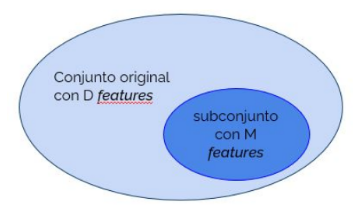
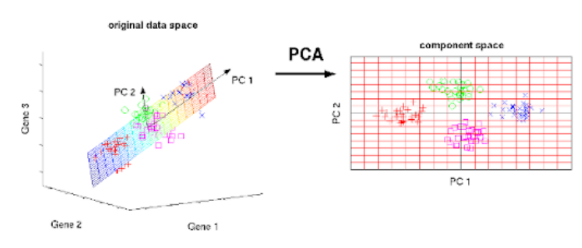
**Reducción de la Dimensionalidad**

A partir de la maldición de la dimensionalidad, sabemos que agregar features mejora la performance de los modelos hasta un cierto punto. Conocemos métodos de features selection que identifican y eligen a las variables más relevantes para el entrenamiento de modelos; se puede decir también que lo que se hace es sacar aquellas variables no agregan valor.

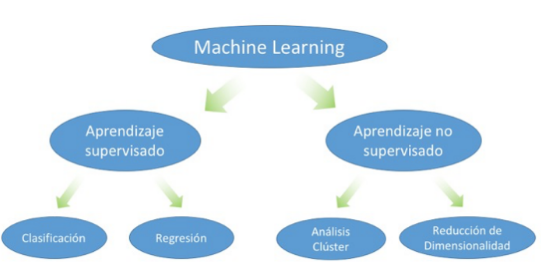


Otra forma de reducir features es con **feature extraction**. Esto consiste en **construir nuevas variables a partir de las existentes**, de forma específica (**transformación lineal**) y con un **orden de importancia asociado.** Luego, se seleccionarán las más relevantes, a partir de las cuales se construirán los modelos.

Ejemplo: tenemos 3 genes (Gene1, Gene2 y Gene3) a partir de los cuales generamos 3 nuevas variables (PC1, PC2 y PC3) y nos quedamos con las 2 primeras que son las más importantes. Entonces pasamos de 3 a 2 dimensiones, pero sin perder tanta información como si hubiéramos filtrado las variables originales de entrada.

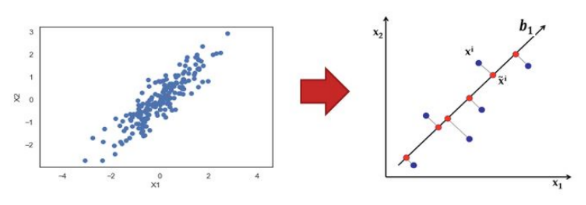


**PCA**: **Principal Component Analysis:** Se trata de un algoritmo de aprendizaje no supervisado, que se usa para reducir dimensiones. También podemos usarlo como modo de visualización, control de ruido, variables latentes.

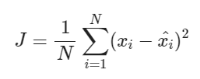


Con **PCA** podemos **proyectar datos a un subespacio de menor dimensionalidad**, manteniendo la **mayor cantidad posible de la información**.

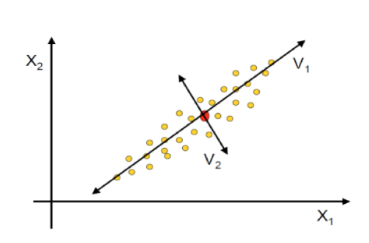
Por ejemplo, contamos con 2 variables x1 y x2 correlacionadas. Queremos proyectarlas sobre una sola coordenada:



Lo que hace **PCA** es **buscar** una **proyección que minimice** el **error cuadrático medio de reconstrucción**.Que maximice la varianza de los datos.

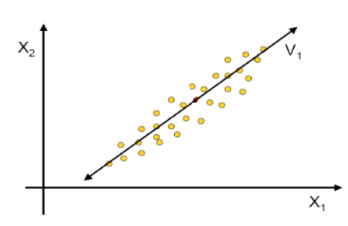
****

Tenemos un Vector V1, tomamos otro V2 que sea perpendicular (u ortogonal, son sinónimos) al V1. Con estos 2 vectores, creamos un **nuevo sistema de coordenadas** en lugar de x1 y x2; a este nuevo sistema lo llamamos **Componentes Principales**. V1 y V2 son el primer y segundo componentes principales:



**PCA define** cada **Componente Principal** como aquél **vector con mayor varianza de los datos**. El primer componente se ubica en la dirección donde más varían los datos. **El segundo es perpendicular al primero** y **toma la mayor cantidad de la variabilidad restante**; y así sucesivamente (a partir de la 4ta dimensión ya no lo podemos dimensionar gráficamente en el espacio). **Los componentes principales son linealmente independientes entre sí**. Una exigencia de PCA es que todos los **Componentes Principales** (CP) estén **normalizados**. Esto hace que se genere una **base ortonormal** (ortogonal y de igual medida) **con** una **cantidad de dimensiones** igual al espacio de las **features originales**.

De esta manera, **si dejamos sólo las primeras Componentes Principales**, lo que hacemos es **reducir la dimensionalidad del dataset** **manteniendo la mayor cantidad posible** de **información**. IE: Si excluimos V2 y nos quedamos solo con V1 en el ejemplo que venimos viendo:



**PCA – Ejemplo:** Vamos a aplicar PCA sobre un dataset, y haciendo esto vamos a ver que al hacerlo podemos:

* **Descubrir Variables Latentes:** Encontrar **variables que no se observan directamente**, pero que podemos inferir a partir de otras variables.
* **Facilitar la visualización de datasets multivariados**, si reducimos las features a dos dimensiones.
* **Conocer los loadings**, que representan los pesos de las variables originales sobre cada CP y conocer así su influencia.

Vamos a trabajar con un dataset de los contribuyentes de 20 provincias argentinas, exceptuando las 4 más significativas económicamente.

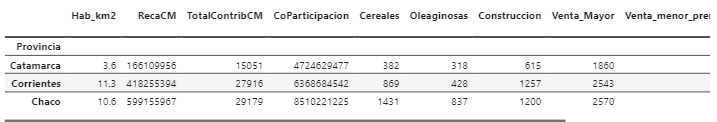
**En Python:**

df = pd.read\_csv(‘../Data/provincias\_actividad.csv’, sep = ‘;’, index\_col=’Provincia’)

print(‘Filas: ’, df.shape[0]. ‘Columnas: ’, df.shape[1])



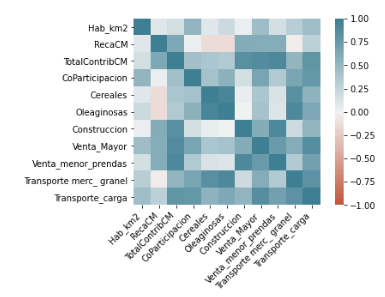
df.head(3)



corr = df.corr()

ax = sns.heatmap(corr, vmin=1, vamx=1, center=0, cmap=sns.diverging\_palette(20, 220, n=200), square = True)

ax.set\_xticklabels(ax.get\_xticklabels(), rotation = 45, horizontalalignment =’right’)



Vemos que las variables no son completamente independientes entre ellas, entonces podemos aplicar PCA.

std = StandardScaler()

df\_std = pd.DataFrame(std.fit\_transform(df))

from sklearn.decomposition import PCA

pca\_df = PCA()

Al instanciar un objeto de la clase PCA sin argumentos, PCA va a conservar todos los CP. Al ajustar con el método fit, se calculan los componentes principales ajustando el dataset al modelo.

pca\_df.fit(df\_std)



np.set\_printoptions(precision=6, suppress=True)

pca\_df.explained\_variance\_



Acá **conseguimos la varianza explicada de cada componente principal**. Puede verse que **a medida que vamos avanzando** en los componentes principales subsiguientes, **la misma va disminuyendo**.

pca\_df.explained\_variance\_ratio\_



Podemos ver así la **proporción de la varianza explicada**. Con los **primeros 3 componentes principales tenemos el 88% de los casos**. La sumatoria de los ratios de todos los componentes principales da 1.

Vamos a graficar el ratio de varianza explicada en función de la cantidad de componentes. Podemos aplicar la **técnica del bastón quebrado** con este gráfico para decidir con qué número de componentes quedarnos: Donde se quiebra la curva; o cuando se estabiliza el descenso:

plt.figure(figsize=(6,5))

plt.plot(range(11), pca\_df.explained\_variance\_ratio\_, ‘-o’, label = ‘Componente individual’)

plt.plot(range(11), np.cumsum(pca\_df.explained\_variance\_ratio\_), ‘-s’, label = ‘Acumulado’)

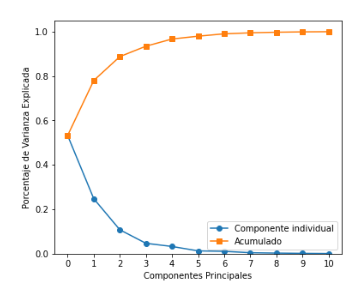
plt.ylabel(‘Porcentaje de varianza explicada’)

plt.xlabel(‘Componentes principales’)

plt.ylim(0,1.05)

plt.xticks(range(11))

plt.legend(loc=0)



Vamos a usar el producto escalar para verificar la ortogonalidad de los componentes principales.

dots = np.ones(pca\_df.components\_.shape[0], pca\_df.components\_.shape[0])

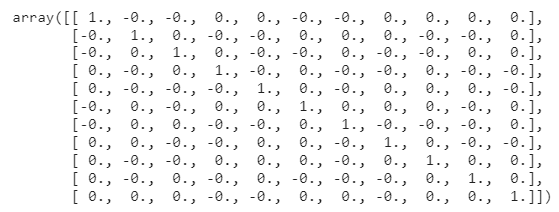
for i in range(pca\_df.components\_.shape[0]):

for j in range(i+1, pca\_df.components\_.shape[0]:

dots[i,j] = np.dot(pca\_df.components\_[i], pca\_df.components\_[j])

dots[j, i] = dots[i, j]

dots

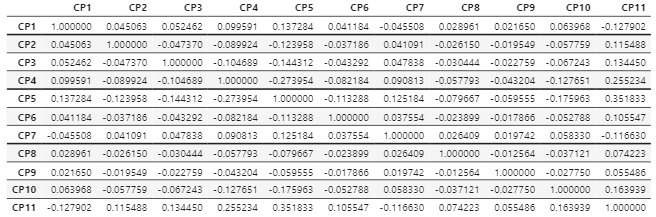


Ahora vamos a calcular la correlación entre los componentes principales. A diferencia de lo que nos pasaba con las variables originales, podemos ver que están poco correlacionados entre sí, con valore en general cercanos a cero.

name\_columns = [‘CP1’, ‘CP2’, ‘CP3’, ‘CP4’, ‘CP5’, ‘CP6’, ‘CP7’, ‘CP8’, ‘CP9’, ‘CP10’, ‘CP11’]

pcs\_df = pd.DataFrame(pca\_df.components\_, columns = name\_columns)

pcs\_df.corr()



**PCA – Características**:

Los Componentes Principales son combinaciones lineales de las variables originales.

Si el **peso de la variable en el componente principal** es **positivo**, entonces ambos tienen **correlación positiva** y **la variable influye sobre el Componente Principal**.

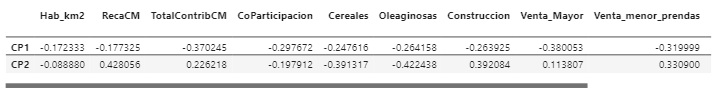
**Si el loading es negativo**, la variable **correlaciona en forma negativa** con el Componente Principal. La **variable también influye** sobre el **Componente Principal, pero de forma negativa**.

**En Python:**

pca\_loadings = pd.DataFrame(pca\_df.components\_.T, index = df.columns).iloc[:,:2]

pca\_loadings.rename(columns = {0: ‘CP1’, 1:’CP2’}, inplace=True)

pca\_loadings.T



Podemos ver que Transporte\_carga y Venta\_Mayor son los que más influyen en forma negativa en CP1; mientras que para CP2, RecaCM es la que más influye de forma positiva, y Oleaginosas es la que más la influye en forma negativa. Básicamente, hay que **mirar los signos** para **ver si influyen de forma positiva o negativa**, y los **valores absolutos** para **ver cuáles influyen más y cuáles menos**, independientemente de en qué forma influyen.

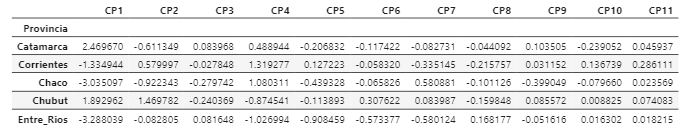
Si consideramos sólo CP1 y CP2, podemos representar los datos gráficamente, perdiendo la menor cantidad de información posible.

**En Python:**

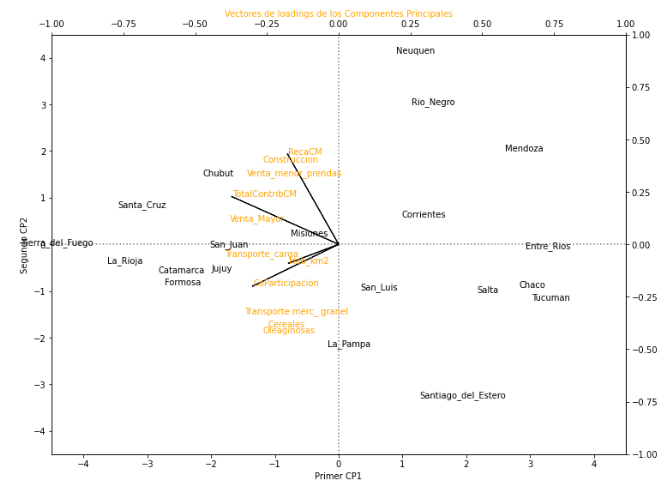
name\_columns = [‘CP1’, ‘CP2’, ‘CP3’, ‘CP4’, ‘CP5’, ‘CP6’, ‘CP7’, ‘CP8’, ‘CP9’, ‘CP10’, ‘CP11’]

df\_pca = pd.DataFrame(pca\_df.fit\_transform(df\_std), columns = name\_columns, index = df.index)

df\_pca.head(5)



biplot(df\_pca, pca\_loadings)



Podemos deducir **variables latentes** a partir del gráfico:

CP1 es la variable con mayor varianza. Si damos vuelta el signo, tiene todos sus valores positivos. Se interpreta como un **componente de tamaño**, que ordena las provincias por tamaño a nivel económico.

CP2 tiene valores positivos y negativos. Se puede interpretar como un **componente de forma**. Los valores negativos más pequeños corresponden a provincias con mayor importancia a nivel productos de campo (Cereales, Oleaginosas, Transporte a granel). Valores positivos mayores parecen asociados a provincias con mayor importancia en generación de energía (Neuquén, Río Negro, Mendoza, Chubut).

**PCA – Representación Algebráica**. [Bienvenidos al infierno de nomenclaturas no explicadas ie: sección Tolosa Paz (que garcha!!!!)]

PCA transforma el dataset original X en un nuevo dataset U que representa las observaciones sobre el nuevo sistema de coordenadas de los componentes principales.

Podemos representarlo así:



W es el conjunto de autovectores que se calcula a partir de los autovalores de la matriz de covarianza de X.

Cada autovector es una columna de W. El total de k columnas seleccionado determina la cantidad k de componentes principales que usamos para reducir la dimensionalidad.

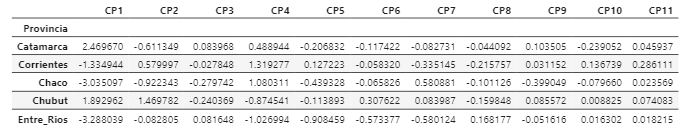
Los autovectores se ordenan de acuerdo a orden de mayor a menor de sus autovalores.

**En Python:**

name\_columns = [‘CP1’, ‘CP2’, ‘CP3’, ‘CP4’, ‘CP5’, ‘CP6’, ‘CP7’, ‘CP8’, ‘CP9’, ‘CP10’, ‘CP11’]

df\_pca = pd.DataFrame(pca\_df.fit\_transform(df\_std), columns = name\_columns, index = df.index)

df\_pca.head(5)

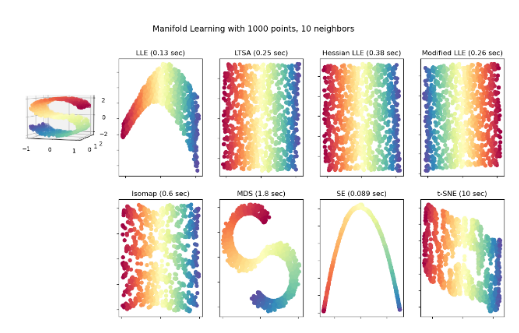


Al aplicar fit\_transform los features originales X (df\_std) al modelo pca\_df, generamos el nuevo dataset U (df\_pca).

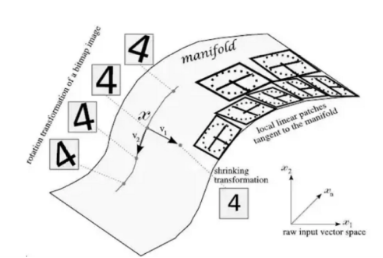
**Conclusiones:**

* PCA es un algoritmo no supervisado, que se aplica para reducir dimensiones.
* A partir de un proceso de **Feature Extraction**, se crean nuevas variables a partir de las variables existentes, llamadas componentes principales, y se selecciona un subconjunto de estos componentes principales buscando minimizar la pérdida de información.
* Es posible **descubrir variables latentes** a inferir a partir de los CPs.
* Permite **visualizar datasets multivariados** al reducir las features originales a 2 dimensiones.
* **Conocer los loadings**, que indican cuánto y en qué dirección influye cada variable en cada CP.

**Manifold Learning:** Se trata de una **técnica que reduce la dimensionalidad de los datos** con **transformaciones no lineales.** Sebasa en la idea de que la dimensionalidad de los datasets es artificialmente alta:



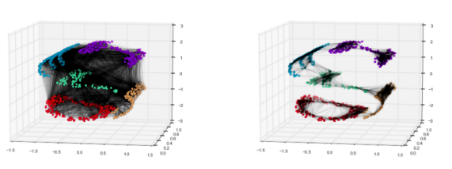
Busca representar los **manifolds:** Superficies suaves y curvas dentro del espacio multidimensional. Las imágenes están formadas por sutiles transformaciones (local linear patches) que PCA no encuentra. Se trata de **tangentes** (rectas que tocan un punto en la curva) que pueden ser calculadas. Suelen usarse muchos parches para poder representar con exactitud al manifold.



Se supone de fondo que una estructura de alta dimensión concentra la información más relevante en un pequeño número de manifolds de baja dimensión (**Hipótesis de Manifold**). Implica dos fuertes suposiciones:

1. Que la **distribución de probabilidad** en conjuntos de datos como imágenes o texto está **altamente concentrada**.
2. **Conectividad**: Que los puntos de datos relevantes están conectados a otros puntos de datos relevantes.

Mientras que PCA preserva la forma de los datos, manifold learning evalúa la distancia entre agrupamientos de puntos. Tratan la superficie curvada como una composición de pequeños barrios:



Lo que hace Manifold Learning es, a partir de las propiedades geométricas de los datos:

* **Implementar clustering** con grupos de puntos similares.
* **Reducir dimensionalidad**, proyectar puntos en un espacio de dimensionalidad menor, preservando la estructura.
* Dados puntos etiquetados y no etiquetados, crear una **función de etiquetado**. (modelo semi-supervisado / supervisado).

Es posible calcular la cercanía con diferentes criterios:

* **Probabilísticos:** la densidad acorta las distancias.
* **Cluster:** Puntos en regiones conectadas comparten las mismas propiedades.
* **Manifold:** medir la distancia a lo largo de los manifold**.**
* **Version mixta**: Dos puntos cercanos son aquellos conectados por un camino corto que atraviesa regiones de alta densidad.

**t-SNE**:t-distributed Stochastic Neighbor Embedding. Genera una representación en 2D que preserva la distancia entre los puntos. Aquellos puntos que estaban cerca en el dataset original, quedan cerca en el dataset resultante y los que estaban lejos, quedan lejos. Lo que hace este algoritmo es **convertir las relaciones o similitudes** del espacio original en **distribuciones t de Student**. Entonces, t-SNE se centra en la **estructura local de los datos** y trata de **extraer grupos locales agrupados** en vez de tratar de desenrollar o desplegar los datos.

**En Python:**

Vamos a trabajar con un conjunto de imágenes de dígitos escritos a mano. Sólo tomando números entre 1 y 6.

from time import time

import numpy as np

import matplotlib.pyplot as plt

from matplotlib import offsetbox

from sklearn import (manifold, datasets)

digits = datasets.load\_digits(n\_class=6)

X = digits.data

y = digits.target

n\_samples, n\_features = X.shape

n\_neighbors = 30

n\_img\_per\_row = 20

img = np.zeros((10 \* n\_img\_per\_row, 10 \* n\_img\_per\_row))

for i in range(n\_img\_per\_row):

ix = 10 \* i + 1

for j in range(n\_img\_per\_row):

iy = 10 \* j + 1

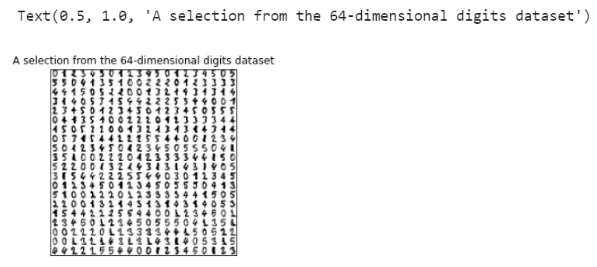
img[ix: ix + 8, iy: iy+8] = X[i \* n\_img\_per\_row + j].reshape((8,8))

plt.imshow(img, cmap = plt.cm.binary)

plt.xticks([])

plt.yticks([])

plt.title(‘A selection from the 64-dimensional digits dataset’)



Con t-NSE vamos a reducir la dimensionalidad representando los dígitos en 2 dimensiones y luego agrupándolos según su valor real indicado en digits.target. Para esto vamos a usar los siguientes hiperparámetros:

* **n\_components** = 2. Reducimos a 2 dimensiones.
* **Init=’pca’**. Se recomienda usar previamente otro método de reducción de dimensionalidad para llevarlo a un número razonable de dimensiones (IE: 50) en caso de que el dataset tenga alta dimensionalidad.

Para graficar la clasificación, usamos la función **plot\_embedding**.

**En Python:**

plt.figure(figsize=(10,8))

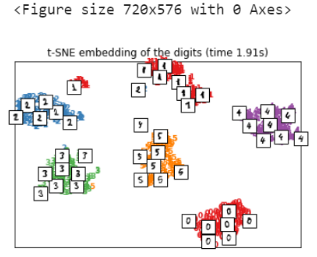
tsne = manifold.TSNE(n\_components = 2, init = ‘pca’, random\_state = 0)

t0 = time()

X\_tsne = tsne.fit\_transform(X)

plot\_embedding(X\_tsne, ‘t-SNE embedding of the digits (time %.2fs)’ % (time() – t0))

plt.show()



**Conclusiones:**

* Mientras que **PCA** se basa en **transformaciones lineales** para reducir la dimensionalidad; **Manifold learning** se basa en **transformaciones no lineales**.
* **PCA** busca crear varios **hiperplanos lineales** para presentar las dimensiones; **Manifold Learning** busca representar **manifolds** (superficies suaves y curvas dentro del espacio multidimensional)
* **PCA** funciona bien con **datos separados en el espacio** (sparse), con **relaciones tangentes, paralelas, envolventes u ortogonales**. **Manifold Learning** en cambio, trabaja mejor con **datos agrupados**, **densos, sesgados,** con **valores extremos**.