

# PROYECTO FINAL

## EXPOSICIÓN DE ARTE

Álvaro Larraya Conejo

## Índice:

PÁGINAS	CONTENIDO
2	Introducción
3	Extracción de datos
4	PCA
5	Elegir el número de clusters
6	Clusters por categorías
7	Conclusión y bibliografía

## Introducción:

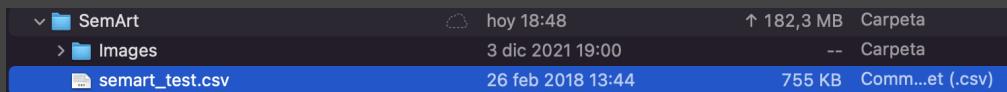
El objetivo de este trabajo es realizar una exposición “interesante”. Cuento con una gran colección de cuadros e información sobre ellos. El enfoque que le he dado al proyecto es realizar un método para estructurar la información de forma que en cada sala haya obras relacionadas entre sí, aunque a simple vista parezca que no.

¿Cómo lo he conseguido?: me he ayudado de conocimientos sobre clasificación no supervisada, más específicamente clustering. Así el proceso es automatizado, lo cual nos permite “jugar” con una cantidad de variables que una persona no podría manejar y además en caso de que haya futuras exposiciones podremos reutilizar el programa.

He decidido usar python, ya que tiene librerías muy potentes tanto para extracción de datos como para machine learning, lo cual me permite realizar código breve, expresivo y eficiente.

## Extracción de datos:

Los datos los obtenemos de SemArt (<http://researchdata.aston.ac.uk/id/eprint/380/>), en esta colección hay cuadros e información relevante sobre ellos. Al haber gran cantidad de obras he decidido quedarme con una pequeña parte del dataset para aligerar el proceso de desarrollo. Nada más descargado el zip borré todos los archivos menos las imágenes y la información de los casos de test:



Lo primero que hice fue obtener información de, por lo menos, 100 cuadros distintos:

```
def leerDescripciones(numDescripciones, fichero):
    dataset = np.loadtxt(fichero, delimiter = '\n', max_rows=numDescripciones, encoding="ISO-8859-1")
    cabecera = dataset[0].split('\t')
    numColumnas = len(cabecera)
    columnas = {}
    for i in range(numColumnas):
        columnas[i] = cabecera[i]
    dataset1 = []
    for i in range(1,numDescripciones):
        if len(dataset[i].split('\t')) == numColumnas:
            dataset1.append(dataset[i].split('\t'))
    dataset = np.array(dataset1)
    return dataset, columnas

dataset, columnas = leerDescripciones(150, 'SemArt/semart_test.csv')

print(dataset.shape)
print(columnas)

(142, 9)
{0: 'IMAGE_FILE', 1: 'DESCRIPTION', 2: 'AUTHOR', 3: 'TITLE', 4: 'TECHNIQUE', 5: 'DATE', 6: 'TYPE', 7: 'SCHOOL', 8: 'TIMEFRAME'}
```

Luego obtuve los cuadros de los que habla la información que acababa de aprender:

```
def guardarCuadros(pathDirectorio,dataset):
    pathsCompletos = pathDirectorio+'/*'
    cuadrosDistintos = np.unique(dataset[:,0])
    cuadros = []
    nombreCuadros = []
    for nombreCuadro in cuadrosDistintos:
        nombreCuadros.append(dataset[dataset[:,0] == nombreCuadro,3])
        cuadro = cv2.imread(pathDirectorio+'/'+nombreCuadro)
        cuadros.append(cuadro)
    nombreCuadros = np.array(nombreCuadros).ravel()
    return cuadros,nombreCuadros

cuadros, nombreCuadros = guardarCuadros('SemArt/Images',dataset)
```

## PCA (Principal Components Analysis):

Para poder distinguir las características de las obras entre sí más claramente y clasificarlas mejor he decidido expresar los píxeles en una nueva base en la que los primeros elementos contienen el mayor porcentaje de la información. Este proceso es conocido como PCA. Además nos viene muy bien porque no todos los cuadros tienen la misma dimensión, hay algunos más grandes que otros. Al aplicar PCA como los primeros elementos tienen la mayor parte de la información, si no guardamos los últimos elementos no perdemos mucha información. Además nos interesa normalizar los datos, ya que el algoritmo K-Means intenta minimizar las distancias entre los centroides y los ejemplos, si utilizamos una distinta escala en las variables la distancia es muy dependiente del rango de valores en el que se mueven las variables, lo cual distorsiona la realidad. Esta es la implementación:

```
def aplicaPCACuadros(cuadros,nComponentes):
    pca = PCA(n_components=nComponentes)
    PCAs = []
    suma = 0
    for cuadro in cuadros:
        pca.fit(cv2.cvtColor(cuadro, cv2.COLOR_BGR2GRAY))
        PCAs.append(pca.singular_values_)
        suma += np.sum(pca.explained_variance_ratio_)
    print('De media mantenemos el {}% de la información de los cuadros'.format(round(suma/len(cuadros)*100)))
    PCAs = np.array(PCAs)
    return PCAs

nComponentes = 80
PCAs = aplicaPCACuadros(cuadros,nComponentes)
PCAs = StandardScaler().fit_transform(PCAs)
print(PCAs.shape)

De media mantenemos el 95.0432% de la información de los cuadros
(142, 80)
```

He elegido que se mantengan 80 componentes de la nueva base, ya que perdemos solo un 5% de la información.

## Elegir el número de clusters:

Para elegir en cuántas clases dividir las obras he usado el coeficiente de silueta, el cual es un valor entre 1 y -1 que expresa cuán de parecidos son los cuadros al representante de su cluster. Cuanto más cerca de 1 más se parece y cuanto más cerca de -1 menos. Probé con diferentes número de clusters y me quedé con aquel que tuviese un coeficiente de silueta más alto. Pero antes he dividido los cuadros por categorías. Aquí está cómo lo he hecho:

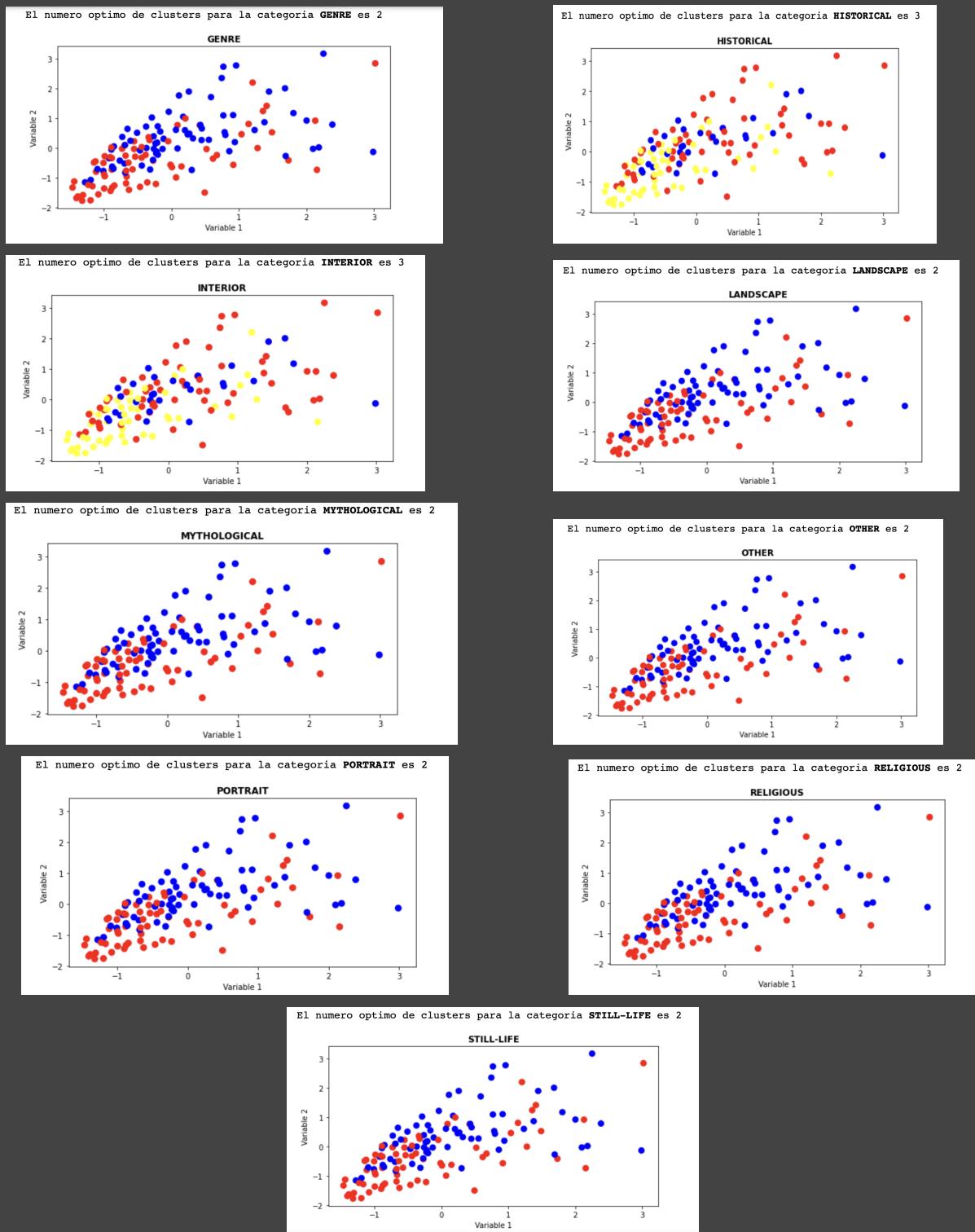
```
def cogeCuadrosTipo(tipo, nombreCuadros, PCAs):
    nombresCuadrosDelTipo = dataset[dataset[:,6] == tipo, 3]
    indices = np.zeros((nombreCuadros.size)).astype('bool')
    for nombreCuadroDelTipo in nombresCuadrosDelTipo:
        indices = indices+(nombreCuadroDelTipo == nombreCuadros)
    cuadrosDelTipo = PCAs[indices]
    return cuadrosDelTipo

#creo una lista de cuadros separados por categorias
categorias = np.unique(dataset[:,6])
cuadrosPorCategorias = []
for categoria in categorias:
    cuadrosPorCategorias.append(cogeCuadrosTipo(categoria, nombreCuadros, PCAs))

def eligeNumeroClusters(nComponentes, cuadros):
    puntuacionesSilueta = np.zeros((nComponentes-2))
    for k in range(2,nComponentes):
        kMeans = KMeans(n_clusters = k, random_state = 42).fit(cuadros)
        puntuacionesSilueta[k-2] = silhouette_score(cuadros,kMeans.labels_)
    kOptimo = np.argmax(puntuacionesSilueta)+2
    return kOptimo

cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#FFFF00', '#0000FF'])
for i,cuadros in enumerate(cuadrosPorCategorias):
    kOptimo = eligeNumeroClusters(min(len(cuadros), nComponentes), cuadros)
    print('El numero optimo de clusters para la categoria \033[1m{}\033[0m es {}'.format(categorias[i], kOptimo))
    kMeans = KMeans(n_clusters = kOptimo, random_state = 42).fit(PCAs)
    plt.figure(figsize=(8,4))
    plt.subplot(111)
    plt.scatter(PCAs[:, 0], PCAs[:, 1], c=kMeans.labels_, cmap=cmap_bold, s=60)
    plt.xlabel('Variable 1')
    plt.ylabel('Variable 2')
    plt.title(categorias[i].upper(), fontweight="bold")
    plt.show()
```

# Clusters por categorías de cuadros:



En las fotos puede parecer que los clusters no son los idóneos, pero es porque los ejemplos están representados en dos dimensiones (solo vemos dos variables), cuando en realidad cada cuadro está formado por 80 variables. Es verdad que las dos primeras variables son las que más información de la base original tienen, pero aun así no son muy representativas

## **Conclusiones finales:**

Quitando las categorías “interior” e “historical”, que están divididas en tres clases, todas están partidas en dos clusters. Por lo que pondría, si fuese posible, tantas salas como clases han salido y metería en ellas los cuadros de cada cluster junto con su información en su respectiva sala. Si hubiese escasez de salas, intentaría distribuir el espacio por categorías y dentro de las categorías por clusters.

## **Bibliografía:**

- <https://scikit-learn.org/stable/>
- [https://es.wikipedia.org/wiki/Silhouette\\_\(clustering\)](https://es.wikipedia.org/wiki/Silhouette_(clustering))
- <https://numpy.org/doc/stable/reference/>
- <https://opencv.org/>
- <https://es.stackoverflow.com/>