Seminar 9: Core Machine Learning 3

LSE ME314: Introduction to Data Science and Machine Learning

MODIFIED
July 17, 2025

Plan for Today

The focus of today's session is on ensembles and unsupervised ML. First, we will explore ensembles and demonstrate how combining the predictions of multiple models can boost predictive performance. After this, we will look at applying clustering, an unsupervised technique. Next, we will use unsupervised ML for dimensionality reduction. Finally, there are multiple exercises available to specify and train your own ensemble model as well as apply unsupervised ML techniques.

```r
options(repos = c(CRAN = "https://cloud.r-project.org"))
install.packages("tidyverse")
```

```
The downloaded binary packages are in
	/var/folders/95/c0789kd55z1css3h1gwwtrj40000gn/T//RtmpfBoDtC/downloaded_packages
```

```r
install.packages("ggthemes")
```

```
The downloaded binary packages are in
	/var/folders/95/c0789kd55z1css3h1gwwtrj40000gn/T//RtmpfBoDtC/downloaded_packages
```

```r
install.packages("palmerpenguins")
```

```
The downloaded binary packages are in
	/var/folders/95/c0789kd55z1css3h1gwwtrj40000gn/T//RtmpfBoDtC/downloaded_packages
```

```r
install.packages("rattle")
```

```
The downloaded binary packages are in
	/var/folders/95/c0789kd55z1css3h1gwwtrj40000gn/T//RtmpfBoDtC/downloaded_packages
```

```r
install.packages("FactoMineR")
```

```
The downloaded binary packages are in
	/var/folders/95/c0789kd55z1css3h1gwwtrj40000gn/T//RtmpfBoDtC/downloaded_packages
```

```r
install.packages("factoextra")
```

```
The downloaded binary packages are in
    /var/folders/95/c0789kd55z1css3h1gwwtrj40000gn/T//RtmpfBoDtC/downloaded_packages
```

```
library(tidyverse)
```

```
Warning: package 'ggplot2' was built under R version 4.3.3

Warning: package 'tibble' was built under R version 4.3.3

── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✔ dplyr     1.1.4     ✔ readr     2.1.5
✔ forcats   1.0.0     ✔ stringr   1.5.1
✔ ggplot2   3.5.2     ✔ tibble    3.3.0
✔ lubridate 1.9.3     ✔ tidyr     1.3.1
✔ purrr     1.0.2
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts
to become errors
```

```
library(ggthemes)
library(palmerpenguins)
```

```
Warning: package 'palmerpenguins' was built under R version 4.3.3
```

```
library(rattle)
```

```
Loading required package: bitops

Warning: package 'bitops' was built under R version 4.3.3

Rattle: A free graphical interface for data science with R.
Version 5.5.1 Copyright (c) 2006-2021 Togaware Pty Ltd.
Type 'rattle()' to shake, rattle, and roll your data.
```

```
library(FactoMineR)
library(factoextra)
```

```
Welcome! Want to learn more? See two factoextra-related books at https://goo.gl/ve3WBa
```

```
# setwd("'/Users/christycoulson/Documents/Jobs/Work/Academia/Teaching/ME314/24-25/Semi
library(reticulate)
```

```
Warning: package 'reticulate' was built under R version 4.3.3
```

```
use_python("/Users/charlottekuberka/anaconda3/bin/python", required = TRUE)
```

```python
import sys
import subprocess

print(f"🧠 Python executable in use: {sys.executable}\n")
```

```
🧠 Python executable in use: /Users/charlottekuberka/anaconda3/bin/python
```

```python
# List of pip package names to install
pip_packages = [
    "numpy",
    "pandas",
    "matplotlib",
    "seaborn",
    "scikit-learn"
]

# Attempt to install each package
for package in pip_packages:
    try:
        subprocess.check_call([sys.executable, "-m", "pip", "install", package])
        print(f"Installed: {package}")
    except Exception as e:
        print(f"Failed to install {package}: {e}")
```

```
0
Installed: numpy
0
Installed: pandas
0
Installed: matplotlib
0
Installed: seaborn
0
Installed: scikit-learn
```

```python
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
import sklearn.metrics
```

Part 1: Ensembles

Ensemble methods combine multiple base learners to reduce prediction error by leveraging their diverse inductive biases. Rather than relying on a single model, ensembles integrate multiple algorithms — each trained on the same task — to minimize generalization error through techniques such as averaging (for regression) or majority voting (for classification). Theoretical work by van der Laan et al. (2007) shows that the Super Learner, a cross-validated ensemble that weights base learners according to their performance, is asymptotically guaranteed to perform at least as well as the best convex combination of candidate learners in terms of expected loss. By doing so, ensembles can reduce variance (e.g., via bagging), reduce bias (e.g., via boosting), or optimize a weighted trade-off between the two — making them particularly useful in scenarios where no single model clearly dominates across all regions of the input space.

In the following code, we demonstrate this by fitting a standalone logistic regression model, a standalone random forest, a standalone gradient boosting model, and a Super Learner (stacking) ensemble comprising of all 3. We then compare their performance on a classification task to illustrate the benefits of ensemble learning. Below is an implementation in python (using `scikit-learn`).

First, let's load some data with python. We will use the canonical Breast Cancer Wisconsin (Diagnostic) dataset, which contains 569 samples with 30 numeric features describing cell nuclei from digitized images, labeled as either malignant or benign. We wikk break this data up into a train and test set, as seen below:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, cross_val_score, cross_val_predi
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, StackingClassifier, HistGradientB
from sklearn.metrics import accuracy_score, roc_auc_score, make_scorer

# Load and split data
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
```

Next, we're going to define the stack of base-learners that we want to include in our ensemble.

```
# Define base learners
lr = LogisticRegression(max_iter=1000, solver='liblinear')
rf = RandomForestClassifier(n_estimators=100, random_state=42)
gb = HistGradientBoostingClassifier(random_state=42)
```

Now, we're going to fit our standalone base-learners (models) with 5-fold cross-validation and evaluate them based on accuracy and ROC-AUC. We have to define our evaluation metrics prior to CV using the `make_scorer()` function.

```
from sklearn.model_selection import cross_val_score
from sklearn.metrics import make_scorer, roc_auc_score

scoring_acc = 'accuracy'
scoring_auc = make_scorer(roc_auc_score)

for name, model in [('LR', lr), ('RF', rf), ('GB', gb)]:
    print(f"About to start CV for {name}…", flush=True)
    print("  → running accuracy CV…", flush=True)
    scores = cross_val_score(
        model, X_train, y_train,
        cv=3,                # fewer folds for quick check
        scoring=scoring_acc,
        n_jobs=1             # single-core: avoid multiprocessing issues
    )
    print("Accuracy done", flush=True)

    print("Running AUC CV…", flush=True)
    aucs = cross_val_score(
        model, X_train, y_train,
        cv=5,
        scoring=scoring_auc,
        n_jobs=1
    )
    print("AUC done", flush=True)

    print(f"{name} CV Accuracy = {scores.mean():.3f} ± {scores.std():.3f}", flush=True
    print(f"{name} CV AUC = {aucs.mean():.3f} ± {aucs.std():.3f}", flush=True)
```

```
About to start CV for LR…
  → running accuracy CV…
Accuracy done
Running AUC CV…
AUC done
LR CV Accuracy = 0.955 ± 0.011
LR CV AUC = 0.954 ± 0.020
About to start CV for RF…
  → running accuracy CV…
Accuracy done
Running AUC CV…
AUC done
RF CV Accuracy = 0.962 ± 0.022
RF CV AUC = 0.970 ± 0.036
About to start CV for GB…
  → running accuracy CV…
Accuracy done
Running AUC CV…
AUC done
GB CV Accuracy = 0.967 ± 0.013
GB CV AUC = 0.959 ± 0.022
```

Now, we're going to take our cross-validated models and fit them on the **full** training data. We will follow this with OOS evaluation and retrieve both the predicted *classes* and *probabilities*.

```
from sklearn.metrics import accuracy_score, roc_auc_score

# Fit and evaluate Standalone Logistic Regression
lr.fit(X_train, y_train)
```

```
▾ LogisticRegression

LogisticRegression(max_iter=1000, solver='liblinear')
```

```
y_pred_lr = lr.predict(X_test)
y_prob_lr = lr.predict_proba(X_test)[:, 1]
acc_lr = accuracy_score(y_test, y_pred_lr)
auc_lr = roc_auc_score(y_test, y_prob_lr)
print(f"Standalone Logistic Regression: Accuracy = {acc_lr:.3f}, AUC = {auc_lr:.3f}")
```

```
Standalone Logistic Regression: Accuracy = 0.942, AUC = 0.987
```

```
# Fit and evaluate Random Forest
rf.fit(X_train, y_train)
```

```
▾ RandomForestClassifier

RandomForestClassifier(random_state=42)
```

```
y_pred_rf = rf.predict(X_test)
y_prob_rf = rf.predict_proba(X_test)[:, 1]
acc_rf = accuracy_score(y_test, y_pred_rf)
auc_rf = roc_auc_score(y_test, y_prob_rf)
print(f"Standalone Random Forest: Accuracy = {acc_rf:.3f}, AUC = {auc_rf:.3f}")
```

```
Standalone Random Forest: Accuracy = 0.936, AUC = 0.991
```

```
# Fit and evaluate Gradient Boosting
gb.fit(X_train, y_train)
```

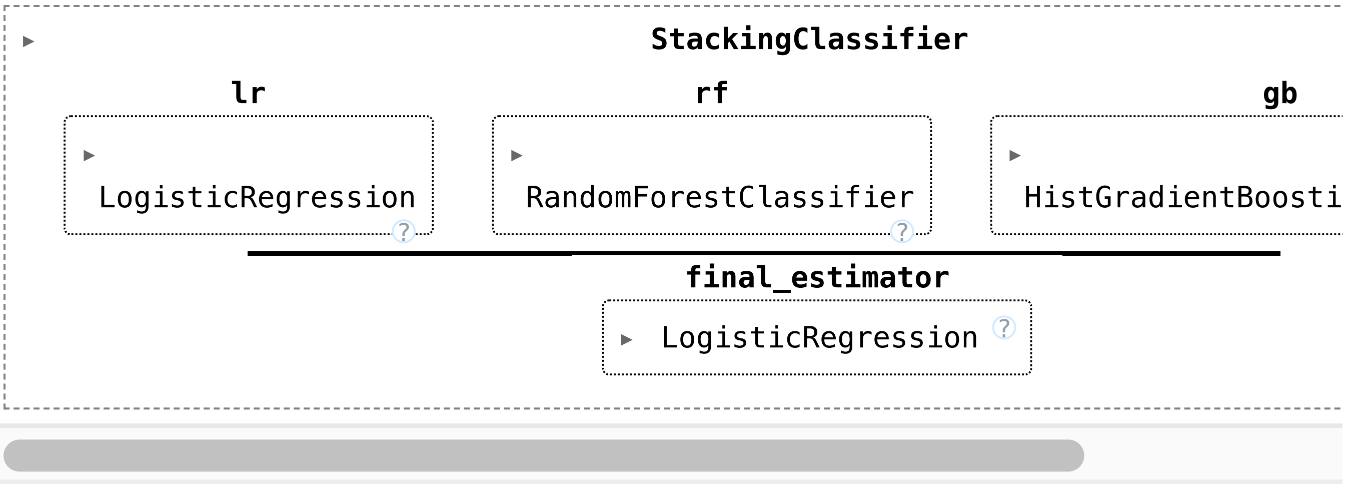
```
▾ HistGradientBoostingClassifier

HistGradientBoostingClassifier(random_state=42)
```

```
y_pred_gb = gb.predict(X_test)
y_prob_gb = gb.predict_proba(X_test)[:, 1]
acc_gb = accuracy_score(y_test, y_pred_gb)
auc_gb = roc_auc_score(y_test, y_prob_gb)
print(f"Standalone Gradient Boosting: Accuracy = {acc_gb:.3f}, AUC = {auc_gb:.3f}")
```

```
Standalone Gradient Boosting: Accuracy = 0.959, AUC = 0.992
```

After training our standalone model, we'll fit our ensemble below:

```
# Super Learner (Stacking)
estimators = [('lr', lr), ('rf', rf), ('gb', gb)]

stack = StackingClassifier(
    estimators=estimators,
    final_estimator=LogisticRegression(solver='liblinear'),
    cv=5
)

stack.fit(X_train, y_train)
```



```
y_pred_stack = stack.predict(X_test)
y_prob_stack = stack.predict_proba(X_test)[:, 1]

acc_stack = accuracy_score(y_test, y_pred_stack)
auc_stack = roc_auc_score(y_test, y_prob_stack)
print(f"Super Learner (Stacking): Accuracy = {acc_stack:.3f}, AUC = {auc_stack:.3f}")
```

```
Super Learner (Stacking): Accuracy = 0.965, AUC = 0.993
```

And finally, let's visualise the performances of these competing models with ROC curves.

```
from sklearn.metrics import roc_curve, auc

# Compute predicted probabilities for the positive class for all models
y_prob_lr = lr.predict_proba(X_test)[:, 1]
y_prob_rf = rf.predict_proba(X_test)[:, 1]
y_prob_gb = gb.predict_proba(X_test)[:, 1]
y_prob_stack = stack.predict_proba(X_test)[:, 1]

# Compute ROC curves and AUC for each model
fpr_lr, tpr_lr, _ = roc_curve(y_test, y_prob_lr)
auc_lr = auc(fpr_lr, tpr_lr)

fpr_rf, tpr_rf, _ = roc_curve(y_test, y_prob_rf)
auc_rf = auc(fpr_rf, tpr_rf)

fpr_gb, tpr_gb, _ = roc_curve(y_test, y_prob_gb)
auc_gb = auc(fpr_gb, tpr_gb)

fpr_stack, tpr_stack, _ = roc_curve(y_test, y_prob_stack)
auc_stack = auc(fpr_stack, tpr_stack)

# Plot ROC curves for all models
plt.figure(figsize=(8, 6))
```

```
<Figure size 800x600 with 0 Axes>
```

```
plt.plot(fpr_lr, tpr_lr, label=f"Logistic Regression (AUC = {auc_lr:.3f})")
```

```
[<matplotlib.lines.Line2D object at 0x29a5eb910>]
```

```
plt.plot(fpr_rf, tpr_rf, label=f"Random Forest (AUC = {auc_rf:.3f})")
```

```
[<matplotlib.lines.Line2D object at 0x29a97c890>]
```

```
plt.plot(fpr_gb, tpr_gb, label=f"Gradient Boosting (AUC = {auc_gb:.3f})")
```

```
[<matplotlib.lines.Line2D object at 0x29a97dfd0>]
```

```
plt.plot(fpr_stack, tpr_stack, label=f"Stacking Ensemble (AUC = {auc_stack:.3f})")
```

```
[<matplotlib.lines.Line2D object at 0x29a97dd10>]
```

```
plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
```

```
[<matplotlib.lines.Line2D object at 0x299973cd0>]
```

```
plt.xlabel('False Positive Rate')
```

```
Text(0.5, 0, 'False Positive Rate')
```

```
plt.ylabel('True Positive Rate')
```

```
Text(0, 0.5, 'True Positive Rate')
```

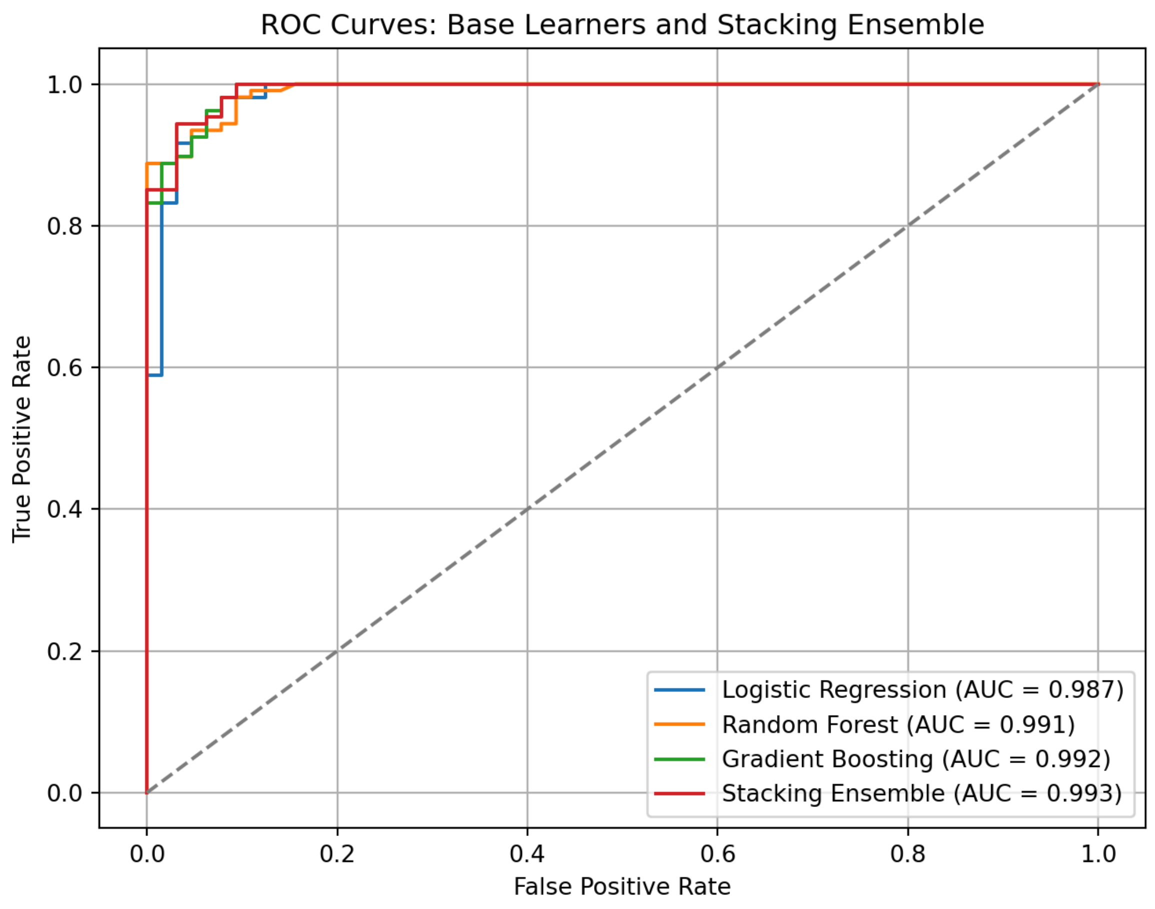
```
plt.title('ROC Curves: Base Learners and Stacking Ensemble')
```

```
Text(0.5, 1.0, 'ROC Curves: Base Learners and Stacking Ensemble')
```

```
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend object at 0x29a769d50>
```

```
plt.grid(True)
plt.show()
```



Questions:

- Which model performs best?
- How do accuracy and AUC differ, and why might they lead to different conclusions about model performance?
- How does using cross-validation inside the StackingClassifier help reduce overfitting?
- What trade-offs are involved in adding more complex models to the ensemble?

Part 2: Unsupervised Machine Learning

Unsupervised machine learning is particularly useful for exploratory analyses such as customer segmentation, anomaly detection, or feature extraction. In these contexts, the ML practitioners intends to uncover hidden patterns without predefined labels (unlabelled data). The algorithm must discover structure or patterns on its own, for example by clustering similar instances or reducing dimensionality. Unsupervised ML is ideal when you don't have predefined outcomes and want to explore hidden groupings or anomalies. In contrast, supervised ML relies on labeled examples (input–output pairs) to train models that can predict or classify new data. While unsupervised methods reveal the data's intrinsic organisation, supervised methods learn an explicit mapping from inputs to known targets.

Part 2.1: Clustering

In many real-world problems, we're faced with data where labels or outcomes aren't available. In these cases, we often want to discover hidden structure or groupings in the data. For this, we might want to use unsupervised ML techniques, and one of the most widely used in this space is clustering. Clustering is fundamentally inductive: we ask the algorithm to infer patterns in the data based on similarities between observations, rather than based on outcomes we provide.

Clustering allows us to group observations that are similar to one another across multiple dimensions. It's useful in exploratory data analysis, customer segmentation, and many other applications that benefit from the discovery of hidden groupings.

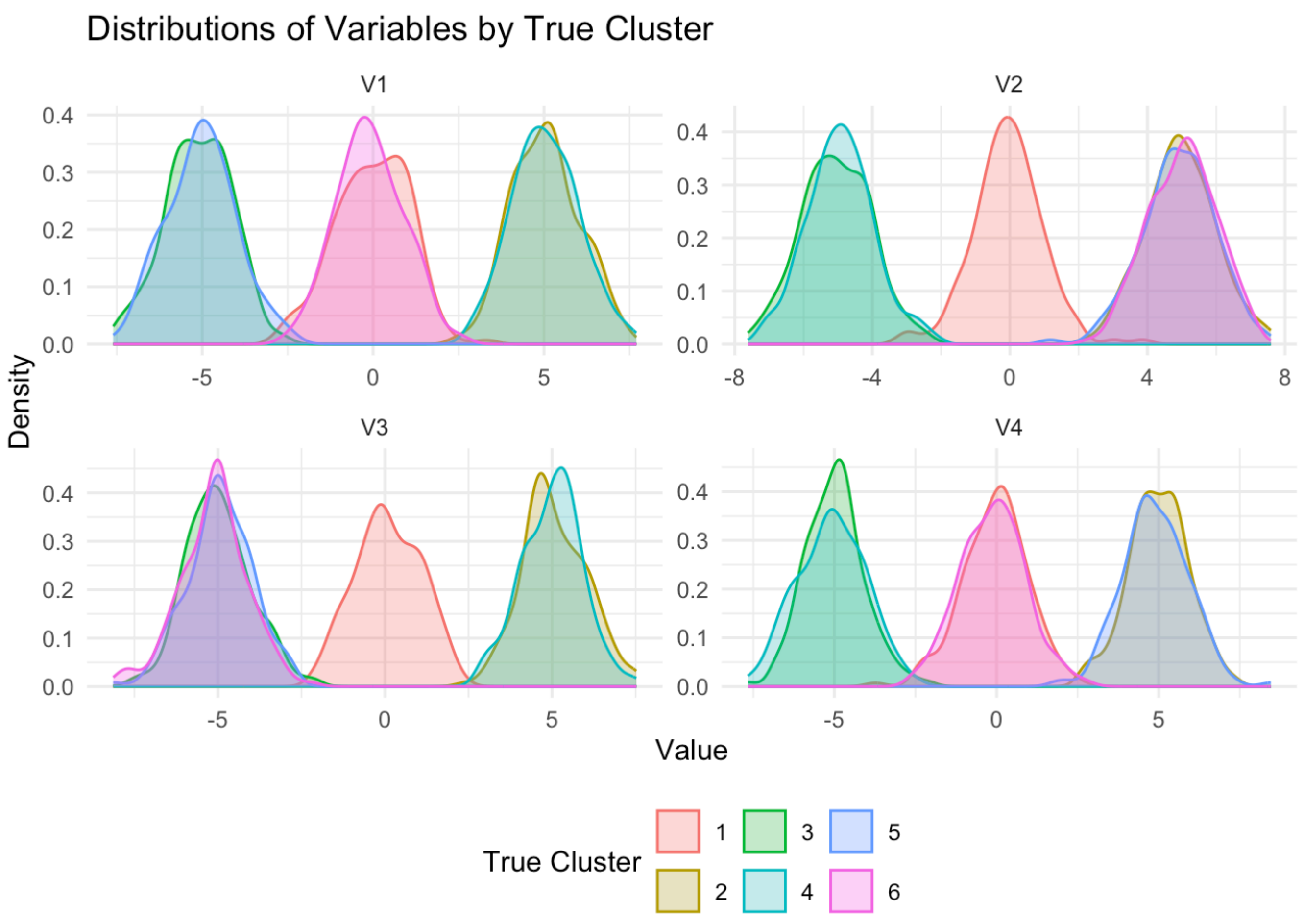
Today, we'll motivate the idea of clustering using the popular k-means algorithm, which attempts to partition observations into k groups by minimizing the distance between each point and its assigned group's center. Before we apply k-means, let's first generate some synthetic data to simulate a clustering problem. We'll create a dataset with 1,000 observations, 4 variables, and 6 underlying true clusters. While these clusters are known in this synthetic example (for teaching purposes), we'll pretend we don't know them and try to uncover them using k-means.

```r
set.seed(123)

# Generate synthetic dataset
centers <- matrix(
    c(
        0, 0, 0, 0,
        5, 5, 5, 5,
        -5, -5, -5, -5,
        5, -5, 5, -5,
        -5, 5, -5, 5,
        0, 5, -5, 0
    ),
    nrow = 6, ncol = 4, byrow = TRUE
)

n <- 1000

# Assign true cluster for each sample
t <- sample(1:6, size = n, replace = TRUE)

# Create data.frame
X_vals <- sapply(1:4, function(j) rnorm(n, mean = centers[t, j], sd = 1))
X <- as.data.frame(X_vals)
colnames(X) <- paste0("V", 1:4)
X$true_cluster <- factor(t)

str(X)
```

```
'data.frame':	1000 obs. of  5 variables:
 $ V1          : num  -6 -0.16 -5.53 5.22 4.08 ...
 $ V2          : num  -5.21 6.09 -5.03 4.91 5.46 ...
 $ V3          : num  -4.32 -4.05 -4.27 6.01 6.05 ...
 $ V4          : num  -5.655 -0.467 -5.316 6.706 4.897 ...
 $ true_cluster: Factor w/ 6 levels "1","2","3","4",..: 3 6 3 2 2 6 3 5 4 6 ...
```

Before jumping into clustering, we typically start with some exploratory data analysis (EDA). Here, we'll look at the distributions of each variable and see whether any of them provide a clear, unambiguous separation between the clusters. This helps set expectations about how hard the clustering task might be.

```r
library(ggplot2)

ggplot(pivot_longer(X[, 1:5],
    cols = starts_with("V"),
    names_to = "variable",
    values_to = "value"
), aes(x = value, color = true_cluster, fill = true_cluster)) +
    geom_density(alpha = 0.3) +
    facet_wrap(~variable, scales = "free") +
    labs(
        title = "Distributions of Variables by True Cluster",
        x = "Value", y = "Density",
        color = "True Cluster",
        fill = "True Cluster"
    ) +
    theme_minimal() +
    theme(legend.position = "bottom")
```



As we can see, each individual variable shows substantial overlap between the true clusters. This makes sense: no single variable alone can cleanly distinguish all 6 groups. In fact, this is a key motivation for clustering algorithms like k-means — they consider all dimensions jointly, capturing patterns in the multivariate space that are invisible in any one variable viewed in isolation.

Let's now suppose that, based on our intuition or domain knowledge, we suspect there might be **three** underlying clusters in this dataset. This kind of hypothesis is common in real-world scenarios - perhaps an analyst believes there are three customer types, three political blocs, or three behavioral profiles. Crucially, in practice, we don't know the true number of clusters (which in our synthetic data is 6). Our goal with clustering is to discover latent structure in the data — even if we don't know the "correct" number of groups in advance.

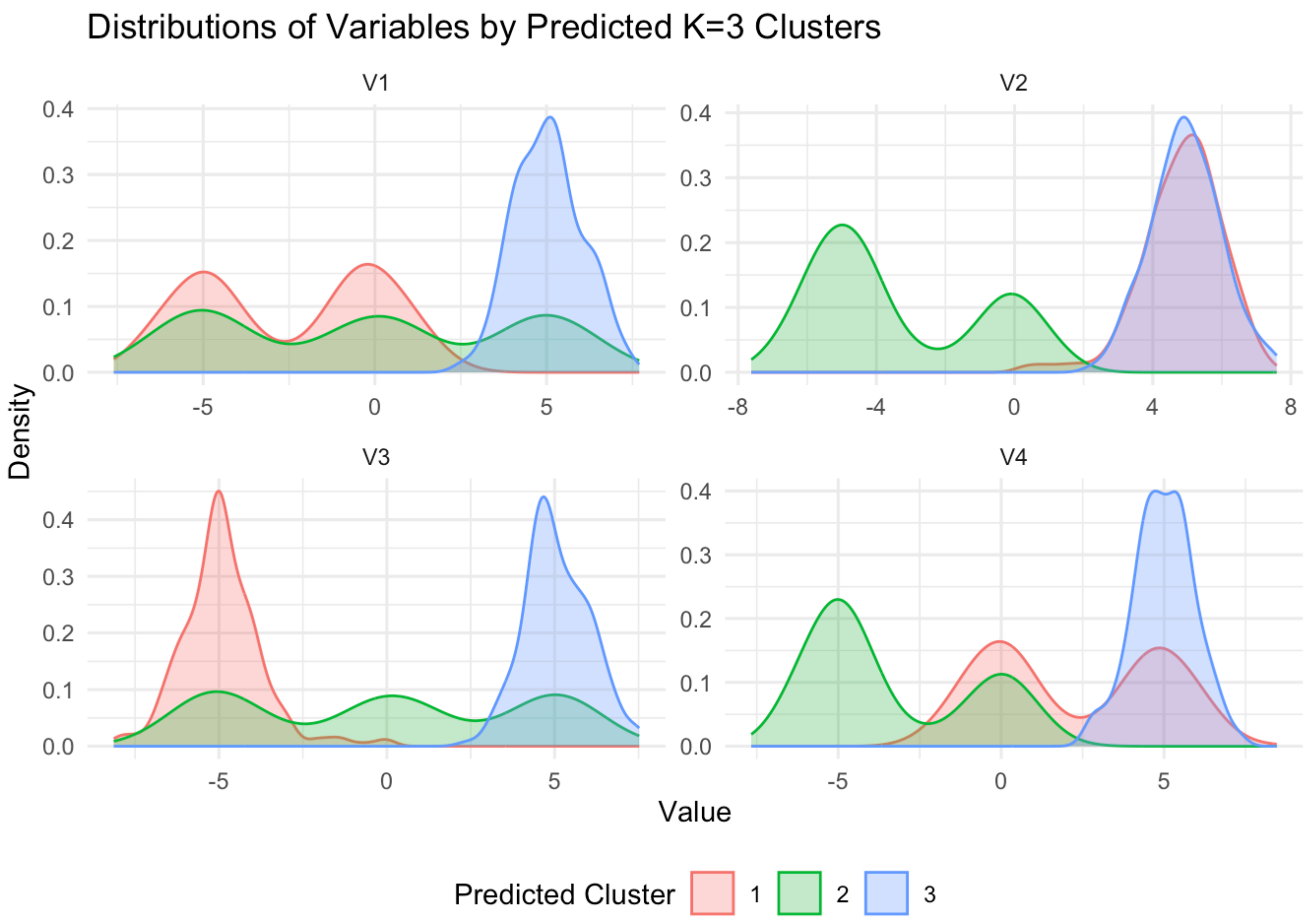
To explore this, we'll apply k-means clustering, a classic and efficient clustering method. We're going to use the `kmeans()` function from base R to assign each data point to one of 3 clusters. The algorithm will randomly initialise centroids, assign each observation to its nearest centroid, update centroids based on current cluster membership, and repeat until convergence. Set the `nstart` argument to 50 to ensure the algorithm tries 50 random initial configurations and selects the best one (this helps avoid poor local minima).

```
# for reproducibility
set.seed(123)

# Run k-means with k = 3
?kmeans()

k3_model <- kmeans(X[, 1:4], centers = 3, nstart = 50)


# Add predicted cluster assignments to the data
X$k3_cluster <- factor(k3_model$cluster)

# Inspect the first few rows
head(X)
```

```
          V1        V2        V3         V4 true_cluster k3_cluster
1 -5.9997470 -5.211958 -4.320995 -5.6546440            3          2
2 -0.1596040  6.090603 -4.048821 -0.4670049            6          1
3 -5.5276936 -5.030238 -4.272276 -5.3163228            3          2
4  5.2161565  4.909193  6.013760  6.7061310            2          3
5  4.0810450  5.457256  6.053263  4.8974015            2          3
6 -0.9005947  5.392604 -6.775277 -0.1736446            6          1
```

In the real-world, we would not have ground-truth labels against which to compare our estimated clusters, so it is difficult to evaluate our pattern detection. Let's visualise our predicted clusters to see how they vary by predictors.

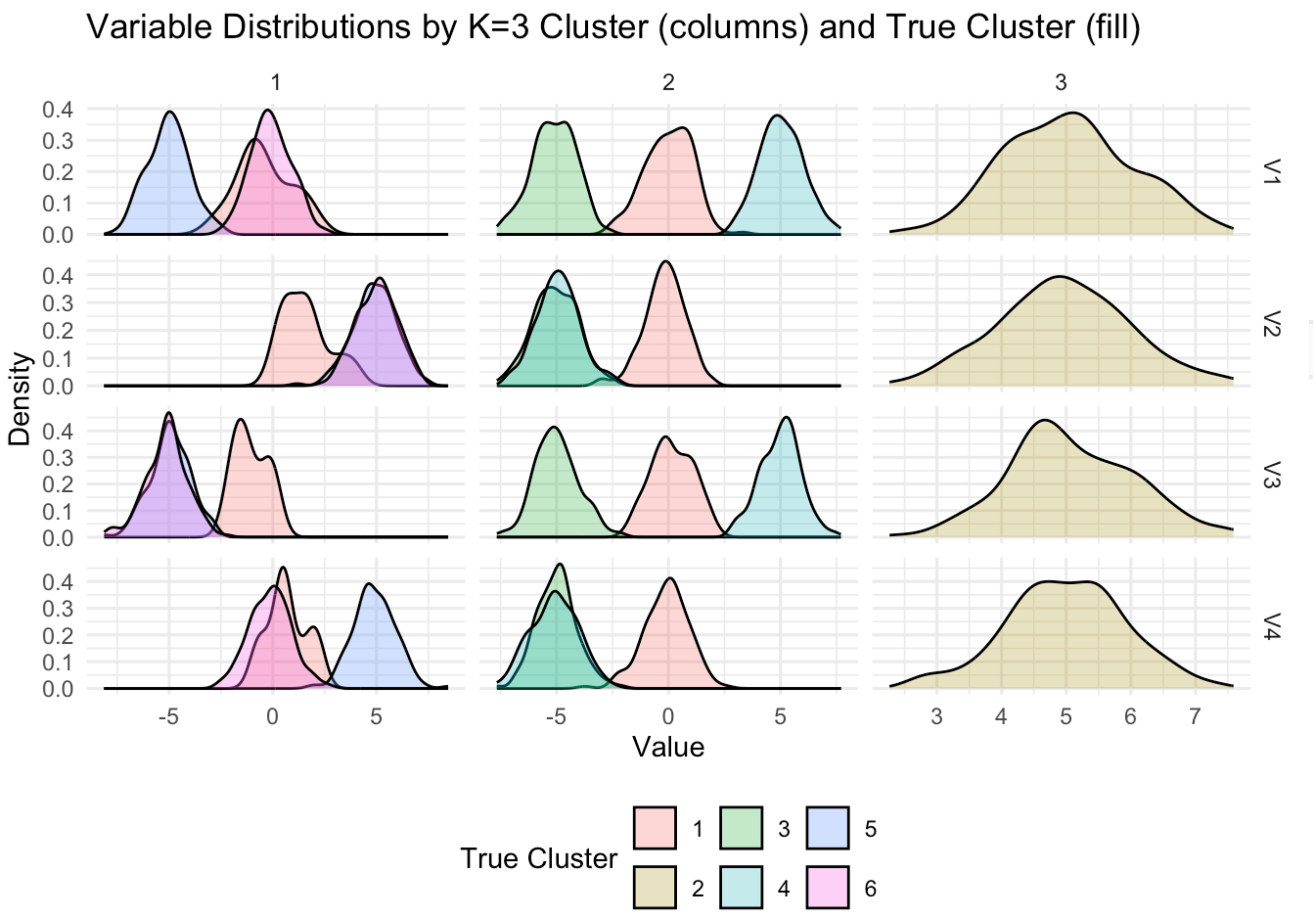
```
# Visualize predicted clusters (k = 3) by each variable
ggplot(
    pivot_longer(X[, c("V1", "V2", "V3", "V4", "k3_cluster")],
        cols = starts_with("V"),
        names_to = "variable",
        values_to = "value"
    ),
    aes(x = value, color = k3_cluster, fill = k3_cluster)
) +
    geom_density(alpha = 0.3) +
    facet_wrap(~variable, scales = "free") +
    labs(
        title = "Distributions of Variables by Predicted K=3 Clusters",
        x = "Value", y = "Density",
        color = "Predicted Cluster",
        fill = "Predicted Cluster"
    ) +
    theme_minimal() +
    theme(legend.position = "bottom")
```



It is difficult to know if we have effectively separated the clusters when plotting predicted clusters against features. Let's leverage the fact that we know the true clusters to examine how well our model has done. The code below does this, with each column containing a predicted cluster and each colour representing a 'true' cluster.

```
# Plot both true and predicted cluster distributions
ggplot(X %>%
    pivot_longer(
        cols = starts_with("V"),
        names_to = "variable",
        values_to = "value"
    ), aes(x = value, fill = true_cluster)) +
    geom_density(alpha = 0.3) +
    facet_grid(rows = vars(variable), cols = vars(k3_cluster), scales = "free") +
    labs(
```

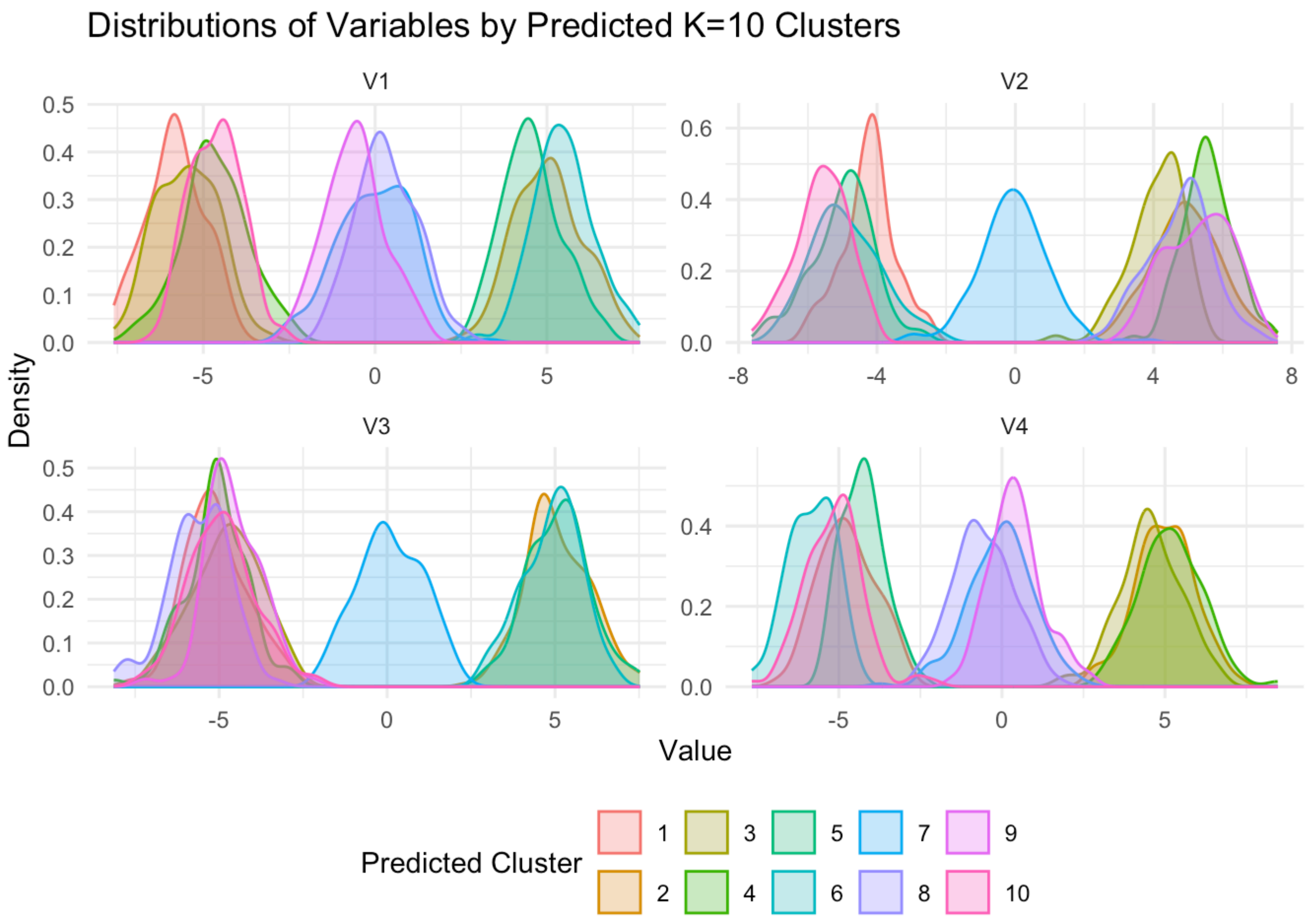
```r
        title = "Variable Distributions by K=3 Cluster (columns) and True Cluster (fil
        x = "Value", y = "Density",
        fill = "True Cluster"
    ) +
    theme_minimal() +
    theme(legend.position = "bottom")
```



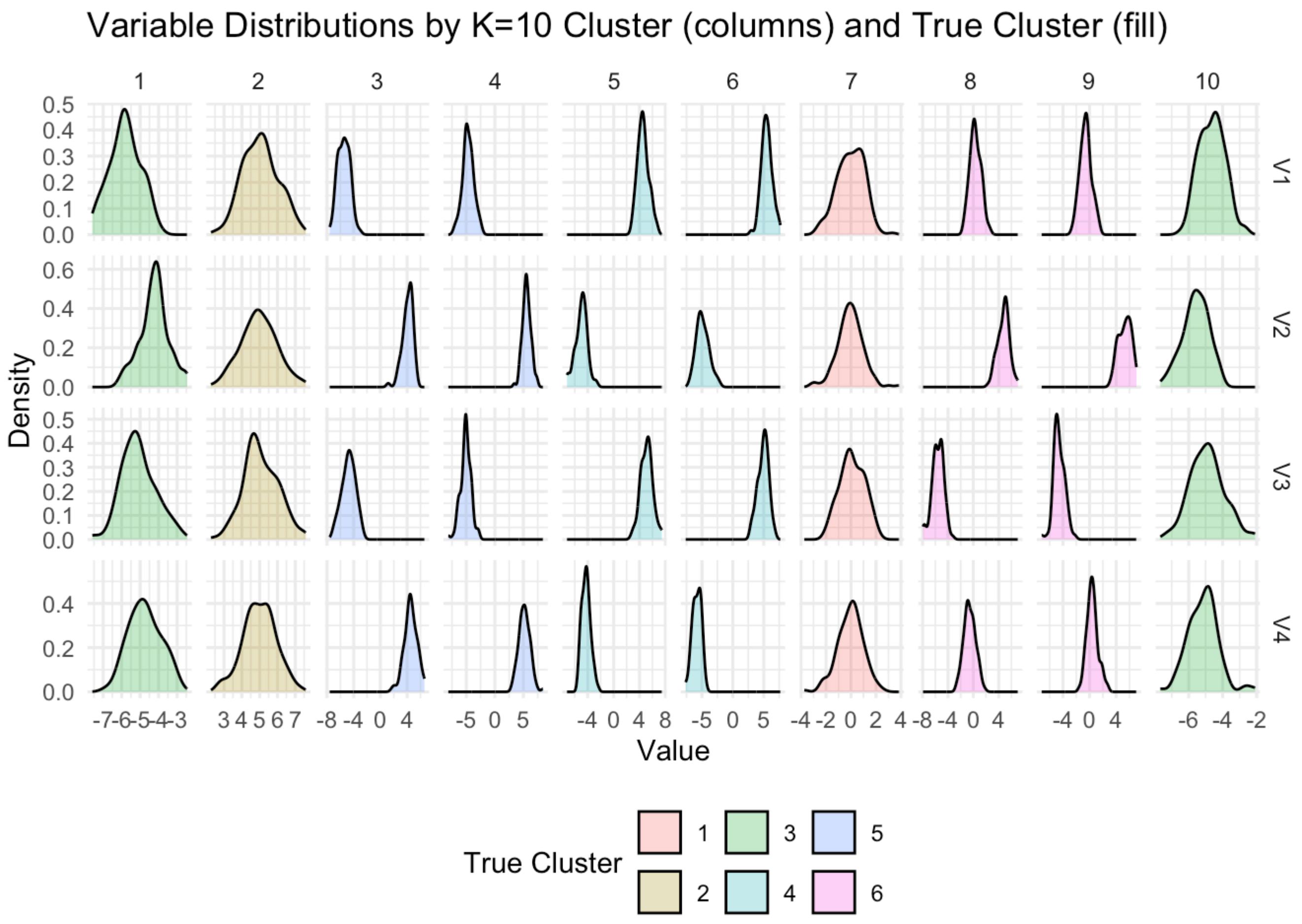
As we can see, our predicted clusters 1 and 2 still contain mixtures of true clusters. Upon inspection, we might want to increase the value of `k` to identify more clusters. Suppose your colleague says they think there are 10 clusters. In the following 3 code chunks, Re-run our k-means model with `k = 10,` assign the predicted cluster to a new `k10_cluster column,` and visualise it as above by plotting against both the features and the true clusters.

```r
# Run k-means with k = 10
set.seed(123)
k10_model <- kmeans(X[, 1:4], centers = 10, nstart = 50)

# Add predicted cluster assignments
X$k10_cluster <- factor(k10_model$cluster)
```

```r
# Visualize predicted clusters (k = 10) by each variable
ggplot(
    pivot_longer(X[, c("V1", "V2", "V3", "V4", "k10_cluster")],
        cols = starts_with("V"),
        names_to = "variable",
        values_to = "value"
    ),
    aes(x = value, color = k10_cluster, fill = k10_cluster)
) +
    geom_density(alpha = 0.3) +
    facet_wrap(~variable, scales = "free") +
    labs(
        title = "Distributions of Variables by Predicted K=10 Clusters",
        x = "Value", y = "Density",
        color = "Predicted Cluster",
        fill = "Predicted Cluster"
    ) +
    theme_minimal() +
    theme(legend.position = "bottom")
```



```
# Compare predicted k10 clusters against true clusters
ggplot(
    X %>%
        pivot_longer(
            cols = starts_with("V"),
            names_to = "variable",
            values_to = "value"
        ),
    aes(x = value, fill = true_cluster)
) +
    geom_density(alpha = 0.3) +
    facet_grid(rows = vars(variable), cols = vars(k10_cluster), scales = "free") +
    labs(
        title = "Variable Distributions by K=10 Cluster (columns) and True Cluster (fi
        x = "Value", y = "Density",
        fill = "True Cluster"
    ) +
    theme_minimal() +
    theme(legend.position = "bottom")
```



The last visualisation is particularly interesting. It looks as though we have **over-separated**, with some of the predicted clusters capturing only fragments of a true cluster. This highlights a key limitation of unsupervised learning: without ground-truth labels, it is difficult to determine the “right” number of clusters.

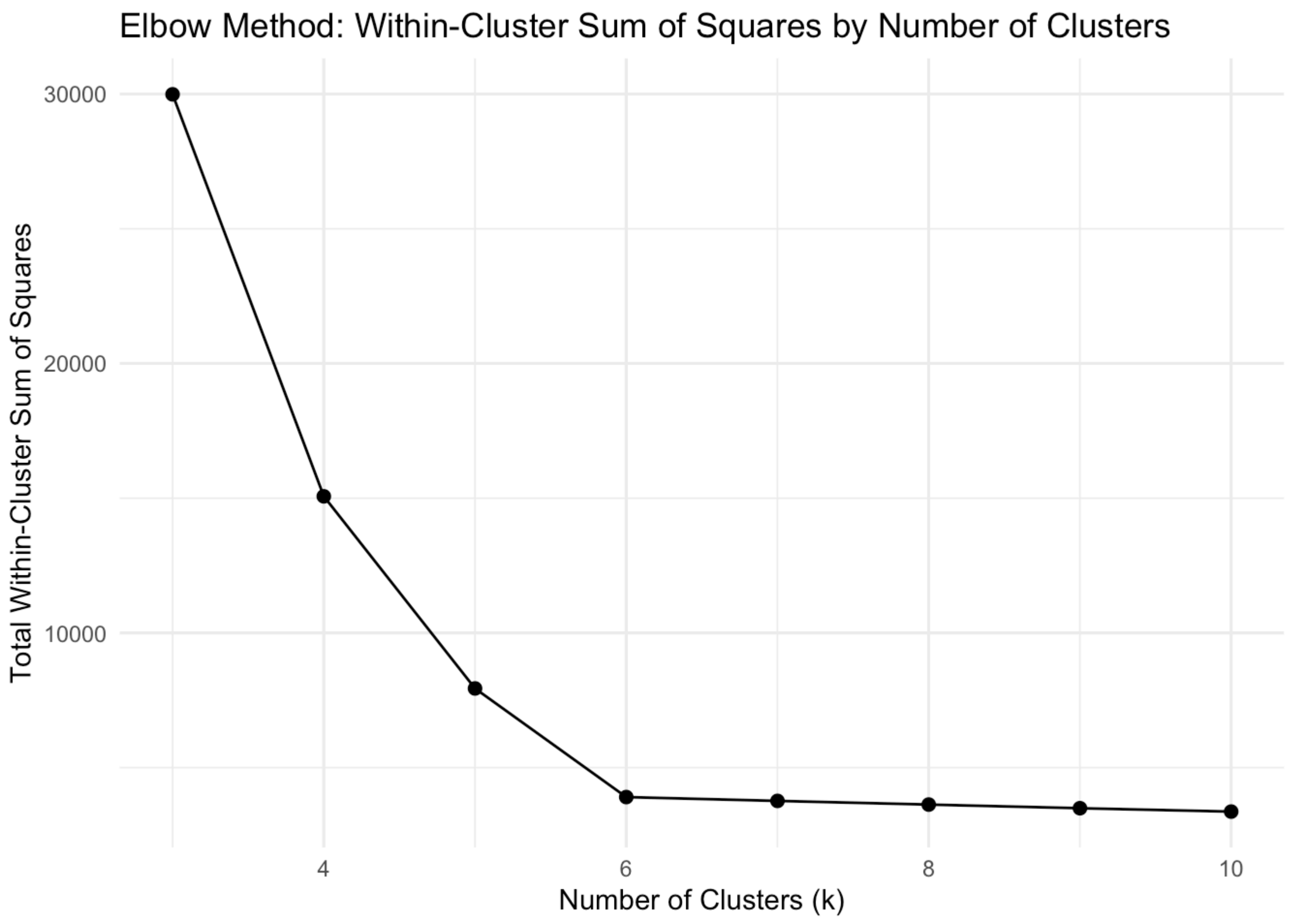
Choosing a larger value of k may produce tighter and more granular groupings, but these groupings are not guaranteed to correspond to meaningful or interpretable structure in the data. In fact, overfitting can occur when we mistake noise or local patterns for true underlying clusters.

To help address this challenge, we often use internal validation metrics to choose a suitable value for `k`. One common approach is the elbow method, which involves plotting the within-cluster sum of squares (WSS) across different values of `k`. A sharp drop followed by a “flattening” of the curve suggests a good balance between model complexity and fit. Now, we’re going to iterate through `k = 3` to `k = 10` and use the elbow method to guide our choice of the optimal number of clusters. We will then visualise the estimated WSS per value of k. This plot helps us visually determine whether there’s a clear “elbow” point — an indication of the most appropriate number of clusters to retain.

```r
# For scree plot (elbow method)
set.seed(123)

wss_vals <- sapply(3:10, function(k) {
    kmeans(X[, 1:4], centers = k, nstart = 50)$tot.withinss
})

# Plot the WSS for each k
elbow_df <- data.frame(
    k = 3:10,
    wss = wss_vals
)

head(elbow_df)
```

```
  k       wss
1 3 29988.683
2 4 15069.083
3 5  7938.546
4 6  3907.476
5 7  3765.119
6 8  3629.238
```

```r
ggplot(elbow_df, aes(x = k, y = wss)) +
    geom_line() +
    geom_point(size = 2) +
    labs(
        title = "Elbow Method: Within-Cluster Sum of Squares by Number of Clusters",
        x = "Number of Clusters (k)",
        y = "Total Within-Cluster Sum of Squares"
    ) +
    theme_minimal()
```

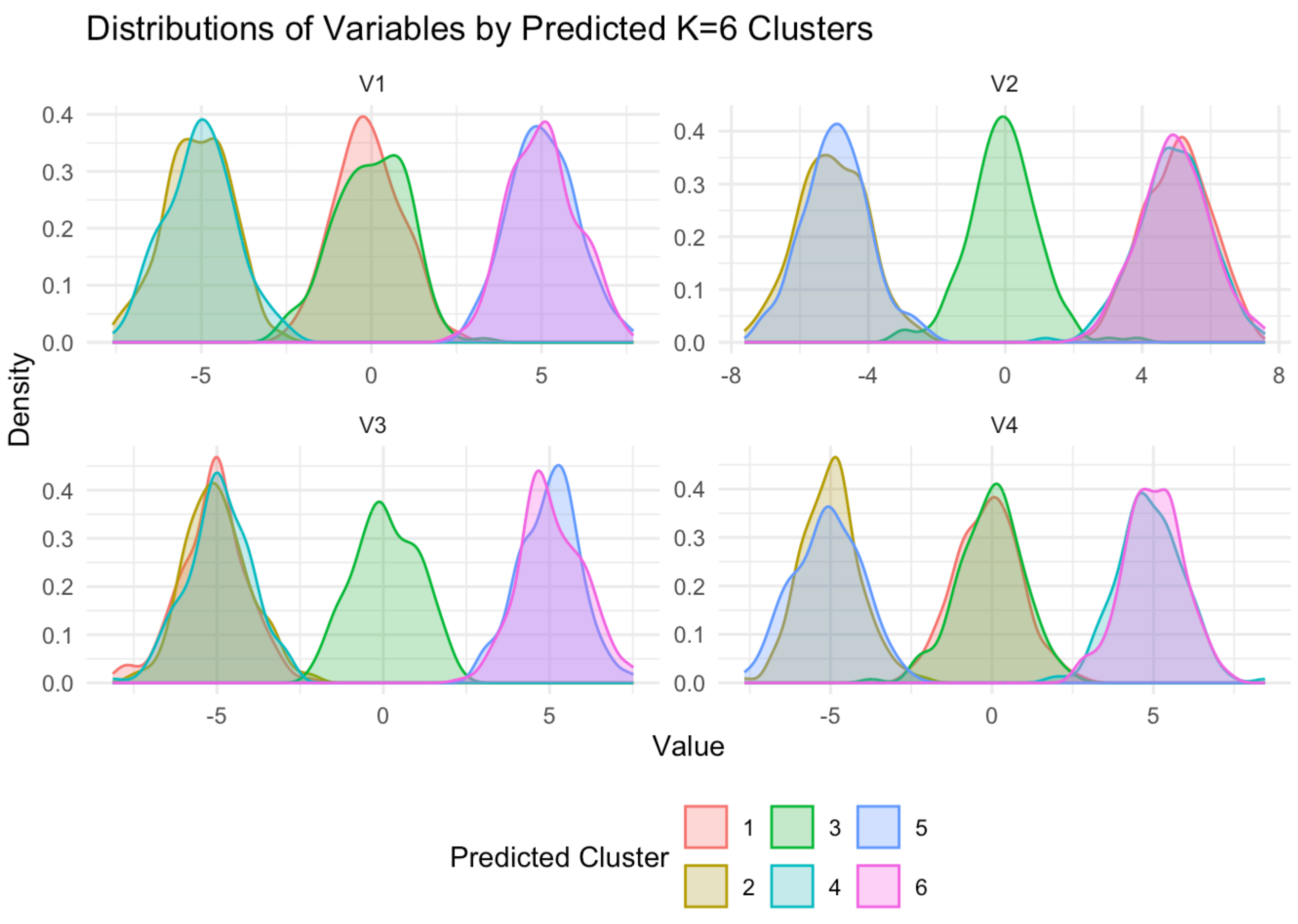


Questions:

- What is the optimal value of k according to the elbow method?
- What is the approximate WSS for the optimal value of k?

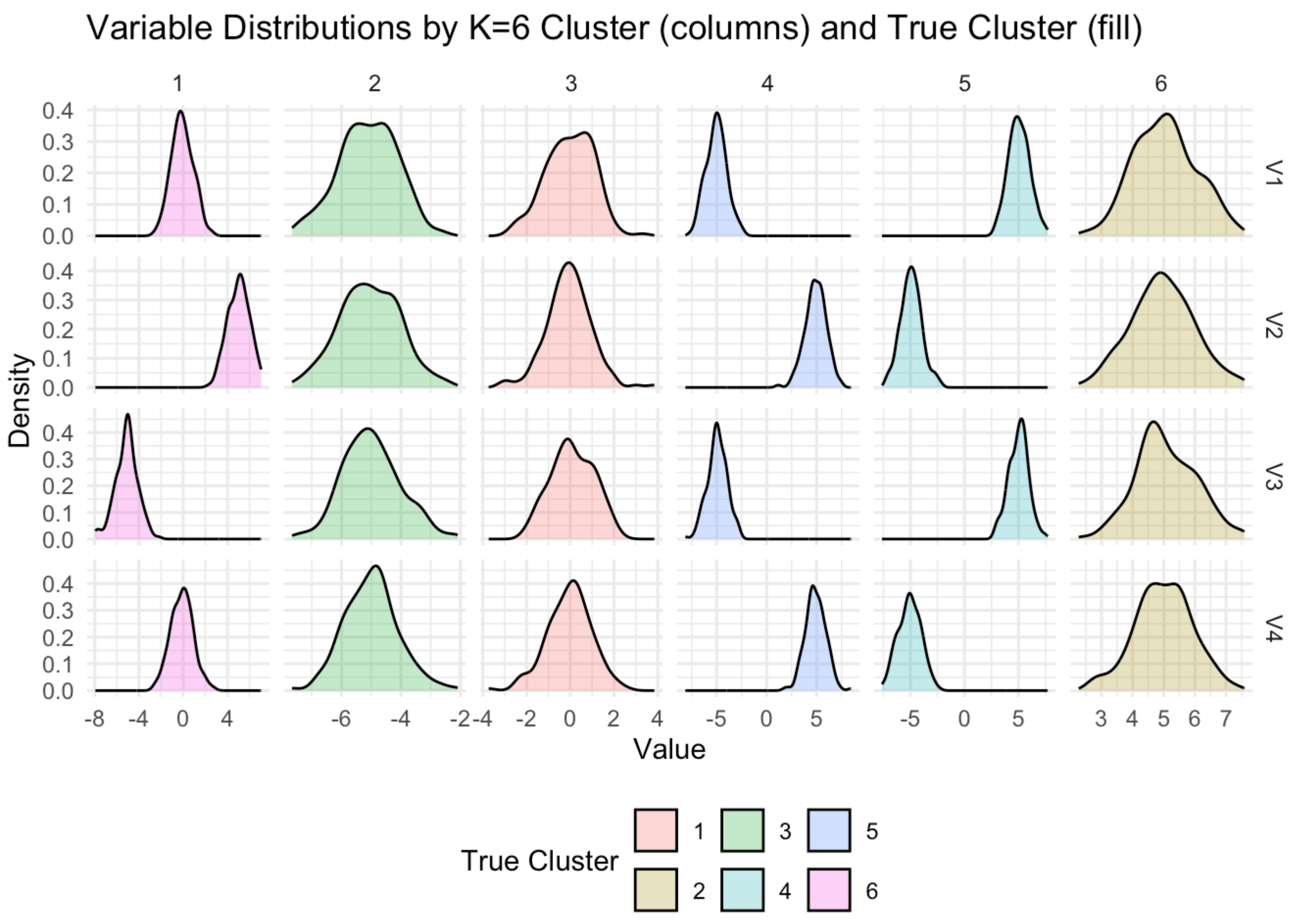
Now that the elbow method has suggested a reasonable choice for `k`, we'll fit a k-means clustering model with said number of clusters, assign the predicted cluster labels to a new column `k6_cluster,` and visualise the results.

```r
set.seed(123)

# Fit k−means with k = 6
k6_model <- kmeans(X[, 1:4], centers = 6, nstart = 50)

# Add predicted cluster labels to the data
X$k6_cluster <- factor(k6_model$cluster)

# Visualise predicted clusters by variable
ggplot(
    pivot_longer(X[, c("V1", "V2", "V3", "V4", "k6_cluster")],
        cols = starts_with("V"),
        names_to = "variable",
        values_to = "value"
    ),
    aes(x = value, color = k6_cluster, fill = k6_cluster)
) +
    geom_density(alpha = 0.3) +
    facet_wrap(~variable, scales = "free") +
    labs(
        title = "Distributions of Variables by Predicted K=6 Clusters",
        x = "Value", y = "Density",
        color = "Predicted Cluster",
        fill = "Predicted Cluster"
    ) +
    theme_minimal() +
    theme(legend.position = "bottom")
```



Now let's use the ground-truth labels to inspect how well the predicted clusters map onto the true clusters:

```
# Visualise true cluster composition within each predicted cluster
ggplot(
    X %>%
        pivot_longer(
            cols = starts_with("V"),
            names_to = "variable",
            values_to = "value"
        ),
    aes(x = value, fill = true_cluster)
) +
    geom_density(alpha = 0.3) +
    facet_grid(rows = vars(variable), cols = vars(k6_cluster), scales = "free") +
    labs(
        title = "Variable Distributions by K=6 Cluster (columns) and True Cluster (fil
        x = "Value", y = "Density",
        fill = "True Cluster"
    ) +
    theme_minimal() +
    theme(legend.position = "bottom")
```



Questions:

- Do any of our predicted clusters contain more than one true clusters?
- Do any of the true clusters contain more than one of our predicted clusters?

Part 2.2: Dimensionality Reduction

So far, we've explored clustering as a way to uncover hidden structure in our data. Now, we turn to another family of unsupervised machine learning techniques: dimensionality reduction. At its core, dimensionality reduction aims to find a lower-dimensional representation of high-dimensional data that preserves as much of the original structure or variance as possible. This is especially useful when working with complex datasets where visualising or interpreting patterns across many variables is difficult. By projecting the data into two or three dimensions, we can uncover underlying structure, visualise groupings or trends, and even improve the performance of downstream models by removing redundancy or noise.

To begin motivating this approach, we will return to our simulated data from Part 1. We've already seen that it can be challenging to visually inspect high-dimensional data—especially when cluster separation is not obvious from individual variables. Dimensionality reduction offers a solution by projecting the data into a lower-dimensional space (typically 2D or 3D), allowing us to visualise overall structure and potential clustering.

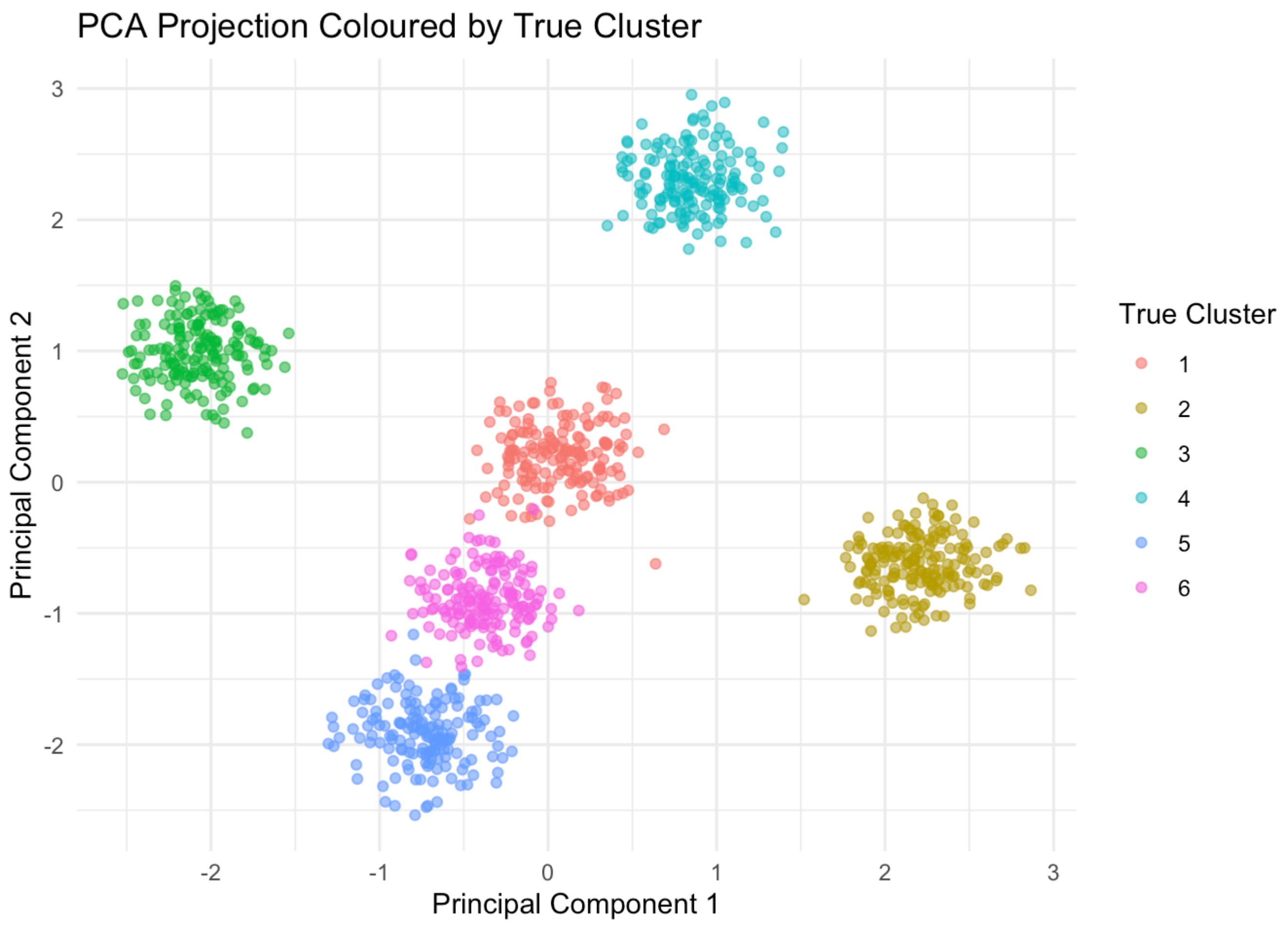
One of the most common techniques for this is Principal Component Analysis (PCA). PCA is a linear method that finds new axes (called principal components) that successively capture the greatest variance in the data. We can then plot the data on the first two principal components to see the dominant structure. In R, PCA can be performed using the `prcomp()` function from the `stats` package, which is loaded by default. Below, we apply PCA to our four simulated variables and visualise the results using `ggplot2`. These visualisations let us assess how well the true or predicted clusters are separated in the lower-dimensional space.

In the following code chunk, we perform PCA on our simulated data.

```r
# Perform PCA using base R's prcomp() function
# scale. = TRUE standardizes the variables before computing PCA
pca_result <- prcomp(X[, c("V1", "V2", "V3", "V4")], scale. = TRUE)

# Create a dataframe with the first two principal components and cluster labels
pca_df <- data.frame(
    PC1 = pca_result$x[, 1],
    PC2 = pca_result$x[, 2],
    true_cluster = X$true_cluster,
    k6_cluster = X$k6_cluster
)
```

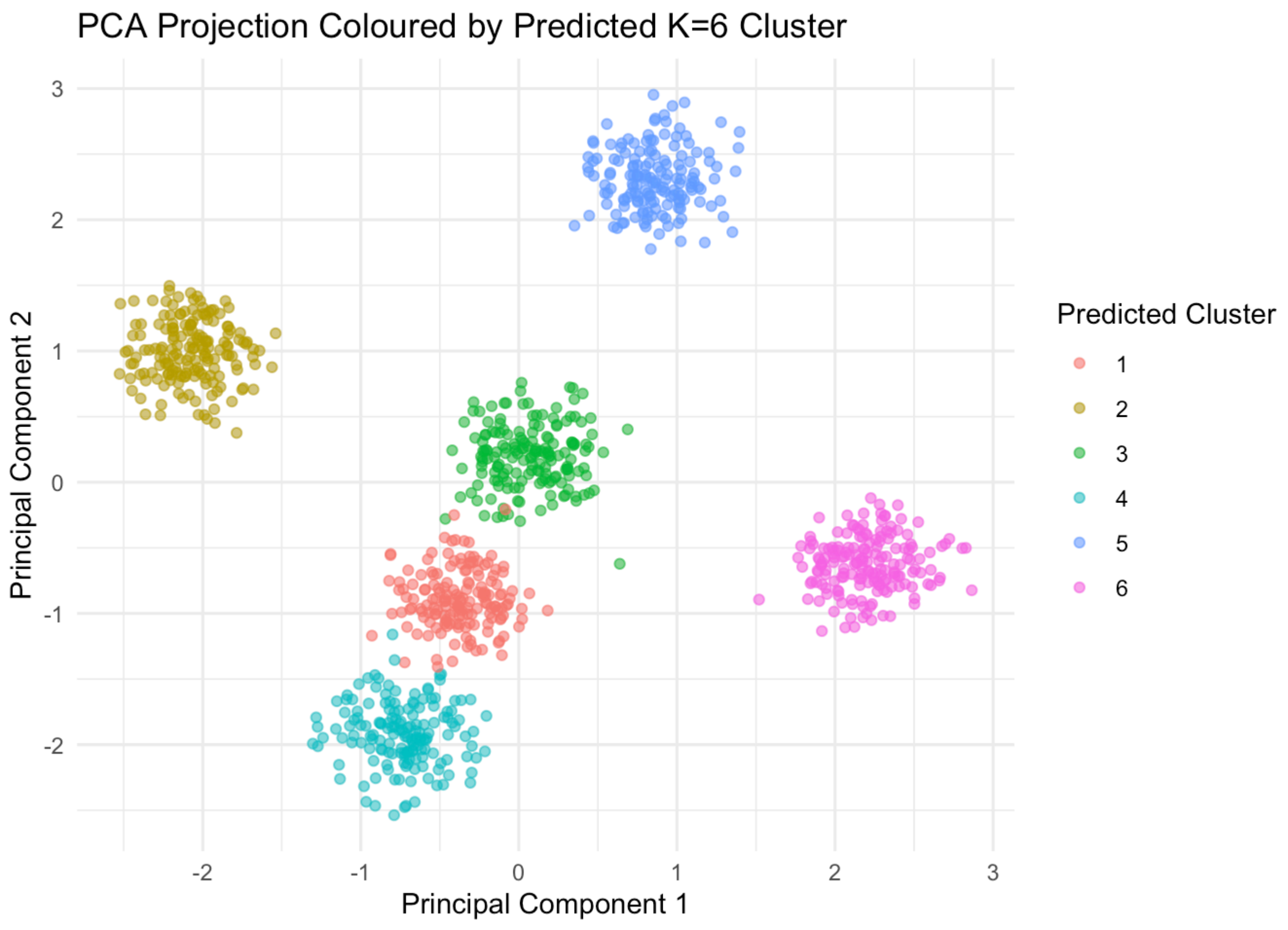
Next, we visualise our observations by how they are loaded on the two first principal components compared to their true cluster. What does this tell us?

```r
# Visualise PCA projection coloured by true cluster
ggplot(pca_df, aes(x = PC1, y = PC2, color = true_cluster)) +
    geom_point(alpha = 0.6) +
    labs(
        title = "PCA Projection Coloured by True Cluster",
        x = "Principal Component 1",
        y = "Principal Component 2",
        color = "True Cluster"
    ) +
    theme_minimal()
```



Next, we'll visualise our PCA loadings versus the predicted clusters.

```r
# Visualise PCA projection coloured by predicted cluster (k = 6)
ggplot(pca_df, aes(x = PC1, y = PC2, color = k6_cluster)) +
    geom_point(alpha = 0.6) +
    labs(
        title = "PCA Projection Coloured by Predicted K=6 Cluster",
        x = "Principal Component 1",
        y = "Principal Component 2",
        color = "Predicted Cluster"
    ) +
    theme_minimal()
```



Questions:

- How do the two graphs compare? What does this mean?
- Have we effectively separated the clusters?
- Does our dimensionality reduction effectively illustrate separation between clusters?

Now, let's apply PCA to a real-world dataset: the `USArrests` data. This dataset contains statistics on violent crime rates and urbanization for each US state in 1973. Since the variables are measured on different scales (e.g., murder rate vs. percent urban population), PCA will help us reduce dimensionality while capturing the most important variation in the data. This can reveal underlying patterns and relationships between states based on crime profiles. We can specify the `center` and `scale.` arguments in `prcomp()` to standardise the data by subtracting the mean and dividing by the standard deviation of each variable. This ensures that variables with larger numeric ranges do not dominate the principal components, allowing PCA to treat all variables equally regardless of their original scale.

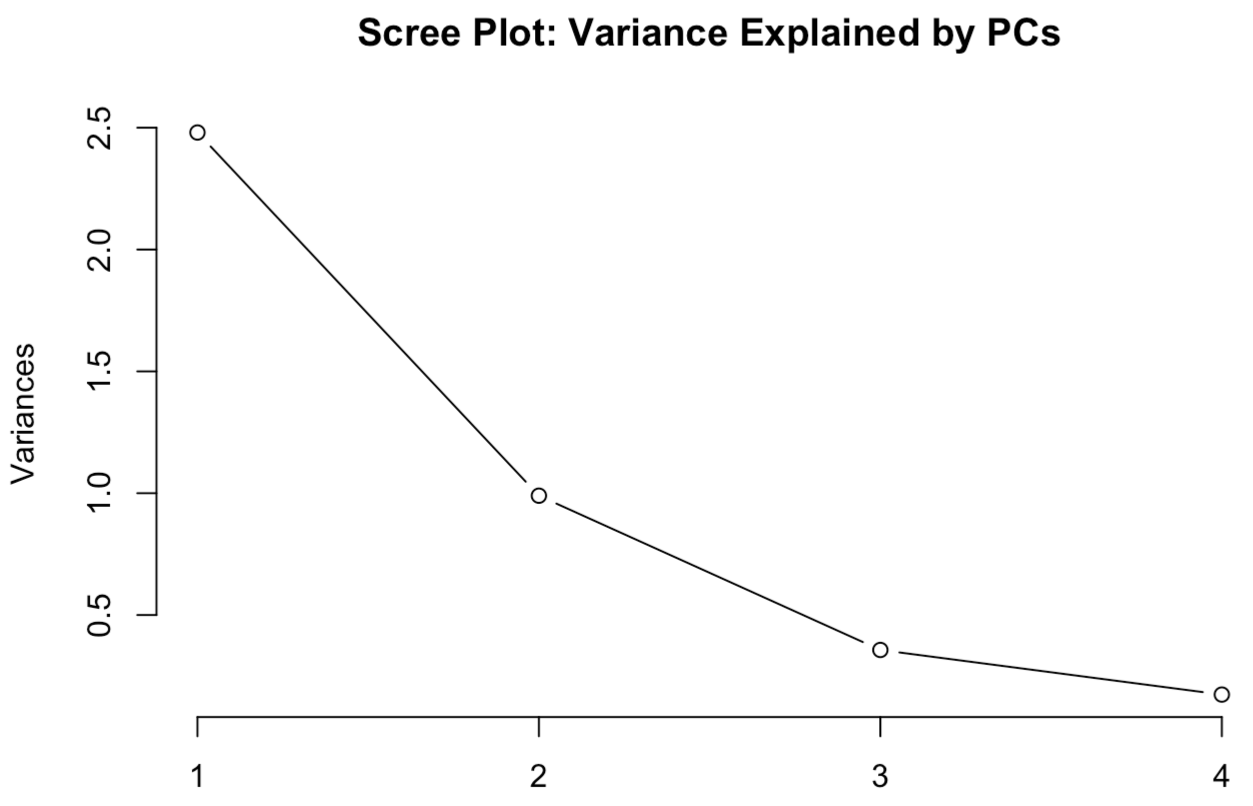
```r
# Load dataset
data("USArrests")

# Inspect data structure
str(USArrests)
```

```
'data.frame':	50 obs. of  4 variables:
 $ Murder  : num  13.2 10 8.1 8.8 9 7.9 3.3 5.9 15.4 17.4 ...
 $ Assault : int  236 263 294 190 276 204 110 238 335 211 ...
 $ UrbanPop: int  58 48 80 50 91 78 77 72 80 60 ...
 $ Rape    : num  21.2 44.5 31 19.5 40.6 38.7 11.1 15.8 31.9 25.8 ...
```

```
# Perform PCA with scaling (important since variables have different units)
pca_usa <- prcomp(USArrests, center = TRUE, scale. = TRUE)

# View summary of importance of components
summary(pca_usa)
```

```
Importance of components:
                          PC1    PC2     PC3     PC4
Standard deviation     1.5749 0.9949 0.59713 0.41645
Proportion of Variance 0.6201 0.2474 0.08914 0.04336
Cumulative Proportion  0.6201 0.8675 0.95664 1.00000
```

```
# Plot variance explained by components (scree plot)
plot(pca_usa, type = "l", main = "Scree Plot: Variance Explained by PCs")
```

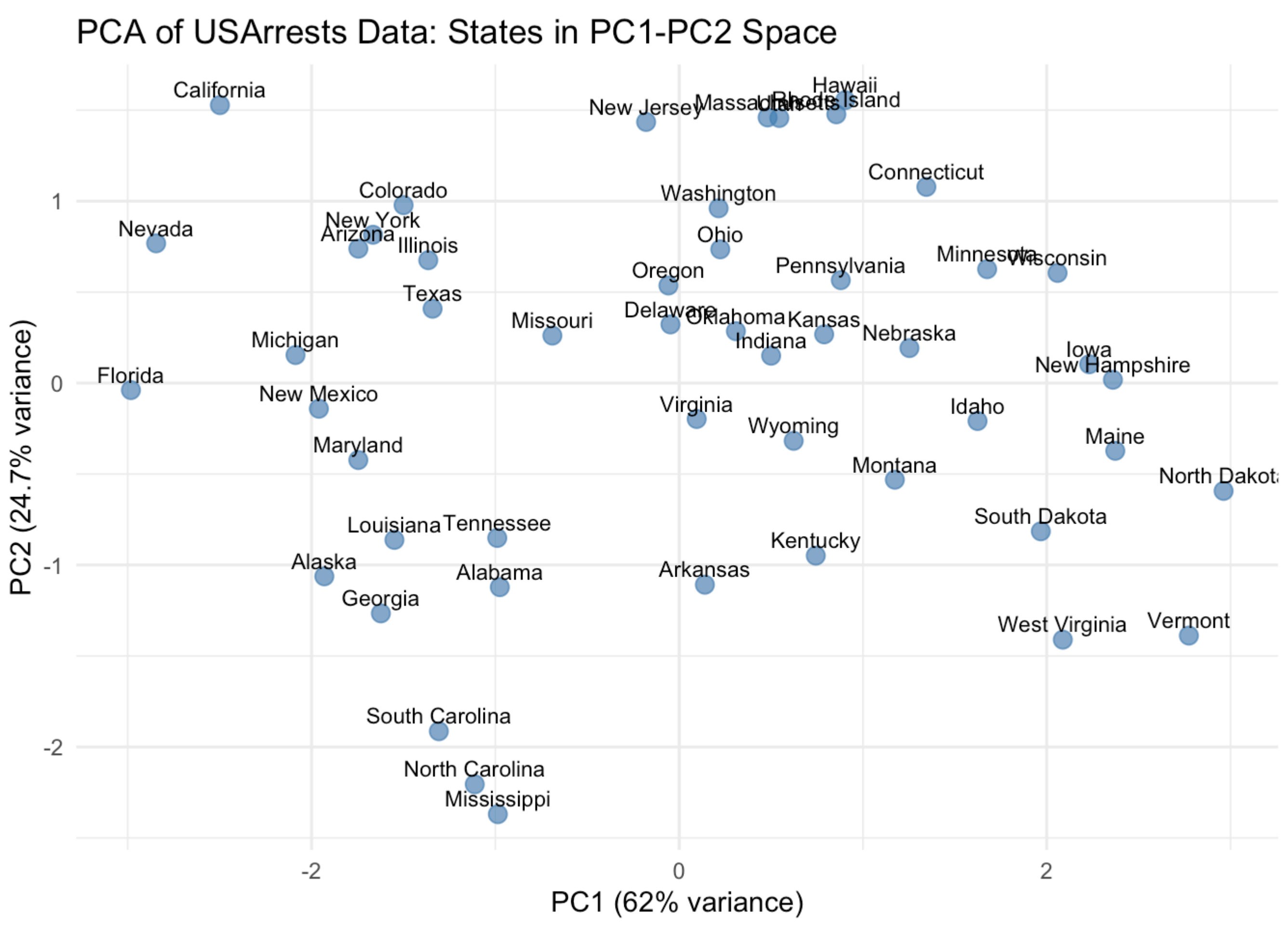


```
# Create a data frame with PC scores
pc_df <- as.data.frame(pca_usa$x)
pc_df$State <- rownames(USArrests)

pca_usa$rotation
```

```
                PC1        PC2        PC3         PC4
Murder   -0.5358995 -0.4181809  0.3412327  0.64922780
```

```
Assault  -0.5831836 -0.1879856  0.2681484 -0.74340748
UrbanPop -0.2781909  0.8728062  0.3780158  0.13387773
Rape     -0.5434321  0.1673186 -0.8177779  0.08902432
```

```r
ggplot(pc_df, aes(x = PC1, y = PC2, label = State)) +
    geom_point(color = "steelblue", size = 3, alpha = 0.7) +
    geom_text(vjust = -0.5, size = 3) +
    labs(
        title = "PCA of USArrests Data: States in PC1-PC2 Space",
        x = paste0("PC1 (", round(summary(pca_usa)$importance[2, 1] * 100, 1), "% vari
        y = paste0("PC2 (", round(summary(pca_usa)$importance[2, 2] * 100, 1), "% vari
    ) +
    theme_minimal()
```



Questions:

- How many principal components are there? Why can't we have more?
- Which states cluster together in this plot? What common characteristics might explain these groupings?
- How much variance is captured by PC1 and PC2? (*Hint*: check the scree plot or summary output.)
- What might the principal component loadings tell us about the combinations of crime rates and urbanization driving these patterns?

Part 3: Additional Exercises

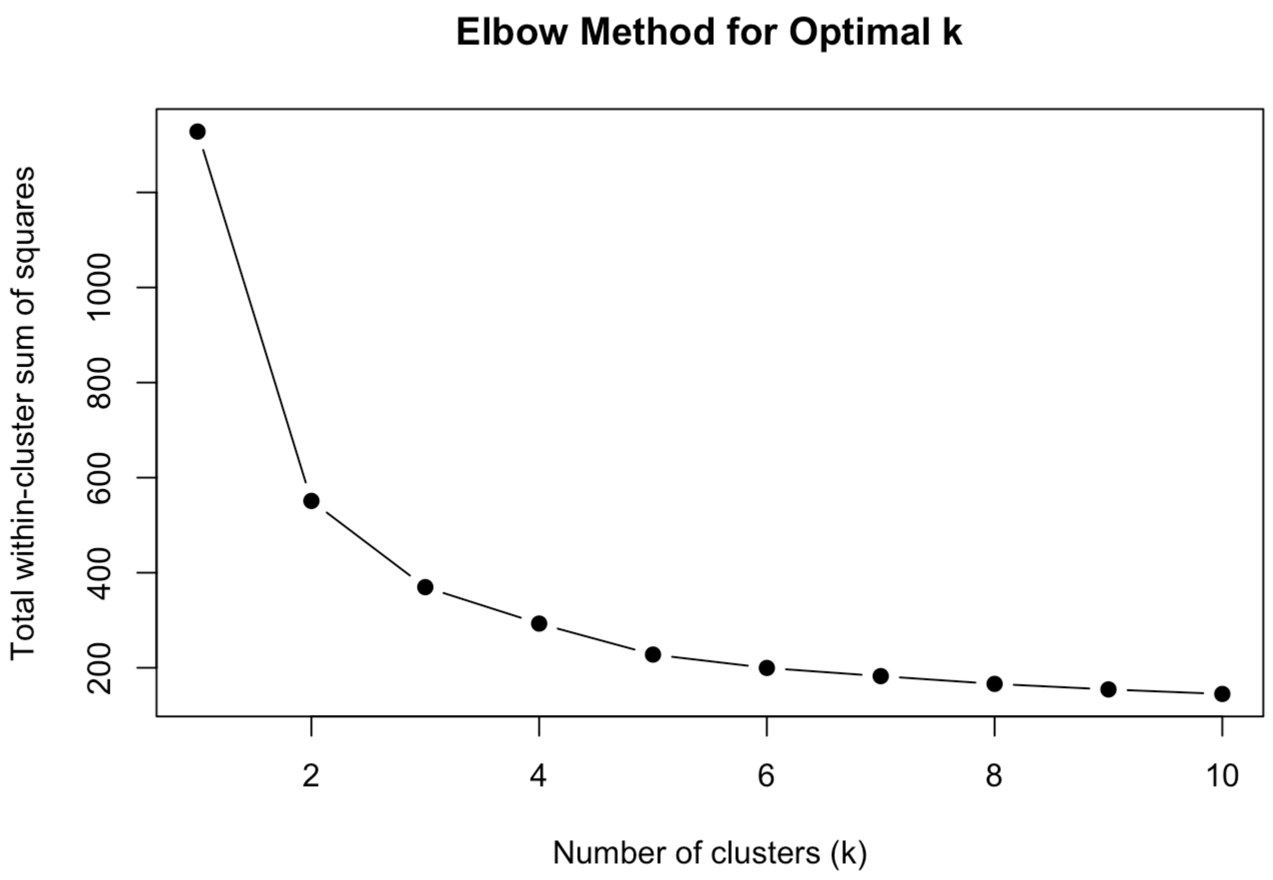
3.1: Applying K-means clustering to a real-world example

In this exercise, you'll apply K-means clustering to explore patterns in real-world data without labels. Use the `UCI Mall Customer` dataset to segment customers based on annual income and spending behavior. Your task is to use K-means clustering to identify customer segments. Visualize the resulting clusters and interpret what each cluster might represent in terms of marketing strategy.

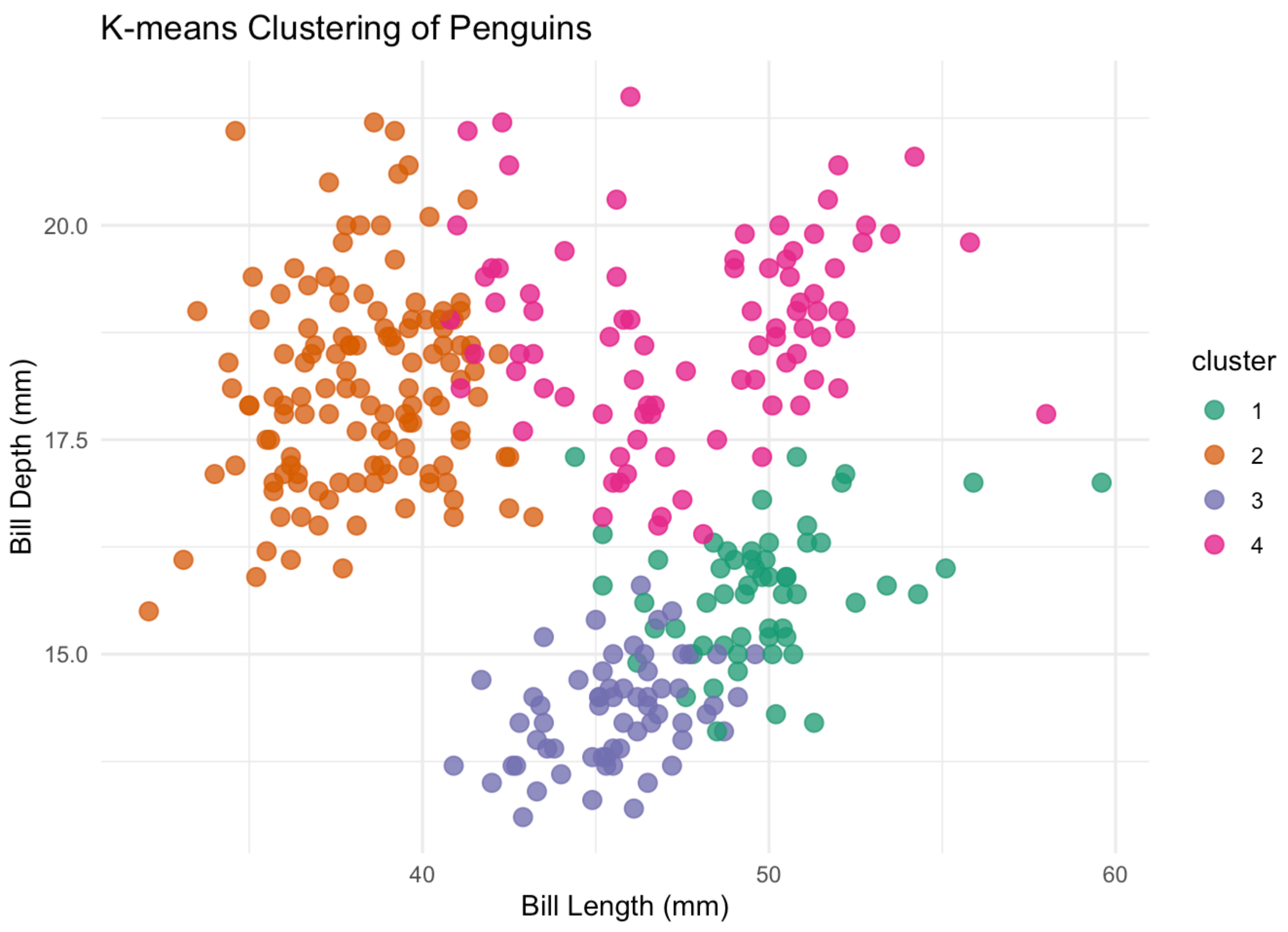
```
library(palmerpenguins)

# Load and clean the data
data <- penguins %>%
    drop_na() %>%
    select(bill_length_mm, bill_depth_mm, flipper_length_mm, body_mass_g)

# Scale the numeric features
data_scaled <- scale(data)

# Determine the optimal number of clusters using the elbow method
wss <- map_dbl(1:10, ~ kmeans(data_scaled, centers = .x, nstart = 25)$tot.withinss)

wss
```

```
 [1] 1328.0000  551.0113  369.6527  293.0222  227.8203  199.6812  182.4599
 [8]  166.1243  154.4656  144.9250
```

```
# Plot the elbow curve
plot(1:10, wss,
    type = "b", pch = 19,
    xlab = "Number of clusters (k)",
    ylab = "Total within-cluster sum of squares",
    main = "Elbow Method for Optimal k"
)
```



```
# Perform k-means clustering with chosen k (e.g., 3)
set.seed(123)

km <- kmeans(data_scaled, centers = 4, nstart = 50)

# Add cluster assignments to original data
data_clustered <- data %>%
    mutate(cluster = factor(km$cluster))

# Visualize clusters using first two features
ggplot(data_clustered, aes(x = bill_length_mm, y = bill_depth_mm, color = cluster)) +
    geom_point(size = 3, alpha = 0.8) +
    labs(
        title = "K-means Clustering of Penguins",
        x = "Bill Length (mm)",
        y = "Bill Depth (mm)"
    ) +
    theme_minimal() +
    scale_color_brewer(palette = "Dark2")
```



3.2: Applying PCA to a real-world example

In this exercise, you'll apply PCA to reduce the `Wine` dataset from UCI ML Repository to two principal components. Visualise the transformed data and assess wherether PCA separates the wine classes meaningfully. How many principal components would you use here? This would depend on how much variance is necessary to capture!

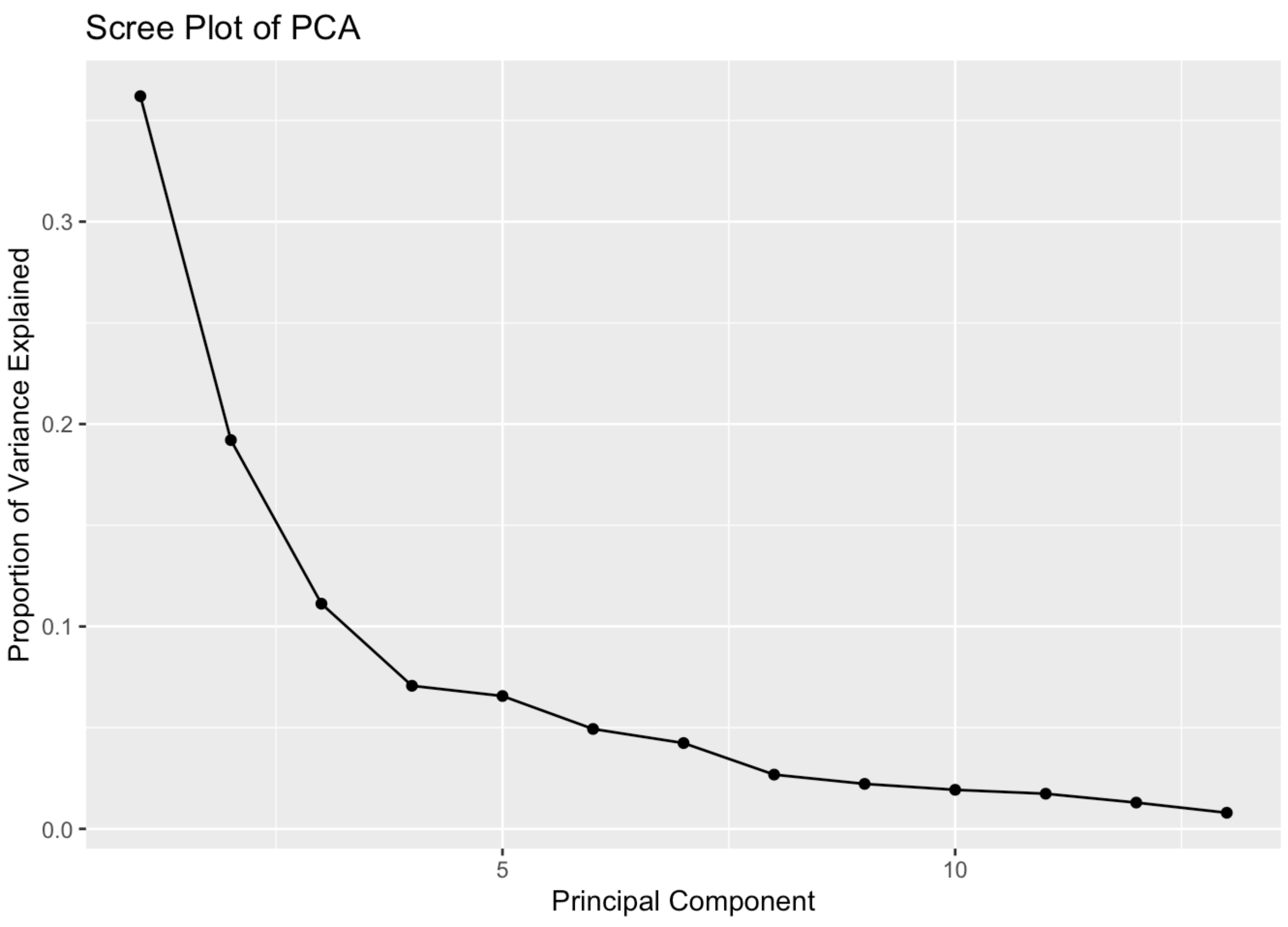
```
# Load required libraries
library(rattle) # for wine dataset
library(FactoMineR) # for PCA
library(factoextra) # for visualization

# Load the wine data
data(wine, package = "rattle")

# Inspect the dataset
head(wine)
```

```
  Type Alcohol Malic  Ash Alcalinity Magnesium Phenols Flavanoids Nonflavanoids
1    1   14.23  1.71 2.43       15.6       127    2.80       3.06          0.28
2    1   13.20  1.78 2.14       11.2       100    2.65       2.76          0.26
3    1   13.16  2.36 2.67       18.6       101    2.80       3.24          0.30
4    1   14.37  1.95 2.50       16.8       113    3.85       3.49          0.24
5    1   13.24  2.59 2.87       21.0       118    2.80       2.69          0.39
6    1   14.20  1.76 2.45       15.2       112    3.27       3.39          0.34
  Proanthocyanins Color  Hue Dilution Proline
```

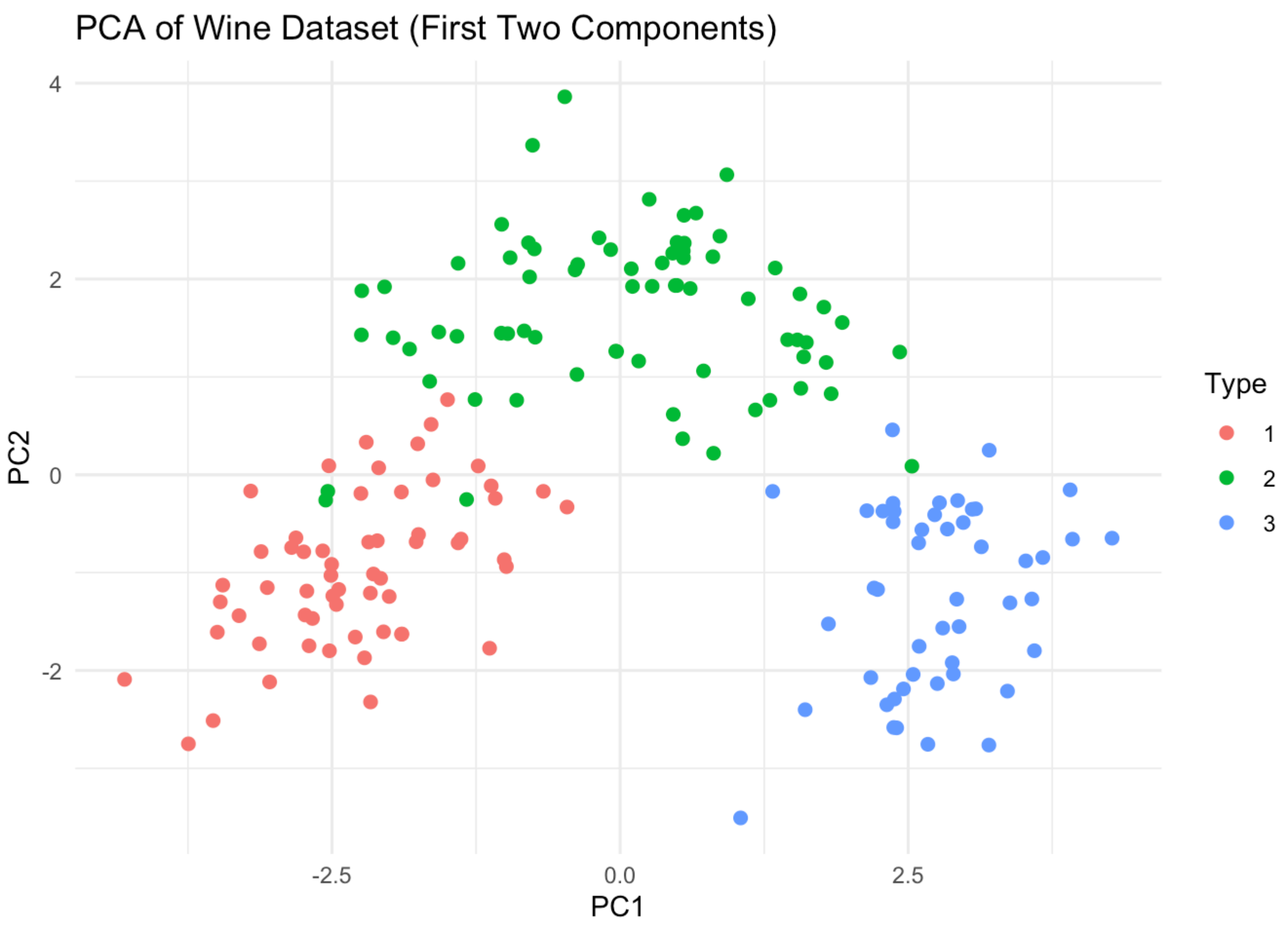
```
1            2.29  5.64 1.04     3.92    1065
2            1.28  4.38 1.05     3.40    1050
3            2.81  5.68 1.03     3.17    1185
4            2.18  7.80 0.86     3.45    1480
5            1.82  4.32 1.04     2.93     735
6            1.97  6.75 1.05     2.85    1450
```

```r
# Separate features and target
wine_data <- wine[, -1] # remove the first column (class label)
wine_labels <- wine$Type

# Standardise the data before PCA
wine_scaled <- scale(wine_data)

# Apply PCA
pca_result <- prcomp(wine_scaled)

# Create a dataframe of the first two PCs
pca_df <- as.data.frame(pca_result$x[, 1:2])
pca_df$Type <- wine_labels


# Scree plot: proportion of variance explained
var_explained <- pca_result$sdev^2 / sum(pca_result$sdev^2)

qplot(1:length(var_explained), var_explained, geom = "line") +
    geom_point() +
    labs(x = "Principal Component", y = "Proportion of Variance Explained") +
    ggtitle("Scree Plot of PCA")
```

```
Warning: `qplot()` was deprecated in ggplot2 3.4.0.
```



```r
# Plot the first two principal components
ggplot(pca_df, aes(x = PC1, y = PC2, color = Type)) +
    geom_point(size = 2) +
    labs(
        title = "PCA of Wine Dataset (First Two Components)",
        x = "PC1", y = "PC2"
    ) +
    theme_minimal()
```



3.3: Build your own ensemble pipeline

In this exercise, you'll implement an ensemble model by combining multiple base learners using `scikit-learn`'s `VotingClassifier` or `StackingClassifier`. Use the `Heart Disease` dataset to predict whether a patient is at risk of heart disease. Your task is to build three individual models (e.g., logistic regression, decision tree, and support vector machine), and then combine them into an ensemble. Compare the ensemble's performance to each individual model using ROC-AUC.

```python
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import StackingClassifier
from sklearn.metrics import roc_auc_score

# Load the heart disease dataset from OpenML
heart = fetch_openml(name='heart-disease', version=1, as_frame=True)
df = heart.frame

# Define features and target
X = df.drop(columns=['target'])
y = df['target'].astype(int)

# Encode categorical columns if any
for col in X.select_dtypes(include=['category', 'object']):
    X[col] = pd.Categorical(X[col]).codes

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Base models
estimators = [
    ('lr', LogisticRegression(max_iter=1000, solver='liblinear')),
    ('dt', DecisionTreeClassifier(random_state=42)),
    ('svc', SVC(probability=True, kernel='rbf', random_state=42))
]

# Stacking classifier with logistic regression as final estimator
stacking_clf = StackingClassifier(
    estimators=estimators,
    final_estimator=LogisticRegression(max_iter=1000),
    passthrough=False,
    cv=5
)

# Train base models individually to compare
for name, model in estimators:
    model.fit(X_train, y_train)
```

```
LogisticRegression(max_iter=1000, solver='liblinear')
DecisionTreeClassifier(random_state=42)
SVC(probability=True, random_state=42)
```

```
# Predict probabilities for individual base models
y_prob_individual = {}
for name, model in estimators:
    y_prob_individual[name] = model.predict_proba(X_test)[:, 1]

# Fit stacking classifier
stacking_clf.fit(X_train, y_train)
```

```
StackingClassifier(cv=5,
                   estimators=[('lr',
                                 LogisticRegression(max_iter=1000,
                                                    solver='liblinear')),
                                ('dt', DecisionTreeClassifier(random_state=42)),
                                ('svc', SVC(probability=True, random_state=42))],
                   final_estimator=LogisticRegression(max_iter=1000))
```

```
y_prob_stack = stacking_clf.predict_proba(X_test)[:, 1]

# Evaluate individual models
print("Individual Model ROC-AUC Scores:")
```

```
Individual Model ROC-AUC Scores:
```

```
for name, prob in y_prob_individual.items():
    print(f"{name}: {roc_auc_score(y_test, prob):.3f}")
```

```
lr: 0.862
dt: 0.726
svc: 0.741
```

```
# Evaluate stacking ensemble
print(f"\nStacking Ensemble ROC-AUC Score: {roc_auc_score(y_test, y_prob_stack):.3f}")
```

```

Stacking Ensemble ROC-AUC Score: 0.861
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

plt.figure(figsize=(8, 6))
```

```
<Figure size 800x600 with 0 Axes>
```

```
# Plot ROC for individual base models
for name, prob in y_prob_individual.items():
    fpr, tpr, _ = roc_curve(y_test, prob)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, lw=2, label=f'{name} (AUC = {roc_auc:.3f})')
```

```
[<matplotlib.lines.Line2D object at 0x29a76b910>]
[<matplotlib.lines.Line2D object at 0x2a0553710>]
[<matplotlib.lines.Line2D object at 0x2a056c990>]
```

```
# Plot ROC for stacking ensemble
fpr_stack, tpr_stack, _ = roc_curve(y_test, y_prob_stack)
roc_auc_stack = auc(fpr_stack, tpr_stack)
plt.plot(fpr_stack, tpr_stack, color='black', lw=3, linestyle='--', label=f'Stacking E
```

```
[<matplotlib.lines.Line2D object at 0x29a9cf550>]
```

```
plt.plot([0, 1], [0, 1], color='grey', lw=1, linestyle='--')
```

```
[<matplotlib.lines.Line2D object at 0x29a98a810>]
```

```
plt.xlim([0.0, 1.0])
```

```
(0.0, 1.0)
```

```
plt.ylim([0.0, 1.05])
```

```
(0.0, 1.05)
```

```
plt.xlabel('False Positive Rate')
```

```
Text(0.5, 0, 'False Positive Rate')
```

```
plt.ylabel('True Positive Rate')
```

```
Text(0, 0.5, 'True Positive Rate')
```

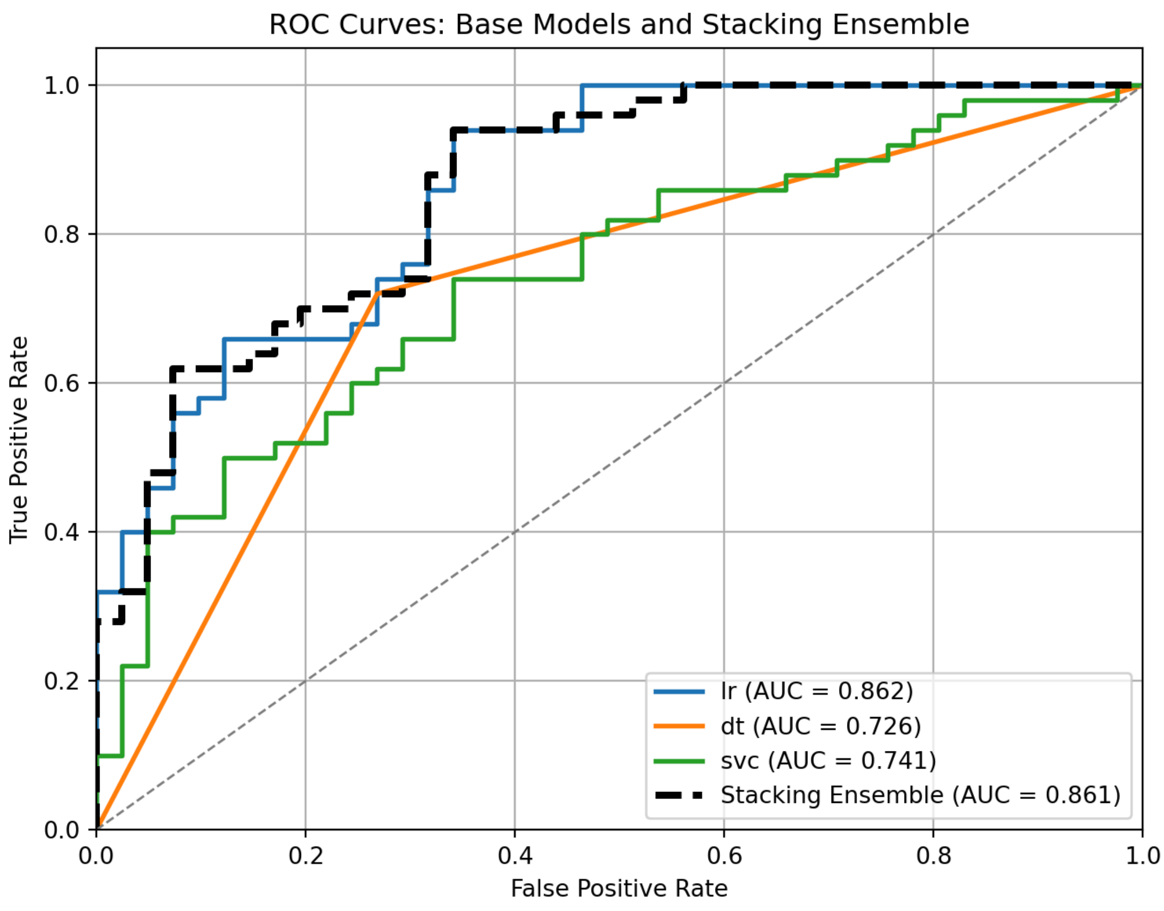
```
plt.title('ROC Curves: Base Models and Stacking Ensemble')
```

```
Text(0.5, 1.0, 'ROC Curves: Base Models and Stacking Ensemble')
```

```
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend object at 0x29a9d9d50>
```

```
plt.grid(True)
plt.show()
```



References

van der Laan, M. J., Polley, E. C., & Hubbard, A. E. (2007). Super learner. Statistical applications in genetics and molecular biology, 6, Article25. https://doi.org/10.2202/1544-6115.1309

By the end of this seminar, you should be able to:

- Understand the motivation and benefits of ensemble learning methods like stacking.
- Implement multiple base classifiers and combine them into an ensemble using `scikit-learn`.
- Perform unsupervised clustering using K-means to identify meaningful groups in unlabeled data.
- Visualise clustering results and principal components to interpret patterns and structure in data.
- Use dimensionality reduction techniques like PCA to simplify complex datasets while retaining important variance.
- Integrate both supervised and unsupervised learning approaches to comprehensively analyze data.