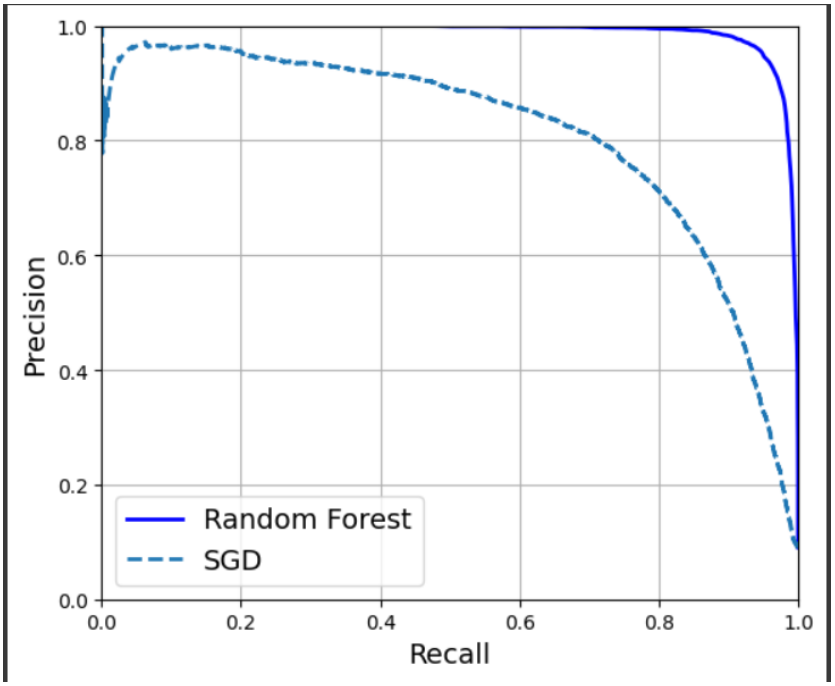
CSCI 581 – ASSIGNMENT 2

AKSHITH MACHARLA

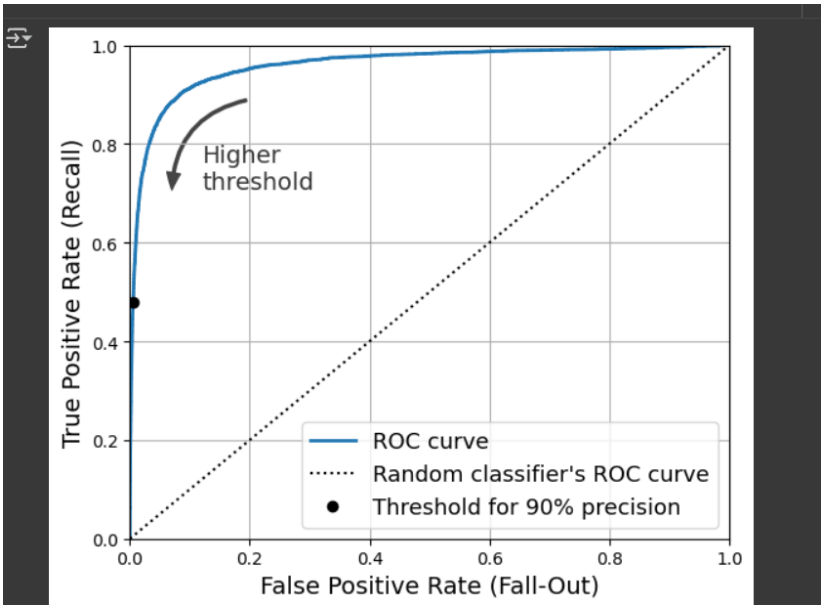
012642059

1) Provide a screen shot of the PR curve and ROC:

PR_curve:



ROC_CURVE:



1a)

a) Context. In a binary classifier every prediction is compared to the true label and counted as one of four raw outcomes:

- TP (True Positive): model predicted positive and it was actually positive.
- FP (False Positive): model predicted positive but it was actually negative.
- TN (True Negative): model predicted negative and it was actually negative.
- FN (False Negative): model predicted negative but it was actually positive.

- **Precision (P)** — *How accurate are our positive predictions?*
  Among all items the model **predicted as positive**, the fraction that are truly positive:

$$P = \frac{TP}{TP + FP}$$

Recall (R) — *How completely did we find the positives?*
Among all items that are **actually positive**, the fraction the model correctly found:

$$R = \frac{TP}{TP + FN}$$

Edge conditions

- If TP+FP =0 (the model never predicts positive), precision is undefined.
- If TP+FN=0 (no actual positives), recall is undefined.

Precision =TP/ (TP+FP) gauges the correctness of positive calls, while Recall =TP/(TP+FN) gauges the coverage of actual positives.

1b)

PR AUC = area under the Precision–Recall curve across all decision thresholds. It summarizes the trade-off between precision (TP/(TP+FP)) and recall (TP/(TP+FN)).

Why it's useful:

- **Threshold-independent:** One number that evaluates performance over *all* thresholds.
- **Focuses on positives:** Directly reflects few FP (good precision) and few FN (good recall).
- **Works well with class imbalance:** Unlike ROC AUC, PR AUC isn't inflated by negatives. Its baseline ≈ positive rate, so improvements are meaningful.
- **Average Precision:** Often reported as **AP**, i.e., the average precision achieved while recall increases from 0→1.
- **Measures ranking quality:** High PR AUC means the model ranks true positives above negatives reliably, so we can choose a threshold later.

Higher is better. The model keeps precision high while achieving high recall.

In simpler words, PR AUC is one number that tells how well a model finds real positives while avoiding false rates—especially when positives are rare.

1c)

What ROC is
The ROC curve plots True Positive Rate (TPR = Recall = TP/(TP+FN)) on the y-axis versus False Positive Rate (FPR = FP/(FP+TN)) on the x-axis while the decision threshold is varied. ROC AUC summarizes this curve into one number (higher is better).

Why ROC is useful in addition to PR

- **Explicit view of negatives:** ROC shows **FPR** directly, so we see how often negatives are incorrectly labeled as positive as recall increases.
- **Less sensitive to class imbalance:** ROC is less affected by the proportion of positives vs. negatives than PR, so it remains informative when class prevalence changes.

- **Clear operating-point selection:** The ROC curve makes it easy to choose a threshold that satisfies constraints such as a maximum allowable FPR or a minimum required TPR.

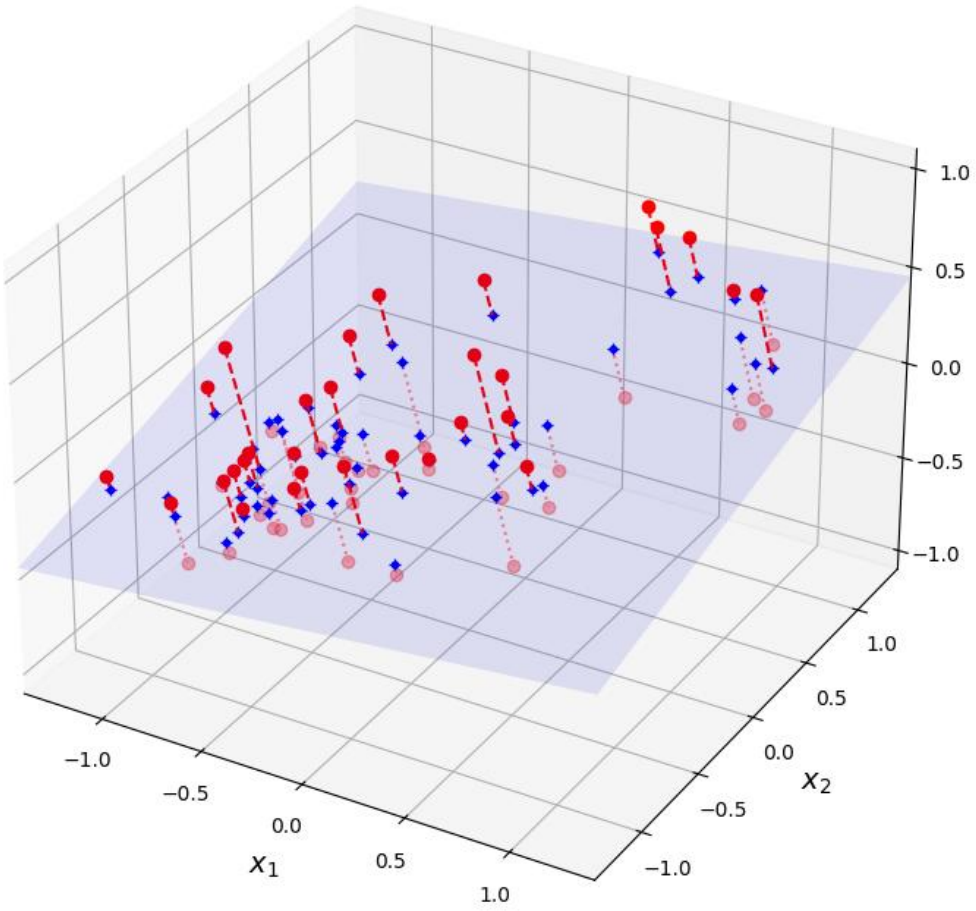
How to use ROC to pick a threshold

- **Limit false positives:** We choose the highest threshold whose FPR ≤ α (the allowed false-positive rate). Read the corresponding TPR at that point.
- **Guarantee sensitivity:** We choose the lowest threshold whose TPR ≥ β (the required recall). Check the resulting FPR.
- **Balanced choice:** We pick the point closest to (0,1) (top-left of the ROC plot)
- Apply the selected threshold to model scores to produce final class labels.

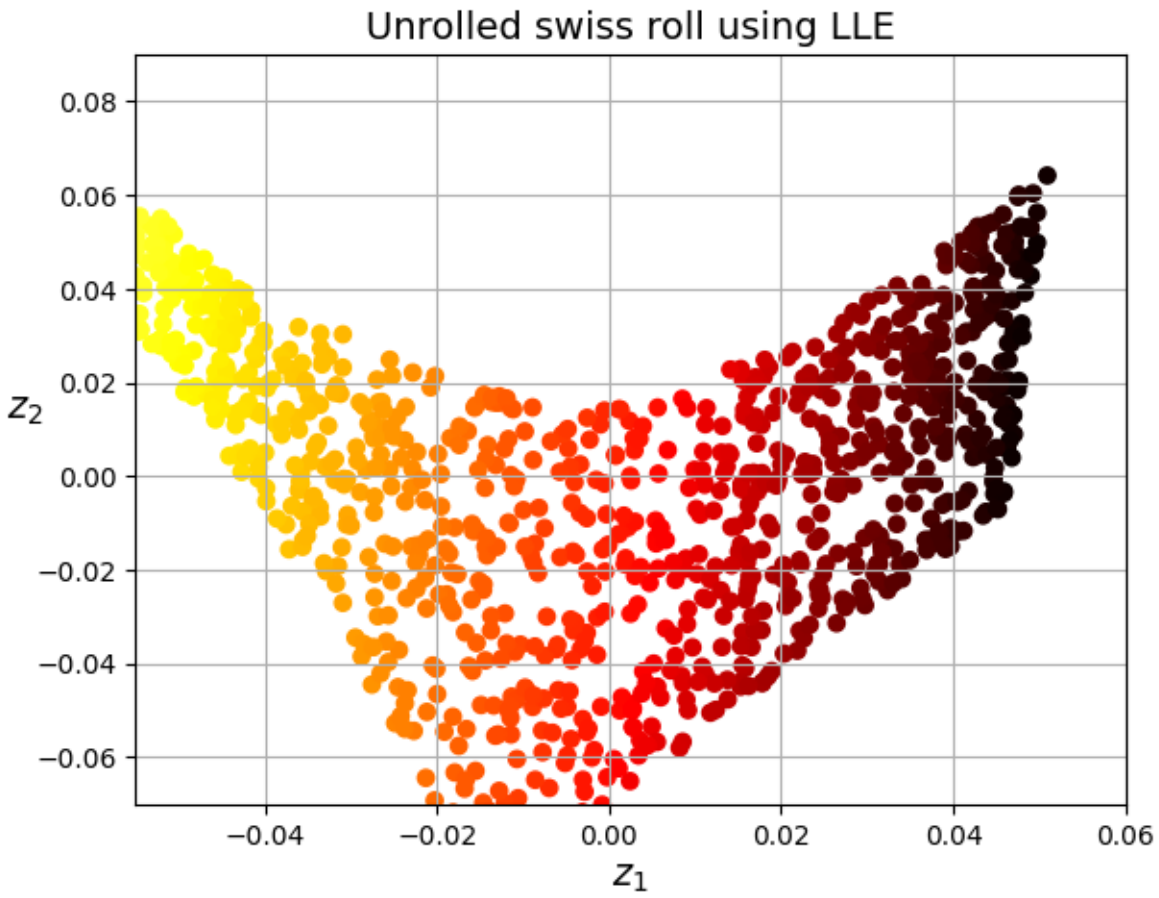
Finally, PR focuses on performance with respect to the positive class (precision and recall). ROC adds a direct view of the false-positive rate and supports principled threshold selection under explicit performance constraints.

2a)

simple 3D data projection onto a 2D plane



more complex Swiss roll manifold
dimensional reduction:



2a) Because many 3D datasets don't lie near a *flat* surface. A simple 2D plane is flat; if the data lives on a curved, twisted surface (a *manifold*) like the Swiss roll, flattening it by straight projection crushes different layers together, destroying neighborhood structure and important information.

- **PCA (and any linear projection) is flat.** It looks for a single plane that best fits the data. This works only when the data roughly sits on a flat sheet in 3D (a *linear subspace*).
- **Curved data ≠ flat plane.** In the Swiss roll, points that are far apart along the roll become **stacked on top of each other** when we drop to a plane—so distances and neighborhoods get scrambled.
- **Information loss.** After such a projection, points that shouldn't meet will overlap; clusters can merge; decision boundaries can become much more complicated or meaningless.

- **What to use instead.** For curved structures, we use **manifold learning** methods (e.g., LLE, Isomap, t-SNE) that try to **unroll** the surface before reducing to 2D, preserving local relationships.

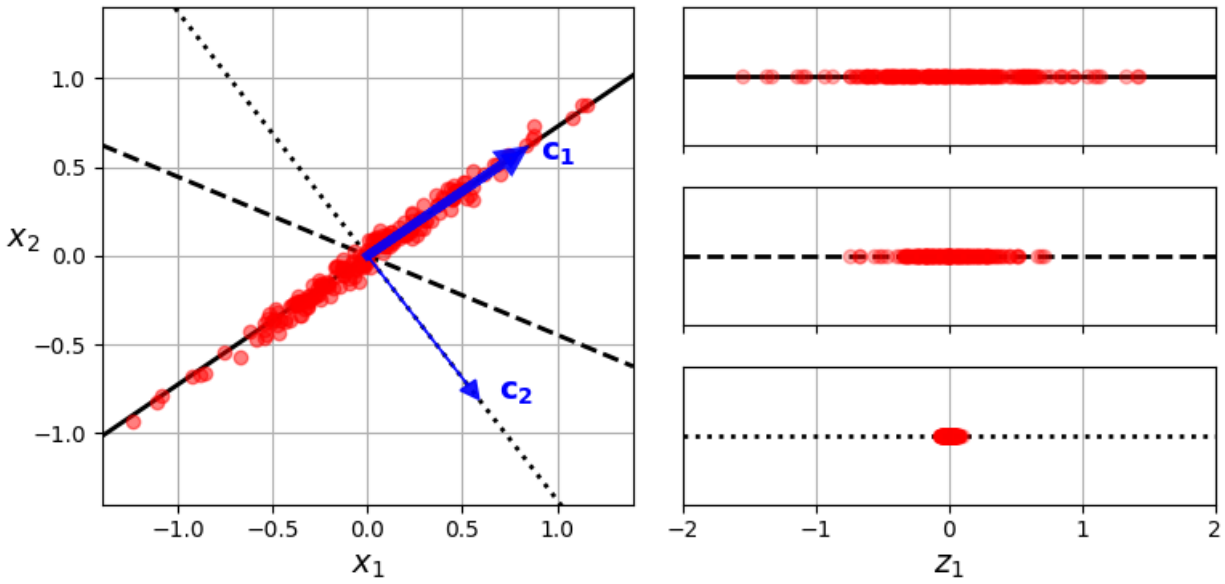
2b) Principle behind PCA (Principal Component Analysis)

PCA finds a new set of perpendicular axes (called principal components) that best summarize the data.

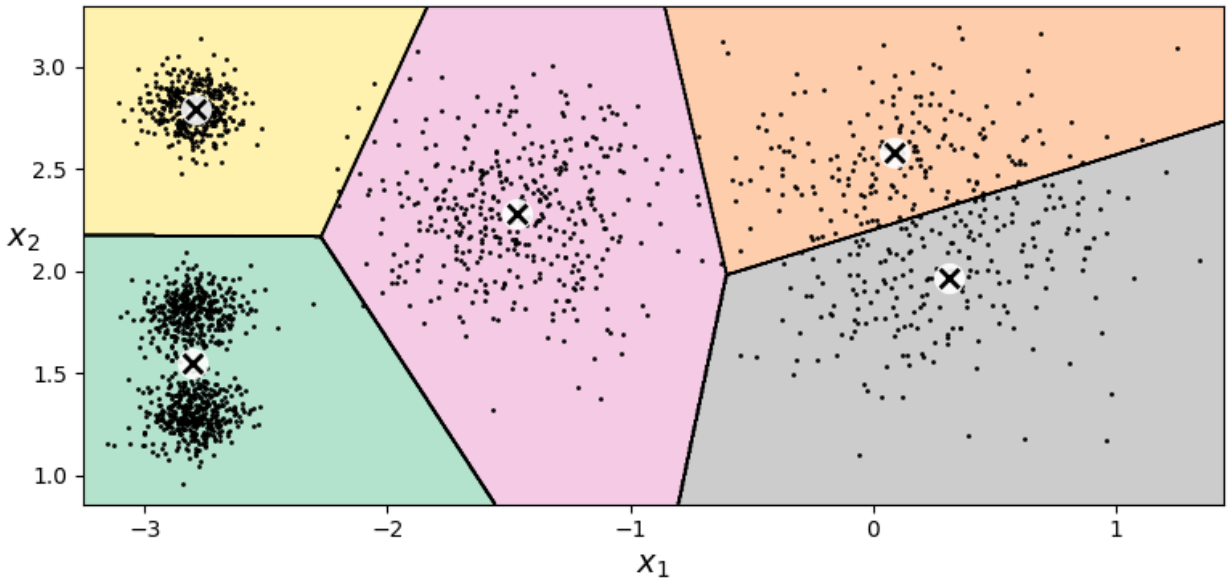
- The 1st principal component is the direction that captures the maximum variance in the data.
- The 2nd principal component is orthogonal to the first and captures as much remaining variance as possible, and so on.
  Equivalently, PCA chooses the projection that minimizes the average squared reconstruction error between the original points and their projections onto these axes.

What Figure 8-7 shows

- **Left panel:** A 2D scatter with three candidate directions drawn. The solid line aligns with the spread of the data—this is the best single projection because it preserves the most variance. The dashed and dotted lines preserve less. The bold arrows labeled c_1 and c_2 depict the first and second principal components (orthogonal to each other).
- Right panels: The result of projecting the same data onto each of the three candidate directions. We can see that projecting onto the solid line keeps the data spread out (high variance), while the dotted line collapses the spread (low variance). This visually demonstrates why PCA chooses the solid-line direction as the first principal component.



3)



3a) It means: when K-Means labels space by "nearest centroid," the plane gets split into cells where every point in a cell is closer to its cell's centroid than to any other centroid.

That partition is exactly a Voronoi tessellation (Voronoi diagram):

- One polygonal cell per centroid.
- Cell boundaries are the perpendicular bisectors between pairs of centroids (points equidistant to the two).
- In 2D we see convex polygons; in higher dims they're convex polyhedra.

So, our decision-boundary plot is a Voronoi diagram of the centroids—each colored region shows the "nearest-centroid" territory that K-Means would assign to the same cluster.

3b)

Why K-means might fail & how to avoid it

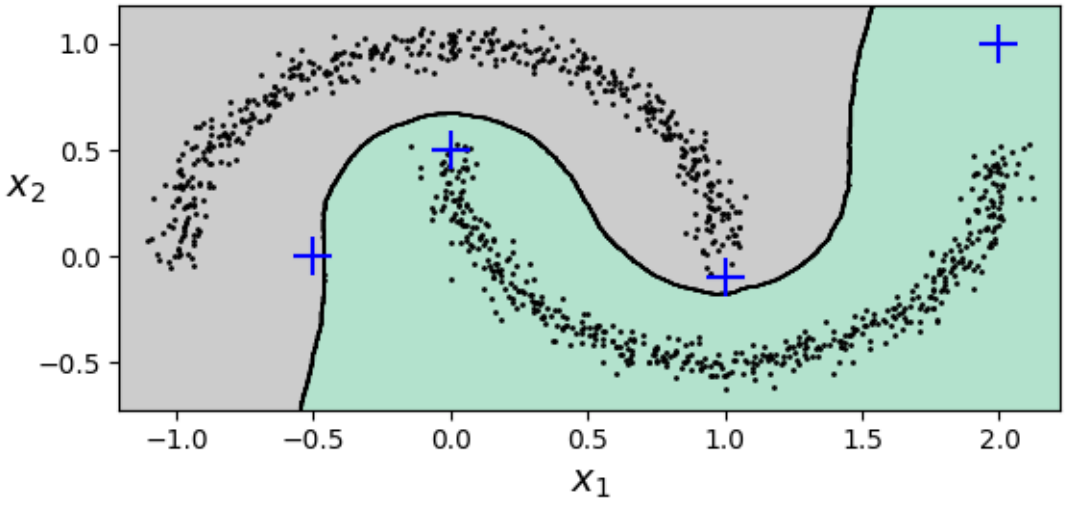
- **Wrong number of clusters (k):** If k is too small/large, clusters get merged or split arbitrarily; inertia always drops with larger k, so it's not a reliable chooser by itself.

- **Sensitive to initialization:** Different random starts can converge to different (sub-optimal) solutions (local minima).
- **Assumes spherical, equal-size, equal-density clusters:** It struggles with elongated/elliptical shapes, clusters of very different sizes/densities, or curved structures (e.g., two moons).
- **Scale sensitivity:** Features on larger scales dominate distance calculations, distorting clusters.
- **Outliers/noise:** A few faraway points can drag centroids and boundaries.

At least two ways to avoid failure (plus a few more)

1. **Use k-means++ initialization + multiple restarts (n_init):** Better seeds and repeated runs reduce bad local minima.
2. **Choose k with data-driven criteria:** We can use the elbow (inertia-vs-k) and silhouette score/diagrams to pick a sensible k instead of minimizing inertia alone.
3. **Scale features first:** Standardize/normalize so all dimensions contribute fairly.
4. **Use a more appropriate algorithm when shapes/densities violate K-means assumptions:**
    - **Gaussian Mixture Models (GMMs)** for ellipsoidal clusters with different sizes/orientations.
    - **DBSCAN** (or spectral/agglomerative) for arbitrary shapes or varying densities and to handle noise.
5. **Mitigate outliers:** Remove/clip obvious outliers or use robust preprocessing before clustering.

DBSCAN:



3c) WE **Use DBSCAN instead of K-Means when:**

- Clusters are non-spherical / arbitrarily shaped (e.g., two “moons”, Swiss-roll neighborhoods). K-Means assumes roughly spherical, equal-variance clusters.
- We don’t know the number of clusters in advance. DBSCAN infers it; K-Means requires us to set *k*.
- WE need built-in outlier detection. DBSCAN labels low-density points as –1 (noise); K-Means assigns every point to some cluster.
- Cluster densities are uneven. K-Means struggles when clusters differ in size/density; DBSCAN can still separate dense groups from sparse regions (with a good eps).
- Decision boundaries are highly non-linear. DBSCAN’s density connectivity captures curved/linked structures; K-Means yields linear Voronoi splits.

Prefer K-Means when:

- Clusters are roughly spherical, similar in size/density, *k* is known, and we need speed/scale (very large datasets).

3d)

Sensitive to eps & min_samples: Picking good values is tricky; poor choices merge/split clusters or mark many points as noise.

Struggles with varying densities: If true clusters have different densities, one eps can't fit all—DBSCAN may miss or merge clusters.

No built-in predict() for new points: Scikit-Learn's DBSCAN lacks a predict() method; we need a separate classifier (e.g., k-NN) to label new data.

High-dimensional data issues: Distance metrics become less meaningful; performance and cluster quality degrade in many dimensions.

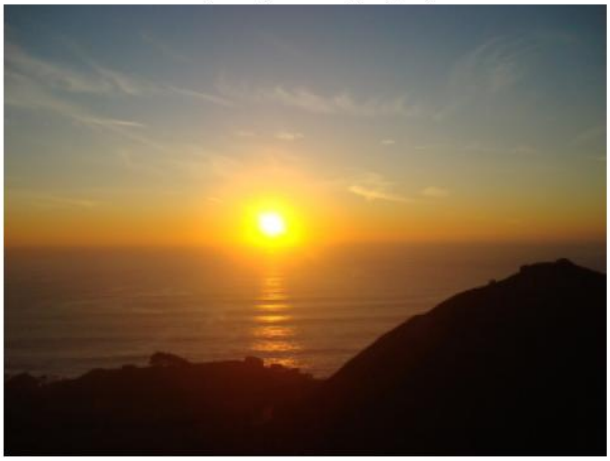
Memory/complexity caveats: With large eps, scikit-learn's implementation can need up to **$O(m^2)$** memory; runtime can grow and be slower than K-Means on big sets.

Requires feature scaling & metric choice: Results are sensitive to scale and the distance metric; unscaled features can distort density estimates.

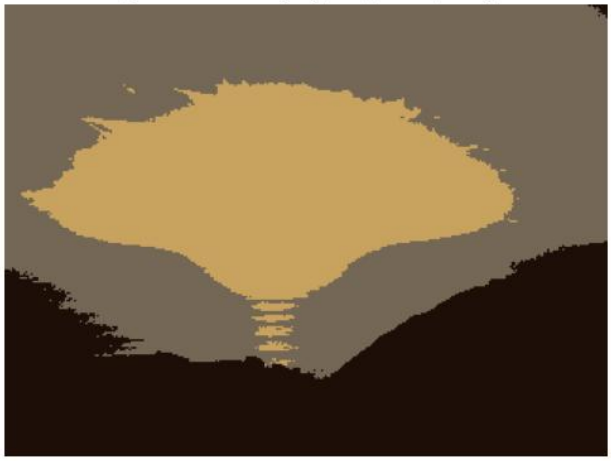
4) This is 4 classes : sky water sun and land

Three classes: water sky and sun

Original (sun-image.ppm)



Quantized K=3 (sky/sun/water/land)



Original (bricks.jpg)

Quantized K=7

4a)

I used Python.

Why Python?

- **Fast to prototype & matches the references.** The MachineLearningMastery and official OpenCV tutorials I cited use the **Python cv.kmeans** API directly, so I could reproduce

their flow line-for-line and focus on the ML part (reshaping to M×3, calling kmeans, mapping labels→centers, plotting with Matplotlib).

- **Strong ecosystem.** numpy for array ops and matplotlib for side-by-side screenshots made the assignment (and explanation) straightforward without extra boilerplate.

Disadvantages of choosing Python (vs. C):

1. **Lower raw performance.** Pure-Python loops are slow and the GIL limits true multi-threaded CPU speedups. For tight real-time pipelines, C is faster.
2. **Runtime type errors.** Dynamic typing makes it easy to pass the wrong shape/dtype and only discover it at runtime (C would catch more at compile time).
3. **Packaging/ports can be messy.** Managing wheels for opencv-python and matching system libraries can be tricky across machines; C can ship a single static binary.
4. **Memory & control.** Less fine-grained control over memory/layout compared to C; tougher on embedded/edge devices with strict constraints.

4b) Getting OpenCV's kmeans() call exactly right

- Input must be (H·W)×3 float32, not uint8; the criteria tuple must be (type, max_iter, eps); and we must pass attempts and an init flags value. One wrong dtype/shape and it silently misbehaves or throws.
- After clustering, labels is (H·W,1) and centers is (K,3)—so the correct remap is centers[labels].reshape(H,W,3) and then cast back to uint8. Getting that indexing/reshape right is the kernel of the app.

2. **BGR vs RGB + reproducible screenshots**

- OpenCV reads BGR, while Matplotlib expects RGB. Without cvtColor(..., COLOR_BGR2RGB) the side-by-side display (required by the assignment) looks wrong.
- Headless/Colab rendering and saving a single PNG with both original and quantized images (titles, no axes, tight layout) is a bit of glue code that the starter snippets don't include.

3. **Choosing K to match the *semantic* ask with an unsupervised tool**

- We want *sky/sun/water* (sun image) and *brick colors* (Legos). Plain k-means only "sees" colors, not concepts, so picking K that separates those regions cleanly takes trial/error. Too small K merges classes; too big K fragments them.

4. **Handling randomness & stability**

- Different initializations can shift clusters, especially on the sun image. I had to use multiple attempts and fix the random init mode (KMEANS_RANDOM_CENTERS) to get stable-enough results for screenshots.

5. **Packaging it as a repeatable mini-app**

- Adding small but necessary bits the starters don't show: image download/IO, path checks, parameterization (K, attempts), console prints (compactness, unique colors), and a single function that does read → quantize → display → save.

6. **Edge cases**

- Mixed formats (e.g: .ppm) and large images can blow up memory/time if we forget the reshape/cast details or use an excessive K.