17 Dimensionality reduction

Dimensionality reduction can be a good choice when you suspect there are "too many" variables. An excess of variables, usually predictors, can be a problem because it is difficult to understand or visualize data in higher dimensions. For example, in high dimensional biology experiments, one of the first tasks is to determine if there are any unwanted trends in the data (e.g., effects not related to the question of interest, such as lab-to-lab differences). Debugging the data is difficult when there are hundreds of thousands of dimensions, and dimensionality reduction can be an aid for exploratory data analysis.

Another potential consequence of having a multitude of predictors is possible harm for a model. The simplest example is a method like ordinary linear regression where the number of predictors should be less than the number of data points used to fit the model. Another issue is multicollinearity, where between-predictor correlations can negatively impact the mathematical operations used to estimate a model. If there are an extremely large number of predictors, it is fairly unlikely that there are an equal number of real underlying effects. Predictors may be measuring the same latent effect(s), and thus such predictors will be highly correlated. Many dimensionality reduction techniques thrive in this situation. In fact, most can only be effective when there are such relationships between predictors that can be exploited.

> When starting a new modeling project, reducing the dimensions of the data may provide some intuition about how hard the modeling problem may be.

Principal component analysis (PCA) is one of the most straightforward methods for reducing the number of columns in the data set because it relies on linear methods and it is unsupervised (i.e., does not consider the outcome data). For a high dimensional classification problem, an initial plot of the main PCA components might show a clear separation between the classes. If this is the case, then it is fairly safe to assume that a linear classifier might do a good job. However, the converse is not true; a lack of separation does not mean that the problem is insurmountable.

The dimensionality reduction methods discussed here are generally *not* feature selection methods. Methods such as PCA represent the original predictors using a smaller subset of new features. All of the original predictors are required to compute these new features. The exception to this are sparse methods that have the ability to completely remove the impact of predictors when creating the new features.

This chapter has two goals:

- Demonstrate how to use recipes to create a small set of features that capture the main aspects of the original predictor set.
- Describe how recipes can be used on their own (as opposed to being used in a workflow object, as in Section 8.2).

The latter is helpful when testing or debugging a recipe. However, as described in Section 8.2, the best way to use a recipe for modeling is from within a workflow object.

In addition to the **tidymodels** package, this chapter uses the following packages: **baguette**, **beans**, **bestNormalize**, **corrplot**, **discrim**, **embed**, **ggforce**, **klaR**, **learntidymodels**[20], **mixOmics**[21], and **uwot**. .

17.1 A PICTURE IS WORTH A THOUSAND... BEANS

Koklu and Ozkan (2020) describe methods for determining the varieties of dried beans in an image. From their manuscript:

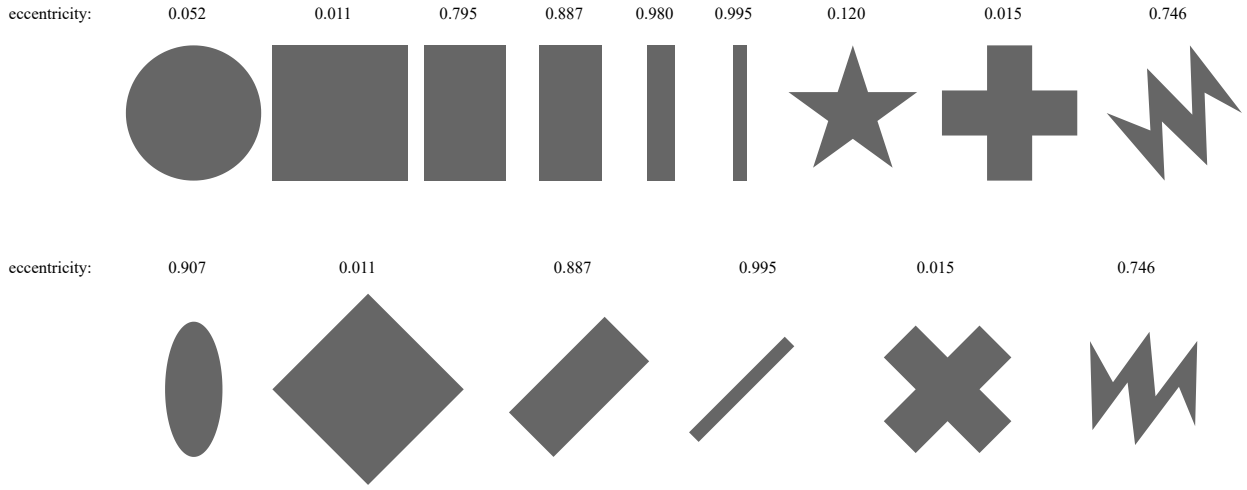
> The primary objective of this study is to provide a method for obtaining uniform seed varieties from crop production, which is in the form of population, so the seeds are not certified as a sole variety. Thus, a computer vision system was developed to distinguish seven different registered varieties of dry beans with similar features in order to obtain uniform seed classification. For the classification model, images of 13,611 grains of 7 different registered dry beans were taken with a high-resolution camera.

Each image contains multiple beans. The process of determining which pixels correspond to a particular bean is called *image segmentation*. These pixels can be analyzed to produce features for each bean, such as color and morphology (i.e., shape). These features are then used to model the outcome (bean variety) because different bean varieties look different. The training data comes from a set of manually labeled images, and this data set is used to create a predictive model that can distinguish between seven bean varieties: Cali, Horoz, Dermason, Seker, Bombay, Barbunya, and Sira. Producing an effective model can help manufacturers quantify the homogeneity of a batch of beans.

There are numerous methods to quantify shapes of objects (Mingqiang, Kidiyo, and Joseph 2008). Many are related to the boundaries or regions of the object of interest. Example of features include:

- The **area** (or size) can be estimated using the number of pixels in the object or the size of the convex hull around the object.
- We can measure the **perimeter** using the number of pixels in the boundary as well as the area of the bounding box (the smallest rectangle enclosing an object).
- The **major axis** quantifies the longest line connecting the most extreme parts of the object. The **minor axis** is perpendicular to the major axis.
- We can measure the **compactness** of an object using the ratio of the object's area to the area of a circle with the same perimeter. For example, the symbols "•" and "×" have very different compactness.
- There are also different measures of how oblong or **elongated** an object is. For example, the **eccentricity** statistic is the ratio of the major and minor axes. There are also related estimates for roundness and convexity.

Notice the eccentricity for these different shapes:



Shapes such as circles and squares have low eccentricity while oblong shapes have high values. Also, the metric is unaffected by the rotation of the object.

Many of these image features have high correlations; objects with large areas are more likely to have large perimeters. There are often multiple methods to quantify the same underlying characteristics (e.g. size).

In the bean data, 16 morphology features were computed: area, perimeter, major axis length, minor axis length, aspect ratio, eccentricity, convex area, equiv diameter, extent, solidity, roundness, compactness, shape factor 1, shape factor 2, shape factor 3, and shape factor 4. The latter four are described in Symons and Fulcher (1988). While the dimensionality of these data is not very large compared to many real-world modeling problems, it does provide a nice working example to demonstrate how to reduce the number of features.

We can begin by loading the data:

```
library(tidymodels)
tidymodels_prefer()
library(beans)
```

> It is important to maintain good data discipline when evaluating dimensionality reduction techniques, especially if you will use them within a model.

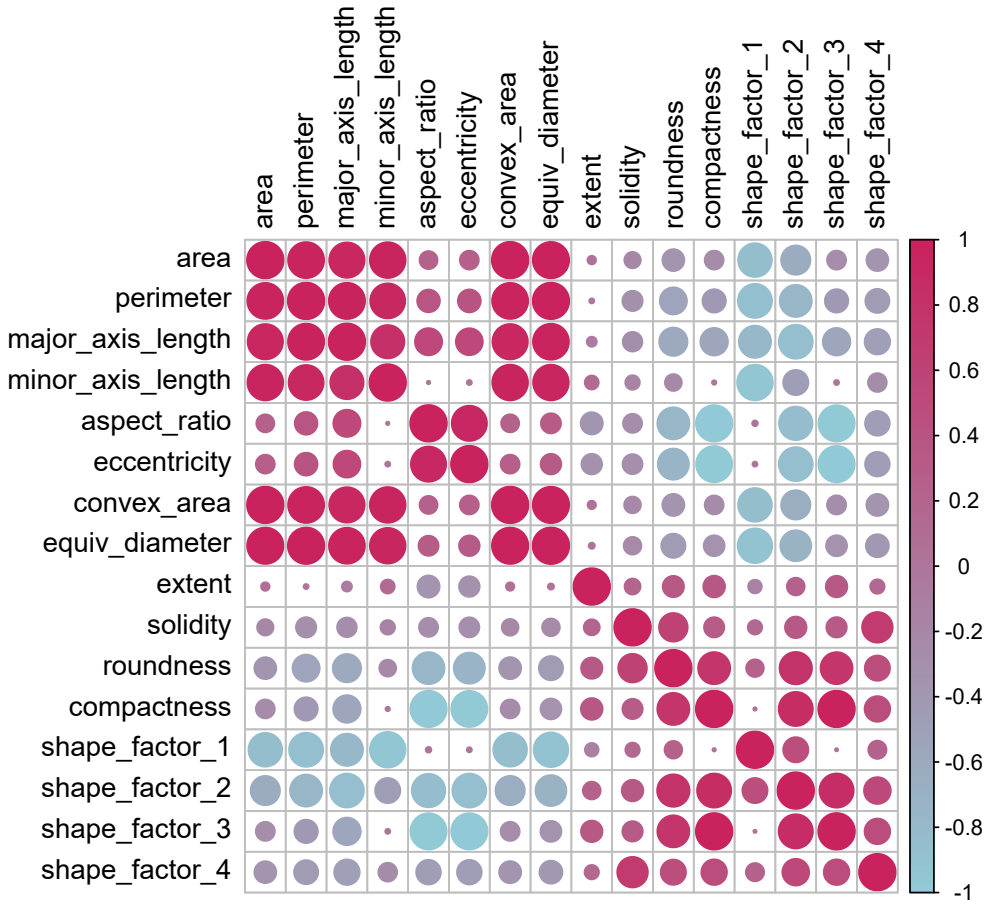
For our analyses, we start by holding back a testing set with `initial_split()`. The remaining data are split into training and validation sets:

```
set.seed(1701)
bean_split <- initial_split(beans, strata = class, prop = 3/4)

bean_train <- training(bean_split)
bean_test  <- testing(bean_split)

set.seed(1702)
bean_val <- validation_split(bean_train, strata = class, prop = 4/5)
bean_val$splits[[1]]
#> <Training/Validation/Total>
#> <8163/2044/10207>
```

To visually assess how well different methods perform, we can estimate the methods on the training set (n = 8163 beans) and display the results using the validation set (n = 2044).

Before beginning, we can spend some time investigating our data. Since we know that many of these shape features are probably measuring similar concepts, let's take a look at the correlation structure of the data:

```
library(corrplot)
tmwr_cols <- colorRampPalette(c("#91CBD765", "#CA225E"))
bean_train %>%
  select(-class) %>%
  cor() %>%
  corrplot(col = tmwr_cols(200), tl.col = "black")
```



While we don't take the time to do it here, it is also important to see if this correlation structure significantly changes across the outcome categories. This can help create better models.

17.2 A STARTER RECIPE

It's time to look at these data in a smaller space. We can start with a basic recipe to preprocess the data prior to any dimensionality reduction steps. Several predictors are ratios and so are likely to have skewed distributions. Such distributions can wreak havoc on variance calculations (such as the ones used in PCA). The **bestNormalize** package has a step that can enforce a symmetric distribution for the predictors. We'll use this to mitigate the issue of skewed distributions.

```
library(bestNormalize)
bean_rec <-
  # Use the training data from the bean_val split object
  recipe(class ~ ., data = analysis(bean_val$splits[[1]])) %>%
  step_zv(all_numeric_predictors()) %>%
  step_orderNorm(all_numeric_predictors()) %>%
  step_normalize(all_numeric_predictors())
```

> Remember that when invoking the `recipe()` function, the steps are not estimated or executed in any way.

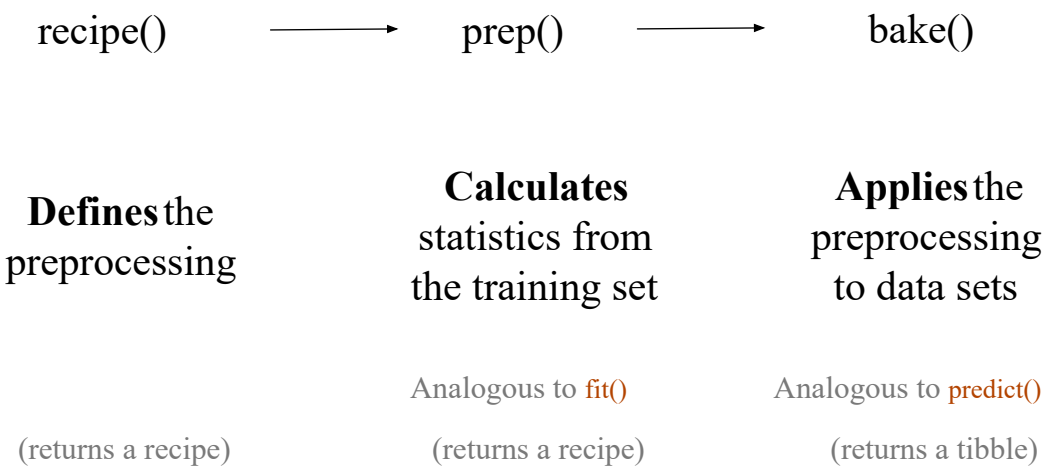
This recipe will be extended with additional steps for the dimensionality reduction analyses. Before doing so, let's go over how a recipe can be used *outside* of a workflow.

17.3 RECIPES IN THE WILD

As mentioned in Section 8.2, a workflow containing a recipe uses `fit()` to estimate the recipe and model, then `predict()` to process the data and make model predictions. There are analogous functions in the **recipes** package that can be used for the same purpose:

- `prep(recipe, training)` fits the recipe to the training set.
- `bake(recipe, new_data)` applies the recipe operations to `new_data`.

In summary:



Let's look at each of these functions in more detail.

17.3.1 PREPARING A RECIPE

Let's estimate `bean_rec` using the training set data:

```
bean_rec_trained <- prep(bean_rec)
bean_rec_trained
#> Recipe
#>
#> Inputs:
#>
#>       role #variables
#>    outcome          1
#>  predictor         16
#>
#> Training data contained 8163 data points and no missing data.
#>
#> Operations:
#>
#> Zero variance filter removed no terms [trained]
#> orderNorm transformation on area, perimeter, major_axis_length, minor_axis... [trained]
#> Centering and scaling for area, perimeter, major_axis_length, minor_axis_leng... [traine
```

> Remember that `prep()` for a recipe is like `fit()` for a model.

Note in the output that the steps have been trained and that the selectors are no longer general (i.e., `all_numeric_predictors()`); they now show the actual columns that were selected. Also, `prep(bean_rec)` does not require the `training` argument. You can pass any data into that argument, but omitting it means that the original `data` from the call to `recipe()` will be used. In our case, this was the training set data.

One important argument to `prep()` is `retain`. When `retain = TRUE` (the default), the estimated version of the training set is kept within the recipe. This data set has been pre-processed using all of the steps listed in the recipe. Since `prep()` has to execute the recipe as it proceeds, it may be advantageous to keep this version of the training set so that, if that data set is to be used later, redundant calculations can be avoided. However, if the training set is big, it may be problematic to keep such a large amount of data in memory. Use `retain = FALSE` to avoid this.

Once new steps are added to this estimated recipe, re-applying `prep()` will only estimate the untrained steps. This will come in handy when we try different feature extraction methods below.

> If you encounter errors when working with a recipe, `prep()` can be used with its `verbose` option to troubleshoot:

```
bean_rec_trained %>%
  step_dummy(cornbread) %>%  # <- not a real predictor
  prep(verbose = TRUE)
#> oper 1 step zv [pre-trained]
#> oper 2 step orderNorm [pre-trained]
#> oper 3 step normalize [pre-trained]
#> oper 4 step dummy [training]
#> Error: Can't subset columns that don't exist.
#> ✖ Column `cornbread` doesn't exist.
```

Another option that can help you understand what happens in the analysis is `log_changes`:

```
show_variables <-
  bean_rec %>%
  prep(log_changes = TRUE)
#> step_zv (zv_SqX2i): same number of columns
#>
#> step_orderNorm (orderNorm_x4c8K): same number of columns
#>
#> step_normalize (normalize_NF9ZV): same number of columns
```

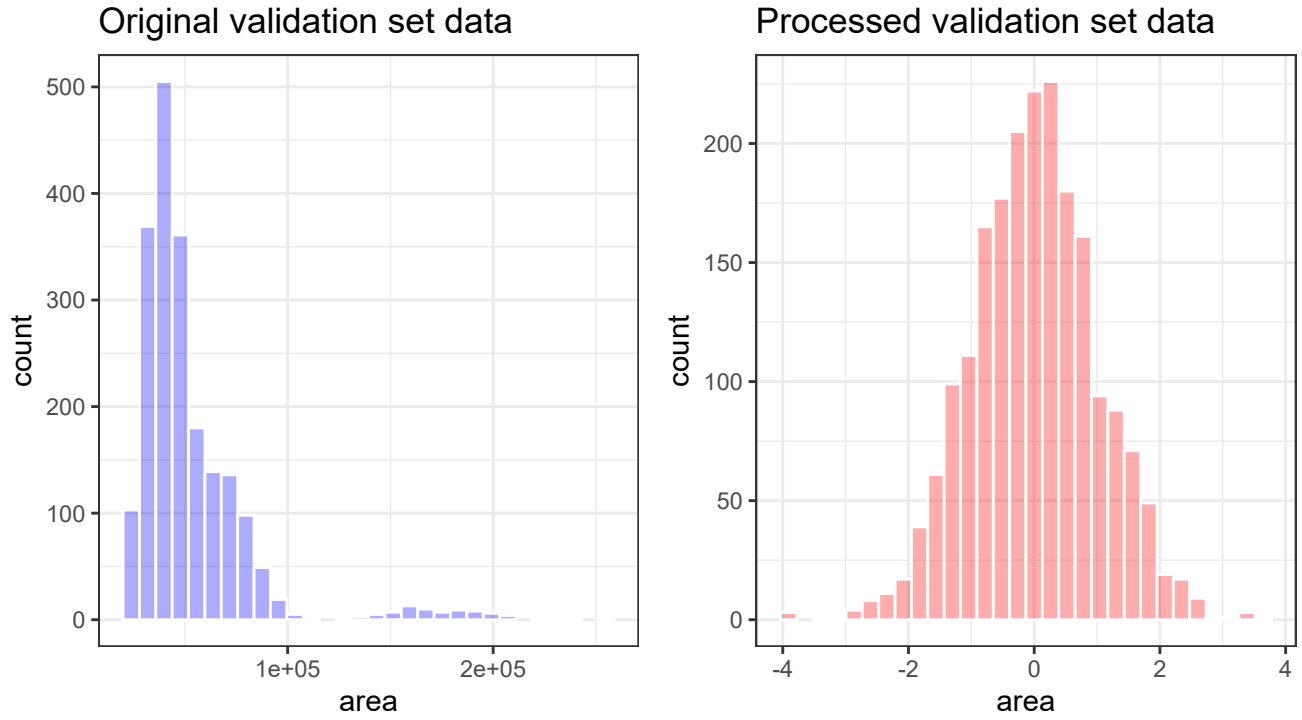
17.3.2 BAKING THE RECIPE

> Using `bake()` with a recipe is much like using `predict()` with a model; the operations estimated from the training set are applied to any data, like testing data or new data at prediction time.

For example, the validation set samples can be processed:

```
bean_validation <- bean_val$splits %>% pluck(1) %>% assessment()
bean_val_processed <- bake(bean_rec_trained, new_data = bean_validation)
```

Here are histograms of the `area` predictor before and after the recipe was prepared:



There are two important aspects of `bake()` that are worth noting here.

First, as previously mentioned, using `prep(recipe, retain = TRUE)` keeps the existing processed version of the training set in the recipe. This enables the user to use `bake(recipe, new_data = NULL)`, which returns that data set without further computations. For example:

```
bake(bean_rec_trained, new_data = NULL) %>% nrow()
#> [1] 8163
bean_val$splits %>% pluck(1) %>% analysis() %>% nrow()
#> [1] 8163
```

If the training set is not pathologically large, using this value of `retain` can save a lot of computational time.

Second, additional selectors can be used in the call to specify which columns to return. The default selector is `everything()`, but more specific directives can be used.

`prep()` and `bake()` will be used in the code below to illustrate some of these options.

17.4 FEATURE EXTRACTION TECHNIQUES

Since recipes are the primary option in tidymodels for dimensionality reduction, let's write a function that will estimate the transformation and plot the resulting data in a scatter plot matrix via the **ggforce** package:

```
library(ggforce)
plot_validation_results <- function(recipe, dat = assessment(bean_val$splits[[1]])) {
  recipe %>%
    # Estimate any additional steps
    prep() %>%
    # Process the data (the validation set by default)
    bake(new_data = dat) %>%
    # Create the scatterplot matrix
    ggplot(aes(x = .panel_x, y = .panel_y, col = class, fill = class)) +
    geom_point(alpha = 0.4, size = 0.5) +
    geom_autodensity(alpha = .3) +
    facet_matrix(vars(-class), layer.diag = 2) +
    scale_color_brewer(palette = "Dark2") +
    scale_fill_brewer(palette = "Dark2")
}
```

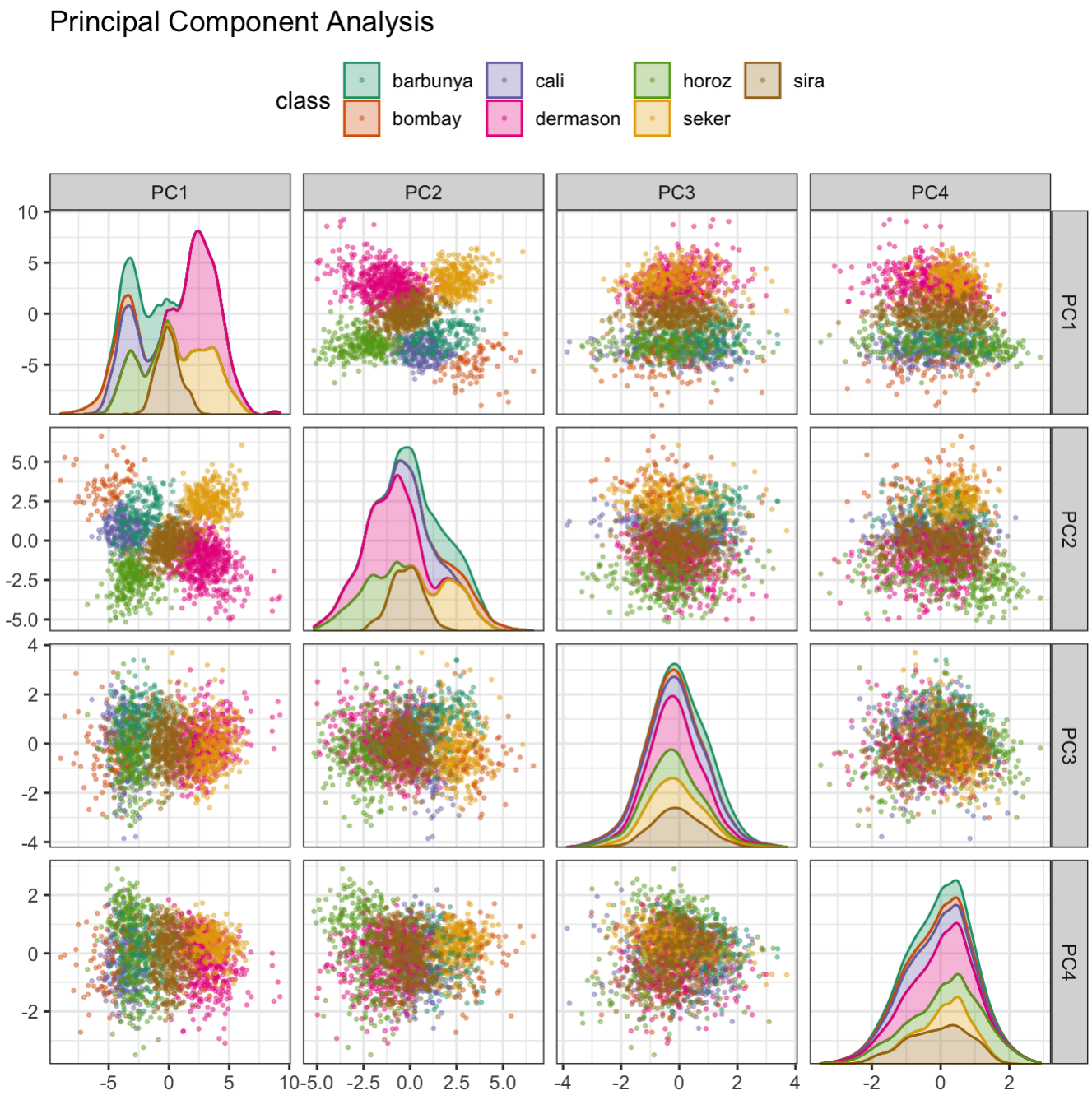
We will reuse this function several times below.

A series of several feature extraction methodologies are explored here. An overview of most can be found in Section 6.3.1 of Kuhn and Johnson (2020) and the references therein. The UMAP method is described in McInnes, Healy, and Melville (2020).

17.4.1 PRINCIPAL COMPONENT ANALYSIS

As previously mentioned, PCA is an unsupervised method that uses linear combinations of the predictors to define new features. These features attempt to account for as much variation as possible in the original data. We add `step_pca()` to the original recipe and use our function to visualize the results on the validation set:

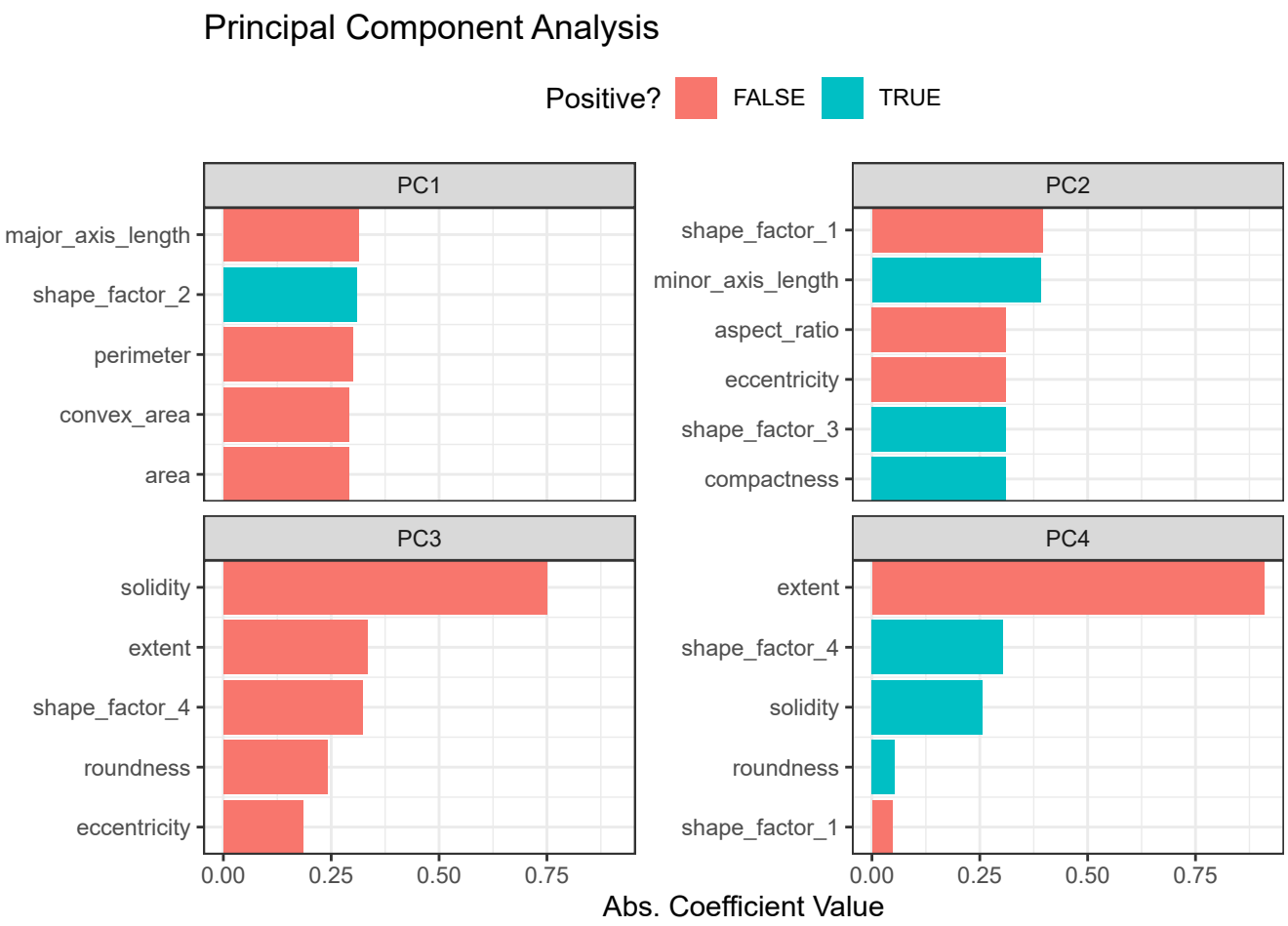
```
bean_rec_trained %>%
  step_pca(all_numeric_predictors(), num_comp = 4) %>%
  plot_validation_results() +
  ggtitle("Principal Component Analysis")
```



We see that the first two components, especially when used together, do an effective job separating the classes. This may lead us to expect that the overall problem of classifying these beans will not be especially difficult.

Recall that PCA is unsupervised. For these data, it turns out that the PCA components that explain the most variation in the predictors also happen to be predictive of the classes. What features are driving performance? The **learntidymodels** package, found on GitHub, has functions that can help visualize the top features for each component. We'll need the prepared recipe; the PCA step is added below along with a call to `prep()` :

```
library(learntidymodels)
bean_rec_trained %>%
  step_pca(all_numeric_predictors(), num_comp = 4) %>%
  prep() %>%
  plot_top_loadings(component_number <= 4, n = 5) +
  ggtitle("Principal Component Analysis")
```



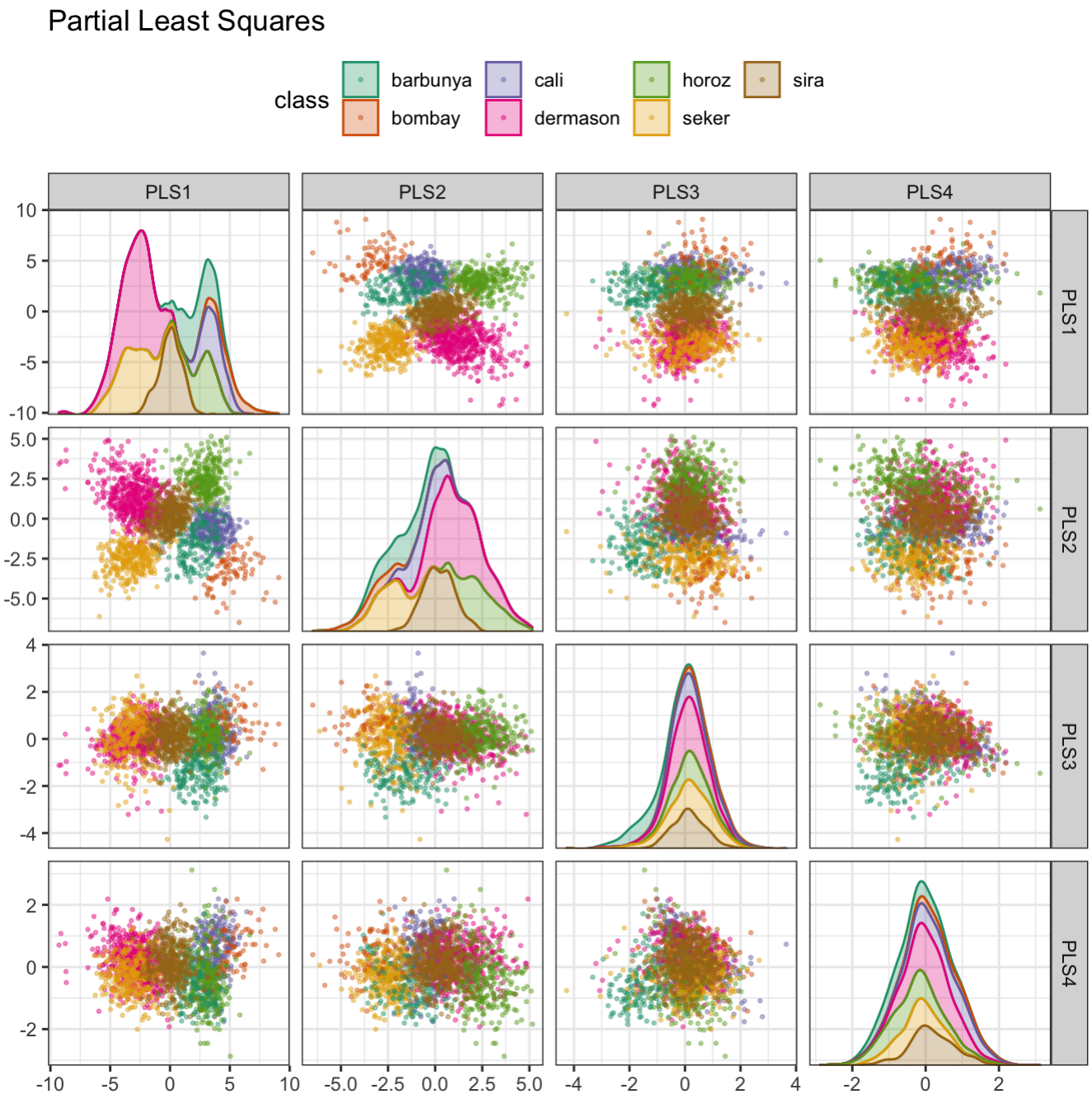
The top loadings are mostly related to the cluster of correlated predictors shown in the top left portion of the previous correlation plot: perimeter, area, major axis length, and convex area. These are all related to bean size. Shape factor 2, from Symons and Fulcher (1988), is the area over the

cube of the major axis length and is therefore also related to bean size. Measures of elongation appear to dominate the second PCA component.

17.4.2 PARTIAL LEAST SQUARES

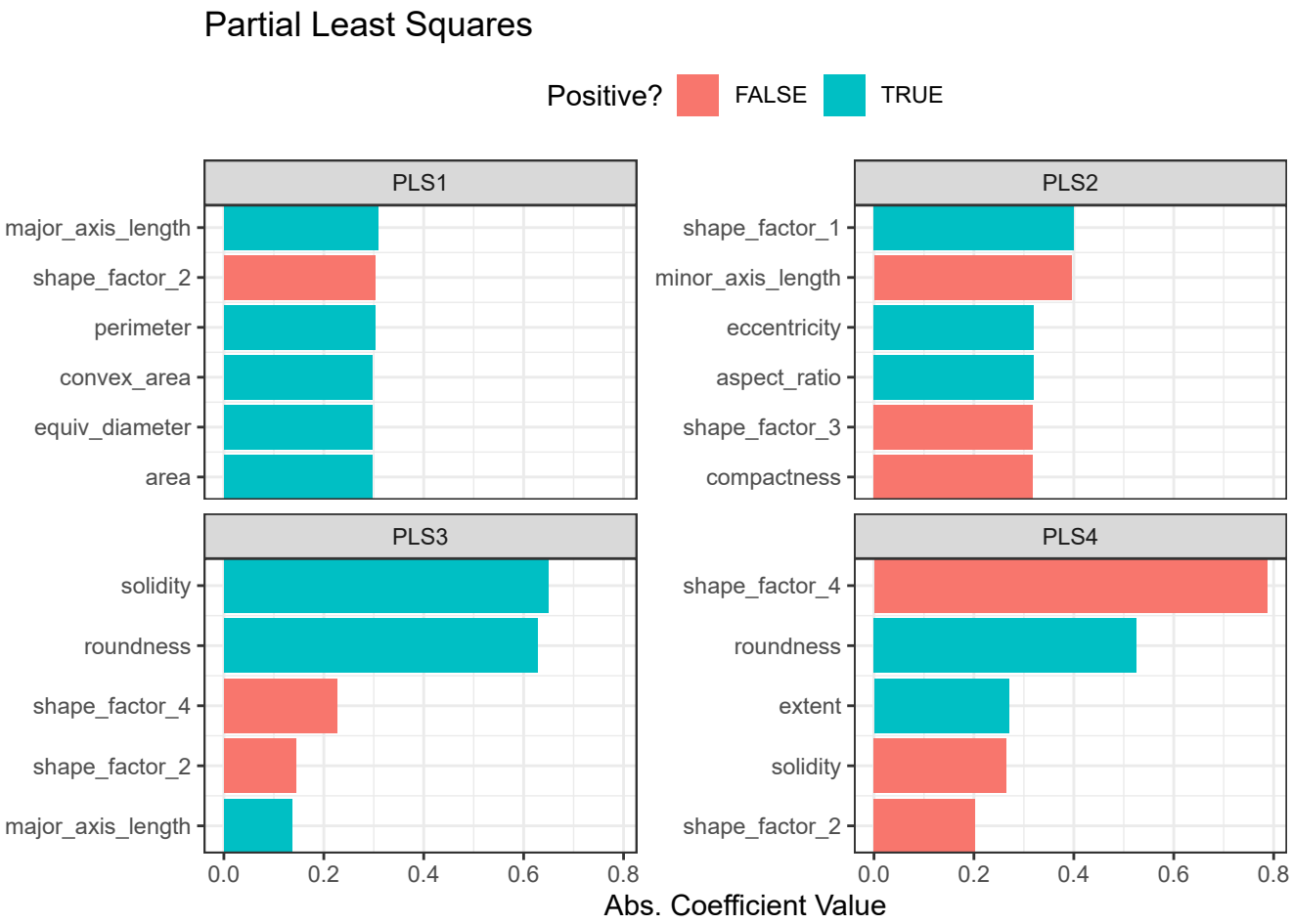
PLS is a supervised version of PCA. It tries to find components that simultaneously maximize the variation in the predictors while also maximizing the relationship between those components and the outcome.

```
bean_rec_trained %>%
  step_pls(all_numeric_predictors(), outcome = "class", num_comp = 4) %>%
  plot_validation_results() +
  ggtitle("Partial Least Squares")
```



The first two PLS components are nearly identical to the first two PCA components! We find this result because those PCA components are so effective at separating the varieties of beans. The remaining components are different.

```
library(learntidymodels)
bean_rec_trained %>%
  step_pls(all_numeric_predictors(), outcome = "class", num_comp = 4) %>%
  prep() %>%
  plot_top_loadings(component_number <= 4, n = 5, type = "pls") +
  ggtitle("Partial Least Squares")
```



Solidity (i.e., the density of the bean) drives the third PLS component, along with roundness. Solidity may be capturing bean features related to "bumpiness" of the bean surface since it can measure irregularity of the bean boundaries. Extent is the largest effect in the fourth component; it is an image texture measure similar to solidity.

17.4.3 INDEPENDENT COMPONENT ANALYSIS

ICA is slightly different than PCA in that it finds components that are as statistically independent from one another as possible (as opposed to being uncorrelated). It can be thought of as maximizing the "non-Gaussianity" of the ICA components.

```
bean_rec_trained %>%
  step_ica(all_numeric_predictors(), num_comp = 4) %>%
  plot_validation_results() +
  ggtitle("Independent Component Analysis")
```

Inspecting this plot, there does not appear to be much separation between the classes in the first few components when using ICA.

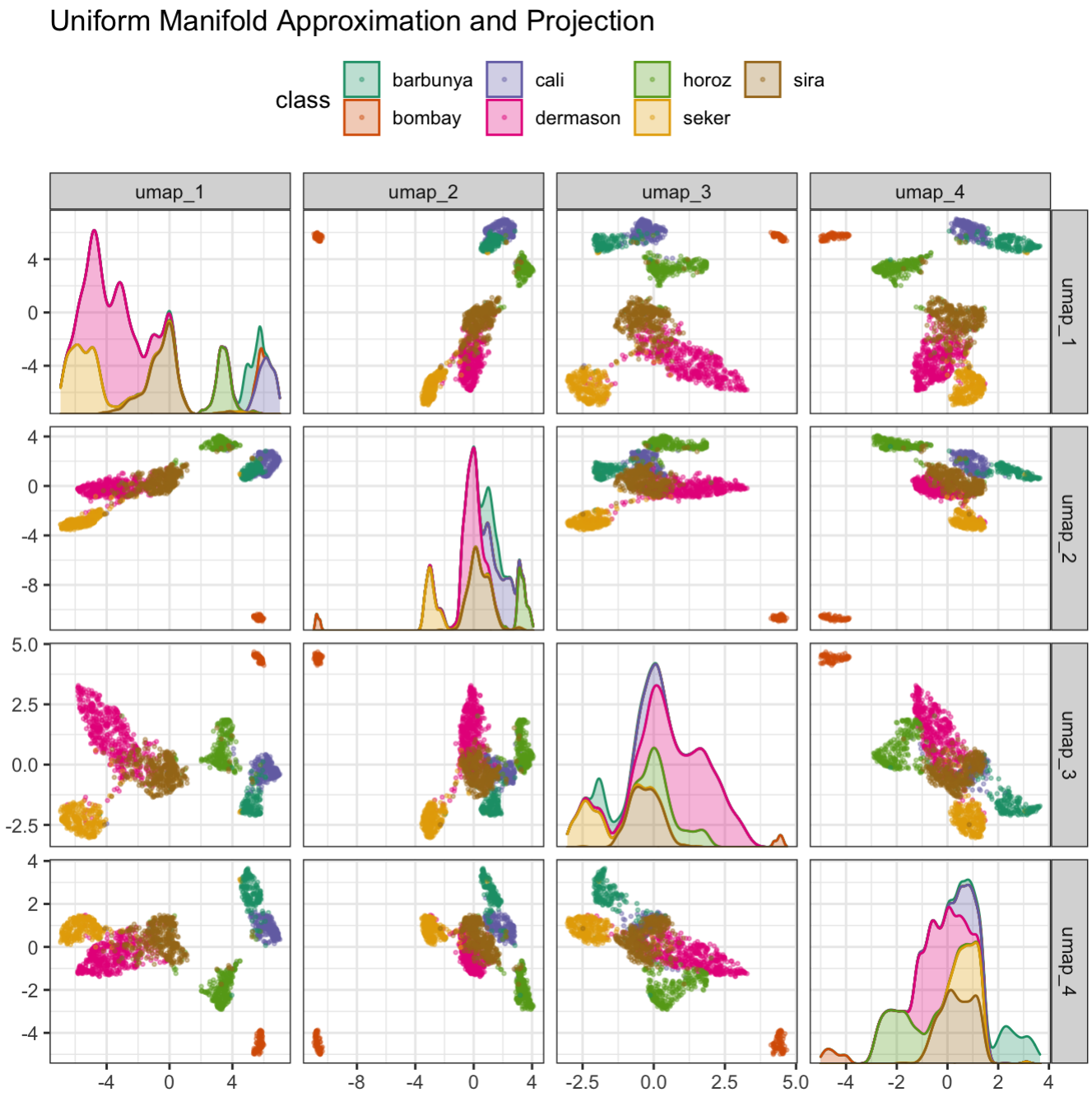
17.4.4 UNIFORM MANIFOLD APPROXIMATION AND PROJECTION

UMAP is similar to the popular t-SNE method for nonlinear dimension reduction. In the original high-dimensional space, UMAP uses a distance-based nearest neighbor method to find local areas of the data where the data points are more likely to be related. The relationship between data points is saved as a directed graph model where most points are not connected.

From there, UMAP translates points in the graph to the reduced dimensional space. To do this, the algorithm has an optimization process that uses cross-entropy to map data points to the smaller set of features so that the graph is well approximated.

To create the mapping, the **embed** package contains a step function for this method:

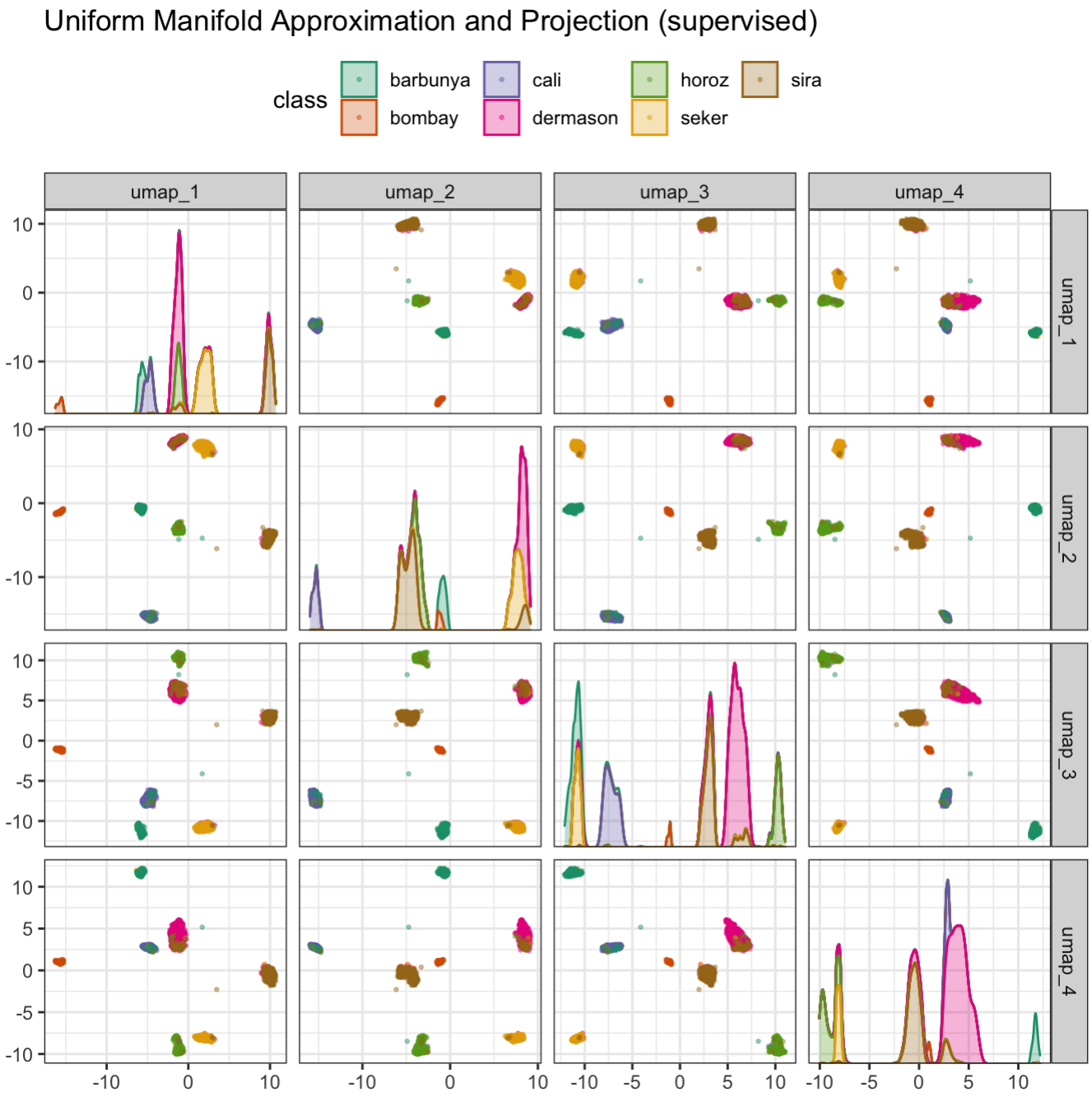
```
library(embed)
bean_rec_trained %>%
  step_umap(all_numeric_predictors(), num_comp = 4) %>%
  plot_validation_results() +
  ggtitle("Uniform Manifold Approximation and Projection")
```



While the between-cluster space is pronounced, the clusters can contain a heterogeneous mixture of classes.

There is also a supervised version of UMAP:

```
bean_rec_trained %>%
  step_umap(all_numeric_predictors(), outcome = "class", num_comp = 4) %>%
  plot_validation_results() +
  ggtitle("Uniform Manifold Approximation and Projection (supervised)")
```



The supervised method looks promising for modeling the data.

UMAP is a powerful method to reduce the feature space. However, it can be very sensitive to tuning parameters (e.g. the number of neighbors and so on). For this reason, it would help to experiment with a few of the parameters to assess how robust the results are for these data.

17.5 MODELING

Both the PLS and UMAP methods are worth investigating in conjunction with different models. Let's explore a variety of different models with these dimensionality reduction techniques (along with no transformation at all): a single layer neural network, bagged trees, flexible discriminant analysis (FDA), naive Bayes, and regularized discriminant analysis (RDA).

Now that we are back in "modeling mode", we'll create a series of model specifications and then use a workflow set to tune the models. Note that the model parameters are tuned in conjunction with the recipe parameters (e.g. size of the reduced dimension, UMAP parameters).

```
library(baguette)
library(discrim)

mlp_spec <-
  mlp(hidden_units = tune(), penalty = tune(), epochs = tune()) %>%
  set_engine('nnet') %>%
  set_mode('classification')

bagging_spec <-
  bag_tree() %>%
  set_engine('rpart') %>%
  set_mode('classification')

fda_spec <-
  discrim_flexible(
    prod_degree = tune()
  ) %>%
  set_engine('earth')

rda_spec <-
  discrim_regularized(frac_common_cov = tune(), frac_identity = tune()) %>%
  set_engine('klaR')

bayes_spec <-
  naive_Bayes() %>%
  set_engine('klaR')
```

```
bean_rec <-
  recipe(class ~ ., data = bean_train) %>%
  step_zv(all_numeric_predictors()) %>%
  step_orderNorm(all_numeric_predictors()) %>%
  step_normalize(all_numeric_predictors())

pls_rec <-
  bean_rec %>%
  step_pls(all_numeric_predictors(), outcome = "class", num_comp = tune())

umap_rec <-
  bean_rec_trained %>%
  step_umap(
    all_numeric_predictors(),
    outcome = "class",
    num_comp = tune(),
    neighbors = tune(),
    min_dist = tune()
  )
```

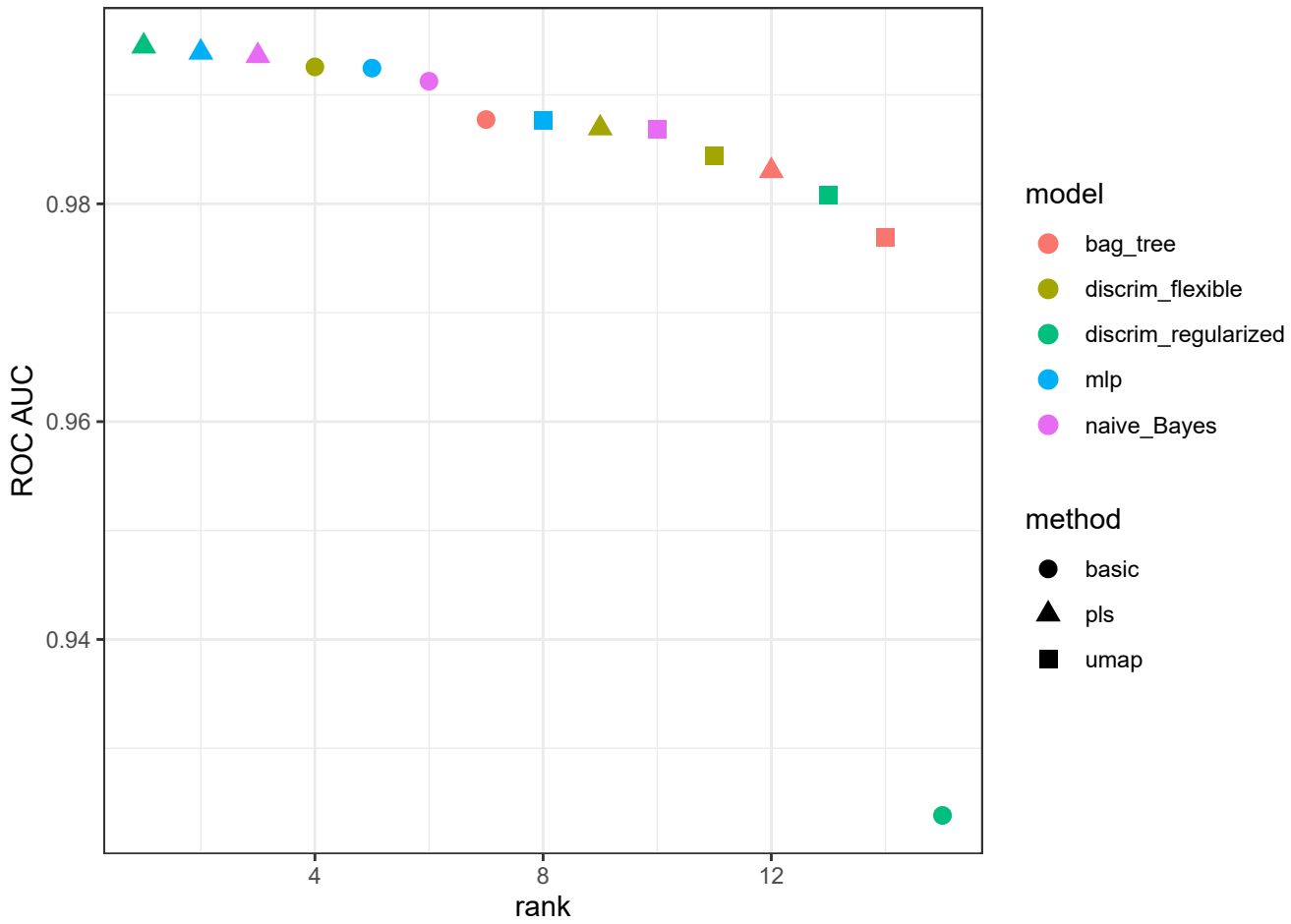
Once again, **workflowsets** take the preprocessors and models and crosses them. The `control` option `parallel_over` is set so that the parallel processing can work simultaneously across tuning parameter combinations. The `workflow_map()` function applies grid search to optimize the model/preprocessing parameters (if any) across 10 parameter combinations. The multiclass area under the ROC curve is estimated on the validation set.

```
ctrl <- control_grid(parallel_over = "everything")
bean_res <-
  workflow_set(
    preproc = list(basic = class ~., pls = pls_rec, umap = umap_rec),
    models = list(bayes = bayes_spec, fda = fda_spec,
                  rda = rda_spec, bag = bagging_spec,
                  mlp = mlp_spec)
  ) %>%
  workflow_map(
    verbose = TRUE,
    seed = 1703,
    resamples = bean_val,
    grid = 10,
    metrics = metric_set(roc_auc),
    control = ctrl
  )
#> i    No tuning parameters. `fit_resamples()` will be attempted
#> i  1 of 15 resampling: basic_bayes
#> ! validation: preprocessor 1/1, model 1/1 (predictions): Numerical 0 probability for all
#> ✓  1 of 15 resampling: basic_bayes (9.3s)
#> i  2 of 15 tuning:     basic_fda
#> ✓  2 of 15 tuning:     basic_fda (3.2s)
#> i  3 of 15 tuning:     basic_rda
#> ✓  3 of 15 tuning:     basic_rda (8.2s)
#> i    No tuning parameters. `fit_resamples()` will be attempted
#> i  4 of 15 resampling: basic_bag
#> ✓  4 of 15 resampling: basic_bag (5.6s)
#> i  5 of 15 tuning:     basic_mlp
#> ✓  5 of 15 tuning:     basic_mlp (7.1s)
#> i  6 of 15 tuning:     pls_bayes
#> ✓  6 of 15 tuning:     pls_bayes (9.3s)
#> i  7 of 15 tuning:     pls_fda
#> ✓  7 of 15 tuning:     pls_fda (7.6s)
#> i  8 of 15 tuning:     pls_rda
#> ✓  8 of 15 tuning:     pls_rda (16.8s)
```

```
#> i  9 of 15 tuning:     pls_bag
#> ✓  9 of 15 tuning:     pls_bag (15.7s)
#> i 10 of 15 tuning:     pls_mlp
#> ✓ 10 of 15 tuning:     pls_mlp (26.2s)
#> i 11 of 15 tuning:     umap_bayes
#> ✓ 11 of 15 tuning:     umap_bayes (3m 18.7s)
#> i 12 of 15 tuning:     umap_fda
#> ✓ 12 of 15 tuning:     umap_fda (3m 37s)
#> i 13 of 15 tuning:     umap_rda
#> ✓ 13 of 15 tuning:     umap_rda (4m 4.1s)
#> i 14 of 15 tuning:     umap_bag
#> ✓ 14 of 15 tuning:     umap_bag (3m 31.5s)
#> i 15 of 15 tuning:     umap_mlp
#> ✓ 15 of 15 tuning:     umap_mlp (3m 15.1s)
```

We can rank the models by their validation set estimates of the area under the ROC curve:

```
rankings <-
  rank_results(bean_res, select_best = TRUE) %>%
  mutate(method = map_chr(wflow_id, ~ str_split(.x, "_", simplify = TRUE)[1]))

tidymodels_prefer()
filter(rankings, rank <= 5) %>% dplyr::select(rank, mean, model, method)
#> # A tibble: 5 × 4
#>    rank  mean model               method
#>   <int> <dbl> <chr>               <chr>
#> 1     1 0.994 discrim_regularized pls
#> 2     2 0.994 mlp                 pls
#> 3     3 0.994 naive_Bayes         pls
#> 4     4 0.993 discrim_flexible    basic
#> 5     5 0.992 mlp                 basic

rankings %>%
  ggplot(aes(x = rank, y = mean, pch = method, col = model)) +
  geom_point(cex = 3) +
  theme(legend.position = "right") +
  labs(y = "ROC AUC")
```



It is clear from these results that most models give very good performance; there are few bad choices here. For demonstration, we'll use the RDA model with PLS features as the final model. We will finalize the workflow with the numerically best parameters, fit it to the training set, then evaluate with the test set:

```
rda_res <-
  bean_res %>%
  extract_workflow("pls_rda") %>%
  finalize_workflow(
    bean_res %>%
      extract_workflow_set_result("pls_rda") %>%
      select_best(metric = "roc_auc")
  ) %>%
  last_fit(split = bean_split, metrics = metric_set(roc_auc))

rda_wflow_fit <- rda_res$.workflow[[1]]
```

```
collect_metrics(rda_res)
#> # A tibble: 1 × 4
#>   .metric .estimator .estimate .config
#>   <chr>   <chr>          <dbl> <chr>
#> 1 roc_auc hand_till      0.995 Preprocessor1_Model1
```

Pretty good! We'll use this model in the next chapter to demonstrate variable importance methods.

17.6 CHAPTER SUMMARY

Dimensionality reduction can be a helpful tool for exploratory data analysis as well as modeling. The **recipes** and **embed** packages contain steps for a variety of different methods and **workflowsets** facilitates choosing an appropriate method for a data set. This chapter also discussed how recipes can be used on their own, either for debugging problems with a recipe or directly for exploratory data analysis and data visualization.

REFERENCES

Koklu, M, and IA Ozkan. 2020. "Multiclass Classification of Dry Beans Using Computer Vision and Machine Learning Techniques." *Computers and Electronics in Agriculture* 174: 105507.

Kuhn, M, and K Johnson. 2020. *Feature Engineering and Selection: A Practical Approach for Predictive Models*. CRC Press.

McInnes, L, J Healy, and J Melville. 2020. "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction."

Mingqiang, Y, K Kidiyo, and R Joseph. 2008. "A Survey of Shape Feature Extraction Techniques." In *Pattern Recognition*, edited by PY Yin. Rijeka: IntechOpen. https://doi.org/10.5772/6237.

Symons, S, and RG Fulcher. 1988. "Determination of Wheat Kernel Morphological Variation by Digital Image Analysis: I. Variation in Eastern Canadian Milling Quality Wheats." *Journal of Cereal Science* 8 (3): 211–18.

20. The **learntidymodels** package can be found at its GitHub site↩

21. The **mixOmics** package is not available on CRAN, but instead on Bioconductor.↩