Centro de Investigación en Matemáticas, A.C.

Reconocimiento de Patrones

Jose Miguel Saavedra Aguilar

CIMAT

Examen 2. Ejercicio 1

In [1]:
```python
from PIL import Image
import numpy as np
```

Sea θ^* el minimizador de la función

$$f(\theta) = \sum_i |y_i - \theta|,$$

donde y_i son observaciones de una v.a. Y.

Sea θ^n el n-ésimo estimador de θ^*, entonces:

$$h_i(\theta \mid \theta^n) = 0.5\frac{(y_i - \theta)^2}{|y_i - \theta^n|} + 0.5|y_i - \theta^n|$$

mayoriza $|y_i - \theta|$ para todo θ.

Demostración: Supongamos que $y_i \neq \theta^n$, entonces

$$\frac{1}{2|y_i - \theta^n|}\left(|y_i - \theta| - |y_i - \theta^n|\right)^2 \geq 0,$$

$$\frac{1}{2|y_i - \theta^n|}\left(|y_i - \theta|^2 - 2|y_i - \theta||y_i - \theta^n| + |y_i - \theta^n|^2\right) \geq 0$$

$$\frac{1}{2}\left(\frac{|y_i - \theta|^2}{|y_i - \theta^n|} - 2|y_i - \theta| + |y_i - \theta^n|\right) \geq 0$$

$$0.5\frac{|y_i - \theta|^2}{|y_i - \theta^n|} - |y_i - \theta| + 0.5|y_i - \theta^n| \geq 0$$

$$0.5\frac{(y_i - \theta)^2}{|y_i - \theta^n|} + 0.5|y_i - \theta^n| \geq |y_i - \theta|$$

Por lo tanto, $h_i(\theta \mid \theta^n) \geq |y_i - \theta|$.

Además, evaluando en $\theta = \theta^n$ tenemos que

$$h_i(\theta^n \mid \theta^n) = 0.5\frac{(y_i - \theta^n)^2}{|y_i - \theta^n|} + 0.5|y_i - \theta^n| = |y_i - \theta^n|,$$

de forma que $h_i(\theta \mid \theta^n)$ mayoriza $|y_i - \theta^n|$.

Ahora, el algoritmo MM para encontrar la mediana de un conjunto de datos X, también conocido como algoritmo de Weiszfeld es el siguiente:

In [ ]:
```
def mm_median(X, m=None, max_iters=20, tol=1e-4, random_state=None):
    """
    MM method for computing the median of a dataset X.
    Parameters:
    - X: array-like, shape (n_samples, n_features)
        The input data.
    - m: array-like, shape (n_features,), optional
        Initial guess for the median. If None, it will be initialized to the mean o
    - max_iters: int, optional
        Maximum number of iterations for convergence.
    - tol: float, optional
        Tolerance for convergence.
    - random_state: int, optional
        Random seed for reproducibility.
    """
    X = np.asarray(X)
    if random_state is not None:
        np.random.seed(random_state)
    n_samples, n_features = X.shape
    # Initialize median
    if m is None:
        m = np.mean(X, axis=0)
    elif len(m) != n_features:
        raise ValueError(f"Expected median with {n_features} features, but got {len
    else:
        m = np.asarray(m)
    u = np.ones(n_samples)
    for _ in range(max_iters):
        # Compute weights inversely proportional to distance from current median
        w = np.linalg.norm(X - m, axis=1)
        w = 1 / (w + tol)
        # Update median as weighted mean
        new_m = np.dot(w, X) / np.dot(w, u)
        if np.linalg.norm(new_m - m) < tol:
            break
        m = new_m
    return m
```

A continuación, el algoritmo de k-medianas que utiliza las medianas calculadas por el algoritmo MM anterior:

In [ ]:

```
def kmedians(X, k, max_iters = 30, tol=1e-4, random_state=None, medians=None):
    """
    K-medians clustering algorithm using the MM method for computing medians.
    Parameters:
    - X: array-like, shape (n_samples, n_features)
        The input data.
    - k: int
        The number of clusters.
    - max_iters: int, optional
        Maximum number of iterations for convergence.
    - tol: float, optional
        Tolerance for convergence.
    - random_state: int, optional
        Random seed for reproducibility.
    - medians: array-like, shape (k, n_features), optional
        Initial medians for the clusters. If None, they will be randomly initialize
    """
    X = np.asarray(X)
    if random_state is not None:
        np.random.seed(random_state)
    n_samples, n_features = X.shape
    # Initialize medians
    if medians is None:
        indices = np.random.choice(n_samples, size=k, replace=False)
        medians = X[indices]
    elif len(medians) != k:
        raise ValueError(f"Expected {k} medians, but got {len(medians)}")
    elif np.asarray(medians).shape[1] != n_features:
        raise ValueError(f"Expected medians with {n_features} features, but got {np
    else:
        medians = np.array(medians, copy=True)
    for _ in range(max_iters):
        # Compute distances from each point to each centroid
        distances = np.linalg.norm(X[:, np.newaxis] - medians, axis=2)
        # Assign each point to the nearest centroid
        labels = np.argmin(distances, axis=1)
        new_medians = medians.copy()
        for j in range(k):
            mask = (labels == j)
            if np.any(mask):
                # Update centroid as the weighted median of its cluster
                new_medians[j] = mm_median(X[mask], m=medians[j])
        # Check for convergence
        if np.linalg.norm(new_medians - medians) < tol:
            break
        medians = new_medians.copy()
    return medians, labels
```

Definimos funciones auxiliares para convertir la imagen a datos, y la función inversa:

In [3]:

```
def image_to_features(image_path):
    # Load an image and convert it to RGB format
    img = Image.open(image_path).convert('RGB')
    img_np = np.array(img)
    n, m, c = img_np.shape
```

```
    # Reshape image array to a 2D array where each row is a pixel's RGB values
    features = img_np.reshape(-1, 3)
    # Return the features and the original image shape (rows, columns)
    return features, (n, m)

def features_to_image(features, shape):
    # Reshape the 2D feature array back to the original image shape
    n, m = shape
    return features.reshape(n, m, 3).astype(np.uint8)
```

Para los 4 valores de k, (2, 3, 5 y 10), se ejecuta el algoritmo de k-medianas para colorear la imagen:

In [ ]:
```
# Quantize image for different numbers of clusters using k-medians
K = [2, 3, 5, 10]

features, shape = image_to_features("Figures/foto.jpg")  # Convert image to feature

for k in K:
    medians, labels = kmedians(features, k=k, random_state=2025)  # Cluster pixels
    output = np.zeros_like(features)  # Prepare array for quantized image
    for i in range(shape[0] * shape[1]):
        output[i] = medians[labels[i]]  # Assign each pixel the median color of its
    reconstructed_img = features_to_image(output, shape)  # Convert features back t
    Image.fromarray(reconstructed_img).save(f"kmedians_{k}.png")  # Save quantized
```

Las imágenes resultantes se encuentran guardadas con el nombre 'kmedians_x.png', donde x es el número de medianas utilizadas.