Geron

Outline breve (págs. 384 – 393)

- **Concepto y justificación del PCA**
    - Identifica el hiperplano (espacio de bajas dimensiones) que preserva la **máxima varianza** y proyecta los datos sobre él (Fig. 8-2 y 8-7).
    - Preservar varianza ≡ minimizar la distancia cuadrática media entre datos originales y proyectados.
- **Cálculo mediante descomposición SVD (Ejemplo, SVD no entra)**
    - Factorización X = U\Sigma V^{ T}; las filas de VTV^{ T} (es decir X=UΣV.T) son los **componentes principales**.
    - Requiere centrar los datos antes de aplicar SVD.
- **Implementación manual (NumPy)**
    - Centrar: X_c = X - \bar{x}.
    - `np.linalg.svd(X_c)` → obtener V.T
    - Selección de las primeras d columnas W_d.
    - Proyección: X_{d} = X_c W_d.
- **Uso con Scikit-Learn**
    - `PCA(n_components=2).fit_transform(X)` realiza SVD y centrado automático.
    - Atributos: `components_` (V^{ T}), `explained_variance_ratio_` (fracción de varianza por PC).
- **Elección del número de dimensiones**
    - **Codo de varianza**: curva acumulada (Fig. 8-8) muestra dónde la ganancia adicional se aplana.
    - `n_components=0.95` preserva el 95 % de la varianza y el valor exacto se guarda en `n_components_`.
    - Ejemplo MNIST: 784 → 154 PCs al 95 % de varianza; con modelos potentes (Random Forest) bastan incluso 23 PCs.
- **PCA dentro de un pipeline (tubería) supervisada (Ejemplo, no entra)**
    - `make_pipeline(PCA(), RandomForestClassifier())` combinado con `RandomizedSearchCV` para ajustar simultáneamente `pca__n_components` y

`rf__n_estimators`.

- Ilustra que la reducción de dimensionalidad puede tratarse como otro hiperparámetro del modelo final.

Este segmento presenta el flujo completo: intuición geométrica → cálculo con SVD → API de Scikit-Learn → criterios prácticos para decidir cuántos componentes conservar y cómo integrarlo en pipelines y búsquedas de hiperparámetros.

Marsland

Outline breve (págs. 155 – 158)

- **Motivación y contexto**
  - Reducción de dimensionalidad: rotar y trasladar ejes para alinear los datos con las direcciones de mayor varianza, simplificando la representación sin perder estructura esencial.
- **Fundamento matemático del PCA**
  - Definición formal: Y = P^{ T}X donde P es la matriz de rotación que diagonaliza la matriz de covarianza (cov(Y) se vuelve diagonal).
  - Eigen-descomposición: vectores propios ortogonales (p_i) y valores propios (\lambda_i) indican cuánto “estiramiento” hay en cada dirección; dimensiones con \lambda_i pequeños aportan poco y pueden descartarse.
- **Algoritmo paso a paso**
  - Centrar los datos restando la media (matriz BB).
  - Calcular la matriz de covarianza C = B^{ T}B.
  - Obtener valores y vectores propios (np.linalg.eig).
  - Ordenar por \lambda decreciente y seleccionar las L componentes con \lambda_i > \eta.
  - Proyectar: X_{red} = P_L^{ T}B.
- **Código de referencia (NumPy)**
  - Función `pca(data, nRedDim, normalizar)` que implementa los pasos anteriores y reconstruye la matriz original si se desea.
- **Ejemplos ilustrativos**
  - **Fig. 6.6:** rotación/traslación de ejes determinada por PCA.
  - **Fig. 6.7:** datos 2-D (elipse) comprimidos a 1-D sobre el eje principal.

- **Fig. 6.8:** separación clara de las tres clases del conjunto Iris usando las dos primeras componentes.

PCA

El análisis de componentes principales (PCA) es, con diferencia, el algoritmo de reducción de dimensionalidad más popular. Primero identifica el hiperplano más cercano a los datos y, a continuación, proyecta los datos sobre él, como se muestra en la Figura 8-2.

Preservación de la varianza.

Antes de proyectar el conjunto de entrenamiento en un hiperplano de menor dimensión, primero debe elegir el hiperplano correcto. Por ejemplo, un conjunto de datos 2D simple se representa a la izquierda en la Figura 8-7, junto con tres ejes diferentes (es decir, hiperplanos 1D). A la derecha se muestra el resultado de la proyección del conjunto de datos en cada uno de estos ejes. Como puede observar, la proyección en la línea continua preserva la varianza máxima (arriba), mientras que la proyección en la línea punteada preserva muy poca varianza (abajo) y la proyección en la línea discontinua preserva una cantidad intermedia de varianza (centro).

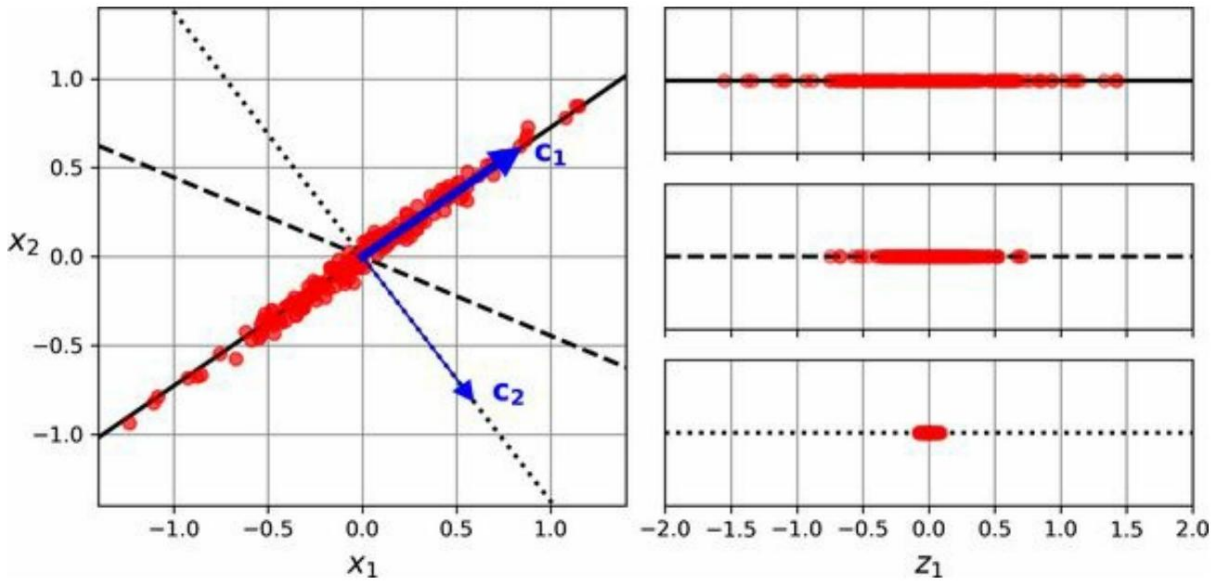

Figura 8-7. Selección del subespacio en el que proyectar

Parece razonable seleccionar el eje que preserva la máxima varianza, ya que probablemente perderá menos información que las demás proyecciones. Otra justificación de esta elección es que es el eje que minimiza la distancia cuadrática media entre el conjunto de datos original y su proyección sobre dicho eje. Esta es la idea, bastante simple, del ACP.[4]

Componentes principales

El análisis de componentes principales (PCA) identifica el eje que representa la mayor cantidad de varianza en el conjunto de entrenamiento. En la Figura 8-7, es la línea continua. También encuentra un segundo eje, ortogonal al primero, que representa la mayor cantidad de varianza restante. En este ejemplo 2D, no hay opción: es la línea punteada. Si se tratara de un conjunto de datos de mayor dimensión, el PCA también encontraría un tercer eje, ortogonal a los dos ejes anteriores, y un cuarto, un quinto, y así sucesivamente: tantos ejes como dimensiones tenga el conjunto de datos.

El eje i^{el} se denomina componente principal (CP) i^{el} de los datos. En la Figura 8-7 , el primer CP es el eje donde se encuentra el vector c_1, y el segundo es el eje donde se encuentra el vector c_2. En la Figura 8-2, los dos primeros CP están en el plano de proyección, y el tercer CP es el eje ortogonal a dicho plano. Después de la proyección, en la Figura 8-3, el primer CP corresponde al eje z_1, y el segundo CP corresponde al eje z_2.

> **NOTA**
>
> Para cada componente principal, el análisis de componentes principales (PCA) encuentra un vector unitario centrado en cero que apunta en la dirección del componente principal (CP). Dado que dos vectores unitarios opuestos se encuentran en el mismo eje, la dirección de los vectores unitarios devueltos por el PCA no es estable: si se perturba ligeramente el conjunto de entrenamiento y se vuelve a ejecutar el PCA, los vectores unitarios podrían apuntar en dirección opuesta a los vectores originales. Sin embargo, generalmente se mantendrán en los mismos ejes. En algunos casos, un par de vectores unitarios puede incluso rotar o intercambiarse (si las varianzas a lo largo de estos dos ejes son muy cercanas), pero el plano que definen generalmente permanecerá invariable.

¿Cómo se pueden encontrar los componentes principales de un conjunto de entrenamiento? Afortunadamente, existe una técnica estándar de factorización matricial llamada descomposición en valores singulares (SVD), que permite descomponer la matriz X del conjunto de entrenamiento en la multiplicación matricial , donde V contiene la unidad de tres matrices U Σ V, vectores que definen todos los componentes principales que se buscan, como se muestra en la Ecuación 8-1.

Ecuación 8-1. Matriz de componentes principales

V = c 1 c 2 cn

El siguiente código Python utiliza la función svd() de NumPy para obtener todos los componentes principales del conjunto de entrenamiento 3D representado en la Figura 8-2, entonces extrae los dos vectores unitarios que definen las dos primeras PC:

```
importar numpy como np

X = [...] # crea un pequeño conjunto de datos 3D
X_centrado = X - X.media(eje=0)
U, s, Vt = np.linalg.svd(X_centrado)
c1 = Vt[0]
c2 = Vt[1]
```

ADVERTENCIA

El PCA asume que el conjunto de datos está centrado alrededor del origen. Como verá, el análisis de componentes principales de Scikit-Learn... Las clases de PCA se encargan de centrar los datos por usted. Si implementa PCA usted mismo (como en el ejemplo anterior), o si utiliza otras bibliotecas, no olvide centrar los datos primero.

Proyección a d dimensiones. Una vez identificados

todos los componentes principales, se puede reducir la dimensionalidad del conjunto de datos a d dimensiones proyectándolo sobre el hiperplano definido por los primeros d componentes principales. Seleccionar este hiperplano garantiza que la proyección conserve la mayor varianza posible. Por ejemplo, en la Figura 8-2, el conjunto de datos 3D se proyecta al plano 2D definido por los dos primeros componentes principales, conservando gran parte de la varianza del conjunto de datos. Como resultado, la proyección 2D se asemeja mucho al conjunto de datos 3D original.

Para proyectar el conjunto de entrenamiento sobre el hiperplano y obtener un conjunto de datos reducido X de dimensionalidad d, calcule la multiplicación de la matriz X del conjunto de entrenamiento d-proyección por la $_{d}$, definida como la matriz que contiene los primeros d matriz W de columnas de V, como se muestra en la Ecuación 8-2.

Ecuación 8-2. Proyección del conjunto de entrenamiento a d dimensiones

$$X_{d\text{-proy}} = XW_d$$

El siguiente código Python proyecta el conjunto de entrenamiento en el plano definido por los dos primeros componentes principales:

```
W2 = Vt[:2].T
X2D = X_centrado @ W2
```

¡Listo! Ahora sabes cómo reducir la dimensionalidad de cualquier conjunto de datos proyectándolo a cualquier número de dimensiones, preservando al máximo la varianza.

Uso de Scikit-Learn. La

clase PCA de Scikit-Learn utiliza SVD para implementar el PCA, tal como hicimos anteriormente en este capítulo. El siguiente código aplica el PCA para reducir la dimensionalidad del conjunto de datos a dos dimensiones (tenga en cuenta que se encarga automáticamente de centrar los datos):

```
de sklearn.decomposition importar PCA

pca = PCA(n_componentes=2)
X2D = pca.fit_transform(X)
```

Después de ajustar el transformador PCA al conjunto de datos, su atributo component_ contiene la transposición de W_d: contiene una fila para cada uno de los primeros d componentes principales.

Razón de varianza explicada

Otro dato útil es la relación de varianza explicada de cada componente principal, disponible a través de la variable explained_variance_ratio_.
La razón indica la proporción de la varianza del conjunto de datos que se encuentra a lo largo de cada componente principal. Por ejemplo, veamos las razones de varianza explicadas de los dos primeros componentes del conjunto de datos 3D representado en la Figura 8-2:

```
>>> pca.explained_variance_ratio_
array([0,7578477, 0,15186921])
```

Este resultado indica que aproximadamente el 76 % de la varianza del conjunto de datos se concentra en el primer PC y aproximadamente el 15 % en el segundo PC. Esto deja aproximadamente el 9 % para el tercer PC, por lo que es razonable suponer que este probablemente contenga poca información.

Cómo elegir la cantidad correcta de dimensiones En

lugar de elegir arbitrariamente la cantidad de dimensiones a las que reducir, es más sencillo elegir la cantidad de dimensiones que sumen una porción suficientemente grande de la varianza, digamos, el 95 % (una excepción a esta regla, por supuesto, es si está reduciendo la dimensionalidad para la visualización de datos, en cuyo caso querrá reducir la dimensionalidad a 2 o 3).

El siguiente código carga y divide el conjunto de datos MNIST (introducido en Capítulo 3) y realiza PCA sin reducir la dimensionalidad, luego calcula la cantidad mínima de dimensiones requeridas para preservar el 95 % de la varianza del conjunto de entrenamiento:

```
desde sklearn.datasets importar fetch_openml

mnist = fetch_openml('mnist_784', como_marco=Falso)
Tren_X, tren_Y = mnist.datos[:60_000], mnist.objetivo[:60_000]
Prueba_x, prueba_y = mnist.datos[60_000:], mnist.objetivo[60_000:]

pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.ratio_de_varianza_explicada) d =
np.argmax(cumsum >= 0.95) + 1 # d es igual a 154
```

Podría entonces establecer n_components=d y ejecutar el PCA de nuevo, pero existe una mejor opción. En lugar de especificar el número de componentes principales que desea conservar, puede establecer n_components como un valor de punto flotante entre 0.0 y 1.0, que indica la proporción de varianza que desea conservar:

```
pca = PCA(n_componentes=0,95)
X_reducido = pca.fit_transform(X_train)
```

El número real de componentes se determina durante el entrenamiento y se almacena en el atributo n_components_:

```
>>> pca.n_componentes_154
```

Otra opción es graficar la varianza explicada en función del número de dimensiones (simplemente graficar la suma acumulada; véase la Figura 8-8). Generalmente, la curva presenta un codo, donde la varianza explicada deja de crecer rápidamente. En este caso, se puede observar que reducir la dimensionalidad a aproximadamente 100 dimensiones no perdería mucha varianza explicada.

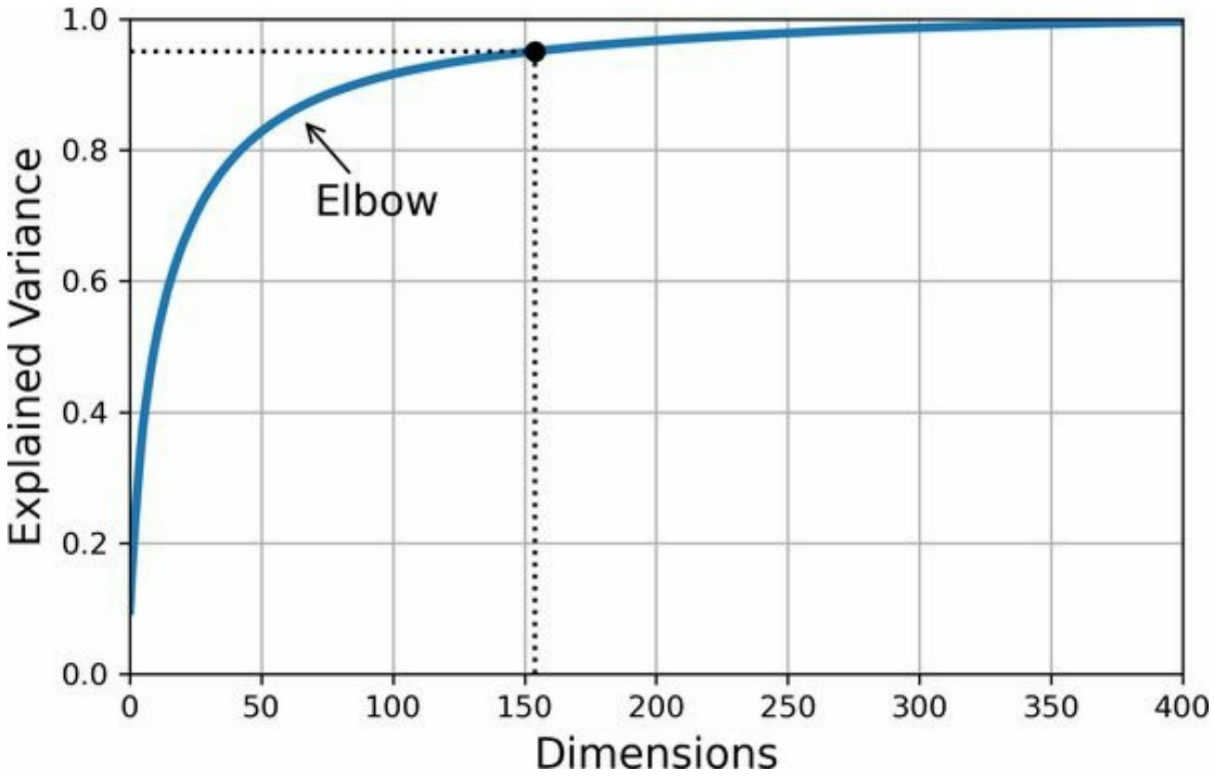

Figura 8-8. Varianza explicada en función del número de dimensiones.

Por último, si utiliza la reducción de dimensionalidad como paso de preprocesamiento para una tarea de aprendizaje supervisado (p. ej., clasificación), puede ajustar el número de dimensiones como lo haría con cualquier otro hiperparámetro (véase el capítulo 2). Por ejemplo, el siguiente ejemplo de código crea una canalización de dos pasos: primero reduce la dimensionalidad mediante PCA y luego clasifica mediante un bosque aleatorio. A continuación, utiliza RandomizedSearchCV para encontrar una buena combinación de hiperparámetros tanto para el PCA como para el clasificador de bosque aleatorio. Este ejemplo realiza una búsqueda rápida, ajustando solo dos hiperparámetros, entrenando con solo 1000 instancias y ejecutando solo 10 iteraciones. Sin embargo, puede realizar una búsqueda más exhaustiva si dispone de tiempo.

```
de sklearn.ensemble importar RandomForestClassifier
```

```
desde sklearn.model_selection importar RandomizedSearchCV desde
sklearn.pipeline importar make_pipeline

clf = make_pipeline(PCA(estado_aleatorio=42),
                    ClasificadorForestalAleatorio(estado_aleatorio=42))
param_distrib =
    { "pca__n_componentes": np.arange(10, 80),
    "clasificadorforestalaleatorio__n_estimadores": np.arange(50, 500)

} rnd_search = BúsquedaAleatoriaCV(clf, parámetro_distribución, n_iter=10, cv=3,
                                   estado_aleatorio=42)
rnd_search.fit(X_train[:1000], y_train[:1000])
```

Veamos los mejores hiperparámetros encontrados:

```
>>> print(rnd_search.best_params_)
{'randomforestclassifier__n_estimators': 465, 'pca__n_components': 23}
```

Es interesante observar lo bajo que es el número óptimo de componentes: ¡redujimos un conjunto de datos de 784 dimensiones a tan solo 23! Esto se debe a que usamos un bosque aleatorio, un modelo bastante potente. Si, en cambio, usáramos un modelo lineal, como un SGDClassifier, la búsqueda detectaría que necesitamos conservar más dimensiones (unas 70).

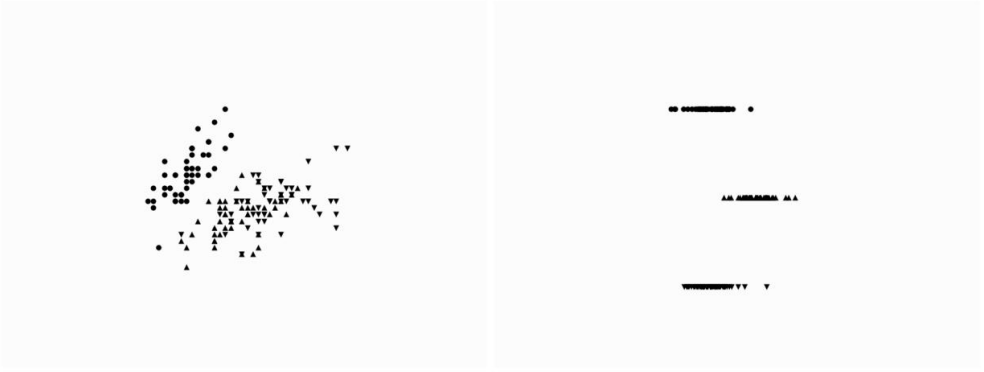

FIGURA 6.5 Gráfico de los datos del iris que muestra las tres clases: izquierda: antes y derecha: después de aplicar LDA.

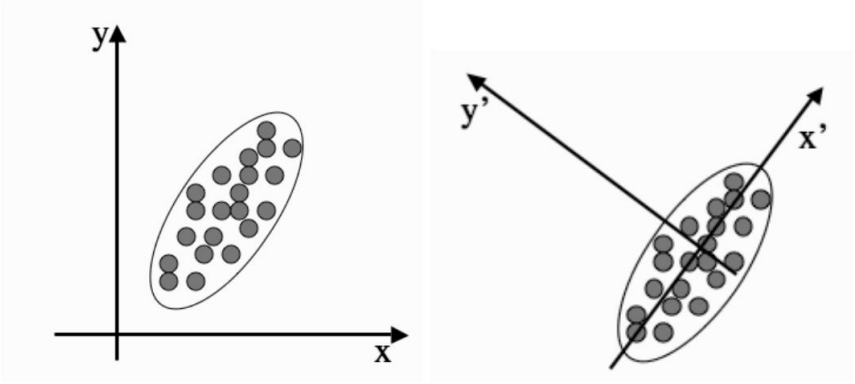

FIGURA 6.6 Dos conjuntos diferentes de ejes de coordenadas. El segundo consiste en una rotación y traslación del primero y se determinó mediante el Análisis de Componentes Principales.

Sin comprometer los resultados de un algoritmo de aprendizaje. De hecho, puede mejorarlos, ya que a menudo eliminamos parte del ruido en los datos.

La pregunta es cómo elegir los ejes. El primer método que veremos es el Análisis de Componentes Principales (ACP). La idea de un componente principal es que representa la dirección de los datos con la mayor variación. El algoritmo primero centra los datos restando la media, luego elige la dirección con la mayor variación y coloca un eje en esa dirección. Después, observa la variación restante y encuentra otro eje ortogonal al primero que cubra la mayor parte posible de la variación restante. A continuación, itera esto hasta agotar los ejes posibles. El resultado final es que toda la variación se produce a lo largo de los ejes del conjunto de coordenadas, por lo que la matriz de covarianza es diagonal: cada nueva variable no está correlacionada con ninguna variable excepto consigo misma. Algunos de los ejes que se encuentran al final tienen muy poca variación, por lo que pueden eliminarse sin afectar la variabilidad de los datos.

Para expresarlo de forma más formal, tenemos una matriz de datos X y queremos rotarla para que los datos se ubiquen en las direcciones de máxima variación. Esto significa que multiplicamos nuestra matriz de datos por una matriz de rotación (a menudo escrita como PT) de modo que Y = PT X, donde P se elige de modo que la matriz de covarianza de Y sea diagonal, es decir,

$$\text{cov}(Y) = \text{cov}(P^T X) = \begin{matrix} \lambda 1 & 0 & 0 & \dots & 0 \\ 0 & \lambda 2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \lambda N \end{matrix} . \quad (6.7)$$

Podemos obtener una interpretación diferente de esto utilizando algo de álgebra lineal y la definición de covarianza para ver que:

$$\text{cov}(Y) = E[YY^T] \quad (6.8)$$
$$= E[(P^T X)(P^T_{\text{INCÓGNITA}})^T] \quad (6.9)$$
$$= E[(P^T X)(X^T P)] \quad (6.10)$$
$$= P^T E(XX^T)P \quad (6.11)$$
$$= P^T \text{cov}(X)P. \quad (6.12)$$

Las dos cosas adicionales que necesitábamos saber eran que $(P^T X)^T = X^T P^{TT} = X^T P$ y que $E[P] = P$ (y obviamente lo mismo para P^T) ya que no es una matriz dependiente de datos. Esto entonces nos dice que:

$$P\text{cov}(Y) = PP^T \text{cov}(X)P = \text{cov}(X)P, \quad (6.13)$$

Donde hay un hecho engañoso, a saber, que para una matriz de rotación $P^T = P^{-1}$. Esto simplemente dice que para invertir una rotación giramos en la dirección opuesta en la misma cantidad que girado hacia adelante.

Como cov(Y) es diagonal, si escribimos P como un conjunto de vectores columna $P = [p1, p2, \dots, pN]$ entonces:

$$P\text{cov}(Y) = [\lambda 1 p1, \lambda 2 p2, \dots, \lambda N\, pN], \quad (6.14)$$

lo cual (escribiendo las λ variables en una matriz como $\lambda = (\lambda 1, \lambda 2, \dots, \lambda N)^T$ y $Z = \text{cov}(X)$) conduce a una ecuación muy interesante:

$$\lambda pi = Zpi \text{ para cada } pi . \quad (6.15)$$

A primera vista no parece muy interesante, pero lo importante es darse cuenta de que λ es un vector columna, mientras que Z es una matriz completa, y se puede aplicar a cada uno de los vectores pi que componen P. Dado que λ es solo un vector columna, todo lo que hace es reescalar los pis; no puede rotarlo o hacer algo complicado por el estilo. Esto nos indica que, de alguna manera, hemos encontrado una matriz P de manera que para las direcciones en las que se escribe P, la matriz Z no se tuerce o rotar esas direcciones, pero simplemente reescalarlas. Estas direcciones son lo suficientemente especiales como para Tienen un nombre: son vectores propios, y la cantidad en que reescalan los ejes (el λs) se conocen como valores propios.

Todos los vectores propios de una matriz cuadrada simétrica A son ortogonales entre sí. Esto indica Nos dice que los vectores propios definen un espacio. Si creamos una matriz E que contiene los vectores propios (normalizados) vectores propios de una matriz A como columnas, entonces esta matriz tomará cualquier vector y rotará en lo que se conoce como el espacio propio. Dado que E es una matriz de rotación, $E^{-1} = E^T$, de modo que Para rotar el vector resultante fuera del espacio propio es necesario multiplicarlo por E^T, dónde Por "normalizado", me refiero a que los vectores propios tienen una longitud unitaria. Entonces, ¿qué deberíamos hacer? ¿Entre rotar el vector hacia el espacio propio y rotarlo de regreso al exterior? La respuesta es que podemos estirar los vectores a lo largo de los ejes. Esto se hace multiplicando el vector por un

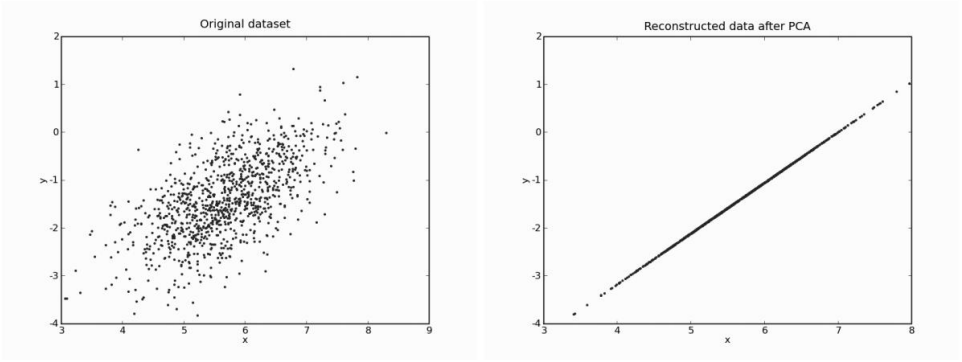

FIGURA 6.7 Al calcular los componentes principales del conjunto de datos 2D de la izquierda y usar solo el primero para reconstruirlo, se produce la línea de datos que se muestra a la derecha, que corre a lo largo del eje principal de la elipse de la que se tomaron las muestras de los datos.

Matriz diagonal que tiene los valores propios a lo largo de su diagonal, D. Por lo tanto, podemos descomponer cualquier matriz cuadrada simétrica A en el siguiente conjunto de matrices: A = EDET, , Y esto es lo que como hemos aplicado a nuestra matriz de covarianza anterior. Esto se denomina descomposición espectral.

Antes de continuar con el algoritmo, hay otro aspecto útil a tener en cuenta. Los valores propios nos indican cuánto estiramiento debemos realizar a lo largo de las dimensiones correspondientes de sus vectores propios. Cuanto mayor sea este reescalamiento, mayor será la variación a lo largo de esa dimensión (ya que si los datos ya estuvieran distribuidos equitativamente, el valor propio sería cercano a 1). Por lo tanto, las dimensiones con valores propios altos presentan una gran variación y, por lo tanto, son dimensiones útiles. Mientras que, para aquellas con valores propios bajos, todos los puntos de datos están muy agrupados y no hay mucha variación en esa dirección. Esto significa que podemos descartar dimensiones con valores propios muy pequeños (normalmente menores que algún parámetro seleccionado).

Es hora de ver el algoritmo que necesitamos.

El algoritmo de análisis de componentes principales

• Escribe N puntos de datos xi = (x1i , • $_{x2i}$, . . . , xM i) como vectores fila

Coloca estos vectores en una matriz X (que tendrá tamaño N × M)

• Centrar los datos restando la media de cada columna y colocándola en la matriz B

• Calcular la matriz de covarianza C = $\frac{1}{norte}$ BT B

• Calcular los valores propios y vectores propios de C, de modo que V−1CV = D, donde V contiene la Los vectores propios de C y D son la matriz de valores propios diagonal M × M

• Ordena las columnas de D en orden de valores propios decrecientes y aplica el mismo orden a las columnas de V

• Rechazar aquellos con valor propio menor que algún η, dejando L dimensiones en los datos

---

NumPy puede calcular los valores y vectores propios. Ambos se devuelven en evals,evecs = np.linalg.eig(x). Esto facilita la implementación del algoritmo:

```
def pca(datos,nRedDim=0,normalizar=1):

    # Datos centrales
    m = np.mean(data,axis=0)
    datos -= m

    # Matriz de covarianza
    C = np.cov(np.transpose(data))

    # Calcular valores propios y ordenarlos en orden descendente
    evals,evecs = np.linalg.eig(C) indices =
    np.argsort(evals) indices = indices[::-1]
    evecs = evecs[:,indices] evals
    = evals[indices]


    si nRedDim>0:
        evecs = evecs[:,:nRedDim]

    Si se normaliza:
        para i en rango(np.shape(evecs)[1]): evecs[:,i] /
            np.linalg.norm(evecs[:,i]) * np.sqrt(evals[i])

    # Produce la nueva matriz de datos
    x = np.dot(np.transpose(evecs),np.transpose(data))
    # Calcular nuevamente los datos
    originales y=np.transpose(np.dot(evecs,x))+m
    return x,y,evals,evecs
```

En las Figuras 6.7 y 6.8 se muestran dos ejemplos diferentes del uso del ACP. El primero muestra datos bidimensionales de una elipse que se mapean en un componente principal, que se encuentra a lo largo del eje principal de la elipse. La Figura 6.8 muestra las dos primeras dimensiones de los datos del iris y muestra que las tres clases se distinguen claramente tras aplicar el ACP.

6.2.1 Relación con el Perceptrón Multicapa

Veremos (en la Sección 14.3.2) que el PCA puede utilizarse en el algoritmo SOM para inicializar los pesos, reduciendo así el aprendizaje necesario y resulta muy útil para la reducción de dimensionalidad. Sin embargo, existe otra razón por la que quienes se interesan por las redes neuronales se interesan por el PCA. Ya la mencionamos al hablar del MLP autoasociativo en la Sección 4.4.5. El MLP autoasociativo calcula algo muy similar a los componentes principales de los datos en los nodos ocultos, y esta es una de las maneras de comprender el funcionamiento de la red.

Por supuesto, calcular los componentes principales con una red neuronal no es necesariamente una buena idea. El PCA es lineal (simplemente rota y traslada los ejes, no puede hacer nada más complejo). Esto es evidente si pensamos en la red, ya que son los nodos ocultos los que...