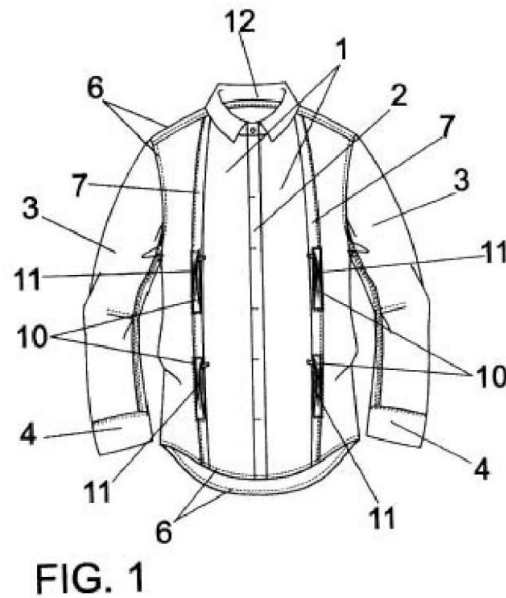


PROJECTE

ETIQUETADOR



Intel·ligència Artificial

Universitat Autònoma de Barcelona

Pablo Aguilar Ruiz - 1636546
Jaume Costa Calvoparra - 1636129
Youssef Cahouach Guella Ikhlaf - 1638618

Index

1. Introducció	3
2. Mètodes d'anàlisi	4
1. Implementació Retrieval_by_color	4
2. Implementació Retrieval_by_shape	4
3. Implementació Retrieval_combined	4
4. Implementació Kmean_statistics	5
5. Implementació get_shape_accuracy	5
6. Implementació get_color_accuracy	6
7. Experiments	7
7.1. Experiments d'anàlisi qualitatiu	7
7.2. Experiments d'anàlisi quantitatiu	9
3. Millores als mètodes de classificació	12
1. Inicialitzacions de Kmeans	12
2. Diferents heurístiques per BestK	13
2.1 Implementació interClassDistance	13
2.2 Implementació coeficientfisher	13
3. Find_BestK	14
3.1 Implementació find_bestk_update	14
4. Experiments	15
4. Conclusions	22

1. Introducció

L'equitatge d'imatges és un problema complex que requereix una solució eficaç. En aquest cas, donat un conjunt d'imatges d'un catàleg de roba desenvoluparem els algorismes que permetin aprendre a etiquetar automàticament imatges per tipus de peça i per color. Una manera d'abordar aquest problema és utilitzant algorismes de clustering com el **KMeans** i el **KNN**.

El que volem aconseguir amb el **KMeans** és trobar els agrupaments més importants del núvol de punts que ens donen com a mostra, la qual es tracta d'un espai de tres dimensions. Aquest algorisme pren com entrada: X (imatge que volem analitzar), K (número de clústers que s'utilitzen) i opcions addicionals (mètode d'init centroides, màxim número d'iteracions...). Les funcions que hem implementat per fer l'algorisme han sigut:

Funcions KMeans		
_init_options	_init_X	_init_centroid
distance	get_labels	get_centroids
converges	fit	withinClassDistance
find_bestK	get_color	

Respecte al KNN el que volem fer és una classificació supervisada que utilitzarem per assignar les etiquetes de tipus roba. Per poder realitzar la classificació de la forma de les imatges, el que fem és servir com a característiques els píxels de la imatge després de modificar-la a escala de grisos. Per representar la forma de la roba i assignar l'etiqueta de tipus de roba a la nova imatge, utilitzem `read_dataset()`. Per aprendre a classificar la roba, fem ús de **train_data** i finalment per assignar l'etiqueta de tipus de roba a una nova imatge utilitzem **predict()**. Les funcions que hem implementat per fer l'algorisme han sigut:

Funcions KNN			
_init_train	get_k_neighbours	get_class	predict

2. Mètodes d'anàlisi

1. Implementació Retrieval_by_color

La funció **retrieval_by_color** itera sobre les etiquetes i comprova si algun dels valors de les etiquetes està en la llista de colors a buscar. Si és així, afegeix la imatge corresponent a la llista **colorIm**. Finalment, retorna la llista **colorIm** amb totes les imatges que compleixen amb els criteris de cerca.

```
def retrieval_by_color(images, etiq, color):
    colorIm = []
    for i, valores in enumerate(etiq):
        if any(valor in color for valor in valores):
            colorIm.append(images[i])
    return colorIm
```

Fig 1. Codi sobre el retrieval_by_color

2. Implementació Retrieval_by_shape

La funció **retrieval_by_shape** itera sobre les etiquetes i comprova si algun dels valors de les etiquetes està en la forma a buscar. Si és així, afegeix la imatge corresponent a la llista **formaIm**. Finalment, retorna la llista **formaIm** amb totes les imatges que compleixen amb els criteris de cerca.

```
def retrieval_by_shape(images, etiq, forma):
    formaIm = []
    for i, valores in enumerate(etiq):
        if any(valor in forma for valor in valores):
            formaIm.append(images[i])
    return formaIm
```

Fig 2. Codi sobre el retrieval_by_shape

3. Implementació Retrieval_combined

La funció **retrieval_combined** itera sobre les etiquetes de forma i color al mateix temps utilitzant la funció zip. Comprova si algun dels valors de les etiquetes de forma està en la forma a buscar i si algun dels valors de les etiquetes de color està en el color a buscar. Si és així, afegeix la imatge corresponent a la llista **formaColorIm**. A més, amb el vector **ok** busca si és coincident amb el **GroundTruth**. Finalment, retorna la llista **formaColorIm** amb totes les imatges que compleixen amb els criteris de cerca.

```
def retrieval_combined(images, etiqC, etiqF, forma, color, infoF, infoC):
    info = []
    formaColorIm = []
    ok = []
    for i, (valoresC, valoresF) in enumerate(zip(etiqC, etiqF)):
        if (valoresF in forma) and any(valor in color for valor in valoresC):
            formaColorIm.append(images[i])
            valoresC[:] = list(set(valoresC))
            if infoF[i] == valoresF and any(valorC in color for valorC in infoC[i]):
                ok.append(1)
            else:
                ok.append(0)
            info.append([infoC[i], infoF[i]])
    formaColorIm = np.array(formaColorIm)
    return formaColorIm, ok, info
```

Fig 3. Codi sobre el retrieval_combined

4. Implementació Kmean_statistics

La funció `Kmean_statistics` rep com a entrada la classe **Kmeans** amb un conjunt d'imatges i un valor **Kmax** que representa la màxima K que es vol analitzar. En aquest cas, es declaren tres llistes: **WCDs**, **num_iterations** i **total_times**. Després, es recorre un bucle **for** des de **k=2** fins a **Kmax + 1** i per cada iteració es canvia el valor de K de la classe Kmeans pel valor de l'iteració actual, es guarda el temps d'inici en la variable **start_time**, s'executa la funció **fit** amb les imatges com a entrada, es guarda el temps final en la variable **end_time**, s'afegeix el valor de la funció **withinClassDistance** a la llista **WCDs**, s'afegeix el nombre d'iteracions a la llista **num_iterations** i s'afegeix el temps total a la llista **total_times**. Finalment, es retornen les tres llistes.

```
def Kmean_statistics(km, Kmax):
    WCDs = []
    num_iterations = []
    total_times = []
    for K_iteration in range(2, Kmax + 1):
        km.K = K_iteration
        start_time = time.time()
        km.fit()
        end_time = time.time()
        WCDs.append(km.withinClassDistance())
        num_iterations.append(km.num_iter)
        total_times.append(end_time - start_time)
    return WCDs, num_iterations, total_times
```

Fig 4. Codi sobre el Kmean_stadistic

5. Implementació get_shape_accuracy

La funció **get_shape_accuracy** rep com a entrada les etiquetes que s'han obtingut en aplicar el **KNN** i el **Ground-Truth** d'aquestes. En aquest cas, declarem una variable **correct_shape** que comença amb un valor de 0. Després, es recorren les etiquetes i el Ground-Truth amb un bucle **for** i es comparen les **etiquetes (labels)** amb el **Ground-Truth**. Si coincideixen, s'incrementa en 1 el valor de **correct_shape**. Finalment, es calcula la precisió dividint **correct_shape** pel nombre total d'etiquetes i es retorna aquest valor.

```
def get_shape_accuracy(knn_labels, ground_t):
    correct_shape = 0
    for label, gtruth in zip(knn_labels, ground_t):
        if label == gtruth:
            correct_shape += 1
    accuracy = correct_shape / len(knn_labels)
    return accuracy
```

Fig 5. Codi sobre el get_shape_accuracy

6. Implementació get_color_accuracy

La funció **get_color_accuracy** rep com a entrada les etiquetes que s'han obtingut en aplicar el **KMeans** i el **Ground-Truth** d'aquestes. En aquest cas, es declaren dos conjunts **km_label** (etiquetes trobades) i **gt_label** (etiquetes del GT) que contenen les etiquetes sense duplicar. Després, es calcula la intersecció entre **km_label** i **gt_label** i es guarda en la variable **intersection_labels**. Després es fa el mateix però calculant la unió entre aquests. Finalment, es calcula la precisió dividint la longitud de la intersecció per la longitud de l'unió i es retorna aquest valor. En aquest cas hem aplicat la mesura de similitud que vam veure a classe per obtenir la precisió del color:

Intersecció / Unió

Fig 5. Mesura de similitud implementada

```
def get_color_accuracy(kmeans_labels, ground_t):
    correct_colors = 0
    total_colors = 0
    for km_label, gt_label in zip(kmeans_labels, ground_t):
        union_labels = set(km_label).union(set(gt_label)) # Unió entre kmlabels y groundtruth
        intersection_labels = set(km_label).intersection(set(gt_label)) # Intersecció entre kmlabels i groundtruth
        correct_colors += len(intersection_labels) / len(union_labels) # Divisió entre intersecció i unió
        total_colors += 1
    percent_accuracy = (correct_colors / total_colors) * 100
    return percent_accuracy
```

Fig 6. Codi sobre el get_color_accuracy



7. Experiments




7.1. Experiments d'anàlisi qualitatiu

1r Experiment:

Hem realitzat un experiment per avaluar el rendiment del nostre codi utilitzant la funció `retrieval_combined`. En aquest cas, aquest experiment es farà sobre el programa base on la $K = 2$ en el KNN i en el Kmeans $K = 4$ i finalment, la $K_{max} = 10$ i fent ús d'un llindar del 20%. No hem utilitzat les funcions `retrieval_by_color` ni `retrieval_by_shape`, ja que `retrieval_combined` les inclou.

A l'hora d'analitzar els resultats, fem ús de la funció ***visualize_retrieval*** on obtenim les imatges que escolleix els nostres algorismes i també indica si les imatges escollides coincideixen amb l'objectiu.

Objectiu	Precisió
Black Flip flops	<p>Busqueda: Black Flip Flops</p> <p>[['Blue', 'Brown', 'Flip Flops'], [['Black', 'Blue', 'Flip Flops'], ['Grey', 'White', 'Flip Flops'], ['Blue', 'Purple', 'Flip Flops']]</p>  <p>[['Black', 'Red', 'Flip Flops'], ['Blue', 'Green', 'Flip Flops'], ['Black', 'Yellow', 'Flip Flops'], ['Black', 'Red', 'Flip Flops']]</p> <p>[['Black', 'Red', 'Flip Flops'], ['Blue', 'Red', 'Flip Flops'], ['Black', 'Red', 'Flip Flops'], ['Black', 'White', 'Flip Flops']]</p> <p>5/10</p>
Pink Dresses	<p>Busqueda: Pink Dresses</p> <p>[['Pink', 'Dresses'], [['Green', 'Pink', 'White', 'Dresses'], ['Pink', 'White', 'Dresses']], ['Pink', 'Dresses']]</p>  <p>[['Blue', 'Pink', 'Purple', 'Dresses'], ['Black', 'Grey', 'Dresses'], ['Black', 'Grey', 'Pink', 'Dresses'], ['Black', 'Grey', 'Pink', 'Dresses']]</p> <p>[['Pink', 'Dresses'], [['Black', 'Purple', 'Dresses']], ['Pink', 'Dresses']]</p> <p>8/10</p>

<p>Blue Jeans</p>	<p>Busqueda: Blue Jeans</p>  <p>9/10</p>
<p>Brown Heels</p>	<p>Busqueda: Brown Heels</p>  <p>8/10</p>
<p>Purple Shirts</p>	<p>Busqueda: Purple Shirts</p>  <p>8/10</p>

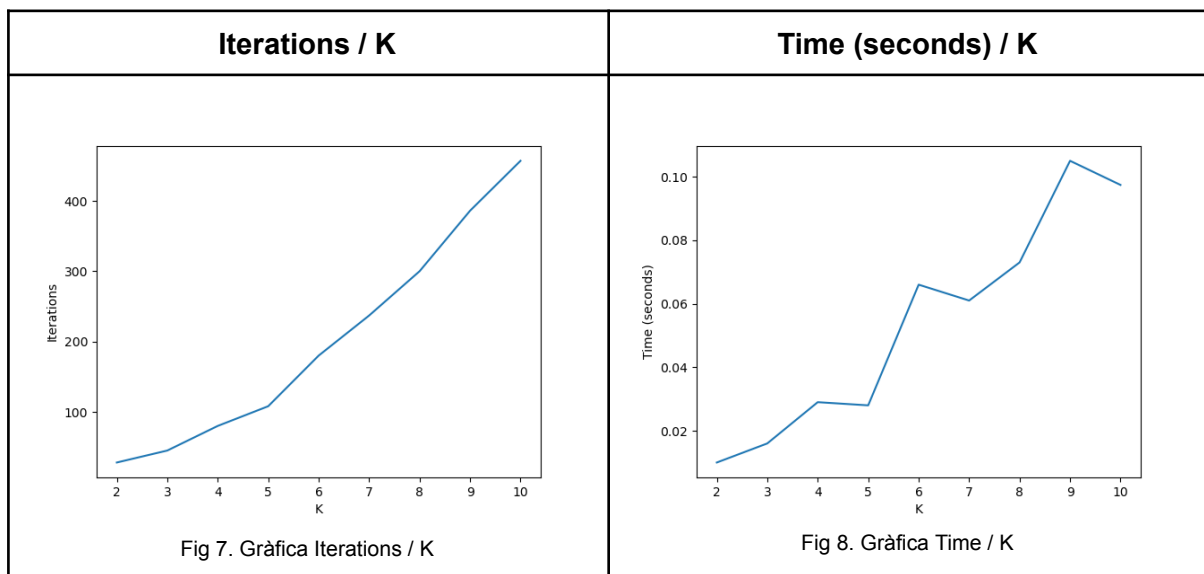
En aquest cas, hem aplicat el nostre algoritme a 5 peces de roba de diferents colors. Per cada cerca, hem obtingut 10 peces i gràcies al groundTruth hem sabut si els resultats eren correctes. La cerca amb menys encerts és la de les Flip Flops. Creiem que aquest error es deu al fet que les Flip Flops tenen molts colors irregulars i per això tenim tants errors només en aquesta cerca. Aquesta hipòtesi es veu reforçada quan descobrim que en el color

accuracy obtenim un 43,5%. mentre que en altres cerques amb menys varietat de colors obtenim molt bons resultats , 8/10 i 9/10.

7.2. Experiments d'anàlisi quantitatiu

1r Experiment:

Hem realitzat un experiment en els nostres algorismes utilitzant la funció **kmean_statistics** per determinar el valor ideal de **k** quan utilitzem l'heurística **intra-class**. Hem provat diferents valors de **k** des de 2 fins a un màxim de 10 i hem mesurat el temps de convergència, la relació entre WCD i **k**, i el nombre d'iteracions en relació amb els valors de **k**. Aquest experiment s'ha aplicat a 150 imatges i aquests són els resultats obtinguts:



Podem observar en la primera gràfica, com el nombre d'iteracions augmenta de forma lineal a mesura que augmenta el valor de **k**. Això és degut a que a mesura que augmentem el valor de **K**, haurem de fer més iteracions sobre cada imatge analitzada.

Pel que fa al temps d'execució, podem observar que es formen grans pics quan s'aplica **K = 6** o **K = 9**, també podem observar aquesta situació però a l'inversa on apliquem una **K = 5**, de forma que obtenim un temps que és menor que quan apliquem **K = 4**. Això pot ser degut a altres factors que poden afectar el temps d'execució, com la inicialització dels centroides i la convergència de l'algoritme. Per tant, és possible que en alguns casos el temps d'execució sigui similar o menor per a diferents valors de **K** i diferents nombres d'iteracions.

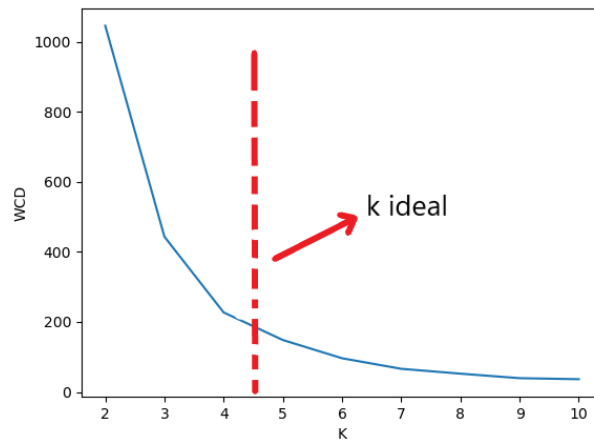


Fig 9. Gràfica WCD / K

Finalment, a l'hora d'analitzar la relació que hi ha entre la distància **withinClassDistance** i els diferents valors de K, podem utilitzar la gràfica resultant per trobar un valor ideal de K a l'hora d'aplicar l'heurística Intra-Class. En la pròpia gràfica indiquem el valor ideal de K on la corba comença a suavitzar-se o aplanar-se. En aquest cas el punt es troba entre dos valors de K. Podem triar qualsevol dels dos valors K = 4 o K = 5, encara que generalment es tria el valor més petit.

2n Experiment:

Hem realitzat un altre experiment per analitzar la precisió del nostre algoritme per a diferents valors de K en la detecció del color correcte de les peces de roba, que en aquest cas hem fet ús de 150 imatges. Hem utilitzat el procediment de similitud que consisteix en dividir la intersecció entre els colors de les imatges i els colors del ground_truth pel total de colors (unió) presents en ambdós conjunts.

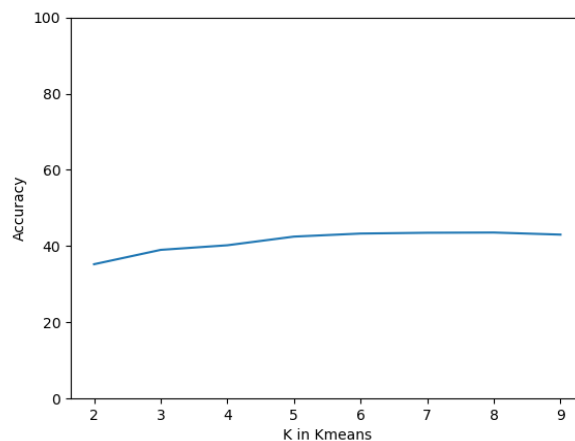


Fig 10. Gràfica de precisió de colors

En aquest cas la gràfica mostra que a mesura que augmenta el valor de K, la precisió augmenta molt poc i deixa d'augmentar en K = 9. Això ens pot indicar que valors més grans de K poden no proporcionar una millora significativa en la precisió i poden augmentar la complexitat i el temps d'execució de l'algoritme. Per tant, cal aplicar una millora en el Kmeans, ja sigui utilitzant una altra heurística com per exemple inter-class o fisher, o modificant alguna funció del KMeans.

3r Experiment:

En aquest experiment hem analitzat la precisió que té el nostre algorisme KNN per detectar la forma exacta de totes les peces de roba aplicant diferents valors de K. Aquest experiment ha sigut aplicat sobre 500 imatges en total i hem obtingut el següent resultat en forma de gràfica:

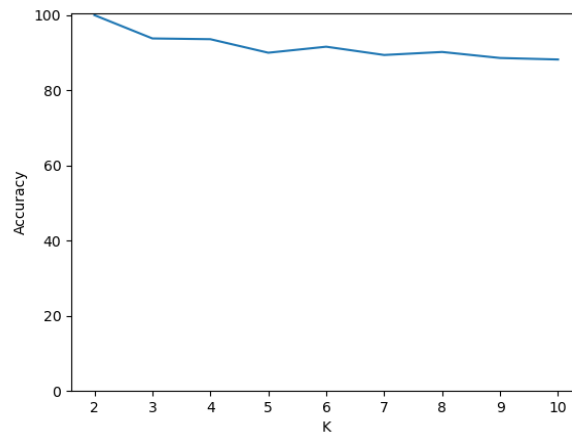


Fig 11. Gràfica de precisió de forma

Podem observar que a l'hora d'aplicar una K amb valor 2 obtenim una precisió del 100% però a mesura que augmentem el valor de K, aquesta precisió disminueix progressivament de forma lineal. Un valor de K molt petit pot fer que el model sigui sensible a les variacions en les dades d'entrenament, mentre que un valor de K molt gran pot fer que el model sigui menys precís ja que s'inclouen veïns que estan més lluny els qual poden no ser rellevants per a la classificació. Per tant, el valor $K = 2$ és ideal per la identificació de la forma de les peces de roba.

3. Millores als mètodes de classificació

1. Inicialitzacions de Kmeans

Hem creat dues opcions noves en la inicialització del *init_centroids*. En primer lloc, la primera inicialització que hem implementat ha sigut el “**random**” en la qual, tots els punts s’inicien aleatòriament.

```
elif self.options['km_init'].lower() == 'random':
    indices_aleatorios = np.random.choice(self.X.shape[0], size=self.K, replace=False)
    self.centroids = self.X[indices_aleatorios]
```

Figura 12. Implementació init_centroids random

En segon lloc, hem implementat una inicialització inversa al “**first**” on els punts s’inicien pel final.

```
elif self.options['km_init'].lower() == 'last':
    # Asignar la última fila de self.X a la primera fila de centroids
    self.centroids[0] = self.X[-1]

    # Inicializar contador
    contK = 0

    # Iterar sobre las filas de self.X en orden inverso
    for fila in self.X[-2::-1]:
        # Verificar si la fila actual está en centroids
        esta_en_centroids = np.any(np.all(fila == self.centroids[:contK + 1], axis=1))

        if not esta_en_centroids:
            # Si la fila no está en centroids, aumentar el contador contK y asignar la fila a centroids
            contK += 1
            self.centroids[contK] = fila

    # Verificar si se han alcanzado K filas únicas en centroids
    if contK == self.K - 1:
        break
```

Figura 13. Implementació init_centroids last

```
km = KMeans(imgs[i], k_actual, options={'km_init': 'first'})
```

Figura 14. Inicialització amb opció km_init first

2. Diferents heurístiques per BestK

2.1 Implementació interClassDistance

La funció **interClassDistance** calcula la distància inter-class, és a dir, la distància promig que tenim entre els punts d'un clúster i els punts d'un altre clúster. La variable **dist** emmagatzema la distància interclass total i es calcula amb les distàncies al quadrat entre els punts del cluster i la resta de centroides.

```
def interClassDistance(self):
    dist = 0
    N = len(self.X)
    for i in range(len(self.centroids)):
        points_cluster = self.X[self.labels == i]
        if len(points_cluster) > 0:
            other_centroids = self.centroids[np.arange(len(self.centroids)) != i]
            for c in other_centroids:
                dist += np.sum((points_cluster - c) ** 2)
    inter_class_dist = dist / N
    return inter_class_dist
```

Figura 14. Implementació interClassDistance

2.2 Implementació coeficientfisher

La funció **coeficientfisher** calcula el coeficient de fisher, el qual es calcula mitjançant la divisió de la distància **intra-class** entre la distància **inter-class**.

```
def coeficientfisher(self):
    return self.withinClassDistance() / self.interClassDistance()
```

Figura 15. Implementació fisher

3. Find_BestK

3.1 Implementació find_bestk_update

La funció **find_bestk_update** és semblant a la funció original de find_bestk, però a diferència d'aquesta última, en la nova funció podem treballar amb les distàncies de intra-class, inter-class o el coeficient de fisher i a l'hora podem anar modificant el valor del llindar per trobar el cas òptim. Primer es selecciona el mètode que s'ha escollit i de forma iterativa s'aconsegueix trobar el millor valor de K dins del llindar també seleccionat.

```
def find_bestk_update(self, max_K, heuristica, llindar):
    self.fit() # Fem el fit.
    if heuristica == "Fisher":
        auxiliar = self.coeficientfisher() # Fisher
    elif heuristica == "Inter":
        auxiliar = self.interClassDistance() # Inter-class
    elif heuristica == "Intra":
        auxiliar = self.withinClassDistance() # Intra-class
    else:
        return -1

    x = False
    self.K += 1
    while self.K <= max_K and not x:
        self.fit()
        if heuristica == "Fisher":
            w = self.coeficientfisher() # Fisher
            percentatgeW = (w / auxiliar) * 100
        elif heuristica == "Inter":
            w = self.interClassDistance() # Inter-class
            percentatgeW = (auxiliar / w) * 100
        else:
            w = self.withinClassDistance() # Intra-class
            percentatgeW = (w / auxiliar) * 100

        if (100 - percentatgeW) < llindar:
            x = True
            break
        else:
            auxiliar = w
            self.K += 1 # Provar un altre valor de K

    if not x:
        self.K = max_K

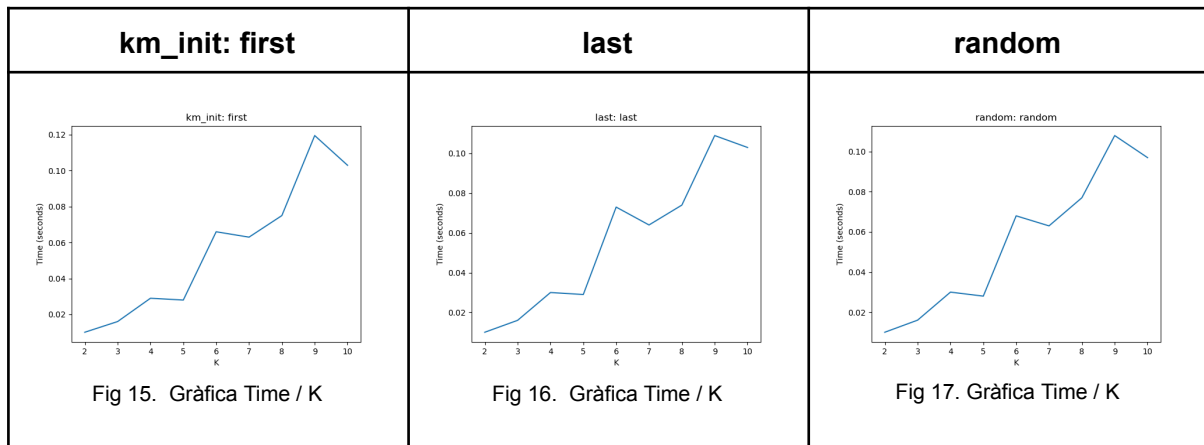
    self.fit()
```

Figura 16. Implementació find_bestk_update

4. Experiments

1r Experiment:

Per fer els experiments amb les diferents opcions d'inicialització del kmeans, utilitzem la funció `Kmeans_statistics`. Aquesta funció ens retorna 3 gràfiques, però només utilitzarem la de segons/k. Provarem les 3 maneres diferents d'inicialitzar i mirarem quina és més ràpida que les altres.



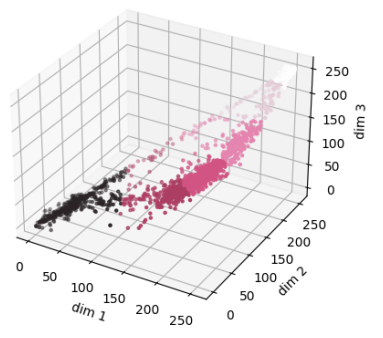

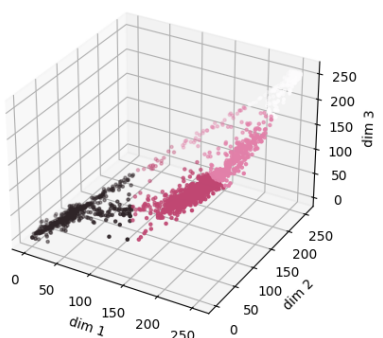

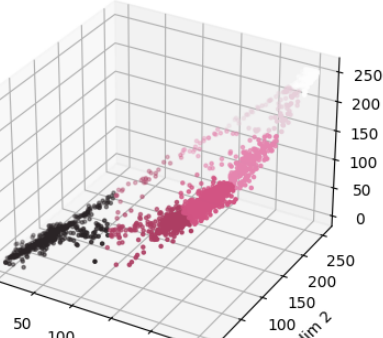




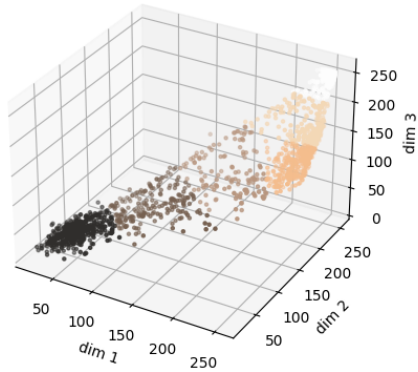

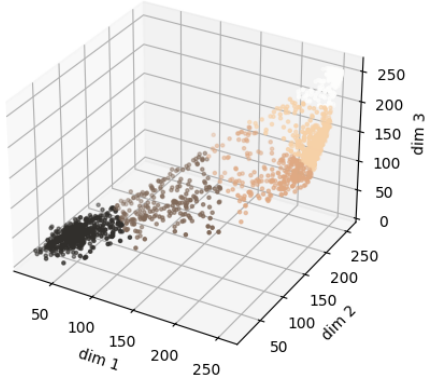

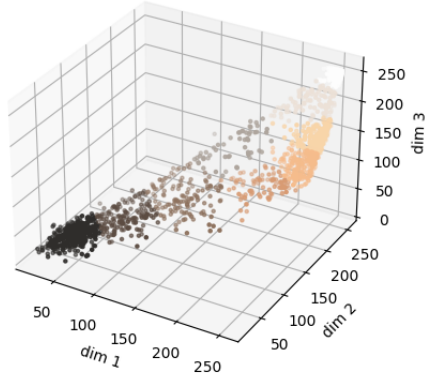
En resum, podem concloure que hi ha una certa variació, però no és prou significativa per determinar si hi ha hagut un temps d'execució major o menor. No obstant això, si observem la gràfica amb detall, podríem apreciar una certa millora en l'opció 'last'.



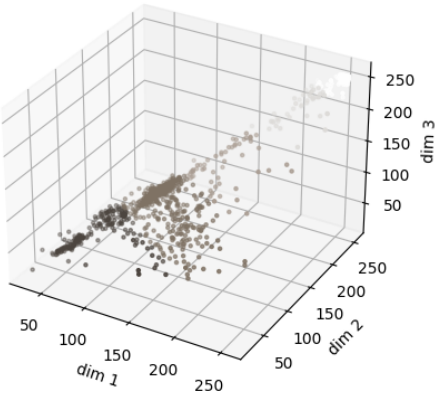

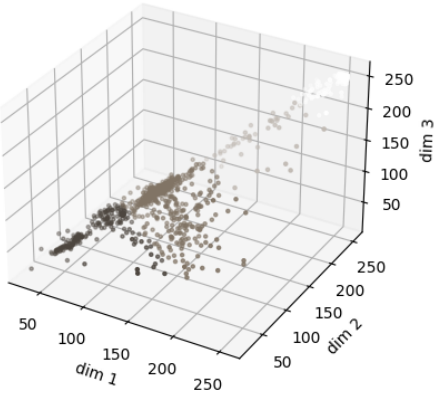

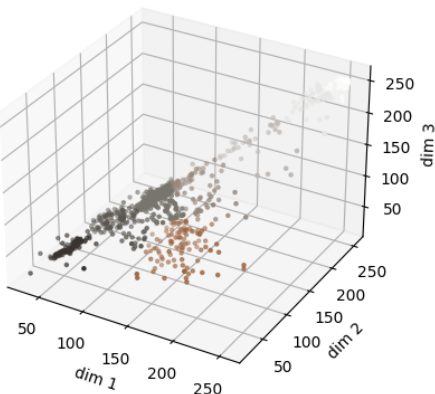
2n Experiment:



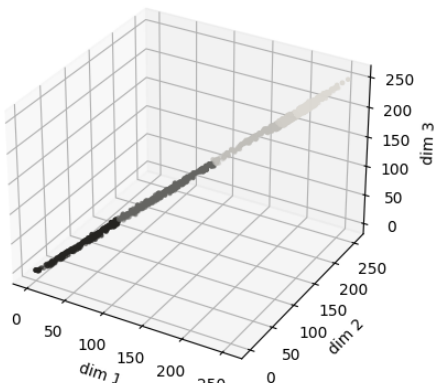

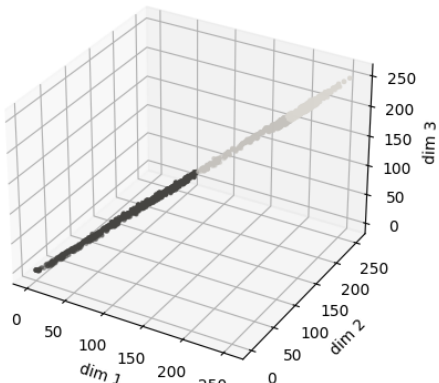

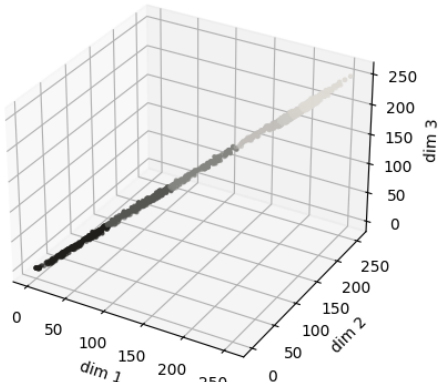
Per tal de demostrar l'eficàcia de quina heurística és millor dins de la funció `find_bestk_update` (inter-class, intra-class o fisher), hem realitzat un experiment on hem estudiat diverses imatges amb les diferents heurístiques, comprovant quina del les tres ofereix una qualitat superior. El valor de la `KMax` es manté constant a 10 i passem un valor inicial de `K=2`, aquest valor a l'hora del resultat no importa gaire perquè la funció `find_bestK_update` serà l'encarregada de cercar la `K` ideal, però inicialment necessita un valor per realitzar la primera comprovació. Finalment, en tots els casos apliquem un llinda amb valor 20.

Aplicant aquest estudi, obtenim els següents resultats:

Imatge Original	Imatge Resultat	Núvol de Punts
	Intra-class 	
	Inter-class 	
	Fisher 	

Imatge Original	Imatge Resultat	Núvol de Punts
	Intra-class 	
	Inter-class 	
	Fisher 	

Imatge Original	Imatge Resultat	Núvol de Punts
	Intra-class 	
	Inter-class 	
	Fisher 	

Imatge Original	Imatge Resultat	Núvol de Punts
	Intra-class 	
	Inter-class 	
	Fisher 	

Observant les diferents taules amb els resultats que ens ha generat el programa, podem afirmar que en tots els casos el millor mètode que podem utilitzar per aconseguir el millor resultat és el **coeficient de Fisher**.

3r Experiment:


Un cop sabem que la millor opció per trobar la millor imatge és utilitzant el coeficient de fisher. Farem un estudi per saber quin rang de valors de llindar és més òptim a l'hora de definir la imatge, per realitzar aquest experiment es té en compte la qualitat de les imatges, el temps que ha trigat en realitzar-les i el percentatge de accuracy obtingut. Agafarem una mostra de 150 imatges, amb un valor inicial de $K=2$, un $\text{max_K}=10$ i finalment com a mesura agafem el coeficient de Fisher.

Amb aquest paràmetres obtenim els següents resultats:


Imatge Original:



Valor Llindar	Temps (s) / Accuracy (%)	Imatge Kmeans
100	7.83 / 39,00	
80	9.32 / 38,07	
60	15.72 / 40,78	
40	28.74 / 41,51	
20	62.25 / 42,71	

15	85.41 / 42,65	
----	---------------	---

Observem que el temps amb l'increment del llindar és directament proporcional. Quan el llindar disminueix el seu valor, el temps augmenta i també a l'inversa. Aquest fet és degut a que el `find_bestK_update` amb un valor de llindar elevat té un rang més elevat de valors de K acceptats que no pas un valor de llindar petit, per tant com a conseqüència com més petit és el "rang" de k acceptat, més iteracions hem de fer a la funció per trobar la K i a l'hora més temps impliquem. Amb els resultats anteriors hem demostrat que la gran diferència de temps es troba entre llindar = 40 (28.74s) i llindar = 20 (62.25s), podem apreciar que la diferència de resultat entre les dues imatges no és gran significant i ademés el percentatge de accuracy obtingut en llindar=15 és més petit que en llindar=20 ($42,65 < 42,71$), per tant hem d'estudiar un valor de llindar promig superior al de 20. En el nostre cas, estudiem el promig dels dos llindars entre 20 y 40 (llindar = 30) per identificar si es tractaria de la millor opció tant en temps com en qualitat de la imatge:

Valor Llindar	Temps (s)	Imatge Kmeans
30	42.24 / 42,67%	

Efectivament podem comprovar que la nostre hipòtesis era certa, el millor valor de llindar que podem aplicar per trobar l'equilibri entre qualitat de les imatges, temps que es triguen en generar-les i percentatge de accuracy, és amb el **llindar = 30** o un rang de **llindar=30-40**. El temps es redueix considerablement en relació a llindar=20 (62.25s a 42.24s) i el percentatge de accuracy obtingut en aquest últim és molt semblant a l'altre ($42,67\%/42,71\%$). Amb aquest experiment també arribem a la conclusió de que quan més qualitat vulguem, més petit serà aquest llindar (sense disminuir del 20), però a l'hora més temps i recursos consumirem.

4. Conclusions

Després d'aplicar els algoritmes de clustering KMeans i KNN en el nostre conjunt de dades d'imatges de roba, hem pogut comprovar que ambdós algoritmes tenen una gran capacitat d'etiquetatge en el referent a tipus de peça i color, però aquesta no arriba a ser gairebé del tot efectiva perquè, com es mostra als experiments, no s'aconsegueix arribar a etiquetar bé el 100% de les imatges.

Els principals problemes que vam trobar va ser l'equitetatge de colors utilitzant l'algoritme KMeans. La precisió més alta que vam aconseguir en analitzar els colors de les peces de roba va ser del 43,5%, utilitzant un valor de K de 8 per al KMeans. Aquesta falta de precisió es pot veure en el primer experiment amb ***retrieval_combined***.

Aquesta pràctica ens ha fet millorar diversos aspectes del nostre aprenentatge, tant a nivell de pràctica, com en l'àmbit educatiu referent a programació. Hem après a desenvolupar les nostres habilitats en python i numpy, hem pogut aprendre a tractar amb algorismes de clustering aplicat sobre un conjunt d'imatges i après a traduir les gràfiques que ens donaven aquests algorismes per obtenir informació rellevant per als nostres experiments. Hem millorat la nostra capacitat d'anàlisis respecte a funcions donades: entendre que fan, com cridar-les, correcció d'errors al utilitzar-les... que ens ha sigut molt útil a l'hora de realitzar els experiments. Finalment, hem après a implementar codi nou sense tenir un test que ens comprovi la funció, a través dels experiments hem comprovat que les nostres funcions estiguin fetes correctament i a l'hora hem demostrat les millores del nostre projecte.

El nostre projecte encara té marge de millora. Hem demostrat que els resultats en el KNN són molt bons, arribant a un 100% d'accuracy amb $K = 2$. Però, com hem demostrat abans, el Kmeans encara dona un possible marge de millora, així doncs creiem que el que hauríem de fer són modificacions en l'algorisme per aconseguir que es pugui millorar el codi respecte el referent a discernir els colors.