DATA SOCIETY®

Clustering - day 2

“One should look for what is and not what he thinks should be.”
-Albert Einstein.

Module completion checklist

Objective	Complete
Define DBSCAN and how it compares to k-means	
Parameter estimation for ϵ and MinPts	
Run and visualize DBSCAN for an arbitrary distance	
Optimize parameters of DBSCAN and discuss pitfalls	
Recognize high dimensional data and the challenges that it introduces	
Illustrate the concept of PCA	
Transform data to run PCA on CMP dataset and apply the PCA algorithm	
Choose the optimal number of dimensions to retain and discuss common pitfalls of PCA	

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `af-werx` folder

```
# Set `main_dir` to the location of your `af-werx` folder (for Linux).
main_dir = "/home/[username]/af-werx"
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac).
main_dir = '/Users/[username]/af-werx'
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Windows).
main_dir = "C:\\Users\\[username]\\af-werx"
```

```
# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = main_dir + "/data"
```

Loading packages

- Load the packages we will be using

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import NearestNeighbors
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
np.set_printoptions(suppress=True) #<- suppress scientific notations
```

Working directory

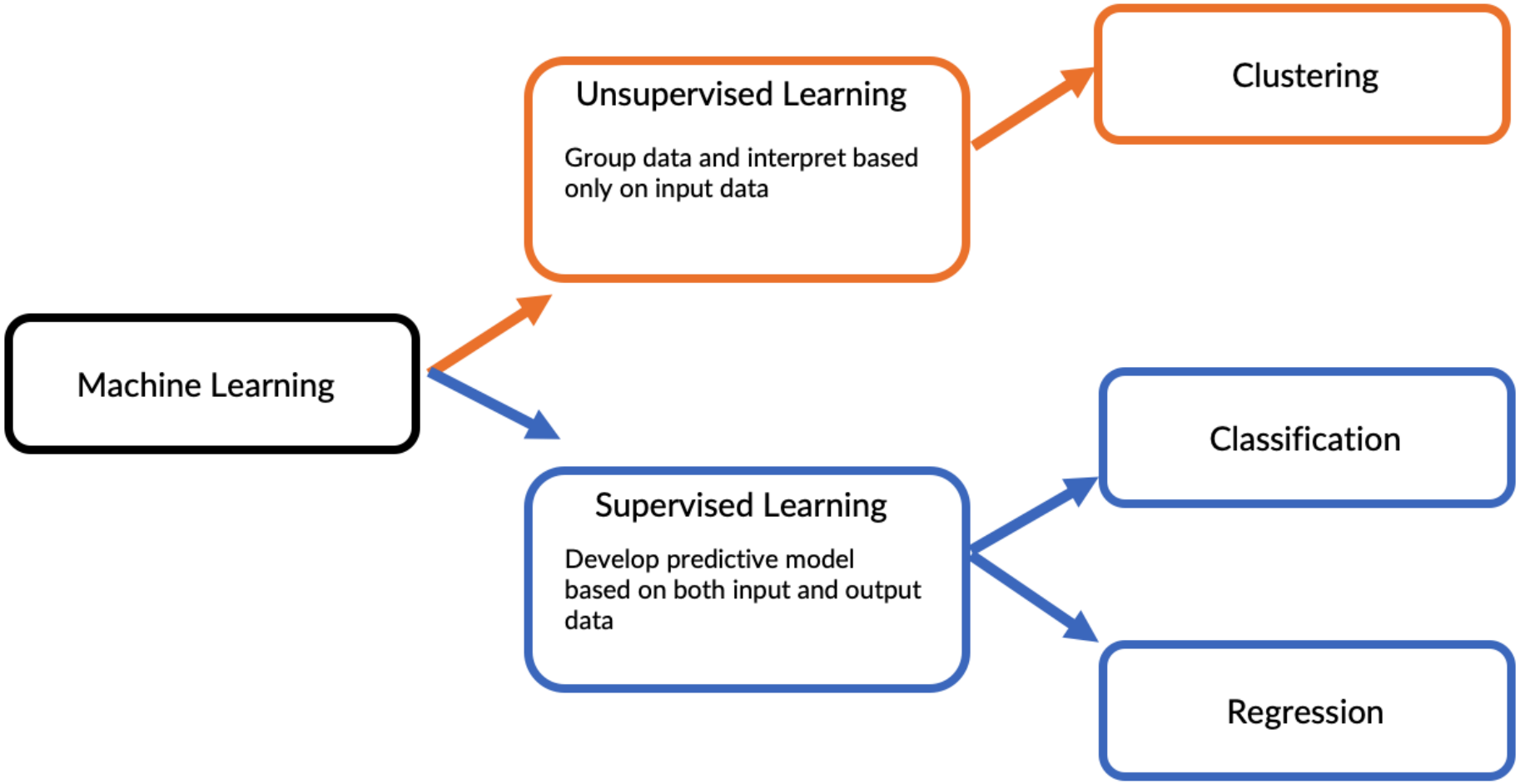
- Set working directory to `data_dir`

```
# Set working directory.
os.chdir(data_dir)
```

```
# Check working directory.
print(os.getcwd())
```

```
/home/[user-name]/Desktop/af-werx/data
```

Recap: Supervised vs. unsupervised learning

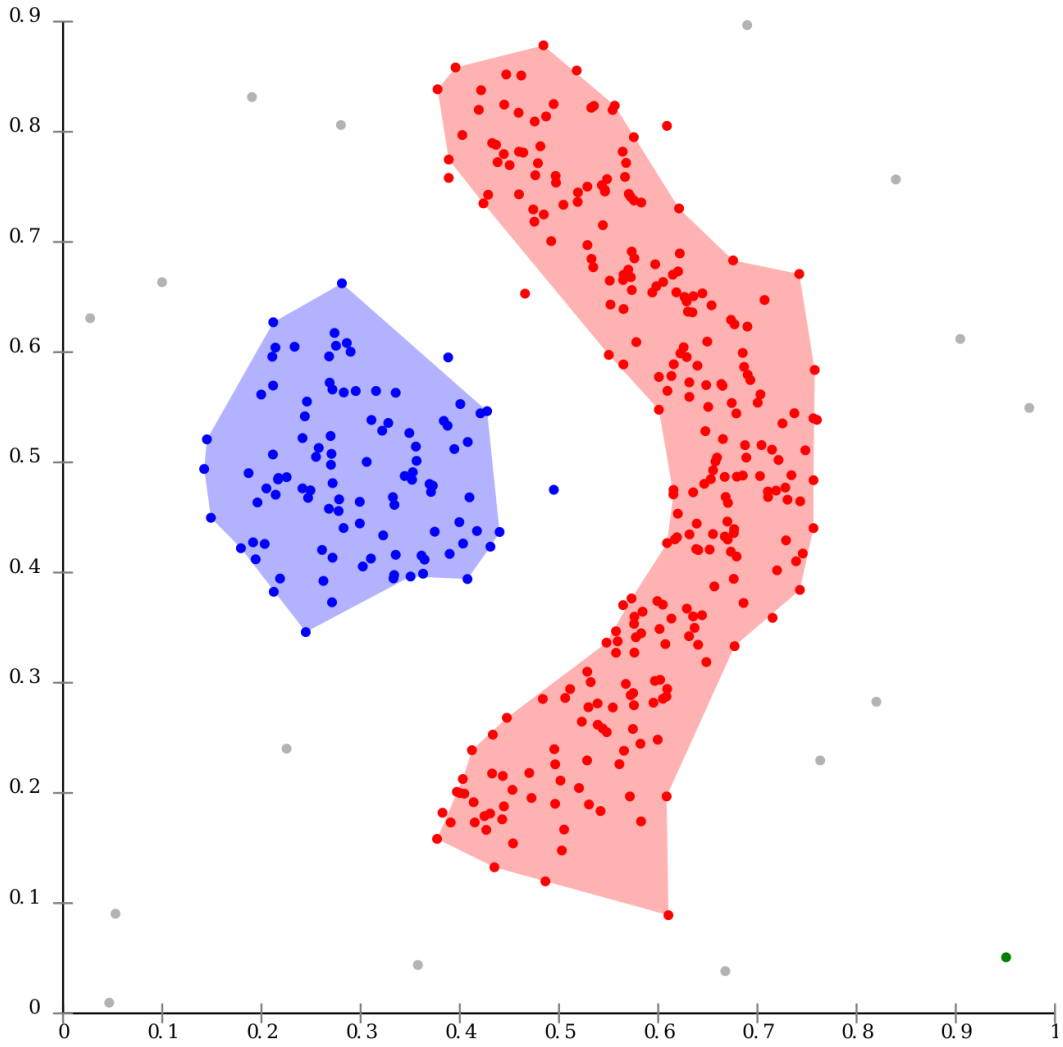


Recap: clustering

- **What is clustering?**
  - Unsupervised machine learning method for finding similarity between groups
- **What is k-means?**
  - Determines `k` centroids in the data and clusters points by assigning them to the nearest centroid
  - Finds cluster centers that are representative to the cluster regions of the data

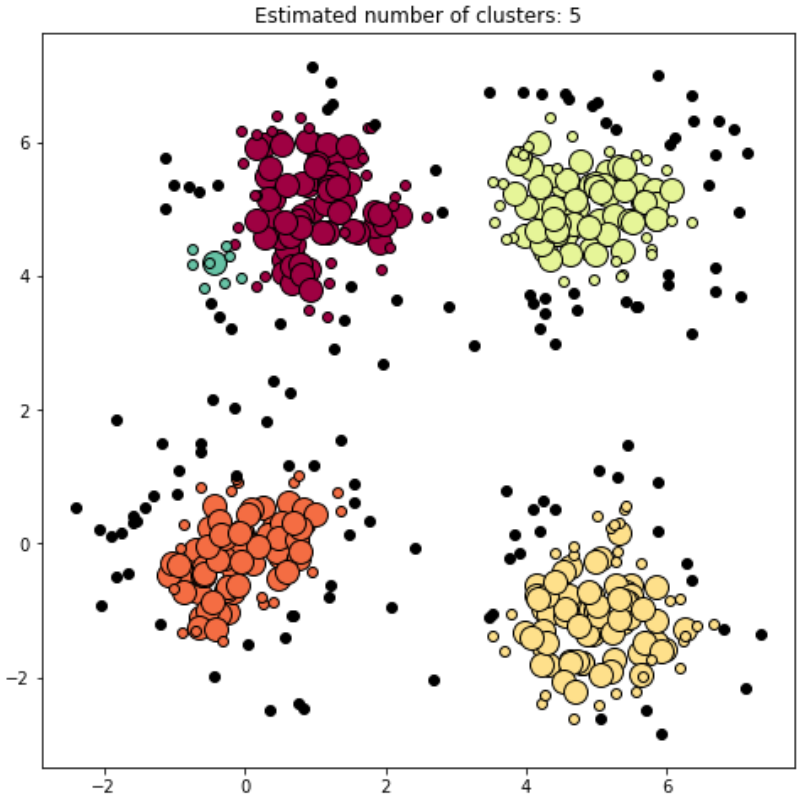
K-means vs DBSCAN

- While k-means is easy to understand, it does not take outliers into consideration
- All points are assigned to a cluster even if they do not belong to any
- Outliers assigned to clusters can lead to **misleading interpretations of collected data**
- **Density-based algorithm** locates regions of high density from regions of low density
- **What do we mean by density in this context?**
  - It is the **number of data points within a specified radius** from a random data point



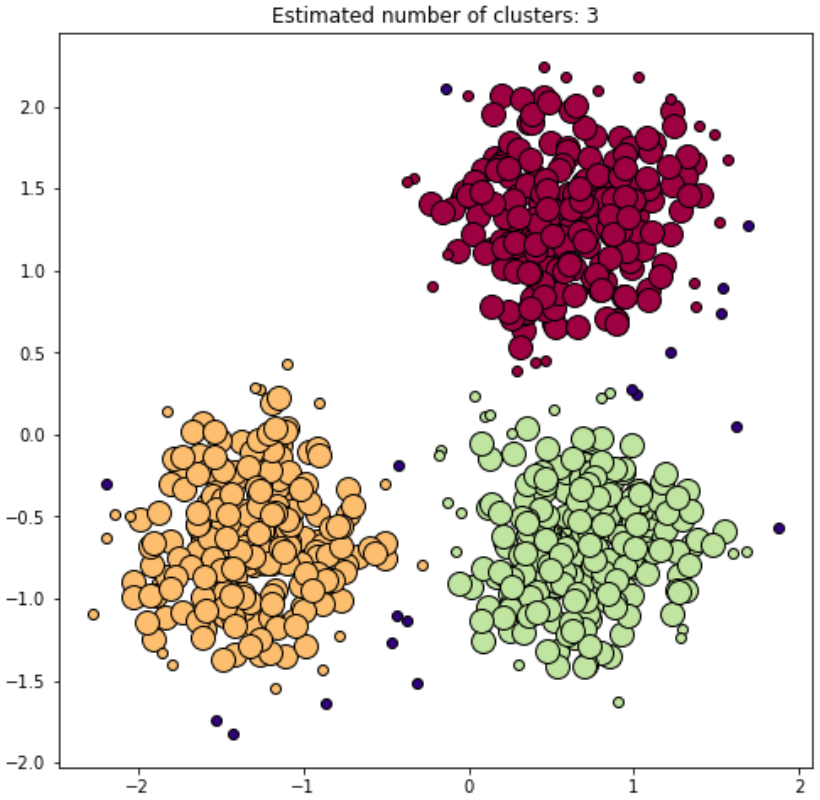
Why use DBSCAN?

- Unlike k-means, there is no requirement to set the number of clusters beforehand
- It can discover clusters of **arbitrary shape**
- It can **identify outliers** (points that are not part of any cluster)



DBSCAN

- Stands for **Density-Based Spatial Clustering of Applications with Noise**
- Popular density-based clustering algorithm
- Identifies **dense** clusters of points, hence it determines clusters of complex shapes
- Identifies outliers as points that are in **low-density regions**



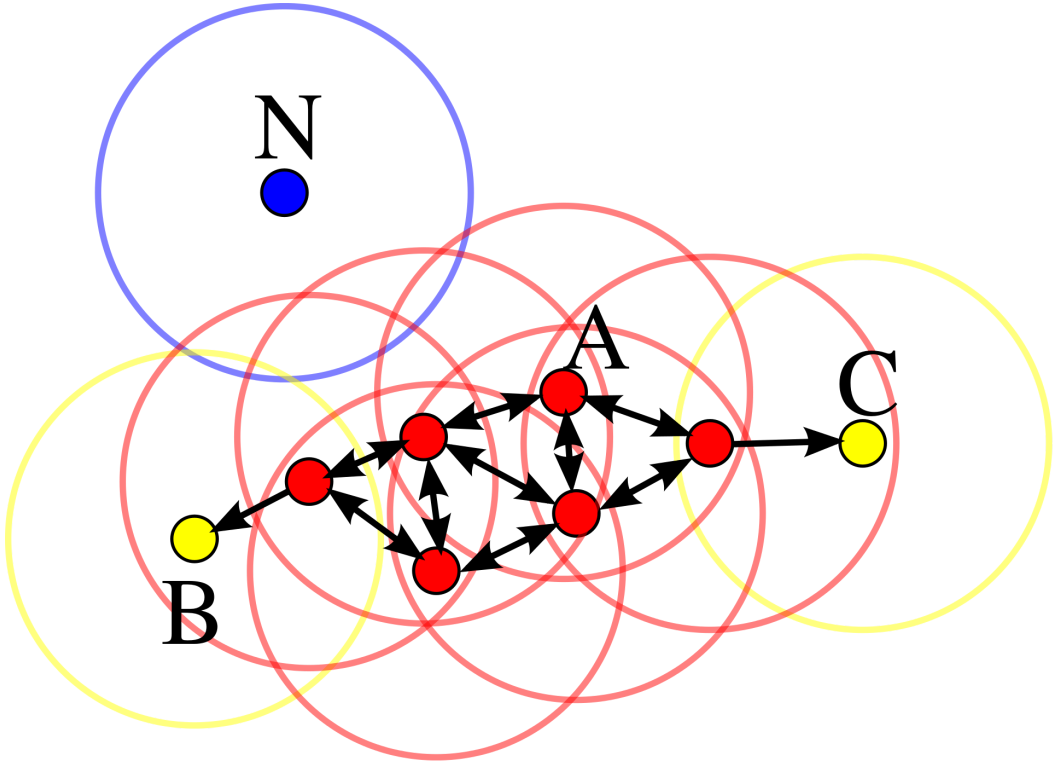
DBSCAN algorithm

- The algorithm can be summarized into the following steps:

1. Pick a random point
2. Compute its neighborhood to determine if it is a core point or an outlier
3. If it's a core point, expand the cluster by adding points directly in reach
4. Add all density-reachable points by jumping neighborhoods of the points assigned to the cluster
5. If an outlier is added to a cluster, reassign it as a border point
6. Repeat the same steps until all data points are assigned to a cluster or labeled an outlier

DBSCAN algorithm

- `MinPts` is set to `4` in the diagram
- The red points are **core points which form a single cluster** as they are all **reachable from one another**
- Yellow points `B` and `C` are not core points, but are reachable from them
- Hence, they belong to the cluster
- Blue point `N` is an outlier as it is neither a core point, nor reachable from them

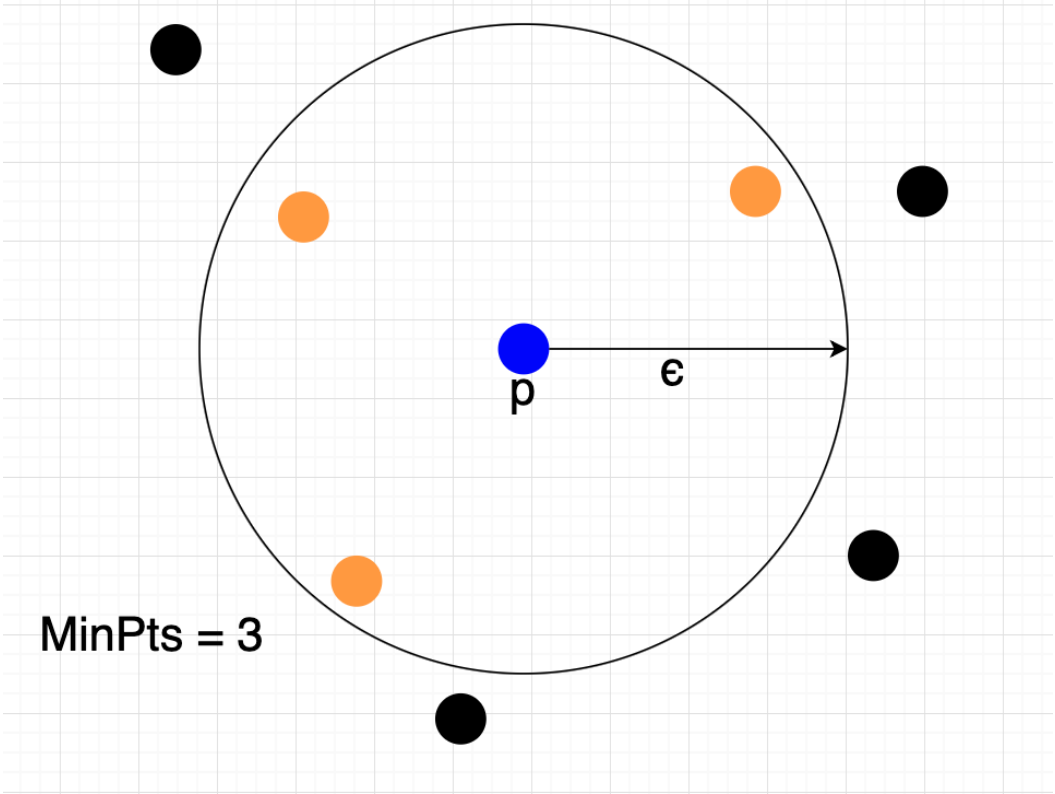


Module completion checklist

Objective	Complete
Define DBSCAN and how it compares to k-means	✔
Parameter estimation for ϵ and min_samples	
Run and visualize DBSCAN for an arbitrary distance	
Optimize parameters of DBSCAN and discuss pitfalls	
Recognize high dimensional data and the challenges that it introduces	
Illustrate the concept of PCA	
Transform data to run PCA on CMP dataset and apply the PCA algorithm	
Choose the optimal number of dimensions to retain and discuss common pitfalls of PCA	

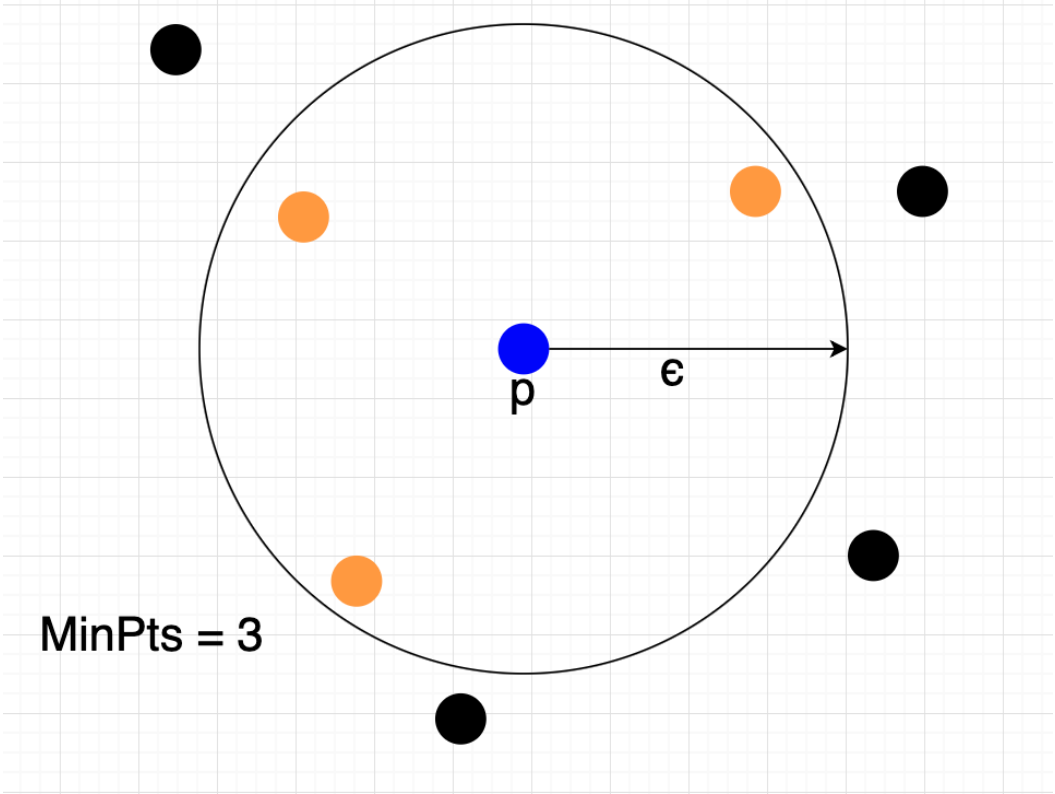
Parameters

- **There are 2 parameters in DBSCAN:**
- ϵ:
  - the **maximum distance between two points** for them to be in the **same neighborhood**
  - it can also be defined as the radius of our neighborhood around a data point p
  - if the distance between a data point and p is this value or lower, then that point is a neighbor of p



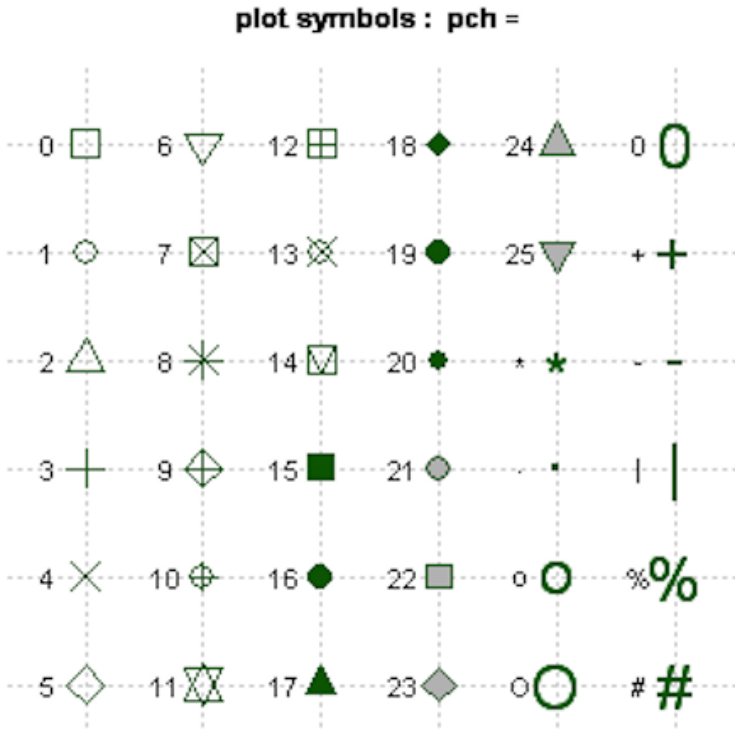
Parameters

- `MinPts`:
  - the **minimum number of data points** required to form a dense region
  - if we set `MinPts` as 10, then we need 10 points in the neighborhood region to form a dense region



Assigning data points

- Based on ϵ, the data points are assigned to three categories:
- **Core point** `p`:
  - Point `p` is a core point if at least `minPts` points are within distance ϵ of it
- **Border point** `q`:
  - Does not have `minPts` points within distance ϵ of it
  - Still part of the cluster as it's reachable from a core point `p`,
- **Outlier** `o`:
  - Neither a core point or a border point
  - The **"other"** class



Parameter tuning

- ϵ:
  - Setting ϵ to be very small will mean that a **large part of data will not be clustered**
  - Setting ϵ to be very large will result in **all the points forming a single cluster**
  - We can use a **k-distance graph** to find the optimal value for ϵ
- `MinPts`:
  - Can be derived from a **number of dimensions (D)** in the dataset, such as

$$MinPts \geq D + 1$$

  - Larger datasets should generally have larger `MinPts` value

Two variable dataset

- We will now walk through DBSCAN using a sample dataset with our CMP dataset
- We will subset two variables and run clustering on that:
  - `BiologicalMaterial01`
  - `BiologicalMaterial10`

Prepare data for DBSCAN

For unsupervised clustering, we need to:

- Prepare:
  - Remove the labels so that we can use unsupervised learning
- Clean:
  - Address **NAs**, missing values
  - Confirm or convert variable types to numeric
  - Scale data

Load the dataset

- We will now load the dataset and save it as `CMP`

```
CMP = pd.read_csv("ChemicalManufacturingProcess.csv")
```

```
print(CMP.head())
```

```
   Yield  BiologicalMaterial01  ...  ManufacturingProcess44  ManufacturingProcess45
0  38.00                  6.25  ...                     1.8                     2.4
1  42.44                  8.01  ...                     1.9                     2.2
2  42.03                  8.01  ...                     1.8                     2.3
3  41.42                  8.01  ...                     1.8                     2.1
4  42.49                  7.47  ...                     1.7                     2.1

[5 rows x 58 columns]
```

Subset the data

- We will subset `BiologicalMaterial01` and `BiologicalMaterial01` to `CMP_subset`

```
# Subset two variables from CMP dataset
CMP_subset = CMP[['BiologicalMaterial01', 'BiologicalMaterial10']]
CMP_subset.head()
```

```
   BiologicalMaterial01  BiologicalMaterial10
0                  6.25                  3.46
1                  8.01                  3.46
2                  8.01                  3.46
3                  8.01                  3.46
4                  7.47                  3.05
```

Data cleaning: NAs

- First, we check how many NAs there are in each column

```
print(CMP_subset.isnull().sum())
```

```
BiologicalMaterial01    0
BiologicalMaterial10    0
dtype: int64
```

Numeric variables

- Let's check if our variables are numeric

```
# Check data type of our variables.
print(CMP_subset.dtypes)
```

```
BiologicalMaterial01    float64
BiologicalMaterial10    float64
dtype: object
```

StandardScalar

- Why do we scale our data?
  - In clustering, the math behind the algorithm involves **calculating distance**
  - Distance calculations are **very sensitive to scale**
  - **If we do not scale** our variables to all be on the same numeric scale, our **distances will be relative to numbers** instead of to each other

StandardScalar

- Let's instantiate the scaler and transform our `CMP_subset` dataset
- We create a new object `CMP_subset_scaled`

```
# Instantiate MinMaxScaler.
scaler = StandardScaler()

# Scale the dataframe.
CMP_subset_scaled = scaler.fit_transform(CMP_subset)
```

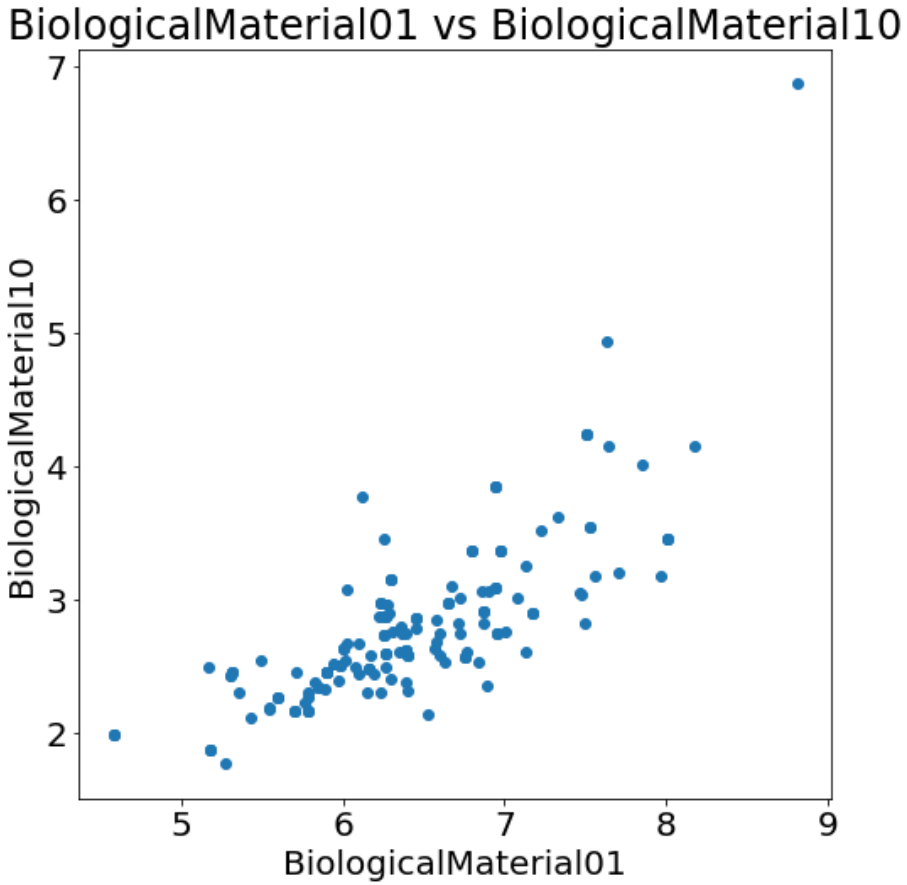
```
# Convert back to dataframe, making sure to name the columns again.
CMP_subset_scaled = pd.DataFrame(CMP_subset, columns = CMP_subset.columns)
print(CMP_subset.head())
```

```
   BiologicalMaterial01  BiologicalMaterial10
0                  6.25                  3.46
1                  8.01                  3.46
2                  8.01                  3.46
3                  8.01                  3.46
4                  7.47                  3.05
```

Visualize data

- Let us visualize the data
- Let's plot these two variables just to see what the interaction looks like

```
plt.scatter(CMP_subset_scaled['BiologicalMaterial01'],
 CMP_subset_scaled['BiologicalMaterial10'],
label='True Position')
plt.title('BiologicalMaterial01 vs
BiologicalMaterial10')
plt.xlabel('BiologicalMaterial01')
plt.ylabel('BiologicalMaterial10')
```



Knowledge check 1

Exercise 1

Module completion checklist

Objective	Complete
Define DBSCAN and how it compares to k-means	✔
Parameter estimation for ϵ and MinPts	✔
Run and visualize DBSCAN for an arbitrary distance	
Optimize parameters of DBSCAN and discuss pitfalls	
Recognize high dimensional data and the challenges that it introduces	
Illustrate the concept of PCA	
Transform data to run PCA on CMP dataset and apply the PCA algorithm	
Choose the optimal number of dimensions to retain and discuss common pitfalls of PCA	

sklearn.cluster.DBSCAN

```
DBSCAN(eps = ..,
       min_samples = ..,
       ..)
```

- The main parameters are:
  - `eps`: maximum distance between two samples for them to be in the same neighborhood
  - `min_samples`: number of samples in a neighborhood for a point to be considered as a core point

sklearn.cluster.DBSCAN

class sklearn.cluster. **DBSCAN** (*eps=0.5, min_samples=5, metric='euclidean', metric_params=None, algorithm='auto', leaf_size=30, p=None, n_jobs=None*) [source]

Perform DBSCAN clustering from vector array or distance matrix.

DBSCAN - Density-Based Spatial Clustering of Applications with Noise. Finds core samples of high density and expands clusters from them. Good for data which contains clusters of similar density.

Read more in the User Guide.

Parameters:

eps : *float, optional*
The maximum distance between two samples for them to be considered as in the same neighborhood.

min_samples : *int, optional*
The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself.

metric : *string, or callable*
The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by `sklearn.metrics.pairwise_distances` for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square. X may be a sparse matrix, in which case only "nonzero" elements may be considered neighbors for DBSCAN.

New in version 0.17: metric *precomputed* to accept precomputed sparse matrix.

metric_params : *dict, optional*
Additional keyword arguments for the metric function.

New in version 0.19

DBSCAN: model

- We now run the model on `CMP_cluster_scaled`
- We can tweak the parameters ϵ and `MinPts` to get a plot which makes sense
- For now, we set ϵ to have a radius of `0.4`
- We set `MinPts` (`min_samples` in the function) to `3`
- That means that we want 3 samples in a neighborhood with a radius of 0.4

```
# Let's run DBSCAN.
dbscan = DBSCAN(eps=0.4, min_samples = 3)
clusters = dbscan.fit_predict(CMP_subset_scaled)
```

DBSCAN: plot

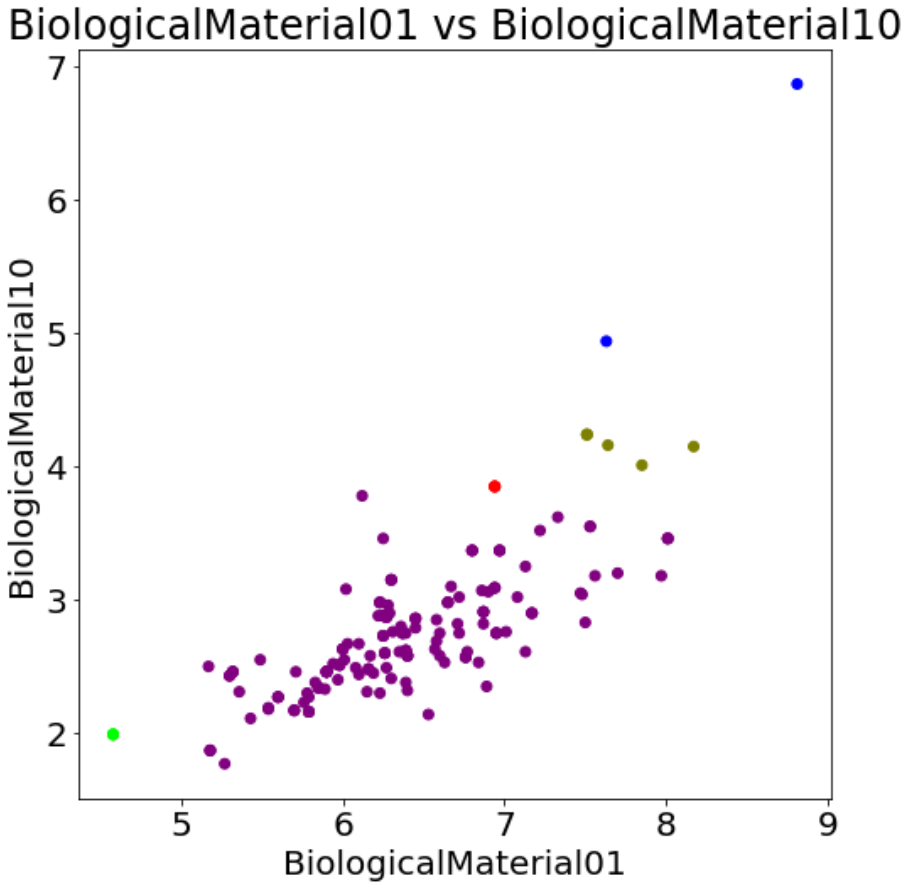
```
# Plot the cluster assignments.
plt.scatter(CMP_subset_scaled['BiologicalMaterial01'],
 CMP_subset_scaled['BiologicalMaterial10'], c =
clusters, cmap = "brg", label='True Position')

plt.title('BiologicalMaterial01 vs
BiologicalMaterial10')

plt.xlabel('BiologicalMaterial01')

plt.ylabel('BiologicalMaterial10')
```

- We can see two clusters (one big and one small) with a few outliers scattered around them



DBSCAN: optimizing eps

- We can find the optimal value for ϵ using a **k-distance graph**
- It consists of computing the average distances of every point to its k-nearest neighbors
- k-value refers to `MinPts`, which will be specified by the user
- The optimal `eps` parameter will be determined by the "knee" of the plot

DBSCAN: optimizing eps

- We need to define two functions to plot the k-distance graph
- `Get_distance_mean` is used to calculate the mean of the neighbor's distance

```
# Let's define the parameters needed for the functions.
n_trainingData = 0          # amount of training data evaluated using KNNdistPlot
labels = 0                  # array containing the labels of every point on the plane
n_clusters = 0              # number of clusters


def Get_distanceMean(points,minPts,previous_distanceMean):
# points: list containing the training points you want to use
# minPts: minimum number of points to be considered a cluster
# previous_distanceMean: the previous mean of the distances
# return: average distance between the points

        if (minPts < len(points)):
            nbrs = NearestNeighbors(n_neighbors=minPts).fit(points)
            distances, indices = nbrs.kneighbors(points)
            d_mean = distances.mean()
            return d_mean
        else:
            return previous_distanceMean
```

DBSCAN: k-distance graph

- `KNNdist_plot` plots the k-NN distance with respect to the amount of data and returns the optimal knee point `eps` value

```
def KNNdist_plot(points,minPts):
        epsPlot = []
        current_distanceMean = previous_distanceMean = 0
        knee_value = knee_found = 0

        for i in range (0,len(points),5):
            current_distanceMean = Get_distanceMean(points[i:],minPts,previous_distanceMean)
            df = current_distanceMean - previous_distanceMean

            if ((df > 0.02) & (i > 1) & (knee_found == 0)):
                knee_value = current_distanceMean
                knee_found = 1
                n_trainingData = i

            epsPlot.append( [i,current_distanceMean] )
            previous_distanceMean = current_distanceMean

        # Plot the kNNdistPlot.
        for i in range(0, len(epsPlot)):
                    plt.scatter(epsPlot[i][0],epsPlot[i][1],c='r',s=3,marker='o')
        plt.axhline(y=knee_value, color='g', linestyle='-')
        plt.axvline(x=n_trainingData , color='g', linestyle='-')
        plt.title("kNN distance")
        plt.show()
        print("Knee value: x=" + str(n_trainingData) + " , y=" + str(knee_value))
        print("Optimal eps value:" + str(knee_value))
```

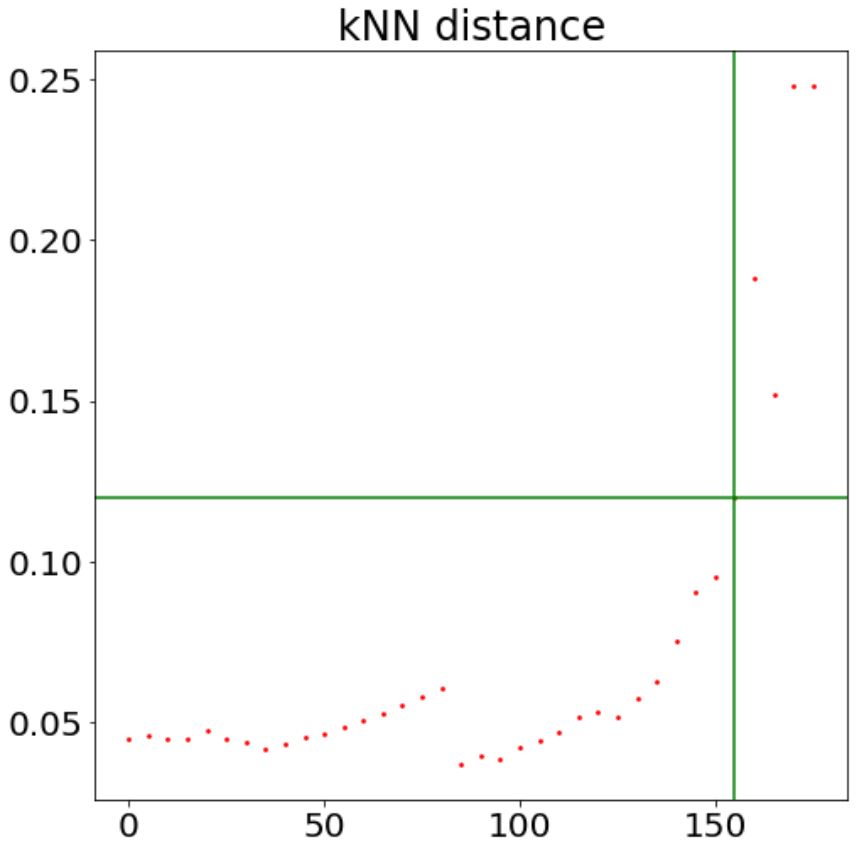
DBSCAN: k-distance graph

- Let's plot our k-distance graph for `CMP_subset_scaled`

```
KNNdist_plot(CMP_subset_scaled, 3)
```

- The k-distances are plotted in an ascending order
- We can see that the optimal `eps` value should be around `0.1198`

```
Knee value: x=155 , y=0.11982332963547687
Optimal eps value:0.11982332963547687
```



DBSCAN: optimized model

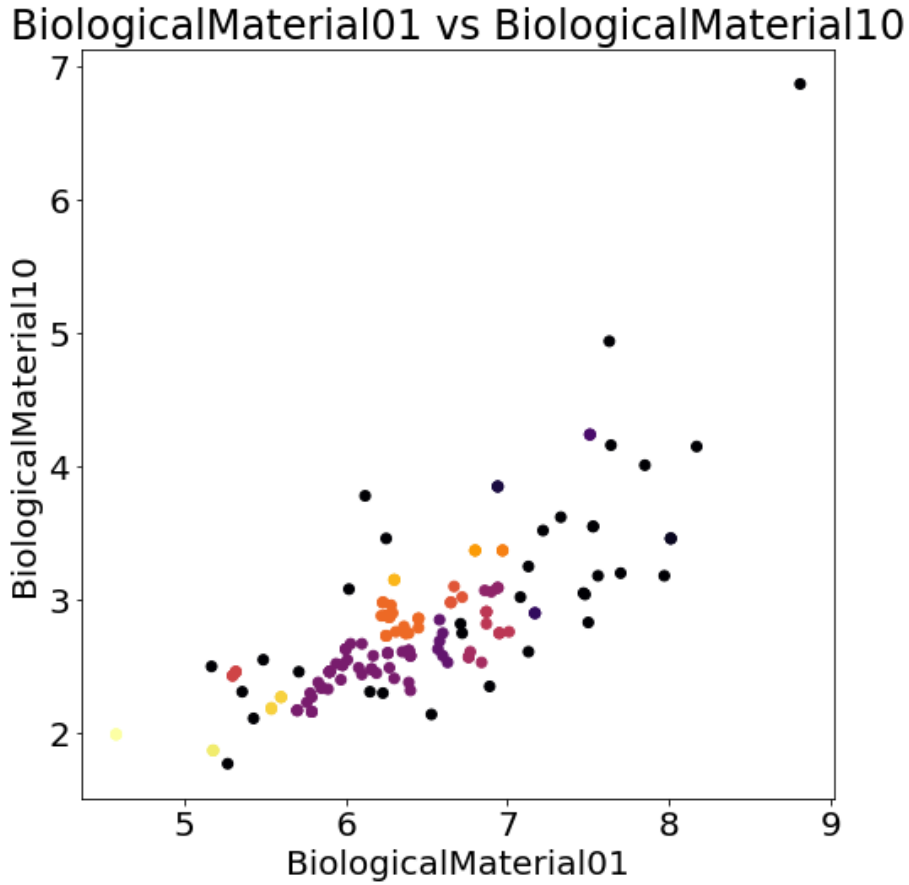
- Rerun the model with optimized `eps` value
- You can also set different `MinPts` value based on your domain knowledge

```
# DBSCAN
dbscan = DBSCAN(eps=0.1198, min_samples = 3)
optimized_clusters = dbscan.fit_predict(CMP_subset_scaled)
```

DBSCAN: plot optimized scatterplot

```
# Plot the cluster assignments.
plt.scatter(CMP_subset_scaled['BiologicalMaterial01'],
 CMP_subset_scaled['BiologicalMaterial10'], c =
optimized_clusters, cmap="inferno", label='True
Position')
plt.title('BiologicalMaterial01 vs
BiologicalMaterial10')
plt.xlabel('BiologicalMaterial01')
plt.ylabel('BiologicalMaterial10')
```

- We can see that more clusters formed with the decrease in `eps`



DBSCAN: summary

Good:

- Supports outlier detection
- Not sensitive to noise
- Does not need number of clusters specified beforehand
- Can detect arbitrary cluster shapes

Bad:

- Data has to be well understood to set clustering parameters ϵ and `MinPts`
- Cannot produce accurate results for varied densities or if density is too sparse
- Computationally intensive to find the neighborhood of each point
- Does not work well for high dimensional data
- Slower than other clustering methods in `scikit-learn`

Knowledge check 2

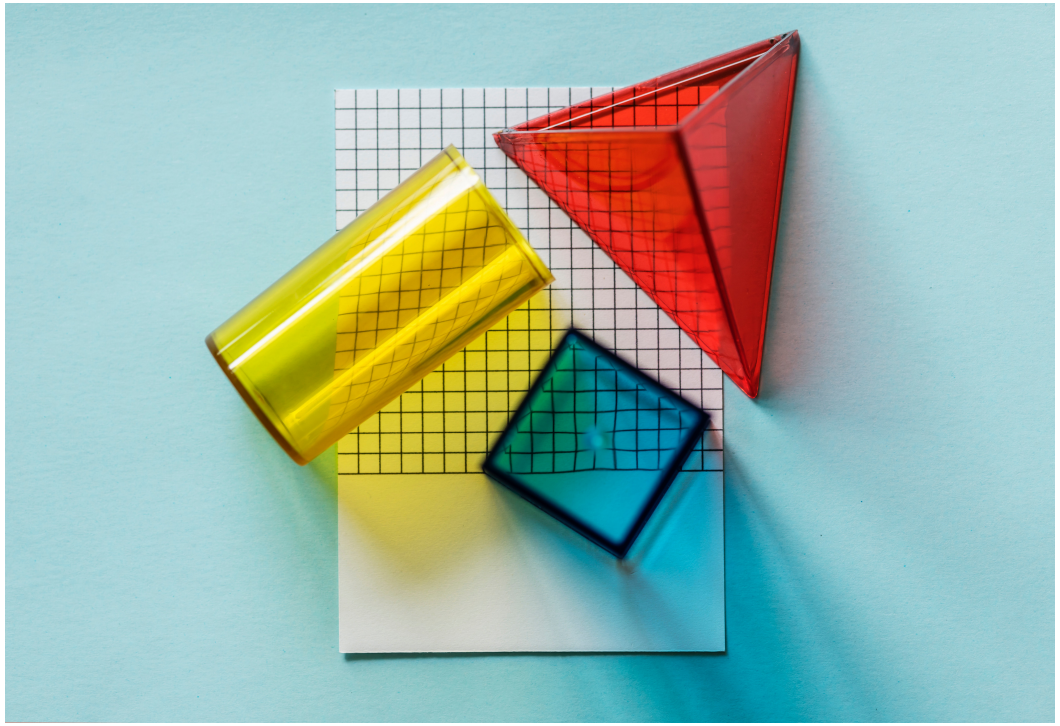
Exercise 2

Module completion checklist

Objective	Complete
Define DBSCAN and how it compares to k-means	✔
Parameter estimation for ϵ and MinPts	✔
Run and visualize DBSCAN for an arbitrary distance	✔
Optimize parameters of DBSCAN and discuss pitfalls	✔
Recognize high dimensional data and the challenges that it introduces	
Illustrate the concept of PCA	
Transform data to run PCA on CMP dataset and apply the PCA algorithm	
Choose the optimal number of dimensions to retain and discuss common pitfalls of PCA	

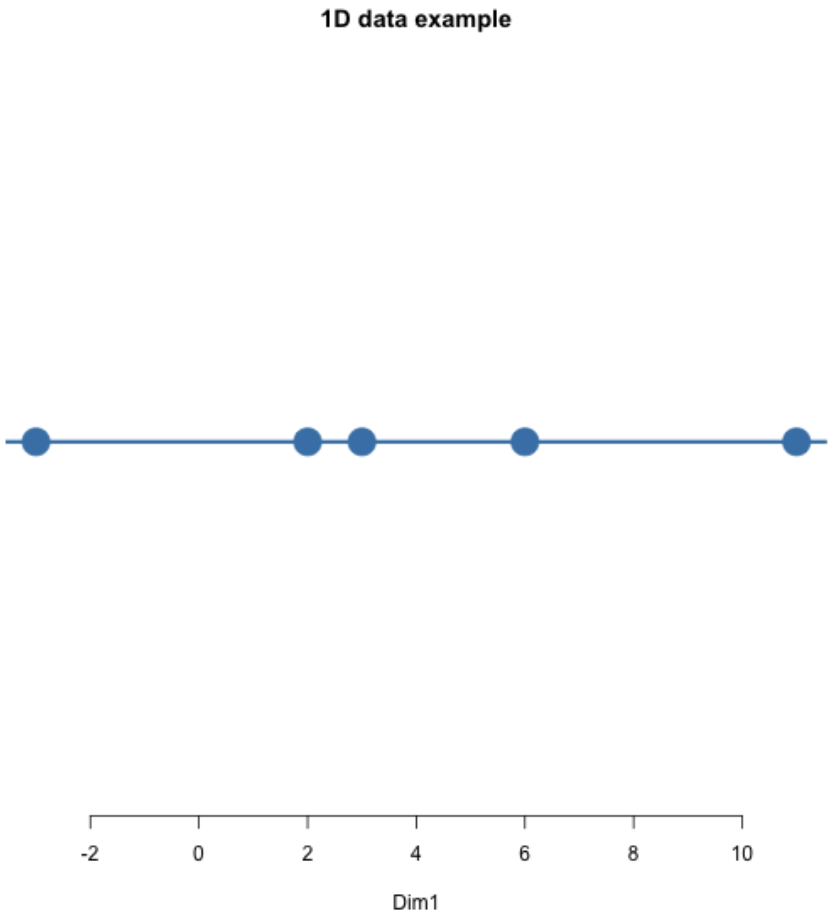
Dimensions of data

- Dimensions in statistics refers to **how many attributes a dataset has**
- **High dimensional data** means that the number of dimensions are staggeringly high, which makes calculations and predictions difficult
- We **reduce dimensionality** to **simplify the understanding of data**, either numerically or visually
- We will do so using **Principle Component Analysis (PCA)**



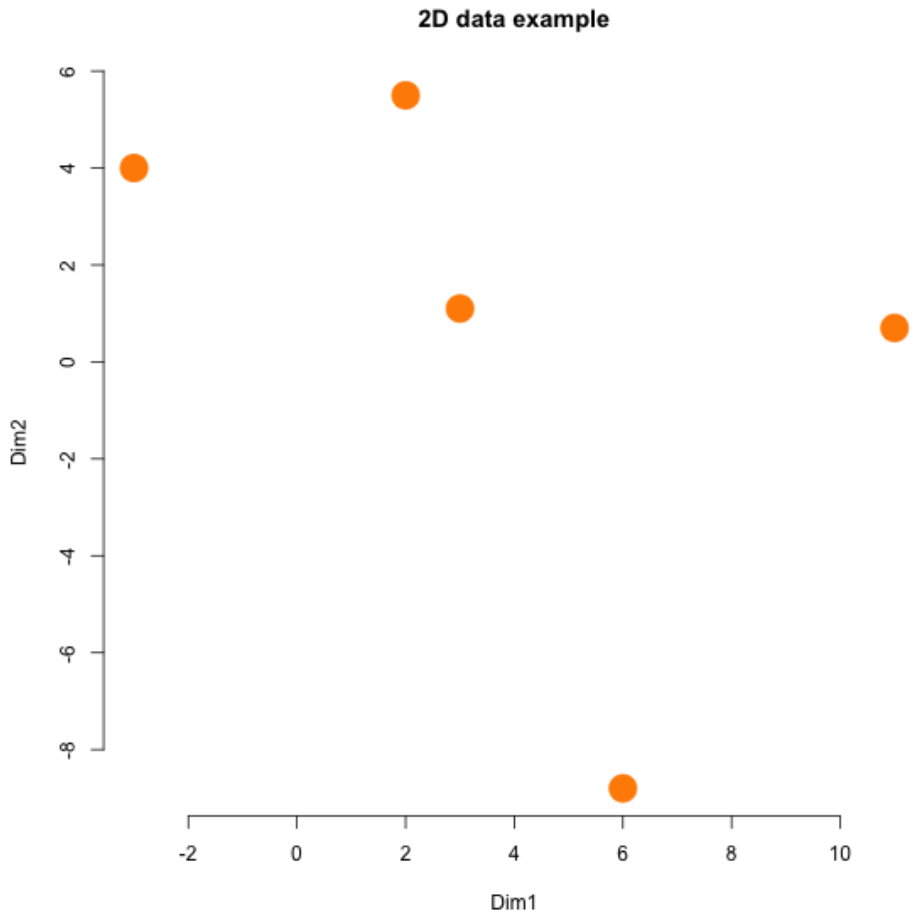
Data in dimensions

```
[1] -3  2  3  6 11
```



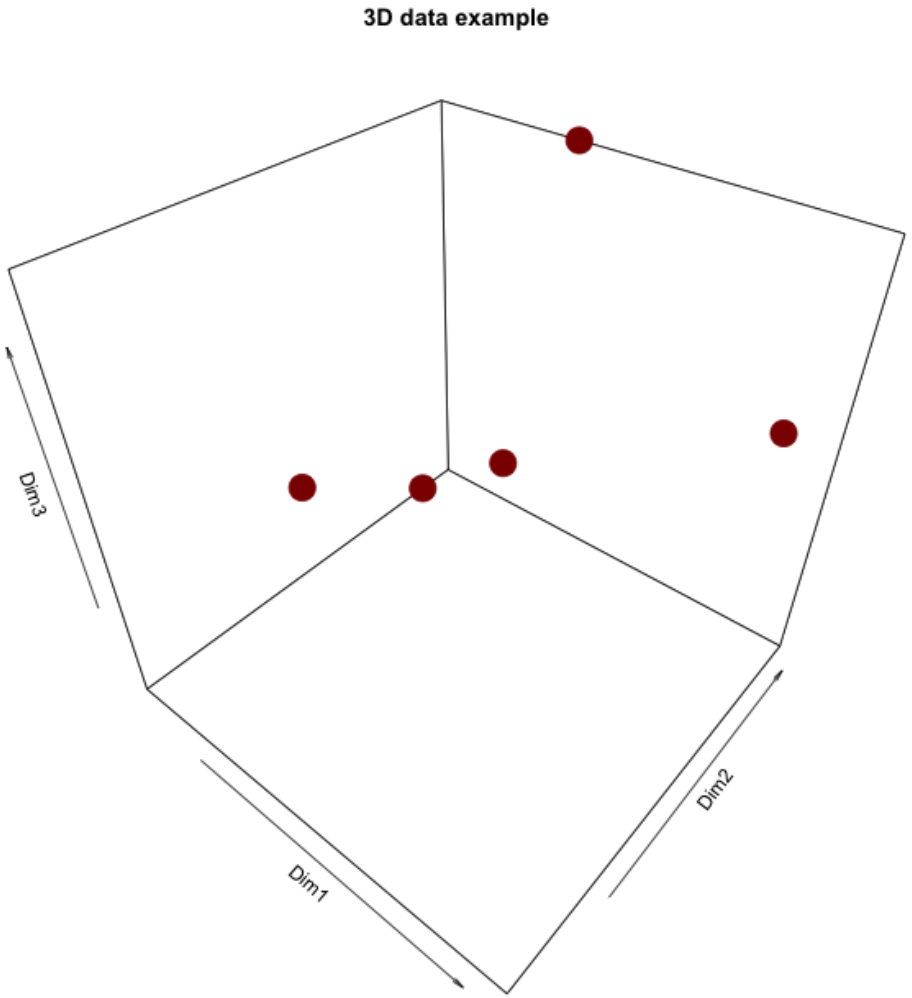
Data in two dimensions: a matrix with 2 columns

```
     Dim1 Dim2
[1,]   -3  4.0
[2,]    2  5.5
[3,]    3  1.1
[4,]    6 -8.8
[5,]   11  0.7
```



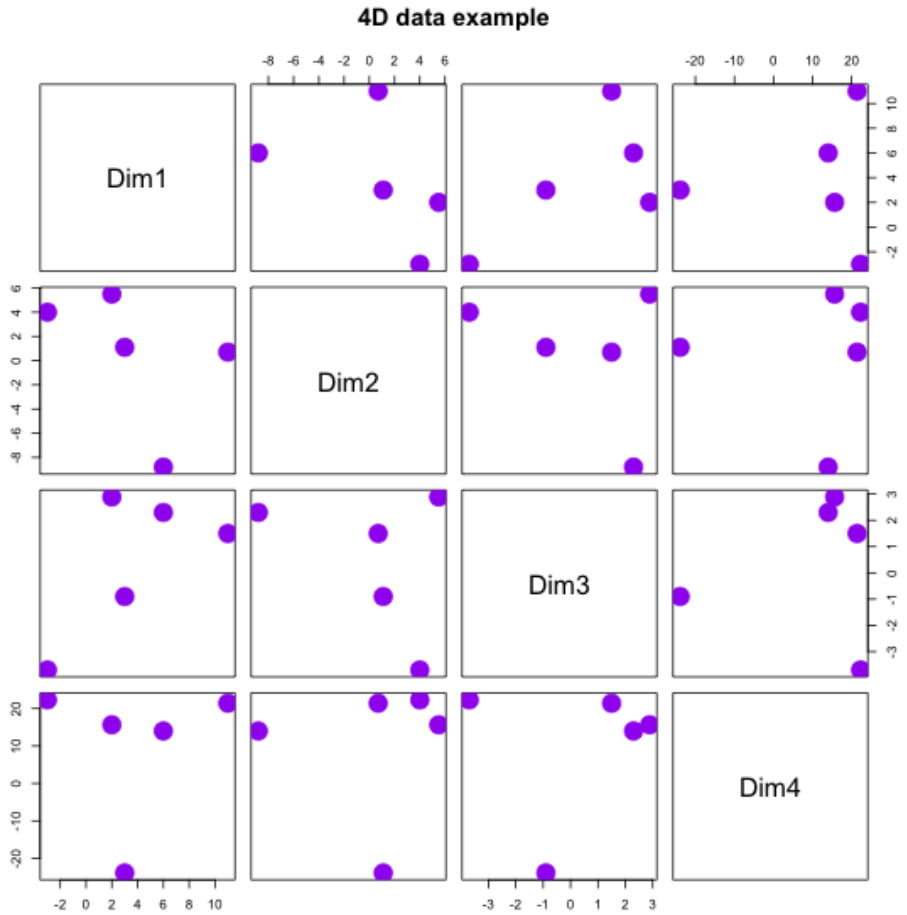
Data in three dimensions: a matrix with 3 columns

```
      Dim1 Dim2  Dim3
[1,]   -3  4.0 -3.70
[2,]    2  5.5  2.89
[3,]    3  1.1 -0.90
[4,]    6 -8.8  2.30
[5,]   11  0.7  1.50
```



Data in four+ dimensions: matrix with 4+ columns

```
      Dim1 Dim2  Dim3   Dim4
[1,]   -3  4.0 -3.70  22.30
[2,]    2  5.5  2.89  15.65
[3,]    3  1.1 -0.90 -23.90
[4,]    6 -8.8  2.30  14.00
[5,]   11  0.7  1.50  21.40
```



High dimensional data

- Most datasets contain **tens, hundreds, or thousands of variables**
- Anything beyond three dimensions is **hard to visualize** because *human perception is limited to 3 dimensions!*
- High-dimensional geometry is **counter-intuitive**

High-dimensional data: the curse

- **Computation**
  - **Computation time** needed may **increase exponentially** depending on the algorithm
- **Distance calculations**
  - More data points lead to **more distance between the observations**
  - Rendering distance calculations for high-dimensional data becomes not so effective*
- **Storage**
  - **Size of the data storage** increases with an increase in dimensions
  - e.g. Database size, file system, etc

*Read more about the strange geometry of high-dimensional spaces here: ***https://shapeofdata.wordpress.com/2013/04/02/the-curse-of-dimensionality/***

Module completion checklist

Objective	Complete
Define DBSCAN and how it compares to k-means	✔
Parameter estimation for ϵ and MinPts	✔
Run and visualize DBSCAN for an arbitrary distance	✔
Optimize parameters of DBSCAN and discuss pitfalls	✔
Recognize high dimensional data and the challenges that it introduces	✔
Illustrate the concept of PCA	
Transform data to run PCA on CMP dataset and apply the PCA algorithm	
Choose the optimal number of dimensions to retain and discuss common pitfalls of PCA	

Intro to Principle Component Analysis (PCA)

Think of a way to describe a product on a supermarket shelf - for example, you can give one *can of tomato soup* several characteristics:

1. Price
2. Brand name
3. Weight
4. Label size
5. Height of the can
6. Diameter of the bottom / top
7. Volume
8. Placement on shelf

A set of characteristics like these 8 is also known as **features**, **variables**, or **dimensions**. Yes, our humble soup can live in 8 dimensions (!), where the 8 characteristics give the can a unique **position** in that 8D space

Do we need all those dimensions?

Potential problems

1. Highly **correlated** and collinear **variables**: remember aliased and highly collinear variables?
2. **High dimensionality** that renders distance measures necessary for methods like `kmeans` are often inefficient and sometimes useless
3. **Overfitting**

Problems that PCA can solve

1. Creates new **uncorrelated variables** from the existing data
2. **Reduces the number of dimensions**
3. **Retains as much original information as we need** by allowing us to choose how many dimensions to keep

Removing linear combinations

Volume of a cylinder (which is the shape of our can) has the following formula:

$$V = \pi r^2 h$$

where r is the **radius of the bottom / top** of the can and h is the **height of the can**

We can safely discard the **volume** variable, since it's a linear combination of two variables: height (h) and diameter (D) of the bottom of the can

1. Price
2. Brand name
3. Weight
4. Label size
5. **Height of the can**
6. **Diameter of the bottom/top**
7. **Volume**
8. Position on the shelf

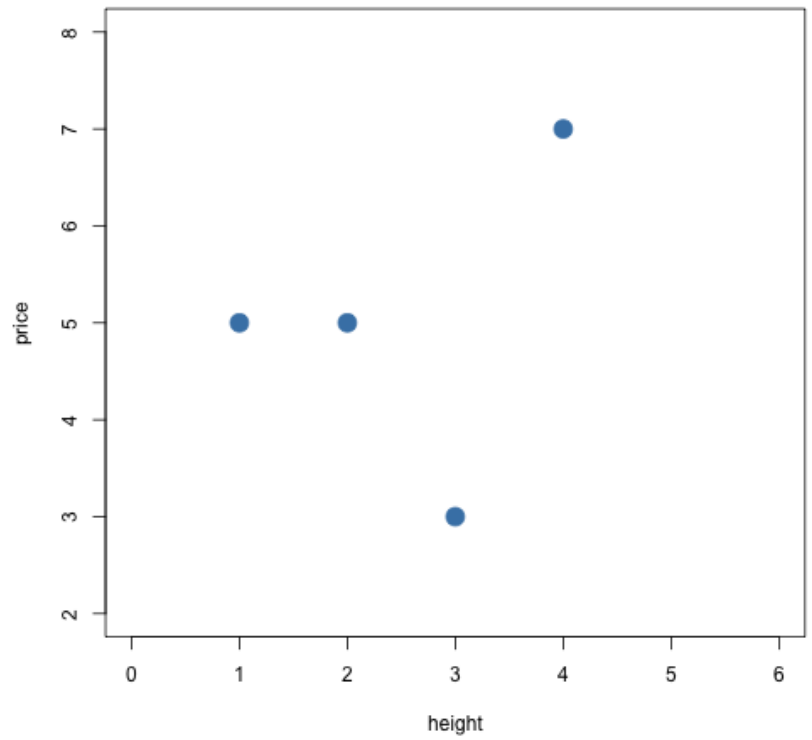
Addresses correlation

- Does the price of the soup depend on the *weight / height / diameter / position on the shelf*?
- It most likely does, but it's hard to quantify that deterministically
- One way to solve that problem is to compute **correlation** between the variables

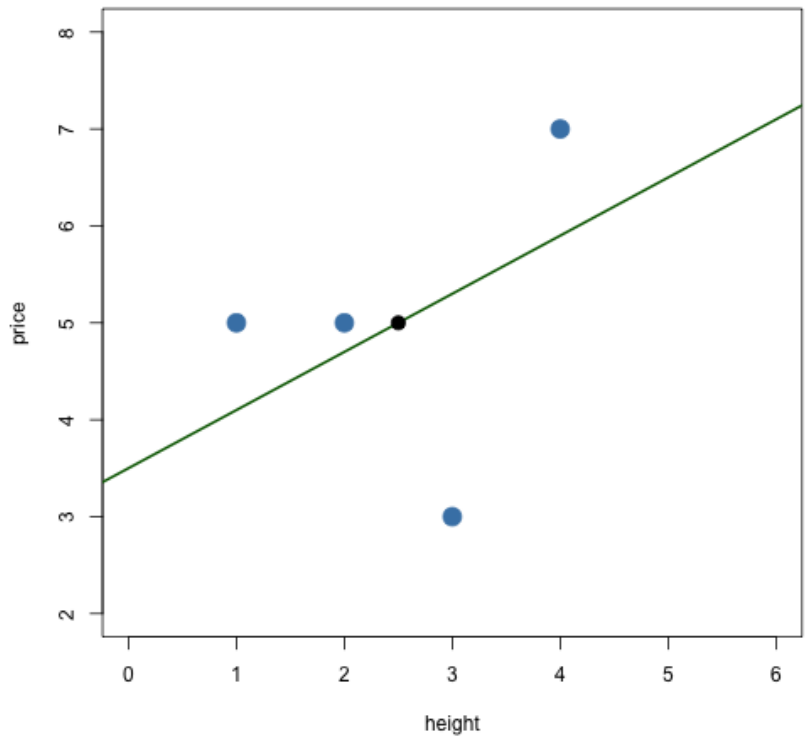
1. **Price**
2. Brand name
3. *Weight*
4. Label size
5. *Height of the can*
6. *Diameter of the bottom/top*
7. Volume
8. *Position on the shelf*

Plot points

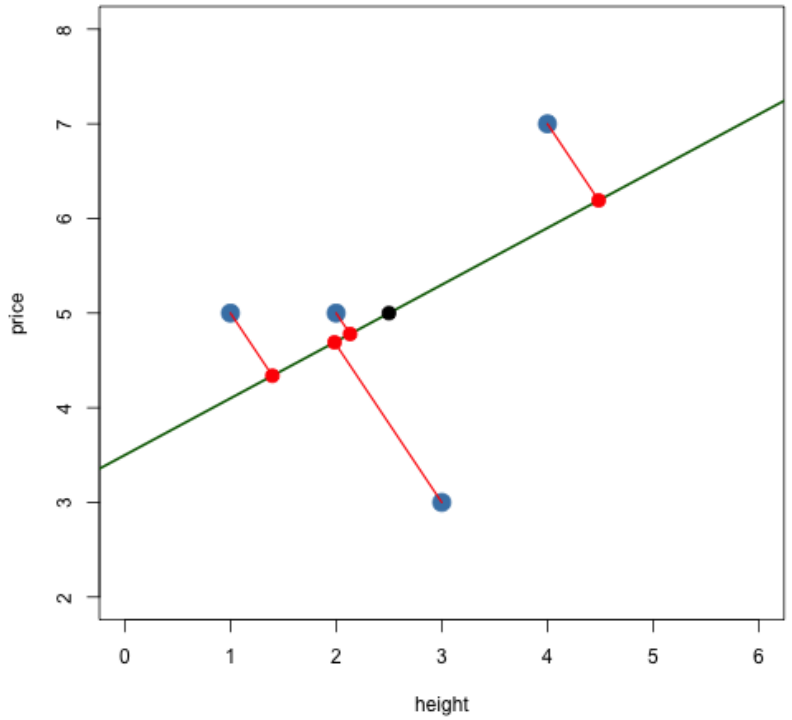
- This is how a sample dataset looks plotted on a 2D plane



- Draw an arbitrary line that passes through center of data

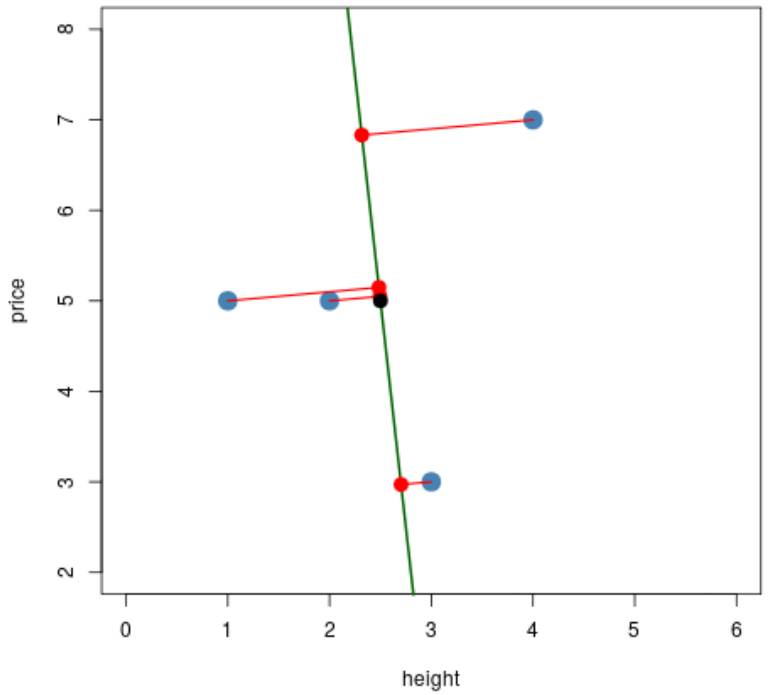


Projection



- We have produced a **projection** of 2D data onto 1D line
- The distance between the red points is the **amount of variation** in our data explained by this **projection**
- When we reduce the dimensionality of data, we want to have the red line segments as short as possible (i.e. **minimize the error**) and increase the distance between the red points (i.e. **maximize the amount of variation** in the original data explained by projection)

Error & variance



- Notice that the distance between the red points changes as the line rotates
- The length of the red segments is **decreasing** as the distance between the red dots is **increasing**
- The key feature of PCA is that it *guarantees* the **smallest error** *and* the **biggest variance** between the projections!

Properties of PCA

PCA creates a set of **new variables**. The number of **new features**, or ***principal components***, is equal to the number of original variables

Principal Components have a few important properties:

- **The 1st** principal component explains **the most variation** in data
- **The 2nd** principal component explains **most of the remaining variation**
- **The 3rd** principal component explains **most of the remaining variation not explained by the first two**, etc.

The **variance explained** by either of the principal components **does not overlap**

All principal components are **uncorrelated**, which means that if we compute a correlation matrix on principal components aside from the diagonal entries, it would be filled with zeroes!

Clarifying selection vs extraction

Feature selection: selecting a subset of variables from a set of original variables

- *Filter methods* (e.g. information gain, distance measures, correlation-based feature selection)
- *Wrapper methods* (e.g. sequential forward selection, sequential backward elimination, randomized hill climbing, genetic algorithms)
- *Embedded methods* (e.g. decision trees and random forests, weighted Naive Bayes, feature selection using weight vector of SVM)

Feature extraction: performing data transformation / factorization, generating new sets of variables and selecting from them

- Principal component analysis (PCA)
- Non-negative matrix factorization (NMF)
- Linear discriminant analysis (LDA)
- Autoencoder

PCA's uses go beyond soup

- Database management and optimization
- Search optimization
- Avoiding overfitting in classification
- Removing collinearity in regression
- Image storage and search

Knowledge check 3

Module completion checklist

Objective	Complete
Define DBSCAN and how it compares to k-means	✔
Parameter estimation for ϵ and MinPts	✔
Run and visualize DBSCAN for an arbitrary distance	✔
Optimize parameters of DBSCAN and discuss pitfalls	✔
Recognize high dimensional data and the challenges that it introduces	✔
Illustrate the concept of PCA	✔
Transform data to run PCA on CMP dataset and apply the PCA algorithm	
Choose the optimal number of dimensions to retain and discuss common pitfalls of PCA	

Prepare data for PCA

- Before performing PCA on any data, make sure to:
  - Check for **NAs** and impute data if necessary
  - Remove the **outcome variable**
  - **Scale data** (or use appropriate function argument)
- We will use our entire `CMP` dataset to understand PCA, so that it can select the right variables

Data cleaning: NAs

- First, we check if there are any NAs there in each column

```
print(CMP.isnull().sum())
```

- Our dataset contains NAs
- Impute NAs with mean of the column
- *Note that there are other ways to impute NAs, and it always depends on the use case and your knowledge of the data*

```
def ImputeNAwithMean(dataset):
    for i in range(0, dataset.shape[1]):
        CMP.iloc[:, i].fillna(CMP.iloc[:,
i].mean(), inplace = True)
```

```
ImputeNAwithMean(CMP)
```

Running PCA: outcome variable

- We remove the outcome variable `Yield` before we run PCA
- Let's split our predictors into `X` and the target variable into `y`

```
y = CMP[['Yield']]
X = CMP.drop(['Yield'], axis = 1)
```

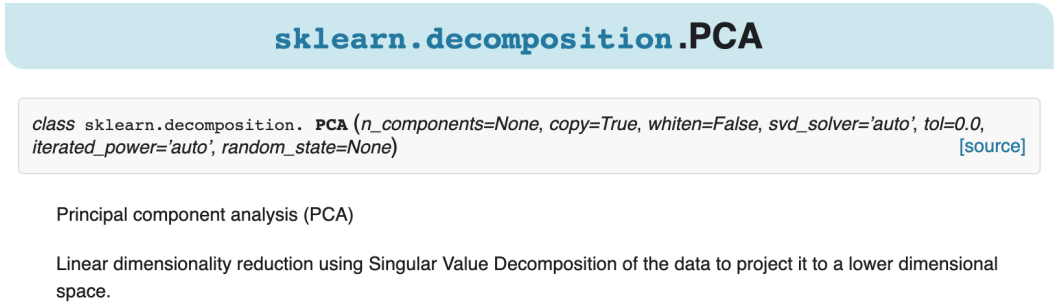
PCA: scaling our data

- Let's scale the predictors stored in `X`

```
sc = StandardScaler()
X = sc.fit_transform(X) # <- fits the data, then transforms it
```

sklearn PCA

- We simply import `PCA` from `sklearn` library
- It reduces dimensions using **Singular Value Decomposition**
- There are 3 common methods:
  - `fit`: fit the model with the data
  - `fit_transform`: fit the model with the data and apply dimensionality reduction on it
  - `transform`: apply dimensionality reduction on the data



sklearn.decomposition.PCA

class sklearn.decomposition. **PCA** (*n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', random_state=None*) [source]

Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

Running PCA

- We run the PCA on the predictors X

```
pca = PCA()
X_pca = pca.fit_transform(X)
```

```
print(X_pca)
```

```
[[ 1.8103035  -0.16068295 -2.55010525 ... -0.01909473 -0.01961089
   0.        ]
 [-6.30850362  0.14408354 -0.416486   ... -0.03131058  0.0000835
  -0.        ]
 [-6.33284024  1.46192552 -1.07694984 ...  0.02203391  0.00079282
   0.        ]
 ...
 [-1.97470716 -7.35401416  7.08545549 ... -0.02450514  0.01711104
  -0.        ]
 [-2.52387545 -7.62398094  7.04741544 ... -0.00764151 -0.02465447
   0.        ]
 [-2.86349627 -7.03039762  6.73177619 ...  0.02191039 -0.02866175
   0.        ]]
```

Module completion checklist

Objective	Complete
Define DBSCAN and how it compares to k-means	✔
Parameter estimation for ϵ and MinPts	✔
Run and visualize DBSCAN for an arbitrary distance	✔
Optimize parameters of DBSCAN and discuss pitfalls	✔
Recognize high dimensional data and the challenges that it introduces	✔
Illustrate the concept of PCA	✔
Transform data to run PCA on CMP dataset and apply the PCA algorithm	✔
Choose the optimal number of dimensions to retain and discuss common pitfalls of PCA	

PCA: explained variance

- The attribute `explained_variance_ratio_` computes the amount of variance explained by each component
- It captures the **entire variation** contained in original data in decreasing order **without overlapping**
- We can see that PC1 explains around `18%` of the variation, PC2 explains `9.8%` of the variation, and so on

```
# The amount of variance that each PC explains.
explained_variance =
pca.explained_variance_ratio_
print(explained_variance)
```

```
[0.18103851 0.09834738 0.09457927 0.07409538
0.0532337  0.04600286
 0.04347031 0.04148782 0.03658387 0.03346445
0.02865301 0.02527776
 0.02198857 0.02145875 0.02037568 0.01862916
0.01809444 0.01487389
 0.01347901 0.01166374 0.01113735 0.00986294
0.00922804 0.00912082
 0.00833892 0.00739484 0.00704992 0.0058547
0.0048006  0.0047724
 0.00343182 0.00283547 0.0026427  0.00241197
0.00195101 0.00183364
 0.00166188 0.001547   0.00129235 0.00120568
0.00090652 0.00074499
 0.00061195 0.0005471  0.00041759 0.00039766
0.0003221  0.00026719
 0.00023839 0.00016623 0.00008855 0.00004313
0.0000396  0.00002632
 0.00000865 0.00000239 0.        ]
```

Computing cumulative variance explained

- When we need to decide **how many dimensions to retain**, we would like to know how much variance a certain number of principal components explain **cumulatively**
- **Cumulative variance explained** can be computed by using the `cumsum` function on the explained variance ratio

```
cum_percent_var_explained = np.cumsum(np.round(pca.explained_variance_ratio_, decimals=4)*100)
print(cum_percent_var_explained)
```

```
[18.1  27.93 37.39 44.8  50.12 54.72 59.07 63.22 66.88 70.23 73.1  75.63
 77.83 79.98 82.02 83.88 85.69 87.18 88.53 89.7  90.81 91.8  92.72 93.63
 94.46 95.2  95.9  96.49 96.97 97.45 97.79 98.07 98.33 98.57 98.77 98.95
 99.12 99.27 99.4  99.52 99.61 99.68 99.74 99.79 99.83 99.87 99.9  99.93
 99.95 99.97 99.98 99.98 99.98 99.98 99.98 99.98 99.98]
```

- In this case, if we were to keep just the 1st principal component, we would have retained `18%` of variation in our data, if we decided to keep the first 2 principal components, that number would be around `28%`

PCA: choosing the dimensions to retain

- We can also choose the **number of principle components** to keep for our data
- We will be reducing the features to 10 principle components
- We can do so by specifying the number of components when instantiating the `PCA` function

```
pca_10 = PCA(n_components = 10)
X_10_pca = pca_10.fit_transform(X)

print(pca_10.explained_variance_ratio_)
```

```
[0.18103851 0.09834738 0.09457927 0.07409538 0.0532337  0.04600286
 0.04347031 0.04148782 0.03658387 0.03346445]
```

```
cum_percent_var_explained_10 = np.cumsum(np.round(pca_10.explained_variance_ratio_, decimals=4)*100)
print(cum_percent_var_explained_10)
```

```
[18.1  27.93 37.39 44.8  50.12 54.72 59.07 63.22 66.88 70.23]
```

- So 10 principle components can explain around `70%` of the variance from these features

PCA: choosing the dimensions to retain

How much variation should we retain in our data?

- Depends on the business problem!
- Select a few dimensions that explain most of the variation in the data
- Ideally, variance should be at least 60%

Use of PCA

- PCA can be further used for **prediction and classification purposes** based on the principal component scores
- We can use the principal components for **supervised learning methods** which were covered in earlier classes

PCA pitfalls and alternatives

PCA fails if:

- Data is not linearly correlated
- *Information* in data ≠ *variation*, sometimes components in data with small eigenvalues carry important information, when we discard them, we lose that information
- Strict independence of variables is of key importance, because *uncorrelated* ≠ *independent*!
- Data is not scaled (*easy to solve with scaling*)
- Lots of missing data (*data imputation may not always help!*)

Alternatives to PCA

- **Linear Discriminant Analysis (LDA)** - for linearly separable data
- **Quadratic Discriminant Analysis (QDA)** - for non-linearly separable data
- **Independent Component Analysis (ICA)**
- **Neural Networks (NNs)** are compatible to high dimensionality and many other pitfalls of other machine learning methods, they often perform well where other methods fail (*but they are not one size fits all either!*)

Knowledge check 4

Exercise 3

Module completion checklist

Objective	Complete
Define DBSCAN and how it compares to k-means	✔
Parameter estimation for ϵ and MinPts	✔
Run and visualize DBSCAN for an arbitrary distance	✔
Optimize parameters of DBSCAN and discuss pitfalls	✔
Recognize high dimensional data and the challenges that it introduces	✔
Illustrate the concept of PCA	✔
Transform data to run PCA on CMP dataset and apply the PCA algorithm	✔
Choose the optimal number of dimensions to retain and discuss common pitfalls of PCA	✔

Workshop!

- Workshops are to be completed in the afternoon either with a dataset for a capstone project or with another dataset of your choosing
- Make sure to annotate and comment your code so that it is easy for others to understand what you are doing
- This is an exploratory exercise to get you comfortable with the content we discussed today
- Today you will:
  - Load and prepare your dataset
  - Implement a DBSCAN clustering and find if your data has any outliers
  - Implement PCA to reduce dimensions in your data
  - Find the total explained variance of how much you are retaining from your dataset

Congratulations on completing this module!