Machine Learning

Università Roma Tre
Dipartimento di Ingegneria

Anno Accademico 2021 - 2022

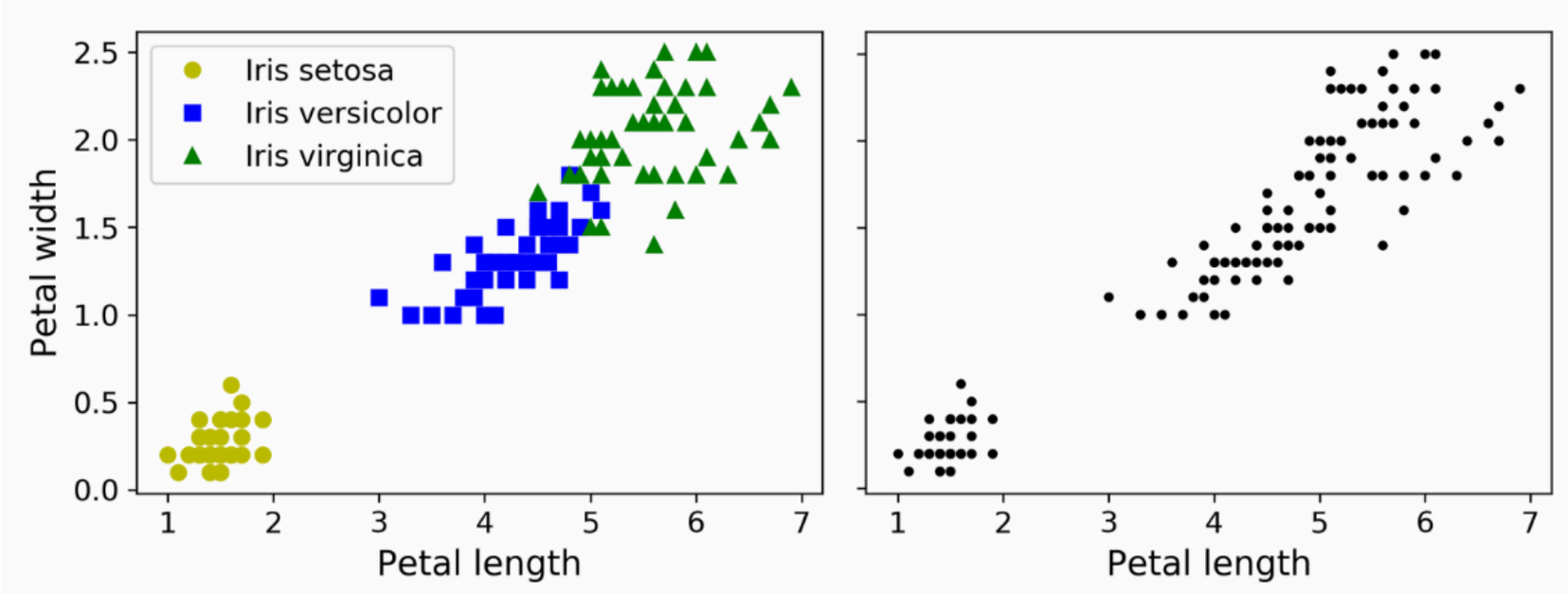
Esercitazione: Clustering (Ex 08)

Sommario

- Preprocessing: Scaling
- Scaling in Scikit-learn
- Scaling e classificazione
- Scikit-learn e K-Means
- Esempi di limiti dell'algoritmo K-Means

Clustering

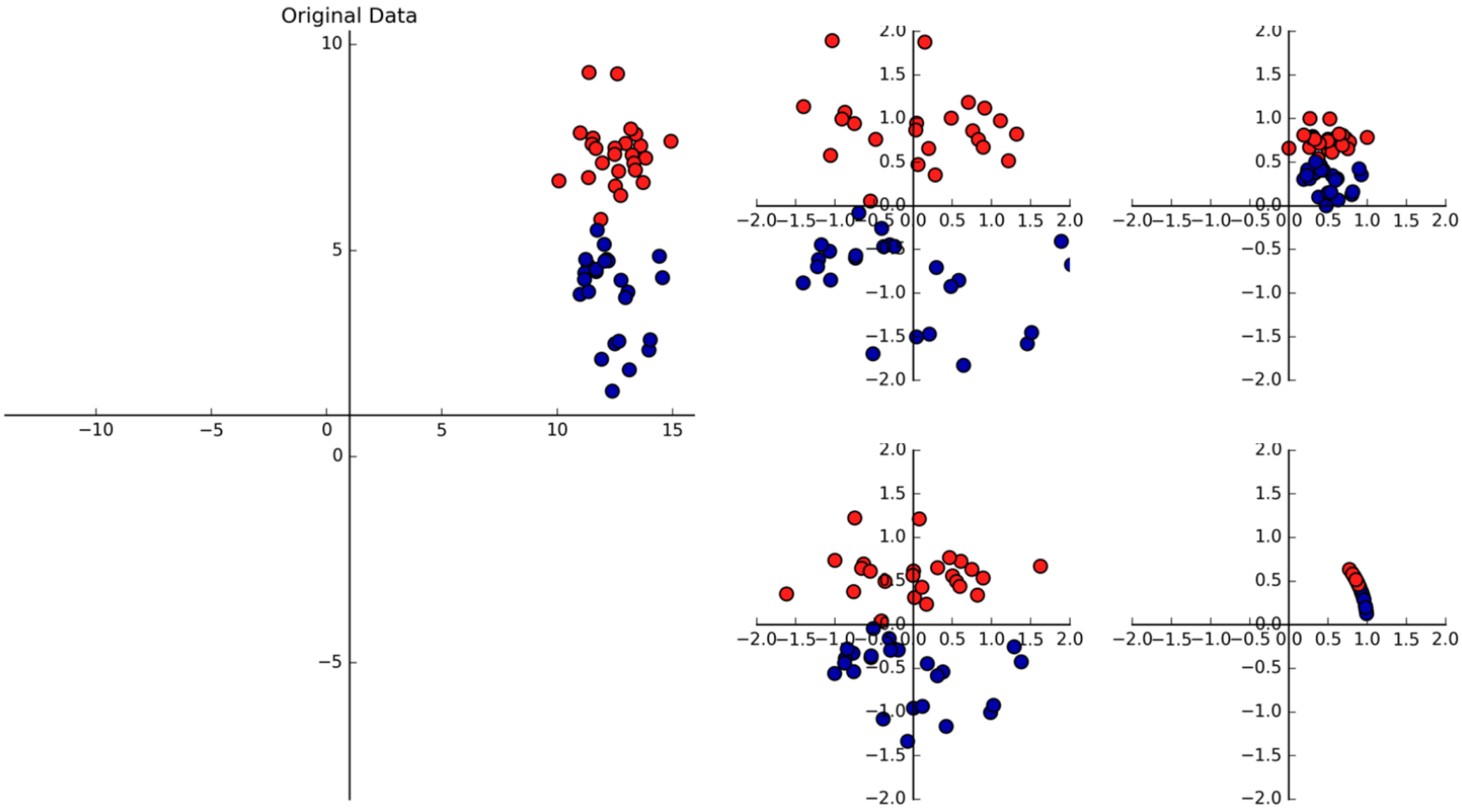
- Ci focalizziamo sugli algoritmi di *clustering*. Esistono anche *trasformazioni unsupervised*, utili per creare nuove rappresentazioni utili per analizzare dati o per darli in input a successivi algoritmi. Un approccio comune è la *riduzione di dimensionalità*, dove le N dimensioni corrispondenti alle features vengono "compresse" in poche dimensione (es. 2 o 3).
- La challenge del clustering è capire se l'algoritmo applicato su dati non etichettati (cioè senza output) riesce comunque a trovare qualcosa di utile.
- Esempio: Classification (sx) e Clustering senza label (dx)



Preprocessing: Scaling

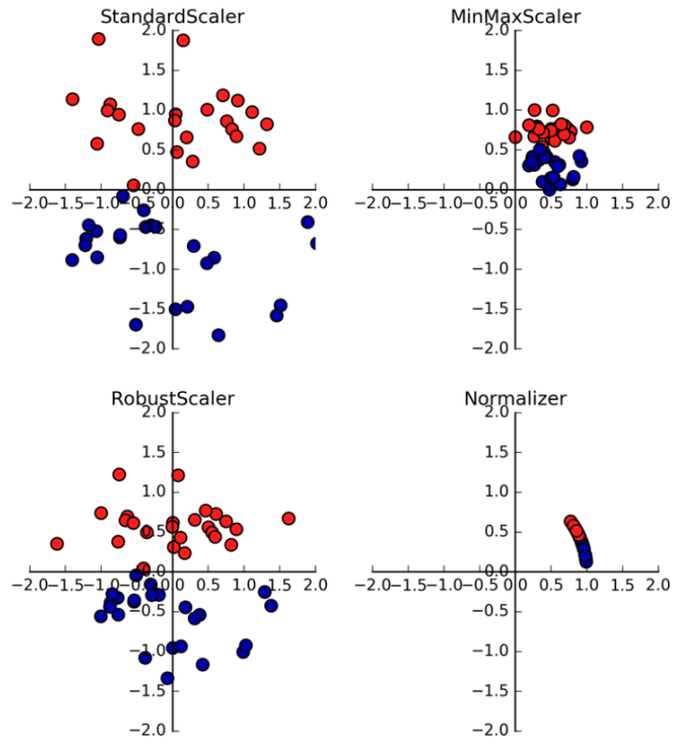
- Alcuni algoritmi di ML sono sensibili allo *scaling* dei dati. Per tale motivo spesso si opera un rescaling e shifting.
- Vediamo qualche esempio dalla libreria mglearn:

```
mglearn.plots.plot_scaling()
```



Preprocessing: Scaling

- Il diagramma mostra 4 scaler della libreria scikit-learn.
  - *StandardScaler*: garantisce media 0 e varianza 1
    Non garantisce alcun intervallo max e min
  - *RobustScaler*: approccio statistico simile,
    usa mediana e quartili, è meno sensibile
    agli *outliers*.



  - *MinMaxScaler*: sposta i dati nell'intervallo [0,1]
  - *Normalizer*: effettua un rescaling in modo che
    la distanza euclidea sia pari a 1, cioè proietta
    i punti su una circonferenza (o sfera) di raggio 1.
    Ogni punto è scalato per l'inverso della lunghezza.
    Utile quando si ha interesse soprattutto riguardo la direzione, piuttosto
    che della lunghezza del feature vector.

Scikit-learn: Scaling

- Usiamo il breast cancer dataset per testare i vari scaling su un contesto supervised con algoritmo SVM/SVC:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data,
cancer.target,
random_state=1)
print(X_train.shape)
print(X_test.shape)
```

```
>> (426, 30)
>> (143, 30)
```

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

# consideriamo solo X_train, non il y_train
scaler.fit(X_train)

>> MinMaxScaler(copy=True, feature_range=(0, 1))

# trasformiamo i dati
X_train_scaled = scaler.transform(X_train)
```

Scikit-learn: Scaling

```
# stampa i valori delle features prima e dopo il rescaling
print("transformed shape: {}".format(X_train_scaled.shape))
print("per-feature minimum before scaling:\n {}".format(X_train.min(axis=0)))
print("per-feature maximum before scaling:\n {}".format(X_train.max(axis=0)))
print("per-feature minimum after scaling:\n {}".format(
X_train_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n {}".format(
X_train_scaled.max(axis=0)))
```

```
>> transformed shape: (426, 30)
per-feature minimum before scaling:
[ 6.98 9.71 43.79 143.50 0.05 0.02 0. 0. 0.11
0.05 0.12 0.36 0.76 6.80 0. 0. 0. 0.
0.01 0. 7.93 12.02 50.41 185.20 0.07 0.03 0.
0. 0.16 0.06]
per-feature maximum before scaling:
[ 28.11 39.28 188.5 2501.0 0.16 0.29 0.43 0.2
0.300 0.100 2.87 4.88 21.98 542.20 0.03 0.14
0.400 0.050 0.06 0.03 36.04 49.54 251.20 4254.00
0.220 0.940 1.17 0.29 0.58 0.15]
per-feature minimum after scaling:
[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
per-feature maximum after scaling:
[ 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Scikit-learn: Scaling

- Applichiamo lo scaling anche sul X_test

```
# transform test data
X_test_scaled = scaler.transform(X_test)
# print test data properties after scaling
print("per-feature minimum after scaling:
\n{}".format(X_test_scaled.min(axis=0)))
print("per-feature maximum after scaling:
\n{}".format(X_test_scaled.max(axis=0)))
```

```
>> per-feature minimum after scaling:
[ 0.034 0.023 0.031 0.011 0.141 0.044 0. 0. 0.154 -0.006
-0.001 0.006 0.004 0.001 0.039 0.011 0. 0. -0.032 0.007
0.027 0.058 0.02 0.009 0.109 0.026 0. 0. -0. -0.002]
per-feature maximum after scaling:
[ 0.958 0.815 0.956 0.894 0.811 1.22 0.88 0.933 0.932 1.037
0.427 0.498 0.441 0.284 0.487 0.739 0.767 0.629 1.337 0.391
0.896 0.793 0.849 0.745 0.915 1.132 1.07 0.924 1.205 1.631]
```

- Non sono nel range [0,1], è corretto?

Scikit-learn: Scaling

- Applichiamo lo scaling anche sul X_test

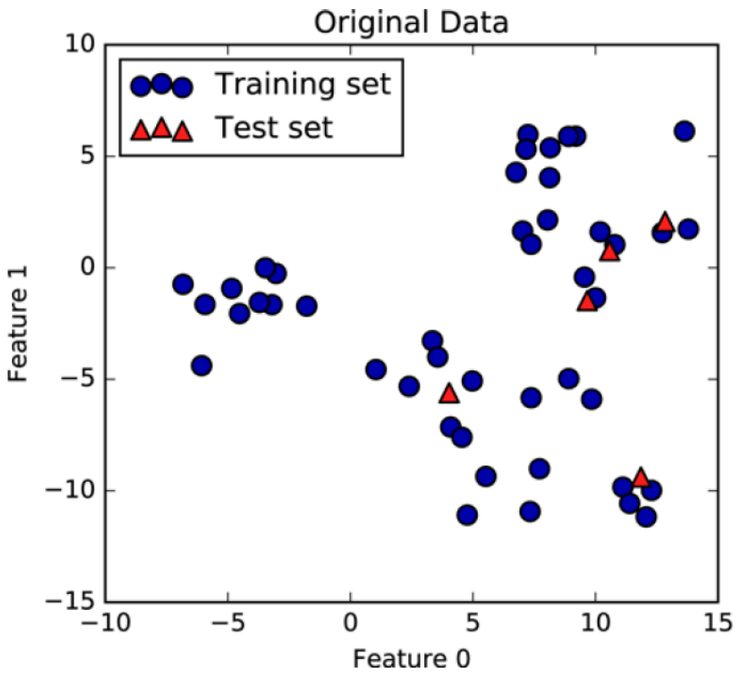
```
# transform test data
X_test_scaled = scaler.transform(X_test)
# print test data properties after scaling
print("per-feature minimum after scaling:
\n{}".format(X_test_scaled.min(axis=0)))
print("per-feature maximum after scaling:
\n{}".format(X_test_scaled.max(axis=0)))
```

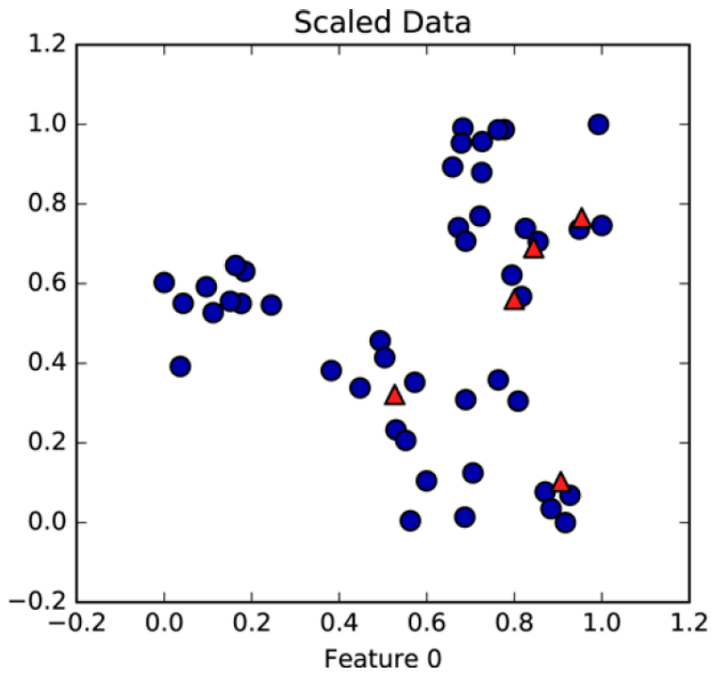
```
>> per-feature minimum after scaling:
[ 0.034 0.023 0.031 0.011 0.141 0.044 0. 0. 0.154 -0.006
-0.001 0.006 0.004 0.001 0.039 0.011 0. 0. -0.032 0.007
0.027 0.058 0.02 0.009 0.109 0.026 0. 0. -0. -0.002]
per-feature maximum after scaling:
[ 0.958 0.815 0.956 0.894 0.811 1.22 0.88 0.933 0.932 1.037
0.427 0.498 0.441 0.284 0.487 0.739 0.767 0.629 1.337 0.391
0.896 0.793 0.849 0.745 0.915 1.132 1.07 0.924 1.205 1.631]
```

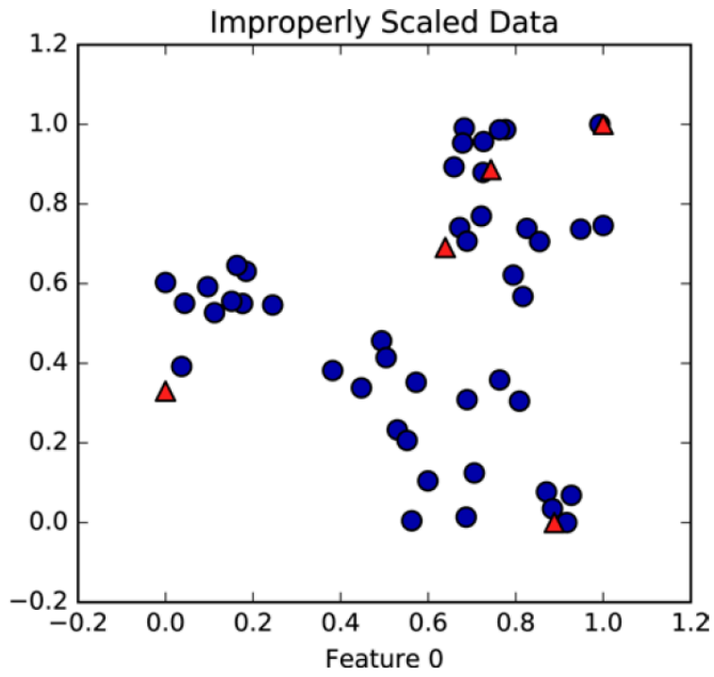
- Non sono nel range [0,1], è corretto?
  - Sì, perché il max e min sono stati ricavati dal training set, e possono essere distinti da quelli nel X_test.

Scikit-learn: Scaling

- Cosa succede se applicassimo due distinti rescaling sul training e sul test set?

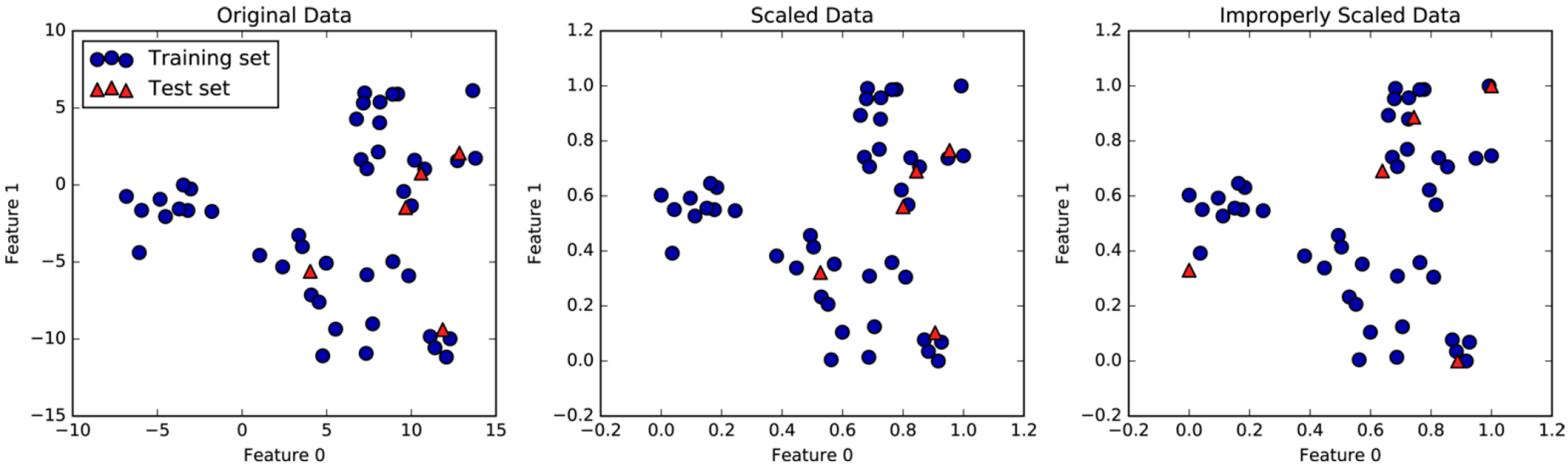






Scikit-learn: Scaling

- Cosa succede se applicassimo due distinti rescaling sul training e sul test set?



- Le istanze nel test set sono state scalate in modo improprio rispetto ai valori originali, e si trovano in posizioni relative diverse da quelle originali.

Scikit-learn: Scaling

- Nota: In scikit-learn, gli scaler hanno spesso il metodo fit_transform() che combina le 2 operazioni:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

X_scaled = scaler.fit(X).transform(X)

# stesso risultato ma più efficient
X_scaled_d = scaler.fit_transform(X)
```

Scikit-learn: Scaling e Classificazione

- Esercizio: Impiega il MinMaxScaler sul dataset breast cancer e impiega l'algoritmo di classificazione SVC(C=100). Confronta la performance senza scaling.

```
from sklearn.svm import SVC

X_train, X_test, y_train, y_test =
    train_test_split(cancer.data, cancer.target, random_state=0)

...
```

Scikit-learn: Scaling e Classificazione

- Esercizio: Impiega il MinMaxScaler e StandardScaler sul dataset breast cancer e impiega l'algoritmo SVC(C=100). Confronta la performance senza scaling.

```
from sklearn.svm import SVC

X_train, X_test, y_train, y_test =
     train_test_split(cancer.data, cancer.target, random_state=0)

svm = SVC(C=100)
svm.fit(X_train, y_train)
print("Test set accuracy: {:.2f}".format(svm.score(X_test, y_test)))
```

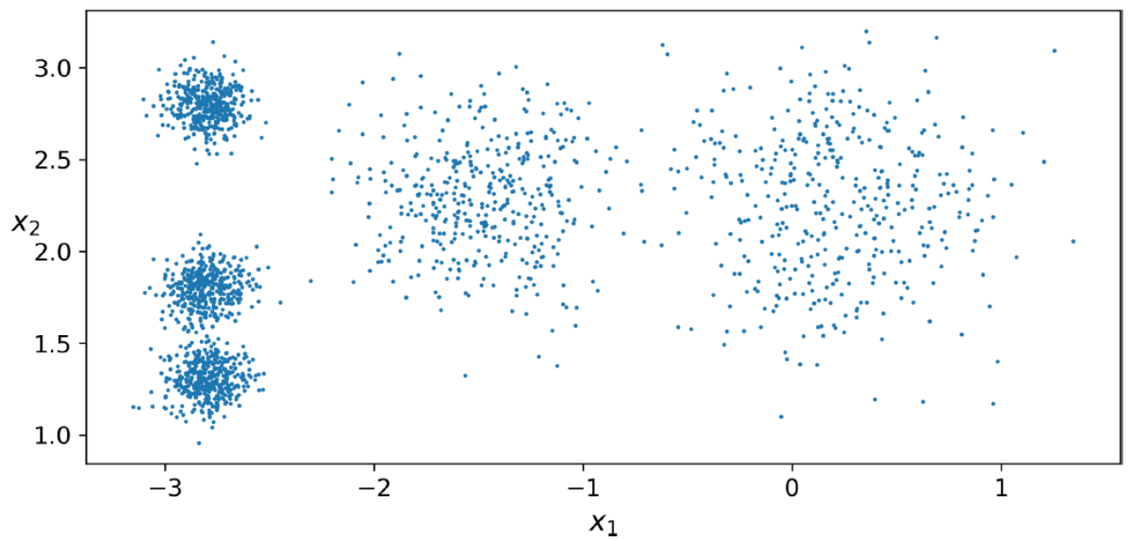
```
>> Test set accuracy: 0.63
```

```
# con scaling
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

svm.fit(X_train_scaled, y_train)

print("Scaled test set accuracy: {:.2f}".format(
svm.score(X_test_scaled, y_test)))
```

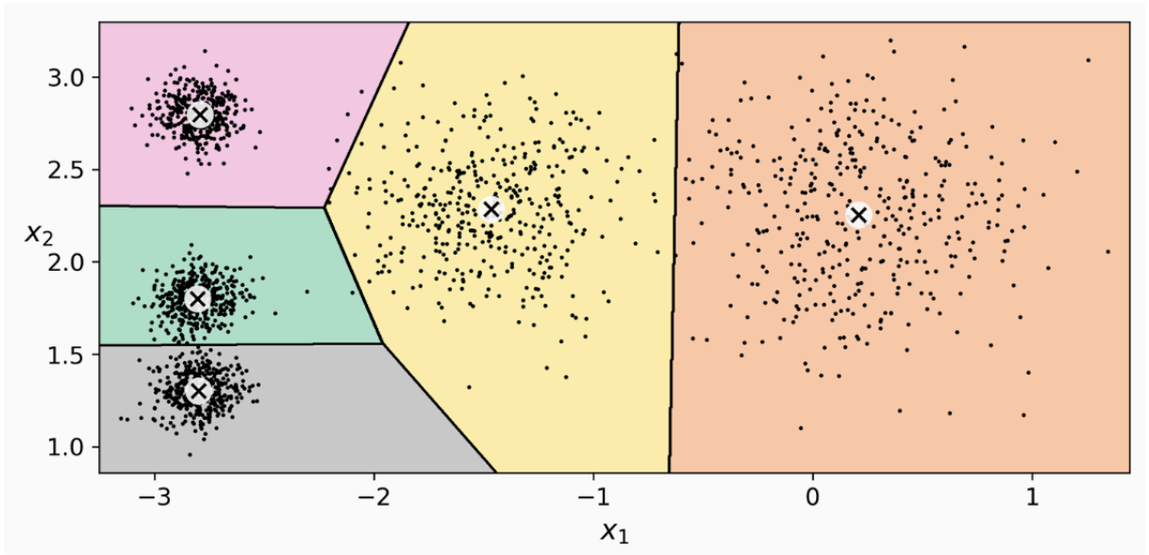
```
>> Scaled test set accuracy: 0.97 (con StandardScaler si ottiene 0.96)
```

Scikit-learn: K-means

- Scikit-learn implementa l'algoritmo con la classe *KMeans*. Il parametro *n_clusters* è richiesto per specificare il numero di cluster.
- Supponiamo di avere il seguente dataset:
- L'output dell'algoritmo può essere rappresentato col diagramma Voronoi Tesselation.




```
from sklearn.cluster import KMeans

k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)

print (y_pred)
>> array([4, 0, 1, ..., 2, 1, 0],
dtype=int32)

print (y_pred is kmeans.labels_)
>> True
```



Scikit-learn: Limiti K-means

- Possiamo ottenere le coordinate dei 5 centroidi:

```
kmeans.cluster_centers_
>> array([[-2.80389616, 1.80117999],
[ 0.20876306, 2.25551336],
[-2.79290307, 2.79641063],
[-1.46679593, 2.28585348],
[-2.80037642, 1.30082566]])
```

- E predire la classe di nuove istanze:

```
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
kmeans.predict(X_new)
>> array([1, 1, 2, 2], dtype=int32)
```

- **Nota**: K-Means non si comporta molto bene con cluster che hanno diametri molto distinti tra loro, poiché l'algoritmo valuta solo la distanza col centroide.

Scikit-learn: K-means

- Invece dell'*hard clustering* visto finora, dove l'output è un singolo cluster, possiamo ottenere uno score (anche chiamato *similarity score* o *affinity*) per ogni cluster col *soft clustering* mediante la funzione *transform*():

```
kmeans.transform(X_new)
>> array([[2.81093633, 0.32995317, 2.9042344 , 1.49439034, 2.88633901],
[5.80730058, 2.80290755, 5.84739223, 4.4759332 , 5.84236351],
[1.21475352, 3.29399768, 0.29040966, 1.69136631, 1.71086031],
[0.72581411, 3.21806371, 0.36159148, 1.54808703, 1.21567622]])
```

- È possibile impostare i centroidi iniziali in modo manuale col parametro *init*:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

- L'iperparametro *n_init* specifica quante volte l'algoritmo deve essere eseguito prima di selezionare la soluzione migliore ottenuta.

Scikit-learn: K-means

- Per valutare la bontà della soluzione si misura il costo basato sulla cluster heterogeneity, chiamato anche *inertia* del modello, cioè la distanza quadratica media con i centroidi.

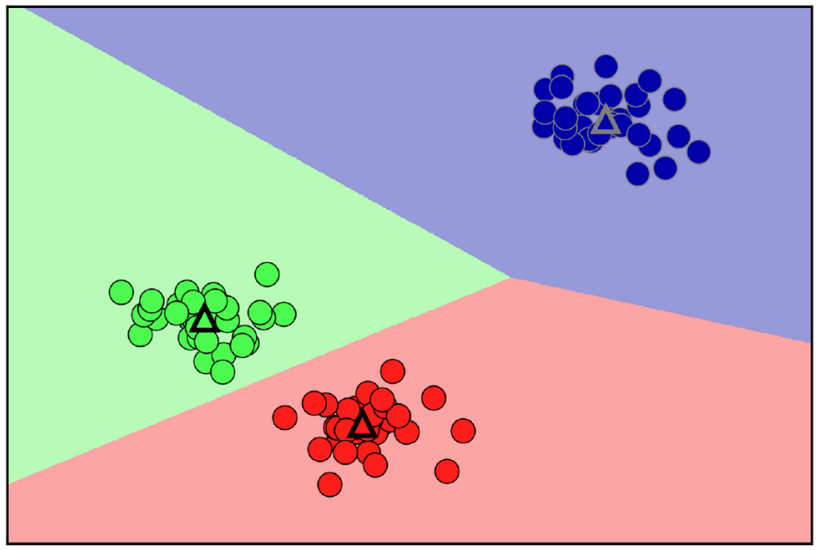
```
kmeans.inertia_
>> 211.59853725816856

kmeans.score(X)
>> -211.59853725816856
```

- **Nota**: di default KMeans() usa l'inizializzazione dei centroidi proposta in K-Means++. Se vuoi impiegare quella dell'algoritmo originale, imposta il parametro *init='random'*.

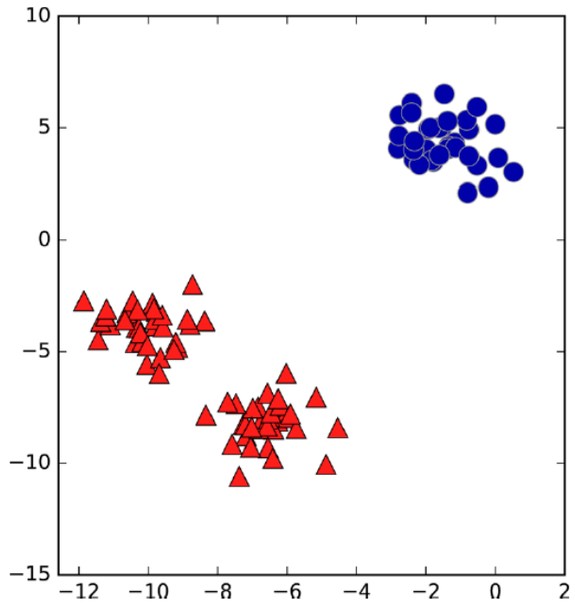
Scikit-learn: K-means

- Esempio con un dataset toy:

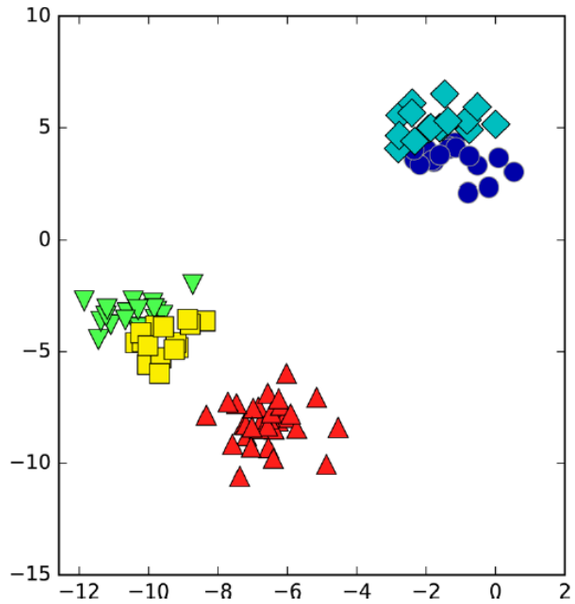
```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# generate synthetic two-dimensional data
X, y = make_blobs(random_state=1)

kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```



n_clusters=3



n_clusters=2

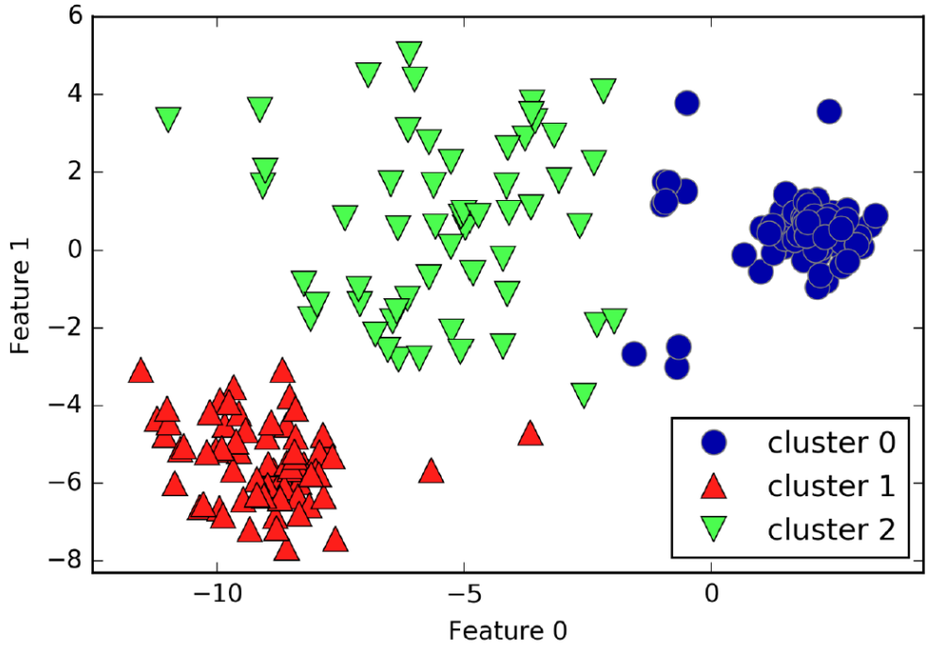


n_clusters=4

Scikit-learn: Limiti K-means

```
X_varied, y_varied = make_blobs(n_samples=200,
cluster_std=[1.0, 2.5, 0.5],
random_state=170)
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(X_varied)

mglearn.discrete_scatter(X_varied[:, 0], X_varied[:, 1], y_pred)
plt.legend(["cluster 0", "cluster 1", "cluster 2"], loc='best')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Secondo te è un output ideale?

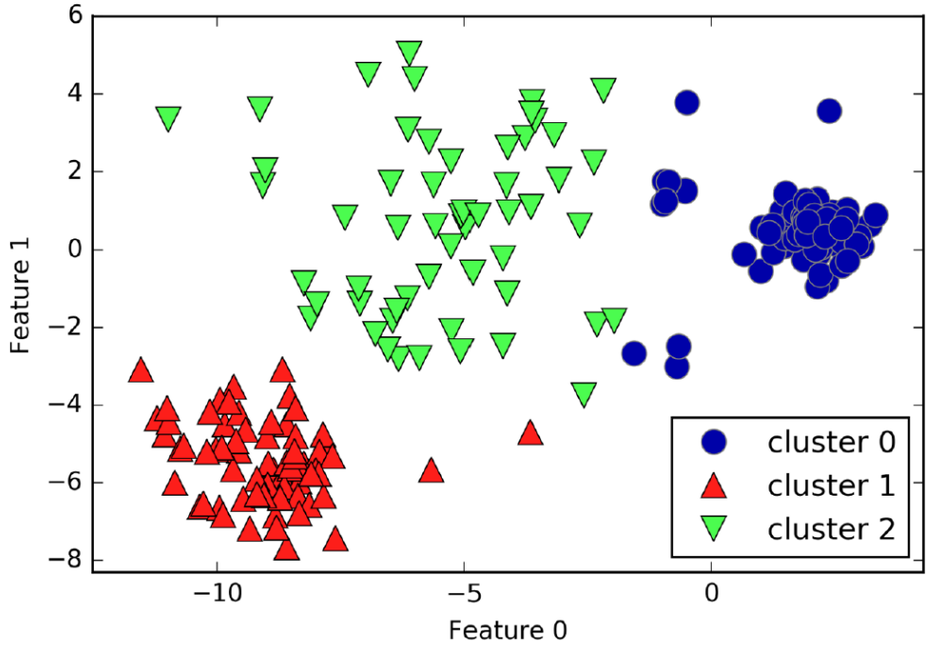


Scikit-learn: Limiti K-means

```
X_varied, y_varied = make_blobs(n_samples=200,
cluster_std=[1.0, 2.5, 0.5],
random_state=170)
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(X_varied)

mglearn.discrete_scatter(X_varied[:, 0], X_varied[:, 1], y_pred)
plt.legend(["cluster 0", "cluster 1", "cluster 2"], loc='best')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

K-means assume che ogni cluster abbia lo stesso diametro, e definisce la boundary tra i cluster esattamente a metà tra i due centroidi.

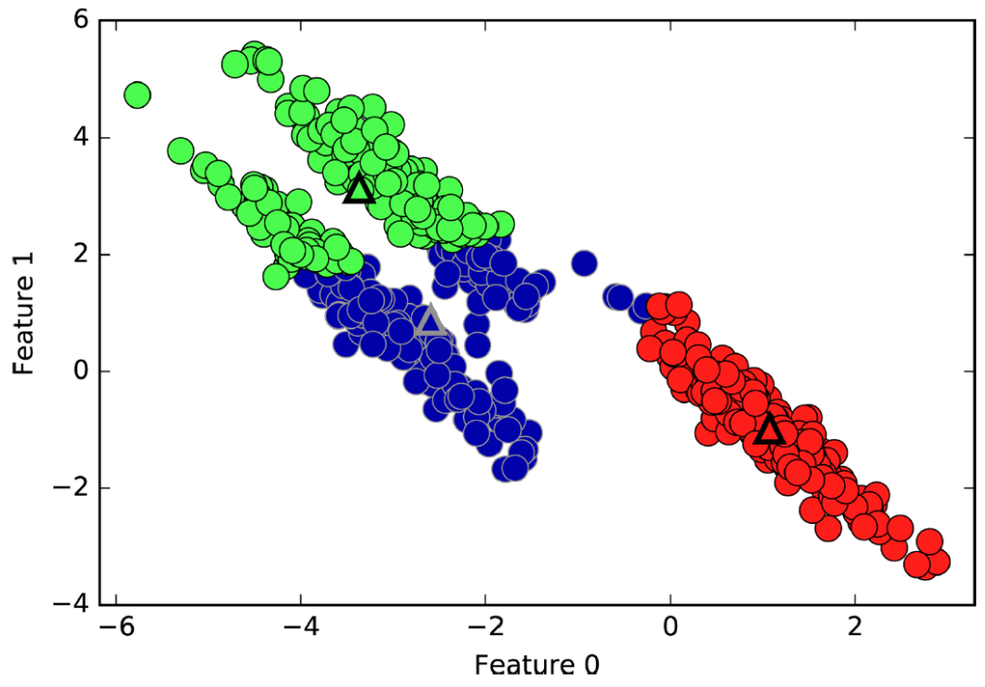
Alcuni punti del grafico potevano essere classificati in modo diverso.



Scikit-learn: Limiti K-means

```
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)

# trasforma i dati per mezzo di una distribuzione gaussiana
transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)

kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm3)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
marker='^', c=[0, 1, 2], s=100, linewidth=2, cmap=mglearn.cm3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Secondo te è un output ideale?

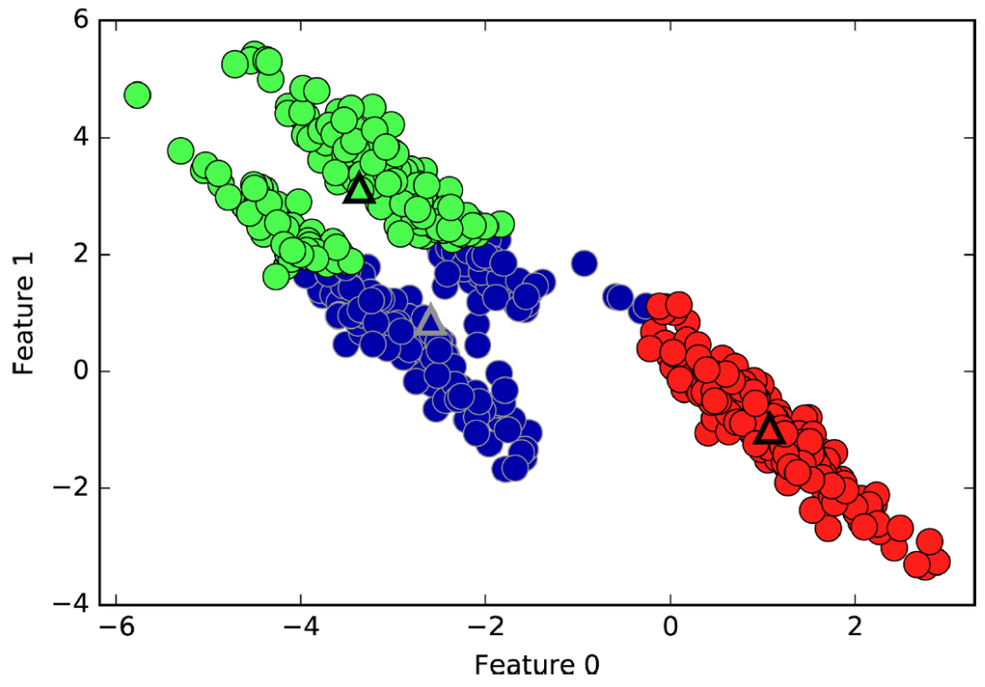


Scikit-learn: Limiti K-means

```
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)

# trasforma i dati per mezzo di una distribuzione gaussiana
transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)

kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm3)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
    marker='^', c=[0, 1, 2], s=100, linewidth=2, cmap=mglearn.cm3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

I dati sono distribuiti ("allungati") sulla diagonale, non seguono una distribuzione sferica.

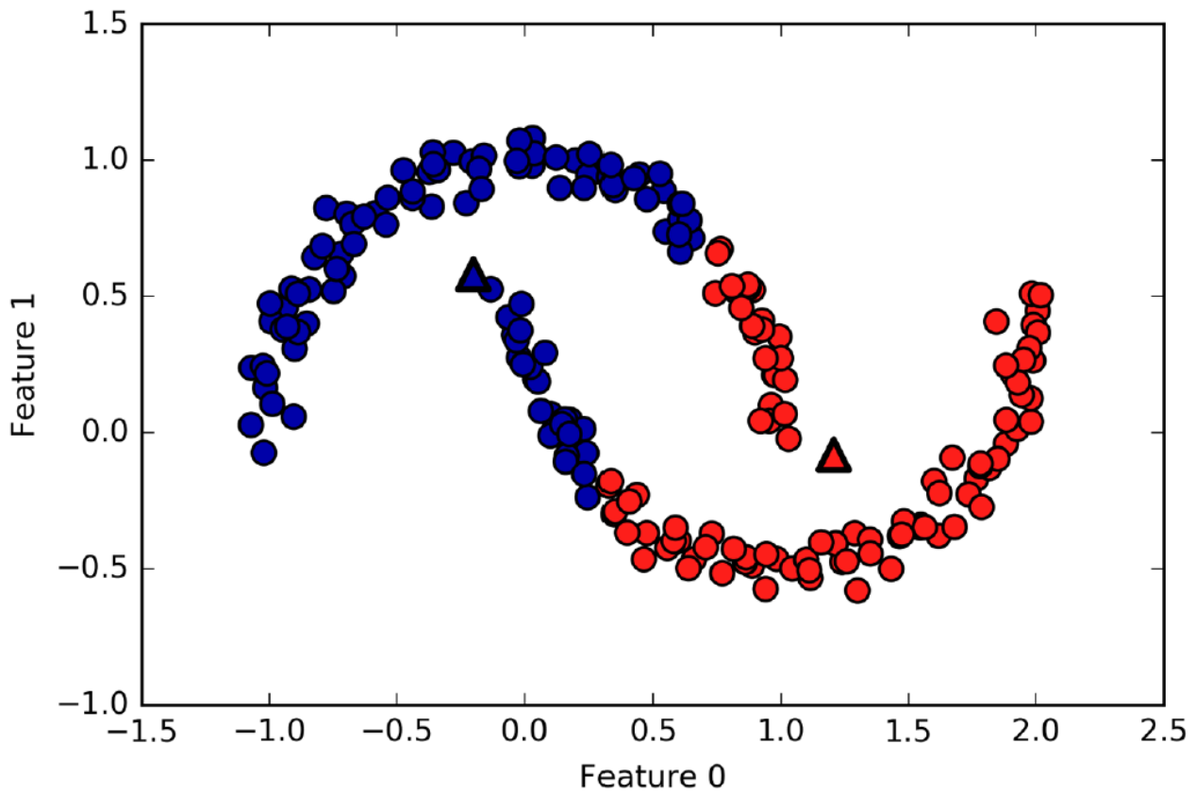
L'algoritmo valuta solo la distanza dal centroide.



Scikit-learn: Limiti K-means

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_pred = kmeans.predict(X))

plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm2, s=60)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
marker='^', c=[mglearn.cm2(0), mglearn.cm2(1)], s=100, linewidth=2)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Shape complesse non sono valutate correttamente.



Testi di Riferimento

- Andreas C. Müller, Sarah Guido. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media 2016
- Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media 2017