93.75 - Métodos Numéricos Avanzados

Instituto Tecnológico de Buenos Aires

---

# Trabajo Práctico 1

# Reconocimiento Facial

---


*Profesores:*

Pablo Ignacio, Fierens
Adrián Omar, Alvarez

*Grupo:*

Esteban, Kramer 55200
Oliver, Balfour 55177
Martina, Scomazzon 55410

# Índice

# 1. Resumen del Proyecto

Este trabajo consiste en describir las metodologías utilizadas para desarrollar un sistema de reconocimiento facial. Se desarrolló un sistema que, dada la foto de la cara de una persona, dirá a que sujeto de una base de datos pertenece. Para desarrollar dicho sistema, nos basamos en algoritmos algo antiguos para el pre-procesamiento de los datos:

1. *Principal Component Analysis* (PCA).
2. *Kernel Principal Component Analysis* (KPCA).

# 2. Introducción

Los sistemas de reconocimiento facial son una tecnología que permite identificar a una persona rápidamente analizando las características biométricas de su rostro. En una fase inicial, se usaban modelos geométricos simples, pero actualmente el análisis está ligado a sofisticados procesos matemáticos y algoritmos de coincidencia. Estos, consiguen medir la forma y las estructuras únicas de los rostros.

Por medio de los conocimientos adquiridos a lo largo de la cursada e investigando, logramos desarrollar nuestro propio sistema de reconocimiento facial. Los métodos utilizados para realizar el sistema fueron KPCA y PCA. Dado que estos métodos requieren el calculo de autovectores y autovalores, implementamos la descomposición QR para su iteración y búsqueda. Además, se implemento un método para obtener los *singular vectors*.

En este informe, nos proponemos mostrar como funciona nuestra implementación del sistema, y comparar los resultados obtenidos mediante los diferentes métodos que implementamos.

# 3. Implementación

Con el fin de lograr nuestro objetivo, reconocer caras a través de aplicaciones numéricas, hemos diseñado distintos algoritmos que nos ayudaran a lograrlo.
En el siguiente apartado daremos una breve descripción sobre la implementación de los mismos.

## 3.1. Tecnología utilizada

El código fue realizado en base a las implementaciones provistas por la cátedra. El mismo fue desarrollado en **Python 3**.

### 3.1.1. Librerías Utilizadas

Detallaremos algunas de las librerías externas mencionadas a lo largo del proyecto y para que han sido utilizadas

- **numpy**: Operaciones de matrices
- **sklearn**: Algoritmos de aprendizaje
- **imageio**: Leer imagenes
- **matplotlib.pyplot**: Graficar error
- **configparser**: Parseo del archivo de configuración

## 3.2. Descomposición QR

Antes de adentrarnos en el código per se, nos parece pertinente comentar y discutir: ¿Para que sirve la descomposición QR? ¿Existe mas de una forma? En caso de existir, ¿Cual es mejor?

La descomposición o factorización QR sirve para obtener autovalores y autovectores de una matriz. Esencialmente,

$$A = QR \tag{1}$$

Donde A representa una matriz cuadrada de $n\ x\ n$, R una matriz triangular superior, y por ultimo Q, matriz ortogonal tal que,

$$I = Q^T Q \tag{2}$$

I describe la matriz identidad.

Existen diversas formas de obtener dicha descomposición, donde se destacan: *Gram-Schmidt*, *Householder*, *Givens*. Hemos implementado dos de ellos, Gram-Schmidt y Householder. Luego de algunas investigaciones, logramos determinar que el mejor método a aplicar era el método de las rotaciones de Householder ya que el mismo posee un factor de ortogonalización mas preciso, es decir, su error, es menor. Además, es mas estable numéricamente hablando. A continuación, la implementación de nuestros métodos:

### 3.2.1. Gram - Schmidt

```
import numpy as np

def gram_schmidt(matrix):

    m, n = matrix.shape
    # Initialize matrix with zeros

    Q = np.zeros((m, n))
    R = np.zeros((n, n))
    A = np.copy(matrix)

    for i in range(n):
        v = A[:, i]
        norm = np.linalg.norm(v)
        Q[:, i] = v / norm
        R[i, i] = norm
        for j in range(i+1,n):
            q = np.transpose(Q[:, i])
            R[i, j] = q.dot(A[:, j])
            A[:, j] = A[:, j] - R[i, j] * q

    return Q, R
```

#### 3.2.2. Rotación de Householder

```
import numpy as np

def householder(matrix):

    m,n = matrix.shape
    R = np.asmatrix(matrix)
    Q = np.eye(m)

    for i in range(n):

        #find H = I-bvv'
        normx = np.linalg.norm(R[i:, i])
        x = R[i:, i]

        p = np.multiply(-1,np.sign(x[0]))
        u = x[0] - np.multiply(normx, p)
        v = np.divide(x,u)
        v[0] = 1
        b = np.divide(np.multiply(np.multiply(-1,p),u), normx)

        R[i:, :] = R[i:, :] - np.multiply(b,np.outer(v, v).dot(R[i:, :]))
        Q[:, i:] = Q[:, i:] - np.multiply(b,Q[:, i:].dot(np.outer(v, v)))

    return Q,R
```

### 3.3. Autovalores y Autovectores

Recapitulando, la descomposición QR nos ayuda a obtener los autovalores y autovectores de una matriz. Nos ayuda, pero no es suficiente, por lo tanto iteraremos sobre el mismo hasta converger a una matriz triangular superior.

A partir de esta iteración obtendremos los autovalores y autovectores. Los autovalores están representados en la diagonal de la matriz triangular superior obtenida, mientras que los autovectores no son mas que el producto sucesivo de las columnas de Q.

Como cota, hemos decidido colocar un valor: *MAX_PRECISION* el cual determina la diferencia entre una cota y la anterior, si la diferencia, es menor al mismo, el algoritmo corta. Caso contrario, seguirá iterando hasta *MAX_ITERATIONS*. Esto nos permitirá, obtener el resultado mas exacto posible.

```
import numpy as np

MAX_ITERATIONS = 100
MAX_PRECISION = 10 ** -4
```

```
def iterate_QR(matrix):
    eigenvectors = np.identity(matrix.shape[0])
    A = np.copy(matrix)


    for i in range(MAX_ITERATIONS):
        Q,R = householder(A)
        A = R.dot(Q)
        new_eigenvectors = eigenvectors.dot(Q)
        if np.linalg.norm(np.subtract(new_eigenvectors, eigenvectors))
        < MAX_PRECISION:
            break
        eigenvectors = new_eigenvectors

    eigenvalues = np.diag(A)
    sort = np.argsort(np.absolute(eigenvalues))[::-1]

    return eigenvalues[sort], eigenvectors[sort]
```

### 3.4. Descomposición en Valores Singulares

A lo largo del proyecto, trabajamos con matrices grandes lo que dificulta la obtención de cierta información, como puede ser los autovalores y autovectores de las matrices que conforman las imágenes. Es por ello que decidimos realizar una descomposición en valores singulares, tal que

$$A_{nxp} = U_{nxn} S_{nxp} V^T_{pxp} \tag{3}$$

donde

$$I_{nxn} = U^T U \tag{4}$$

$$I_{pxp} = V^T V \tag{5}$$

Calculando los valores singulares obtenemos los autovalores y autovectores de $AA^T$ y $A^T A$. Este método fue utilizado para PCA.

```
import numpy as np

def my_svd(matrix):

        W = matrix.dot(np.transpose(matrix))
        S, U = descending_eig(W)
        S = np.sqrt(np.abs(S))
        V = np.transpose(matrix).dot(U)
        S1 = np.diag(S)
        for k in range(S1.shape[0]):
            S1[k,k] = 1/S1[k,k]

        V = V.dot(S1)
        return S, np.asmatrix(V.T)
```

### 3.5. KPCA y PCA

Para la implementación de dichos métodos nos basamos en el código provisto por la cátedra.

Poseemos un archivo de configuración en el cual podemos modificar fácilmente los input, en el mismo se decide la cantidad de personas e imágenes a analizar, donde se encuentran las mismas, tamaño de ellas y que método deseamos utilizar: KPCA o PCA.

Según el método seleccionado sera como se comporte el código. Si bien como se desempeña cada uno es distinto, en ambos casos primero entrenamos y luego testeamos.

En el caso de estar en KPCA utilizaremos la descomposición QR para la obtención de autovalores y autovectores, mejor dicho, la iteración del mismo. Mientras que si la opción elegida fue PCA se utilizara la descomposición en valores singulares para la obtención de dichos datos.

## 4. Resultados

Al correr nuestro sistema, pudimos observar los siguientes resultados:

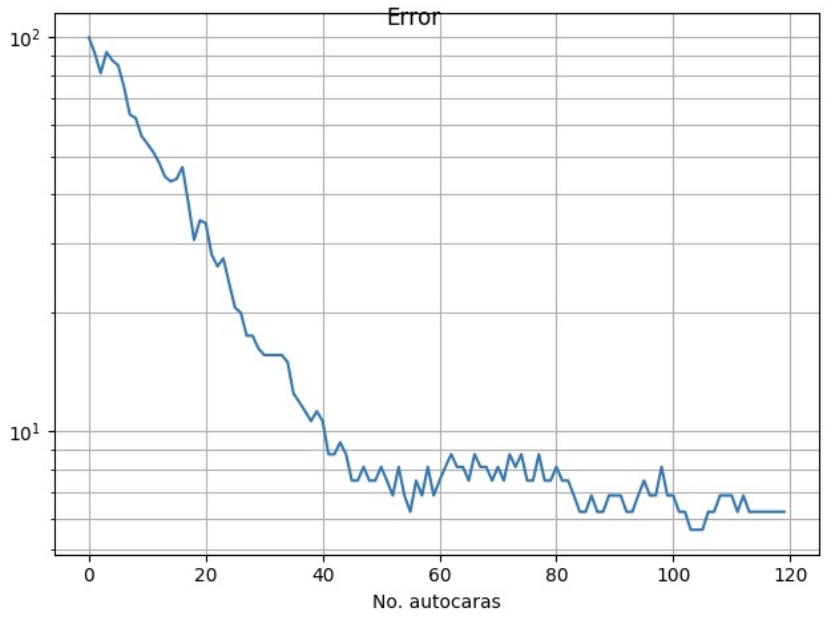


Figura 1: Error en el método KPCA

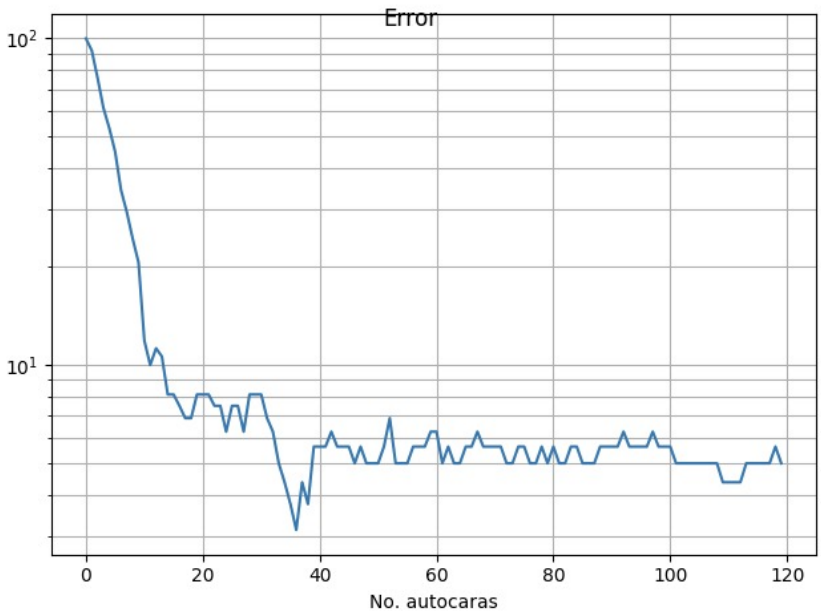


Figura 2: Error en el método PCA

En ambos casos podemos notar como el error disminuye a medida que aumenta el numero de auto-caras. Se puede ver que el error llega a ser menor en KPCA.

Luego decidimos comparar los tiempos de procesamiento de cada método que implementamos para ver su eficiencia. El tiempo de procesamiento es muy importante ya que el reconocimiento de rostro debe ser rápido.

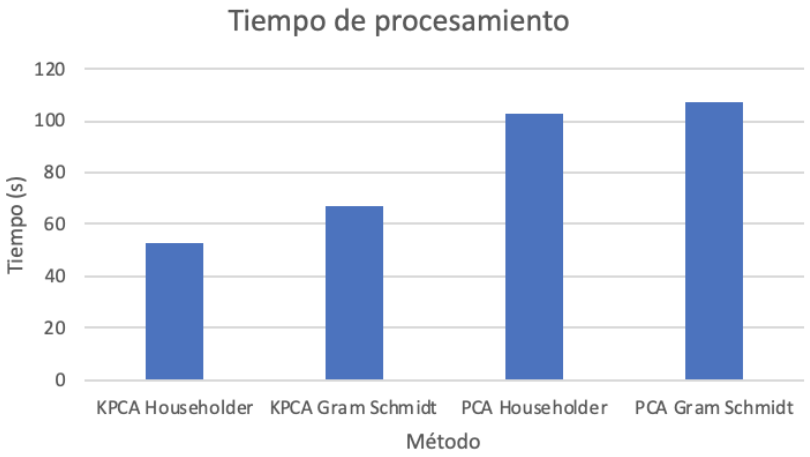


Figura 3: Tiempos de procesamiento de los diferentes métodos

Se puede ver que el método mas eficiente es KPCA con Householder como método de transformación. Mientras que PCA con Gram Schmidt es el menos eficiente, tardando casi el doble que el primero.

La diferencia entre usar Householder y Gram Schmidt es bastante grande ya que nuestras pruebas fueron realizadas en una base de datos pequeña pero en la realidad, son bases de datos muy grandes donde utilizar Householder generaría un gran ahorro de tiempo de procesamiento.

# 5. Conclusión

Analizando los datos obtenidos, podemos afirmar que KPCA produce mejores resultados que PCA en términos de error y tiempo de procesamiento. KPCA resulta mas estable que PCA, al estar mapeando a un espacio mayor puede resolver las direcciones de mayor varianza que no son lineales mientras mantiene las operaciones realizadas en el espacio de origen.

Creemos que para nuestra base de datos, se podría llegar a utilizar PCA pero el método más recomendable es KPCA por los motivos mencionados anteriormente.

# Referencias

[1] Reconocimiento facial: usos y aplicaciones, *Neus Martinez*

[2] Presentaciones provistas por la cátedra

[3] Código provisto por la cátedra

[4] Descomposición en Valores Singulares (2019). En Wikipedia. Recuperado el 3 de Octubre de 2019

[5] Descomposición en Valores Singulares (2019). En Wikipedia. Recuperado el 3 de Octubre de 2019

[6] Singular Value Decomposition (SVD) tutorial (2019). En http://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm. Recuperado el 3 de Octubre de 2019

[7] Householder Transformacion (2019). En http://mlwiki.org/index.php/Householder_Transformation. Recuperado el 3 de Octubre de 2019