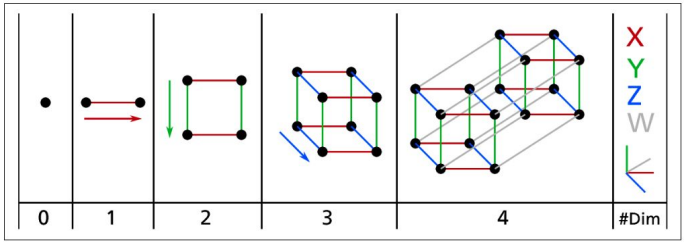
# Reducción de dimensiones
# Aprendizaje no Supervisado

FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LA REPÚBLICA URUGUAY

# Agenda

- Reducción de dimensiones:
  - Motivación
  - Enfoques principales: proyección y aprendizaje de variedades
  - Proyección
    - PCA - Principal component analysis
  - Aprendizaje de variedades
    - LLE - Locally linear embedding
- Aprendizaje no supervisado
  - Clustering
  - Estimación de densidades
  - Detección de anomalías

# Maldición de la dimensionalidad




- Hay “mucho espacio” en altas dimensiones
- Datos esparsos
  - Instancias de entrenamiento podrían estar muy separadas
  - Predicciones implican extrapolaciones muy grandes

**Probabilidad que un punto esté a menos de 0.001 de la frontera de un hipercubo unitario**

| n | 1 - 0.998^n |
|---|---|
| 2 | 0.3996% |
| 3 | 0.5988% |
| 4 | 0.7976% |
| 10 | 1.9821% |
| 100 | 18.1433% |
| 500 | 63.2489% |
| 1000 | 86.4935% |
| 5000 | 99.9955% |

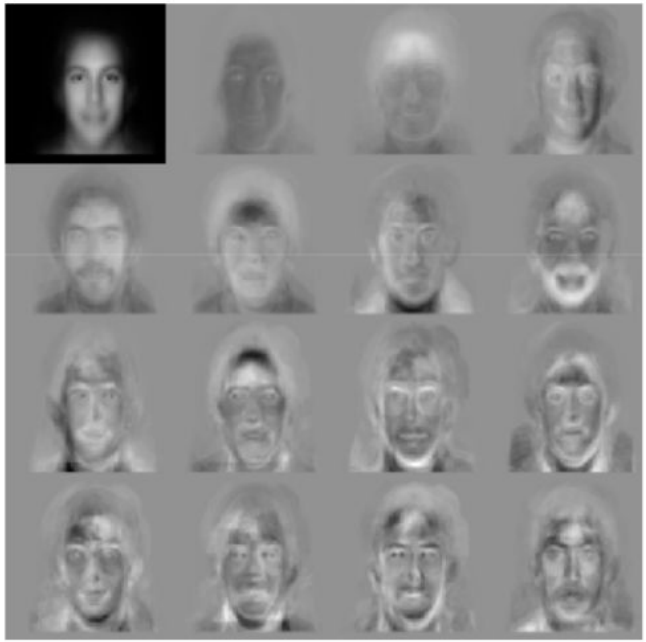
**Distancia media entre puntos aleatorios en un hipercubo unitario**

| n | cota inferior | cota superior |
|---|---|---|
| 2 | 0.4714 | 0.5450 |
| 3 | 0.5774 | 0.6818 |
| 4 | 0.6667 | 0.7950 |
| 5 | 0.7454 | 0.8938 |
| 10 | 1.0541 | 1.2778 |
| 100 | 3.3333 | 4.0784 |
| 1000 | 10.5409 | 12.9087 |
| 10000 | 33.3333 | 40.8244 |
| 100000 | 105.4093 | 129.0993 |
| 1000000 | 333.3333 | 408.2482 |

https://mathworld.wolfram.com/HypercubeLinePicking.html

# Bendición de la no uniformidad

- Los datos no se reparten uniformemente en el espacio
- Los datos “viven” realmente en espacios de menor dimensión

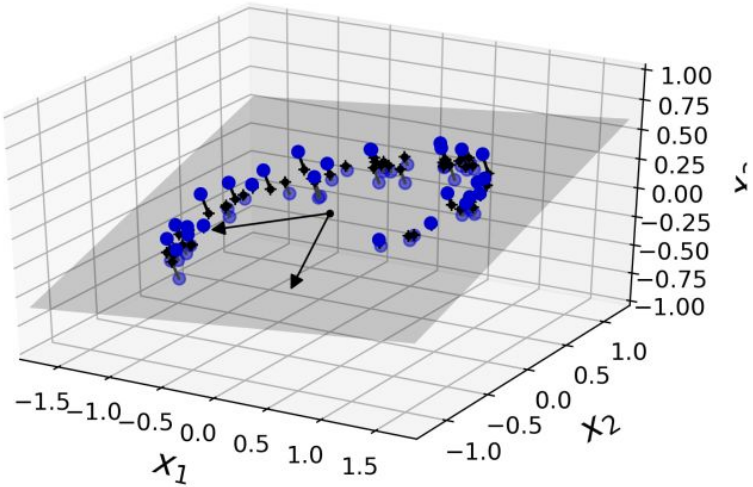


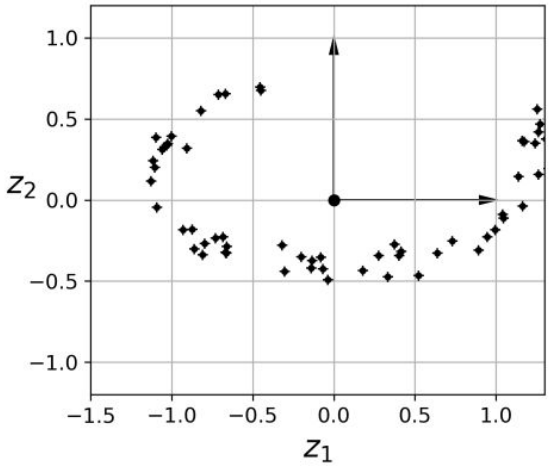
# Enfoques principales:  proyección y aprendizaje de variedades

# Enfoques principales

## Proyección

- Ejemplo:
  - Los datos del espacio ambiente (R3 ) se distribuyen cercanos a un plano (R2 ).
  - Proyectando ortogonalmente en ese plano obtenemos un conjunto de datos en R2 .

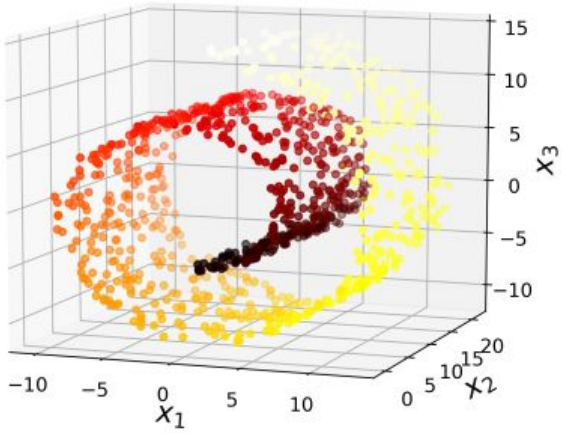


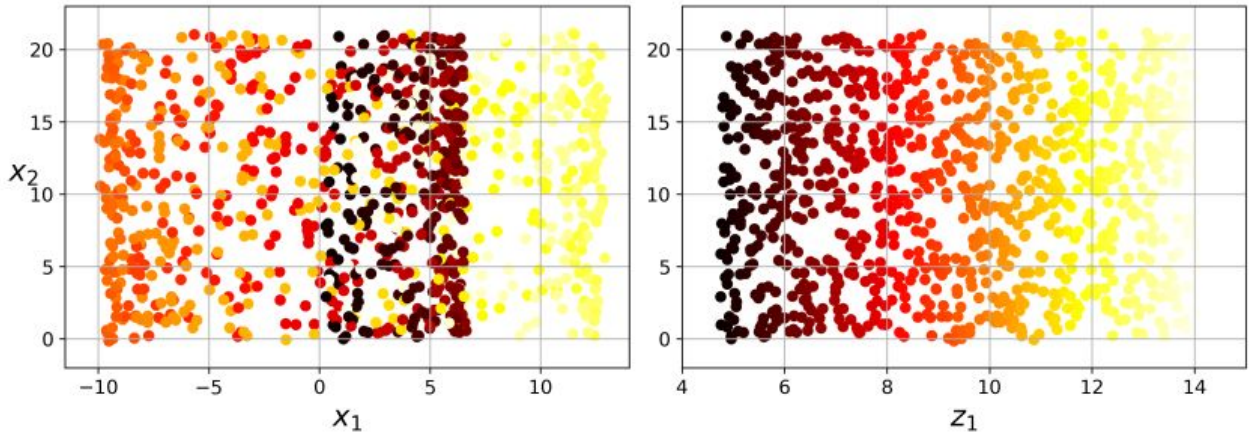



# Enfoques principales

## Proyección

- Limitante:
  - Existen datos distribuidos en subespacios (como el Swiss roll) que no se representan adecuadamente por un conjunto de proyecciones ortogonales.



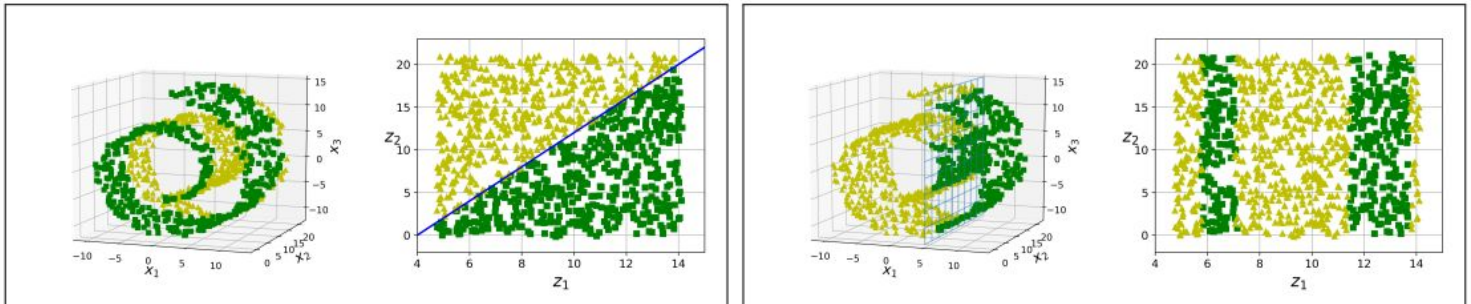



# Enfoques principales

**Manifold learning (aprendizaje de variedades)**

- Variedad: espacio topológico que localmente se asemeja al espacio euclideo cerca de cada punto. Una variedad n-dimensional es un espacio topológico con la propiedad de que cada punto tiene una vecindad que es homeomorfa $R^n$ .
- Ejemplos:
  - un círculo es una variedad de dimensión 1 (localmente homeomorfo a R)
  - el Swiss roll es una variedad de dimensión 2 (localmente homeomorfo a $R^2$

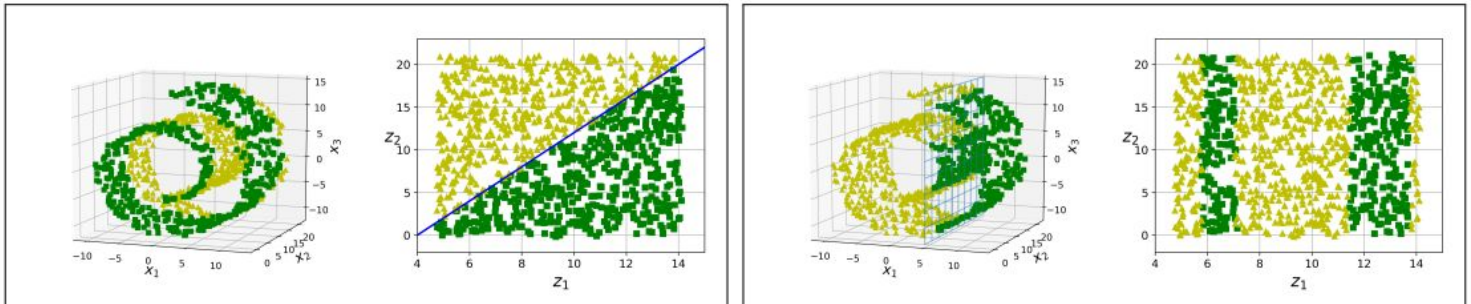
- Manifold learning:
  - métodos que modelan la variedad a la que pertenecen los datos.



# Enfoques principales

## Manifold learning (aprendizaje de variedades)

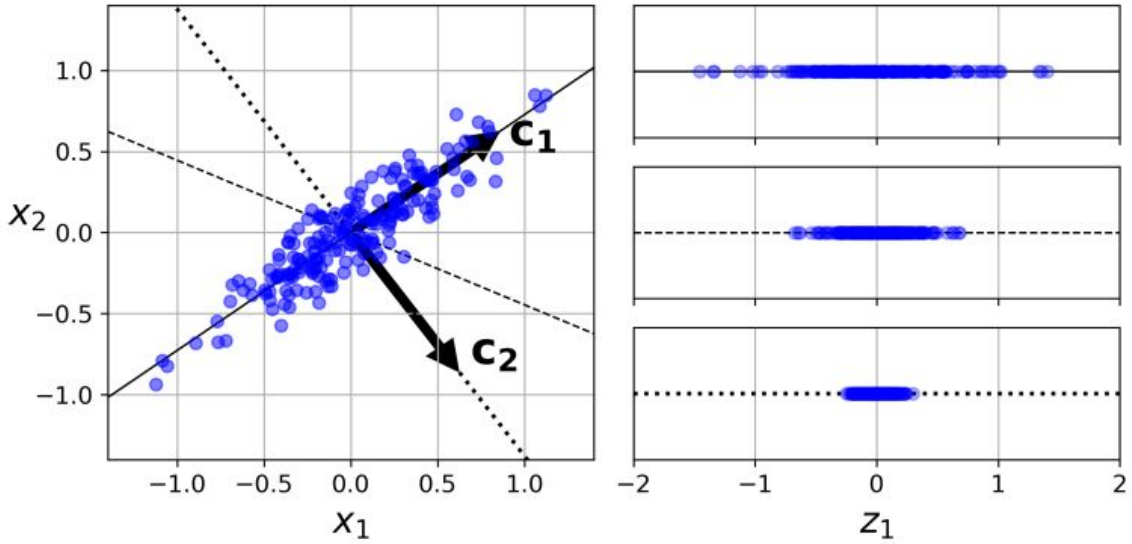
- Manifold assumption:
  - los datos se distribuyen cerca de una variedad. Los datos reales en general lo verifican.
- Ejemplo:
  - Imágenes de MNIST. El espacio ambiente es de dimensión 28 × 28 = 784, pero los dígitos tiene una estructura que restringe enormemente los grados de libertad.
- Supuesto implícito:
  - el método de predicción (clasificación o regresión) será más efectivo si los datos se representan en la variedad.



# Proyección

# Análisis de componentes principales (PCA)

- Consiste en encontrar un conjunto de direcciones ortogonales que mejor preservan la varianza de los datos, centrados en su media.
- Otra interpretación: los ejes principales son aquellas direcciones que minimizan el MSE entre el conjunto de datos y su proyección ortogonal.



# PCA en Scikit-Learn

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)

>>> pca.explained_variance_ratio_
array([0.84248607, 0.14631839])
```

La clase PCA implementa PCA mediante SVD. Centra los datos previo a calcular la SVD.

El método `explained_variance_ratio_` devuelve la proporción de la varianza total de cada componente principal.

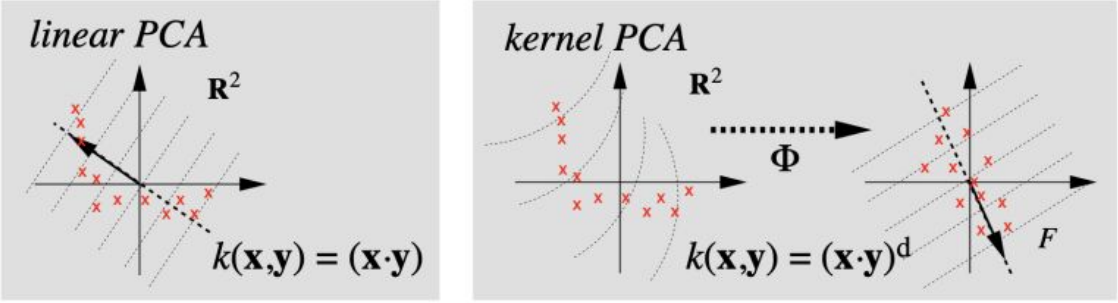
- Una buena forma de determinar la cantidad de componentes principales es considerar tantas hasta sumar una cantidad significativa de la varianza (e.g. 95%). Dos formas:

```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

# Kernel PCA

- Los métodos basados en kernels permiten transformar los datos a espacios de muy alta dimensión de forma implícita, y calcular los productos internos en ese espacio mediante una función de kernel correspondiente (*kernel trick*).
- En SVM, los kernels permiten encontrar hiperplanos separadores en el espacio transformado, que corresponden a fronteras de decisión no lineales en el espacio original.
- Kernel PCA: aplicar el *kernel trick* a PCA para lograr proyecciones no lineales complejas (lineales en el espacio transformado), que permitan reducir mejor la dimensionalidad.




*B. Schölkopf, A. Smola, and K.-R. Müller, "Kernel principal component analysis," in *Artificial Neural Networks — ICANN'97* (W. Gerstner, A. Germond, M. Hasler, and J.-D. Nicoud, eds.), (Berlin, Heidelberg), pp. 583–588, Springer Berlin Heidelberg, 1997

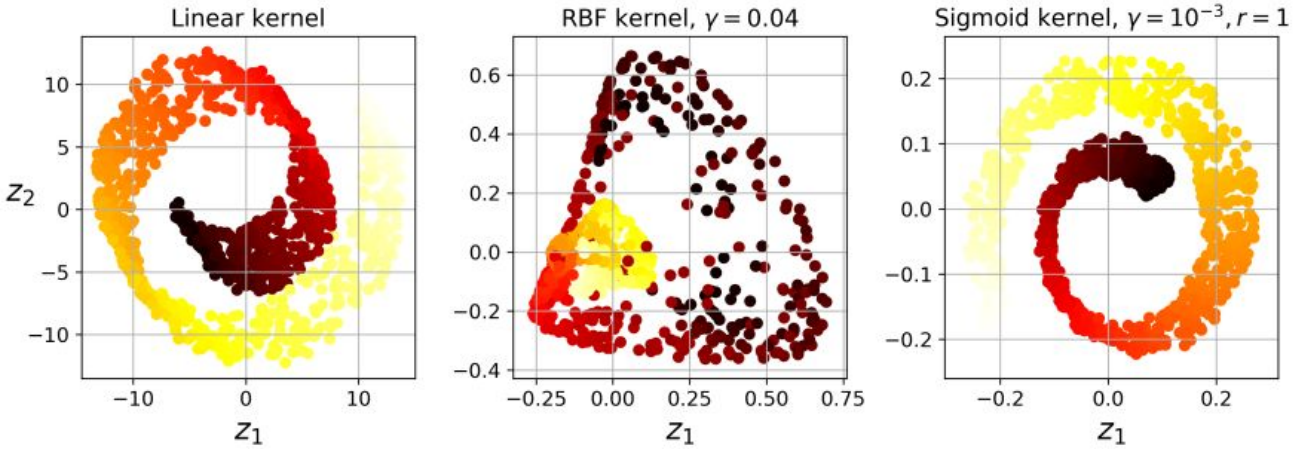
# Kernel PCA en Scikit-Learn

- Cálculo de dos componentes principales de Kernel PCA con kernel RBF.

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

- Resultados para distintos kernels sobre *Swiss roll*:



# Kernel PCA en Scikit-Learn

Selección del Kernel e hiperparámetros: Al igual que para SVM, esto se efectúa ensayando distintos kernels y buscando sus parámetros óptimos mediante grid search y validación cruzada.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression())
    ])

param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)

>>> print(grid_search.best_params_)
{'kpca__gamma': 0.043333333333333335, 'kpca__kernel': 'rbf'}
```

# Otras variantes de PCA

- Randomized PCA
  - Limitar el cálculo a una estimación aproximada de los vectores singulares
  - Baja el costo computacional

  - En Scikit-Learn el parámetro svd_solver controla el tipo de algoritmo de SVD.
    - Randomized PCA: “randomized”, SVD convencional: “full”.
    - En “auto”: usa Randomized PCA si m o n exceden 500 y si d es menor al 80% de m o n.

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

- Incremental PCA
  - PCA clásico requiere cargar todos los datos en memoria
  - Permite actualizar la proyección con nuevos datos en forma incremental

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

# Manifold learning

# Locally linear embedding (LLE)

- Técnica de manifold learning que no se basa en proyecciones.
- Efectúa reducción de dimensionad no lineal.
- Busca preservar la estructura local de distancias del espacio original en el espacio reducido, asumiendo que la variedad es localmente lineal.

Dos etapas:

1. Para cada punto $x_i$, evaluar cómo se puede representar como combinación lineal de sus vecinos $k$ vecinos más cercanos $x_j$, $j \in \mathcal{N}_i$.
2. Luego, se busca el conjunto de puntos transformados que preserven esos coeficientes en su representación lineal.

---

*S. T. Roweis and L. K. Saul, "Nonlinear Dimensionality Reduction by Locally Linear Embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000

# Locally Linear Embedding (LLE)

1. Estimar los coeficientes de las combinaciones lineales locales (espacio ambiente, dim $n$):

$$\widehat{\mathrm{W}} = \underset{\mathrm{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left( \mathrm{x_i} - \sum_{j \in \mathcal{N}_i} w_{i,j} \mathrm{x_j} \right)^2, \quad \text{s.t.} \quad \sum_{j \in \mathcal{N}_i} w_{i,j} = 1.$$

2. Encontrar el conjunto de datos de dim $d < n$ que responde a la misma estructura local:

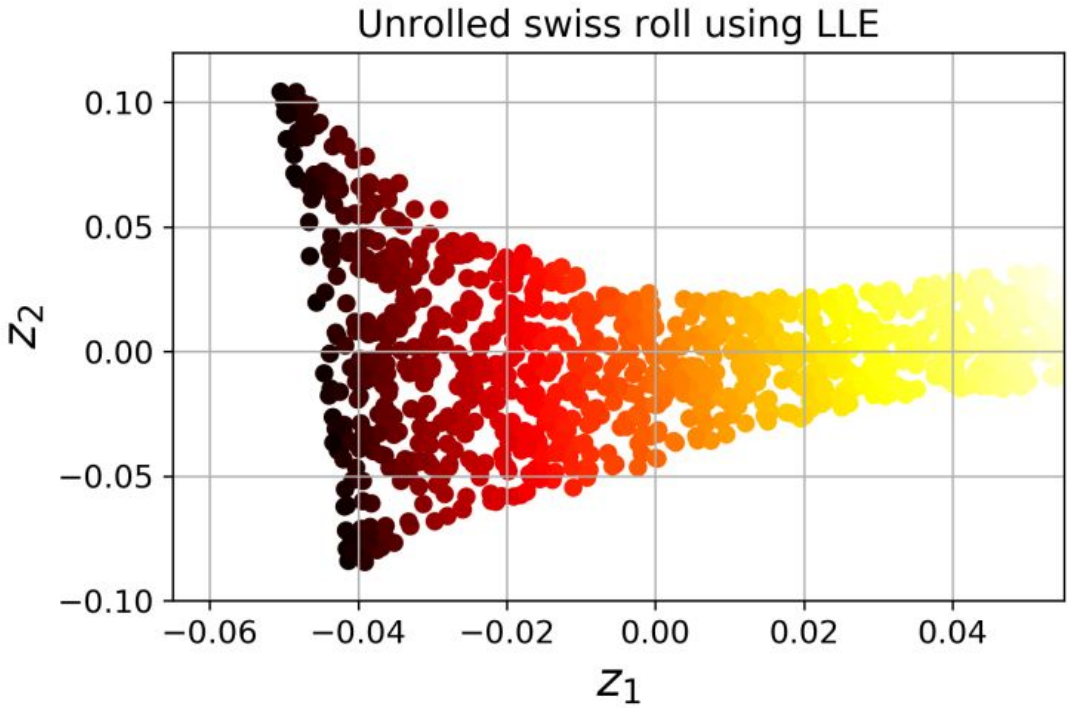
$$\widehat{\mathrm{Y}} = \underset{\mathrm{Y}}{\operatorname{argmin}} \sum_{i=1}^{m} \left( \mathrm{y_i} - \sum_{j \in \mathcal{N}_i} \widehat{w}_{i,j} \mathrm{y_j} \right)^2,$$

$$\text{s.t.} \quad \sum_{i=1}^{m} \mathrm{y_i} = 0, \quad \frac{1}{m} \sum_{i=1}^{m} \mathrm{y_i} \mathrm{y_i}^T = I.$$

# Locally Linear Embedding (LLE)

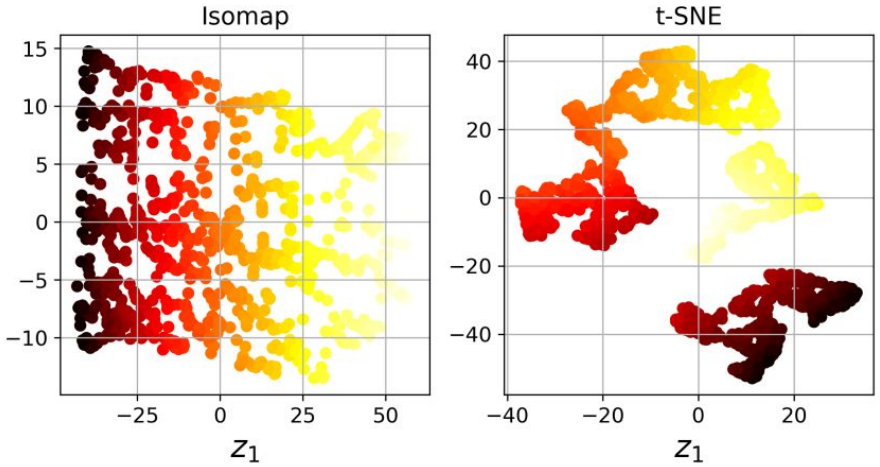
```python
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```



# Otras técnicas de reducción de dimensionalidad

- Random projections
- Multidimensional scaling (MDS)
- Isomap
- t-Distributed Stochastic Neighbor Embedding (t-SNE)
- Linear Discriminant Analysis (LDA)
- Laplacian Eigenmaps



# Aprendizaje no supervisado

# Aprendizaje no supervisado

- La mayor parte de los datos disponibles no están etiquetados. Su potencial etiquetado requiere intervención humana (muchas veces expertos especializados) y su costo es muy alto.

**Aprendizaje no supervisado**

- Métodos de reducción de dimensionalidad
- Clustering:
  - descubrir estructura dentro de un conjunto de datos, agrupándolos en subconjuntos (clusters) que muestren una cierta coherencia o similitud interna.
- Estimación de densidades:
  - estimar las densidades de probabilidad subyacentes a la generación de los datos observados.
- Detección de anomalías:
  - detectar aquellas muestras con baja probabilidad de haber sido generadas por el modelo de normalidad que explica los datos (muestras alejadas de los clusters, o muestras poco probables de acuerdo a las densidades estimadas).

# Clustering

# Clustering

- **Objetivo:** descubrir estructura dentro de un conjunto de datos D = {x1 , . . . , xn }, dividiéndolo en subconjuntos que muestren una cierta coherencia.
  - **Coherencia**: muestras dentro de un mismo grupo o cluster son más parecidas entre sí que a las muestras de otros clusters.
  - **Muestras parecidas**: noción de similitud o de distancia entre muestras.

- La mayor parte de los métodos de clustering son de dos tipos:

  a) **Particionales**: producen una única partición, D1 , . . . , Dc , que optimiza una función criterio

  b) **Jerárquicos**: jerarquía de particiones anidadas; cada nivel de la jerarquía es en si mismo una partición, obtenida por unión de clusters de la jerarquía inferior.

- Importante:

  Un método de clustering siempre produce clusters, aunque éstos no existan realmente ⇒ todo método de clustering debe ser seguido de una etapa de validación de los clusters obtenidos.

# Medidas de similitud entre muestras

- Debe ser adaptada al problema en particular. Elección no es trivial
- No tiene por qué ser una distancia

- Si se usan distancias
  - Correlaciones entre características pueden distorsionar estas distancias; es común normalizar los datos previamente.
  - Ejemplo de distancias

Ejemplo: Distancias de Minkowski (normas $\ell^p$).
$p = 2$: Euclidea; $p = 1$: Manhattan; $p = \infty$: $d(\mathrm{x}, \mathrm{x}') = \max_i |x_i - x_i'|$

$$d(\mathrm{x}, \mathrm{x}') = \left( \sum_{i=1}^{d} |x_i - x_i'|^p \right)^{1/p}, \quad p \geq 1.$$

# Funciones criterio

- Evalúan la calidad de una partición.
  - De la elección del criterio dependerá la forma de los clusters que se obtienen al optimizarlo.

- Ejemplo:
  Criterio SSE (sum of squared errors), criterios de mínima varianza

$$SSE = \sum_{i=1}^{c} \sum_{x \in \mathcal{D}_i} \|x - \boldsymbol{\mu}_i\|_2^2, \text{ con } \boldsymbol{\mu}_i = \frac{1}{n_i} \sum_{x \in \mathcal{D}_i} x, \; n_i = \#\mathcal{D}_i.$$

# Clustering: K-Means

# K-Means

- Es uno de los métodos particionales.
- Consiste en una minimización iterativa del SSE

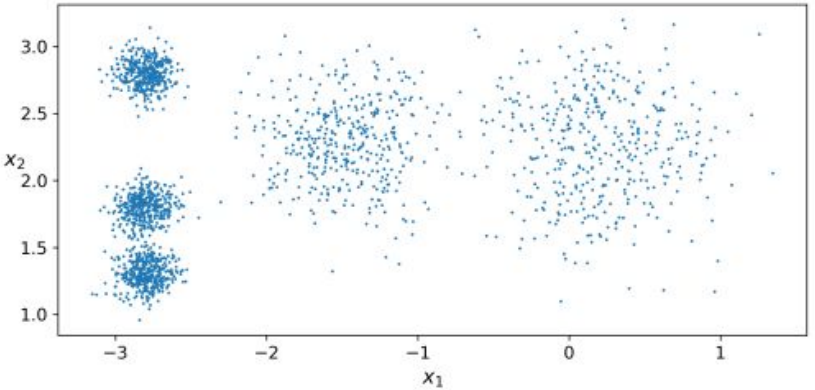
- Algoritmo:

  1. Elegir una partición inicial en c grupos al azar
  2. Calcular las medias μi de cada cluster
  3. Seleccionar secuencialmente un punto x ∈ D , y si corresponde, reasignarlo al cluster que m $\|x - \mu_i\|_2$
  4. Si no hay más reasignaciones en todo D , terminar; si no, volver al paso 2.

# K-Means en Scikit Learn

Entrenemos k-means sobre el conjunto de datos de la figura.

- Fijamos k=5 clusters
- Creamos una instancia de la clase `KMeans` con 5 clusters
- Corremos el método `fit_predict`.




```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

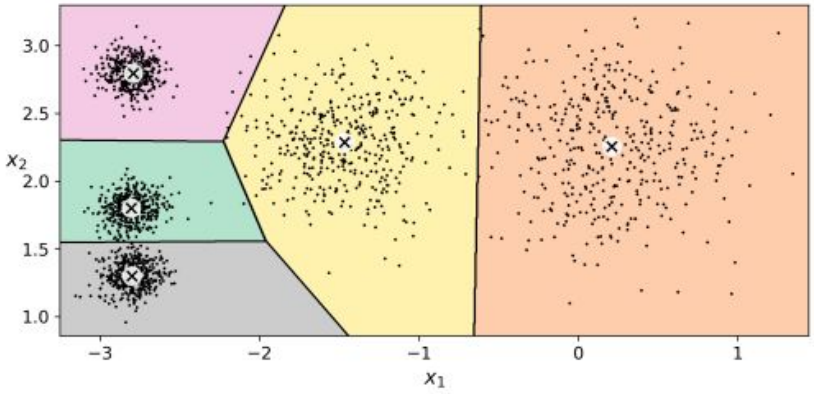
El método devuelve un vector `y_pred` en la variable `labels_` de la instancia, que asigna una etiqueta correspondiente al cluster al cual es asignado cada muestra.

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

La variable `cluster_centers_` contiene la coordenada de los centroides de los clusters.

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

# K-Means en Scikit Learn




También podemos caracterizar un punto según su distancia a los centroides.

```
>>> 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]])
```

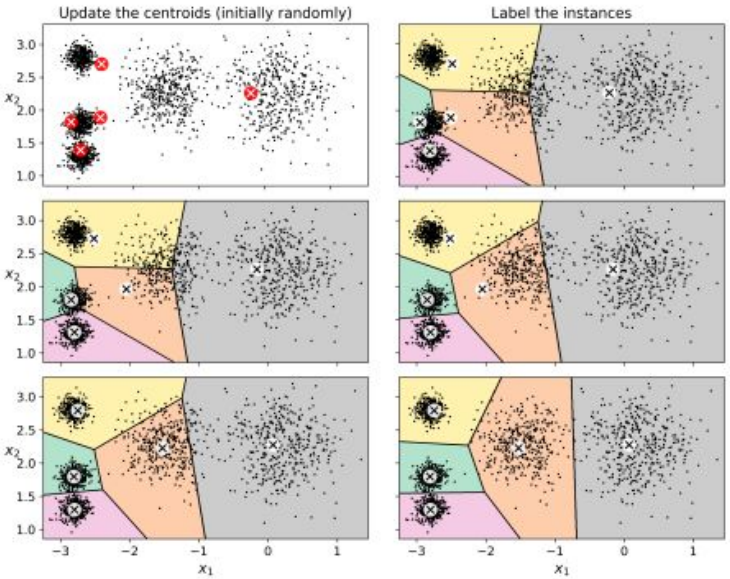
Podemos predecir el cluster al cual pertenecen nuevas muestras (se las asigna al cluster cuyo centroide está más cerca):

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

Puede usarse para de reducción de dimensionalidad, de la dimensión $m$ original a la dimensión $k$.

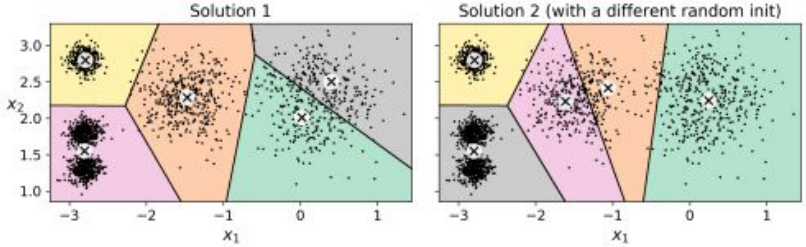
# K-Means - Inicialización

Evolución con un buena inicialización.




En este caso converge en tres iteraciones.

Otras inicializaciones pueden conducir a soluciones sub-óptimas.




Cómo resolvemos esto:

- Métodos de inicialización.
- Métricas de calidad de la partición.

# K-Means - Inicialización

## Métodos de inicialización de los centroides

- Inicialización manual: cuando los centroides se conocen aproximadamente, se pueden especificar con el hiperparámetro `init`, y seteando `n_init=1`:

```
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)
```

- Correr k-means con varias inicializaciones al azar, conservando la solución que alcanza el menor costo, e.g. `inertia_`: $SSE = \sum_{i=1}^{c} \sum_{x \in \mathcal{D}_i} \|x - \boldsymbol{\mu}_i\|_2^2$. Se usa `init='random'`.

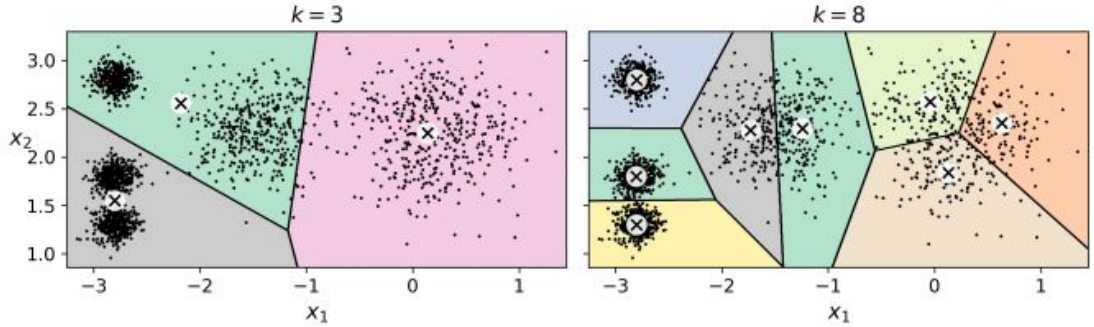
- Algoritmo k-means++*: elige los centroides al azar favoreciendo que estén lo más alejados unos de otros. Es la inicialización por defecto (`init='k-means++'`).

---

*D. Arthur and S. Vassilvitskii, "k-means++: the advantages of careful seeding," in *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, (Philadelphia, PA, USA), pp. 1027–1035, Society for Industrial and Applied Mathematics, 2007

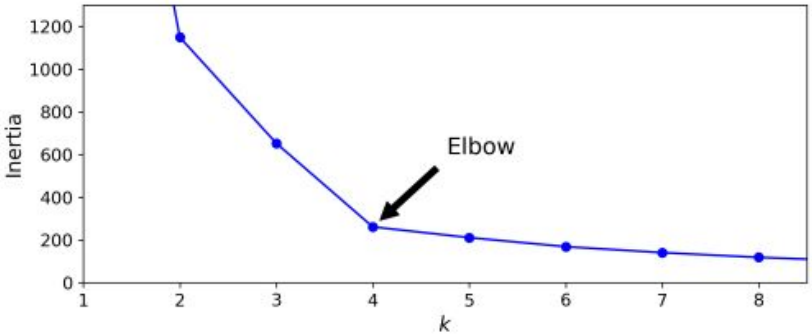
# K-Means - Cantidad de clusters ? Estimación basada en la inercia

En general desconocemos cuántos clusters hay. En el ejemplo anterior:




El valor de la inercia no es una medida adecuada: tiende a bajar cuanto más grande es $k$.

Método gráfico basado en comportamiento de la inercia con K: en general la cantidad de clusters óptima se encuentra cerca del codo (cambio de comportamiento).



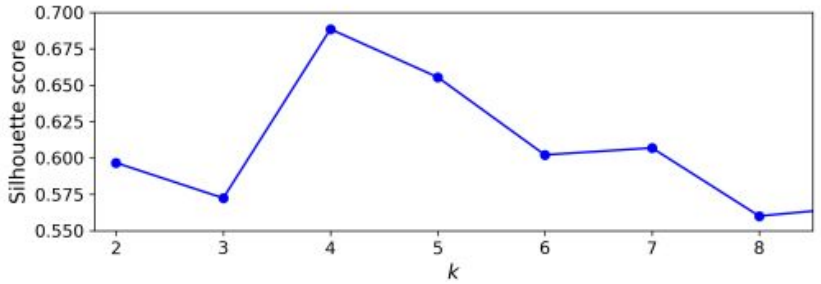
# K-Means - Cantidad de clusters ? Estimación basada en silhouettes

Silouettes: método gráfico más preciso pero computacionalmente más costoso.

- Se define el coeficiente *silouette* de una muestra como $s = (b - a)/\max(a, b)$, con $a$ distancia media a puntos dentro del cluster, y $b$ la distancia media a los puntos del siguiente cluster más cercano.
- $s \in [-1, +1]$; $s = 1$ es un punto centrado en el cluster, $s = 0$ es un punto cercano al borde del cluster, $s = -1$ indica que el punto puede haber sido asignado a un cluster incorrecto.

Coeficiente medio de silhouette:

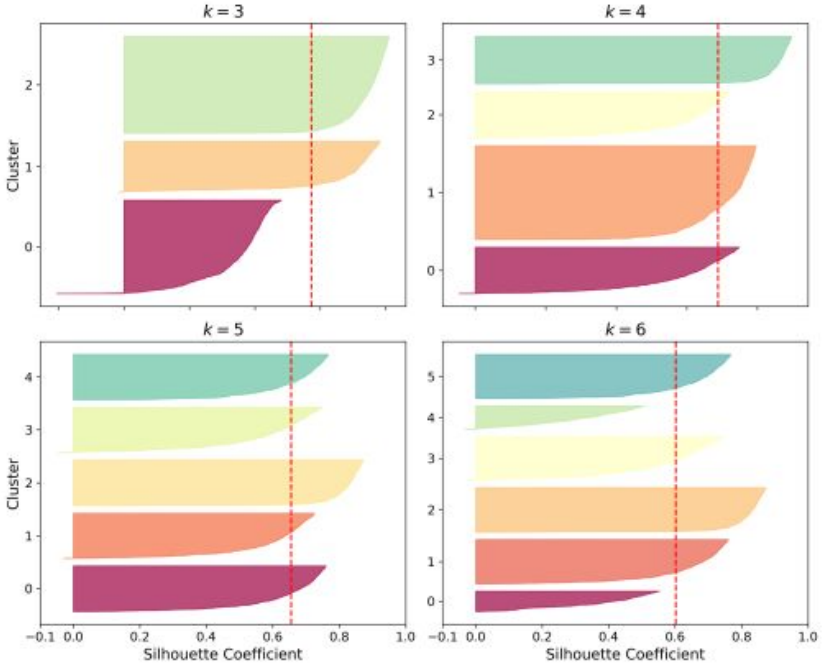
```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```



# K-Means - Cantidad de clusters ? Estimación basada en silhouettes

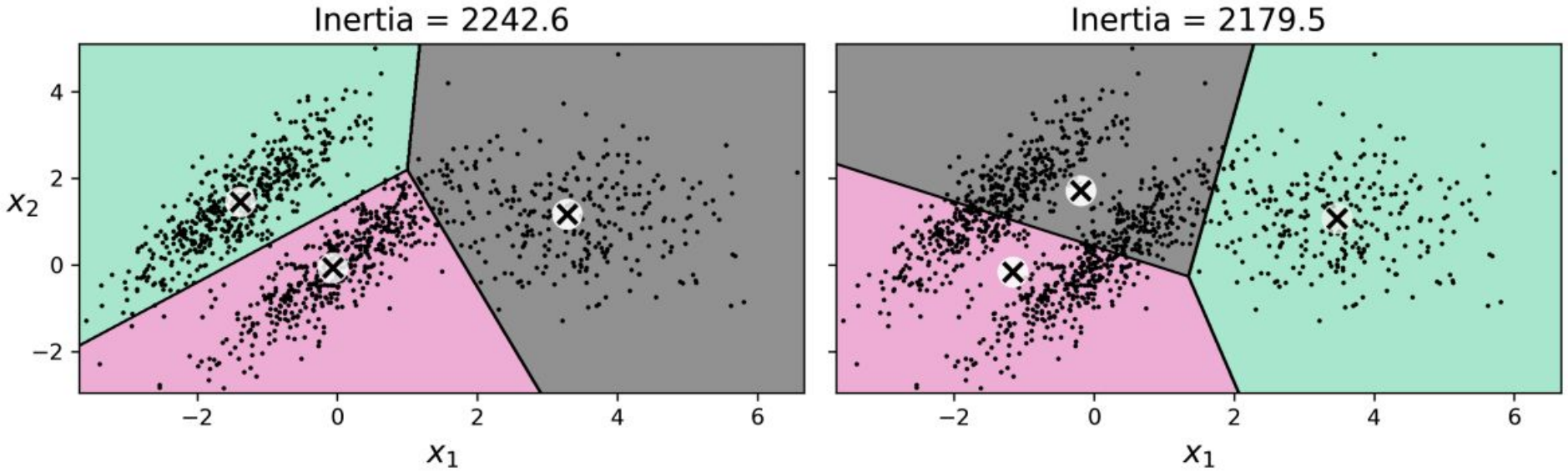
Diagrama de silhouettes:

- coeficiente de silhouette de cada muestra, de mayor a menor, dentro de cada cluster
- buenos $k$ tienen los coeficientes distribuidos por encima del score de silhoutte medio



# Debilidades de K-Means

- Es muy usado pero...
- Hay que estimar K
- Requiere cierta uniformidad.
  Algunos problemas con clusters de diferentes tamaños, diferentes densidades o cluster no isotrópicos

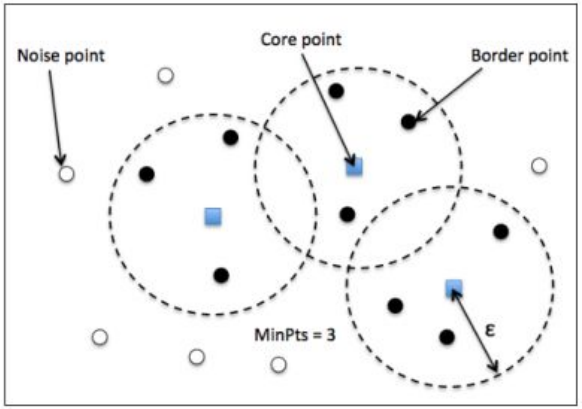


# Clustering: DBSCAN

# Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

Define clusters como regiones continuas de alta densidad, de la manera siguiente:

- Para cada punto, calcular cuántos puntos se encuentran a distancia inferior a ε (ε-vecindad).
- Un punto con al menos min_samples puntos en su ε-vecindad, se lo considera un núcleo.
- Todos los puntos en la ε-vecindad de un núcleo pertenecen al mismo cluster. Esta ε-vecindad puede incluir otros núcleos; y así formar un mismo cluster.
- Un punto que no es un núcleo y no tiene otro núcleo en su ε-vecindad es considerado ruido.



# DBSCAN en Scikit Learn

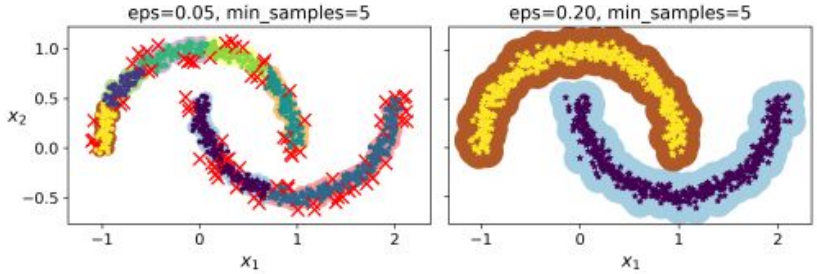
Base de datos `moons`.

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

```
>>> dbscan.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0, ...,  3,  2,  3,  3,  4,  2,  6,  3])

>>> len(dbscan.core_sample_indices_)
808
>>> dbscan.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])
>>> dbscan.components_
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
                  ...
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```




DBSCAN permite encontrar clusters de formas arbitrarias. Funciona correctamente si los clusters son suficientemente densos y está separados por regiones poco densas.

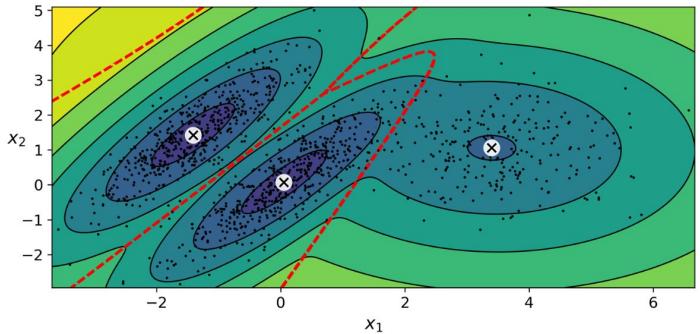
# Mezcla de gaussianas

# Mezcla de gaussianas (GMM)

Modelo probabilístico que asume que las muestras {x1 , x2 , . . . , xm } son independientes, y fueron generadas por un proceso cuya densidad es una mezcla de densidades gaussianas, de parámetros desconocidos. Asumimos que el número de clusters k es conocido.

## Modelo generativo:

1. A cada muestra $x_i$ se le asocia una variable categórica $z_i \in \{1, 2, \ldots, k\}$, tal que $P(z_i = j) = \phi_j$. El suceso $\{z_i = j\}$ indica que $x_i$ es asignada al cluster $j$-ésimo.
2. Si $\{z_i = j\}$, $x_i$ se obtiene como realización del proceso gaussiano de media $\mu_j$ y matriz de covarianza $\Sigma_j$, i.e. $x_i \sim \mathcal{N}(\mu_j, \Sigma_j)$.



# GMM en Scikit Learn

Dada la matriz de datos $X \in \mathbb{R}^{m \times n}$, los parámetros $\{\phi_j, \boldsymbol{\mu}_j, \Sigma_j, j = 1, \ldots, k\}$ se pueden obtener mediante la clase `GaussianMixture`, que implementa el algoritmo EM (Expectation-Maximization) para gaussianas.

EM es un algoritmo iterativo que para GMMs garantiza convergencia a un óptimo local.

- `n_components`: cantidad de gaussianas en la mezcla
- `n_init`: cantidad de inicializaciones. Se prueban todas y se conserva la solución que mejor ajusta la mezcla de gaussianas.
- Se puede ver a posteriori si el algoritmo convergió y en cuántas iteraciones, inspeccionando las variables `converged_` y `n_iter_`.

```
from sklearn.mixture import GaussianMixture
gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
>>> gm.means_
array([[ 3.39909717,  1.05933727],
       [-1.40763984,  1.42710194],
       [ 0.05135313,  0.07524095]])
>>> gm.covariances_
array([[[ 1.14807234, -0.03270354],
        [-0.03270354,  0.95496237]],

       [[ 0.63478101,  0.72969804],
        [ 0.72969804,  1.1609872 ]],

       [[ 0.68809572,  0.79608475],
        [ 0.79608475,  1.21234145]]])
```

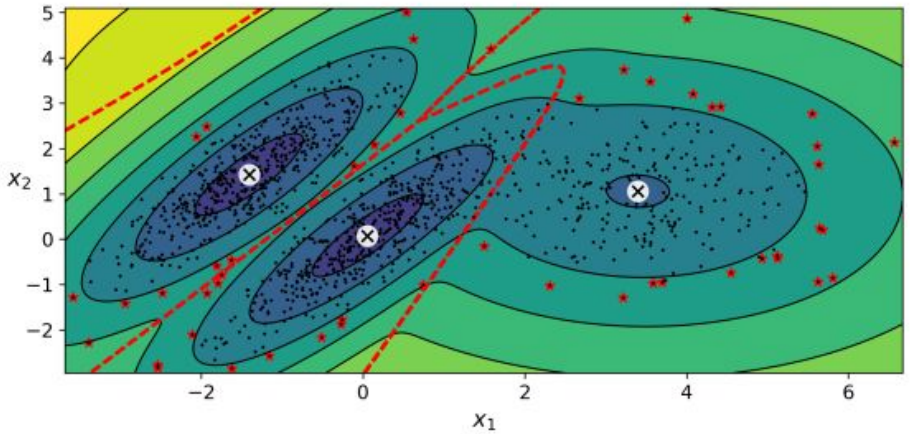
# Detección de anomalías con GMM

- El método `score_samples()` devuelve el logaritmo de la densidad a posteriori de la mezcla para una muestra:

```
>>> gm.score_samples(X)
array([-2.60782346, -3.57106041, -3.33003479, ..., -3.51352783,
       -4.39802535, -3.80743859])
```

- Usando este método podemos decidir que el 4% de las muestras menos probables de haber sido generadas por la mezcla (las de menor densidad a posteriori) son anomalías:

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```




FACULTAD DE
INGENIERÍA
UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY