Clase8 IMA539

Alejandro Ferreira Vergara

April 14, 2023

1 Extracción de características para reducir la Dimensionalidad

1.1 ¿Qué aprenderemos hoy?

- **Análisis de Componentes Principales (PCA)**
- **Análisis de Discriminante Lineal (LDA)**
- **Reducción de dimensión para visualización**
- **Incrustación de vecinos estocásticos t-distribuidos (t-SNE)**

2 Análisis de Componentes Principales

- **Algoritmo de Aprendizaje No Supervisado.**
- **Transforma o proyecta (linealmente) los datos en un nuevo espacio de características.**

En el contexto de la reducción de la dimensionalidad, la extracción de características puede entenderse como un enfoque de compresión de datos con el objetivo de mantener la mayor parte de la información relevante.

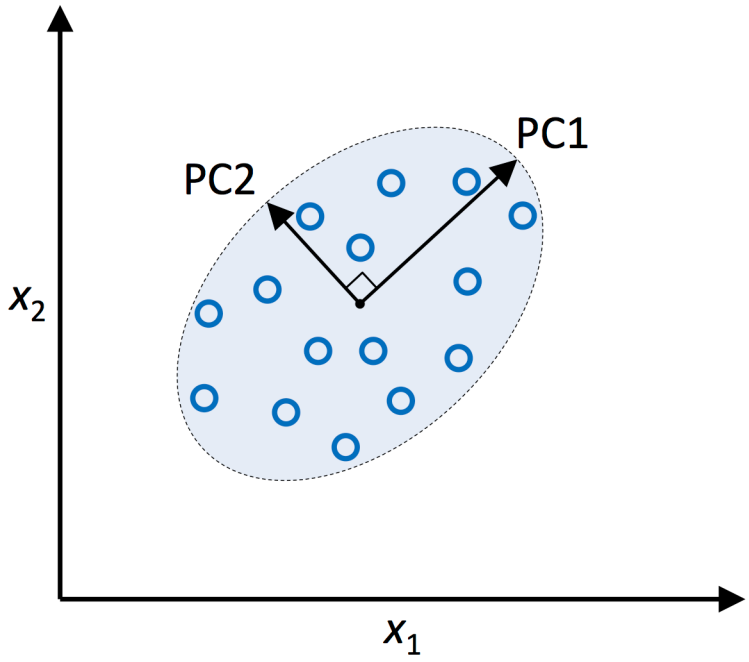
En la práctica, la extracción de características no sólo se utiliza para mejorar el espacio de almacenamiento (en memoria) o la eficiencia computacional del algoritmo de aprendizaje, sino que también puede mejorar el rendimiento predictivo, al reducir **la maldición de la dimensionalidad**, especialmente si trabajamos con modelos no regularizados.

PCA nos ayuda a identificar patrones basados en la **correlación entre las características**. De esta manera, **PCA tiene como objetivo encontrar las direcciones de máxima varianza en los datos y los proyecta en un nuevo subespacio con dimensión igual o menor que el original**. Los ejes ortogonales (componentes principales) del nuevo subespacio pueden interpretarse como las direcciones de máxima varianza dada la restricción de que los nuevos ejes de características son ortogonales entre sí.

[1]:
```
from IPython.display import Image

Image(filename=r'clase8/8_1.png', width=400)
```

[1]:



Al utilizar PCA para reducir la dimensión de los datos, construimos una matriz de transformación W de dimensión $d \times k$, que nos permite mapear un vector de una muestra x en un nuevo subespacio de características k-dimensional, con $k \leq d$.

$$x = [x_1, x_2, \cdots, x_d], \quad x \in \mathbb{R}^d$$

$$z = xW, \quad W \in \mathbb{R}^{d \times k}$$

$$z = [z_1, z_2, \cdots, z_k], \quad z \in \mathbb{R}^k$$

Algoritmo PCA:

1. Normalizar el conjunto de datos X, $N \times d$-dimensional.
2. Construir la matriz de covarianza.
3. Descomponer la matriz de covarianza en sus vectores y valores propios.
4. Ordenar los valores propios por orden decreciente para clasificar los correspondientes vectores propios.
5. Seleccionar k vectores propios, donde estos corresponden a los vectores asociados a los k mayores valores propios, siendo k la dimensión del nuevo subespacio de características ($k \leq d$).
6. Construir una matriz de proyección W a partir de los k primeros vectores propios.

7. Transformar el conjunto de datos X de entrada utilizando la matriz de proyección W, ie, calcular $z = xW$ para todo $x \in X$.

[ ]:
```
import pandas as pd

df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/
 ↪machine-learning-databases/wine/wine.data',header=None)

df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
                   'Color intensity', 'Hue',
                   'OD280/OD315 of diluted wines', 'Proline']

df_wine.head(10)
```

[ ]:
```
df_wine['Class label'].value_counts()
```

[ ]:
```
from sklearn.model_selection import train_test_split

X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪stratify=y, random_state=0)
```

Paso 1: Estandarizar

[ ]:
```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

Paso 2: Matriz de covarianza

La Matriz de covarianza es de dimensión $d \times d$. Almacena las covarianzas por pares entre las diferentes características. Por ejemplo, la covarianza entre dos características x_j y x_k puede calcularse mediante la siguiente ecuación:

$$\sigma_{jk} = \frac{1}{N} \sum_{i=1}^{N} \left(x_j^{(i)} - \mu_j \right) \left(x_k^{(i)} - \mu_k \right)$$

μ_j y μ_k son las medias muestrales de las características j y k, respectivamente.

Una covarianza positiva entre dos características indica que las características aumentan o disminuyen juntas, mientras que una covarianza negativa indica que las características varían en direcciones opuestas.

Matriz de covarianza de tres características:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

Los vectores propios de la matriz de covarianza representan los componentes principales (las direcciones de máxima varianza), mientras que los valores propios correspondientes representan su magnitud.

Recordatorio:

Si v es un vector propio, entonces $\Sigma v = \lambda v$, λ es un escalar; el valor propio.

Paso 3: Descomponer matriz de covarianza

[ ]:
```
import numpy as np

cov_mat = np.cov(X_train_std.T)
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)

print('\nValores Propios: \n%s' % eigen_vals)
```

Varianza total y explicada:

Como queremos reducir la dimensionalidad de nuestro conjunto de datos, comprimiéndolo en un nuevo subespacio de características, sólo seleccionaremos el subconjunto de los vectores propios (componentes principales) que contiene la mayor parte de la información (varianza). Los valores propios definen la magnitud de los vectores propios, por lo que tenemos que ordenar los valores propios por magnitud decreciente.

La proporción de varianza explicada de un valor propio λ_j es simplemente: $\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$.

Paso 4: Ordenar valores propios

[ ]:
```
tot = sum(eigen_vals)
var_exp = [(i / tot) for i in sorted(eigen_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
```

[ ]:
```
import matplotlib.pyplot as plt

plt.bar(range(1, 14), var_exp, alpha=0.5, align='center',
        label='varianza explicada (individual)')
plt.step(range(1, 14), cum_var_exp, where='mid',
         label='varianza explicada acumulada')
plt.ylabel('Varianza Explicada (Ratio)')
plt.xlabel('Indice de las Componentes Principales')
plt.legend(loc='best')
plt.tight_layout()
# plt.savefig('images/05_02.png', dpi=300)
plt.show()
```

2.1 Transformación de las características

Paso 5: Seleccionar k vectores propios, que corresponden a los k mayores valores propios.

Paso 6: Construir una matriz de proyección W a partir de los k vectores propios seleccionados.

Paso 7: Transformar el conjunto de datos X de entrada (dimesión $N \times d$) utilizando la matriz de proyección W y así obtener el nuevo subespacio de características k-dimensional y un conjunto de datos X_0 de dimensión $N \times k$, con $k \leq d$.

[ ]:
```
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i]) for i in␣
 ↪range(len(eigen_vals))]
eigen_pairs.sort(key=lambda k: k[0], reverse=True)

w = np.hstack((eigen_pairs[0][1][:, np.newaxis], eigen_pairs[1][1][:, np.
 ↪newaxis]))

print('Matriz W:\n', w)
print('Dim W:\n', w.shape)
```

[ ]:
```
X_train_std[0].dot(w)
```

[ ]:
```
X_train_std[15].dot(w)
```

Ahora, $X_0 = XW$

[ ]:
```
X_train_pca = X_train_std.dot(w)
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']

for l, c, m in zip(np.unique(y_train), colors, markers):
    plt.scatter(X_train_pca[y_train == l, 0],X_train_pca[y_train == l, 1],c=c,␣
 ↪label=l, marker=m)

plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower right')
plt.tight_layout()
# plt.savefig('images/05_03.png', dpi=300)
plt.show()
```

2.2 PCA con Scikit-Learn

[ ]:
```
from sklearn.decomposition import PCA

pca = PCA()
X_train_pca = pca.fit_transform(X_train_std)
pca.explained_variance_ratio_
```

[ ]:
```
plt.bar(range(1, 14), pca.explained_variance_ratio_, alpha=0.5, align='center')
plt.step(range(1, 14), np.cumsum(pca.explained_variance_ratio_), where='mid')
plt.ylabel('Varianza Explicada (Ratio)')
plt.xlabel('Indice de las Componentes Principales')
plt.show()
```

[ ]:
```
X_train_pca.shape
```

¿Cuánto contribuyen las características a las Componenetes Principales?

[ ]:
```
pca.components_.T * np.sqrt(pca.explained_variance_)
```

[ ]:
```
factor_loadings = pca.components_.T * np.sqrt(pca.explained_variance_)

fig, ax = plt.subplots()

ax.bar(range(13), factor_loadings[:, 0], align='center')
ax.set_ylabel('Factor de cargas para PC 1')
ax.set_xticks(range(13))
ax.set_xticklabels(df_wine.columns[1:], rotation=90)

plt.ylim([-1, 1])
plt.tight_layout()
#plt.savefig('figures/05_05_03.png', dpi=300)
plt.show()
```

[ ]:
```
fig, ax = plt.subplots()

ax.bar(range(13), factor_loadings[:, 1], align='center')
ax.set_ylabel('Factor de cargas para PC 2')
ax.set_xticks(range(13))
ax.set_xticklabels(df_wine.columns[1:], rotation=90)

plt.ylim([-1, 1])
plt.tight_layout()
#plt.savefig('figures/05_05_03.png', dpi=300)
plt.show()
```

3 Análisis de Discriminante Lineal

- **Algoritmo de Aprendizaje Supervisado.**
- **Proyecta (linealmente) los datos en un nuevo espacio de características.**

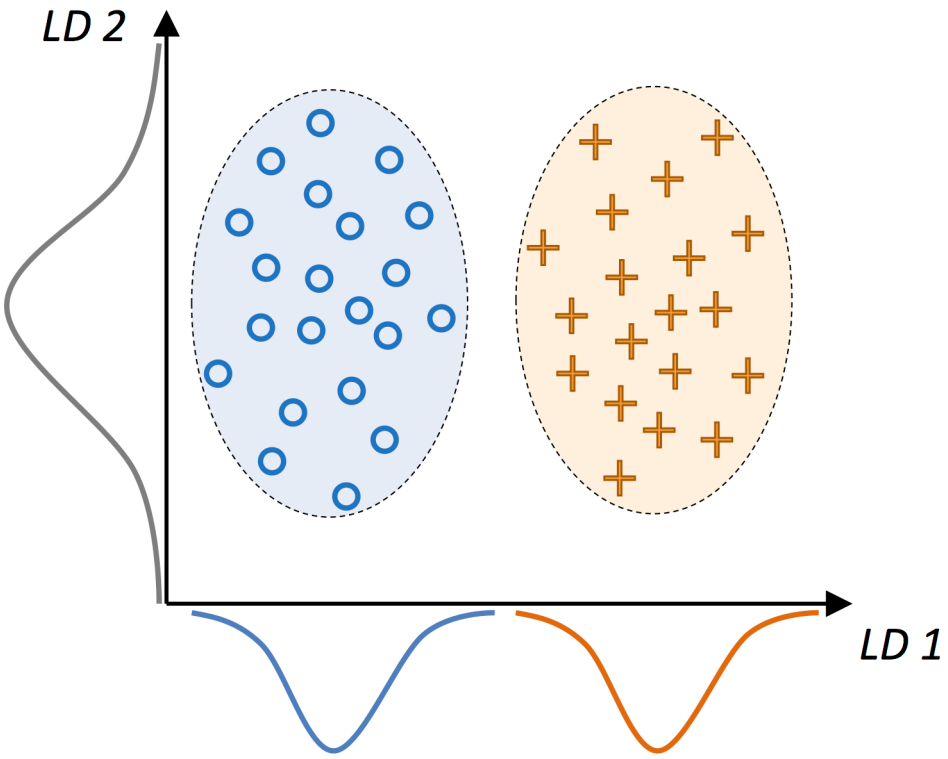
LDA supone que los datos se distribuyen normalmente. Además, supone que las clases tienen matrices de covarianza idénticas y que las características son estadísticamente independientes entre sí.

Mientras que PCA intenta encontrar los ejes de componentes ortogonales de máxima varianza en

un conjunto de datos, **el objetivo de LDA es encontrar el subespacio de características que optimiza la separabilidad de las clases**.

[2]:
```
Image(filename=r'clase8/8_2.png', width=400)
```

[2]:



Algoritmo LDA:

1. Normalizar el conjunto de datos $N \times d$-dimensional.
2. Para cada clase, calcular el vector medio d-dimensional (centroide).
3. Construir la matriz de dispersión entre clases, S_B, y la matriz de dispersión dentro de la clase, S_W.
4. Calcular los vectores propios y los correspondientes valores propios de la matriz $S_W^{-1} S_B$.
5. Ordenar los valores propios por orden decreciente para clasificar los correspondientes vectores propios.
6. Elegir los k vectores propios que corresponden a los k mayores valores propios, para construir una matriz de transformación $d \times k$-dimensional, W (los vectores propios son las columnas de esta matriz).

7. Proyectar las muestras en el nuevo subespacio de características utilizando la matriz de transformación W.

3.1 LDA con Scikit-Learn

[ ]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

lda = LDA(n_components=2)
X_train_lda = lda.fit_transform(X_train_std, y_train)
```

[ ]:
```
print(X_train_lda[:10,:])
print(X_train_lda.shape)
```

Actividad 1: Plotear el conjunto x_train en el nuevo espacio de características 2D que genera LDA

[ ]:
```

```

[ ]:
```
plt.bar(range(1,3), lda.explained_variance_ratio_, alpha=0.5,
        align='center',label='Discriminación Individual',tick_label=[1,2])
plt.step(range(1,3), np.cumsum(lda.explained_variance_ratio_), where='mid',
         label='Discriminación Acumulada')
plt.ylabel('Tasa de Discriminación')
plt.xlabel('Discriminantes Lineales')
plt.ylim([-0.1, 1.1])
plt.legend(loc='best')
plt.show()
```

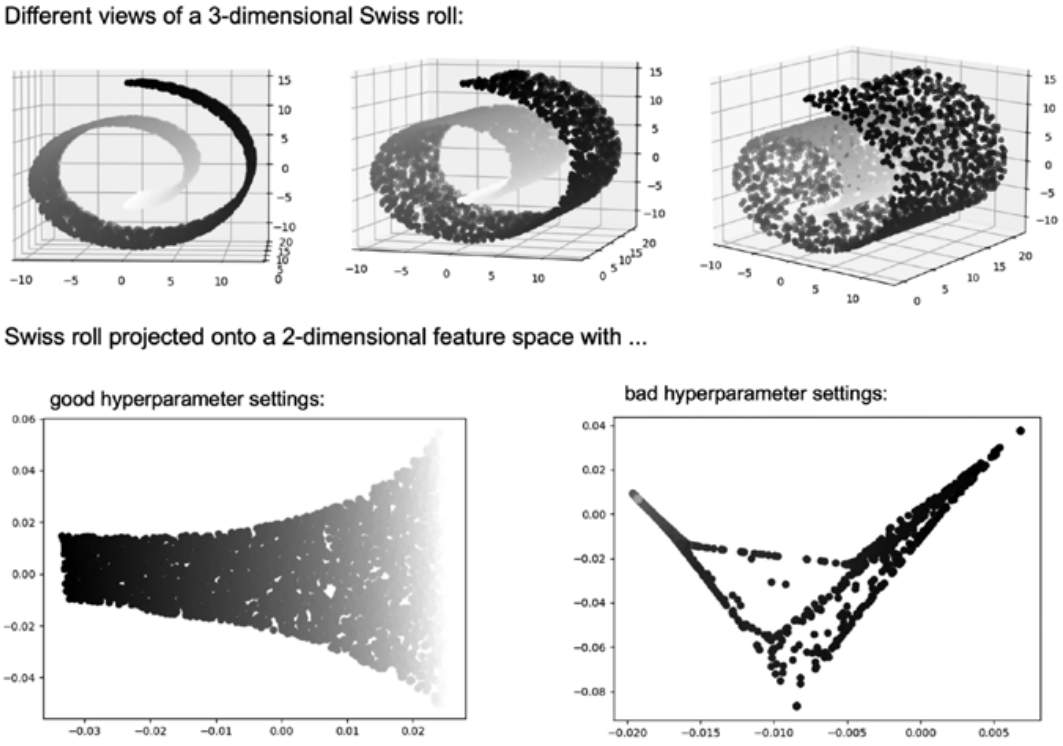
4 Visualización y reducción de dimensionalidad no lineal

La reducción de dimensionalidad no lineal , también conocida como **Manifold Learning** , se refiere a varias técnicas relacionadas que tienen como objetivo proyectar datos de alta dimensión en variedades latentes de menor dimensión, con el objetivo de visualizar los datos en el espacio de baja dimensión o aprender el mapeo (ya sea desde el espacio de alta dimensión a la incrustación de baja dimensión o viceversa) en sí mismo.

Manifold Learning puede considerarse como un intento de generalizar los algoritmos lineales como PCA para que sean sensibles a la estructura no lineal de los datos. Aunque existen variantes supervisadas, los algoritmos típicos de este enfoque son de naturaleza No Supervisada, es decir, aprenden la estructura de los datos de alta dimensión a partir de los datos mismos, sin el uso de clasificaciones predeterminadas.

[3]:
```
Image(filename=r'clase8/8_3.png', width=600)
```

[3]:



4.1 Incrustación de vecinos estocásticos t-distribuidos (t-SNE)

La incrustación de vecinos estocásticos t-distribuidos (t-SNE) es un método estadístico para visualizar datos de alta dimensión al otorgar a cada punto de datos una ubicación en un mapa de dos o tres dimensiones.

En pocas palabras, t-SNE modela puntos de datos en función de sus distancias por pares en el espacio de características de alta dimensión (original). Luego, encuentra una distribución de probabilidad de distancias por pares en el nuevo espacio de menor dimensión que que es cercana la distribución de probabilidad de distancias por pares en el espacio original. En otras palabras, t-SNE aprende a incrustar puntos de datos en un espacio de menor dimensión de modo que se conserven las distancias por pares del espacio original.

t-SNE convierte afinidades de puntos en probabilidades. Las afinidades en el espacio original están representadas por probabilidades conjuntas gaussianas y las afinidades en el espacio incrustado están representadas por distribuciones t de Student. Esto permite que t-SNE sea particularmente sensible a la estructura local.

El algoritmo t-SNE comprende dos etapas principales:

1. t-SNE construye una distribución de probabilidad sobre pares de objetos de alta dimensión de tal manera que a los objetos similares se les asigna una probabilidad más alta, mientras que a los puntos diferentes se les asigna una probabilidad más baja.

2. t-SNE define una distribución de probabilidad similar sobre los puntos en el mapa de baja dimensión y minimiza (utilizando el descenso de gradiente) la **divergencia Kullback-Leibler (divergencia KL)** entre las dos distribuciones con respecto a las ubicaciones de los puntos en el mapa.

La divergencia KL no es convexa, por lo que a veces es útil probar diferentes semillas y seleccionar la incrustación con la divergencia KL más baja.

Algunas ventajas sobre las técnicas tradicionales:

- Revela la estructura oculta en algunos datos.
- Revela datos que se encuentran en múltiples o en diferentes grupos.
- Reduce la tendencia a juntar puntos en el centro.

Las desventajas de usar *t*-SNE son:

- *t*-SNE es computacionalmente costoso y puede tomar varias horas en conjuntos de datos de millones de muestras, donde PCA terminaría en segundos o minutos.
- El algoritmo es estocástico y múltiples ejecuciones con diferentes semillas pueden generar diferentes incrustaciones. Sin embargo, es conveniente elegir la incrustación con el menor error.
- La estructura global no se conserva explícitamente. Este problema se mitiga inicializando puntos con PCA (usando *init='pca'*).

[ ]:
```
from sklearn.datasets import load_digits

digits = load_digits()

fig, ax = plt.subplots(1, 10,dpi=200)

for i in range(10):
    ax[i].imshow(digits.images[i], cmap='Greys')

# plt.savefig('figures/05_12.png', dpi=300)
plt.show()
```

[ ]:
```
digits.data.shape
```

[ ]:
```
y_digits = digits.target
X_digits = digits.data
```

4.2 *t*-SNE con Scikit-Learn

[ ]:
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2,init='pca',random_state=123)
X_digits_tsne = tsne.fit_transform(X_digits)
```

[ ]:
```
import matplotlib.patheffects as PathEffects

def plot_projection(x, colors):
    f = plt.figure(figsize=(8, 8))
    ax = plt.subplot(aspect='equal')
    for i in range(10):
        plt.scatter(x[colors == i, 0],x[colors == i, 1])
    for i in range(10):
        xtext, ytext = np.median(x[colors == i, :], axis=0)
        txt = ax.text(xtext, ytext, str(i), fontsize=24)
        txt.set_path_effects([PathEffects.Stroke(linewidth=5,␣
 ↪foreground="w"),PathEffects.Normal()])

plot_projection(X_digits_tsne, y_digits)
# plt.savefig('figures/05_13.png', dpi=300)
plt.show()
```

[ ]:
```
lda = LDA(n_components=2)
X_digits_lda = lda.fit_transform(X_digits, y_digits)

plot_projection(X_digits_lda, y_digits)
plt.show()
```

Actividad 2: Realizar el ejercicio anterior, utilizando PCA.

[ ]:
```

```