Two Simple ML Algorithms [1]

Shaobo Li

University of Kansas

[1] Partially based on Hastie, et al. (2009) ESL, and James, et al. (2013) ISLR

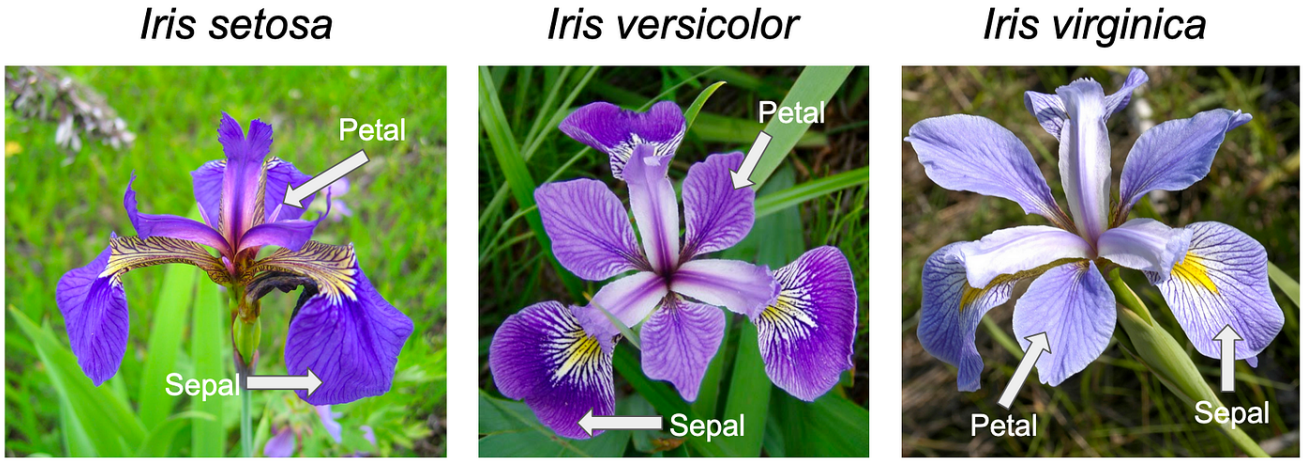
Outline

We will discuss two simple ML algorithms

- k-means clustering —- An unsupervised learning algorithm
- k-nearest neighbors —- A supervised learning algorithm

We use the famous Iris dataset to study these two methods

The famous Iris dataset



- 150 observations and four numeric flower measurements
- Target label: Species $= \{$setosa, versicolor, virginica$\}$
- **K-means:** ignore the Species label and discover groups from the measurements
- **KNN:** use the Species label to predict the species of a new flower

Clustering – an unsupervised learning method

- Goal: find subgroups among a sample of observations
  - Not based on any single variable (e.g. gender, race)
  - Based on all given variables

Clustering – an unsupervised learning method

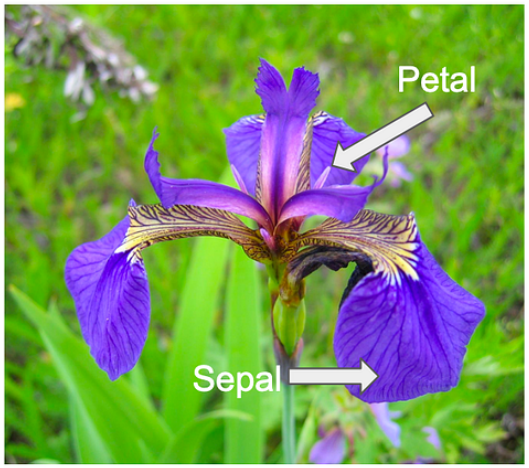
- Goal: find subgroups among a sample of observations
  - Not based on any single variable (e.g. gender, race)
  - Based on all given variables
- The number of subgroups is subjective

Clustering – an unsupervised learning method

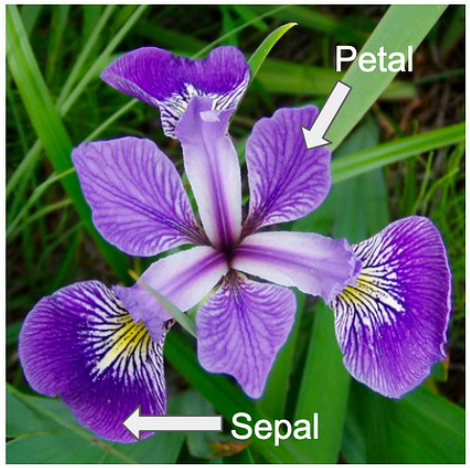
- Goal: find subgroups among a sample of observations
  - Not based on any single variable (e.g. gender, race)
  - Based on all given variables
- The number of subgroups is subjective
- Approaches:
  - K-means clustering
  - Hierarchical clustering
  - Model-based clustering

Iris data example

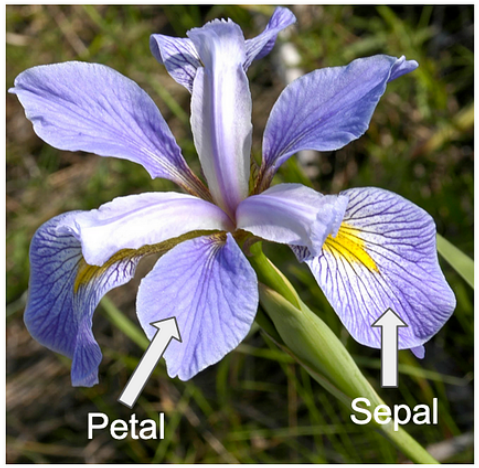
Iris setosa



Iris versicolor



Iris virginica



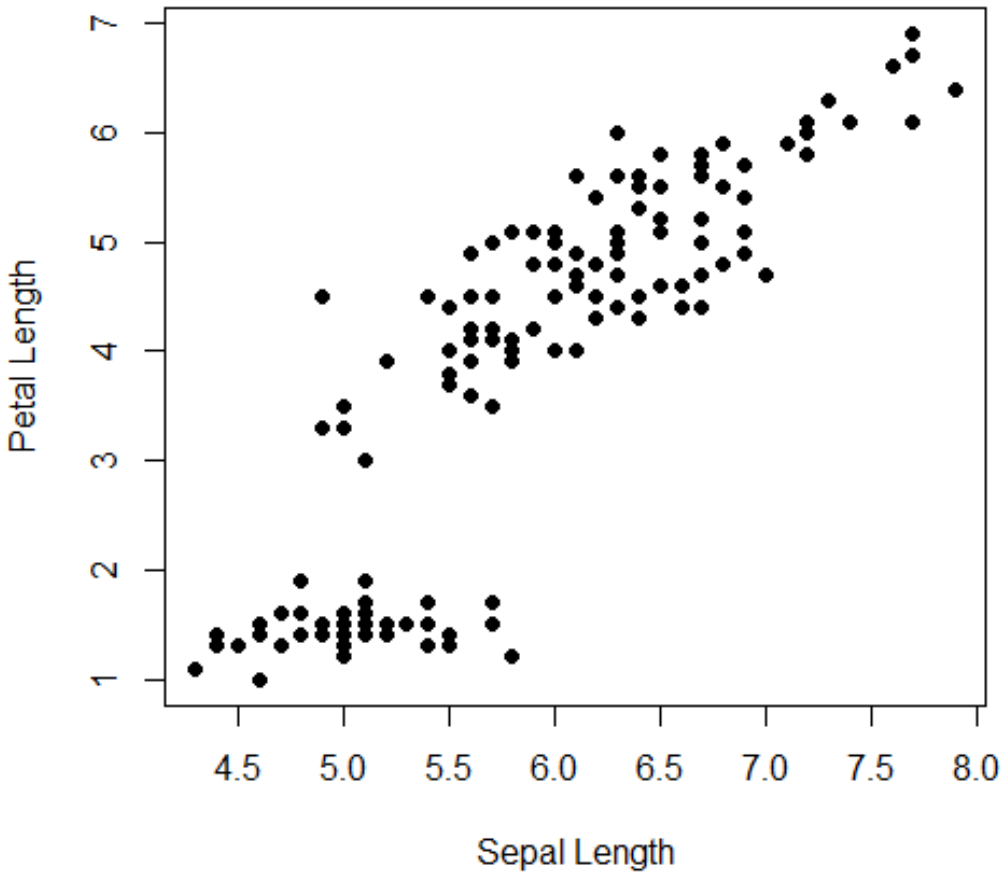
- Learning goal: group these 150 irises to multiple groups based on the 4 features
- We delete the Species column and pretend only the four features are observed

K-means

- K: number of groups/segments
  - This is subjective but a reasonable value can be determined by certain rules.
- Means: the center of each group
- The math: computing distance between a data point to the current group center
- Standardization: a key step before fitting the model
  - Why? To ensure different features are on the same scale
  - How? $\frac{X-\bar{X}}{sd(X)}$;
  - $\bar{X}$ is the mean of X and $sd(X)$ is standard deviation of X

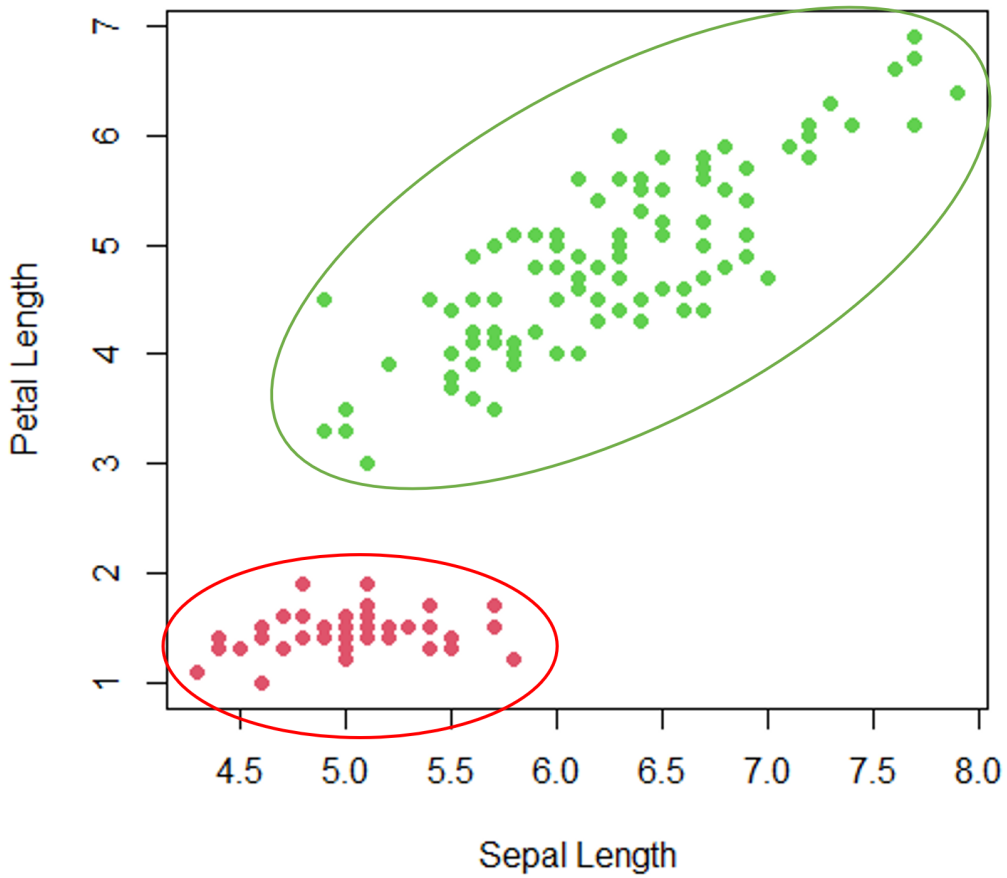
Iris data example

To illustrate, we only consider two features: petal and sepal length



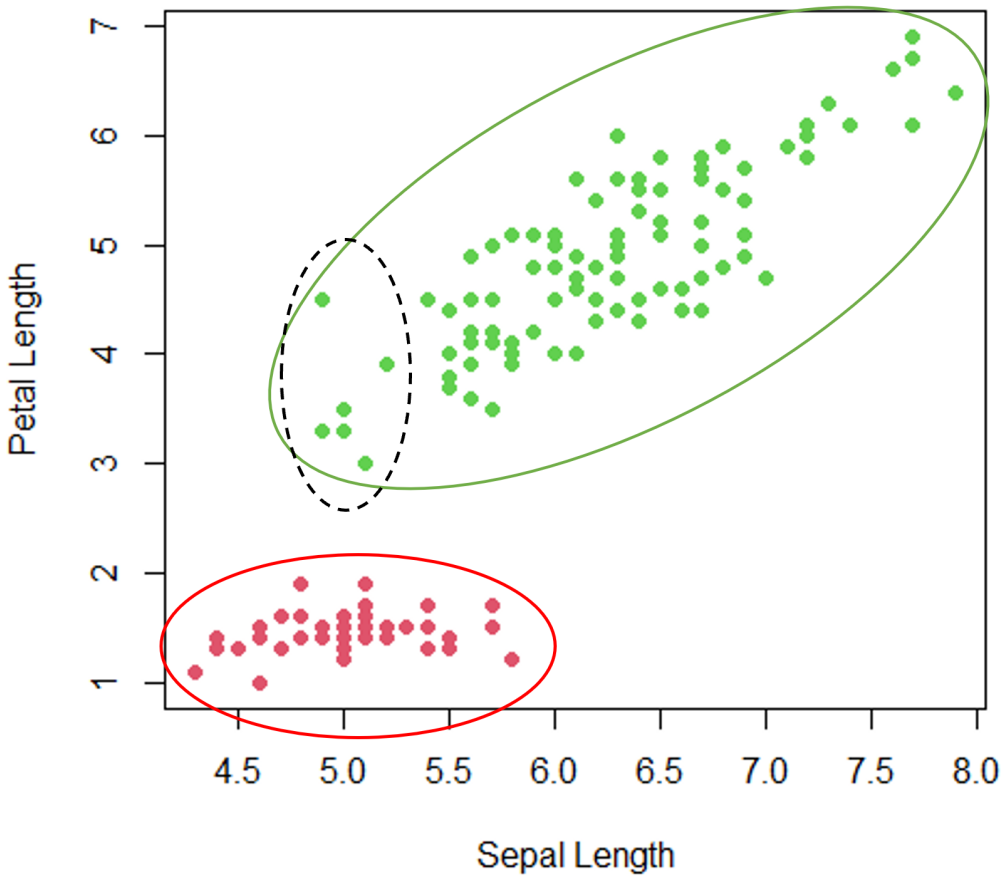
Iris data example

To illustrate, we only consider two features: petal and sepal length



Iris data example

To illustrate, we only consider two features: petal and sepal length

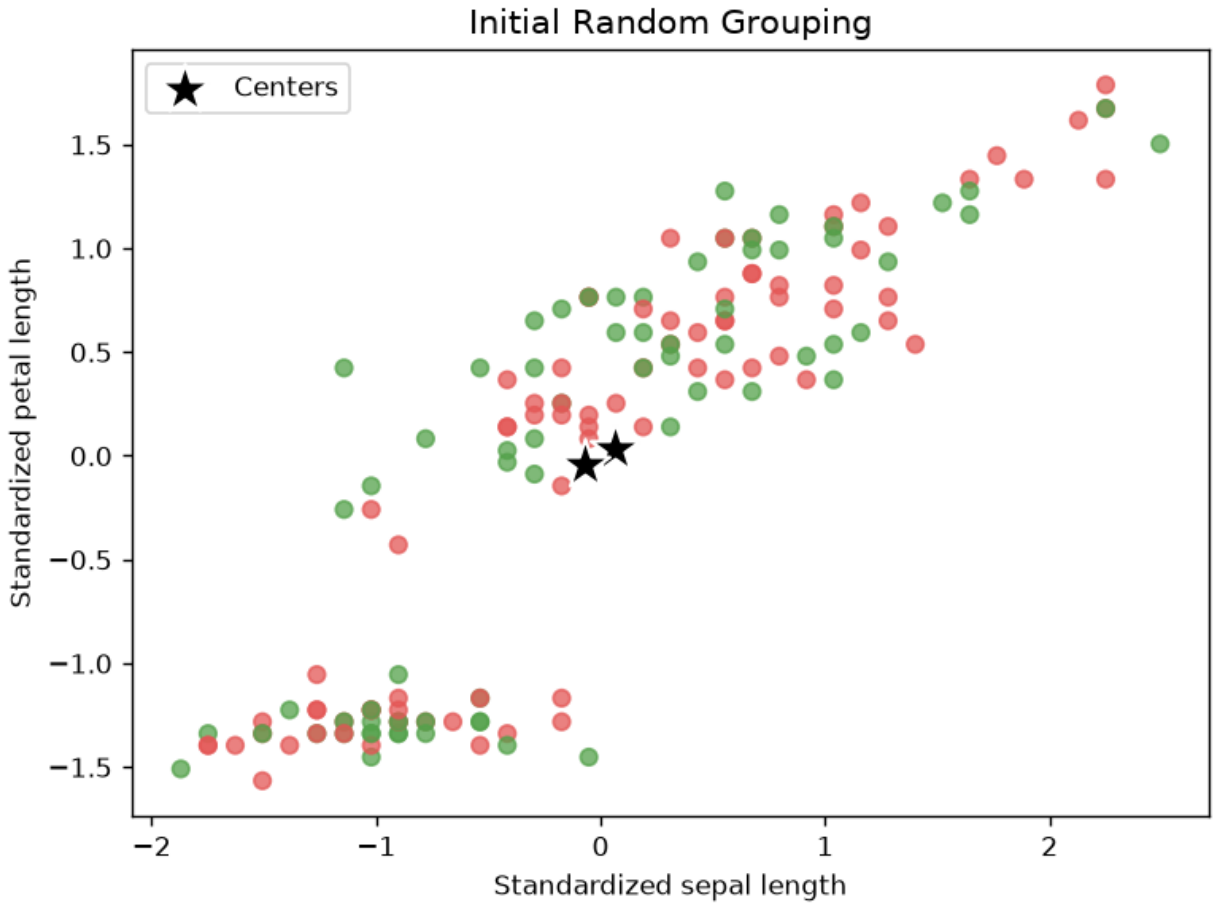


K-means clustering – step-by-step

Run the following Python code and explain each step.

```
cluster_features = ["sepal length (cm)",
                    "petal length (cm)"]
X_cluster_raw = iris_df[cluster_features]
cluster_scaler = StandardScaler()
X_cluster = cluster_scaler.fit_transform(X_cluster_raw)

K = 2
rng = np.random.default_rng(750)
labels = rng.integers(0, K, size=len(X_cluster))
centers = np.vstack([
    X_cluster[labels == cluster_id].mean(axis=0)
    for cluster_id in range(K)
])
```

This is a random grouping (first step)

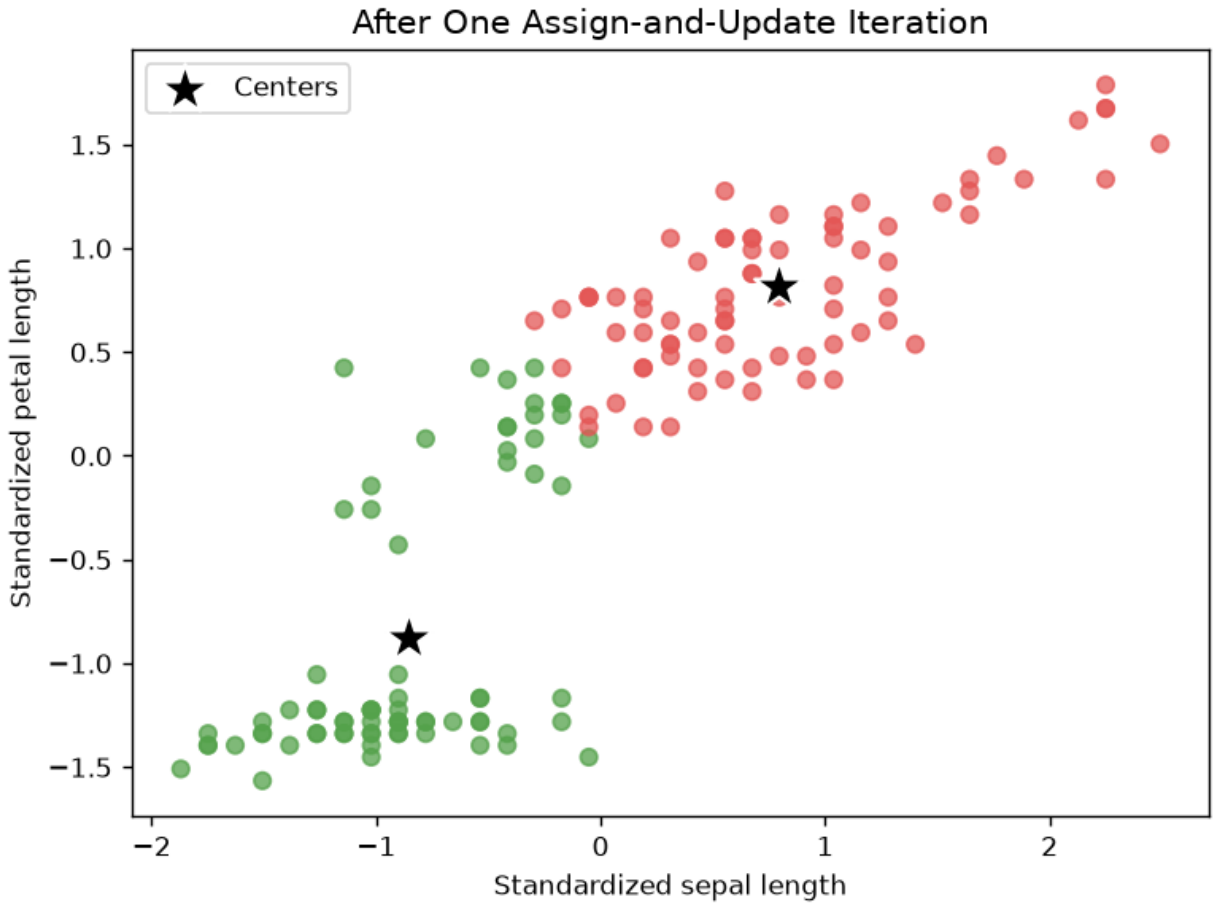


The next step

Assign each observation to its closest center, then recompute the centers.

```
def euclidean(x, y):
    return np.sqrt(np.sum((x - y) ** 2))

distances = np.array([
    [euclidean(row, center) for center in centers]
    for row in X_cluster
])

labels = distances.argmin(axis=1)
centers = np.vstack([
    X_cluster[labels == cluster_id].mean(axis=0)
    for cluster_id in range(K)
])
```

Data points are regrouped (second step)



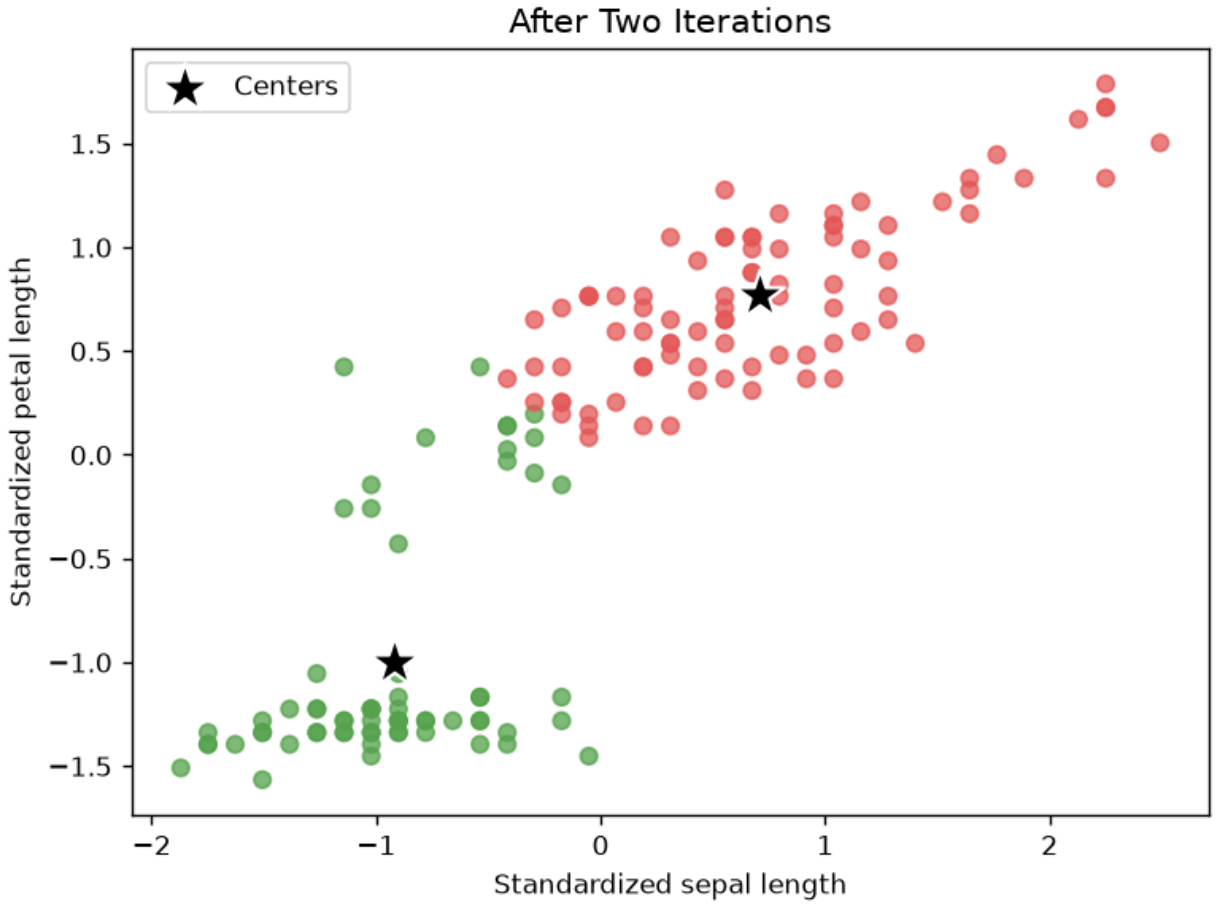
How does this happen?

Let's repeat the second step

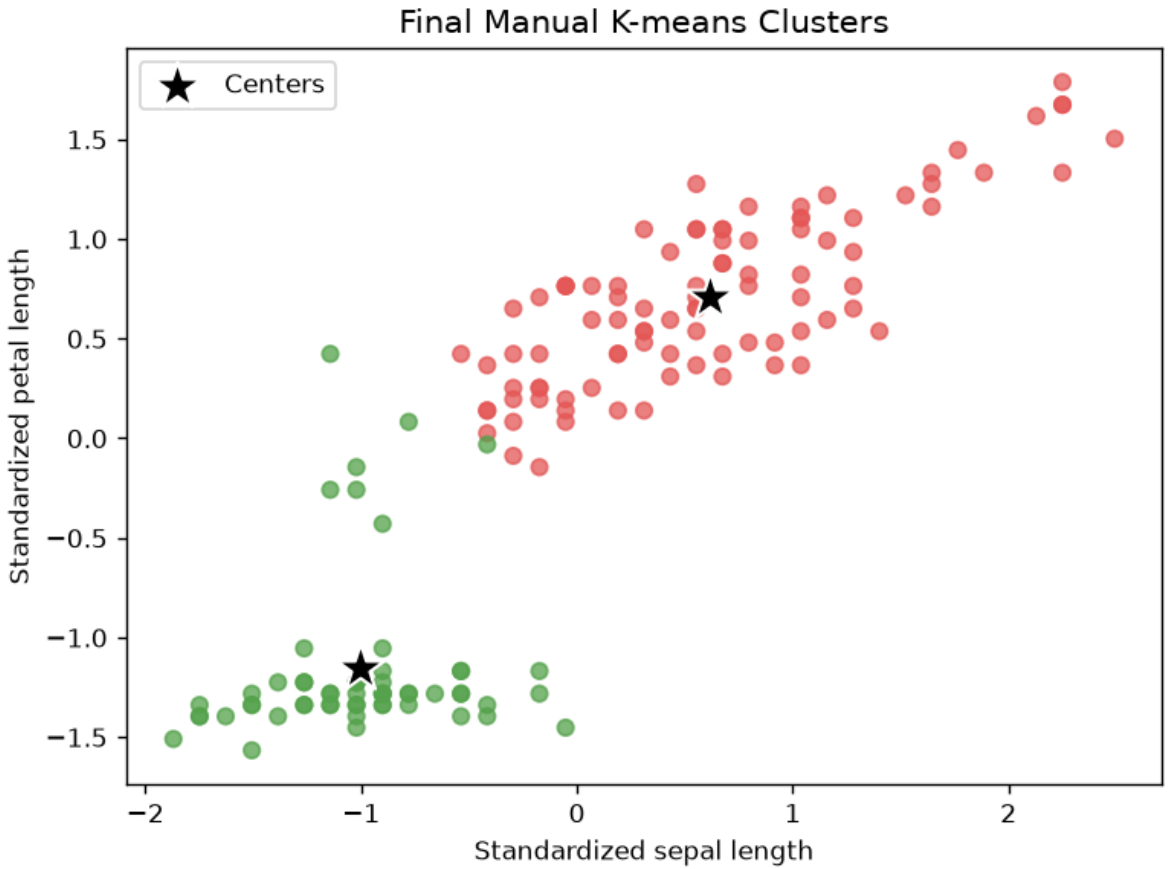
```
distances = np.array([
    [euclidean(row, center) for center in centers]
    for row in X_cluster
])

labels = distances.argmin(axis=1)
centers = np.vstack([
    X_cluster[labels == cluster_id].mean(axis=0)
    for cluster_id in range(K)
])
```

Note that the code does not change at all. Why?

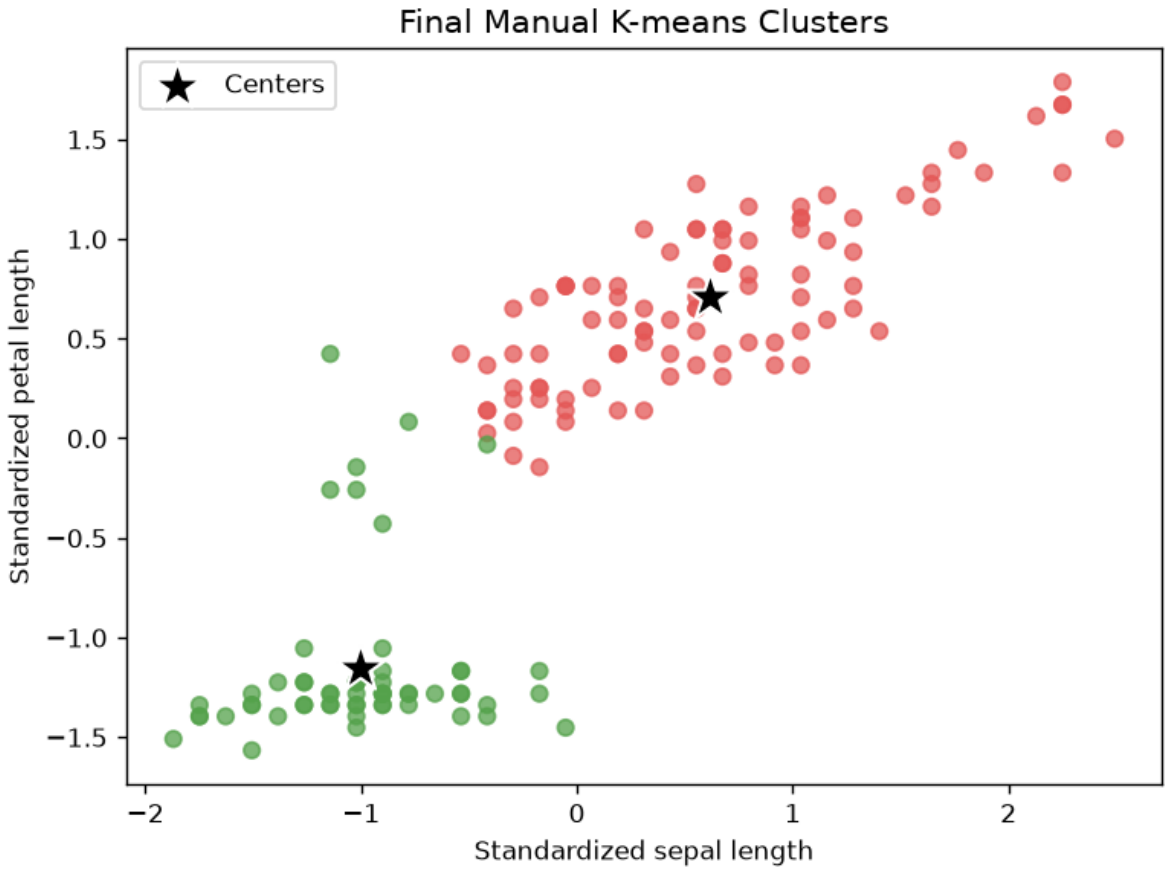
Data points are regrouped again



We can keep repeating this step, until...



We can keep repeating this step, until...



Members in each cluster do not change, which means the algorithm converges. How can we translate it into some numeric scores?

Statistics behind k-means clustering

The algorithm attempts to

- Minimize variance within clusters
- Maximize variance between clusters
- How about total variance?

Statistics behind k-means clustering

The algorithm attempts to

- Minimize variance within clusters
- Maximize variance between clusters
- How about total variance?

Instead of computing variance, we compute sum squared error (SSE).

- For a single variable X, SSE is defined as

$$SSE(X) = \sum_{i=1}^{n} (X_i - \bar{X})^2$$

Statistics behind k-means clustering

The algorithm attempts to

- Minimize variance within clusters
- Maximize variance between clusters
- How about total variance?

Instead of computing variance, we compute sum squared error (SSE).

- For a single variable X, SSE is defined as

$$SSE(X) = \sum_{i=1}^{n} (X_i - \bar{X})^2$$

- For multiple variables $\mathbf{X} = (X_1, ..., X_p)$, SSE is defined as

Statistics behind k-means clustering

The algorithm attempts to

- Minimize variance within clusters
- Maximize variance between clusters
- How about total variance?

Instead of computing variance, we compute sum squared error (SSE).

- For a single variable X, SSE is defined as

$$SSE(X) = \sum_{i=1}^{n} (X_i - \bar{X})^2$$

- For multiple variables $\mathbf{X} = (X_1, ..., X_p)$, SSE is defined as

$$SSE(\mathbf{X}) = \sum_{i=1}^{n} \sum_{j=1}^{p} (X_{ij} - \bar{X}_j)^2$$

Statistics behind k-means clustering

- $SSE(\mathbf{X})$ for entire sample: total sum squared error (SST).

Statistics behind k-means clustering

- $SSE(\mathbf{X})$ for entire sample: total sum squared error (SST).
- $SSE(\mathbf{X})$ for each cluster: within-group sum squared error (WSS).

Statistics behind k-means clustering

- $SSE(\mathbf{X})$ for entire sample: total sum squared error (SST).
- $SSE(\mathbf{X})$ for each cluster: within-group sum squared error (WSS).
- Sum of WSS across all clusters: total within-group sum squared error (TWSS)

Statistics behind k-means clustering

- $SSE(\mathbf{X})$ for entire sample: total sum squared error (SST).
- $SSE(\mathbf{X})$ for each cluster: within-group sum squared error (WSS).
- Sum of WSS across all clusters: total within-group sum squared error (TWSS)
- $SST - TWSS$: Between-group sum square

Statistics behind k-means clustering

- $SSE(\mathbf{X})$ for entire sample: total sum squared error (SST).
- $SSE(\mathbf{X})$ for each cluster: within-group sum squared error (WSS).
- Sum of WSS across all clusters: total within-group sum squared error (TWSS)
- $SST - TWSS$: Between-group sum square
- Exercise:
  Revisit the algorithm we just performed. Compute the above three measures at the end of each step. How are they changing over iterations?

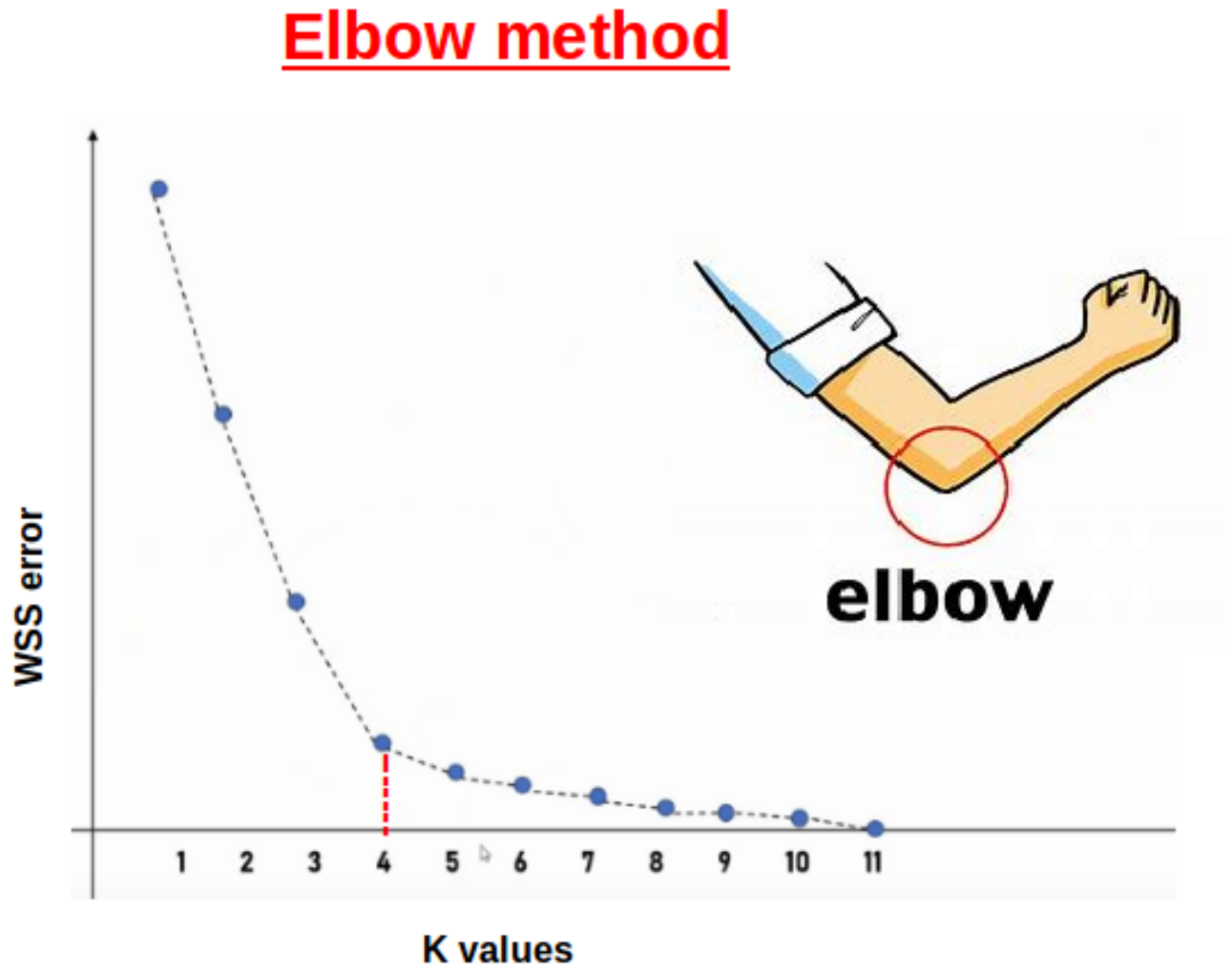
Choosing K

- Balancing the tradeoff between model complexity versus goodness of fit
  - Larger $K \implies$ higher complexity (poor interpretability) and better fit (maybe overfitting)
  - Smaller $K \implies$ lower complexity (better interpretability) and less precise
- A common approach to find optimal K: Elbow Method

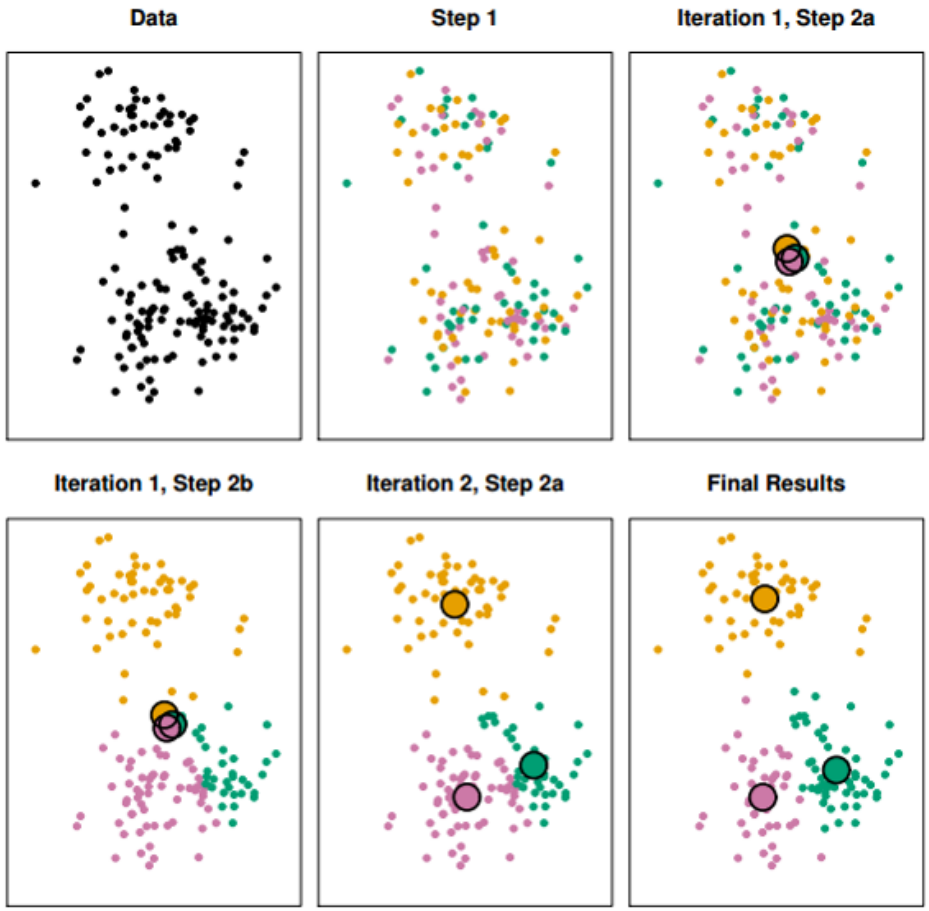
Choosing K

- Balancing the tradeoff between model complexity versus goodness of fit
  - Larger $K \implies$ higher complexity (poor interpretability) and better fit (maybe overfitting)
  - Smaller $K \implies$ lower complexity (better interpretability) and less precise
- A common approach to find optimal K: Elbow Method
  Plot TWSS vs. K and look for the "elbow" where additional clusters provide diminishing returns in variance reduction.

Elbow method



K-means algorithm recap



- Here is a very good animation to illustrate k-means clustering algorithm. [link]

K-means algorithm recap

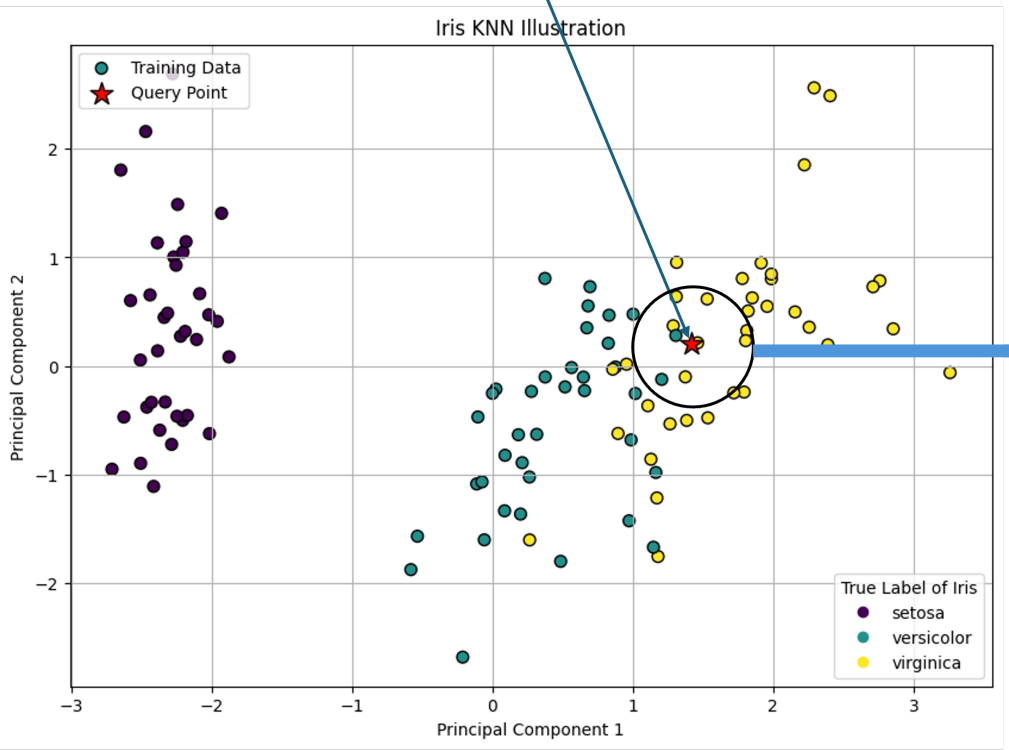
1. Initialize k centers (for example, by randomly assigning observations to clusters)
2. For each data point, find the closest center and label it (e.g., using different colors). Now you have k clusters
3. Re-calculate the centers of current clusters
4. Repeat step 2 and 3 until the centers do not change

K-nearest neighbors (KNN)

- Supervised learning
- Handles both regression and classification problems
- Intuition: predict for new data points by looking for similar data points in the training sample

An Illustration with Iris data

sepal_length	sepal_width	petal_length	petal_width
0.796	-0.132	0.99	0.791



	distance	species
0	0.260	virginica
1	0.314	virginica
2	0.339	virginica
3	0.396	versicolor
4	0.553	virginica
5	0.577	virginica
6	0.593	virginica
7	0.621	virginica
8	0.689	versicolor
9	0.709	virginica

More Details

- For classification,

$$\hat{Y} = \arg\max_{c} \sum_{\mathbf{x}_i \in N_k(\mathbf{x})} I(y_i = c)$$

- For regression,

$$\hat{Y} = \frac{1}{n_k} \sum_{\mathbf{x}_i \in N_k(\mathbf{x})} y_i$$

$N_k(\mathbf{x})$ is the neighbor of $\mathbf{x}$ defined by the k closest points $\mathbf{x}_i$.

- k: size of the neighbor (How does k influence the model?)
- The key computation: from training sample find the neighbor (of size k) of a new data point $\mathbf{x}$

KNN algorithm illustration

Data preparation

```
# RUN: Standardize the full sample, then create a reproducible
      train-test split.
X = iris.data.to_numpy()
y = iris.target.to_numpy()

scaler = StandardScaler()
X = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=750
)

print("Training observations:", len(X_train))
print("Test observations:", len(X_test))
print("Training class counts:", np.bincount(y_train))
```

KNN algorithm illustration

Key component – computing distance

```
# RUN: Calculate the distance from one test observation
    to every training row.
distances = []
for row in X_train:
    distances.append(euclidean(X_test[0], row))

distances = np.array(distances)

print("Number of calculated distances:", len(distances))
print("Smallest distance:", round(distances.min(), 3))
```

KNN algorithm illustration

Find neighbor

```
# RUN: Locate the k nearest training observations.
k = 10
neighbor_index = np.argsort(distances)[:k]
neighbor_labels = y_train[neighbor_index]
votes = Counter(neighbor_labels)
prediction = votes.most_common(1)[0][0]
print("Predicted species:", iris.target_names[prediction])

neighbor_table = pd.DataFrame({
    "distance": distances[neighbor_index],
    "species": iris.target_names[neighbor_labels]
}).round(3)
display(neighbor_table)
```

KNN algorithm illustration

Make prediction (majority vote)

```
# RUN: Use majority voting to make the manual prediction
    .
def majority_vote(labels):
    return Counter(labels).most_common(1)[0][0]

prediction = majority_vote(neighbor_labels)

print("Votes:", Counter(iris.target_names[
    neighbor_labels]))
print("Predicted species:", iris.target_names[prediction
    ])
print("Actual species:", iris.target_names[y_test[0]])
```

KNN algorithm illustration

Make prediction (majority vote)

```
# RUN: Use majority voting to make the manual prediction
    .
def majority_vote(labels):
    return Counter(labels).most_common(1)[0][0]

prediction = majority_vote(neighbor_labels)

print("Votes:", Counter(iris.target_names[
    neighbor_labels]))
print("Predicted species:", iris.target_names[prediction
    ])
print("Actual species:", iris.target_names[y_test[0]])
```

- We just predicted one data point in the testing sample!

KNN algorithm illustration

Make prediction (majority vote)

```
# RUN: Use majority voting to make the manual prediction
    .
def majority_vote(labels):
    return Counter(labels).most_common(1)[0][0]

prediction = majority_vote(neighbor_labels)

print("Votes:", Counter(iris.target_names[
    neighbor_labels]))
print("Predicted species:", iris.target_names[prediction
    ])
print("Actual species:", iris.target_names[y_test[0]])
```

- We just predicted one data point in the testing sample!
- Repeat the same process for the rest data points in the testing sample.

Use scikit-learn for KNN

Practically!

```
# RUN: Fit and evaluate a KNN classifier.
knn_model = KNeighborsClassifier(n_neighbors=10)
knn_model.fit(X_train, y_train)
test_predictions = knn_model.predict(X_test)

print("First scikit-learn prediction:", iris.
    target_names[test_predictions[0]])
print("Test accuracy:", round(accuracy_score(y_test,
    test_predictions), 3))
```

- In general, you need to try different k (model tuning)
  - Use cross-validation (how would you define the CV score?)
  - Find the best k
- Deploy the model
  - or compare with other models

K-NN algorithm recap

- Standardization (why?)
- For each data point to be predicted (known X unknown Y), compute the distance between this X to each $X_{\text{train},i}$ of training data points.
- Sort and find the group of k training data points with smallest distance (k nearest neighbor).
- The unknown Y to be predicted = Majority vote or average of Y_{train}'s in the group (neighbor) found in previous step.

What is the impact of k – an example[2]

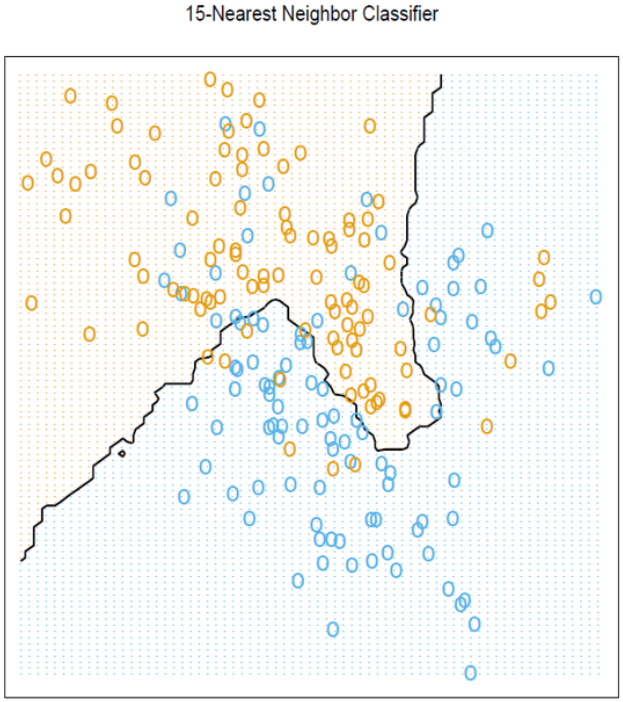


FIGURE 2.2. *The same classification example in two dimensions as in Figure 2.1. The classes are coded as a binary variable* (BLUE $= 0$, ORANGE $= 1$) *and then fit by 15-nearest-neighbor averaging as in (2.8). The predicted class is hence chosen by majority vote amongst the 15-nearest neighbors.*

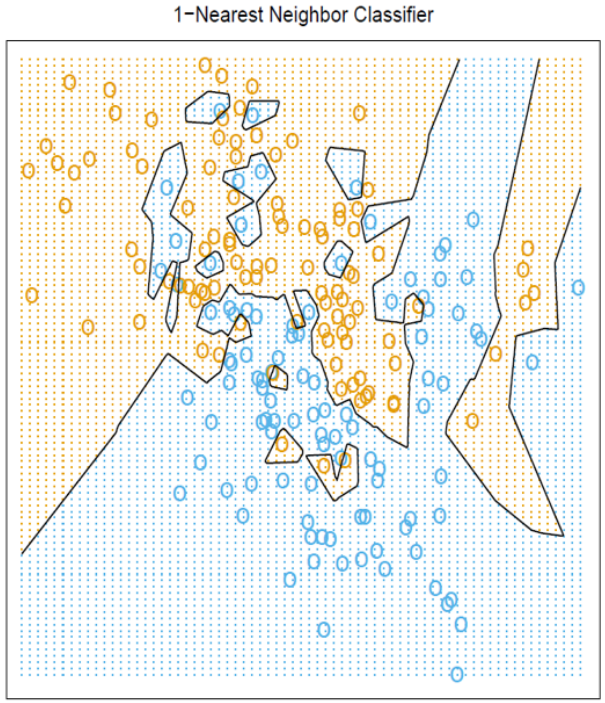


FIGURE 2.3. *The same classification example in two dimensions as in Figure 2.1. The classes are coded as a binary variable* (BLUE $= 0$, ORANGE $= 1$), *and then predicted by 1-nearest-neighbor classification.*

[2]Source: ESL pp.15-16

Distance-based Similarity Measures