# **Unsupervised Learning**

# 

# **Supervised Learning**

# Dataset =

# X

# ,

# y

# X

# = feature matrix

# (

# n

# ,

# p

# )

# y

# = targets vector

# (

# n

# ,

# 1

# )

# Find

# h

# β

# (

# X

# )

# as close to

# y

# as possible

# **Unsupervised Algorithms**

# find patterns in

# X

# , **without supervision from a target**

# y

# Unsupervised Learning helps us **reduce dimensions**

# Feature Engineering/less features (saves time)

# Compress (saves space)

# It also allows us to **cluster data** (= group data points based on similarities)

# Understand data (explore, visualize, etc.)

# Find anomalies/outliers?

# Recommendations

# Semi-supervised classifications

## **Plan**

# Principal Component Analysis (PCA)

# Clustering with K-Means Algorithm

## **1. Principal Component Analysis (PCA)**

# Squashes our high-dimensional dataset down into a lower dimension

# Aims to find the best linear combination of features (= columns) that best represents the underlying structure of the data

#### **Remember Linear Regression Variants?**

# *Polynomial*

# ^

# y

# =

# β

# 0

# +

# β

# 1

# X

# 1

# +

# β

# 2

# X

# 2

# 1

# *Log transformation*

# ^

# y

# =

# β

# 0

# +

# β

# 1

# log

# (

# X

# 1

# )

# *Linear combination of features (while avoiding multicollinearity)*

# ^

# y

# =

# β

# 0

# +

# β

# 1

# X

# 1

# +

# β

# 2

# (

# X

# 2

# +

# X

# 3

# )

# 👉 PCA = finding the **best linear combination** of features

# 

# Z

# 1

# =

# a

# 11

# X

# 1

# +

# a

# 12

# X

# 2

# +

# a

# 13

# X

# 3

# Z

# 2

# =

# a

# 21

# X

# 1

# +

# a

# 22

# X

# 2

# +

# a

# 23

# X

# 3

# Z

# 3

# =

# a

# 31

# X

# 1

# +

# a

# 32

# X

# 2

# +

# a

# 33

# X

# 3

# 

# Canceling **ALL** multicollinearity

# **Ranking** the newly created PCs

# Z

# from most to least important

# Z

# i

# is a so-called Principal Component (PC)

# 

# PCA is essentially a projection of the data that is

# oriented towards specific directions, defined by the **Principal Components**

# orthonormal to each other (0 multicollinearity)

# ranked by decreasing "explaining power" (measured by the variance of our data when projected onto this PC)

#### **Intuition**

# If we had to keep **only one direction** to describe our data, this direction should

# **preserve** most of the **variance** in the data when projected onto it (see spread of red dots)

# minimize "reconstruction errors" (see red lines)

# image.png

# 👉 [stats-exchange story](https://stats.stackexchange.com/a/140579/286995)

# 

# **3 componentsimage.png**👉 [interactive visuals](http://setosa.io/ev/principal-component-analysis/)

# 👉 **PCA helps to reduce dimensions!**

# (

# X

# 1

# ,

# X

# 2

# ,

# X

# 3

# )

# ∼

# (

# Z

# 1

# ,

# Z

# 2

# )

# 

# 📚 [Hands-On Machine Learning](https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/)

### **1.2 Let's Code an Example (with a Wine Dataset)**

# **from** **sklearn.datasets** **import** load\_wine

# 

# wine = load\_wine(as\_frame=**True**)

# X = wine.data

# y = wine.target

# wine\_features = X.columns

# 

# *#⚠️ Data must be centered around its mean before applying PCA ⚠️*

# **from** **sklearn.preprocessing** **import** StandardScaler

# scaler = StandardScaler()

# scaler.fit(X)

# X = pd.DataFrame(scaler.transform(X), columns=wine\_features)

# X

# 

|  | **alcohol** | **malic\_acid** | **ash** | **alcalinity\_of\_ash** | **magnesium** | **total\_phenols** | **flavanoids** | **nonflavanoid\_phenols** | **proanthocyanins** | **color\_intensity** | **hue** | **od280/od315\_of\_diluted\_wines** | **proline** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 1.518613 | -0.562250 | 0.232053 | -1.169593 | 1.913905 | 0.808997 | 1.034819 | -0.659563 | 1.224884 | 0.251717 | 0.362177 | 1.847920 | 1.013009 |
| **1** | 0.246290 | -0.499413 | -0.827996 | -2.490847 | 0.018145 | 0.568648 | 0.733629 | -0.820719 | -0.544721 | -0.293321 | 0.406051 | 1.113449 | 0.965242 |
| **2** | 0.196879 | 0.021231 | 1.109334 | -0.268738 | 0.088358 | 0.808997 | 1.215533 | -0.498407 | 2.135968 | 0.269020 | 0.318304 | 0.788587 | 1.395148 |
| **3** | 1.691550 | -0.346811 | 0.487926 | -0.809251 | 0.930918 | 2.491446 | 1.466525 | -0.981875 | 1.032155 | 1.186068 | -0.427544 | 1.184071 | 2.334574 |
| **4** | 0.295700 | 0.227694 | 1.840403 | 0.451946 | 1.281985 | 0.808997 | 0.663351 | 0.226796 | 0.401404 | -0.319276 | 0.362177 | 0.449601 | -0.037874 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **173** | 0.876275 | 2.974543 | 0.305159 | 0.301803 | -0.332922 | -0.985614 | -1.424900 | 1.274310 | -0.930179 | 1.142811 | -1.392758 | -1.231206 | -0.021952 |
| **174** | 0.493343 | 1.412609 | 0.414820 | 1.052516 | 0.158572 | -0.793334 | -1.284344 | 0.549108 | -0.316950 | 0.969783 | -1.129518 | -1.485445 | 0.009893 |
| **175** | 0.332758 | 1.744744 | -0.389355 | 0.151661 | 1.422412 | -1.129824 | -1.344582 | 0.549108 | -0.422075 | 2.224236 | -1.612125 | -1.485445 | 0.280575 |
| **176** | 0.209232 | 0.227694 | 0.012732 | 0.151661 | 1.422412 | -1.033684 | -1.354622 | 1.354888 | -0.229346 | 1.834923 | -1.568252 | -1.400699 | 0.296498 |
| **177** | 1.395086 | 1.583165 | 1.365208 | 1.502943 | -0.262708 | -0.392751 | -1.274305 | 1.596623 | -0.422075 | 1.791666 | -1.524378 | -1.428948 | -0.595160 |

# 178 rows × 13 columns

# sns.heatmap(pd.DataFrame(X).corr(), cmap='coolwarm')

# 

# <AxesSubplot:>

# 

#### **a) Compute the Principal Components**

# **from** **sklearn.decomposition** **import** PCA

# 

# pca = PCA()

# pca.fit(X)

# 

# PCA()

# *# Access our 13 PCs*

# W = pca.components\_

# 

# *# Print PCs as COLUMNS*

# W = pd.DataFrame(W.T,

# index=wine\_features,

# columns=[f'PC**{**i**}**' **for** i **in** range(1, 14)])

# W

# 

|  | **PC1** | **PC2** | **PC3** | **PC4** | **PC5** | **PC6** | **PC7** | **PC8** | **PC9** | **PC10** | **PC11** | **PC12** | **PC13** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **alcohol** | 0.144329 | -0.483652 | -0.207383 | -0.017856 | -0.265664 | -0.213539 | -0.056396 | -0.396139 | 0.508619 | 0.211605 | -0.225917 | -0.266286 | 0.014970 |
| **malic\_acid** | -0.245188 | -0.224931 | 0.089013 | 0.536890 | 0.035214 | -0.536814 | 0.420524 | -0.065827 | -0.075283 | -0.309080 | 0.076486 | 0.121696 | 0.025964 |
| **ash** | -0.002051 | -0.316069 | 0.626224 | -0.214176 | -0.143025 | -0.154475 | -0.149171 | 0.170260 | -0.307694 | -0.027125 | -0.498691 | -0.049622 | -0.141218 |
| **alcalinity\_of\_ash** | -0.239320 | 0.010591 | 0.612080 | 0.060859 | 0.066103 | 0.100825 | -0.286969 | -0.427970 | 0.200449 | 0.052799 | 0.479314 | -0.055743 | 0.091683 |
| **magnesium** | 0.141992 | -0.299634 | 0.130757 | -0.351797 | 0.727049 | -0.038144 | 0.322883 | 0.156361 | 0.271403 | 0.067870 | 0.071289 | 0.062220 | 0.056774 |
| **total\_phenols** | 0.394661 | -0.065040 | 0.146179 | 0.198068 | -0.149318 | 0.084122 | -0.027925 | 0.405934 | 0.286035 | -0.320131 | 0.304341 | -0.303882 | -0.463908 |
| **flavanoids** | 0.422934 | 0.003360 | 0.150682 | 0.152295 | -0.109026 | 0.018920 | -0.060685 | 0.187245 | 0.049578 | -0.163151 | -0.025694 | -0.042899 | 0.832257 |
| **nonflavanoid\_phenols** | -0.298533 | -0.028779 | 0.170368 | -0.203301 | -0.500703 | 0.258594 | 0.595447 | 0.233285 | 0.195501 | 0.215535 | 0.116896 | 0.042352 | 0.114040 |
| **proanthocyanins** | 0.313429 | -0.039302 | 0.149454 | 0.399057 | 0.136860 | 0.533795 | 0.372139 | -0.368227 | -0.209145 | 0.134184 | -0.237363 | -0.095553 | -0.116917 |
| **color\_intensity** | -0.088617 | -0.529996 | -0.137306 | 0.065926 | -0.076437 | 0.418644 | -0.227712 | 0.033797 | 0.056218 | -0.290775 | 0.031839 | 0.604222 | -0.011993 |
| **hue** | 0.296715 | 0.279235 | 0.085222 | -0.427771 | -0.173615 | -0.105983 | 0.232076 | -0.436624 | 0.085828 | -0.522399 | -0.048212 | 0.259214 | -0.089889 |
| **od280/od315\_of\_diluted\_wines** | 0.376167 | 0.164496 | 0.166005 | 0.184121 | -0.101161 | -0.265851 | -0.044764 | 0.078108 | 0.137227 | 0.523706 | 0.046423 | 0.600959 | -0.156718 |
| **proline** | 0.286752 | -0.364903 | -0.126746 | -0.232071 | -0.157869 | -0.119726 | 0.076805 | -0.120023 | -0.575786 | 0.162116 | 0.539270 | -0.079402 | 0.014447 |

# ☝️ Each PC is a **linear combination of initial wine features**

#### **b) *Project* our dataset into this new space of PCs**

# X\_proj = pca.transform(X)

# X\_proj = pd.DataFrame(X\_proj, columns=[f'PC**{**i**}**' **for** i **in** range(1, 14)])

# X\_proj

# 

|  | **PC1** | **PC2** | **PC3** | **PC4** | **PC5** | **PC6** | **PC7** | **PC8** | **PC9** | **PC10** | **PC11** | **PC12** | **PC13** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 3.316751 | -1.443463 | -0.165739 | -0.215631 | 0.693043 | -0.223880 | 0.596427 | 0.065139 | 0.641443 | 1.020956 | -0.451563 | 0.540810 | -0.066239 |
| **1** | 2.209465 | 0.333393 | -2.026457 | -0.291358 | -0.257655 | -0.927120 | 0.053776 | 1.024416 | -0.308847 | 0.159701 | -0.142657 | 0.388238 | 0.003637 |
| **2** | 2.516740 | -1.031151 | 0.982819 | 0.724902 | -0.251033 | 0.549276 | 0.424205 | -0.344216 | -1.177834 | 0.113361 | -0.286673 | 0.000584 | 0.021717 |
| **3** | 3.757066 | -2.756372 | -0.176192 | 0.567983 | -0.311842 | 0.114431 | -0.383337 | 0.643593 | 0.052544 | 0.239413 | 0.759584 | -0.242020 | -0.369484 |
| **4** | 1.008908 | -0.869831 | 2.026688 | -0.409766 | 0.298458 | -0.406520 | 0.444074 | 0.416700 | 0.326819 | -0.078366 | -0.525945 | -0.216664 | -0.079364 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **173** | -3.370524 | -2.216289 | -0.342570 | 1.058527 | -0.574164 | -1.108788 | 0.958416 | -0.146097 | -0.022498 | -0.304117 | 0.139228 | 0.170786 | -0.114427 |
| **174** | -2.601956 | -1.757229 | 0.207581 | 0.349496 | 0.255063 | -0.026465 | 0.146894 | -0.552427 | -0.097969 | -0.206061 | 0.258198 | -0.279431 | -0.187371 |
| **175** | -2.677839 | -2.760899 | -0.940942 | 0.312035 | 1.271355 | 0.273068 | 0.679235 | 0.047024 | 0.001222 | -0.247997 | 0.512492 | 0.698766 | 0.072078 |
| **176** | -2.387017 | -2.297347 | -0.550696 | -0.688285 | 0.813955 | 1.178783 | 0.633975 | 0.390829 | 0.057448 | 0.491490 | 0.299822 | 0.339821 | -0.021866 |
| **177** | -3.208758 | -2.768920 | 1.013914 | 0.596903 | -0.895193 | 0.296092 | 0.005741 | -0.292914 | 0.741660 | -0.117969 | -0.229964 | -0.188788 | -0.323965 |

# 178 rows × 13 columns

# ☝️ 178 wine bottles, each expressed as a linear combination of 13 Principal Components

# ✅ As expected, the PCA reduces multicollinearity to the absolute minimum (0)!

# sns.heatmap(X\_proj.corr(), cmap='coolwarm');

# 

# 

# ✅ Our wine dataset is also easier to observe in this new space

# *# 2D-slice*

# 

# plt.figure(figsize=(13,5))

# plt.subplot(1,2,1)

# plt.title('X1 vs. X0 before PCA (initial space)'); plt.xlabel('X0'); plt.ylabel('X1')

# plt.scatter(X.iloc[:,0], X.iloc[:,1])

# 

# plt.subplot(1,2,2)

# plt.title('PC1 vs PC2 (new space)'); plt.xlabel('PC 1'); plt.ylabel('PC 2')

# plt.scatter(X\_proj.iloc[:,0], X\_proj.iloc[:,1]);

# 

# 

# Adding true labels makes it even clearerimage.png

#### **💡 "Projecting" data onto a new space is a simple matrix multiplication**

# 

# *# Computational proof*

# W = pca.components\_.T

# print("Shape of W: ", W.shape)

# print("Shape of X", X.shape)

# 

# Shape of W: (13, 13)

# Shape of X (178, 13)

# 

# np.allclose(

# pca.transform(X),

# np.dot(X,W)

# )

# 

# True

### **1.3 How are Principal Components Computed (Mathematically)?**

# This is the hard part

# 

# We can do it with NumPy

# np.linalg.eig(M) computes the eig\_vals and eig\_vecs of M

# Covariance Matrix =

# X

# T

# X

# of shape (p,p) (if features are centered)

# ❗️ eig() decomposition can take very long

# ℹ️ [Eigenvalues and Eigenvectors](https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors)

# *# Compute PCs*

# eig\_vals, eig\_vecs = np.linalg.eig(np.dot(X.T,X))

# 

# *# Show all 13 principal components (unranked)*

# W = pd.DataFrame(eig\_vecs,

# index=wine\_features,

# columns=[f'PC**{**i**}**' **for** i **in** range(1, 14)])

# W

# 

|  | **PC1** | **PC2** | **PC3** | **PC4** | **PC5** | **PC6** | **PC7** | **PC8** | **PC9** | **PC10** | **PC11** | **PC12** | **PC13** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **alcohol** | -0.144329 | 0.483652 | -0.207383 | 0.017856 | -0.265664 | 0.213539 | 0.056396 | -0.014970 | 0.396139 | -0.266286 | -0.508619 | -0.225917 | 0.211605 |
| **malic\_acid** | 0.245188 | 0.224931 | 0.089013 | -0.536890 | 0.035214 | 0.536814 | -0.420524 | -0.025964 | 0.065827 | 0.121696 | 0.075283 | 0.076486 | -0.309080 |
| **ash** | 0.002051 | 0.316069 | 0.626224 | 0.214176 | -0.143025 | 0.154475 | 0.149171 | 0.141218 | -0.170260 | -0.049622 | 0.307694 | -0.498691 | -0.027125 |
| **alcalinity\_of\_ash** | 0.239320 | -0.010591 | 0.612080 | -0.060859 | 0.066103 | -0.100825 | 0.286969 | -0.091683 | 0.427970 | -0.055743 | -0.200449 | 0.479314 | 0.052799 |
| **magnesium** | -0.141992 | 0.299634 | 0.130757 | 0.351797 | 0.727049 | 0.038144 | -0.322883 | -0.056774 | -0.156361 | 0.062220 | -0.271403 | 0.071289 | 0.067870 |
| **total\_phenols** | -0.394661 | 0.065040 | 0.146179 | -0.198068 | -0.149318 | -0.084122 | 0.027925 | 0.463908 | -0.405934 | -0.303882 | -0.286035 | 0.304341 | -0.320131 |
| **flavanoids** | -0.422934 | -0.003360 | 0.150682 | -0.152295 | -0.109026 | -0.018920 | 0.060685 | -0.832257 | -0.187245 | -0.042899 | -0.049578 | -0.025694 | -0.163151 |
| **nonflavanoid\_phenols** | 0.298533 | 0.028779 | 0.170368 | 0.203301 | -0.500703 | -0.258594 | -0.595447 | -0.114040 | -0.233285 | 0.042352 | -0.195501 | 0.116896 | 0.215535 |
| **proanthocyanins** | -0.313429 | 0.039302 | 0.149454 | -0.399057 | 0.136860 | -0.533795 | -0.372139 | 0.116917 | 0.368227 | -0.095553 | 0.209145 | -0.237363 | 0.134184 |
| **color\_intensity** | 0.088617 | 0.529996 | -0.137306 | -0.065926 | -0.076437 | -0.418644 | 0.227712 | 0.011993 | -0.033797 | 0.604222 | -0.056218 | 0.031839 | -0.290775 |
| **hue** | -0.296715 | -0.279235 | 0.085222 | 0.427771 | -0.173615 | 0.105983 | -0.232076 | 0.089889 | 0.436624 | 0.259214 | -0.085828 | -0.048212 | -0.522399 |
| **od280/od315\_of\_diluted\_wines** | -0.376167 | -0.164496 | 0.166005 | -0.184121 | -0.101161 | 0.265851 | 0.044764 | 0.156718 | -0.078108 | 0.600959 | -0.137227 | 0.046423 | 0.523706 |
| **proline** | -0.286752 | 0.364903 | -0.126746 | 0.232071 | -0.157869 | 0.119726 | -0.076805 | -0.014447 | 0.120023 | -0.079402 | 0.575786 | 0.539270 | 0.162116 |

### **1.4 PCs are ranked by order of importance**

# P

# C

# s

# are ranked by share of **explained variance**

# V

# a

# r

# (

# P

# C

# i

# )

# V

# a

# r

# (

# X

# )

# ❗️ Remember: information comes in the form of variation ❗️ PC with most variance is the most important one

# *# Let's compute it*

# X\_proj.std()\*\*2 / ((X.std()\*\*2).sum())

# 

# PC1 0.361988

# PC2 0.192075

# PC3 0.111236

# PC4 0.070690

# PC5 0.065633

# PC6 0.049358

# PC7 0.042387

# PC8 0.026807

# PC9 0.022222

# PC10 0.019300

# PC11 0.017368

# PC12 0.012982

# PC13 0.007952

# dtype: float64

# ☝️ scikit-learn PCA has indeed already ranked them

# 36% of the dataset’s variance lies along the first axis

# *# Sklearn provides it automatically*

# pca.explained\_variance\_ratio\_

# 

# array([0.36198848, 0.1920749 , 0.11123631, 0.0706903 , 0.06563294,

# 0.04935823, 0.04238679, 0.02680749, 0.02222153, 0.01930019,

# 0.01736836, 0.01298233, 0.00795215])

# plt.plot(pca.explained\_variance\_ratio\_)

# plt.xlabel('Principal Component'); plt.ylabel('**% e**xplained variance');

# 

# 

# PCA redistributes the ratio among the new features in the **most unequal way**

### **1.5 PCA for Dimensionality Reduction**

# 👉 Having computed all PCs, we can now keep only the k most important ones!

# ❓ Why would we want less features? Because it means we can

# compress data

# reduce model complexity & fit time

# reduce overfitting

#### **How to choose k?**

# It's a **trade-off** between compression and performance

# plt.plot(np.cumsum(pca.explained\_variance\_ratio\_))

# plt.ylim(ymin=0)

# plt.title('cumulated share of explained variance')

# plt.xlabel('# of principal component used');

# 

# 

#### **The Elbow Method**

# Look for the *inflection point* in the explained variance chart

# Here, k=3 looks promising

# 

#### **✏️ Test Model Performance (with k=3 Dimensions)**

# *# Fit a PCA with only 3 components*

# pca3 = PCA(n\_components=3).fit(X)

# 

# *# Project your data into 3 dimensions*

# X\_proj3 = pd.DataFrame(pca3.fit\_transform(X), columns=['PC1', 'PC2', 'PC3'])

# 

# *# We have "compressed" our dataset in 3D*

# X\_proj3

# 

|  | **PC1** | **PC2** | **PC3** |
| --- | --- | --- | --- |
| **0** | 3.316751 | -1.443463 | -0.165739 |
| **1** | 2.209465 | 0.333393 | -2.026457 |
| **2** | 2.516740 | -1.031151 | 0.982819 |
| **3** | 3.757066 | -2.756372 | -0.176192 |
| **4** | 1.008908 | -0.869831 | 2.026688 |
| **...** | ... | ... | ... |
| **173** | -3.370524 | -2.216289 | -0.342570 |
| **174** | -2.601956 | -1.757229 | 0.207581 |
| **175** | -2.677839 | -2.760899 | -0.940942 |
| **176** | -2.387017 | -2.297347 | -0.550696 |
| **177** | -3.208758 | -2.768920 | 1.013914 |

# 178 rows × 3 columns

# **from** **sklearn.linear\_model** **import** LogisticRegression

# **from** **sklearn.model\_selection** **import** cross\_val\_score

# 

# print("accuracy 3 PCs")

# print(cross\_val\_score(LogisticRegression(), X\_proj3, y, cv=5).mean())

# 

# print("**\n**accuracy all 13 initial features")

# print(cross\_val\_score(LogisticRegression(), X, y, cv=5).mean())

# 

# accuracy 3 PCs

# 0.9609523809523809

# 

# accuracy all 13 initial features

# 0.9888888888888889

# 

#### **Decompress**

# ❓ Can you **perfectly** reconstruct X from X\_proj3?

# Not if you kept k < 13 dimensions; information has been lost

# We can *approximate* X by reconstructing it with inverse\_transform()

# X\_reconstructed = pca3.inverse\_transform(X\_proj3)

# X\_reconstructed.shape

# 

# (178, 13)

# plt.figure(figsize=(15,4))

# plt.subplot(1,2,1)

# sns.heatmap(X)

# plt.title("original data")

# plt.subplot(1,2,2)

# plt.title("reconstructed data")

# sns.heatmap(X\_reconstructed);

# 

# 

### **1.6 Limitations of PCA**

# Watch out for **manifolds** 👀

# A manifold is an N-dimensional shape that can be bent and twisted into a higher dimensional shape 🍥

# Below we can see our data distribution before and after PCA has been applied

# 

# 📚 [Hands-On Machine Learning](https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/)

# Other dimensionality reduction techniques:

# **t-Distributed Stochastic Neighbor Embedding (t-SNE)** — Aims to reduce dimensionality while keeping similar observations close together and dissimilar ones apart. This is a great technique for visualizing clusters of higher dimensions

# **Kernel PCA** — Captures non-linear patterns (similar principle to SVM kernels)

## **Summary**

# We use PCA to deal with high-dimensional datasets; some pros are:

# Better visualization of the data

# Reduction of the effects of the curse of dimensionality

# Reduction of file size

# PCA compresses the datasets into a lower-dimensional state by projecting observations onto a new space

# More variation, more information, easier to distinguish between observations

# When we use PCA we lose data interpretability

# **2. Clustering (Intro Through K-Means)**

# The process of organizing data points into groups whose members are similar in some way

# Find **categories** (classes, segments) of **unlabelled** data rather than just trying to reduce dimensionality

# 

# 👉 Works better on data that is already clustered, geometrically speaking 👉 Use PCA for dimensionality reduction beforehand Euclidean distances work better in lower dimensions)!

## **2.1 K-Means Explained**

# 

# Choose the number of clusters K to look for

# Initialize K **centroids** at random

# Compute the **mean square distance** between each data point and each centroid

# Assign each data point to the closest centroid (a cluster is formed)

# Compute the mean

# μ

# j

# of each cluster, the result of which becomes your new centroid

# One epoch is done, repeat from step 3!

# 

# **In practice**

# K-means is usually run a few times with different random initializations

# We can use a random mini-batch at each epoch instead of the full dataset

# The algorithm is quite fast

# image.png

## **2.2 Implementation**

#### **In Scikit-learn**

# <https://scikit-learn.org/stable/modules/clustering.html>

# Use

# scikit.clustering.KMeans

# scikit.clustering.MiniBatchKMeans — same but uses batch samples instead of the whole dataset, in order to go faster

#### **📝 Let's try to find k = 3 clusters for our wine dataset**

# (suppose we don't know the true labels)

# 💡 First, let's place ourselves in the Principal Component space we had already computed

# Although not mandatory, applying PCA first helps to separate data more easily!

# X\_proj

# 

|  | **PC1** | **PC2** | **PC3** | **PC4** | **PC5** | **PC6** | **PC7** | **PC8** | **PC9** | **PC10** | **PC11** | **PC12** | **PC13** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 3.316751 | -1.443463 | -0.165739 | -0.215631 | 0.693043 | -0.223880 | 0.596427 | 0.065139 | 0.641443 | 1.020956 | -0.451563 | 0.540810 | -0.066239 |
| **1** | 2.209465 | 0.333393 | -2.026457 | -0.291358 | -0.257655 | -0.927120 | 0.053776 | 1.024416 | -0.308847 | 0.159701 | -0.142657 | 0.388238 | 0.003637 |
| **2** | 2.516740 | -1.031151 | 0.982819 | 0.724902 | -0.251033 | 0.549276 | 0.424205 | -0.344216 | -1.177834 | 0.113361 | -0.286673 | 0.000584 | 0.021717 |
| **3** | 3.757066 | -2.756372 | -0.176192 | 0.567983 | -0.311842 | 0.114431 | -0.383337 | 0.643593 | 0.052544 | 0.239413 | 0.759584 | -0.242020 | -0.369484 |
| **4** | 1.008908 | -0.869831 | 2.026688 | -0.409766 | 0.298458 | -0.406520 | 0.444074 | 0.416700 | 0.326819 | -0.078366 | -0.525945 | -0.216664 | -0.079364 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **173** | -3.370524 | -2.216289 | -0.342570 | 1.058527 | -0.574164 | -1.108788 | 0.958416 | -0.146097 | -0.022498 | -0.304117 | 0.139228 | 0.170786 | -0.114427 |
| **174** | -2.601956 | -1.757229 | 0.207581 | 0.349496 | 0.255063 | -0.026465 | 0.146894 | -0.552427 | -0.097969 | -0.206061 | 0.258198 | -0.279431 | -0.187371 |
| **175** | -2.677839 | -2.760899 | -0.940942 | 0.312035 | 1.271355 | 0.273068 | 0.679235 | 0.047024 | 0.001222 | -0.247997 | 0.512492 | 0.698766 | 0.072078 |
| **176** | -2.387017 | -2.297347 | -0.550696 | -0.688285 | 0.813955 | 1.178783 | 0.633975 | 0.390829 | 0.057448 | 0.491490 | 0.299822 | 0.339821 | -0.021866 |
| **177** | -3.208758 | -2.768920 | 1.013914 | 0.596903 | -0.895193 | 0.296092 | 0.005741 | -0.292914 | 0.741660 | -0.117969 | -0.229964 | -0.188788 | -0.323965 |

# 178 rows × 13 columns

# **from** **sklearn.cluster** **import** KMeans

# 

# *# Fit K-means*

# km = KMeans(n\_clusters=3)

# km.fit(X\_proj)

# 

# KMeans(n\_clusters=3)

# *# The 3 centroids' coordinates (expressed in the space of PCs)*

# km.cluster\_centers\_.shape

# 

# (3, 13)

# *# The 177 observations are classified automatically*

# km.labels\_

# 

# array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,

# 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,

# 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 0, 1, 1, 1, 1,

# 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,

# 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

# 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,

# 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

# 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

# 0, 0], dtype=int32)

# plt.scatter(X\_proj.iloc[:,0], X\_proj.iloc[:,1], c=km.labels\_)

# plt.title('KMeans clustering'); plt.xlabel('PC 1'); plt.ylabel('PC 2');

# 

# 

# 💡 In our case, we *know* the true labels; let's measure performance

# *# Visualization*

# plt.figure(figsize=(13,5))

# 

# plt.subplot(1,2,1)

# plt.scatter(X\_proj.iloc[:,0], X\_proj.iloc[:,1], c=km.labels\_)

# plt.title('KMeans clustering'); plt.xlabel('PC 1'); plt.ylabel('PC 2')

# 

# plt.subplot(1,2,2)

# plt.scatter(X\_proj.iloc[:,0], X\_proj.iloc[:,1], c=y)

# plt.title('True wine labels'); plt.xlabel('PC 1'); plt.ylabel('PC 2');

# 

# 

# *# Accuracy*

# **from** **sklearn.metrics** **import** accuracy\_score

# 

# y\_pred = pd.Series(km.labels\_).map({0:0, 1:2, 2:1}) *# WARNING: change this manually!*

# accuracy\_score(y\_pred, y)

# 

# 0.9662921348314607

#### **Predict?**

# We can use the unsupervised K-means algorithm to **predict** (classify) a new X

# *# Build DF with column names from X\_proj and some random data*

# new\_X = pd.DataFrame(data = np.random.random((1,13)), columns = X\_proj.columns)

# 

# km.predict(new\_X)

# 

# array([1], dtype=int32)

## **2.3 K-Means' Loss Function?**

# km.fit(X) finds parameters

# β

# that minimize a loss

# Each

# β

# j

# parameter is the **centroid**

# μ

# j

# of its respective cluster

# C

# j

# 

# The loss function is called **inertia**

# L

# (

# μ

# )

# 

# = **sum** of **squared distance** between each observation and their **closest centroid**

# = sum of **within-cluster sum of squares** (WCSS)

# = variance

# inertia

# =

# L

# (

# μ

# )

# =

# K

# ∑

# j

# =

# 1

# ∑

# x

# i

# ∈

# C

# j

# (

# |

# |

# x

# i

# −

# μ

# j

# |

# |

# 2

# )

### **Choosing Hyperparameter K**

# Choose K such that the inertia (Kmeans().inertia\_) is minimized

# Use the **elbow method** here as well

# inertias = []

# ks = range(1,10)

# 

# **for** k **in** ks:

# km\_test = KMeans(n\_clusters=k).fit(X)

# inertias.append(km\_test.inertia\_)

# 

# plt.plot(ks, inertias)

# plt.xlabel('k cluster number')

# 

# Text(0.5, 0, 'k cluster number')

# 

### **What can we use it for?**

# 👉 Document classification (finding unlabeled categories or topics)

# 👉 Delivery store optimization (find the optimal number of launch locations)

# 👉 Customer segmentation (classify different types of customer based on their behavior)

# <https://dzone.com/articles/10-interesting-use-cases-for-the-k-means-algorithm>

## **2.4 There are many other clustering approaches**

# <https://scikit-learn.org/stable/modules/clustering.html>

# 

# 

## **Bibliography**

# [PCA explained to your grandmother](https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues) 1700 upvotes on StackExchange 💫

# 📄 [PCA for ML](https://towardsdatascience.com/using-principal-component-analysis-pca-for-machine-learning-b6e803f5bf1e)

# 📄 [KMeans explained](https://towardsdatascience.com/k-means-clustering-explain-it-to-me-like-im-10-e0badf10734a)

# **🚀 Your turn!**

# **Projects Info Session**

# This slide deck contains the slides for the Projects Info Session as Google Slides. Before presenting, make sure to **create a copy** on Google Slides and **adapt the following slides**:

# **Slide 2**

# Change <PROJECTS\_START\_ON> and <PROJECTS\_END\_ON> to the corresponding **dates**for your program

# Change <PROJECTS\_COUNT> to the number of projects your batch will have

# **Slide 11**

# Change <SUBMISSION\_DEADLINE> to the **name of the unit** where the deadline is (e.g. **Recurrent Neural Networks**)

# Change <VALIDATION\_DEADLINE> to the **name of the unit** where the deadline is (e.g. **Automate Model Lifecycle**)

# **Slide 12**

# Change <BATCH\_NUMBER> to the corresponding batch number

# Change <PITCH\_SESSION> to the **name of the unit** where the session is happening (e.g. **User Interface**)

# **Slides**

## [**Google Slides**](https://docs.google.com/presentation/d/1ir9_mUwUphSavHAfclRhxMFKonReaa3fdXCVzMlSm2c/edit?usp=sharing)

# 