Università di Genova | DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

SCUOLA POLITECNICA

Anno accademico 2020/2021

Machine Learning & Data Analysis Final Project:

Studio delle diverse performance ottenute nella risoluzione di un problema di Image Classification con l'utilizzo di algoritmi classici del Machine Learning (SVM, KNN, Decision Tree e Random Forest)

Simone Cella S4334970
Matteo Spinaci S4338765

Premessa:

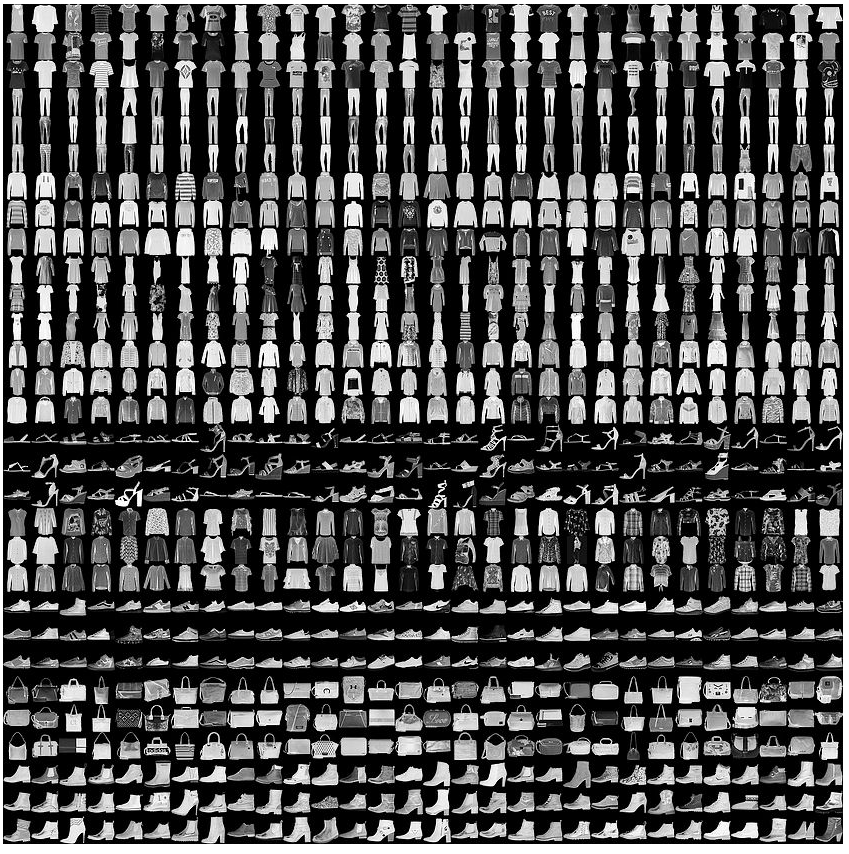
Durante il corso di Machine Learning & Data Analysis abbiamo visto e analizzato diversi algoritmi adatti alla risoluzione di *Image-Classification Problems.* Il nostro obiettivo è di analizzare il loro margine di miglioramento attraverso il tuning degli hyperparameters e la riduzione del numero di features attraverso la PCA usando come dataset di riferimento il Fashion MNIST Dataset.
L'intero progetto è stato realizzato grazie ai tool messi a disposizione da *colab.research.google.com*

Fashion-MNIST Dataset:

Fashion-MNIST è un dataset di dati delle immagini degli articoli di Zalando, costituito da:

- 60'000 esempi per il Training set
- 10'000 esempi per il Test set

Ogni esempio è un'immagine in scala di grigi 28x28 [pixel], associato a un'etichetta di 10 classi.
ecco come appare il dataset in esame:



(possiamo notare come ogni classe prende tre righe)

Come abbiamo detto ci sono 10 classi, le quali andranno suddivise in questo modo:

0. T-shirt/Top
1. Trouser
2. Pullover
3. Dress
4. Coat
5. Sandal
6. Shirt
7. Sneaker
8. Bag
9. Ankle boot

60'000 immagini verranno utilizzate per addestrare gli algoritmi e le restanti 10'000 per valutare la precisione con la quale siamo riusciti a classificare le immagini.
Dopo aver importato tutte le librerie necessarie e dopo aver creato la funzione **plot_confusion_matrix,** la quale ci servirà per plottare la Confusion Matrix, per questioni di comodità utilizzeremo il servizio fornito da TensorFlow per importare il Dataset in esame.

```
from keras.datasets import fashion_mnist

(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
print("Shape of x_train: {}".format(x_train.shape))
print("Shape of y_train: {}".format(y_train.shape))
print()
print("Shape of x_test: {}".format(x_test.shape))
print("Shape of y_test: {}".format(y_test.shape))
```

```
-->Shape of x_train: (60000, 28, 28)
-->Shape of y_train: (60000,)

-->Shape of x_test: (10000, 28, 28)
-->Shape of y_test: (10000,)
```

Il caricamento del set di dati restituisce 4 array NumPy:

- Gli array *train_images* e *train_labels* sono l' *insieme di addestramento* , i dati che il modello utilizza per apprendere.

- Il modello è testato "contro" *l'insieme di test,* i *test_images* e *test_labels* array.

Le immagini sono array NumPy 28x28, con valori compresi tra 0 e 255(scala di grigi). Le etichette sono un array di numeri interi, compresi tra 0 e 9. Questi corrispondono alla *classe* di abbigliamento che l'immagine rappresenta.

```
labelNames = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

Ogni immagine è mappata su una singola etichetta. Poiché i nomi delle classi non sono inclusi nel set di dati li memorizzeremo noi in un array di stringhe (*labelNames*).

Per verificare che i dati siano nel formato corretto e che siamo pronti a lavorare con essi visualizziamo 25 immagini casuali del *set di addestramento* e mostriamo il nome della classe sotto ogni immagine.

```
plt.figure(figsize=(10, 10))
for i in range(25):
    temp = random.randint(0, len(x_train)+1)
    image = x_train[temp]
    plt. (10,10, i+1)
    plt.imshow(image, cmap='gray')
    plt.xticks([])
    plt.yticks([])
    plt.xlabel(labelNames[y_train[temp]])
    plt.tight_layout()
plt.show()
```

Proseguiamo ora a normalizzare i dati ottenuti:

```
# Normalize the data
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255.0
x_test /= 255.0
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1] * x_train.shape[2])
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1] * x_test.shape[2])
print(x_train.shape)
print(x_test.shape)
```

```
-->(60000, 784)
-->(10000, 784)
```

Ora che abbiamo preparato il nostro Dataset possiamo inizializzare gli algoritmi che andremo ad esaminare:

1. ***SVM , Support Vector Machine***
2. ***K-NN , K-Nearest Neighbors***
3. ***Decision Tree***
4. ***Random Forest***

1. SVM - Support Vector Machine Algorithm:

Utilizza degli hyper-planes per separare i samples in due o più categorie.
Per poter trovare l'iperpiano di separazione sono necessarie le funzioni Kernel (Φ), e la migliore funzione Φ è spesso una combinazione di funzioni kernel.
Le implementazioni più famose utilizzano diversi tipi di kernel:

- Linear
- Polynomial
- Radian Basis Function (rbf)
- Sigmoid

Un iperpiano è in grado di effettuare una separazione tra sole due classi, il nostro problema presenta 10 classi e quindi l'algoritmo si adatta (riducendo il problema a più problemi di separazione a 2 grazie a due approcci diversi:

- OVA (one versus all)
- OVO (one versus one)

L' algoritmo SVC implementato da scikit-learn utilizza l'approccio OVO.

Nel caso in un cui non sia possibile trovare un iperpiano di separazione (non separable problem) l'algoritmo usa il Kernel Trick per aumentare di uno la dimensione spaziale del dataset.
Aggiungendo una dimensione è più facile per l'algoritmo costruire un iperpiano e quindi rendere il problema separabile.

Ecco nello specifico il codice dei 3 algoritmi e il codice per report e analisi:

```
# SVC1:
svc = SVC(C=1, kernel='linear', gamma="auto")
svc.fit(x_train, y_train)
# SVC2:
svc2 = SVC(C=10, kernel='rbf', gamma="auto")
svc2.fit(x_train, y_train)
# SVC3:
svc3 = SVC(C=1, kernel='poly', gamma="auto")
svc3.fit(x_train, y_train)
```

```
# SVM report and analysis
y_pred_svc1 = svc.predict(x_test)
svc_f1 = metrics.f1_score(y_test, y_pred_svc1, average= "weighted")
svc_accuracy = metrics.accuracy_score(y_test, y_pred_svc1)
svc_cm = metrics.confusion_matrix(y_test, y_pred_svc1)
```

Tabella riassuntiva dei diversi algoritmi:

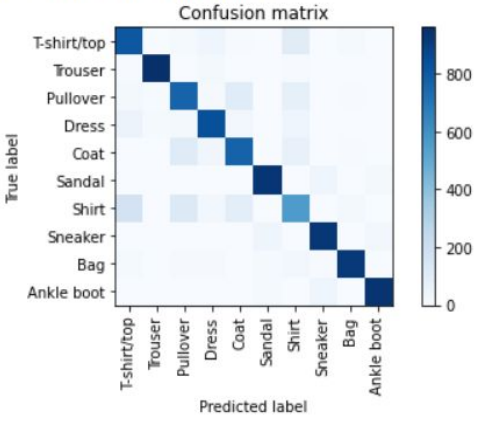
Name	Kernel	Accuracy	Training time
SVC1	{C=1, Kernel = linear, Gamma = auto}	84%	12.40 minute
SVC1.1	{C = 10, Kernel = linear, Gamma = auto}	84%	15 minute
SVC2	{C=10, Kernel = rbf, Gamma = auto}	87%	10.13 minute
SVC2.1	{C = 1, Kernel = rbf, Gamma = auto}	84%	12 minute

SVC3	{C=1, Kernel = poly, Gamma = auto}	66%	50.54 minute

Invece ora andremo a vedere nello specifico i diversi indici di performance e le diverse Confusion-Matrix relative alle diverse SVMx(x = 1,2,3).

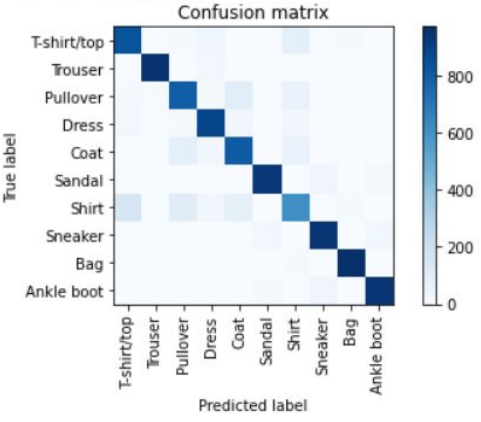
SVM1:

```
F1 score: 0.845599053028593
Accuracy score: 0.8463
Confusion matrix:
 [[815   2  13  45   4   1 108   0  12   0]
 [  6 962   2  22   3   0   4   0   1   0]
 [ 22   6 769   8 109   0  79   0   7   0]
 [ 54  15  19 842  27   0  40   0   3   0]
 [  1   2 114  33 773   0  72   0   5   0]
 [  1   0   0   1   0 936   0  38   3  21]
 [174   2 122  30  93   0 562   0  17   0]
 [  0   0   0   0   0  38   0 934   1  27]
 [ 12   1   8   8   2  15  25   4 925   0]
 [  0   0   0   0   0  15   1  39   0 945]]
Plotting confusion matrix
```



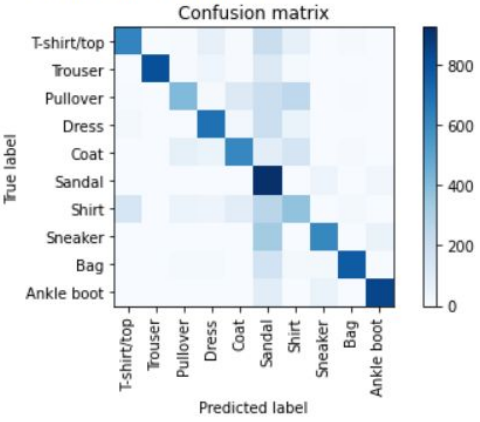
SVM2:

```
F1 score: 0.8706150942287518
Accuracy score: 0.8716
Confusion matrix:
 [[838   3  13  44   2   2  87   0  11   0]
 [  3 961   2  28   3   0   3   0   0   0]
 [ 12   4 799  12 105   0  65   0   3   0]
 [ 28   7  12 887  32   0  31   0   3   0]
 [  0   2  90  30 809   0  67   0   2   0]
 [  0   0   0   1   0 942   0  40   1  16]
 [153   1 108  32  80   0 612   0  14   0]
 [  0   0   0   0   0  26   0 947   0  27]
 [  2   1   1   6   2   3   9   3 973   0]
 [  0   0   0   0   0  11   0  40   1 948]]
Plotting confusion matrix
```



SVM3:

```
F1 score: 0.681186878618693
Accuracy score: 0.6675
Confusion matrix:
 [[622   1   4  74   6 205  77   2   9   0]
 [  3 811   7  41   7 117  14   0   0   0]
 [  7   0 411   9 118 199 249   0   7   0]
 [ 19   2   1 695  22 201  58   0   2   0]
 [  0   2  76  49 616  93 156   0   8   0]
 [  0   0   0   0   0 924   0  44   1  31]
 [148   1  55  45  94 262 374   0  21   0]
 [  0   0   0   0   0 330   0 610   0  60]
 [  0   0  13  11   1 172  19  15 767   2]
 [  0   0   0   1   0  94   1  58   1 845]]
Plotting confusion matrix
```



Riflessioni sui dati ottenuti:

Come ben sappiamo, i vantaggi delle SVM sono molteplici, tra i quali troviamo:

- Buon adattamento anche in caso di dati "sconosciuti"
- Funzionano bene anche con dati non-strutturali come testo, immagini e alberi.
- Il trucco del Kernel-trick è il punto forza di SVM, con una funzione kernel appropriata saremo in grado di risolvere qualsiasi problema complesso.

però sfortunatamente:

- Scegliere una "buona" funzione kernel non è facile.
- il Tempo di addestramento è estremamente lungo per set di dati di grandi dimensioni.

Quindi possiamo affermare che anche nel caso del Dataset in esame l'algoritmo *SVClassifier* (*scikit-learn*) si è presentato estremamente lungo in termini di Training-time (10,12 e 50 minuti). Nonostante ciò si può affermare che riesce ad imparare bene dai dati di training e grazie a ciò raggiunge prestazioni sempre > 80%, in diverse configurazioni.
Possiamo inoltre osservare come i diversi tipi di funzione Kernel influiscono sui dati ottenuti, utilizzando infatti un kernel lineare otteniamo prestazioni minori sia in termini di Training-time che di Accuracy.
Notiamo che i due tipi di kernel che si adattano di più al nostro set di dati sono quelli rbf e quello poly.
Tuttavia non esiste una regola rigida e veloce su quale kernel funzioni meglio in ogni scenario, bisogna testarli tutti e selezionare quello con i migliori risultati sul set di dati di test.

Soluzione all' overfitting:

L' algoritmo SVM è di base molto resistente all' overfitting, ma può essere irrobustito ulteriormente con il tuning per parametro C.
Per alti valori di C l' ottimizzazione sceglierà un iperpiano con un margine dai vettori di supporto più piccolo, se questo iperpiano è in grado di classificare correttamente più punti del training dataset.
Per valori bassi di C verrà invece scelto un iperpiano con un margine più ampio anche se provoca più missclassification del training dataset.
Generalmente per evitare overfitting è meglio un valore di C basso.

2. K-NN - K-Nearest Neighbors:

Trattasi di un algoritmo utilizzato nel riconoscimento di *pattern* per la classificazione di oggetti, basandosi sulle caratteristiche degli oggetti vicini a quello considerato.
In *classificazione* l'oggetto viene classificato in base ad un voto di pluralità dei suoi vicini, oggetto viene assegnato alla classe più comune tra i suoi K vicini più vicini ($k \in N$)
es: se k = 1, l'oggetto è assegnato alla classe di quel singolo vicino più prossimo.
In quanto vi sarà sicuramente una predominanza delle classi con più oggetti talvolta è utile pesare i contributi dei vicini in modo da dare maggior importanza in base alla distanza dell'oggetto considerato.

Algoritmo:

Si compone in 3 fasi:

1. *Fase di apprendimento:*
   Spazio viene partizionato in regioni in base alle posizioni e alle caratteristiche → insieme di apprendimento dell'algoritmo.
2. *Calcolo della distanza:*
   Gli oggetti sono considerati vettori posizione di uno spazio multidimensionale, solitamente si usa la distanza euclidea ($d(P1, P2) = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$), oppure vengono utilizzate anche quelle di *Manhattan* o *Hamming*.
3. *Classificazione:*
   Un punto viene assegnato alla classe C se questa è più frequente fra i k esempi più vicini (vicinanza = distanza tra i punti). I vicini sono presi da un insieme di oggetti per cui è nota la classificazione corretta.

Una volta scelto il valore di K (numero dei vicini) siamo pronti ad andare ad analizzare le diverse performance al variare di questo parametro:

```
knn1 = KNeighborsClassifier(n_neighbors=1)
knn2 = KNeighborsClassifier(n_neighbors=3)
knn3 = KNeighborsClassifier(n_neighbors=10)

# Codice comune a tutti e 4 i modelli:
knn(1,2,3).fit(x_train, y_train)
y_pred_knn(1,2,3) = knn(1,2,3).predict(x_test)
```

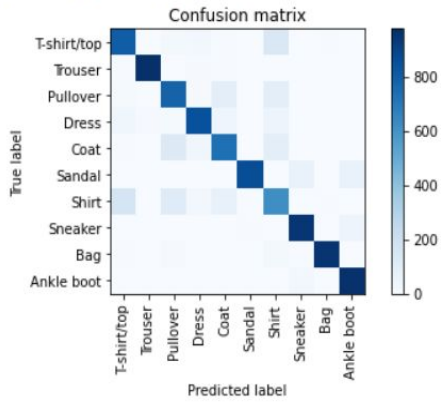
Ed ecco una tabella dei dati ottenuti:

Name	Parameter	Accuracy	Training time
KNN1	{n_neighbors = 1}	85%	15.52 minute
KNN2	{n_neighbors = 3}	85%	15.81 minute
KNN3	{n_neighbors = 10}	85%	18.13 minute

Invece di seguito il report dettagliato delle 3 macchine:

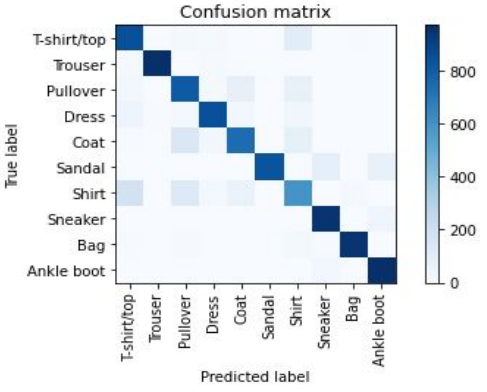
KNN1:

```
F1 score: 0.8503492525016987
Accuracy score: 0.8497
Confusion matrix:
 [[800   2  20  26   5   0 142   1   4   0]
 [  7 975   2   8   4   0   3   0   1   0]
 [ 15   2 782  10  97   0  94   0   0   0]
 [ 35   9  14 850  42   0  48   0   2   0]
 [  5   2 127  34 734   0  97   0   1   0]
 [  0   0   0   0   0 863   2  68   1  66]
 [160   1 117  27  69   0 619   0   7   0]
 [  0   0   0   0   0   5   0 949   0  46]
 [  5   1   9   3   2   0  17   4 958   1]
 [  0   0   0   0   0   2   0  30   1 967]]
Plotting confusion matrix
```



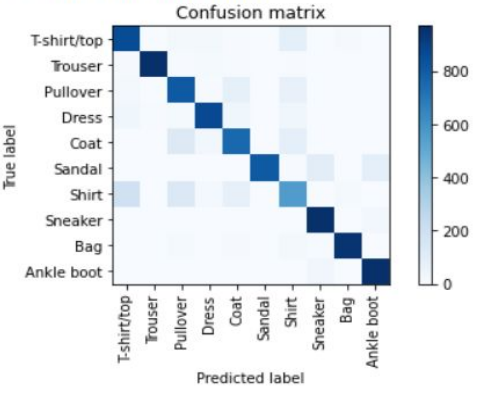
KNN2:

```
F1 score: 0.8539002124666113
Accuracy score: 0.8541
Confusion matrix:
 [[853   1  16  15   3   0 106   1   5   0]
 [  9 971   2  11   4   0   2   0   1   0]
 [ 27   2 812   8  78   0  73   0   0   0]
 [ 48   7  22 855  30   0  36   0   2   0]
 [  5   3 141  24 743   0  82   0   2   0]
 [  1   0   0   1   0 835   3  90   0  70]
 [177   3 132  21  63   0 595   0   9   0]
 [  0   0   0   0   0   3   0 952   0  45]
 [  7   1  10   3   3   1  16   6 952   1]
 [  0   0   0   0   0   2   0  24   1 973]]
Plotting confusion matrix
```



KNN3:

```
F1 score: 0.8506366581732875
Accuracy score: 0.8515
Confusion matrix:
 [[863   0  20  19   5   0  84   1   8   0]
 [  8 964   6  13   4   0   4   0   1   0]
 [ 21   2 813   7  82   0  74   0   1   0]
 [ 37   4  14 871  37   0  35   0   2   0]
 [  1   0 128  25 755   0  90   0   1   0]
 [  1   0   0   1   0 807   5 100   2  84]
 [195   0 135  22  76   0 560   0  12   0]
 [  0   0   0   0   0   2   0 969   0  29]
 [  1   1  14   3   8   0  16   6 949   2]
 [  0   0   0   0   0   1   1  34   0 964]]
Plotting confusion matrix
```



Riflessioni sui dati ottenuti:

Come possiamo notare dai dati ottenuti KNN nelle diverse configurazioni riesce sempre ad ottenere un' accuracy di circa 85%, il che è significativamente importante, però notiamo anche che impiega sempre un tempo minimo di 15 minuti, il che lo rende poco efficiente in termini di training-time.
Il principale svantaggio di KNN è di diventare significativamente più lento all'aumentare del volume dei dati, ma rimane tuttora molto usato nei problemi di classificazione di oggetti simili.

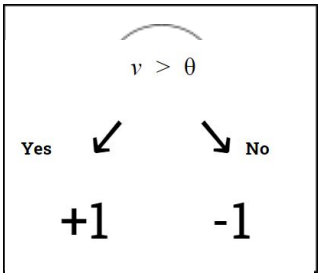
Abbiamo visto come la scelta della K giusta per i nostri dati viene effettuata provando diverse K e scegliendo quella che funziona meglio.

In generale possiamo notare che al ridurre del valore di K le nostre previsioni diventano meno stabili, al contrario, aumentando K, le nostre previsioni diventano man mano sempre più stabili a causa del voto a maggioranza/media e, quindi, più propense a fare previsioni più accurate (fino ad un certo punto).
Grazie a ulteriori dati trovati in rete, abbiamo potuto osservare come fino a k = 5 (tempo medio di training = 1 ora) si riescano ad ottenere accuracy dell'85%, ma con k = 1 risparmiando 20 minuti di training-time si ottenga un'accuratezza inferiore (84%).

Soluzione all' overfitting:

L'overfitting sull' algoritmo KNN si risolve scegliendo accuratamente il valore del parametro k.
Per k = 1 si avrà sempre overfitting, in quanto il punto più vicino di qualsiasi punto del training dataset è per definizione il punto stesso, si avrà quindi sempre un' accuratezza pari a 1.
Genericamente il valore di k deve essere dell' ordine di grandezza di sqrt(n_samples)/2.

3. Decision Tree:



Un Albero di Decisione (*Decision Tree*) è un albero di classificatori (*Decision Stump*) dove ogni nodo interno è associato ad una particolare "domanda" su una caratteristica (*feature*). Da questo nodo dipartono tanti archi quanti sono i possibili valori che la caratteristica può assumere, fino a raggiungere le foglie che indicano la categoria associata alla decisione.

Fig 0 : *esempio di un decision stump*
v è una caratteristica (feature) estratta dall'immagine e θ *una soglia.*

La classificazione avviene partendo dal nodo radice. In ogni nodo una sola caratteristica dell'oggetto viene valutata. Una valutazione tipica è il confronto con un certo valore Xs cosiddetto valore di soglia.
Se vi sono possibili solo due risposte l'albero viene detto *binario.*
Il processo si ferma nel momento in cui il ramo percorso conduce ad una foglia dell'albero: in tal caso il classificatore attribuisce all'oggetto l'etichetta contenuta nella foglia in cui `e arrivato. Nel caso in cui la classificazione interessi variabili discrete, l'albero `e detto decisionale, nel caso di variabili continue `e detto regressione.
Vi sono 2 fasi fondamentali:

1. *Building:*
   fase algoritmica alla costruzione dell'albero decisionale.
2. *Pruning:*
   potatura dopo la crescita dell'albero, tipo di ottimizzazione al fine di eliminare sottoclassi che non offrono particolari vantaggi predittivi.

Parametri dell'algoritmo:

- Criterion :
  - *gini*: basato sul coefficiente di impurità di Gini.
    Una feature si dice pura se basando una nodo su di essa il sample in input arriva sempre al nodo foglia che lo identifica correttamente. Grazie a questo coefficiente l'algoritmo sceglie come root la feature che più si avvicina alla purezza.
  - *entropy*: indicatore basato sull'entropia, variabilità dei valori dell'attributo presenti in un nodo foglia.
- Splitter : strategia di divisione :
  - *best :* il modello si prende la features con la massima importanza.
  - *random :* il modello si prende la features in modo casuale, ma con la stessa distribuzione.

- Max_depth : profondità massima dell'albero.

Inizializziamo quindi, grazie alla libreria di *Scikit-learn* i nostri Decision Tree Classifier:

```
tree0 = DecisionTreeClassifier(max_depth=100, criterion='entropy')
tree1 = DecisionTreeClassifier(max_depth=10, criterion='entropy', splitter = 'best')
tree2 = DecisionTreeClassifier(max_depth=10, criterion='entropy', splitter = 'random')
# Comune a tutti gli alberi:
tree(1,2,3).fit(x_train, y_train)
```

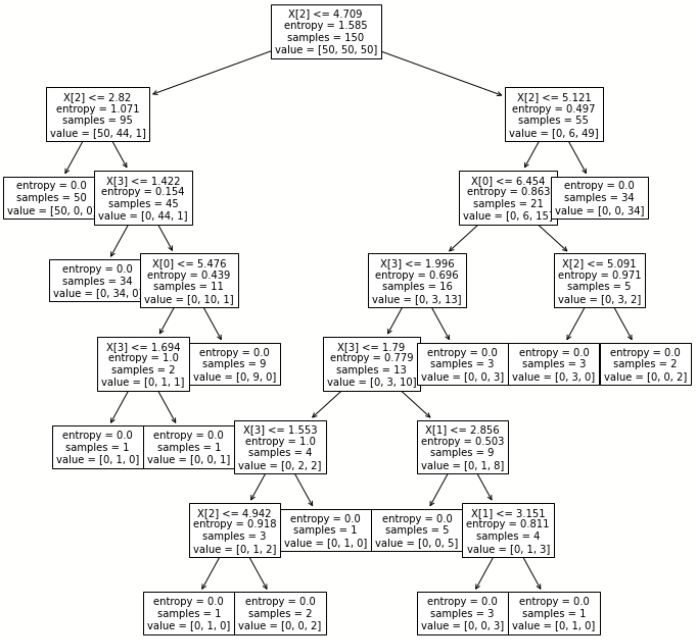
e la successiva analisi:

```
y_pred_tree = tree1.predict(x_test)
tree_f1 = metrics.f1_score(y_test, y_pred_tree, average= "weighted")
tree_accuracy = metrics.accuracy_score(y_test, y_pred_tree)
tree_cm = metrics.confusion_matrix(y_test, y_pred_tree)
print("----------------Decision Tree Report---------------")
print("F1 score: {}".format(tree_f1))
print("Accuracy score: {}".format(tree_accuracy))
print("Confusion matrix: \n", tree_cm)
print('Plotting confusion matrix')

plt.figure()
plot_confusion_matrix(tree_cm, labelNames)
plt.show()

print(metrics.classification_report(y_test, y_pred_tree))
```

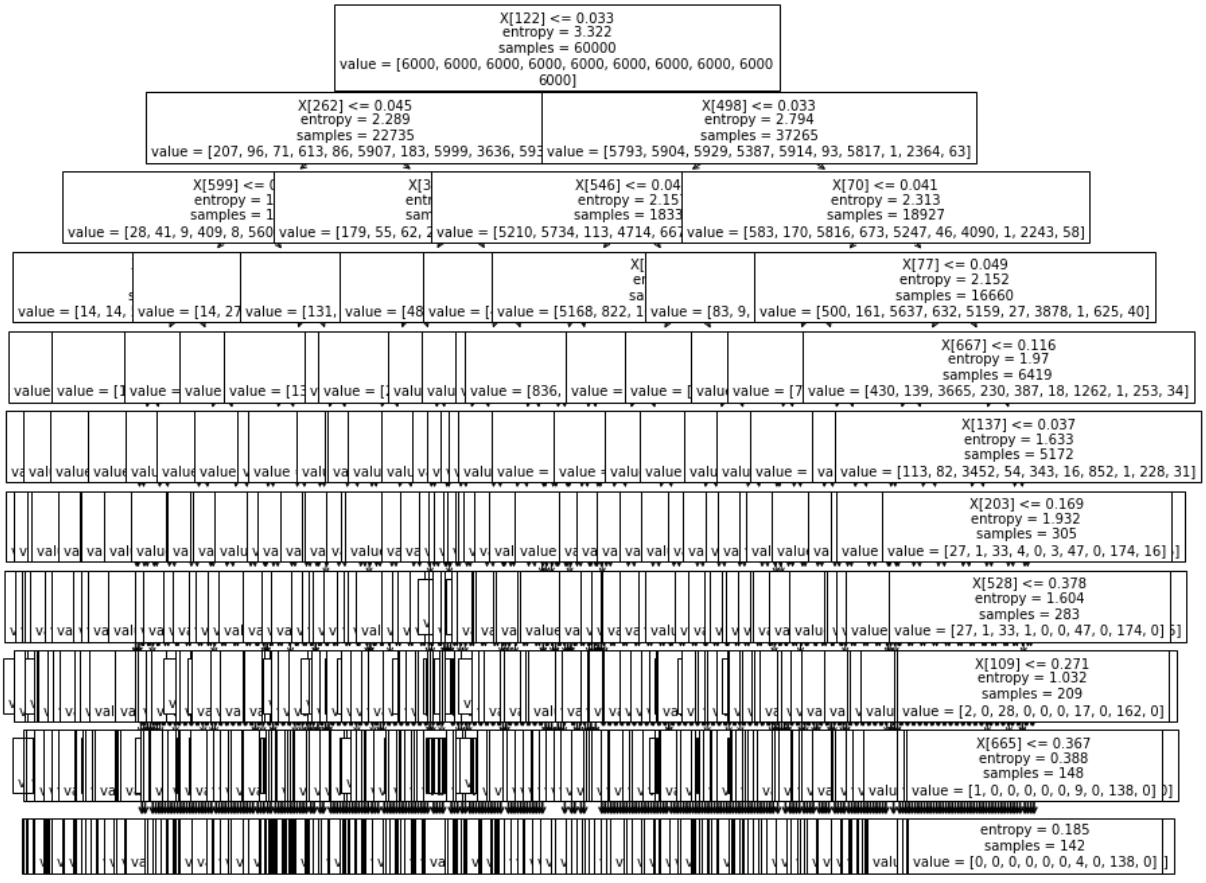
Tabella riassuntiva dei dati ottenuti nelle 3 diverse configurazioni:

Name	**Parameter**	**Accuracy**	**Training time**
Tree0	{max_depth = 100, criterion = 'entropy', splitter = 'best'}	80%	0.76 minute
Tree1	{max_depth = 10, criterion = 'entropy', splitter = 'best'}	81%	0.48 minute
Tree2	{max_depth = 10, criterion = 'entropy', splitter = 'random'}	79%	0.10 minute

Ecco come appare il plot del nostro **Tree2**('max_depth=10,criterion = 'entropy, splitter = *'random'*):



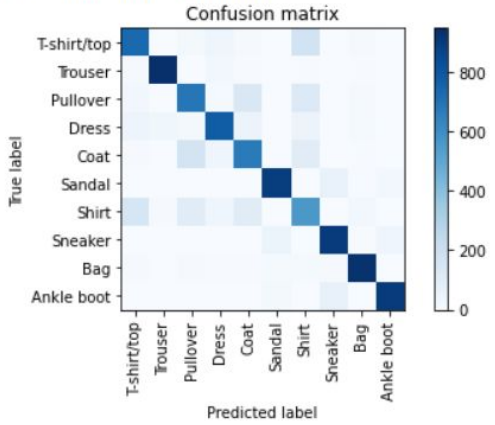
e qui invece possiamo notare quanto il plot di **Tree1**(max_depth = 10, criterion = 'entropy,splitter = 'best')



Report dettagliato:

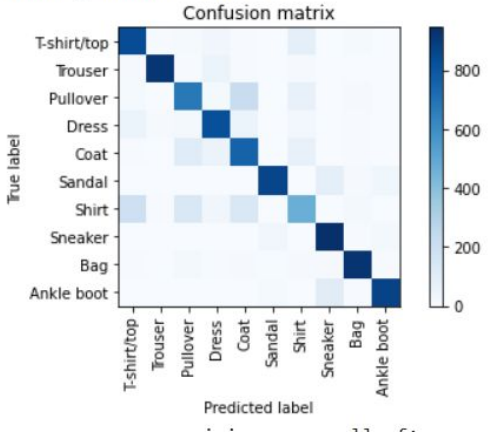
TREE0:

```
F1 score: 0.8020893858420028
Accuracy score: 0.8017
Confusion matrix:
 [[733   3  22  42  10   3 174   1  11   1]
 [ 10 946   3  27   4   0   6   0   4   0]
 [ 25   0 686  13 138   1 126   0  10   1]
 [ 45  35  15 783  54   0  54   0  12   2]
 [ 11   1 164  42 671   0 106   0   5   0]
 [  0   1   1   2   1 895   1  65   7  27]
 [148   8 110  38 105   1 563   0  27   0]
 [  1   0   0   0   0  51   0 903   6  39]
 [ 11   1   9   4   4  14  12   8 934   3]
 [  1   2   1   1   0  22   0  68   2 903]]
Plotting confusion matrix
```



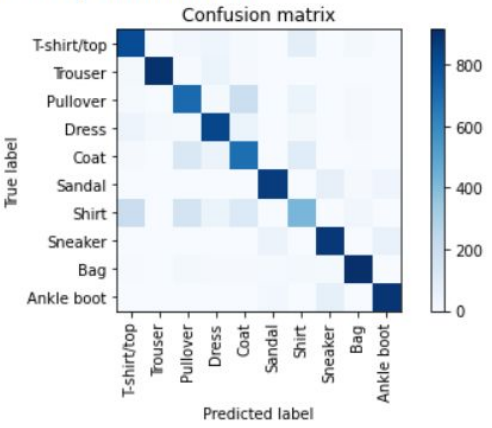
TREE1:

```
F1 score: 0.809022842862855
Accuracy score: 0.811
Confusion matrix:
 [[840   7  13  33   6   3  84   1  12   1]
 [ 11 920   5  48   4   0  10   0   2   0]
 [ 12   1 679  12 216   2  69   0   8   1]
 [ 57  11  20 829  51   0  26   0   4   2]
 [  4   3 110  55 757   0  67   0   4   0]
 [  2   1   1   3   0 867   0  83   9  34]
 [198   5 134  32 137   2 472   0  20   0]
 [  0   0   0   0   0  32   0 945   4  19]
 [  7   1  17   4   9   7  10  10 928   7]
 [  0   0   1   1   0  14   0 107   4 873]]
Plotting confusion matrix
```



TREE2:

```
F1 score: 0.79158156845021
Accuracy score: 0.7942
Confusion matrix:
 [[812   1  22  42  11   2  92   0  18   0]
 [ 15 907   7  56  10   0   3   0   2   0]
 [ 13   1 708  13 200   0  55   0   8   2]
 [ 46  20  13 836  55   2  19   1   8   0]
 [  8   1 130  52 694   0 107   0   7   1]
 [  1   0   2   1   1 870   1  75  10  39]
 [206   2 163  48 125   2 425   0  29   0]
 [  0   0   0   0   0  53   0 883   1  63]
 [  6   2  15   8   9  11  18  13 915   3]
 [  1   0   1   0   0  28   1  76   1 892]]
Plotting confusion matrix
```



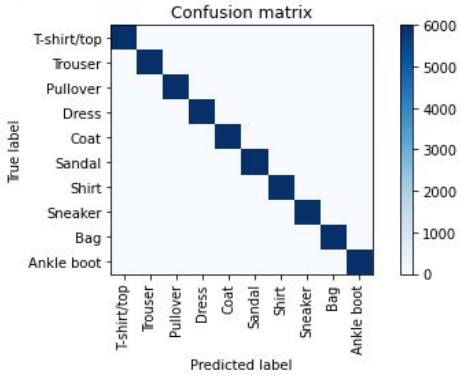
Riflessioni sui dati ottenuti:

Come si può chiaramente notare dai dati ottenuti i DecisionTreeClassifier, sono i modelli più veloci in termini di Training-time, riuscendo a risolvere il nostro problema di classificazione sempre in un tempo < 1 minuto, sorprendente.
Però notiamo anche che le prestazioni in termini di Accuracy non sono così sorprendenti, infatti non riusciamo mai a superare l'81%, anche nel caso di aumento della profondità del nostro albero di decisione.

Soluzione all'overfitting:

L' algoritmo Decision Tree può cadere in overfitting se la max depth è troppo elevata. In particolare con max_depth = 100 l' accuracy sul training dataset è pari a 1.

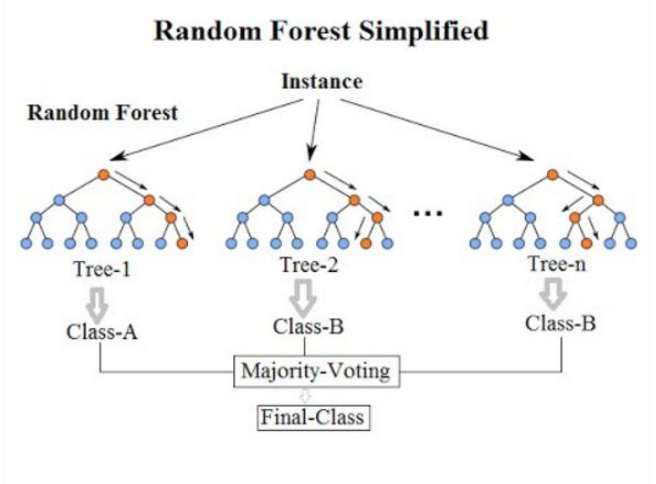
```
-----------------Decision Tree Report---------------
F1 score: 1.0
Accuracy score: 1.0
Confusion matrix:
 [[6000    0    0    0    0    0    0    0    0    0]
 [   0 6000    0    0    0    0    0    0    0    0]
 [   0    0 6000    0    0    0    0    0    0    0]
 [   0    0    0 6000    0    0    0    0    0    0]
 [   0    0    0    0 6000    0    0    0    0    0]
 [   0    0    0    0    0 6000    0    0    0    0]
 [   0    0    0    0    0    0 6000    0    0    0]
 [   0    0    0    0    0    0    0 6000    0    0]
 [   0    0    0    0    0    0    0    0 6000    0]
 [   0    0    0    0    0    0    0    0    0 6000]]
Plotting confusion matrix
```



Un altro hyperparameter da tenere in considerazione è max_features, ossia il numero massimo di feature scelte per costruire i nodi dell' albero. Impostandolo ad "auto" l' algoritmo userà soltando 28 features (sqrt(784)).
Abbiamo notato che con questo max_features = auto il tempo di training diminuiva notevolmente perdendo un po' di accuratezza :

{max_depth = 100, criterion = 'entropy', splitter = 'best', max_features= 'auto'}	78%	0.03 minuti
{max_depth = 10, criterion = 'entropy', splitter = 'best', max_features= 'auto'}	76%	0.02 minuti
{max_depth = 10, criterion = 'entropy', splitter = 'random', max_features= 'auto'}	73%	0.05 minuti

4. Random Forest:



Una foresta casuale, da quì il nome, combina molti alberi decisionali in un unico modello. Individualmente, le previsioni fatte dai singoli *alberi decisionali* potrebbero non essere accurate, combinate insieme, le previsioni saranno in media più vicine al risultato.
Il risultato finale restituito dal Random Forest è la media del risultato numerico restituito dai diversi alberi nel caso di un problema di *regressione*, o la classe restituita dal maggior numero di alberi nel caso di *classificazione*.
Tra i vantaggi di Random Forest troviamo:

- molto veloce a run-time;
- si tratta di uno dei più accurati algoritmi di apprendimento disponibili, anche in caso di dati sbilanciati;
- funziona in modo efficace su database di grandi dimensioni, producendo classificazioni molto accurate;
- fornisce stime di quali variabili sono importanti nella classificazione.

tra gli svantaggi invece:

- in alcuni compiti di classificazione in presenza di rumore, Random Forest può cadere in overfitting;

Per poter inizializzare il nostro modello dovremmo procedere alla scelta degli *hyperparameters*, i quali regoleranno il nostro Random Forest Classifier:

1. *max_depth:*
   percorso più lungo da radice a foglia.
2. *min_sample_split:*

numero minimo di osservazioni richiesto in un nodo per dividerlo

3. *min_samples_leaf:*
   numero minimo di campioni che dovrebbero essere presenti nel nodo foglia.
4. *max_terminal_nodes:*
   condizione sulla divisione dei nodi, limita la crescita dell'albero.
5. *n_estimators:*
   numero di estimatori, sceglierlo alto non è sempre la scelta giusta.
6. *max_samples :*
   quale frazione del Dataset viene assegnata ad ogni singolo nodo.
7. *max_features :*
   numero di funzionalità massime fornite a ciascun albero in una foresta casuale.

Ora siamo pronti ad inizializzare i diversi algoritmi:

```
random_forest = RandomForestClassifier(n_estimators=100, max_features='auto')
random_forest1 = RandomForestClassifier(n_estimators=100, max_depth = 70, criterion = 'entropy')
random_forest2 = RandomForestClassifier(n_estimators=100, max_depth = 70, criterion = 'gini')
# Comune a tutti i random_forest:
random_forest(0,1,2).fit(x_train, y_train)
```

E il codice per l'analisi:

```
y_pred_forest = random_forest.predict(x_test)
random_forest_f1 = metrics.f1_score(y_test, y_pred_forest, average= "weighted")
random_forest_accuracy = metrics.accuracy_score(y_test, y_pred_forest)
random_forest_cm = metrics.confusion_matrix(y_test, y_pred_forest)
```

Tabella riassuntiva dei dati ottenuti nelle 3 diverse configurazioni:

Name	Parameter	Accuracy	Training time
RF	{n_estimators = 100, max_depth=70, max_features = 'auto'}	87%	1.93 minute
RF1	{n_estimators = 100, max_depth = 70, criterion = 'entropy'}	87%	2.14 minute
RF2	{n_estimators = 100, max_depth = 70, criterion = 'gini'}	87%	1.37 minute

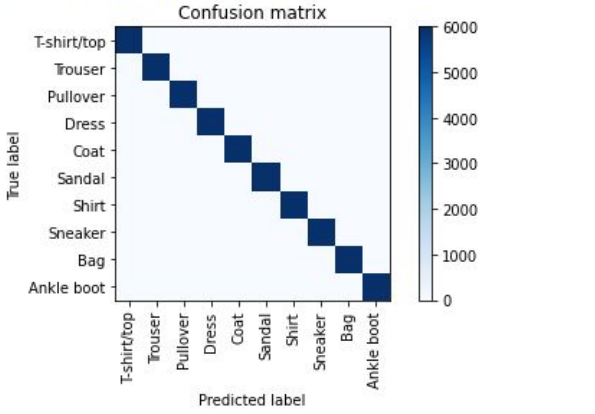
Riflessioni sui dati ottenuti:

Come possiamo notare, senza dubbio la tecnica Random Forest è la più accurata, sia in termini di training-time che di accuracy, in meno di 2 minuti si riescono sempre ad ottenere accuracy che variano tra l'86 e 87 [%], il che indica come la Random Forest sia stato l' algoritmo che ha subito meno l' overfitting.

Soluzione all' overfitting:

Alcune delle considerazioni fatte per il DT sono valide anche per la Random Forest, come ad esempio la scelta della max_depth e di max_features.
Questa base viene irrobustita ulteriormente dalla scelta casuale delle features da associare a ciascun albero (che possono anche ripetersi tra più alberi) e dall' assegnazione casuale di partizioni del training dataset a ciascun albero.
Un hyperparameter di cui abbiamo osservato l'importanza per ridurre l' overfitting è *n_estimators*.
Con *n_estimators* = 100 e *max_depth* = 70 abbiamo overfitting.

```
-----------------Random Forest Report---------------
F1 score: 0.9999833333332176
Accuracy score: 0.9999833333333333
Confusion matrix:
 [[6000    0    0    0    0    0    0    0    0    0]
 [   0 6000    0    0    0    0    0    0    0    0]
 [   0    0 6000    0    0    0    0    0    0    0]
 [   0    0    0 6000    0    0    0    0    0    0]
 [   0    0    0    0 6000    0    0    0    0    0]
 [   0    0    0    0    0 6000    0    0    0    0]
 [   0    0    0    1    0    0 5999    0    0    0]
 [   0    0    0    0    0    0    0 6000    0    0]
 [   0    0    0    0    0    0    0    0 6000    0]
 [   0    0    0    0    0    0    0    0    0 6000]]
Plotting confusion matrix
```



Esso rimane comunque la combinazione di parametri con l'accuratezza più elevata (87%), pensiamo che questo sia dovuto alla mancanza di samples nel dataset di test che si discostano dai samples forniti nel training dataset.

Riflessioni generali e diverse implementazioni possibili:

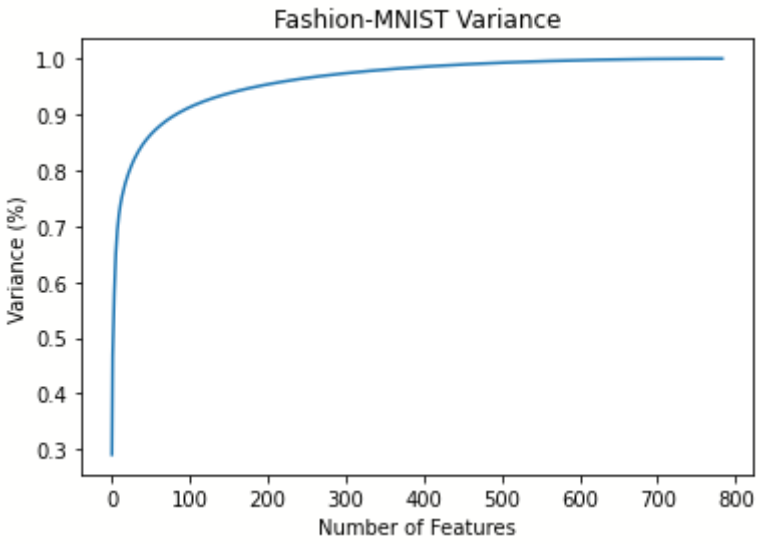
Come abbiamo visto per risolvere il problema di Image-Classification nel caso di Fashion MNIST Dataset, utilizzando 4 diversi algoritmi, esclusi quelli che riguardano la teoria degli alberi (DecisionTreeClassifier & RandomForest), possiamo notare che, seppur con delle prestazioni (in termini di accuratezza) buone, sempre intorno al 84/85 [%], ci troviamo a dover aspettare un sacco di tempo per la fase di training degli algoritmi.
Una possibile soluzione:

Analisi delle Componenti Principali (PCA):

Analisi delle componenti principali, o decomposizione ortogonale propria, è una tecnica per la semplificazione dei dati.
Questo metodo ha lo scopo di ridurre il numero più o meno elevato di variabili che descrivono un insieme di dati a un numero minore di variabili latenti, limitando il più possibile la perdita di informazioni.
In PCA, si indica col termine "informazione" la variabilità totale delle variabili di input originarie, cioè la somma delle varianze delle variabili originarie.

Una variabile, tra le componenti principali, che abbia una scarsa variabilità (relativamente alle altre variabili) può essere trattata approssimativamente come una costante. Omettere PC a bassa variabilità e porre attenzione sulle PC con varianza più elevata, può essere visto come un modo semplice e "sensato" di ridurre la dimensionalità del dataset.
Riassumendo:
La PCA evidenzia le regolarità statistiche dei dati proiettando lo spazio degli ingressi originariamente di dimensione N su uno spazio di dimensione M, con M<N, denominato spazio di uscita, effettuando una riduzione della dimensionalità mantenendo, però quasi invariato il contenuto informativo utile.
Andiamo a vedere come implementare il codice:

```
from sklearn.decomposition import PCA
pca = PCA().fit(x_train)
# Plottiamo la somma degli autovalori
plt.figure()
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Number of Features')
plt.ylabel('Variance (%)') #for each component
plt.title('Fashion-MNIST Variance')
plt.show()
```



Come possiamo notare dall'output del codice, come potremmo utilizzare circa 340 features, al posto di tutte le 784 che abbiamo sempre utilizzato, pertanto possiamo conservare il 96% dei dati.

```
pca = PCA(n_components=340)
pca.fit(x_train)
x_train_pca = pca.transform(x_train)
x_test_pca = pca.transform(x_test)
x_train_pca.shape
```

```
---> (60000, 340)
```

Andiamo a vedere ora, come cambieranno le prestazioni nel caso di utilizzo di x_train_pca nei nostri diversi algoritmi:

```
# SVC:
svc = SVC(C=1, kernel='linear', gamma="auto")
svc.fit(x_train_pca, y_train)
# KNN:
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(x_train_pca, y_train)
# DecisionTree:
tree = DecisionTreeClassifier(max_depth=100, criterion='entropy')
tree.fit(x_train_pca, y_train)
# RandomForest:
random_forest = RandomForestClassifier(criterion='entropy', max_depth=70,
n_estimators=100)
random_forest.fit(x_train_pca, y_train)
```

ed ecco i diversi tempi di training:

```
--->SVM Time: 7.43 minute
--->KNN Time: 1.77 minute
--->Decision Tree Time: 2.03 minute
--->Random Forest Time: 9.44 minute
```

Tabella riassuntiva dei dati ottenuti:

Name	Parameter	Accuracy	Training time
SVC_PCA	{C=1, kernel='linear', gamma="auto"}	84%	7.43 minute
KNN_PCA	{n_neighbors = 5'}	86%	1.77 minute
DT_PCA	{max_depth=100, criterion='entropy'}	76%	2.03 minute
RF_PCA	{criterion='entropy', max_depth=10, n_estimators=100}	**88%**	2.07 minute

Come possiamo notare dai dati ottenuti, sia nel caso di SVC e di KNN i risultati ottenuti sono notevolmente migliori, riusciamo ad ottenere la stessa accuratezza ottenuta in principio, senza modificare nessun parametro, ma con una riduzione del training-time molto importante.
Invece negli algoritmi che utilizzano la teoria degli alberi, possiamo notare come oltre ad una crescita (quasi doppia) del tempo di training, riusciamo anche ad ottenere un'accuratezza notevolmente peggiore.

Considerazioni:

L'utilizzo di algoritmi che implementano la teoria degli alberi in un set di dati come quello descritto, presentano due problemi principali:

- RandomForest e DecisionTree non funzionano bene quando le caratteristiche sono una trasformazione monotona di altre caratteristiche (questo rende gli alberi meno indipendenti l'uno dall'altro).

Il nostro caso è l'esempio perfetto che non sempre la riduzione del numero di features è la scelta migliore, come abbiamo visto nel caso degli algoritmi che implementano gli alberi.

Colab notebooks sviluppati per l'intero progetto:

- *AllModel :* https://colab.research.google.com/drive/1rv0X7a5z94YBZejF4EzlYHXXz69YJ_un?usp=sharing
- *OnlySVM :* https://colab.research.google.com/drive/1tUgOcrcyAEQCpKI_X8R6EEM-ymdFQ9wV?usp=sharing
- *OnlyKNN :* https://colab.research.google.com/drive/1EjTmSsFj5xEF8jhH5cO56ihqVleTwP4v?usp=sharing
- *OnlyDecisionTree :* https://colab.research.google.com/drive/1dcvQaiIyoIYqvT6LK08PTS4X3LKh0rl-?usp=sharing
- *OnlyRandomForest :* https://colab.research.google.com/drive/1uMJwm55RORwDzgulZEy4yrT90Os7zAWc?usp=sharing