Taller de Aprendizaje Automático

Reducción de Dimensionalidad. Aprendizaje no Supervisado

Instituto de Ingeniería Eléctrica
Facultad de Ingeniería

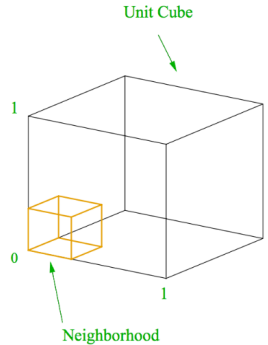
UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

1. Reducción de dimensionalidad
   - Motivación y enfoques principales
   - Análisis de componentes principales
   - Locally Linear Embedding (LLE)

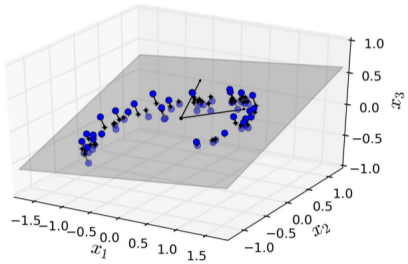
2. Aprendizaje no supervisado
   - Clustering: nociones generales
   - Clustering basado en distancias: k-means
   - Clustering basado en densidad: DBSCAN
   - Mezcla de gaussianas (GMM)

Reducción de dimensionalidad: motivación

- Maldición de la dimensionalidad
- Bendición de la no uniformidad



- Reducción del costo computacional
- Visualización de los datos



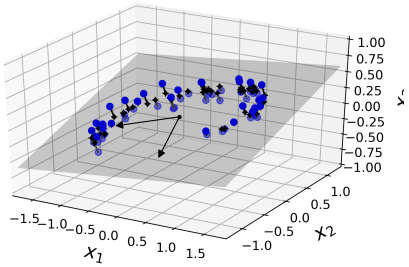
Enfoques principales

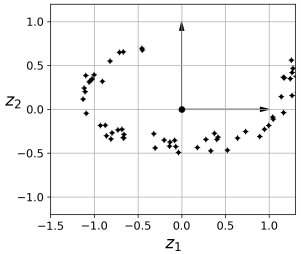
Proyección

- En general los datos no se distribuyen uniformemente en el *espacio ambiente*.
- Es usual que los datos "vivan" en un *subespacio* de dimensión muy inferior.

Ejemplo:

- Los datos del espacio ambiente ($\mathbb{R}^3$) se distribuyen cercanos a un plano ($\mathbb{R}^2$).
- Proyectando ortogonalmente en ese plano obtenemos un conjunto de datos en $\mathbb{R}^2$.

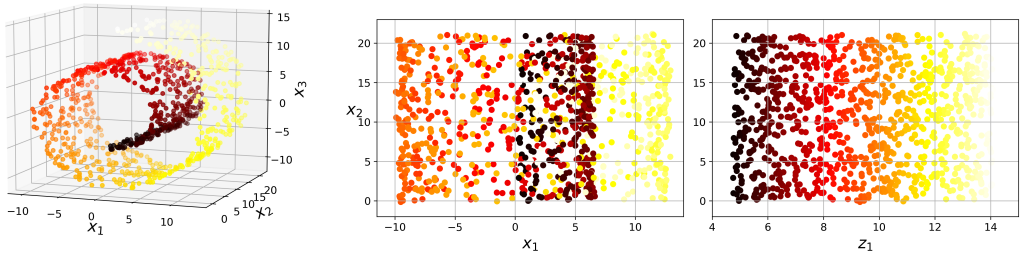




Enfoques principales (cont.)

Proyección: limitantes

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

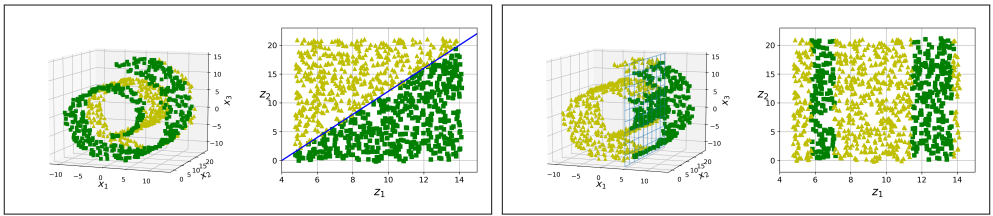


- Afortunadamente, existen técnicas alternativas, como el aprendizaje de variedades.

Enfoques principales (cont.)

Manifold learning (aprendizaje de variedades)

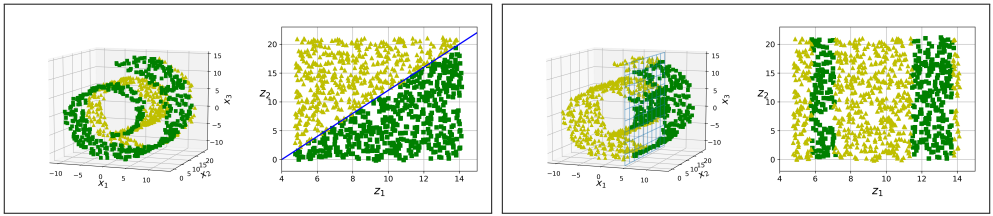
- *Variedad:* espacio topológico que localmente se asemeja al espacio euclideo cerca de cada punto. Una variedad n-dimensional es un espacio topológico con la propiedad de que cada punto tiene una vecindad que es homeomorfa $\mathbb{R}^n$.
- *Ejemplos:* un círculo es una variedad de dimensión 1 (localmente homeomorfo a $\mathbb{R}$); el *Swiss roll* es una variedad de dimensión 2 (localmente homeomorfo a $\mathbb{R}^2$).
- *Manifold learning:* métodos que modelan la variedad a la que pertenecen los datos.



Enfoques principales (cont.)

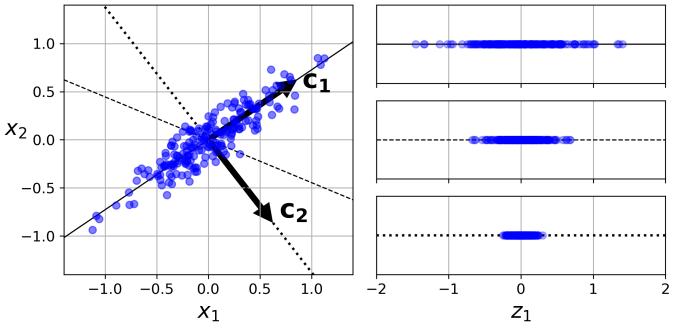
Manifold learning (aprendizaje de variedades)

- *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 $28 \times 28 = 784$, 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. ⚠



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.
- ⚠ Los ejes principales del PCA no son robustos. Como resultan de minimizar el MSE, grandes perturbaciones en pocos datos cambian su dirección.



PCA mediante la descomposición en valores singulares (SVD)

Descomposición en valores singulares

- Dada una matriz $A \in \mathbb{R}^{m \times n}$ cualquiera,

$$\mathrm{SVD}(A) = U \Sigma V^T = \sum_{i=1}^{n} \Sigma_{ii} \mathbf{u}_i \mathbf{v}_i^T,$$

  con $U \in \mathbb{R}^{m \times m}$ y $V \in \mathbb{R}^{n \times n}$ matrices ortogonales, y $\Sigma \in \mathbb{R}^{m \times n}$ matriz rectangular-diagonal con los $\Sigma_{ii} \geq 0$ los valores singular de A. ($\mathbf{u}_i$ y $\mathbf{v}_i$ son las columnas de U y V).
- La descomposición no es única. Si se ordenan los Σ_{ii} de mayor a menor, es única si no hay valores singulares iguales. Si los hay, es única a menos de permutaciones en U y V.

Análisis en componentes principales (cont.)

- Tenemos m datos $\mathbf{x}_i$ de dimensión n (en general $m \gg n$).
- Construimos la matriz de datos $X \in \mathbb{R}^{m \times n}$, donde las filas corresponden a los datos, luego la centramos (sustrayendo en cada $\mu = \frac{1}{m} \sum_{i=1}^{m} \mathbf{x}_i$).
- La matriz de covarianza viene dada por

$$C = \frac{1}{m-1} \bar{X}^T \bar{X}.$$

- Utilizando la descomposición SVD de $\bar{X}$:

$$C = \frac{1}{m-1} V \Sigma^T U^T U \Sigma V^T = \frac{1}{m-1} V \Sigma^T \Sigma V^T.$$

- De esta forma vemos que :
  - Los ejes principales son las columnas de V.
  - Las varianzas en las direcciones principales son $\sigma_i^2 = \frac{\Sigma_{ii}^2}{m-1}$.

Análisis en componentes principales (cont.)

En suma, para obtener la base de vectores principales:

1. Se construye la matriz de datos centrada $\bar{X}$
2. Se calcula SVD$(\bar{X}) = U\Sigma V^T$.

Resultado:

- La base del PCA son las columnas de $V = [v_1, v_2, \ldots, v_n]$.
- La i-ésima dirección principal (en orden de importancia) es v_i.
- Las varianzas correspondientes son $\sigma_i^2 = \Sigma_{ii}^2/(m-1)$ (con $\sigma_1^2 \geq \sigma_2^2 \geq \cdots \geq \sigma_n^2$).
- Los datos, expresados en la base PCA, son las filas de $X_{PCA} = \bar{X} \cdot V$.
- Inversamente, $\bar{X} = X_{PCA} \cdot V^T$.

Reducción a $d < n$ dimensiones:

- En la base PCA: $X_{PCA_{\{1:d\}}} = \bar{X} \cdot V_{\{1:d\}}$.
- Reconstrucción o aproximación en la base canónica (original): $\bar{X}_d = X_{PCA_{\{1:d\}}}(V_{\{1:d\}})^T$.

PCA en Python: NumPy

Ejemplo: proyección del conjunto de datos 3D en los dos ejes principales

Centrado de datos, SVD y extracción de las dos primeras columnas de V:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

Proyección de los datos centrados en el plano:

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

PCA en Python: 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
```

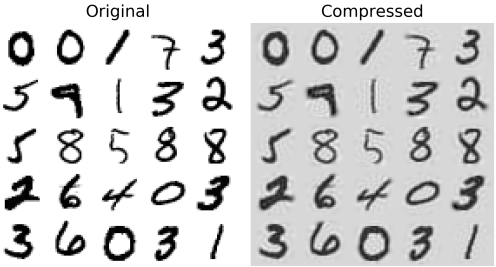
```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

PCA como método de compresión

Ejemplo: MNIST

- Aplicando `PCA` con `n_components=0.95` se puede ver que el 95% de la varianza se concentra en los primeros 154 componentes (del total de $28 \times 28 = 784$).
- Podemos reconstruir la imagen volviendo al espacio original invirtiendo el cambio de base:

$$X_{comprimida} = X_{PCA_{\{1:154\}}}(V_{\{1:154\}})^T.$$



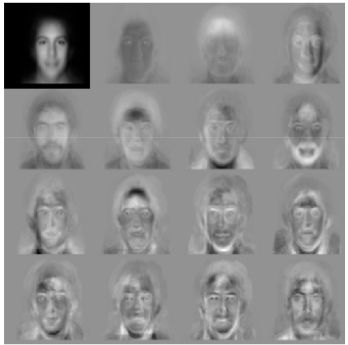
154 componentes, 5% error de reconstrucción

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

PCA como método de compresión (cont.)

Ejemplo: Eigenfaces*†

A partir de 7562 imágenes de caras de 128×128, se obtienen los 15 vectores propios principales. Imagen es combinación lineal de las otras 15. Método clásico de reconocimiento de caras.



*L. Sirovich and M. Kirby, "Low-dimensional procedure for the characterization of human faces," *J. Opt. Soc. Am. A*, vol. 4, pp. 519–524, Mar 1987

†M. Turk and A. Pentland, "Eigenfaces for recognition," *J. Cognitive Neuroscience*, vol. 3, p. 71–86, Jan. 1991

Randomized PCA*

- La complejidad computacional de la SVD es $O(mn^2 + n^3)$.
- Limitar el cálculo a una estimación aproximada de los vectores singulares.
- Para Randomized PCA, es $O(md^2 + d^3)$, acelerando dramáticamente para $d \ll n$.

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)
```

*N. Halko, P. G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM Review*, vol. 53, no. 2, pp. 217–288, 2011

PCA Incremental*

- PCA clásico requiere cargar todos los datos en memoria.
- Es posible escribir los vectores columna de U y V de forma recursiva, lo cual permite obtener algoritmos que van actualizando U y V a medida que se agregan datos.
- Existen varios métodos de PCA Incremental. Scikit-Learn usa el método de Ross et al.*

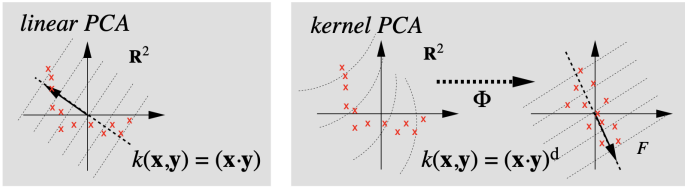
```
from sklearn.decomposition import IncrementalPCA

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)
```

Reducción de dimensionalidad a 154 usando la clase `IncrementalPCA`, con mini-batches de 100 imágenes.

*D. A. Ross, D. A. Ross, J. Lim, J. Lim, R.-S. Lin, R.-S. Lin, M.-H. Yang, and M.-H. Yang, "Incremental learning for robust visual tracking," *International journal of computer vision*, vol. 77, no. 1, pp. 125–141, 2008

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 (cont.)

- Sea Φ la transformación no lineal al espacio de alta dimensión. Asumimos los datos centrado en el espacio de llegada: $\sum_{j=1}^{m} \Phi(\mathbf{x}_j) = 0$.
- En ese espacio, la matriz de covarianza es

$$C_\Phi = \frac{1}{m} \sum_{j=1}^{m} \Phi(\mathbf{x}_j)\Phi(\mathbf{x}_j)^T \quad \text{(aquí los } \Phi(\mathbf{x}_j) \text{ son vectores columnas).}$$

- Buscamos el vector propio $\mathbf{v}$ con valor propio λ:

$$\lambda \mathbf{v} = C_\Phi \mathbf{v} = \frac{1}{m} \sum_{j=1}^{m} \Phi(\mathbf{x}_j)\Phi(\mathbf{x}_j)^T \mathbf{v} = \frac{1}{m} \sum_{j=1}^{m} (\Phi(\mathbf{x}_j)^T \mathbf{v})\Phi(\mathbf{x}_j),$$

por lo que los vectores propios $\mathbf{v}$ pertenecen al subespacio generado por $\{\Phi(\mathbf{x}_1), \ldots, \Phi(\mathbf{x}_m)\}$: existen $\alpha_1, \ldots, \alpha_m$ tales que

$$\mathbf{v} = \sum_{l=1}^{m} \alpha_l \Phi(\mathbf{x}_l).$$

Kernel PCA (cont.)

- Tomando producto interno con $\Phi(\mathbf{x}_i)$, $i = 1, 2, \ldots, m$, en la ecuación de descomposición propia, tenemos el sistema

$$\lambda \Phi(\mathbf{x}_i)^T \mathbf{v} = \Phi(\mathbf{x}_i)^T C_\Phi \mathbf{v},$$
$$\Leftrightarrow \lambda \Phi(\mathbf{x}_i)^T \sum_{l=1}^{m} \alpha_l \Phi(\mathbf{x}_l) = \frac{1}{m} \Phi(\mathbf{x}_i)^T \sum_{j=1}^{m} \sum_{l=1}^{m} \alpha_l \Phi(\mathbf{x}_j) \Phi(\mathbf{x}_j)^T \Phi(\mathbf{x}_l).$$

Introduciendo el kernel $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ tenemos

$$m\lambda \sum_{l=1}^{m} K_{il} \alpha_l = \sum_{j=1}^{m} \sum_{l=1}^{m} K_{ij} K_{jl} \alpha_l, \quad i = 1, 2, \ldots, m.$$

Matricialmente,

$$m\lambda K \boldsymbol{\alpha} = K^2 \boldsymbol{\alpha} \;\Rightarrow\; \boxed{m\lambda \boldsymbol{\alpha} = K \boldsymbol{\alpha}}$$

Kernel PCA (cont.)

- Una vez encontrado $\boldsymbol{\alpha}$ es necesario normalizarlos par que se verifique que $\|\mathbf{v}\| = 1$:

$$1 = \sum_{i,j=1}^{m} \alpha_i \alpha_j \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j) = \sum_{i,j=1}^{m} \alpha_i \alpha_j K_{ij} = \boldsymbol{\alpha}^T K \boldsymbol{\alpha} = \lambda \|\boldsymbol{\alpha}\|^2.$$

  Las proyecciones en los ejes principales $\mathbf{v}_k$ se calculan como

$$\boxed{\mathbf{v}_k^T \Phi(\mathbf{x}) = \sum_{i=1}^{m} \alpha_i^{(k)} \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}) = \sum_{i=1}^{m} \alpha_i^{(k)} k(\mathbf{x}_i, \mathbf{x})}$$

- **Centrado de los datos:** explícita; basta con aplicar una modificación en el kernel

$$\tilde{\Phi}(\mathbf{x}) = \Phi(\mathbf{x}) - \frac{1}{m} \sum_{i=1}^{m} \Phi(\mathbf{x}_i)$$

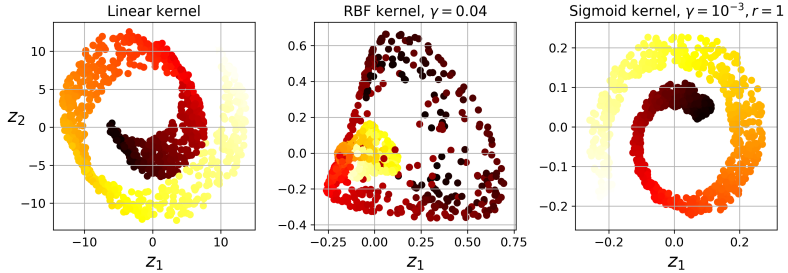
$$\tilde{K}(\mathbf{x}, \mathbf{y}) = K(\mathbf{x}, \mathbf{y}) - \frac{1}{m} \sum_{i=1}^{m} K(\mathbf{x}_i, \mathbf{y}) - \frac{1}{m} \sum_{i=1}^{m} K(\mathbf{x}, \mathbf{x}_i) + \frac{1}{m^2} \sum_{i,j=1}^{m} K(\mathbf{x}_i, \mathbf{x}_j).$$

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 (cont.)

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'}
```

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 $\mathbf{x}_i$, evaluar cómo se puede representar como combinación lineal de sus vecinos k vecinos más cercanos $\mathbf{x}_j$, $j \in \mathcal{N}_i$.
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 (cont.)

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

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

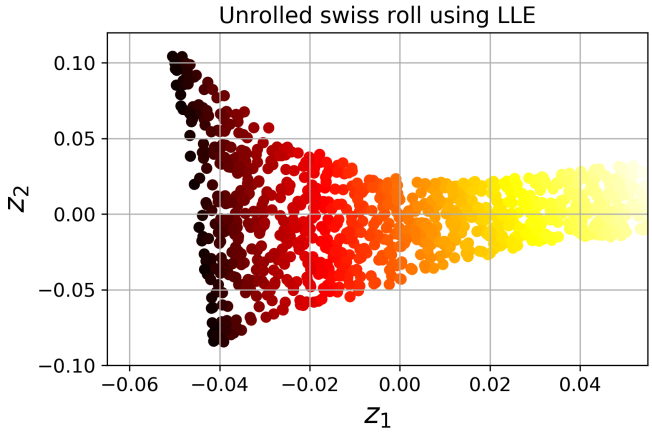
2. Encontrar el conjunto de datos de dim $d < n$ que responde a la misma estructura local:

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

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

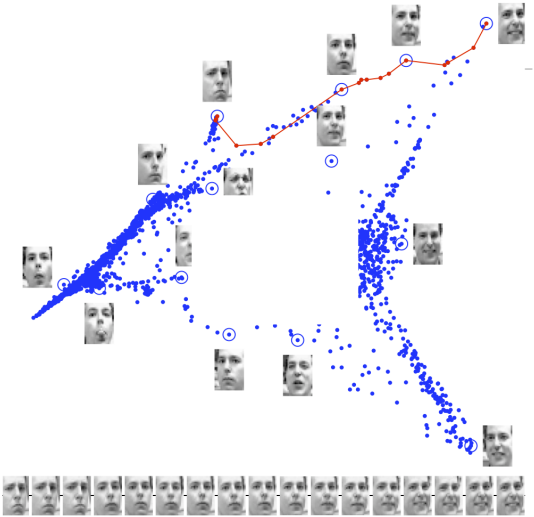
Locally Linear Embedding en Scikit-Learn

```python
from sklearn.manifold import LocallyLinearEmbedding

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



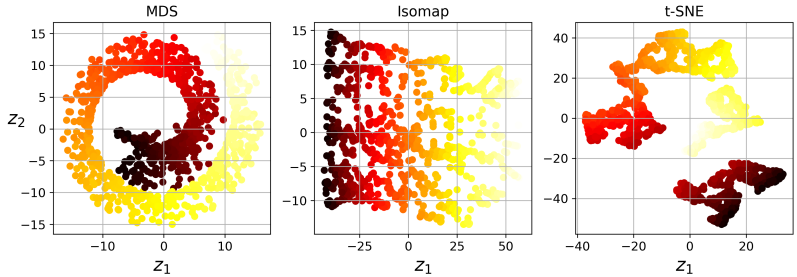
Locally Linear Embedding (cont.)

$m = 1965$ imágenes 20×28 ($n = 560$)
$k = 12$, $d = 2$.



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

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

Además de los métodos de reducción de dimensionalidad, veremos:

- 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

Objetivo: descubrir estructura dentro de un conjunto de datos $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, 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:

- Particionales: producen una única partición, $\mathcal{D}_1, \ldots, \mathcal{D}_c$, que optimiza una función criterio
- 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.

Obs.: Un método de clustering siempre produce clusters, aunque éstos no existan realmente $\Rightarrow$ todo método de clustering debe ser seguido de una etapa de validación de los clusters obtenidos.

Medidas de similitud entre muestras

Para encontrar los agrupamientos naturales, la noción de similitud (no tienen porqué ser distancias) debe ser adaptada al problema particular. Su elección no es trivial.

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

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

- Solo válidas para espacios de características suficientemente isotópicos.
- Correlaciones entre características pueden distorsionar estas distancias; es común normalizar los datos previamente.

Medidas de similitud entre muestras (cont.)

Ejemplo: Distancia de Mahalanobis $d(\mathbf{x}, \mathbf{x}') = \left((\mathbf{x} - \mathbf{x}')^T \Sigma^{-1} (\mathbf{x} - \mathbf{x}')\right)^{1/2}$.

- Asume que los datos siguen una distribución normal con matriz de covarianza Σ (que codifica la correlación entre características).
- Blanquear y usa ℓ^2 equivale a usar distancia de Mahalanobis.
- Sólo vale para datos con distribución normal; catastrófico si se aplica a distribuciones multimodales.

Ejemplo: disimilaridad angular $d(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}' / \|\mathbf{x}\| \|\mathbf{x}'\|$.

- No es una distancia.
- Particularmente valida para datos distribuidos en hiperesferas.

Métodos particionales

Objetivo: dado un conjunto de patrones $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ y una medida de similitud entre patrones, identificar una partición $\mathcal{D}_1, \ldots, \mathcal{D}_c$ que optimice una cierta función criterio.

Aproximadamente $c^n/c!$ particiones posibles (si $n = 100$ y $c = 5$, esto es 10^{67})

$\Rightarrow$ Optimización del criterio por búsqueda exhaustiva fuera de consideración.

$\Rightarrow$ Métodos iterativos, aunque no garanticen convergencia a óptimos globales.

Funciones criterio

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

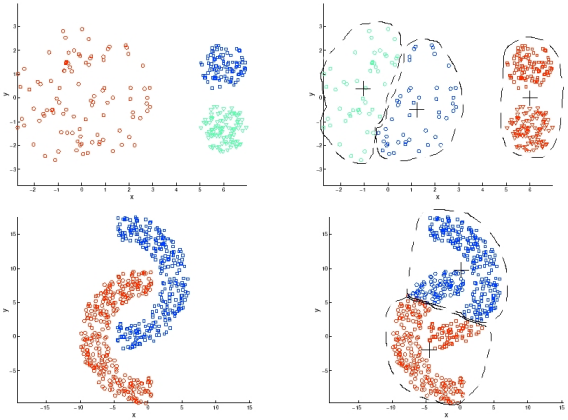
Criterio SSE (sum of squared errors), criterios de mínima varianza

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

Inconvenientes:

- Medida de las varianzas intra-cluster de la partición. Estrictamente, sólo tiene sentido cuando los clusters de los datos son isotrópicos, de distribución normal multivariada.
- Los criterios de mínima varianza favorecen soluciones que dividen clusters grandes.
- Si la cantidad de puntos en los distintos clusters es muy dispar, no permiten alcanzar soluciones que revelan la estructura intrínseca de los datos.

Funciones criterio (cont.)



Resultado de minimizar un criterio de mínima varianza (algoritmo k-means)

Métodos particionales populares

- k-means (Lloyd, 1957. Publicado en 1982)
- DBSCAN (Ester et al., 1996)
- Mean shift (Fukunaga & Hosteler, 1975; Cheng, 1995; Comaniciu, 2002)
- Métodos basados en kernels (e.g. Kernel k-Means: Schölkopf et al., 1996)
- Métodos basados grafos. Spectral clustering (Shi & Malik, 1997; Ng, Jordan & Weiss, 2001)
- ...

Veremos los dos primeros.

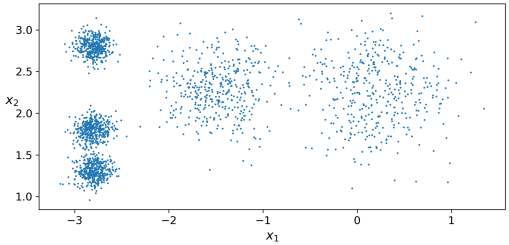
k-means

- Es uno de los métodos usados.
- Consiste en una minimización iterativa del SSE
- Algoritmo:
  1. Elegir una partición inicial en c grupos al azar
  2. Calcular las medias $\boldsymbol{\mu}_i$ de cada cluster
  3. Seleccionar secuencialmente un punto $\mathbf{x} \in \mathcal{D}$, y si corresponde, reasignarlo al cluster que minimiza $\|\mathbf{x} - \boldsymbol{\mu}_i\|_2$
  4. Si no hay más reasignaciones en todo $\mathcal{D}$, 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)
```

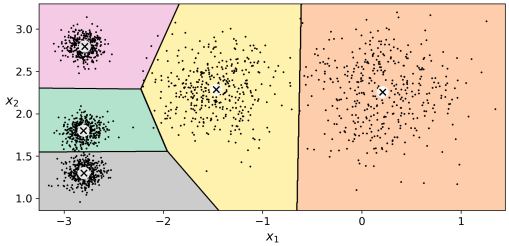
El método devuelve un vector `y_pred` en la variable `labels_` de la instancia, que asigna una etiqueta correspondiente al cluster al cual es asignado cada muestra.

```
>>> 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 (cont.)



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]])
```

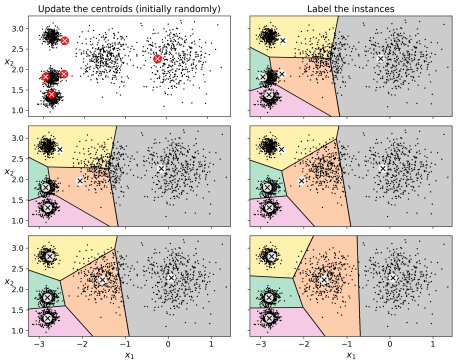
Podemos predecir el cluster al cual pertenecen nuevas muestras (se las asigna al cluster cuyo centroide está más cerca):

```
>>> 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.

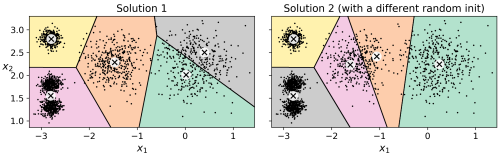
k-means: inicialización

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 (cont.)

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_`: $SSE = \sum_{i=1}^{c} \sum_{\mathbf{x} \in \mathcal{D}_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|_2^2$. 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: versiones optimizadas

k-means acelerado*

- Reduce el costo computacional evitando calcular distancias innecesarias, usando cotas superiores e inferiores obtenidas mediante la desigualdad triangular.
- Método por defecto en scikit-learn (`algorithm='elkan'`).
- Otros métodos basados en cotas sobre la desigualdad triangular: Hamerly (2010), Drake & Hamerly (2012), Hamerly & Drake (2014)

*C. Elkan, "Using the triangle inequality to accelerate k-means," in *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, p. 147–153, AAAI Press, 2003

k-means: versiones optimizadas (cont.)

k-means mediante mini-batches* †

- Suponiendo un conjunto de centroides candidatos $\mathbf{C} = (\mathbf{c}_1, \ldots, \mathbf{c}_k)$, el SSE se escribe

$$SSE(\mathbf{C}) = \frac{1}{2} \sum_{i=1}^{n} \min_{j=1,\ldots,k} \|\mathbf{x}_i - \mathbf{c}_j\|^2.$$

- Se puede obtener una versión de descenso por gradiente: $\mathbf{C}^{t+1} = \mathbf{C}^t + \Delta\mathbf{C}$, con

$$\Delta\mathbf{C} = -\eta_t \frac{\partial SSE}{\partial \mathbf{C}}(\mathbf{C}^t) = \sum_{i=1}^{n} \begin{cases} \eta_k(\mathbf{x}_i - \mathbf{c}_{j(i)}), & j(i) = \operatorname{argmin}_j \|\mathbf{x}_i - \mathbf{c}_j\| \\ 0 & \text{si no.} \end{cases}$$

- k-means por mini-batch procede por SGD aproximando $\Delta\mathbf{C}$ sobre un mini-batch.

---

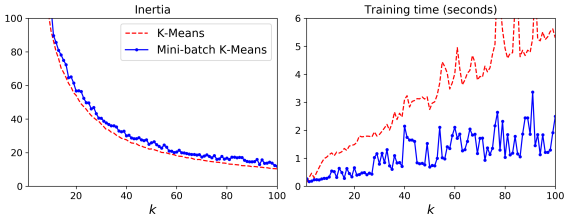
*L. Bottou and Y. Bengio, "Convergence properties of the k-means algorithms," in *Advances in Neural Information Processing Systems* (G. Tesauro, D. Touretzky, and T. Leen, eds.), vol. 7, MIT Press, 1995

†D. Sculley, "Web-scale k-means clustering," in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, (New York, NY, USA), p. 1177–1178, Association for Computing Machinery, 2010

k-means: versiones optimizadas (cont.)

- Mini-batch k-means acelera típicamente un factor de tres y permite ejecutar k-means con grandes volúmenes de datos sin saturar la memoria.
- En general conduce a soluciones de calidad inferior, diferencia relativa que se incrementa al aumentar la cantidad de clusters.

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```



k-medians

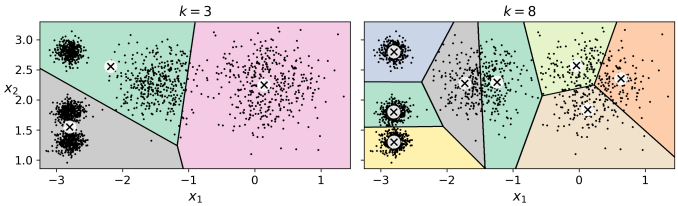
- Variante de k-means, más robusto a outliers (basado en medianas en lugar de medias).
- Minimización iterativa de $\sum_{i=1}^{c} \sum_{\mathbf{x} \in \mathcal{D}_i} \|\mathbf{x} - \mathbf{m}_i\|_1$, donde la coord. k-ésima de $\mathbf{m}_i$ es la mediana en la dirección k de los $\mathbf{x} \in \mathcal{D}_i$.

El algoritmo es igual al k-means, con los cambios siguientes:

- Paso 2: se calculan los $\mathbf{m}_i$ en lugar de los $\boldsymbol{\mu}_i$
- Paso 3: $\mathbf{x}$ se asigna al cluster que minimiza la $\|\mathbf{x} - \mathbf{m}_i\|_1$.

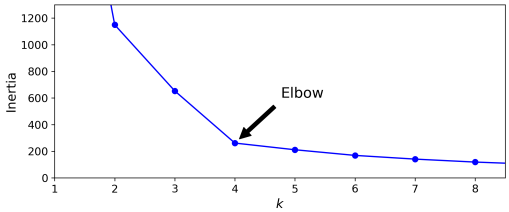
Estimación del número óptimo de clusters

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).



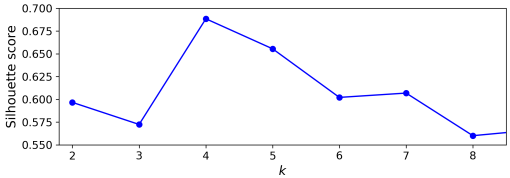
Estimación del número óptimo de clusters (cont.)

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:

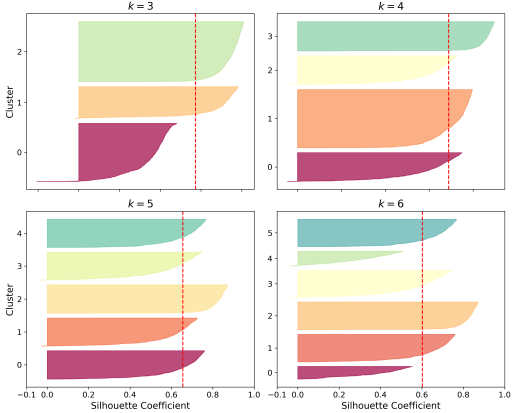
```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```



Estimación del número óptimo de clusters (cont.)

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



Aplicación de k-means como pre-procesamiento para clasificación

Base de dígitos: 1797 imágenes en niveles de gris, 8×8 pixels. Se dividen en 75% de instancias para entrenamiento y 25% para validación.

- Baseline: regresión logística. Accuracy = 96.7%
- Reducción de dimensionalidad con k-means, 50 clusters. Cada imagen ahora se codifica por un vector de dimensión $k = 50$: sus distancias a los 50 centroides. Accuracy = 98.2%
- Elección óptima de k para la clasificación con `GridSearchCV`. Accuracy = 98.4%

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
```

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits)
```

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)
```

```
>>> log_reg.score(X_test, y_test)
0.9666666666666667
```

Aplicación de k-means como pre-procesamiento para clasificación

Base de dígitos: 1797 imágenes en niveles de gris, 8×8 pixels. Se dividen en 75% de instancias para entrenamiento y 25% para validación.

- Baseline: regresión logística. Accuracy = 96.7%
- Reducción de dimensionalidad con k-means, 50 clusters. Cada imagen ahora se codifica por un vector de dimensión $k = 50$: sus distancias a los 50 centroides. Accuracy = 98.2%
- Elección óptima de k para la clasificación con `GridSearchCV`. Accuracy = 98.4%

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression()),
])
pipeline.fit(X_train, y_train)


>>> pipeline.score(X_test, y_test)
0.9822222222222222
```

Aplicación de k-means como pre-procesamiento para clasificación

Base de dígitos: 1797 imágenes en niveles de gris, 8×8 pixels. Se dividen en 75% de instancias para entrenamiento y 25% para validación.

- Baseline: regresión logística. Accuracy = 96.7%
- Reducción de dimensionalidad con k-means, 50 clusters. Cada imagen ahora se codifica por un vector de dimensión $k = 50$: sus distancias a los 50 centroides. Accuracy = 98.2%
- Elección óptima de k para la clasificación con `GridSearchCV`. Accuracy = 98.4%

```
from sklearn.model_selection import GridSearchCV

param_grid = dict(kmeans__n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)
```

```
>>> grid_clf.best_params_
{'kmeans__n_clusters': 90}
>>> grid_clf.score(X_test, y_test)
0.9844444444444445
```

Aplicación de k-means para aprendizaje semi-supervisado

Base de dígitos: 1797 imágenes en niveles de gris, 8×8 pixels. Se dividen en 75% de instancias para entrenamiento y 25% para validación.

- Regresión logística entrenada sobre una muestra de 50 instancias al azar del conjunto de entrenamiento (10 por dígito). Accuracy = 83%
- k-means entrenado con $k = 50$ clusters
- Selección de 50 representativos por cluster: se toma la imagen más cercana a cada centroide. Luego se los etiqueta manualmente

```
n_labeled = 50
log_reg = LogisticRegression()
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

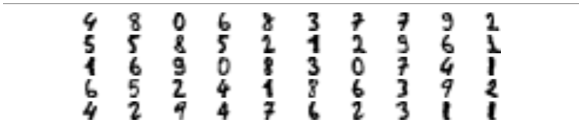
```
>>> log_reg.score(X_test, y_test)
0.8266666666666667
```

Aplicación de k-means para aprendizaje semi-supervisado

Base de dígitos: 1797 imágenes en niveles de gris, 8×8 pixels. Se dividen en 75% de instancias para entrenamiento y 25% para validación.

- Regresión logística entrenada sobre una muestra de 50 instancias al azar del conjunto de entrenamiento (10 por dígito). Accuracy = 83%
- k-means entrenado con $k = 50$ clusters
- Selección de 50 representativos por cluster: se toma la imagen más cercana a cada centroide. Luego se los etiqueta manualmente

```
k = 50
kmeans = KMeans(n_clusters=k)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```



Fifty representative digit images (one per cluster)

```
y_representative_digits = np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])
```

Aplicación de k-means para aprendizaje semi-supervisado

- Regresión logística con los 50 representativos, en lugar de los 50 elegidos al azar. Accuracy = 92.4%
- Propagación de etiquetas: a cada imagen de un cluster se le asigna la etiqueta de su centroide. Accuracy = 92.9%
- Propagación de etiquetas al 20% de los puntos más cercanos a los centroides. Accuracy = 94.2%. Con apenas 50 representativos, no muy lejos del 96.9% obtenido con regresión logística sobre todo el conjunto de entrenamiento.

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.9244444444444444
```

Aplicación de k-means para aprendizaje semi-supervisado

- Regresión logística con los 50 representativos, en lugar de los 50 elegidos al azar. Accuracy = 92.4%
- Propagación de etiquetas: a cada imagen de un cluster se le asigna la etiqueta de su centroide. Accuracy = 92.9%
- Propagación de etiquetas al 20% de los puntos más cercanos a los centroides. Accuracy = 94.2%. Con apenas 50 representativos, no muy lejos del 96.9% obtenido con regresión logística sobre todo el conjunto de entrenamiento.

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]
```

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
>>> log_reg.score(X_test, y_test)
0.9288888888888889
```

Aplicación de k-means para aprendizaje semi-supervisado

- Regresión logística con los 50 representativos, en lugar de los 50 elegidos al azar. Accuracy = 92.4%
- Propagación de etiquetas: a cada imagen de un cluster se le asigna la etiqueta de su centroide. Accuracy = 92.9%
- Propagación de etiquetas al 20% de los puntos más cercanos a los centroides. Accuracy = 94.2%. Con apenas 50 representativos, no muy lejos del 96.9% obtenido con regresión logística sobre todo el conjunto de entrenamiento.

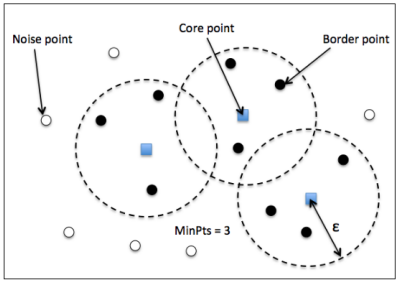
```
percentile_closest = 20

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.9422222222222222
```

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

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

- Para cada punto, 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.



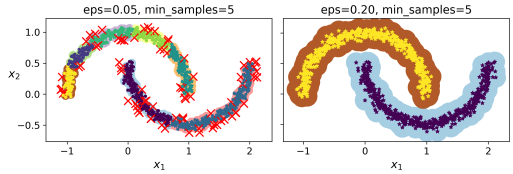
*M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, p. 226–231, AAAI Press, 1996

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]])
```



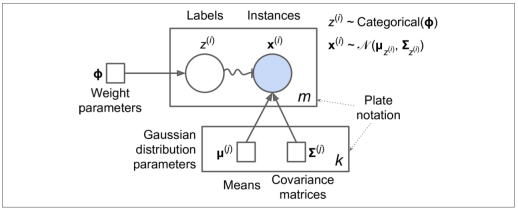
DBSCAN permite encontrar clusters de formas arbitrarias. Funciona correctamente si los clusters son suficientemente densos y está separados por regiones poco densas.

Mezcla de gaussianas (GMM)

Modelo probabilístico que asume que las muestras $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$ 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 $\mathbf{x}_i$ 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 $\mathbf{x}_i$ es asignada al cluster j-ésimo.
2. Si $\{z_i = j\}$, $\mathbf{x}_i$ se obtiene como realización del proceso gaussiano de media $\boldsymbol{\mu}_j$ y matriz de covarianza Σ_j, i.e. $\mathbf{x}_i \sim \mathcal{N}(\boldsymbol{\mu}_j, \Sigma_j)$.



Mezcla de gaussianas en Scikit-Learn

Dada la matriz de datos $\mathbf{X} \in \mathbb{R}^{m \times n}$, 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]]])
```

Mezcla de gaussianas en Scikit-Learn (cont.)

- *Hard clustering:* el método `predict()` permite, dada una muestra, predecir el cluster o la componente a la cual pertenece (aquella para la cual la densidad a posteriori es mayor).
- *Soft clustering:* el método `predict_proba()` devuelve, dada una muestra, la probabilidad a posteriori de pertenencia a cada componente de la mezcla.

```
>>> gm.predict(X)
array([2, 2, 1, ..., 0, 0, 0])
>>> gm.predict_proba(X)
array([[2.32389467e-02, 6.77397850e-07, 9.76760376e-01],
       [1.64685609e-02, 6.75361303e-04, 9.82856078e-01],
       [2.01535333e-06, 9.99923053e-01, 7.49319577e-05],
       ...,
       [9.99999571e-01, 2.13946075e-26, 4.28788333e-07],
       [1.00000000e+00, 1.46454409e-41, 5.12459171e-16],
       [1.00000000e+00, 8.02006365e-41, 2.27626238e-15]])
```

Mezcla de gaussianas en Scikit-Learn (cont.)

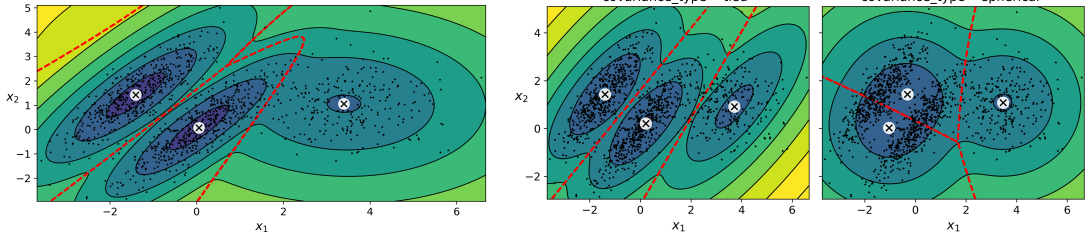
- Como GMM es un modelo generativo, una vez estimados los parámetros de la mezcla, es posible generar nuevas muestras según su distribución:

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

Mezcla de gaussianas en Scikit-Learn (cont.)

El hiperparámetro `covariance_type` controla distintas restricciones sobre las matrices de covarianza de las componentes:

- `"full"`: sin restricciones (opción por defecto)
- `"spherical"`: gaussianas isotrópicas
- `"diagonal"`: ejes principales alineados con los ejes
- `"tied"`: todas las componentes tienen igual matriz de covarianza.



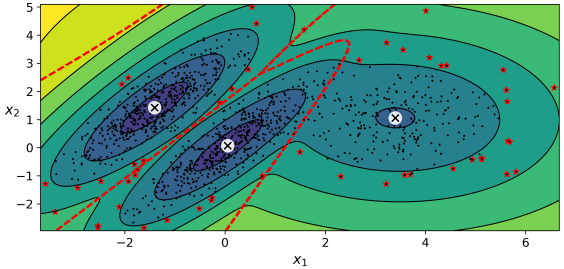
Mezcla de gaussianas: detección de anomalías

- 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]
```



Mezcla de gaussianas: cantidad de clusters desconocido

Dos criterios muy usados, derivados de teoría de la información:

- Criterio de Información de Akaike (AIC):
  $(\widehat{\mathbf{\Theta}}_{(k)}, \widehat{k}) = \arg\min_{\mathbf{\Theta},k} \{-\log p(\mathbf{x}_1, \ldots, \mathbf{x}_m; \mathbf{\Theta}, k) + m_k\}$
- Criterio de Información Bayesiano (BIC):
  $(\widehat{\mathbf{\Theta}}_{(k)}, \widehat{k}) = \arg\min_{\mathbf{\Theta},k} \left\{-\log p(\mathbf{x}_1, \ldots, \mathbf{x}_m; \mathbf{\Theta}, k) + \frac{m_k}{2} \log m\right\}$,

dónde m_k: cantidad de parámetros libres del problema. En mezcla de Gaussianas:

$$m_k = \underbrace{(k-1)}_{\text{priors}} + \underbrace{k \cdot n}_{\text{medias}} + \underbrace{k \cdot \frac{n(n+1)}{2}}_{\text{covarianzas}}.$$

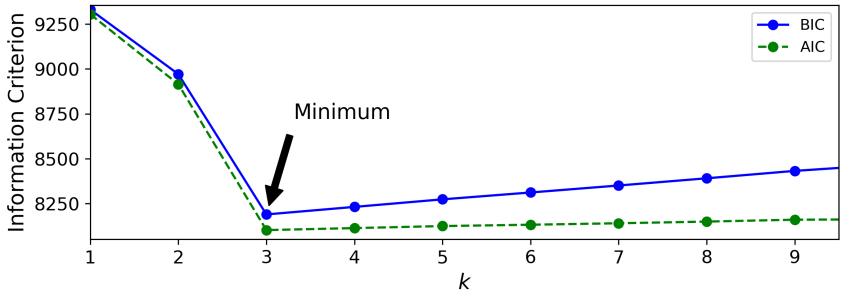
Ambos se tratan de un compromiso entre ajuste a datos y complejidad del modelo.

Mezcla de gaussianas: cantidad de clusters desconocido (cont.)

Se procede explorando distintos k, encontrando los $\widehat{\boldsymbol{\Theta}}_{(k)}$ correspondiente, y conservando la solución que minimiza el criterio.

En scikit-learn se evalúan los criterios como:

```
>>> gm.bic(X)
8189.74345832983
>>> gm.aic(X)
8102.518178214792
```



Referencias I

L. Sirovich and M. Kirby, "Low-dimensional procedure for the characterization of human faces," *J. Opt. Soc. Am. A*, vol. 4, pp. 519–524, Mar 1987.

M. Turk and A. Pentland, "Eigenfaces for recognition," *J. Cognitive Neuroscience*, vol. 3, p. 71–86, Jan. 1991.

N. Halko, P. G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM Review*, vol. 53, no. 2, pp. 217–288, 2011.

D. A. Ross, D. A. Ross, J. Lim, J. Lim, R.-S. Lin, R.-S. Lin, M.-H. Yang, and M.-H. Yang, "Incremental learning for robust visual tracking," *International journal of computer vision*, vol. 77, no. 1, pp. 125–141, 2008.

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.

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

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.

C. Elkan, "Using the triangle inequality to accelerate k-means," in *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, p. 147–153, AAAI Press, 2003.

L. Bottou and Y. Bengio, "Convergence properties of the k-means algorithms," in *Advances in Neural Information Processing Systems* (G. Tesauro, D. Touretzky, and T. Leen, eds.), vol. 7, MIT Press, 1995.

Referencias II

D. Sculley, "Web-scale k-means clustering," in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, (New York, NY, USA), p. 1177–1178, Association for Computing Machinery, 2010.

M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, p. 226–231, AAAI Press, 1996.

A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3rd Edition*.
O'Reilly Media, Inc., 2022.

T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*.
Springer Series in Statistics, New York, NY, USA: Springer New York Inc., 2001.

C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*.
Berlin, Heidelberg: Springer-Verlag, 2006.

W. S. Torgerson, "Multidimensional scaling: I. theory and method," *Psychometrika*, vol. 17, pp. 401–419, 1952.

J. B. Tenenbaum, V. de Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, no. 5500, p. 2319, 2000.

T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, pp. 59–69, Jan. 1982.

D. D. Lee and H. S. Seung, "Learning the parts of objects by nonnegative matrix factorization," *Nature*, vol. 401, pp. 788–791, 1999.

P. Paatero and U. Tapper, "Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values," *Environmetrics*, vol. 5, no. 2, pp. 111–126, 1994.