Capstone, Image classification through shallow and deep learning

Santiago 1975

2025-05-04

1.Introduction

The HarvardX Professional Certificate in Data Science provided an excellent foundation in R programming as applied to statistics, linear modeling, and machine learning. The capstone project provided the opportunity to expand on this foundation through independent learning and experimentation. While image classification was touched upon in the machine learning course (Irizarry, 2024) and textbook (Irizarry, 2019) through the `mnist` example, I had a profound desire to learn more about image classification through deep learning. Although deep learning has historically been performed primarily using Python, TensorFlow for R and torch for R have made the techniques directly available to R users. As the name implies, "deep" learning models contrast with the "shallow" models covered in the machine learning course. The overarching goal of this capstone project was to compare the image classification capabilities of shallow and deep learning models.

As an agricultural researcher, I was also interested in working with a data set relevant to my field of study. Through Kaggle, a data set of images was selected pertaining to the leaves of common bean (*Phaseolus vulgaris*). Images had been labeled as one of three classes: healthy, infected with angular leaf spot, or infected with rust. I was particularly interested in a multi- rather than binary classification challenge and while 4 to 5 classes would have been ideal, the 3 classes in the data set proved to be a good starting place.

Since 2020, upwards of 28 million tons of common bean have been produced per year (FAO, 2025). This equates to greater than 3 kilograms for every person on the planet. Beans are members of the legume family, *Fabaceae*. Like most members of this family, common beans acquire nitrogen through an association with rhizobia (NB: nitrogen-fixing bacteria). As such, beans also have soil regenerating capacities. Angular leaf spot (ALS), caused by the fungus (*Phaeoisariopsis griseola*), is a serious disease of beans. While aerial parts of the plant such as leaves, petioles, stems and pods are affected; symptoms are most recognizable on leaves. Lesions appear as brown spots with a tan or silvery center. Lesions are initially confined to tissue between major veins which gives them their characteristic "angular" appearance. Yield losses due to a reduction in photosynthetic area of 10-50% have been reported in the U.S. and up to 80% in tropical and subtropical countries (Celetti *et al.* 2005). With knowledge of the disease, strategically applied fungicidal treatments are possible.

Bean rust is also caused by a fungus. *Uromyces phaseoli typica* is present throughout the world and can destroy an entire crop under favorable conditions. It can be especially severe in humid areas with moderate temperatures. Rust can occur on all above-ground parts but are most numerous on the undersides of leaves. Lesions are initially small, white, and slightly raised, but later become reddish brown. In field trials, rust reduced yield more than four times that of angular leaf spot (De Jesus *et al.* 2001). With early detection, treatment options are available. Image classification could provide farmers with a tool for the early detection and classification of fungal disease. Provided through a smart phone or similar device, image classification could be especially useful to early season management and planning prior to subsequent seasons.

The project had the following goals:

- Perform image classification pertinent to an agriculturally relevant topic

- Learn the basics of deep learning, tensors, and torch
- Compare the results of deep learning with shallow machine learning models like KNN
- Develop a foundation in image analysis for future exploration of increasingly complex problems

2.Methods

Data acquisition and exploration

Data was obtained from the Kaggle website as Bean Leaf Lesions Classification. With a valid Kaggle Token saved as `kaggle.json` in the working directory, the following code downloads the compressed data file and extracts the contents to a directory named `data`. For more information on obtaining a Kaggle token, read the "Authorization" section on the How to use Kaggle page.

```
credentials <- jsonlite::read_json("kaggle.json")

request(paste0("https://www.kaggle.com/api/v1/datasets/download/",
               "marquis03/bean-leaf-lesions-classification")) |>
  req_auth_basic(credentials$username, credentials$key) |>
  req_perform(path = "bllc_data.zip")

unzip("bllc_data.zip", exdir = "data")

file.remove("bllc_data.zip")
```

The `data` directory requires about 155 MB of disk space and consists of 1,171 files arranged in 8 sub-directories. Images are first split into `train` and `val` directories. Within each of these, images are further split into three directories representing each of three classes of bean leaves: "angular_leaf_spot", "bean_rust", and "healthy". Two indexing files, `train.csv` and `val.csv` are also included in the directory. The indexing files are useful in managing the "batch" reading of images for modeling. The indexing files were adjusted slightly using the following code. The path was adjusted to the environment created upon downloading the images. Integer labels were adjusted from [0, 1, 2] to [1, 2, 3] for modeling. The actual class names were also added for easy referencing.

```
path_r <- paste(getwd(), "data", sep = "/")
file_r <- "train.csv"

df_train_index <- read_csv(paste(path_r, file_r, sep = "/"),
                           show_col_types = FALSE) %>%
  rename(path = `image:FILE`,
         label = category) %>%
  mutate(path = paste(path_r, path, sep = "/")) %>%
  mutate(label = factor(label + 1, levels = c(1:3))) %>%
  mutate(class = case_when(label == 1 ~ "healthy",
                           label == 2 ~ "angular_leaf_spot",
                           label == 3 ~ "bean_rust"))
```

A few entries of the training index are shown for interest. The integer `label` and character `class` represent the modeling "target". The `path` leads to the images that are the ultimate source of the modelling "features".

Table 1. A few entries from the training image index.

path	label	class
G:/My Drive/jw_consu...../data/train/healthy/healthy_train.98.jpg	1	healthy
G:/My Drive/jw_consu.....eaf_spot/angular_leaf_spot_train.233.jpg	2	angular_leaf_spot
G:/My Drive/jw_consu.....a/train/bean_rust/bean_rust_train.87.jpg	3	bean_rust

There were 1034 and 133 images in the training and validation sets respectively. For classification models, the prevalence of each class is important to both training and testing. Class imbalance is often difficult to overcome. The prevalence of each class within the training images is presented in **Figure 1**. Each of the three classes has nearly the same prevalence such that class imbalance should not factor into modeling, nor the evaluation of model performance.

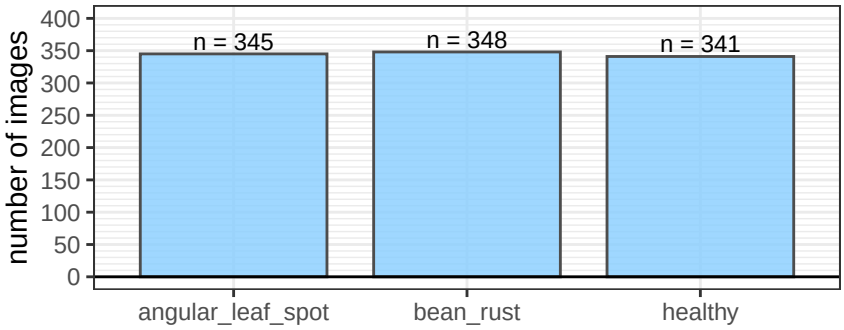

Figure 1. The distribution of images between the three classes was "balanced".

Downloaded images had an initial resolution of 500 x 500 pixels. If vectorized and expressed in wide format for modeling, each of the 1,171 observations in the training set would have up to 750,000 individual features (NB: 500 x 500 x 3 RGB). For a number of reasons, even attempting a modeling of such dimensions would be imprudent. In the machine learning course (Irizarry, 2024), the 28 x 28 pixel (NB: 784 features) gray scale `mnist` data set was assessed using KNN with good success. While neural networks are capable of handling much greater dimensions, 224 x 224 pixels (50,176 features) is a commonly used standard. The 150,528 individual features (NB: 224 x 224 x 3 RGB), however, was still a bit cumbersome for a 32 GB RAM, 10 core CPU. As such, an intermediate resolution to compare both KNN and CNN on a fairly level playing field was desired. Based on simple empirical measures of the duration of KNN training with cross validation, a resolution of 128 x 128 pixels (16,384 feature) was ultimately selected. How the RGB channels were handled is discussed in the modeling approach.

With an image resolution decided and the indexes in place, individual images could be easily accessed and scaled for visualization. The following script pulls a few images from each class and displays them as a panel in **Figure 2**.

```
eda_img <- df_train_index %>%
  group_by(class) %>%
  slice(1:3) %>%
  select(path, class)

img_list <- image_join()

for(i in 1:length(eda_img$path)){
  img_list <- image_read(eda_img$path[i]) %>%
```

```
    image_scale(., "128x128") %>%
    image_annotate(., paste0("_", eda_img$class[i], "_"),
                   location = "+5+5", font = 'Arial', size = 10,
                   color='black', boxcolor = "white") %>%
    image_join(., img_list)
}

image_montage(img_list,
              geometry = 'x120+10+10',
              tile = '3x3')
```

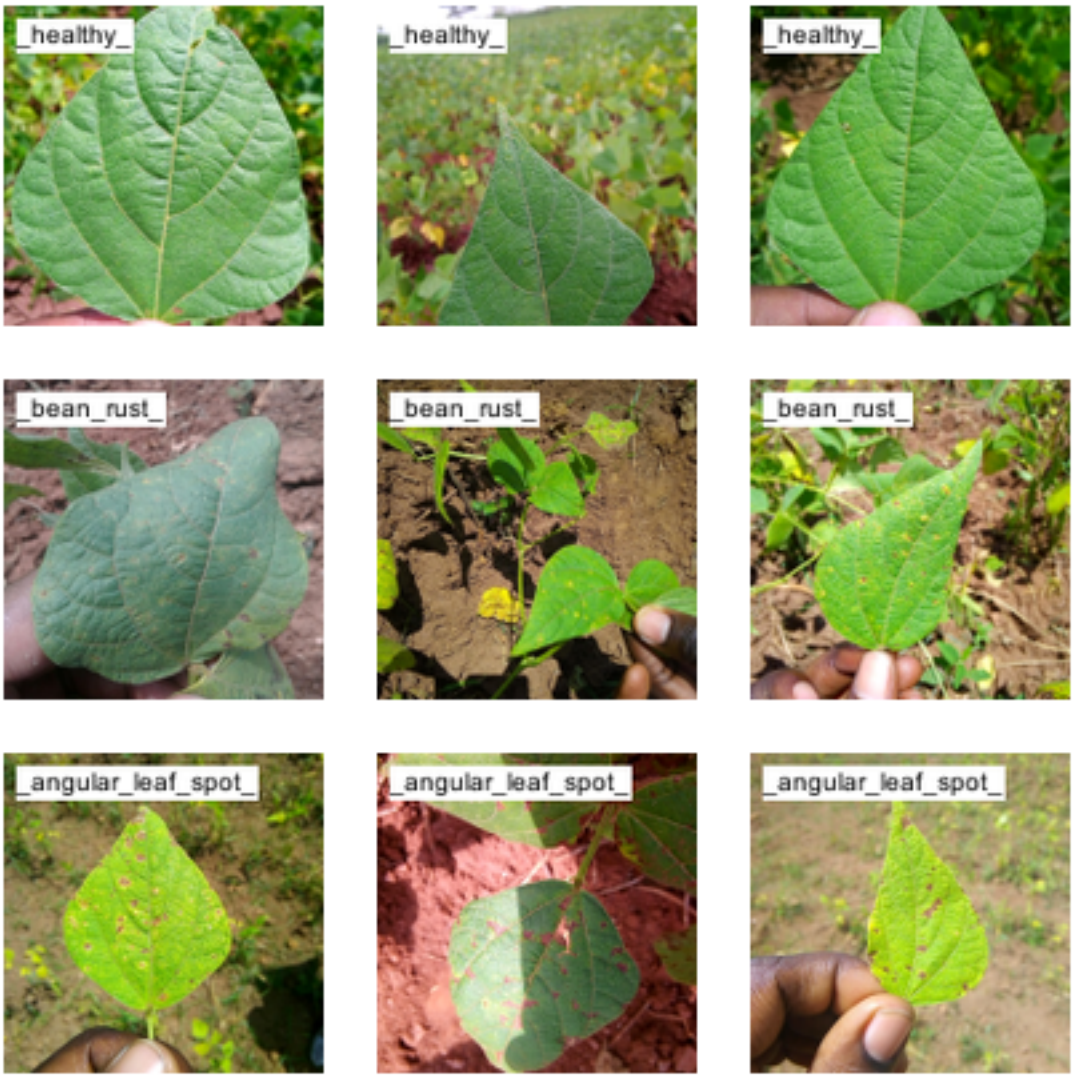

Figure 2. Example images from each of the three classes.

From a brief look at the images, one notices the variation in leaf size, leaf coloration, leaf orientation, leaf shape, background coloration, leaf-to-background proportion, shadows, the presence of fingers, and other characteristics that make this a more difficult classification challenge than that of the `mnist` data set. On the other hand, even to the untrained eye the separation of 'healthy' and diseased leaves seems fairly straightforward. The differentiation between 'bean_rust' and 'angular_leaf_spot', however, appears more difficult.

In fact, the differentiation between 'healthy' and 'diseased' leaves might be an alternative way of evaluating models should such delineation be of practical importance to growers.

Modeling approach

To better understand the abilities of deep neural networks, a series of shallow K nearest neighbor (KNN) models were first produced for comparison. In traditional KNN approaches like those applied to the `mnist` data set, the 2-dimensional XY data of the image is first transformed into a 1-dimensional vector. One can imagine a loss in "spatial" information value accompanying this transformation.

A color image can be envisioned as 3-dimensional XYZ data with the third dimension consisting of red, green, and blue (RGB) 2-dimensional layers. In transforming color images for KNN, several approaches could be taken. First, the 3-dimensional color image can be transformed to a 1-dimensional vector. In addition to losing spatial information for each color channel, such transformations result in extremely high feature numbers. Second, the RGB channels can first be converted to single value gray-scale before transforming the 2-dimensional XY data to a 1-dimensional vector. While the number of resulting features is reduced, information is lost in both the conversion to gray-scale and the spatial transformation. Here, we employed a third approach. First, a KNN gray-scale model was developed as just described. Additionally, separate KNN models were then developed for each of the red, green, and blue channels. The four KNN models were assessed independently, but also combined into an ensemble.

In taking shallow models a step further, principal component analysis (PCA) was then combined with KNN for a second series of models. For the gray-scale and RGB channels, the 1-dimensional vector of predictors was first distilled down to the ten most influential principal components using a randomized PCA algorithm. The principal components were then used to train KNN models. As was done for the KNN approach, the four PCA-KNN models were assessed independently, but also combined into an ensemble.

With a number of shallow models developed, a series of convolutional neural networks (CNN) were then performed for comparison. First, a custom 7-layer CNN was attempted using the 2-dimensional gray-scale images as input. The same model was then repeated with the 3-dimensional RGB images as input. Finally, a well-known, pre-trained CNN, ResNet-18, was evaluated using the 3-dimensional RGB images as input.

The methods and code underlying the three modeling approaches are discussed in more detail in the following sections.

KNN A conventional KNN approach was used to create four models from which a fifth ensemble was also made. The names of the 5 models were:

1. **knn_gray** KNN model trained on 16,384 length vector of normalized gray-scale pixel values
2. **knn_red** KNN model trained on 16,384 length vector of normalized red channel pixel values
3. **knn_green** KNN model trained on 16,384 length vector of normalized green channel pixel values
4. **knn_blue** KNN model trained on 16,384 length vector of normalized blue channel pixel values
5. **knn_ensemble** A majority class ensemble of models 1 to 4

For each of the non-ensemble models, images were transformed to numerical vectors in the same general manner. The following script for `knn_gray` outlines the basic process. An empty object was first defined to hold a matrix of values. A `for loop` was then set up to step through one of the previously defined indexes (NB: train or val). At each step in the loop, the image was read, re-scaled, converted to gray-scale in this case, transformed to a tensor, flattened into a vector, and appended to the matrix. Its important to note that in expressing the 3rd tensor dimension (NB: tensor dimensions indexed from left-to-right) as numeric, values of pixel intensity were converted from 2-dimensional arrays to 1-dimensional vectors by row. At the same time, values of pixel intensity that initially ranged from 0 to 255 were scaled from 0 to 1. While good practice for KNN, scaling such as this is considered integral to neural networks. Row and column names were then formatted prior to combining into a data frame with the target labels from the index. For RGB

channels, the `image_convert()` step was omitted. It was then simply a manner of selecting either the first (red), second (green), or third (blue) tensor layer for vector transformation.

```
train_knn_gray <- NULL

for(i in 1:nrow(df_train_index)){
  tensor_local <- image_read(df_train_index$path[i]) %>%
    image_scale(., "128x128") %>%
    image_convert(colorspace = "Gray") %>%
    transform_to_tensor()

  train_flat <- as.numeric(tensor_local[1,,])
  train_knn_gray <- rbind(train_knn_gray, train_flat)
}

rownames(train_knn_gray) <- paste("img",
                                  str_pad(1:dim(train_knn_gray)[1], width = 4, pad="0"),
                                  sep = "_")
colnames(train_knn_gray) <- paste("x",
                                  str_pad(1:dim(train_knn_gray)[2], width = 4, pad="0"),
                                  sep = "_")

df_train_knn_gray <- data.frame(y = df_train_index$label, train_knn_gray) %>%
  mutate(y = factor(y)) %>%
  remove_rownames(.)
```

With the training and validation data frames created, cross-validation was used to select an optimum k value. Using the `knn_gray` as an example, the following script outlines the 5-fold cross-validation step with 80/20 train/test split. A seed was set for replication by other users. Values between 10 and 100 were surveyed to identify the best k value. The model was then trained and the plot of accuracy as a function of the number of neighbors (k) evaluated visually. The optimum k was saved for downstream operations.

```
set.seed(5446)

control <- trainControl(method = "cv",
                        number = 5,
                        p = 0.8)

train_knn <- train(y ~ .,
                   data = df_train_knn_gray,
                   method = "knn",
                   tuneGrid = data.frame(k = seq(10, 100, by = 10)),
                   trControl = control)

ggplot(train_knn, highlight = TRUE) +
  scale_x_continuous(breaks = seq(10, 100, by = 10),
                     minor_breaks = seq(0, 100, by = 5)) +
  theme_bw()

k_knn_gray <- train_knn$bestTune
```

The optimum k value from cross-validation was then used to re-train the model on the entire training set. Predictions for the model were then made on the validation set and the confusion matrix saved for later analysis. Overall accuracy was the primary metric evaluated across the KNN models.

```
set.seed(5446)

train_knn <- knn3(y ~ .,
                  data = df_train_knn_gray,
                  k = k_knn_gray)

y_hat_knn_gray <- predict(train_knn,
                          df_val_knn_gray,
                          type = "class")

cm_knn_gray <- confusionMatrix(y_hat_knn_gray,
                               factor(df_val_knn_gray$y))
```

After the gray-scale and RGB models were developed, an ensemble of the four models was constructed. The following code outlines testing of `knn_ensemble` on the validation set. Target values from the validation index were used as a scaffold for adding the predictions of the four models making up the ensemble. A few data frame manipulations were performed to determine the majority class of the four models. In the case of ties, the observation was assigned to the lower integer class of the tie. In this way, ties involving 'healthy' predictions and either of the other two classes were considered 'healthy', and ties between 'bean_rust' and 'angular_leaf_spot' were considered 'bean_rust'. The confusion matrix of the ensemble was saved for evaluation.

```
y_hat_knn_ensemble <- df_val_index %>%
  select(-path) %>%
  mutate(y_gray = y_hat_knn_gray,
         y_red = y_hat_knn_red,
         y_green = y_hat_knn_green,
         y_blue = y_hat_knn_blue) %>%
  mutate(observation = row_number()) %>%
  gather(key = "key", value = "value", -c(label, class, observation)) %>%
  group_by(observation, label, value) %>%
  summarize(n = n()) %>%
  group_by(observation) %>%
  arrange(desc(n)) %>%
  group_by(observation) %>%
  slice(1) %>%
  arrange(observation) %>%
  pull(value)

cm_knn_ensemble <- confusionMatrix(factor(y_hat_knn_ensemble),
                                   factor(df_val_index$label))
```

A simple summary of the five KNN models is shown in **Table 2**. Between themselves, `knn-ensemble` exhibited the highest accuracy of 52.6%. Although relatively low, this level of accuracy was not surprising given the criterion on which the images were being classified and modeling techniques themselves. The confusion matrix of `knn_ensemble` is discussed in more detail relative to the top performing PCA-KNN and CNN models.

Table 2. Summary of KNN model performance

id	model	k	accuracy
1	knn_gray	90	0.504

id	model	k	accuracy
2	knn_red	100	0.414
3	knn_green	80	0.504
4	knn_blue	100	0.444
5	knn_ensemble		0.526

PCA-KNN Following the conventional KNN approach, a principal component analysis KNN (PCA-KNN) approach was used to create four models from which a fifth ensemble was also made. The names of the 5 models were:

6. **pca_gray** PCA reduction of normalized gray-scale pixel values to 10 principal components for KNN
7. **pca_red** PCA reduction of normalized red channel pixel values to 10 principal components for KNN
8. **pca_green** PCA reduction of normalized green channel pixel values to 10 principal components for KNN
9. **pca_blue** PCA reduction of normalized blue channel pixel values to 10 principal components for KNN
10. **pca_ensemble** A majority class ensemble of models 6 to 9

Principal components were computed from the training matrices created during KNN modeling. Due to the large size of the matrices, a randomized PCA algorithm was used in place of traditional algorithms such as `prcomp` that can suffer from a lack of memory and long computational times. Only the top 10 principal components were determined and used for downstream modeling. As an example, the following script outlines the basic process used for the `pca_blue` model. The `pca` function from the `mdatools` package significantly accelerates computation after specifying the calculation of 10 components, an oversampling parameter of 5, and an iteration number of 1. Principle components were cleaned up and converted to a data frame for downstream modeling steps. For all training sets, the fraction of variance for the first 10 components was saved for analysis..

```
set.seed(5446)

m2_train = pca(train_knn_blue, ncomp = 10, rand = c(5, 1))

var_pca_blue <- as.numeric(m2_train$res$cal$cumexpvar[10])

train_pca_blue <- m2_train$res$cal$scores

rownames(train_pca_blue) <- paste("img",
                                  str_pad(1:dim(train_pca_blue)[1], width = 4, pad="0"),
                                  sep = "_")
colnames(train_pca_blue) <- paste("x",
                                  str_pad(1:dim(train_pca_blue)[2], width = 4, pad="0"),
                                  sep = "_")

df_train_pca_blue <- data.frame(y = df_train_index$label, train_pca_blue) %>%
  mutate(y = factor(y)) %>%
  remove_rownames(.)
```

For interest, the first two principal components of the gray-scale and RGB models were visualized based on their class labels. While the separation of classes wasn't remarkable, developing patterns in the distribution of class labels was apparent in some cases. A plot from the blue-channel PCA is shown in **Figure 3**. While 'healthy' observations dominate the lower-left corner of the plot, a predominance of 'angular_leaf_spot' can

be observed in the upper-right corner. Observations of 'bean_rust' are concentrated in the center of the plot, but with significant overlap with 'angular_leaf_spot'. When considering this is only two components of the 16,384 initial variables, visually observing any patterns in the data was actually quite remarkable.

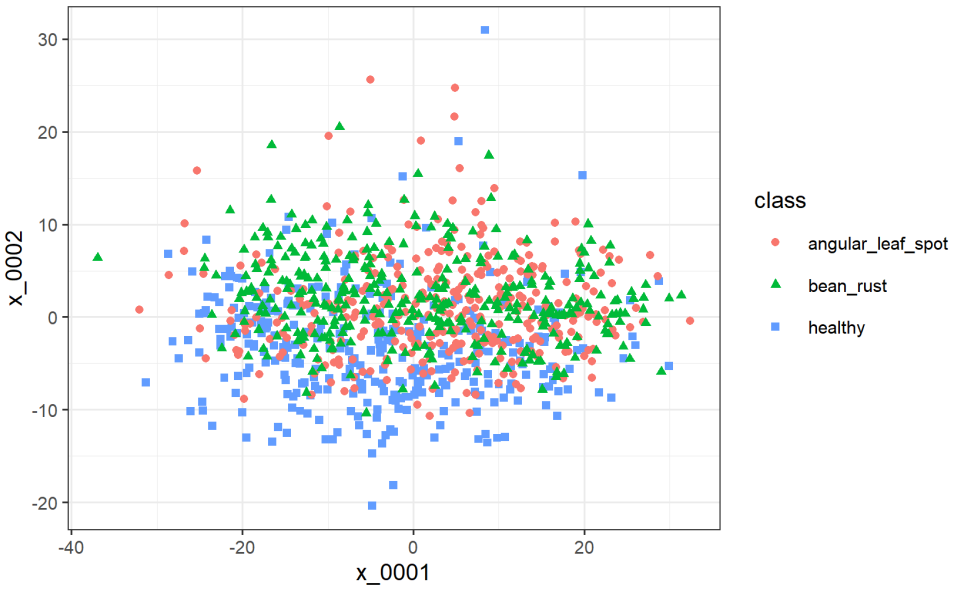

Figure 3. The first two principal components of the green-channel vector of predictors

Rather than calculating the principal components of the validation set, the first 10 components were predicted from the loadings of the training set. The following code provides the basic process used for the gray-scale and RGB channels. As for the training sets, results were converted to data frames prior to modeling.

```
m2_val <- predict(m2, val_gray)

val_pca_gray <- m2_val$scores

rownames(val_pca_gray) <- paste("img",
                                str_pad(1:dim(val_pca_gray)[1], width = 4, pad="0"),
                                sep = "_")
colnames(val_pca_gray) <- paste("x",
                                str_pad(1:dim(val_pca_gray)[2], width = 4, pad="0"),
                                sep = "_")

df_val_pca_gray <- data.frame(y = val_index$label,
                              val_pca_gray) %>%
  mutate(y = factor(y)) %>%
  remove_rownames(.)
```

After PCA, modeling was carried out using the same KNN modeling strategy as described in the preceding section. Briefly, cross-validation was used to first determine an optimum k value. KNN models were then fit and evaluated separately. As was performed for the KNN models, a majority rules ensemble was also developed using the gray-scale and RGB models as input.

A simple summary of the five PCA-KNN models is shown in **Table 3**. Between themselves, `pca-blue` exhibited the highest accuracy of 61.7%. This was quite remarkable considering the reduction from 16,384 features to just 10 principal components. It is important to note, however, that those ten components made

up nearly 55% of the cumulative explained variance. The confusion matrix of `pca_blue` is discussed in more detail relative to the top performing KNN and CNN models.

Table 3. Summary of PCA-KNN model performance

id	model	per_var	k	accuracy
6	pca_gray	48.567	30	0.519
7	pca_red	53.181	80	0.489
8	pca_green	49.807	20	0.481
9	pca_blue	54.799	60	0.617
10	pca_ensemble			0.586

CNN For comparison with the shallow models, three convolutional neural network (CNN) models were developed using Torch for R. The names of the 3 models were:

11. **cnn_gray** 7-layer custom designed CNN trained on 128x128 normalized gray-scale pixel values
12. **cnn_rgb** 7-layer custom designed CNN trained on 128x128x3 normalized RGB pixel values
13. **cnn_resnet** 18-layer pre-trained image classification CNN fine-tuned to 128x128x3 normalized RGB pixel values

Deep Learning was not covered in the machine learning course (Irizarry, 2024). The CNN techniques and Torch for R workflows applied here were learned through the book, "Deep learning and scientific computing with R torch" (Keydana, 2023). To work with torch, `dataset()`'s and `dataloader()`'s need to be set up. A `dataset()` is an R6 object requiring initialization, indexing, and contents methods. As an example, the following script demonstrates the `dataset()` and `dataloader()` used for the training set of the gray scale CNN models. A 'data set generator' is first defined. The previously developed index is used for initialization. Functions for accessing the x and y parameters of the model are then developed. Note the use of `unsqueeze()` and `squeeze()` in getting the different dimension tensors aligned. Finally, a function for determining the number of rows in the data set is provided. The data set generator was then used to create the data set. The newly created data set was then passed to the `dataloader()` function along with a batch size. A batch size of 64 was used throughout the modeling phase. A single batch was always test loaded to make sure the x and y dimensions corresponded to model setup. For CNN, only gray-scale and complete RGB were tested. In fact, the ability to simultaneously handle the RGB channels was one of the main advantages of neural networks.

```
train_dsg <- dataset(
  name = "train_dsg",
  initialize = function(train_index) {
    self$train <- train_index
  },
  .getitem = function(i) {
    x1 <- image_read(self$train$path[i]) %>%
      image_scale(., "128x128") %>%
      image_convert(colorspace = "Gray") %>%
      transform_to_tensor()
    x <- x1[1]$unsqueeze(1)
    y <- torch_tensor(self$train$label[i])$squeeze(1)
    list(x = x, y = y)
  },
  .length = function() {
```

```
    nrow(self$train)
  }
)

train_ds <- train_dsg(train_index)

train_dl <- dataloader(train_ds,
                       batch_size = 32,
                       shuffle = TRUE)

batch <- train_dl %>%
  dataloader_make_iter() %>%
  dataloader_next()

batch[[1]]$size()
batch[[2]]$size()
```

Two neural network configurations were used. The first, a custom defined network with an input layer, 7 hidden layers, and an output layer. The input layer consisted of either 1 (NB: gray-scale) or 3 (NB: RGB) nodes and a output layer of 3 nodes corresponding to the 3 classes being predicted. Within the hidden layers there was a shift-equivariant "feature detector" of 4 convoluted layers with nodes increasing as 128, 256, 512, and 1024, respectively; a shift-invariant pooling layer; and a 2 layer feed-forward neural network that reduced the 1024 nodes to the final scores of the three classes. The code initializing the model is shown below.

```
convnet <- nn_module(
  "convnet",
  initialize = function(){
    self$features <- nn_sequential(
      nn_conv2d(3, 128, kernel_size = 3, padding = 1),
      nn_relu(),
      nn_max_pool2d(kernel_size = 2),
      nn_dropout2d(p = 0.05),
      nn_conv2d(128, 256, kernel_size = 3, padding = 1),
      nn_relu(),
      nn_max_pool2d(kernel_size = 2),
      nn_dropout2d(p = 0.05),
      nn_conv2d(256, 512, kernel_size = 3, padding = 1),
      nn_relu(),
      nn_max_pool2d(kernel_size = 2),
      nn_dropout2d(p = 0.05),
      nn_conv2d(512, 1024, kernel_size = 3, padding = 1),
      nn_relu(),
      nn_max_pool2d(kernel_size = 2),
      nn_dropout2d(p = 0.05),
      nn_conv2d(1024, 1024, kernel_size = 3, padding = 1),
      nn_relu(),
      nn_adaptive_avg_pool2d(c(1, 1)),
      nn_dropout2d(p = 0.05)
    )
    self$classifier <- nn_sequential(
      nn_linear(1024, 1024),
      nn_relu(),
```

```
      nn_linear(1024, 1024),
      nn_relu(),
      nn_linear(1024, 3)
    )
  },
  forward = function(x){
    x <- self$features(x)$squeeze()
    x <- self$classifier(x)
  }
)
```

A pre-trained model, ResNet18, was also employed. ResNet18 is an 18-layer CNN commonly used for image classification. It was trained on the ImageNet dataset and is capable of classifying images into 1000 object categories. The model was obtained through the `torchvision` package. The following code demonstrates it's initialization. The most significant change needed was to the feed-forward neural network reducing the number of nodes to the final scores of the three classes.

```
convnet <- nn_module(
  initialize = function(){
    self$model <- model_resnet18(pretrained = TRUE)
    for(par in self$parameters){
      par$requires_grad_(FALSE)
    }
    self$model$fc <- nn_sequential(
      nn_linear(self$model$fc$in_features, 1024),
      nn_relu(),
      nn_linear(1024, 1024),
      nn_relu(),
      nn_linear(1024, 3)
    )
  },
  forward = function(x){
    self$model(x)
  }
)
```

After initialization, the desired model was setup. In this step, both the loss function and optimizer were defined. Cross entropy loss and the Adam optimizer were used. Both are standard for neural networks and image classification. The Adam optimizer combines the momentum and RMSprop techniques. This is thought to provide a more balanced and efficient optimization process in comparison to other techniques. Addition of the accuracy metric to the modeling log was also performed for reporting.

```
model <- convnet %>%
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_adam,
    metrics = list(
      luz_metric_accuracy()
    ))
```

Before the model was fit, a maximum learning rate was estimated. Learning rate is integral to training performance. With too small of an update, the model might achieve a minimum loss, but take an excessively long time to do so. With too large of an update, the minimum can be "jumped" and loss can be come infinite.

Torch for R contains a built-in function `lr_finder()` which assists in selecting a good learning rate. As demonstrated in the following code, the model and the training data loader are subject to the function.

```
lr_cnn_resnet <- model %>%
  lr_finder(train_dl)
```

The output from the learning rate function is illustrated in **Figure 4.** using `cnn_resnet` as an example. Loss was plotted against the log of learning rate. The untransformed learning rates were displayed as labeled points on the curve. Here, the last point after loss became noisy, but before loss exploded was taken. The value was rounded to the nearest tenth or hundredths place and used as the 'maximum' learning rate upon fitting the model for the first time. In some cases, a lower learning rate was later applied to improve performance.

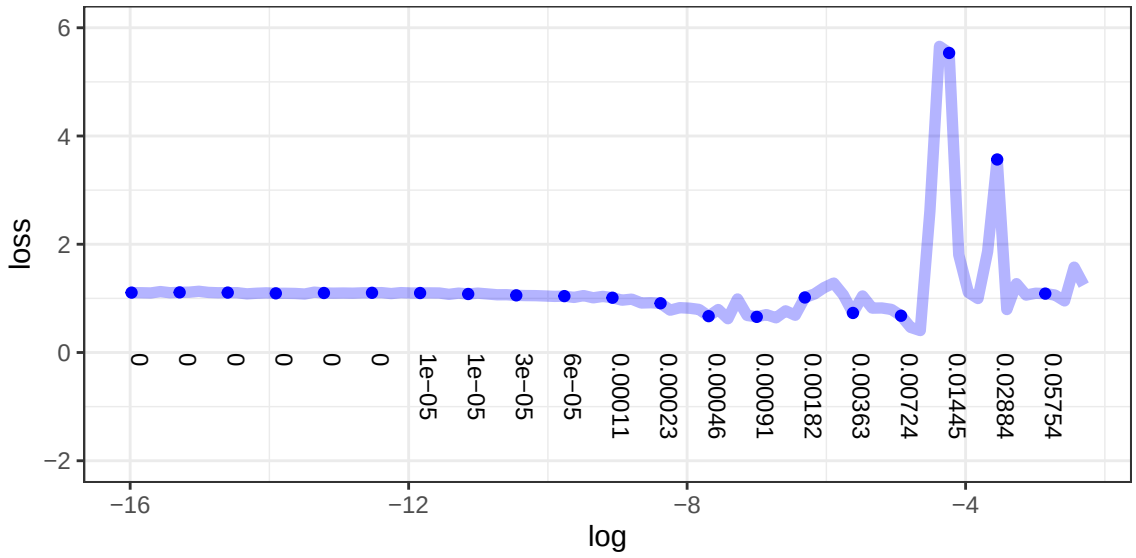

Figure 4. Plot of loss as a function of learning rate for `cnn_resnet` model

The following script provides an example of model fitting. The training data loader was called and the number of epochs set to 50. A few additional arguments improved model training and facilitated assessment. A patience setting of 5 stopped the model early if loss failed to decrease for 5 consecutive epochs. The maximum learning rate from the preceding step was also set. Model checkpoints were saved along with an epoch-by-epoch log of loss and accuracy for training and test sets. This information was essential to identifying the epoch of maximum model performance.

```
fitted <- model %>%
  fit(
    train_dl,
    epochs = 50,
    valid_data = val_dl,
    callbacks = list(
      luz_callback_early_stopping(patience = 5),
      luz_callback_lr_scheduler(
        lr_one_cycle,
        max_lr = 0.001,
        epochs = 50,
        steps_per_epoch = length(train_dl),
```

```
        call_on = "on_batch_end"),
      luz_callback_model_checkpoint(path = "results/cnn_resnet/checkpoints/"),
      luz_callback_csv_logger("results/cnn_resnet/logs_resnet.csv")
    ),
    verbose = TRUE)
```

The epoch-by-epoch log of accuracy and loss for `cnn_resnet` is shown in **Figure 5** as an example. Note that metrics are recorded for both the training and validation data sets. It can be seen that through 20 epochs, validation loss had reached a minimum and accuracy a maximum. Without "early stopping" the model would have "over-trained". Training loss would have decreased to 0 and accuracy increase to 1 without benefit to validation metrics.

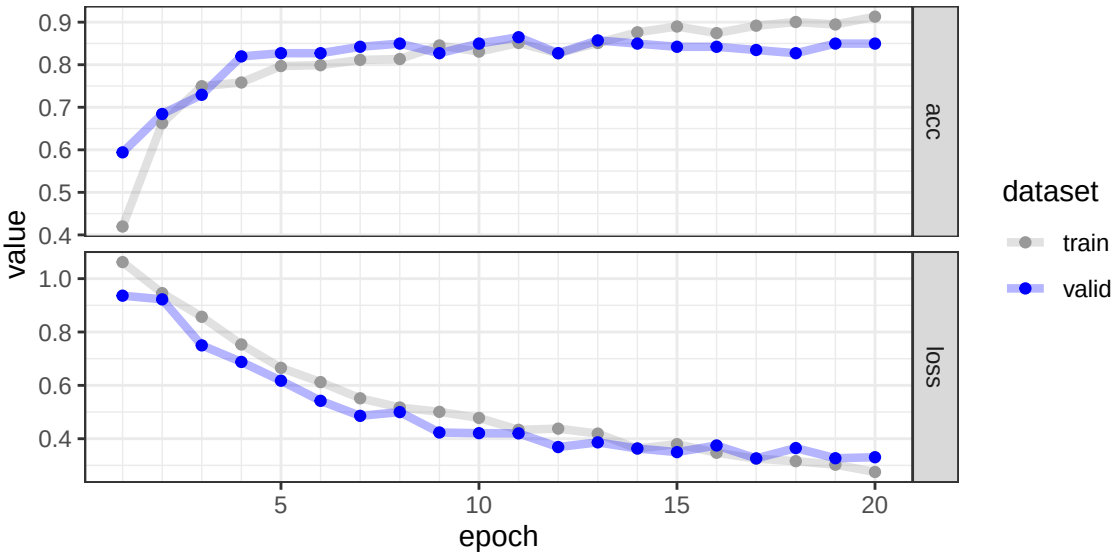

Figure 5. Model accuracy and loss as a function of epoch for `cnn_resnet` model

The log was used to select a specific epoch on which to evaluate model performance. The following example from `cnn_resnet` first selected the epoch with the maximum accuracy. A pattern was then created to get the right file name for the saved models. The `fitted` model was then replaced with the selected model using the `luz_load_checkpoint()` function. A prediction was then made from the validation set. The `nnf_softmax` and `torch_argmax` functions were used to transform the result from class-based probabilities to the same integer factors used in the indexing data frames. In this way, a confusion matrix was created in the same manner as for the KNN and PCA-KNN models. Results were saved for comparison.

```
epoch_cnn_resnet <- log %>%
  filter(set == "valid") %>%
  filter(acc == max(acc))

pattern <- paste0("epoch-", epoch_cnn_resnet$epoch)
path_r <- paste(getwd(), "results/cnn_resnet/checkpoints", sep = "/")
epoch_oi <- list.files(path = path_r, pattern = pattern)

luz_load_checkpoint(fitted,
                    paste(getwd(), "results/cnn_resnet/checkpoints", epoch_oi, sep = "/"))

y_hat_cnn_resnet <- fitted %>%
```

```
  predict(val_dl) %>%
  nnf_softmax(., dim = 2) %>%
  torch_argmax(., dim = 2) %>%
  as.numeric(.) %>%
  factor(., levels = c(1:3))

cm_cnn_resnet <- confusionMatrix(y_hat_cnn_resnet, df_val_index$label)
```

A simple summary of the three CNN models is shown in **Table 4**. Between themselves, `cnn-resnet` exhibited by far the highest accuracy of 86.5%. From the remarkable difference between the ResNet and custom CNN, the author's lack of experience in formulating neural networks was apparent. In fact, `cnn_gray` performed worse than a number of the KNN-based approaches. Nonetheless, the nearly 14% increase in accuracy between `cnn_gray` and `cnn_rgb` demonstrated the importance of using all the information in the RGB channels together. The confusion matrix of `cnn-resnet` is discussed in more detail relative to the top performing KNN and PCA-KNN models.

Table 4. Summary of CNN performance

id	model	input	type	layers	learn_rate	epoch	loss	accuracy
11	cnn_gray	gray-scale	custom	7	0.001	25	0.951	0.541
12	cnn_rgb	rgb	custom	7	0.050	17	0.801	0.677
13	resnet_18	rgb	pre-trained	18	0.001	11	0.420	0.865

3.Results

For each approach (KNN, PCA-KNN, & CNN), the model with the highest accuracy was selected for comparison. The precision, recall, and F1 by class for `knn_ensemble`, `pca_blue`, and `cnn_resnet` is presented in **Figure 6**. Across all metrics and classes, `knn_resnet` outperformed the KNN-based approaches. For 'healthy' observations (NB: class 1), the precision of the KNN-based approaches was surprisingly high at ~70%. This suggested that predictions of health were generally correct for these models. Nonetheless, the ~50% recall of the KNN-based approaches suggested that about half of healthy observations were being incorrectly classified as diseased.

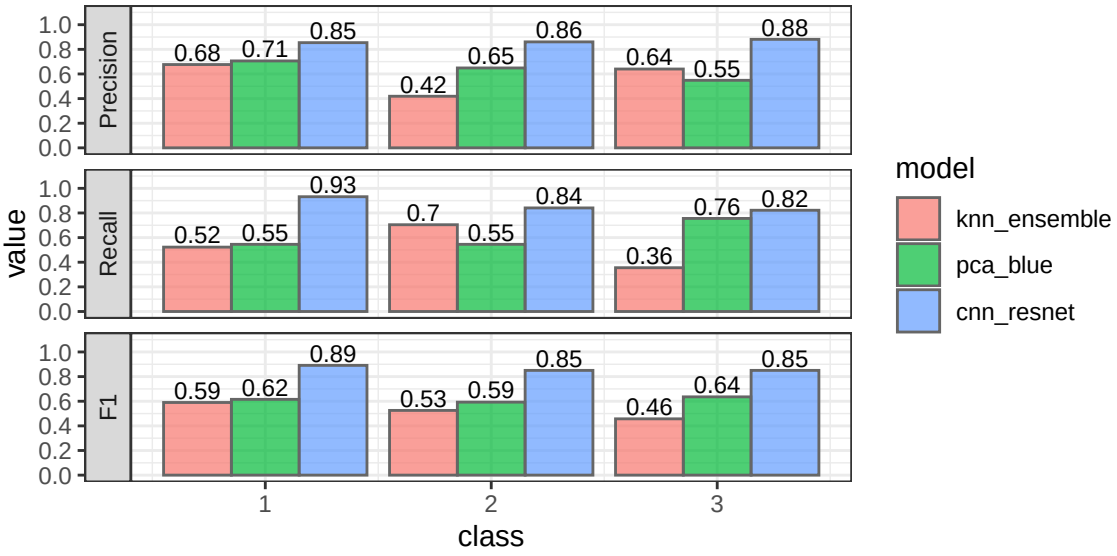

Figure 6. Precision, recall, and F1 score by class for the select models

The inversely correlated class 2/3 precision/recall metrics of the KNN-based approaches was also of interest. The observation was, perhaps, easier to explain in the context of the model-by-model confusion matrices presented in **Figure 7**. While both KNN-based approaches predicted class 1 with similar frequencies, `knn_ensemble` tended to predict class 2 and `pca_blue` tended to predict class 3. This phenomenon elevated recall in the 'over-predicted' class, but at the expense of recall. As a result, neither KNN-based approach could match the prediction accuracy reflected in the `cnn_resnet` model.

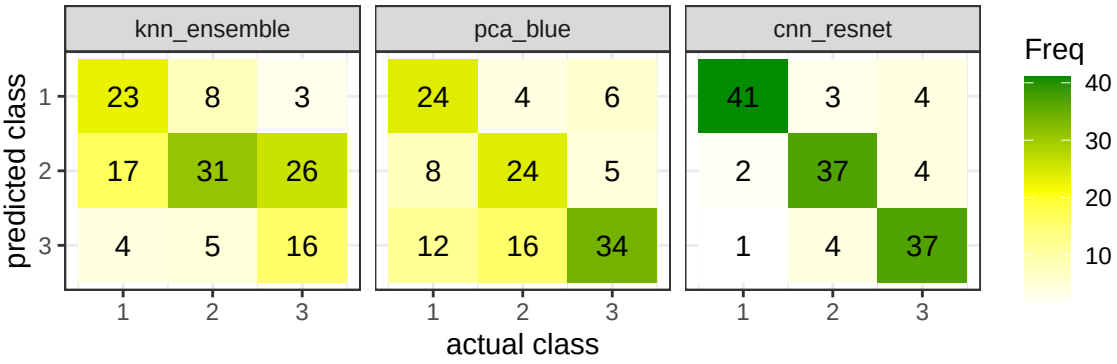

Figure 7. Confusion matrices for the select models

4.Conclusion

It was no surprise that the CNN models outperformed KNN-based approaches in the classification of bean leaf lesions. As alluded to earlier, the images possessed a number of complex attributes that were seemingly difficult for the shallow KNN-based models to capture. The "out-of-the-box" performance of the pre-trained ResNet-18 model was nonetheless surprising. While the custom 7-layer `cnn_rgb` model outperformed KNN-based approaches, it fell far short of `cnn_resnet` in terms of accuracy.

The ~87% accuracy of the ResNet model is approaching utility in terms of the classification of bean leaf lesions. Another 5-10% increase could make it a valuable tool to bean farmers. From the perspective of modeling there is still much room for improvement. This was the author's first attempt at image classification with CNN and torch itself. While the custom models could certainly be improved through more layers, nodes, dropout, regularization, learning rates, optimizers, and other strategies; continued work with the ResNet-18 might be even more fruitful. In working with both KNN and CNN here, an image resolution of 128x128 RGB was used as a compromise. Increasing image resolution to 224x224 should immediately improve accuracy of the ResNet18 model. Furthermore, image augmentation such as rotation, scaling, coloration, and blurring, can also be performed to the further benefit of accuracy.

From a practical perspective, the true importance to farmers of differentiation between bean rust and angular leaf spot should be established. Since both are fungal diseases with similar pathologies and treatments, there may be little benefit in predicting both diseases separately. If models were re-formulated to binary classification of 'healthy' and 'diseased', accuracy is likely to increase. Thinking along these lines, however, also highlights the complete lack of data concerning other leaf conditions that might interfere with the classification discussed here. Can insect pests or nutrient deficiencies result in positive predictions for fungal disease? At this point, the answer to such questions are unknown.

In finale, the image classification models here show promise as a field tool for the diagnosis of bean leaf lesions. Nonetheless, it would be prudent to not rely on such tools as the only method of detection. Rather, they should be developed as part of a multi-modality approach to disease diagnosis.

5.References

- Celetti, Michael J., Melody S. Melzer, and Greg J. Boland. (2005) "Integrated management of angular leaf spot (*Phaeoisariopsis griseola* (Sacc.) Ferr.) on snap beans in Ontario." Plant health progress 6.1:2
- De Jesus, W. C., et al. (2001) "Effects of angular leaf spot and rust on yield loss of *Phaseolus vulgaris*." Phytopathology 91.11: 1045-1053.
- Food and Agriculture Organization of the United Nations (2025) Production: Crops and livestock products.
- Irizarry, Rafael A. (2019) Introduction to data science: Data analysis and prediction algorithms with R. Chapman and Hall/CRC
- Irizarry, Rafael A. (2024) 'Case study: MNIST', PH125.8x: Machine Learning. HarvardX
- Keydana, Sigrid. (2023) Deep learning and scientific computing with R torch. Chapman and Hall/CRC