Week 10 Lecture Code Explanation

Visualization of High-Dimensional Data

I introduced two methods to visualize high-dimensional data: the *tour* and the *scatter plot matrix*.

For the tour example, here's the code:

```
library(langevitour)
library(tourr)
langevitour(olive[, -c(1, 2)] %>% scale(),
            group = c("North", "South", "Sardinia")[olive$region],
            pointSize = 2,
            levelColors = RColorBrewer::brewer.pal(3, "Dark2"))
```

`langevitour` is the main function from the `langevitour` package. The `olive` dataset comes from the `tourr` package, so we also load `tourr`. The first argument of `langevitour` is the dataset or matrix we want to visualize, where variables are columns and observations are rows. In this example, we remove the first two columns from `olive` using `[, -c(1, 2)]`, and then pass the result to the `scale` function to standardize the variables.

The `group` argument specifies the grouping for the visualization, similar to the `color` argument in `ggplot`. The `pointSize` argument controls the size of the points in the animation. Additionally, we use the `levelColors` argument to customize the colors of the data points, selecting three colors from the `RColorBrewer` package's `Dark2` palette.

For the scatter plot matrix example, here's the code:

```
olive[, -c(1, 2)] %>%
  scale() %>%
  as.data.frame() %>%
  mutate(region = c("North", "South", "Sardinia")[olive$region]) %>%
  GGally::ggscatmat(alpha = 0.1, color = "region") +
  theme_light() +
```

```
  xlab("") +
  ylab("") +
  theme(legend.position = "none") +
  scale_color_brewer(palette = "Dark2")
```

After we `scale` the dataset, it returns a matrix instead of a data frame, which is not compatible with the `ggscatmat` function from the `GGally` package—it only accepts `data.frame` objects. To fix this, we use `as.data.frame` to convert the matrix back into a data frame. Next, we add the `region` column back, replacing the original region codes (1, 2, 3) with more readable labels: "North," "South," and "Sardinia."

Why did we drop the `region` column initially and add it back later? The `scale` function only works with numerical columns. If it detects any non-numeric columns, like `region`, it will throw an error.

Once we have our new `data.frame`, we can use it in the `ggscatmat` function. To address the issue of overplotting, we set `alpha` to 0.1 for transparency. We also specify that we want to group by `region` for coloring. The default `ggscatmat` output is a bit plain, so I apply the `theme_light()` theme and remove the x and y axis labels. I also hide the legend, since the grouping is already demonstrated in the tour example.

Finally, we apply the `scale_color_brewer` function to use the `Dark2` color palette, ensuring consistency with the tour example.

Try K-means on `olive`

In slide 27, there's an example of applying K-means clustering on the `olive` dataset:

```
set.seed(12345)
result <- kmeans(scale(olive[, -c(1, 2)]),
                 centers = 3)
table(true = olive$region,
      kmeans = result$cluster)
table(true = olive$region,
      kmeans = c(2, 3, 1)[result$cluster])
```

K-means relies on the random placement of the initial cluster centers, so it's important to use `set.seed` to control this randomness and ensure reproducible results.

Since the variables in the `olive` dataset are measured on different scales, we need to standardize them using the `scale` function. Also, because K-means uses Euclidean distance, it requires all columns to be numerical, which is why we exclude the first two columns (`[, -c(1, 2)]`).

The `region` column, which indicates the true regions, is non-numeric, and therefore must be dropped before running K-means.

K-means requires us to specify the number of clusters in advance, so we set `centers = 3`, aiming for a 3-cluster solution. Once we store the output in `result`, we can use a two-way table to compare the K-means clusters with the true regions. The cluster labels are stored in `result$cluster`, and we provide two vectors to the `table` function: one for the true regions and the other for the K-means clusters. We name these vectors `true` and `kmeans` so that the table output clearly labels the rows and columns.

The initial `table` output provides a comparison, but you might notice that, for example, most observations from region 3 are assigned to cluster 2. To improve readability, it's common to reorder the cluster labels so that the larger values appear on the diagonal of the table. For instance, we can reassign cluster 2 as cluster 3, cluster 3 as cluster 1, and cluster 1 as cluster 2. This rearrangement puts the larger numbers on the diagonal, making the table easier to interpret. You can achieve this relabeling with `c(2, 3, 1)[result$cluster]`.

Now, the table is more intuitive: region 1 is mostly split between clusters 1 and 2, region 2 corresponds primarily to cluster 3, and region 3 also mostly aligns with cluster 3. This indicates that K-means splits region 1 into two groups while recognizing regions 2 and 3 as largely belonging to the same cluster.

Check group means

In slide 28, there's an example showing how to check the means of each cluster and see if they are well separated in high-dimensional space:

```
as_tibble(result$centers) %>%
  mutate(cluster = factor(1:nrow(result$centers))) %>%
  pivot_longer(-cluster, names_to = "variable", values_to = "mean") %>%
  ggplot() +
  geom_line(aes(variable, mean, group = cluster, col = cluster)) +
  theme_light() +
  scale_color_brewer(palette = "Dark2") +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1))
```

In the K-means result, the `centers` field is a matrix, not a `data.frame`, so we need to convert it into a `tibble` or `data.frame` to use with `dplyr`. The `centers` matrix has `k` rows and `p` columns, where `k` is the number of clusters, and `p` is the number of variables used in K-means. Each row represents the means for a particular cluster. After converting it to a `tibble` with `as_tibble`, we also need to add an indicator for each row to show which cluster it represents. We do this using `mutate(cluster = factor(1:nrow(result$centers)))`. Alternatively, you could use `mutate(cluster = factor(1:n()))`.

Next, since we want the variables on the x-axis and the means on the y-axis, and our data has variables as columns, we pivot the `tibble` into a longer format. We exclude the `cluster` column from the pivot, as it's needed to identify the cluster means. The `pivot_longer` function collects all the variables into a new column called `variable`, and their values into another new column called `mean`. This creates a three-column `tibble`, where each row represents the mean of one variable for one cluster.

We then pass this long-form `tibble` into `ggplot`. Instead of using `geom_point`, we use `geom_line` to connect the means, making it easier to visualize the relationships between them. This type of plot is also known as a parallel coordinates plot (https://en.wikipedia.org/wiki/Parallel_coordinates). In `geom_line`, we map the x-axis to `variable`, the y-axis to `mean`, and group and color the lines by `cluster`. Even though the `col` argument already groups the data by cluster, we also need the `group` argument, because in `geom_line`, grouping defines which data points should be connected, and we want points from the same cluster to be connected.

Finally, we enhance the plot by applying the `theme_light()` theme and the `Dark2` color palette from `RColorBrewer`. To avoid overlapping axis labels, we rotate the text on the x-axis by 90 degrees using the `theme` function.

Examine results in 2D

Another way to examine the clustering result is by visualizing the grouping in a 2D data space:

```
scale(olive[, -c(1, 2)]) %>%
  as_tibble() %>%
  ggplot() +
  geom_point(aes(oleic, arachidic, col = factor(result$cluster))) +
  labs(col = "cluster")
```

As before, we need to drop the non-numeric variables, scale the dataset, and convert it back into a `tibble`. The scatter plot is straightforward—you place one variable on the x-axis and another on the y-axis. The key part is converting the cluster labels (`result$cluster`) into a factor vector. If you don't, `ggplot` will use a continuous color scheme instead of a discrete one.

Examin results using tour

To examine the results using the tour, the process is quite similar to visualizing it with `langevitour`, but this time we color by the K-means cluster labels:

```r
set.seed(12345)
result <- kmeans(scale(olive[, -c(1, 2)]),
                 centers = 3)
langevitour(olive[, -c(1, 2)] %>% scale(),
            group = result$cluster,
            pointSize = 2,
            levelColors = RColorBrewer::brewer.pal(3, "Dark2"))
```

The only difference here is that we set `group = result$cluster` to indicate that we want the animation to color the points according to the K-means cluster labels.

Cluster statistics

To obtain cluster statistics using the `fpc` package, you need to provide two arguments to the `cluster.stats` function. The first argument is the distance matrix, and the second is the cluster labels. For K-means, you would use the following code:

```r
fpc::cluster.stats(dist(scale(olive[, -c(1, 2)])), result$cluster)
```

Cut the tree using dendrogram

In slide 46, there's an example of how to visualize hierarchical clustering results using a dendrogram and how to cut the tree:

```r
dat <- data.frame(x = c(8, 2, 7, 10, 5, 3, 1, 1),
                  y = c(2, 4, 2, 4, 5, 10, 10, 7))
rownames(dat) <- c("A", "B", "C", "D", "E", "F", "G", "H")
dat %>%
  dist(method = "euclidean") %>%
  hclust(method = "ward.D2") %>%
  ggdendro::ggdendrogram() +
  geom_hline(yintercept = 10, linetype = 2)
```

Once we have the hierarchical clustering result, we simply pass it to the `ggdendro::ggdendrogram` function. We can use `geom_hline` to draw a horizontal line at a specified `yintercept` to indicate where we want to cut the tree. Setting `linetype = 2` makes the line dashed.

Trying Hierarchical Clustering on `olive`

The R code for hierarchical clustering is provided in slide 48:

```
result <- dist(scale(olive[, -c(1, 2)])) %>%
  hclust(method = "ward.D2")
table(true = olive$region, hclust = cutree(result, 3))
ggdendrogram(result, labels = FALSE)
```

As usual, we drop the true region and the non-numeric columns from the `olive` dataset using `[, -c(1, 2)]`, although you can also use the `select` function if you prefer. We standardize the data with the `scale` function, and the `dist` function returns a distance matrix. Since we didn't specify a method for the `dist` function, it defaults to calculating Euclidean distance. The `hclust` function performs the hierarchical clustering, and we need to specify the linkage method using the `method` argument; here, we use `ward.D2`, a specific version of the Ward linkage.

Next, we can compare the true regions with the cluster labels. Remember that we provide vectors to the `table` function. To obtain a vector of cluster labels from the hierarchical clustering result, we use the `cutree` function and specify the desired number of clusters. The output from `table` indicates that region 1 is assigned to clusters 1 and 2, while regions 2 and 3 are both assigned to cluster 3. The result is quite clear, with many zeros in the table. To enhance clarity, you can rename cluster 1 to cluster 2 and cluster 2 to cluster 1, placing 200 in the first diagonal entry, but the overall conclusion remains unchanged.

We can also visualize the clustering results using `ggdendrogram`. Note that we set `labels = FALSE` to turn off the x-axis labels, as there are too many observations to display effectively.

Examine results using tour

This process is largely the same as how we examine results using the tour for K-means. The only difference is that we specify the `group` argument as `cutree(result, 3)`, indicating that we want to color the data points by these cluster labels, which are obtained by cutting the tree.

```
result <- dist(scale(olive[, -c(1, 2)])) %>%
  hclust(method = "ward.D2")
langevitour(olive[, -c(1, 2)] %>% scale(),
            group = cutree(result, 3),
            pointSize = 2,
            levelColors = RColorBrewer::brewer.pal(3, "Dark2"))
```