = f1_score(y_test, y_pred,
average='macro') y_bin = label_binarize(y_test,
classes=[0,1,2]) roc_auc = roc_auc_score(y_bin,
y_proba, average='macro',multi_class='ovr')
pr_auc = average_precision_score(y_bin, y_proba,
average='macro') print(f"\n{name}")
print(f"Accuracy: {acc:.3f}") print(f"Precision
(macro): {prec:.3f}") print(f"Recall (macro):
{rec:.3f}") print(f"F1-score (macro): {f1:.3f}")
print(f"ROC-AUC (macro): {roc_auc:.3f}")
print(f"PR-AUC (macro): {pr_auc:.3f}") # ROC & PR
for minority class (class 2) with SVM y_bin =
label_binarize(y_test, classes=[0,1,2]) clf =
SVC(kernel='rbf', probability=True) X_train_scaled
=
StandardScaler().fit_transform(X_train)X_test_sca
led =
StandardScaler().fit_transform(X_test)clf.fit(X_trai
n_scaled, y_train) y_proba =
clf.predict_proba(X_test_scaled) fpr, tpr, _ =
roc_curve(y_bin[:,2], y_proba[:,2]) prec, rec, _ =
precision_recall_curve(y_bin[:,2], y_proba[:,2])
plt.figure(figsize=(12,5)) plt.subplot(1,2,1)
plt.plot(fpr, tpr, label='ROC Curve (class 2)')
plt.plot([0,1],[0,1],'--') plt.xlabel('FPR')
plt.ylabel('TPR') plt.title('ROC Curve for Minority
Class') plt.legend() plt.subplot(1,2,2) plt.plot(rec,
prec, label='PR Curve (class 2)') plt.xlabel('Recall')
plt.ylabel('Precision') plt.title('Precision-Recall
```

Curve for Minority Class')plt.legend() plt.tight_layout() plt.show() give me the answer like the last one

Here is a detailed yet clear algorithm for multiclass classification model training and evaluation, including formulas for key metrics and plotting ROC & Precision-Recall curves for an imbalanced class setting:

Multiclass Classification and Evaluation

Import Statements

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, label_binarize
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, roc_auc_score, average_precision_score,
                             roc_curve, precision_recall_curve)
```

Algorithm Steps and Formulas

1. **Generate Imbalanced Multiclass Dataset**
    - Create $n = 3000$ samples, 50 features (informative + redundant), 3 classes with weights $0.6, 0.3, 0.1$, and noise level 2%:

$$(X, y) = \text{make\_classification}(\dots)$$

2. **Split Dataset Stratified Into Train/Test Sets**
$$(X_{\text{train}}, X_{\text{test}}, y_{\text{train}}, y_{\text{test}}) = \text{train\_test\_split}(X, y, \text{test\_size} = 0.2, \text{stratify} = y)$$
3. **Model Training and Evaluation Loop**
    - For each model (Logistic Regression, Decision Tree, SVM):
        - **Scale features** only for Logistic Regression and SVM:

$$X_{\text{train\_scaled}} = \frac{X_{\text{train}} - \mu}{\sigma}, \quad X_{\text{test\_scaled}} = \frac{X_{\text{test}} - \mu}{\sigma}$$

        - Fit the classifier to training data.
        - Predict labels $ \hat{y} $ and probabilities $ \hat{p} $ on test set.
        - Compute metrics (macro averaging used for balanced evaluation):
            - Accuracy:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total predictions}}$$

        - Precision:

$$\text{Precision} = \frac{1}{C}\sum_{c=1}^{C}\frac{TP_c}{TP_c + FP_c}$$

        - Recall:

$$\text{Recall} = \frac{1}{C}\sum_{c=1}^{C}\frac{TP_c}{TP_c + FN_c}$$

        - F1-score:

$$\text{F1} = \frac{1}{C}\sum_{c=1}^{C}\frac{2 \cdot Precision_c \cdot Recall_c}{Precision_c + Recall_c}$$

        - ROC-AUC using One-vs-Rest:

$$\text{ROC-AUC} = \frac{1}{C}\sum_{c=1}^{C}\text{AUC-ROC}_c$$

        - Average Precision (PR-AUC):

$$\text{PR-AUC} = \frac{1}{C}\sum_{c=1}^{C}\text{Average Precision}_c$$

    - Print all metrics for each classifier.

4. **Plot ROC and Precision-Recall Curves for Minority Class**
    - For class 2 (minority), binarize labels and plot:
        - ROC curve: plot False Positive Rate vs True Positive Rate:

$$\text{FPR}, \text{TPR} = \text{roc\textbackslash\_curve}(y_{\text{bin}}^{(2)}, \hat{p}^{(2)})$$

        - Precision-Recall curve: plot Recall vs Precision:

$$\text{Precision}, \text{Recall} = \text{precision\textbackslash\_recall\textbackslash\_curve}(y_{\text{bin}}^{(2)}, \hat{p}^{(2)})$$

        - Display both plots side by side.

Summary Pseudocode

```
1. Import libraries
2. Generate multiclass imbalanced dataset
3. Split data stratified into train/test
4. For each classifier:
   - Scale data if needed
   - Fit model
   - Predict labels and probabilities
   - Compute accuracy, precision, recall, F1 (macro)
   - Compute ROC-AUC and PR-AUC (macro, OvR)
   - Print metrics
5. For minority class:
   - Binarize labels for class 2
```

```
    - Plot ROC curve
    - Plot Precision-Recall curve
    - Show plots
```

This pipeline trains and evaluates multiclass classifiers with appropriate scaling, computes comprehensive metrics including macro averages to handle class imbalance, and visualizes ROC and PR curves for the minority class to assess detection quality in imbalanced settings. [31] [32] [33] [34]

⁂

1. https://www.geeksforgeeks.org/machine-learning/image-segmentation-using-k-means-clustering/
2. https://www.sciencedirect.com/science/article/pii/S1877050915014143
3. https://www.kdnuggets.com/2019/08/introduction-image-segmentation-k-means-clustering.html
4. https://en.wikipedia.org/wiki/K-means_clustering
5. https://www.youtube.com/watch?v=22mpExWh1LY
6. https://en.wikipedia.org/wiki/Image_segmentation
7. https://www.mathworks.com/help/images/ref/imsegkmeans.html
8. https://www.tutorialspoint.com/image-segmentation-by-clustering
9. https://neptune.ai/blog/k-means-clustering
10. https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/
11. https://scikit-learn.org/stable/auto_examples/decomposition/plot_kernel_pca.html
12. https://www.geeksforgeeks.org/machine-learning/ml-introduction-to-kernel-pca/
13. https://scikit-learn.org/1.0/auto_examples/decomposition/plot_kernel_pca.html
14. https://www.reddit.com/r/MachineLearning/comments/fl1rfs/discussion_kernel_pca_vs_regular_pca/
15. https://nirpyresearch.com/pca-kernel-pca-explained/
16. https://www.baeldung.com/cs/kernel-principal-component-analysis
17. https://andrewcharlesjones.github.io/journal/kernel-pca.html
18. https://www.dailydoseofds.com/p/kernelpca-vs-pca/
19. https://www.cs.mcgill.ca/~dprecup/courses/ML/Lectures/ml-lecture13.pdf
20. https://www.youtube.com/watch?v=rle8kKZfvLk
21. https://www.geeksforgeeks.org/machine-learning/image-segmentation-using-k-means-clustering/
22. https://neptune.ai/blog/k-means-clustering
23. https://www.sciencedirect.com/science/article/pii/S1877050915014143
24. https://www.kdnuggets.com/2019/08/introduction-image-segmentation-k-means-clustering.html
25. https://en.wikipedia.org/wiki/K-means_clustering
26. https://www.youtube.com/watch?v=22mpExWh1LY
27. https://www.mathworks.com/help/images/ref/imsegkmeans.html
28. https://www.tutorialspoint.com/image-segmentation-by-clustering
29. https://www.mathworks.com/help/images/color-based-segmentation-using-k-means-clustering.html
30. https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/

31. https://www.evidentlyai.com/classification-metrics/multi-class-metrics
32. https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall
33. https://magai.co/weighted-metrics-for-multi-class-models-explained/
34. https://towardsdatascience.com/multiclass-classification-evaluation-with-roc-curves-and-roc-auc-294fd4617e3a/
35. https://arxiv.org/abs/2008.05756
36. https://www.kaggle.com/code/nkitgupta/evaluation-metrics-for-multi-class-classification
37. https://www.geeksforgeeks.org/machine-learning/metrics-for-machine-learning-model/
38. https://www.youtube.com/watch?v=uXwhX8ox2lk
39. https://accelerated-data-science.readthedocs.io/en/v2.13/user_guide/model_evaluation/Multiclass.html
40. https://docs.aws.amazon.com/machine-learning/latest/dg/multiclass-classification.html