September 12, 2020

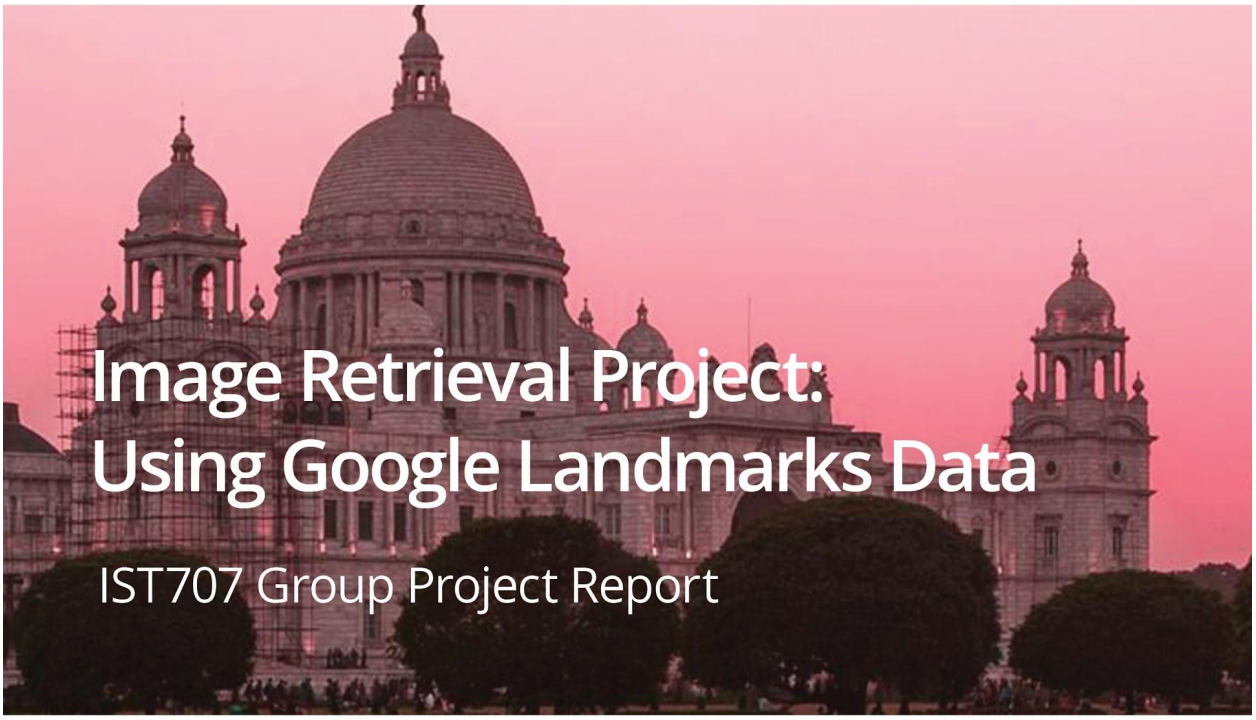


Image Retrieval Project: Using Google Landmarks Data

IST707 Group Project Report

Professor

Jeremy Bolton

Project Team

Daphne Chang
Sathish Kumar Rajendiran
Sharat Sripada

IST707 Data Analytics

School of Information Studies
Syracuse University

Introduction

The General

The project team has chosen the data set that comes from the interesting Kaggle competition “Image retrieval using Google Landmarks data”. The objective was to predict the images correctly by labels, using deep learning techniques (convolutional neural networks) and to compare the results with traditional models, such as Decision tree, k-Nearest Neighbors, Support Vector Machines and Random Forest. In addition, the team investigated the configuration of the architecture of convolutional neural networks (CNNs) and explored hyperparameter-tuning for model-fitting to determine whether CNNs are better than traditional classification algorithms for an image classification task.

Business Problem

Landmark search is important for a search engine company as landmarks account for a large portion of what people like to photograph. To improve landmark image retrieval, Google posted a competition at Kaggle and stated the business problem: Given an image, can you find all of the same landmarks in a dataset?

Essentially, this is an image classification problem. Using a sample set from the Google Landmark image data set provided by Kaggle, the project team attempted to address this business problem using TensorFlow neural network modeling and traditional data mining classification algorithms and compared their accuracy. Due to the tight time frame, the project team selected a small portion of the given dataset to create a proof of concept.

Analysis and Models

About the Data

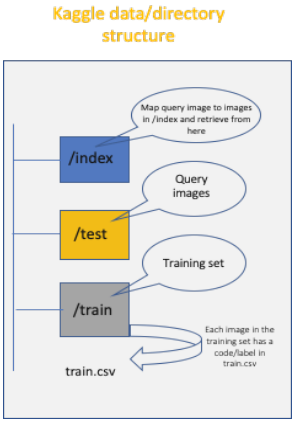


Fig: Kaggle directory structure

The Kaggle data is organized as shown above and comprises /index, /test and /train folders and a train.csv, that maps images under /train to corresponding landmark ids. train.csv would prove to be extremely useful in sampling datasets and splitting train data into validation or test. The train data was ~100GB comprising 1.5 million images and 81314 unique landmarks.

Example of landmark diversity below:

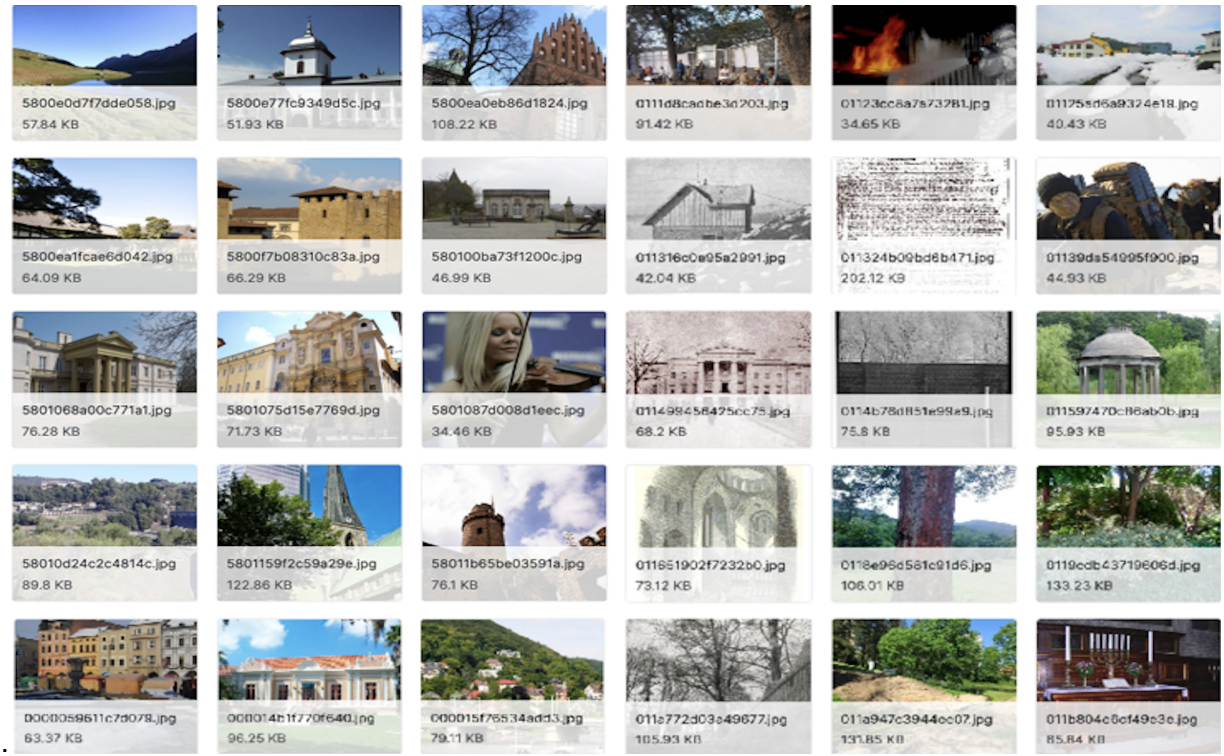


Fig: Examples from the original data set

Strategy

Dealing with a complex dataset involving image processing tasks to solve landmark retrieval problems in a short span of time needed a clear-cut strategy and execution plan. A high-level pipeline is presented here - read left to right, depth-first (stage-wise).

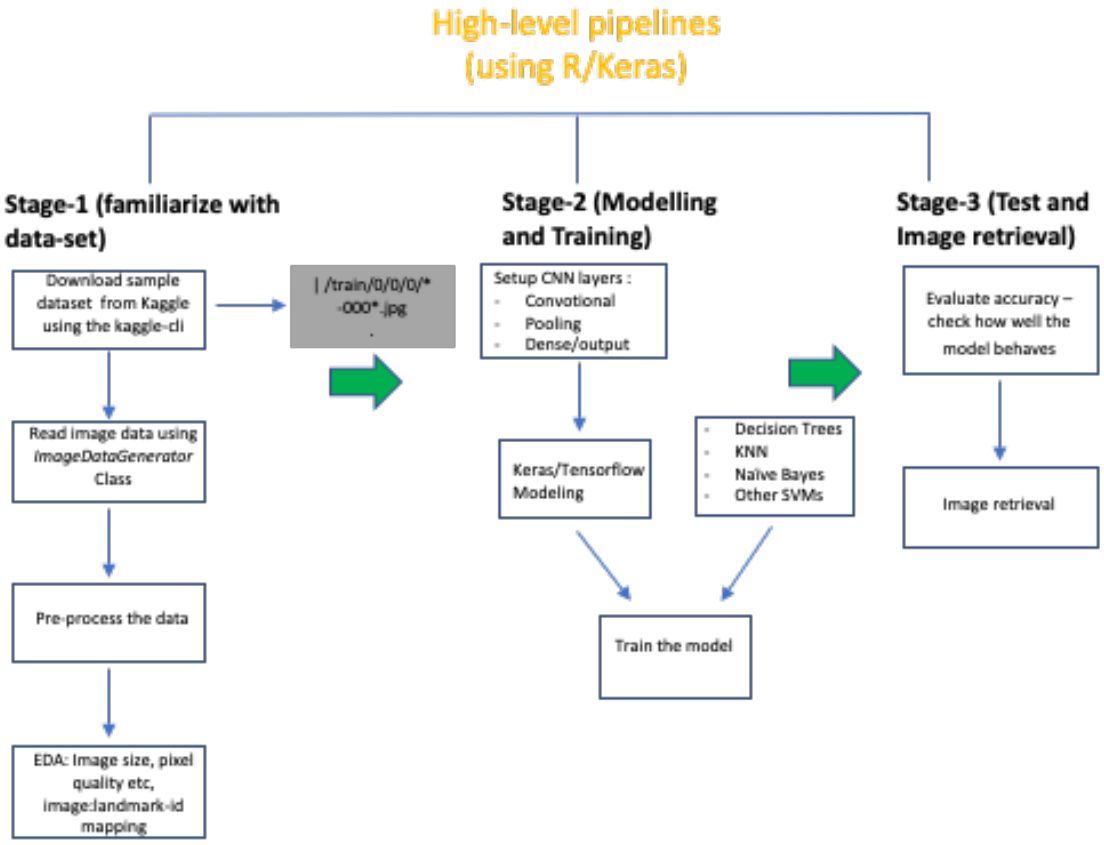


Fig: Work process pipelines

Also, some key decision points were:

- To adopt Keras, a Tensor-flow based library in R will be used for all image processing and analysis
- Leverage shell wrappers, Kaggle-cli etc. for data-retrieval and data split across train, test and validation directories
- Divide and conquer approach - Setup small proof-of-concept experiments and review outcomes.
- The analysis and results would broadly be based on the following experiments:
    - **Experiment-1** - Deep Learning: Multiclass Classifiers (large dataset)
    - **Experiment-2 -** Deep Learning: Binary/Multiclass Classifiers (reduced dataset)
    - **Experiment-3** - Traditional Models: Binary/Multiclass Classifiers (reduced dataset)

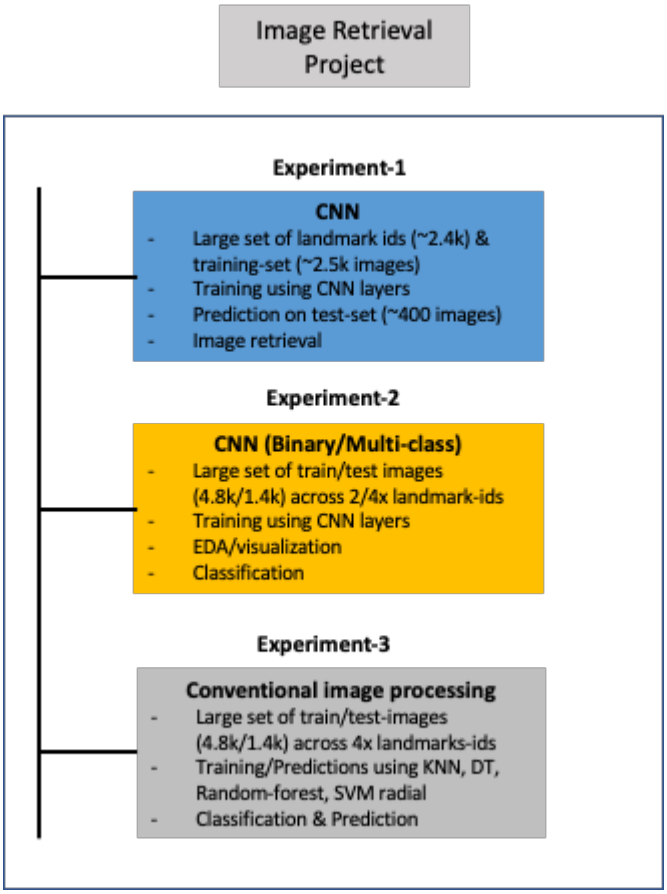


Fig: High-level project execution strategy

Deep Learning: Multiclass Classifiers (large dataset)

The first experiment attempts to solve the retrieval problem albeit, at lower scale. The Kaggle train data was split as follows for this experiment:

- Train - 2,000 landmark ids and 20,000 images*
- Test - 2,000 landmark ids and 10,000 images
- Validation - 2,000 landmark ids and 10,000 images

*Some experiments went as high as 40,000 images to solve problems around what seemed to arise from model overfitting

Images across datasets were mutually exclusive with train and validation used when training the model as below:

```
hist <- model_class %>% fit_generator (
  # training data
  train_images_class,

  # epochs
  steps_per_epoch = as.integer(train_samples_class / batch_size),
  epochs = epochs,

  # validation data
  validation_data = validation_images_class,
  validation_steps = as.integer(valid_samples_class / batch_size),

  # print progress
  verbose = 2
)
```

Fig: Code excerpt showing train and validation data in model training

And, the test dataset used to evaluate model accuracy.

Data Pre-processing

The initial data or image pre-processing involved study of the quality of images, pixels, dimensions etc. before it could be fed to the convolutional neural network.

Below is a sample image (in grayscale):

Fig: Sample landmark image in grayscale

This shows us, the pixel values are between 0-255 and the image is 600 x 800 dimension. We will correspondingly scale down the image as shown in the code excerpt:

```
train_data_gen = image_data_generator (rescale = 1/255)
# Image size scale-down to 100 x 100 images
img_width <- 100
img_height <- 100
target_size <- c (img_width, img_height)

train_images <- flow_images_from_directory(train_path,
                                           train_data_gen,
                                           target_size = target_size,
                                           class_mode = "categorical",
                                           classes = NULL,
                                           color_mode = "rgba",
                                           batch_size = 32)
```

Fig: Code excerpt showing image scale-down

The images are finally, in the following format when input to the neural net:

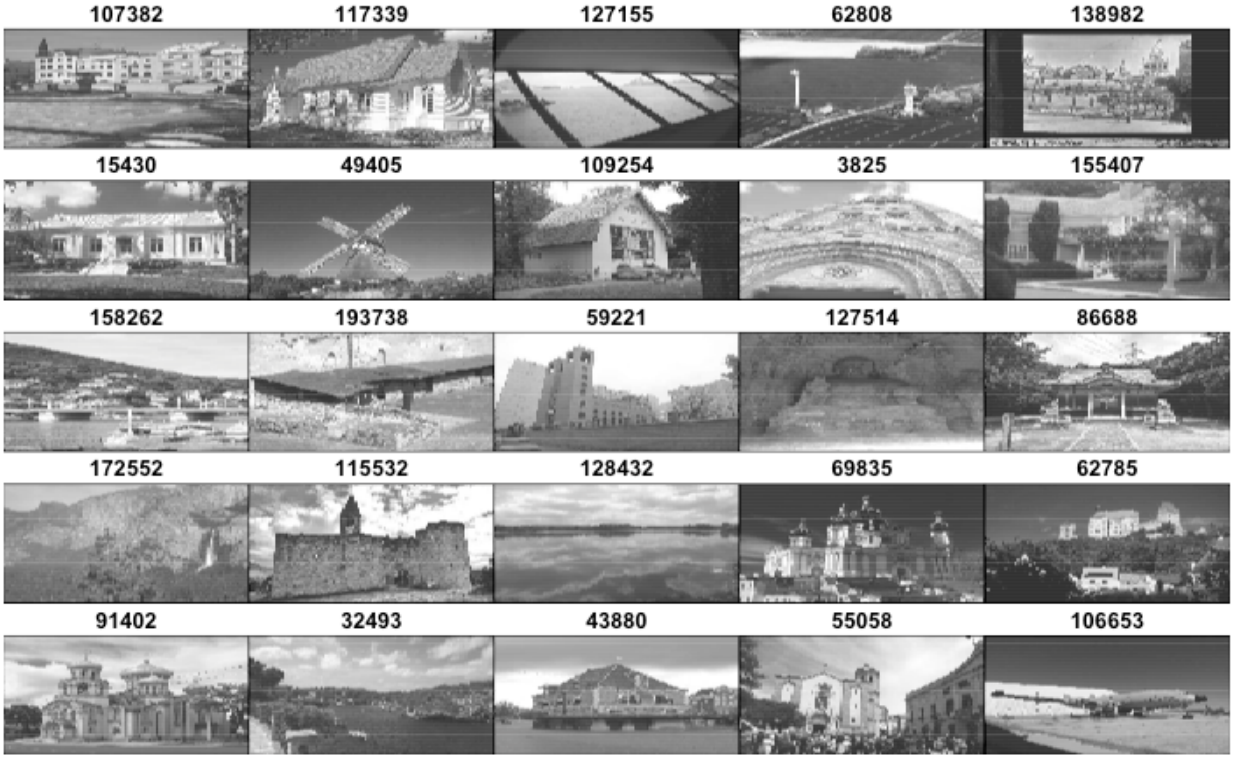


Fig: Sample images input to the neural net

About Convolutional Neural Network (CNN)

CNN is a type of deep-learning model, which dominates in computer vision applications and is widely used for image classification tasks. A CNN comprises of convolution layers, pooling layers, and fully connected layers. It adaptively learns spatial hierarchies of features through a backpropagation algorithm, using gradient descent. In this experiment, the Keras Library was used to build the CNN models with the TensorFlow backend. Keras provides high-level building blocks for building deep-learning models and enables rapid prototyping with minimal lines of code.

CNN (convnet) - comprises layers defined below:

- Convolutional layers - Extract feature-sets from images starting with a basic layer to retrieve details like edges to subsequent layers that can retrieve details from 2D window-like image portions
- Max pooling layer - To aggressively down-sample image feature-sets and uses the *MAX* tensor function to output maximum value from each channel
- Flattening layer - To convert 3D arrays from Convolution layers to 1D arrays
- Dense layer - The dense layer is the output layer, each node represented as a probability and therefore adding up to 1.

The initial convnet architecture and layers were setup as below:

```
Model: "sequential_1"

Layer (type)                 Output Shape              Param #
conv2d_3 (Conv2D)            (None, 100, 100, 32)      1184
activation_3 (Activation)    (None, 100, 100, 32)      0
max_pooling2d_2 (MaxPooling2D)  (None, 50, 50, 32)     0
conv2d_4 (Conv2D)            (None, 50, 50, 64)        18496
activation_4 (Activation)    (None, 50, 50, 64)        0
max_pooling2d_3 (MaxPooling2D)  (None, 25, 25, 64)     0
conv2d_5 (Conv2D)            (None, 25, 25, 128)       73856
activation_5 (Activation)    (None, 25, 25, 128)       0
flatten_1 (Flatten)          (None, 80000)             0
dense_2 (Dense)              (None, 512)               40960512
dense_3 (Dense)              (None, 2349)              1205037

Total params: 42,259,085
Trainable params: 42,259,085
Non-trainable params: 0
```

Fig: Summary of a compiled convnet model

Since the images were complex (often with landmarks camouflaged with other subjects, structures etc.) the convnet was performing poorly likely due to the noise, and the model lacking ability to generalize well enough. Some well-known techniques were sought to solve this problem along with a seemingly overfitting problem:

(1) Add dropout
(2) Reducing the network-size
(3) Increase the training-set and train longer (find the right balance of epochs and time to train so as not to overfit or generalize the model)

Model 1

The first model experiments around adding drop-out layers and experimenting with reducing the network-size. The drop-out functionality randomly drops out or sets to zero a number of output features during training. This is known to be most effective in neural networks.

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 80000)	0
dense (Dense)	(None, 512)	40960512
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 1999)	1025487

Total params: 42,079,535
Trainable params: 42,079,535
Non-trainable params: 0

Fig: Compiled convnet layer with drop-out

The drop-out option is present in Keras via *layer_dropout(<val>)* where val is usually set between 0.2-0.5. This experiment sets *layer_dropout = 0.5*

Further, the network is simplified by reducing the number of units in the dense layer:

```
# Use max pooling
layer_max_pooling_2d(pool_size = c(2,2)) %>%
layer_dropout(0.25) %>%

layer_conv_2d(filter = 128, kernel_size = c(3,3), padding = "same") %>%
layer_activation("relu") %>%


# Flatten max filtered output into feature vector
# and feed into dense layer
layer_flatten() %>%
layer_dense(units = <Number of units>, activation = "relu") %>%
layer_dropout(0.5) %>%
```

Fig: Code excerpt showing reduced units

Finally, to presumably solve an overfitting problem the experiment used larger training-set:
Found 28502 images belonging to 1999 classes. <- Train
Found 11453 images belonging to 1999 classes. <- Validation
Found 9979 images belonging to 1999 classes. <- Test

Model 2

The second model uses the notion of a pre-trained convolutional neural net. For this project, the Visual Geometry Group or commonly referred to as VGG16 was used. The architecture that comprises 16 layers can be visualized as:

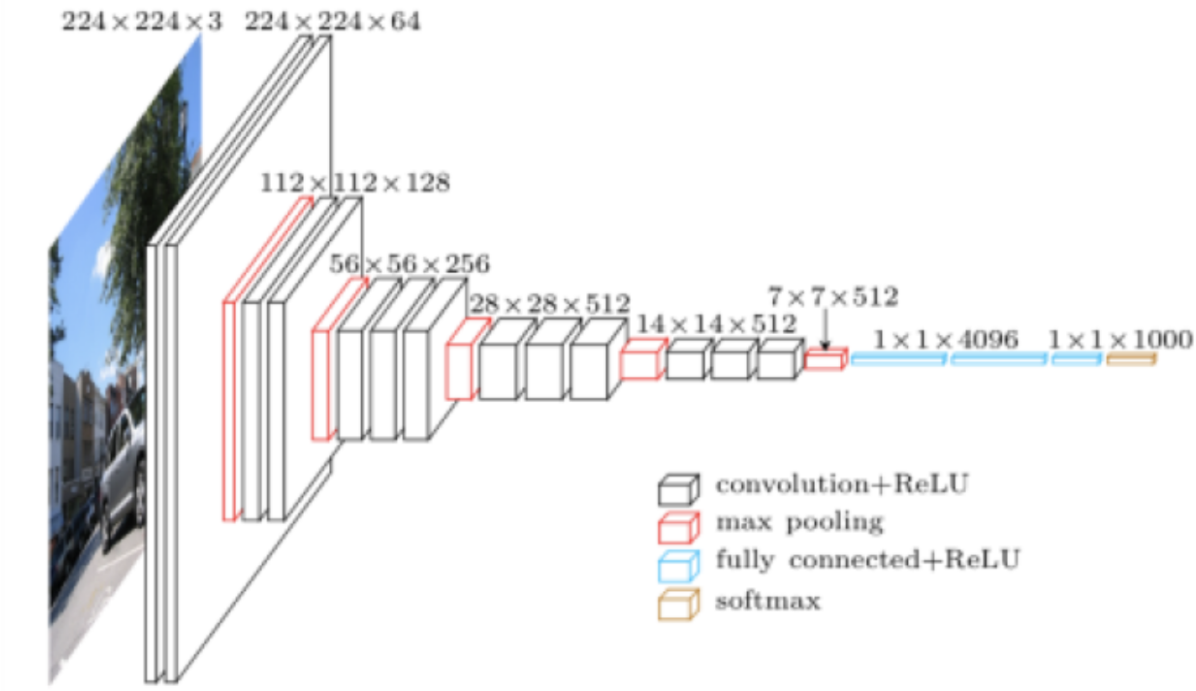


Fig: VGG16 convolutional neural network layers

The VGG model was modified to fit the number of classifiers in the dataset:

```
model_vgg <- application_vgg16(weights = "imagenet")

predictions <- model_vgg$output %>%
  layer_dense(units=<Number of output classifiers>)

model <- keras_model (inputs = model_vgg$input, outputs = predictions)
```

The results are presented in detail in the Results section.

Deep Learning: Binary/Multiclass Classifiers (reduced dataset)

To take a different approach from the first experiment, the second experiment started small — building a binary classifier using convolutional neural networks (CNN) — focusing on solving the classification problem for two labels (20409, 126637), which have the largest numbers of images among the four labels. After trial and error with hyperparameter-tuning, three models gave high accuracy results between 97.3% and 97.7%.

The experiment then expanded to include the whole sample set of four labels (20409, 83144, 113209, 126637). The same architecture for the binary classifier (except for the output layer unit and activation function) was also used to build the multiclass classifier:

- Binary Image Classifier — 2 labels, output layer with 1 unit and sigmoid function
- Multiclass Image Classifier — 4 labels, output layer with 4 units and softmax function.

The three models of the 4-labels classifier reached the accuracy between 89.8% and 93.1%.

The details of the experiment are explained below, starting from sample set selection, data pre-processing, and model analysis, to results summary.

Data Pre-processing

To select images for the sample set for the second experience, the training set from the original data set from Kaggle was inspected and explored. The train.csv file provided image IDs with the Landmark ID assigned to each image. A histogram was plotted to visualize the distribution of images by Landmark ID.

```
# Load train.csv
trainset <- read.csv("train.csv")
head(trainset)

# Count the number of images by image id
dim(trainset)

# all images in the original training set
hist(trainset$landmark_id, main = "Distribution of All Images by Landmark ID", xlab =
"Landmark ID")

# Count and Sort landmark IDs by frequency
landmarkid_freq <- trainset %>% group_by(trainset$landmark_id) %>% tally()
head(landmarkid_freq[rev(order(landmarkid_freq$n)),], 11)
```

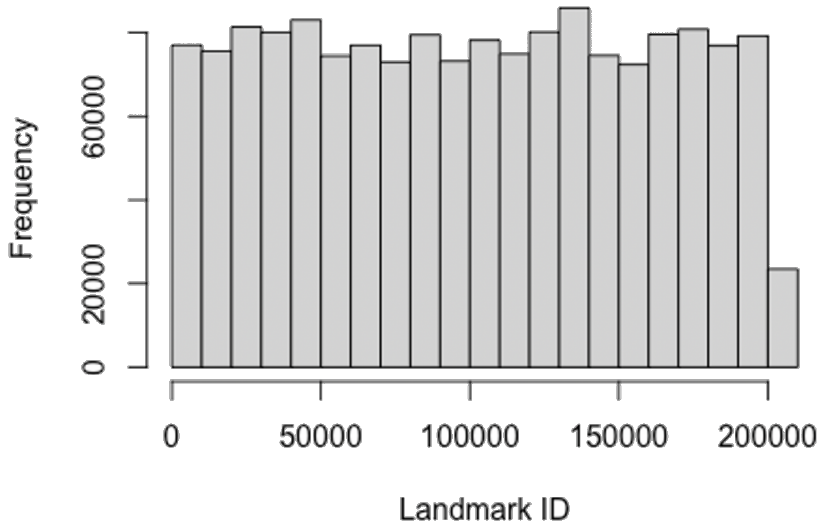


Fig: Distribution of All Images by Landmark ID

Through data exploration, four landmark IDs (labels) from the original Kaggle training data set were found to contain the highest numbers of homogeneous images within each label:

- 20409: 1758 images
- 83144: 1741 images
- 113209: 1135 images
- 126637: 2231 images

The total number of images in the four classes added up to 6865. This formed the sample set for the subsequent second and the third experiments. The images were of various sizes, and of various widths and heights.

```
## Inspect images labeled with each landmark ID in the sample data set
folder <- "data2/train/20409"
files <- Sys.glob(file.path(folder, "*.jpg"))
par(mfrow=c(3,3))
for(i in files[500:508]){
  im <- load.image(i)
  plot(im)
}

folder2 <- "data2/train/83144"
files <- Sys.glob(file.path(folder2, "*.jpg"))
par(mfrow=c(3,3))
for(i in files[700:708]){
  im <- load.image(i)
  plot(im)
}

folder3 <- "data2/train/113209"
files <- Sys.glob(file.path(folder3, "*.jpg"))
par(mfrow=c(3,3))
for(i in files[500:508]){
  im <- load.image(i)
  plot(im)
}

folder4 <- "data2/train/126637"
files <- Sys.glob(file.path(folder4, "*.jpg"))
par(mfrow=c(3,3))
for(i in files[500:508]){
  im <- load.image(i)
  plot(im)
}

dev.off()
```

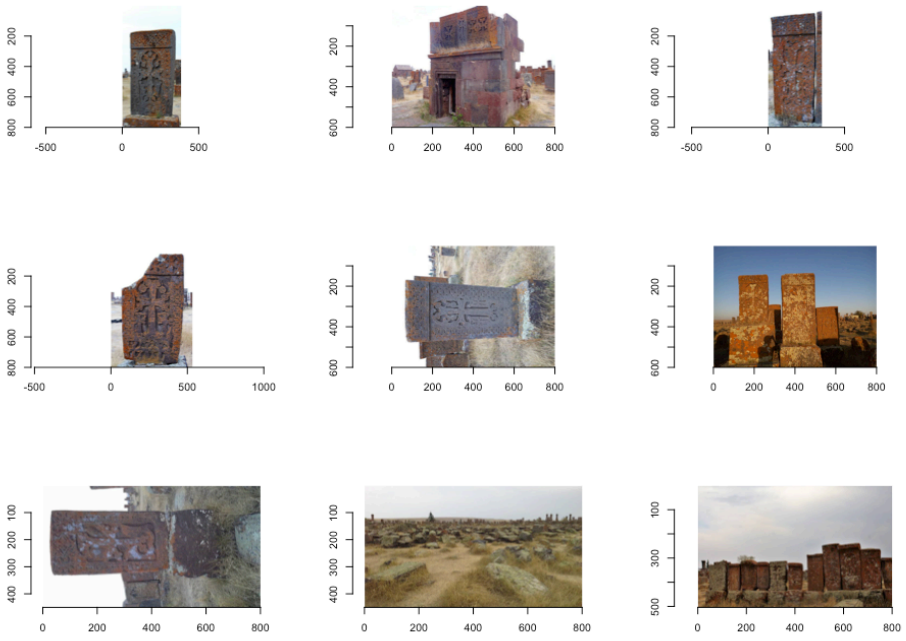


Fig: Sample Images with Landmark ID 20409

Fig: Sample Images with Landmark ID 83144

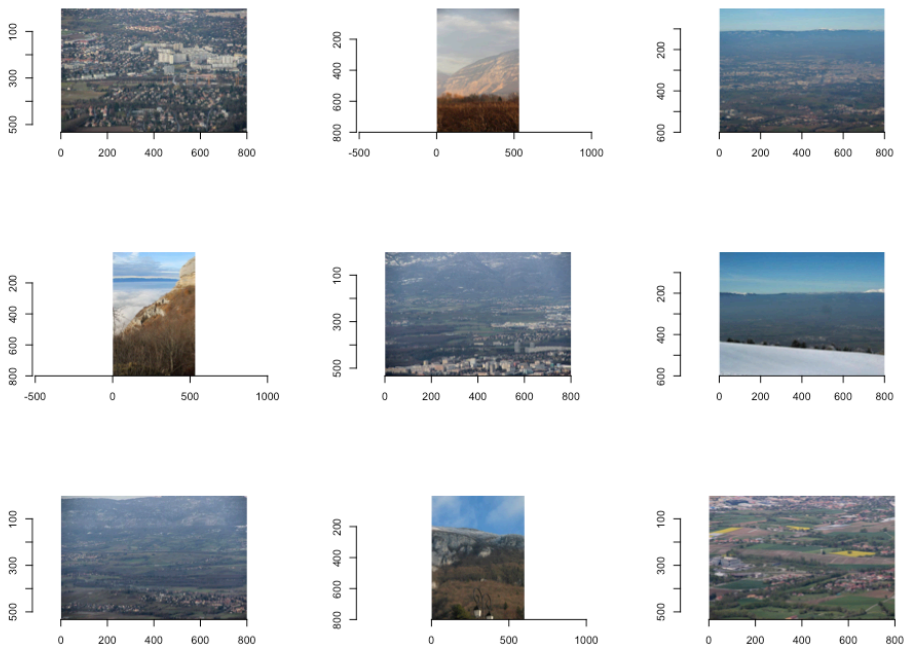


Fig: Sample Images with Landmark ID 113209

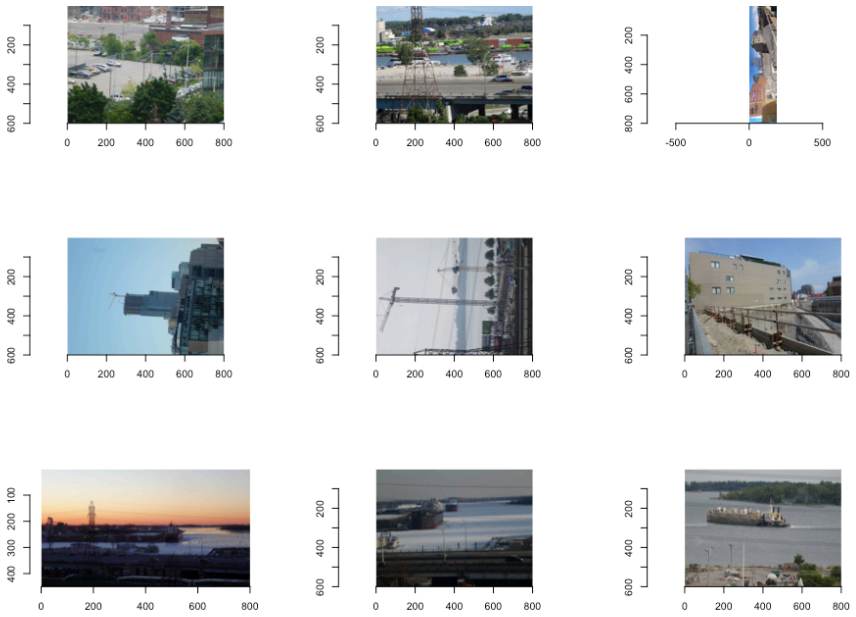


Fig: Sample Images with Landmark ID 126637

Binary Classifier

The images in classes 20409 and 126637 were used as the sample set for the binary classifier. For each class, images were split into the training set (50%), the validation set (25%), and the test set (25%). The same proportion of the split for each class ensured that the training set, validation set, and test set had the same distribution of images as compared with the whole sample data set.

- Training images in class 20409: 879
- Validation images in class 20409: 440
- Testing images in class 20409: 439
- Training images in class 126637: 1115
- Validation images in class 126637: 558
- Testing images in class 126637: 558

To enable the loading of the images from a folder into the network, the image files had to be saved into a folder structure with individual subfolders for training images, validation images, and testing images respectively; within each subfolder were two subfolders, each containing images for a class.

```
# Set up data directory structure for the binary classifier
# Comment out dir.create() after the directories are created
original_dataset_dir <- "~/Downloads/train"
base_dir <- "data"
#dir.create(base_dir)
train_dir <- file.path(base_dir, "train")
#dir.create(train_dir)
validation_dir <- file.path(base_dir, "validation")
#dir.create(validation_dir)
test_dir <- file.path(base_dir, "test")
#dir.create(test_dir)

train_20409_dir <- file.path(train_dir, "20409")
#dir.create(train_20409_dir)
train_126637_dir <- file.path(train_dir, "126637")
#dir.create(train_126637_dir)

validation_20409_dir <- file.path(validation_dir, "20409")
#dir.create(validation_20409_dir)
validation_126637_dir <- file.path(validation_dir, "126637")
#dir.create(validation_126637_dir)

test_20409_dir <- file.path(test_dir, "20409")
#dir.create(test_20409_dir)
test_126637_dir <- file.path(test_dir, "126637")
#dir.create(test_126637_dir)

# Load image indexes
df_20409 <- read.csv("train_20409.csv")
head(df_20409)
df_126637 <- read.csv("train_126637.csv")
head(df_126637)

# For each class, split data to training set 50%, testing set 25%, validation set
25%
set.seed(100)
percent <- .50

# Class 20409
split_20409 <- sample(nrow(df_20409),nrow(df_20409)*percent)
trainImg_20409 <- df_20409[split_20409,]
other_20409 <- df_20409[-split_20409,]
split2_20409 <- sample(nrow(other_20409),nrow(other_20409)*percent)
testImg_20409 <- other_20409[split2_20409,]
valImg_20409 <- other_20409[-split2_20409,]
dim(trainImg_20409)
dim(testImg_20409)
dim(valImg_20409)

# Class 126637
split_126637 <- sample(nrow(df_126637),nrow(df_126637)*percent)
trainImg_126637 <- df_126637[split_126637,]
other_126637 <- df_126637[-split_126637,]
split2_126637 <- sample(nrow(other_126637),nrow(other_126637)*percent)
testImg_126637 <- other_126637[split2_126637,]
valImg_126637 <- other_126637[-split2_126637,]
dim(trainImg_126637)
dim(testImg_126637)
dim(valImg_126637)
```

```
# Locate 20409 train images
loc_train20409_1 <- substr(trainImg_20409$id, 1, 1)
loc_train20409_2 <- substr(trainImg_20409$id, 2, 2)
loc_train20409_3 <- substr(trainImg_20409$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_train20409_1,"/",loc_train20409_2,"/",loc_trai
n20409_3)
# Copy 20409 images to train folder (comment out file.copy() after images are
copied into the directory)
trainf20409 <- paste0(trainImg_20409$id, ".jpg")
#file.copy(file.path(original_dataset_path, trainf20409),
file.path(train_20409_dir))
# Locate 20409 test images
loc_test20409_1 <- substr(testImg_20409$id, 1, 1)
loc_test20409_2 <- substr(testImg_20409$id, 2, 2)
loc_test20409_3 <- substr(testImg_20409$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_test20409_1,"/",loc_test20409_2,"/",loc_test20
409_3)
# Copy 20409 images to test folder
# Comment out file.copy() after images are copied into the directory
testf20409 <- paste0(testImg_20409$id, ".jpg")
#file.copy(file.path(original_dataset_path, testf20409),
file.path(test_20409_dir))
# Locate 20409 validation images
loc_val20409_1 <- substr(valImg_20409$id, 1, 1)
loc_val20409_2 <- substr(valImg_20409$id, 2, 2)
loc_val20409_3 <- substr(valImg_20409$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_val20409_1,"/",loc_val20409_2,"/",loc_val20409
_3)
# Copy 20409 images to validation folder
# Comment out file.copy() after images are copied into the directory
valf20409 <- paste0(valImg_20409$id, ".jpg")
#file.copy(file.path(original_dataset_path, valf20409),
file.path(validation_20409_dir))
```

```
# Locate 126637 train images
loc_train126637_1 <- substr(trainImg_126637$id, 1, 1)
loc_train126637_2 <- substr(trainImg_126637$id, 2, 2)
loc_train126637_3 <- substr(trainImg_126637$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_train126637_1,"/",loc_train126637_2,"/",loc_tr
ain126637_3)
# Copy 126637 images to train folder
# Comment out file.copy() after images are copied into the directory
trainf126637 <- paste0(trainImg_126637$id, ".jpg")
#file.copy(file.path(original_dataset_path, trainf126637),
file.path(train_126637_dir))
# Locate 126637 test images
loc_test126637_1 <- substr(testImg_126637$id, 1, 1)
loc_test126637_2 <- substr(testImg_126637$id, 2, 2)
loc_test126637_3 <- substr(testImg_126637$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_test126637_1,"/",loc_test126637_2,"/",loc_test
126637_3)
# Copy 126637 images to test folder
# Comment out file.copy() after images are copied into the directory
testf126637 <- paste0(testImg_126637$id, ".jpg")
#file.copy(file.path(original_dataset_path, testf126637),
file.path(test_126637_dir))
# Locate 126637 validation images
# Locate 126637 val images
loc_val126637_1 <- substr(valImg_126637$id, 1, 1)
loc_val126637_2 <- substr(valImg_126637$id, 2, 2)
loc_val126637_3 <- substr(valImg_126637$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_val126637_1,"/",loc_val126637_2,"/",loc_val126
637_3)
# Copy 126637 images to validation folder
# Comment out file.copy() after images are copied into the directory
valf126637 <- paste0(valImg_126637$id, ".jpg")
#file.copy(file.path(original_dataset_path, valf126637),
file.path(validation_126637_dir))
```

Read images from directories, convert them into batches of pre-processed floating-point tensors.

```
# Re-scale images by 1/255
train_datagen <- image_data_generator(rescale = 1/255)
validation_datagen <- image_data_generator(rescale = 1/255)
test_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory(
  train_dir,                    # Load images from the train folder
  train_datagen,                # Training data generator
  target_size = c(150, 150),    # Resize images to 150 × 150
  batch_size = 20,
  class_mode = "binary"
)

validation_generator <- flow_images_from_directory(
  validation_dir,
  validation_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)

test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary",
  seed = 100
)

batch <- generator_next(train_generator)
str(batch)
```

```
List of 2
     $ : num [1:20, 1:150, 1:150, 1:3] 0.137 0.353 0.31 0.745 1 ...
     $ : num [1:20(1d)] 0 0 0 0 1 1 1 0 1 1 ...
```

The training data generator output batches of 150 × 150 RGB images (shape (20, 150, 150, 3)) and binary labels (shape (20)), 20 images in each batch. The data were then ready to be fed into the network.

Binary Classifier: Model One: The Base Model

In the first CNN model, the network architecture contained a linear stack of 11 layers, including 4 alternated convolution layers (with relu activation) to augment the network capacity. Each was followed by a pooling layer to down sample feature maps to reduce feature-map coefficients processing and induce spatial-filter hierarchies. As the depth of the feature maps increased from 32 to 128, the size of the feature maps decreased from 148 × 148 to 7 × 7. After these 8 intermediate layers was a flattened layer and two dense layers (i.e. fully connected layers). For the binary classification problem, the output layer (the 2nd dense layer) had a single unit, with the sigmoid activation. The unit produced the probability of one class or the other.

```
tensorflow::tf$random$set_seed(100)

cnn <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(cnn)
```

```
Model: "sequential"

Layer (type)                 Output Shape              Param #
conv2d (Conv2D)              (None, 148, 148, 32)      896
max_pooling2d (MaxPooling2D) (None, 74, 74, 32)        0
conv2d_1 (Conv2D)            (None, 72, 72, 64)        18496
max_pooling2d_1 (MaxPooling2D) (None, 36, 36, 64)      0
conv2d_2 (Conv2D)            (None, 34, 34, 128)       73856
max_pooling2d_2 (MaxPooling2D) (None, 17, 17, 128)     0
conv2d_3 (Conv2D)            (None, 15, 15, 128)       147584
max_pooling2d_3 (MaxPooling2D) (None, 7, 7, 128)       0
flatten (Flatten)            (None, 6272)              0
dense (Dense)                (None, 512)               3211776
dense_1 (Dense)              (None, 1)                 513

Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

Compiling Model One

To configure the learning process, the RMSProp optimizer with the learning rate of 0.0001 was used in the compilation step and `binary_crossentropy` was used as the loss function. "Accuracy" was used as the metrics, which calculated how often predictions equal labels.

```
cnn %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("acc")
)
```

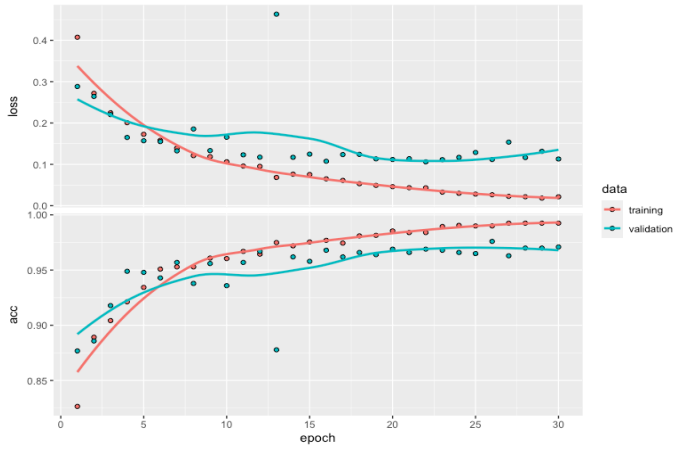
Training Model One

Using `fit_generator()`, the model was trained and validated through 30 epochs. An epoch is an iteration of passing all the training and validation data forward and backward through the neural network. The trained model was saved after training.

```
history <- cnn %>% fit_generator(
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,
  validation_data = validation_generator,
  validation_steps = 50
)

# Save the model
save_model_hdf5(cnn, "models/cnn.h5")
```

The plot showed the loss and accuracy of the model during training and validation through 30 epochs. It was clear from the plot that after 5 epochs the validation loss gradually pulls further above the training loss, and the training accuracy further above the validation accuracy. This was a sign of overfitting. It may have been caused by the small training sample data set (1994 images in 2 classes).

```
# Plot the loss and accuracy of the model during training
plot(history)
```




```
# Reload saved model
#cnn <- load_model_hdf5("models/cnn.h5")

# Evaluate the model
cnn %>% evaluate_generator(test_generator, steps = 50)
```

Even though the model was overfitted, the test accuracy still reached approximately 97.6%.

Binary Classifier: Model Two: Adding Data Augmentation & Dropout

Based on the same architecture, the second model added data augmentation and dropout to resolve the overfitting issue.

Adding Data Augmentation

Overfitting is caused by not having enough training samples from which the model can learn. Therefore, the model is not able to generalize to new data. One way to help resolve the overfitting problem is to use the Data Augmentation strategy to increase the diversity of the existing sample data during training. It does a number of random transformations on the sample images so, at training time, the model is exposed to more aspects of the data; thus the model can generalize better.

Common data augmentation techniques include:

1. Random rotation between 0-180 degrees
2. Random shifting of width and height of images
3. Random zooming inside images
4. Random horizontal flipping

image_data_generator() is used to random transformations are performed on the images. Below is an example of the resulted images after data augmentation techniques are performed.

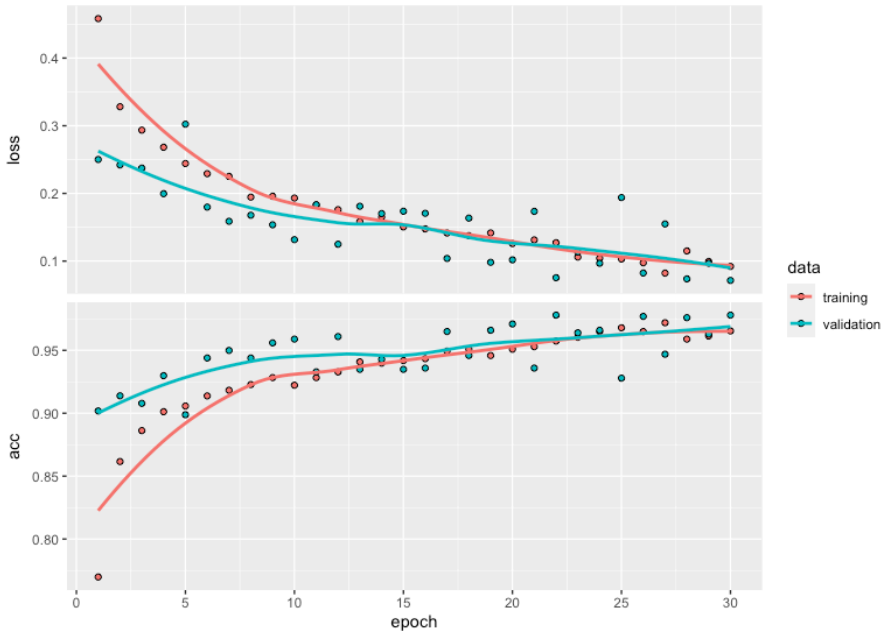
```r
# Configure random transformations
datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE,
  fill_mode = "nearest"
)

# Chooses an image to augment
fnames <- list.files(train_126637_dir, full.names = TRUE)
img_path <- fnames[[123]]

# Resize the image
img <- image_load(img_path, target_size = c(150, 150))

# Converts it to an array
img_array <- image_to_array(img)
img_array <- array_reshape(img_array, c(1, 150, 150, 3))

augmentation_generator <- flow_images_from_data(
  img_array,
  generator = datagen,
  batch_size = 1
)

# Plots the result
op <- par(mfrow = c(2, 2), pty = "s", mar = c(1, 0, 1, 0))
for (i in 1:4) {
  batch <- generator_next(augmentation_generator)
  plot(as.raster(batch[1,,,]))
}

par(op)
```

Fig: An Example of Data Augmentation

Adding Dropout

Data augmentation made the inputs more intercorrelated, because they included a large number of variations of the same images from the original small sample data set. Thus, it may not be enough to resolve the overfitting issue. In order to reduce intercorrelation, a dropout layer was added to the model after the flattened layer. "Dropout" is a regulation technique, which randomly selects neurons to ignore. The rate of the dropout was set as 50% in this model.

```
tensorflow::tf$random$set_seed(100)

cnn2 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

cnn2 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("acc")
)
datagen <- image_data_generator(  # For augmentation of the training data
  rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE
)
test_datagen <- image_data_generator(rescale = 1/255)  # Do not augment validation data
train_generator <- flow_images_from_directory(
  train_dir,
  datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)
validation_generator <- flow_images_from_directory(
  validation_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)
history2 <- cnn2 %>% fit_generator(
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,
  validation_data = validation_generator,
  validation_steps = 50
)
```




```
# Save the model
save_model_hdf5(cnn2, "models/cnn2.h5")

# Plot the loss and accuracy of the model during training
plot(history2)

test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary",
  seed = 100
)

# Evaluate the model
cnn2 %>% evaluate_generator(test_generator, steps = 50)
```

The use of data augmentation and dropout techniques have mostly resolved the overfitting issue. The loss and accuracy plot showed that both training loss and validation loss gradually went down and converged near epoch 30. The accuracy for both gradually increased and converged after epoch 24; however, the validation accuracy rose very slightly above the training accuracy at epoch 30. It could have been caused by the dropout rate (50%) being a bit high. Further testing could be done with lower dropout rates.

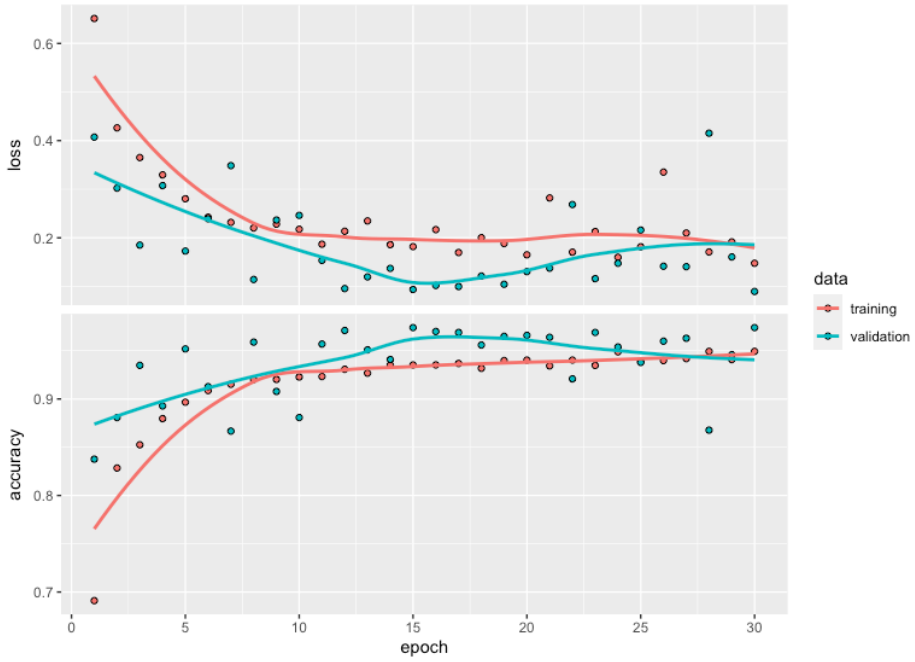
The test accuracy was high, approximately 97.7%, but it was only about 0.1% higher than the first model. It was a bit surprising that the overfitted model had similar high accuracy as the better-fitted model.

Binary Classifier: Model Three: Using RMSProp's Adaptive Learning Rate

The learning rate is a hyperparameter controlling how big the step is taken in gradient descent. In the first two models, RMSProp was used with a very low learning rate (0.0001) to be cautious, preventing overshooting in gradient descent. To test the impact of the low learning rate, in the third model the RMSProp optimizer was used without manually setting the learning rate. In Keras, the default learning rate for RMSProp is 0.001. As one of the adaptive learning rate methods, RMSProp maintains and adapts learning rates for each of the weights in the model.

```
tensorflow::tf$random$set_seed(100)

cnn3 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

cnn3 %>% compile(
  loss = "binary_crossentropy",
  #optimizer = optimizer_rmsprop(lr = 1e-3),
  optimizer = "rmsprop",
  metrics = c("accuracy")
)
datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE
)
test_datagen <- image_data_generator(rescale = 1/255)
train_generator <- flow_images_from_directory(
  train_dir,
  datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)
validation_generator <- flow_images_from_directory(
  validation_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)
```

```
history3 <- cnn3 %>% fit_generator(
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,
  validation_data = validation_generator,
  validation_steps = 50
)
```




```
# Save the model
save_model_hdf5(cnn3, "models/cnn3.h5")

# Plot the loss and accuracy of the model during training
plot(history3)

test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary",
  seed = 100
)

# Evaluate the model
cnn3 %>% evaluate_generator(test_generator, steps = 50)
```

Using the default RMSProp's adaptive learning rate (instead of the manually set low learning rate) in the third model lowered the test accuracy of the model from approximately 97.7% to 97.3%. It showed that manually tuning the learning rate hyperparameter could help improve accuracy, even though the improvement was very small in this case.

Multiclass Classifier

Data Pre-processing

The sample set included four labels (20409, 83144, 113209, 126637). The same architecture for the binary classifier (except for the 4 units for output layer and softmax activation function) was used to build the multiclass classifier.

For each class, images were split into the training set (50%), the validation set (25%), and the test set (25%). The same proportion of the split for each class ensured that the training set, validation set, and test set had the same distribution of images as compared with the whole sample data set.

- Training images in class 20409: 879
- Validation images in class 20409: 440
- Testing images in class 20409: 439

- Training images in class 83144: 870
- Validation images in class 83144: 436
- Testing images in class 83144: 435

- Training images in class 113209: 567
- Validation images in class 113209: 284
- Testing images in class 113209: 284

- Training images in class 126637: 1115
- Validation images in class 126637: 558
- Testing images in class 126637: 558

The image files were saved into a folder structure separated from the folder structure used for the binary classifier. The base folder included individual subfolders for training images, validation images, and testing images respectively; within each subfolder were four subfolders, each containing images for a class.

```
# Set up data directory structure
# Comment out dir.create() after the directories are created
original_dataset_dir <- "~/Downloads/train"
base_dir <- "data2"
#dir.create(base_dir)
train_dir <- file.path(base_dir, "train")
#dir.create(train_dir)
validation_dir <- file.path(base_dir, "validation")
#dir.create(validation_dir)
test_dir <- file.path(base_dir, "test")
#dir.create(test_dir)
```

```
train_20409_dir <- file.path(train_dir, "20409")
#dir.create(train_20409_dir)
train_126637_dir <- file.path(train_dir, "126637")
#dir.create(train_126637_dir)

validation_20409_dir <- file.path(validation_dir, "20409")
#dir.create(validation_20409_dir)
validation_126637_dir <- file.path(validation_dir, "126637")
#dir.create(validation_126637_dir)

test_20409_dir <- file.path(test_dir, "20409")
#dir.create(test_20409_dir)
test_126637_dir <- file.path(test_dir, "126637")
#dir.create(test_126637_dir)

train_83144_dir <- file.path(train_dir, "83144")
#dir.create(train_83144_dir)
train_113209_dir <- file.path(train_dir, "113209")
#dir.create(train_113209_dir)

validation_83144_dir <- file.path(validation_dir, "83144")
#dir.create(validation_83144_dir)
validation_113209_dir <- file.path(validation_dir, "113209")
#dir.create(validation_113209_dir)

test_83144_dir <- file.path(test_dir, "83144")
#dir.create(test_83144_dir)
test_113209_dir <- file.path(test_dir, "113209")
#dir.create(test_113209_dir)
```

```
# Load image indexes
df_20409 <- read.csv("train_20409.csv")
head(df_20409)
df_83144 <- read.csv("train_83144.csv")
head(df_83144)
df_113209 <- read.csv("train_113209.csv")
head(df_113209)
df_126637 <- read.csv("train_126637.csv")
head(df_126637)

# For each class, split data to training set 50%, testing set 25%, validation set 25%
set.seed(100)
percent <- .50

# Class 20409
split_20409 <- sample(nrow(df_20409),nrow(df_20409)*percent)
trainImg_20409 <- df_20409[split_20409,]
other_20409 <- df_20409[-split_20409,]
split2_20409 <- sample(nrow(other_20409),nrow(other_20409)*percent)
testImg_20409 <- other_20409[split2_20409,]
valImg_20409 <- other_20409[-split2_20409,]
dim(trainImg_20409)
dim(testImg_20409)
dim(valImg_20409)

# Class 83144
split_83144 <- sample(nrow(df_83144),nrow(df_83144)*percent)
trainImg_83144 <- df_83144[split_83144,]
other_83144 <- df_83144[-split_83144,]
split2_83144 <- sample(nrow(other_83144),nrow(other_83144)*percent)
testImg_83144 <- other_83144[split2_83144,]
valImg_83144 <- other_83144[-split2_83144,]
dim(trainImg_83144)
dim(testImg_83144)
dim(valImg_83144)
```

```
# Class 113209
split_113209 <- sample(nrow(df_113209),nrow(df_113209)*percent)
trainImg_113209 <- df_113209[split_113209,]
other_113209 <- df_113209[-split_113209,]
split2_113209 <- sample(nrow(other_113209),nrow(other_113209)*percent)
testImg_113209 <- other_113209[split2_113209,]
valImg_113209 <- other_113209[-split2_113209,]
dim(trainImg_113209)
dim(testImg_113209)
dim(valImg_113209)

# Class 126637
split_126637 <- sample(nrow(df_126637),nrow(df_126637)*percent)
trainImg_126637 <- df_126637[split_126637,]
other_126637 <- df_126637[-split_126637,]
split2_126637 <- sample(nrow(other_126637),nrow(other_126637)*percent)
testImg_126637 <- other_126637[split2_126637,]
valImg_126637 <- other_126637[-split2_126637,]
dim(trainImg_126637)
dim(testImg_126637)
dim(valImg_126637)
```

```
# Locate 20409 train images
loc_train20409_1 <- substr(trainImg_20409$id, 1, 1)
loc_train20409_2 <- substr(trainImg_20409$id, 2, 2)
loc_train20409_3 <- substr(trainImg_20409$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_train20409_1,"/",loc_train20409_2,"/",loc_train20409
_3)
# Copy 20409 images to train folder (comment out file.copy() after images are copied
into the directory)
trainf20409 <- paste0(trainImg_20409$id, ".jpg")
#file.copy(file.path(original_dataset_path, trainf20409), file.path(train_20409_dir))
# Locate 20409 test images
loc_test20409_1 <- substr(testImg_20409$id, 1, 1)
loc_test20409_2 <- substr(testImg_20409$id, 2, 2)
loc_test20409_3 <- substr(testImg_20409$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_test20409_1,"/",loc_test20409_2,"/",loc_test20409_3)
# Copy 20409 images to test folder
# Comment out file.copy() after images are copied into the directory
testf20409 <- paste0(testImg_20409$id, ".jpg")
#file.copy(file.path(original_dataset_path, testf20409), file.path(test_20409_dir))
# Locate 20409 validation images
loc_val20409_1 <- substr(valImg_20409$id, 1, 1)
loc_val20409_2 <- substr(valImg_20409$id, 2, 2)
loc_val20409_3 <- substr(valImg_20409$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_val20409_1,"/",loc_val20409_2,"/",loc_val20409_3)
# Copy 20409 images to validation folder
# Comment out file.copy() after images are copied into the directory
valf20409 <- paste0(valImg_20409$id, ".jpg")
#file.copy(file.path(original_dataset_path, valf20409), file.path(validation_20409_dir))
```

```
# Locate 83144 train images
loc_train83144_1 <- substr(trainImg_83144$id, 1, 1)
loc_train83144_2 <- substr(trainImg_83144$id, 2, 2)
loc_train83144_3 <- substr(trainImg_83144$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_train83144_1,"/",loc_train83144_2,"/",loc_train83144
_3)
# Copy 83144 images to train folder (comment out file.copy() after images are copied
into the directory)
trainf83144 <- paste0(trainImg_83144$id, ".jpg")
#file.copy(file.path(original_dataset_path, trainf83144), file.path(train_83144_dir))
# Locate 83144 test images
loc_test83144_1 <- substr(testImg_83144$id, 1, 1)
loc_test83144_2 <- substr(testImg_83144$id, 2, 2)
loc_test83144_3 <- substr(testImg_83144$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_test83144_1,"/",loc_test83144_2,"/",loc_test83144_3)
# Copy 83144 images to test folder
# Comment out file.copy() after images are copied into the directory
testf83144 <- paste0(testImg_83144$id, ".jpg")
#file.copy(file.path(original_dataset_path, testf83144), file.path(test_83144_dir))
# Locate 83144 validation images
loc_val83144_1 <- substr(valImg_83144$id, 1, 1)
loc_val83144_2 <- substr(valImg_83144$id, 2, 2)
loc_val83144_3 <- substr(valImg_83144$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_val83144_1,"/",loc_val83144_2,"/",loc_val83144_3)
# Copy 83144 images to validation folder
# Comment out file.copy() after images are copied into the directory
valf83144 <- paste0(valImg_83144$id, ".jpg")
#file.copy(file.path(original_dataset_path, valf83144), file.path(validation_83144_dir))

# Locate 113209 train images
loc_train113209_1 <- substr(trainImg_113209$id, 1, 1)
loc_train113209_2 <- substr(trainImg_113209$id, 2, 2)
loc_train113209_3 <- substr(trainImg_113209$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_train113209_1,"/",loc_train113209_2,"/",loc_train113
209_3)
# Copy 113209 images to train folder
# Comment out file.copy() after images are copied into the directory
trainf113209 <- paste0(trainImg_113209$id, ".jpg")
#file.copy(file.path(original_dataset_path, trainf113209), file.path(train_113209_dir))
# Locate 113209 test images
loc_test113209_1 <- substr(testImg_113209$id, 1, 1)
loc_test113209_2 <- substr(testImg_113209$id, 2, 2)
loc_test113209_3 <- substr(testImg_113209$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_test113209_1,"/",loc_test113209_2,"/",loc_test113209
_3)
# Copy 113209 images to test folder
# Comment out file.copy() after images are copied into the directory
testf113209 <- paste0(testImg_113209$id, ".jpg")
#file.copy(file.path(original_dataset_path, testf113209), file.path(test_113209_dir))
# Locate 113209 validation images
# Locate 113209 val images
loc_val113209_1 <- substr(valImg_113209$id, 1, 1)
loc_val113209_2 <- substr(valImg_113209$id, 2, 2)
loc_val113209_3 <- substr(valImg_113209$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_val113209_1,"/",loc_val113209_2,"/",loc_val113209_3)
# Copy 113209 images to validation folder
# Comment out file.copy() after images are copied into the directory
valf113209 <- paste0(valImg_113209$id, ".jpg")
#file.copy(file.path(original_dataset_path, valf113209),
file.path(validation_113209_dir))
```

```
# Locate 126637 train images
loc_train126637_1 <- substr(trainImg_126637$id, 1, 1)
loc_train126637_2 <- substr(trainImg_126637$id, 2, 2)
loc_train126637_3 <- substr(trainImg_126637$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_train126637_1,"/",loc_train126637_2,"/",loc_train126
637_3)
# Copy 126637 images to train folder
# Comment out file.copy() after images are copied into the directory
trainf126637 <- paste0(trainImg_126637$id, ".jpg")
#file.copy(file.path(original_dataset_path, trainf126637), file.path(train_126637_dir))
# Locate 126637 test images
loc_test126637_1 <- substr(testImg_126637$id, 1, 1)
loc_test126637_2 <- substr(testImg_126637$id, 2, 2)
loc_test126637_3 <- substr(testImg_126637$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_test126637_1,"/",loc_test126637_2,"/",loc_test126637
_3)
# Copy 126637 images to test folder
# Comment out file.copy() after images are copied into the directory
testf126637 <- paste0(testImg_126637$id, ".jpg")
#file.copy(file.path(original_dataset_path, testf126637), file.path(test_126637_dir))
# Locate 126637 validation images
# Locate 126637 val images
loc_val126637_1 <- substr(valImg_126637$id, 1, 1)
loc_val126637_2 <- substr(valImg_126637$id, 2, 2)
loc_val126637_3 <- substr(valImg_126637$id, 3, 3)
original_dataset_path <-
paste0(original_dataset_dir,"/",loc_val126637_1,"/",loc_val126637_2,"/",loc_val126637_3)
# Copy 126637 images to validation folder
# Comment out file.copy() after images are copied into the directory
valf126637 <- paste0(valImg_126637$id, ".jpg")
#file.copy(file.path(original_dataset_path, valf126637),
file.path(validation_126637_dir))
```

Read images from directories, convert them into batches of pre-processed floating-point tensors.

```
train_datagen <- image_data_generator(rescale = 1/255)
validation_datagen <- image_data_generator(rescale = 1/255)
test_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory(
  train_dir,
  train_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "categorical"
)

validation_generator <- flow_images_from_directory(
  validation_dir,
  validation_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "categorical"
)

test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "categorical",
  seed = 100
)
```

Multiclass Classifier: Model One: The Base Model

The same architecture for the binary classifier (except for the 4 units for output layer and softmax activation function) was used to build the multiclass classifier.

The model contained a linear stack of 11 layers, including 4 alternated convolution layers (with relu activation) to augment the network capacity, each followed by a pooling layer to down sample feature maps to reduce feature-map coefficients processing and induce spatial-filter hierarchies. After these 8 intermediate layers was a flattened layer and two dense layers (i.e. fully connected layers). For the 4-label classification problem, the output layer (the 2nd dense layer) had 4 single units, with the softmax activation. The Softmax regression is a form of multinomial logistic regression that normalizes an input value into a vector of 4 values that follows a probability distribution, where the total sums up to 1.

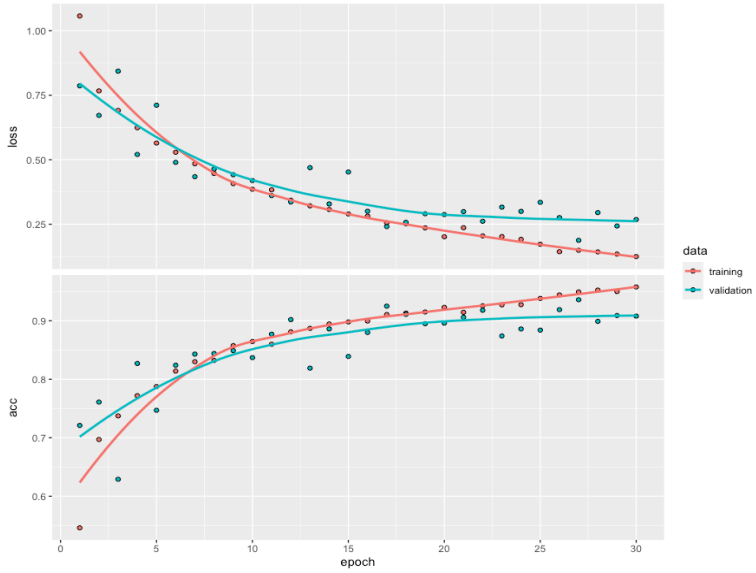
To configure the learning process, the RMSProp optimizer with the learning rate of 0.0001 was used in the compilation step and the `categorical_crossentropy` was used as the loss function. "Accuracy" was used as the metrics, which calculated how often predictions equal labels.

```
tensorflow::tf$random$set_seed(100)

cnn4 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 4, activation = "softmax")

summary(cnn4)
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_12 (MaxPooling2D)  (None, 74, 74, 32)    0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_13 (MaxPooling2D)  (None, 36, 36, 64)    0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_14 (MaxPooling2D)  (None, 17, 17, 128)   0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_15 (MaxPooling2D)  (None, 7, 7, 128)     0
_________________________________________________________________
flatten_3 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_6 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_7 (Dense)              (None, 4)                 2052
=================================================================
Total params: 3,454,660
Trainable params: 3,454,660
Non-trainable params: 0
_________________________________________________________________
```

```
cnn4 %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("acc")
)
```

```
history4 <- cnn4 %>% fit_generator(
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,
  validation_data = validation_generator,
  validation_steps = 50
)
```




```
# Save the model
save_model_hdf5(cnn4, "models/cnn4.h5")

# Plot the loss and accuracy of the model during training
plot(history4)

# Evaluate the model
cnn4 %>% evaluate_generator(test_generator, steps = 50)
```

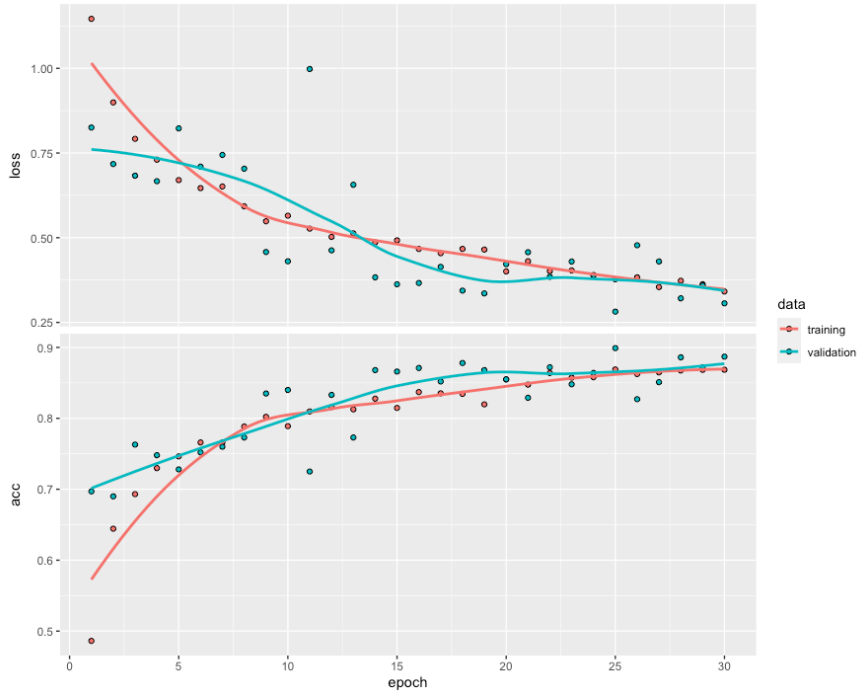
The same as the model one of the binary classifiers, the loss and accuracy plot for this model also showed the sign of overfitting. After 6 epochs, the validation loss gradually rose further above the training loss, and the training accuracy rose further above the validation accuracy. The overfitting problem may have been caused by the relatively small training sample data set (3431 images in 4 classes).

Even though the model was overfitted, the model still reached approximately 91% test accuracy.

Multiclass Classifier: Model Two: Adding Data Augmentation and Dropout

Based on the same architecture for model one, the second model added data augmentation and dropout to resolve the overfitting issue.

```
tensorflow::tf$random$set_seed(100)

cnn5 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 4, activation = "softmax")

cnn5 %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("acc")
)

test_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory(
  train_dir,
  datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "categorical"
)
validation_generator <- flow_images_from_directory(
  validation_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "categorical"
)
history5 <- cnn5 %>% fit_generator(
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,
  validation_data = validation_generator,
  validation_steps = 50
)
```




```
# Save the model
save_model_hdf5(cnn5, "models/cnn5.h5")

# Plot the loss and accuracy of the model during training
plot(history5)

test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "categorical",
  seed = 100
)

cnn5 %>% evaluate_generator(test_generator, steps = 50)
```

As with binary classification model two, the use of data augmentation and dropout techniques mostly resolved the overfitting issue. The loss and accuracy plot showed that both training loss and validation loss gradually went down and converged after epoch 25. The accuracy for both gradually increased and converged after epoch 25; however, the validation accuracy rose very slightly above the training accuracy at epoch 30. It could have been caused by the dropout rate (50%) being a bit high. Further testing could be done with lower dropout rates.

The evaluation of this model showed approximately 89.8% test accuracy, which is lower than the first model which had approximately 91% accuracy. It was a bit surprising that the overfitted model had better accuracy than the better-fitted model.

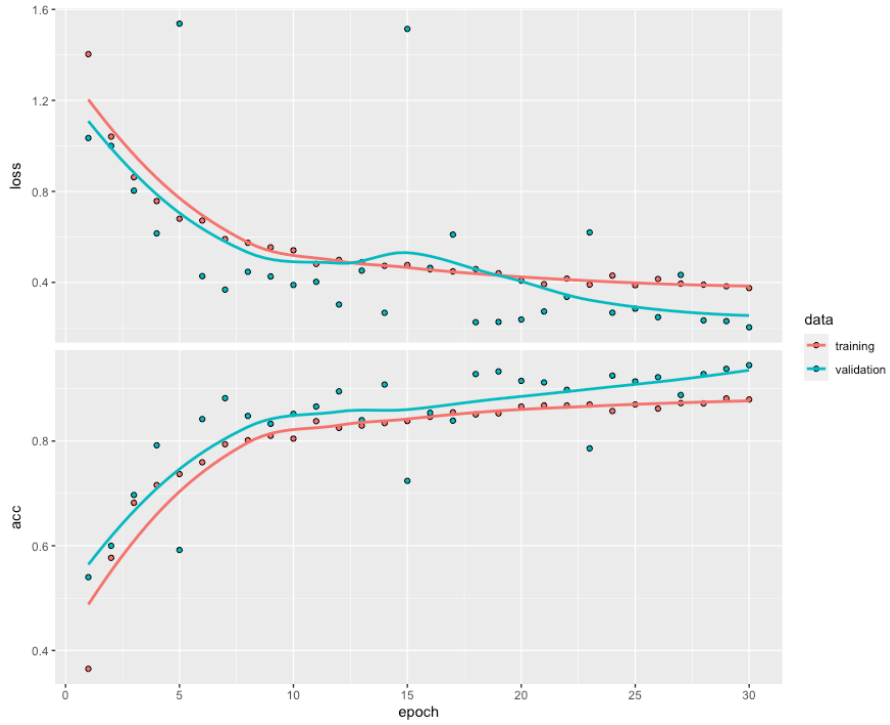
Multiclass Classifier: Model Three: Using RMSProp's Adaptive Learning Rate

With the binary classification models, the optimizer RMSProp was manually set with a very low learning rate (0.0001) for the first two models; removing the manual setting of the learning rate for the third model reduced its accuracy slightly.

The same testing of removing the manual setting of the learning rate was done for this multiclass model, to compare with the result of the binary classification model three.

```
tensorflow::tf$random$set_seed(100)

cnn6 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 4, activation = "softmax")

cnn6 %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  #optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("acc")
)
datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE
)

test_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory(
  train_dir,
  datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "categorical",
  seed = 100
)
validation_generator <- flow_images_from_directory(
  validation_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "categorical",
  seed = 100
)
```

```
history6 <- cnn6 %>% fit_generator(
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,
  validation_data = validation_generator,
  validation_steps = 50
)
```




```
# Save the model
save_model_hdf5(cnn6, "models/cnn6.h5")

# Plot the loss and accuracy of the model during training
plot(history6)

test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "categorical",
  seed = 100
)

cnn6 %>% evaluate_generator(test_generator, steps = 50)
```

Using the default RMSProp's adaptive learning rate (instead of the manually set low learning rate) in the third model improved the test accuracy of the model from approximately 88.2% to 93.1%. Unlike the binary classification model three, RMSProp's adaptive learning rate was more effective than manually tuning the learning rate hyperparameter in this case.

Even though the RMSProp optimizer seemed to work well without having manually configured the learning rate in this case, as the learning rate was an important hyperparameter, further tuning of this parameter could be tested to see if an optimal learning rate could be found to improve accuracy.

CNN: Visualizing Intermediate Activations

Activation maps can be created to visualize the activations on intermediate layers of a convolutional neural network. They are useful for understanding what the network sees and learns from the decomposition of the input on each of its intermediate layers.

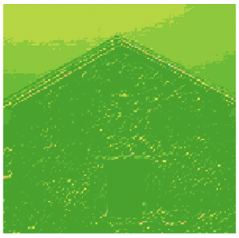
Below is an exploration of how the binary classification model two learns a new image it has not seen before.

First, load a new image for which the network has not been trained. Then, create a Keras model to take batches of images as input, and output the activations of all convolution and pooling layers.

```
# Load the cnn2 model
cnn2 <- load_model_hdf5("models/cnn2.h5")

dev.off()

# Get an input image, not part of the images the network was trained on
img_path <- "data/test/20409/19d05bdaba97701e.jpg"

#Preprocess the image into a 4D tensor
img <- image_load(img_path, target_size = c(150, 150))
img_tensor <- image_to_array(img)
img_tensor <- array_reshape(img_tensor, c(1, 150, 150, 3))
img_tensor <- img_tensor / 255
dim(img_tensor)

# Display the image
plot(as.raster(img_tensor[1,,,]))

# Extract the outputs of 8 imtermediate layers
layer_outputs <- lapply(cnn2$layers[1:8], function(layer) layer$output)

# Create a model that will return these outputs
activation_cnn2 <- keras_model(inputs = cnn2$input, outputs = layer_outputs)

# Run the model in predict mode
# Return a list of arrays: one array per layer activation
activations <- activation_cnn2 %>% predict(img_tensor)

# The activation of the first convolution layer
first_layer_activation <- activations[[1]]
dim(first_layer_activation)

# Define a function to plot a channel
plot_channel <- function(channel) {
  rotate <- function(x) t(apply(x, 2, rev))
  image(rotate(channel), axes = FALSE, asp = 1,
        col = terrain.colors(12))
}
```

Visualize the first channel of the activation of the first layer of the original model.

```
plot_channel(first_layer_activation[1,,,1])
```



Visualize the second channel of the activation of the first layer of the original model.

```
plot_channel(first_layer_activation[1,,,2])
```

Plot all the activations in the network by extracting and plotting every channel for each of the eight activation maps.

```
image_size <- 58
images_per_row <- 16

for (i in 1:8) {

  layer_activation <- activations[[i]]
  layer_name <- cnn2$layers[[i]]$name

  n_features <- dim(layer_activation)[[4]]
  n_cols <- n_features %/% images_per_row

  png(paste0("activations_", i, "_", layer_name, ".png"),
      width = image_size * images_per_row,
      height = image_size * n_cols)
  op <- par(mfrow = c(n_cols, images_per_row), mai = rep_len(0.02, 4))

  for (col in 0:(n_cols-1)) {
    for (row in 0:(images_per_row-1)) {
      channel_image <- layer_activation[1,,,(col*images_per_row) + row + 1]
      plot_channel(channel_image)
    }
  }

  par(op)
  dev.off()
}
```

Fig: Activation Maps on Intermediate Layers of Binary Classification Model Two

The first layer seems to focus on edge detection, retaining almost all the detailed information from the initial input. It is interesting to note that as the image flows through deeper layers, the activations become increasingly abstract. The details on the image become less and less visually interpretable; the pattern encoding becomes less about the visual content of the image itself and more about the information related to the class the image belongs. The sparsity of the activations also increases with the depth of the layer.

Activation maps give an inside look at how the network distills the input information and repeatedly transforms it to filter out visual details, which are irrelevant to the classification task, and magnifies useful information, which is the information about the class of the image. In this sense, it seems to work like a human brain, which learns to transform visual input into high-level visual concepts while filtering out irrelevant visual details.

Traditional Models: Binary/Multiclass Classifiers (reduced dataset)

As the third and final experiment, the goal was to classify and predict the images by labels using conventional classifiers such as Decision trees, k-Nearest Neighbors(kNN), Supply Vector Machines (SVM) and Random Forest. In addition, this experiment also uses a binary and multi-class classifier approach to solve and compare the accuracy among conventional vs deep learning models. In this section, 4 sample classes are used to predict the images.

- 20409 - Tombs
- 83144 - Huts
- 113209 - Mountains
- 126637 - Construction sites

Data Pre-processing

Used the same dataset as that of experiments with *Deep Learning: Binary/Multiclass classifiers (reduced dataset)* earlier; however, the images will be converted into 28*28 grayscale image pixels.

Function to convert images into 28*28 grayscale pixels below,

```
# image size to scale down to (original size varied)
width <- 28
height <- 28

## pbapply is a library to add progress bar *apply functions
## pblapply will replace lapply

extract_feature <- function(dir_path, width, height, label_id, add_label = TRUE) {
  img_size <- width*height
  ## List images in path
  images_names <- list.files(dir_path)
  if (add_label) {
    ## Set label
    label <- label_id
  }
  print(paste("Start processing", length(images_names), "images"))
  ## This function will resize an image, turn it into greyscale
  feature_list <- pblapply(images_names, function(imgname) {
    ## Read image
    img <- readImage(file.path(dir_path, imgname))
    ## Resize image
    img_resized <- resize(img, w = width, h = height)
    ## Set to grayscale
    grayimg <- channel(img_resized, "gray")
    ## Get the image as a matrix
    img_matrix <- grayimg@.Data
    ## Coerce to a vector
    img_vector <- as.vector(t(img_matrix))
    return(img_vector)
  })
  ## bind the list of vector into matrix
  feature_matrix <- do.call(rbind, feature_list)
  feature_matrix <- as.data.frame(feature_matrix)
  ## Set names
  names(feature_matrix) <- paste0("pixel", c(1:img_size))
  if (add_label) {
    ## Add label
    feature_matrix <- cbind(label = label, feature_matrix)
  }
  return(feature_matrix)
}
```

Images from four labels are converted into individual data frames and first column (label) is converted into a factor for classification.

```
## Task 2: Import myImages_20409 ##
image_20409_dir <- "/Users/sathishrajendiran/Documents/R/CNN/data/train/20409"
image_20409_dir
#use EBImage to resize the images to 28×28 and turn them into greyscale
train_data_20409 <- extract_feature(dir_path = image_20409_dir, width = width, height = height,label_id = 20409)
#Each image will be turned into a vector of length 784, with each element representing the value in a pixel
dim(train_data_20409) #[1] 879 785
```

```
[1] "/Users/sathishrajendiran/Documents/R/CNN/data/train/20409"
[1] "Start processing 879 images"
  |++++++++++++++++++++++++++++++++++++++++++++++++++| 100% elapsed=01m 15s
[1] 879 785
```

```
## Task 2: Import myImages_83144 ##
image_83144_dir <- "/Users/sathishrajendiran/Documents/R/CNN/data/train/83144"
image_83144_dir
#use EBImage to resize the images to 28×28 and turn them into greyscale
train_data_83144 <- extract_feature(dir_path = image_83144_dir, width = width, height = height,label_id = 83144)
#Each image will be turned into a vector of length 784, with each element representing the value in a pixel
dim(train_data_83144)
```

```
[1] "/Users/sathishrajendiran/Documents/R/CNN/data/train/83144"
[1] "Start processing 870 images"
  |++++++++++++++++++++++++++++++++++++++++++++++++++| 100% elapsed=01m 46s
[1] 870 785
```

```
## Task 2: Import myImages_113209 ##
image_113209_dir <- "/Users/sathishrajendiran/Documents/R/CNN/data/train/113209"
image_113209_dir
#use EBImage to resize the images to 28×28 and turn them into greyscale
train_data_113209 <- extract_feature(dir_path = image_113209_dir, width = width, height = height,label_id = 113209)
#Each image will be turned into a vector of length 784, with each element representing the value in a pixel
dim(train_data_113209)
```

```
[1] "/Users/sathishrajendiran/Documents/R/CNN/data/train/113209"
[1] "Start processing 567 images"
  |++++++++++++++++++++++++++++++++++++++++++++++++++| 100% elapsed=01m 09s
[1] 567 785
```

```
## Task 2: Import myImages_126637 ##
image_126637_dir <- "/Users/sathishrajendiran/Documents/R/CNN/data/train/126637"
image_126637_dir
#use EBImage to resize the images to 28×28 and turn them into greyscale
train_data_126637 <- extract_feature(dir_path = image_126637_dir, width = width, height = height,label_id = 126637)
#Each image will be turned into a vector of length 784, with each element representing the value in a pixel
dim(train_data_126637)
# head(train_data_126637)
```

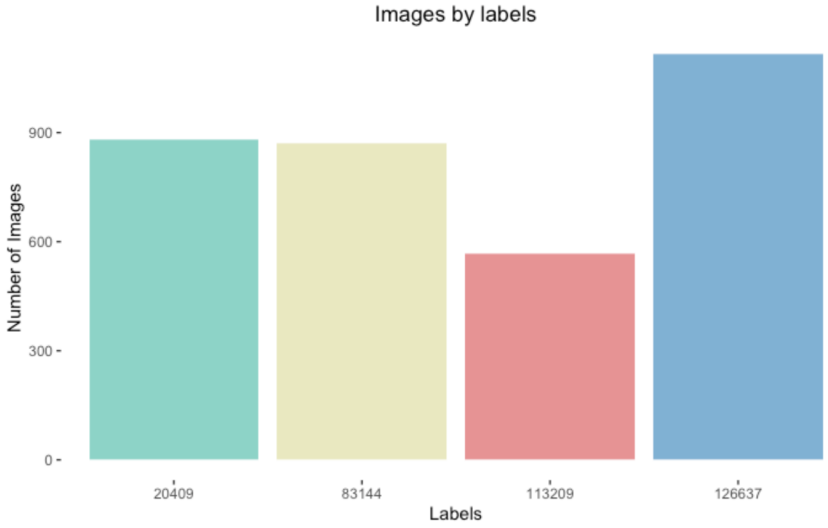
```
[1] "/Users/sathishrajendiran/Documents/R/CNN/data/train/126637"
[1] "Start processing 1115 images"
  |++++++++++++++++++++++++++++++++++++++++++++++++++| 100% elapsed=02m 00s
[1] 1115  785
```

Combined data frame is as below,

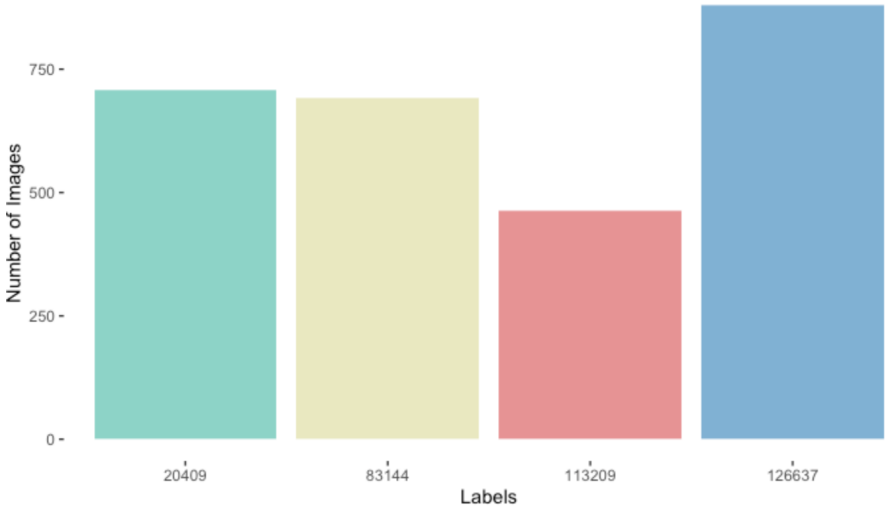
```
## Bind rows in a single dataset
trainImagesAllDF <- rbind(train_data_20409, train_data_83144,train_data_113209,train_data_126637)
dim(trainImagesAllDF) #[1] 3431  785

#convert label column to factor
trainImagesAllDF$label <- as.factor(trainImagesAllDF$label)
dim(trainImagesAllDF)

str(trainImagesAllDF)
```

```
'data.frame':   3431 obs. of  785 variables:
 $ label   : Factor w/ 4 levels "20409","83144",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ pixel1  : num  1 1 1 0.67 0.599 ...
 $ pixel2  : num  1 1 1 0.81 0.628 ...
 $ pixel3  : num  1 1 1 0.703 0.642 ...
 $ pixel4  : num  1 1 1 0.739 0.644 ...
 $ pixel5  : num  1 1 1 0.628 0.66 ...
 $ pixel6  : num  1 1 1 0.763 0.677 ...
 $ pixel7  : num  1 0.983 1 0.756 0.686 ...
 $ pixel8  : num  1 0.935 1 0.767 0.693 ...
 $ pixel9  : num  1 0.524 1 0.739 0.718 ...
 $ pixel10 : num  1 0.482 1 0.749 0.725 ...
 $ pixel11 : num  1 0.316 0.996 0.76 0.733 ...
 $ pixel12 : num  0.999 0.544 0.996 0.565 0.745 ...
 $ pixel13 : num  0.382 0.533 0.689 0.776 0.753 ...
 $ pixel14 : num  0.417 0.31 0.524 0.735 0.771 ...
 $ pixel15 : num  0.387 0.814 0.557 0.704 0.77 ...
 $ pixel16 : num  0.737 0.635 0.743 0.69 0.786 ...
 $ pixel17 : num  0.745 0.582 0.807 0.738 0.798 ...
 $ pixel18 : num  0.658 0.441 0.845 0.705 0.812 ...
 $ pixel19 : num  0.647 0.692 0.923 0.622 0.824 ...
 $ pixel20 : num  0.73 0.446 0.866 0.769 0.735 ...
 $ pixel21 : num  0.505 0.458 0.862 0.695 0.598 ...
 $ pixel22 : num  0.677 0.553 0.84 0.689 0.71 ...
 $ pixel23 : num  0.732 0.655 0.821 0.342 0.725 ...
 $ pixel24 : num  0.693 0.651 0.836 0.65 0.547 ...
 $ pixel25 : num  0.612 0.707 0.754 0.447 0.585 ...
 $ pixel26 : num  0.752 0.798 0.673 0.26 0.696 ...
 $ pixel27 : num  0.565 0.796 0.613 0.608 0.632 ...
 $ pixel28 : num  0.668 0.662 0.598 0.696 0.488 ...
 $ pixel29 : num  1 1 1 0.584 0.604 ...
```

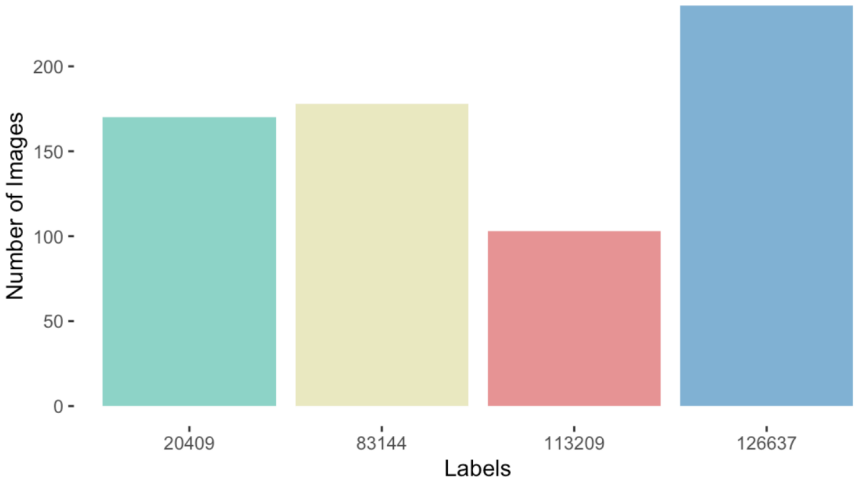
Sample data is further split into train and test data with 80:20 ratio as below,



Train data images by labels



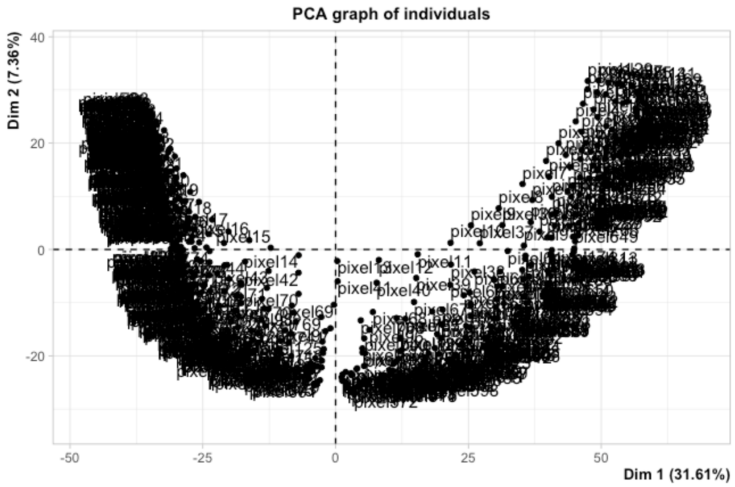
Test data images by labels

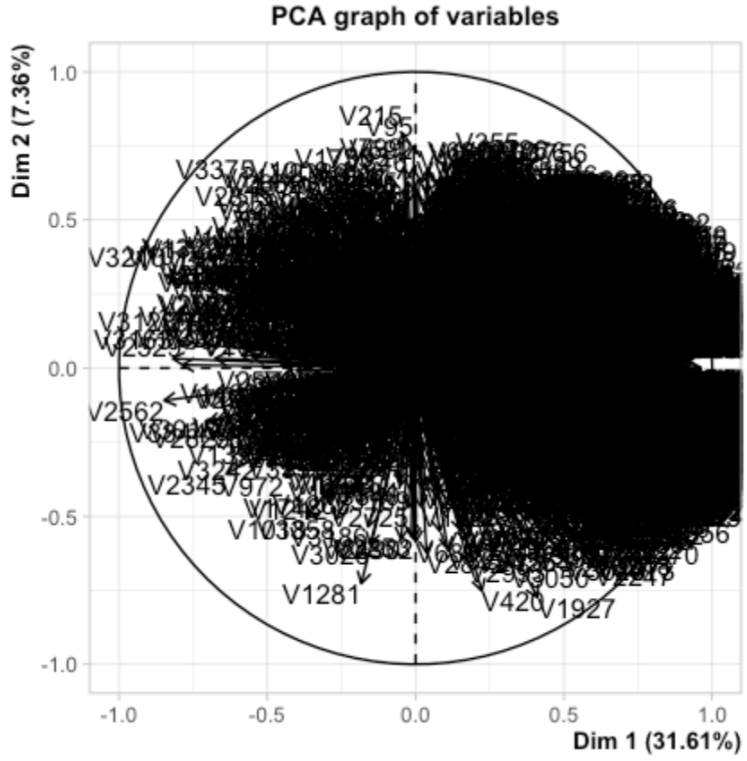


Data Preprocessing - Dimensionality reduction using Principal Component Analysis (PCA)

As the number of predictor variables are still huge (784) - dimensionality reduction method is applied to enhance the dataset. Principal Component Analysis (PCA) is used to reduce the number of dimensions to 30.

```r
#Principal Component Analysis
pca_digits <- PCA(t(select(trainImagesAllDF,-label)),ncp=30)
```





```
#Rename Columns
colnames(imagePCATrainDF) <- c("label","dim1","dim2","dim3","dim4","dim5","dim6","dim7","dim8","dim9","dim10","dim11","dim12","dim13"
                               ,"dim14","dim15","dim16","dim17","dim18","dim19","dim20","dim21","dim22","dim23","dim24","dim25"
                               ,"dim26","dim27","dim28","dim29","dim30")
str(imagePCATrainDF)
dim(imagePCATrainDF)
```

```
'data.frame':   3431 obs. of  31 variables:
 $ label: Factor w/ 4 levels "20409","83144",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ dim1 : num  0.498 0.405 0.439 -0.101 0.397 ...
 $ dim2 : num  0.5206 0.4451 0.4835 0.0515 -0.2893 ...
 $ dim3 : num  0.1175 0.0531 0.11 0.1251 0.0692 ...
 $ dim4 : num  -0.1703 -0.136 -0.3632 -0.0815 -0.111 ...
 $ dim5 : num  -0.1458 0.194 0.142 -0.1708 -0.0188 ...
 $ dim6 : num  0.00143 0.29715 0.03987 0.08689 -0.17395 ...
 $ dim7 : num  -0.00803 0.1158 0.01046 0.05978 0.05938 ...
 $ dim8 : num  -0.0786 0.1147 0.0587 -0.0334 0.1909 ...
 $ dim9 : num  0.0854 0.1102 0.0522 0.0485 0.2051 ...
 $ dim10: num  0.11772 -0.0291 -0.13328 -0.00389 -0.16107 ...
 $ dim11: num  0.04401 0.07651 0.11536 0.22196 -0.00746 ...
 $ dim12: num  0.1305 0.015 0.1299 -0.2429 -0.0477 ...
 $ dim13: num  -0.0374 0.1755 -0.074 -0.0136 -0.1614 ...
 $ dim14: num  0.1316 -0.0492 0.0588 0.0797 0.0578 ...
 $ dim15: num  -0.1597 -0.1268 0.1008 0.1131 0.0941 ...
 $ dim16: num  -0.1065 0.0538 -0.0588 -0.0163 -0.0296 ...
 $ dim17: num  -0.1192 -0.0693 -0.0661 0.0429 -0.0925 ...
 $ dim18: num  0.01149 -0.04418 0.00861 -0.02725 0.01518 ...
 $ dim19: num  0.08038 0.08427 0.04674 0.00787 0.00388 ...
 $ dim20: num  0.0233 -0.0492 -0.1205 0.1527 0.1339 ...
 $ dim21: num  0.057 -0.0136 -0.1807 0.0302 0.0396 ...
 $ dim22: num  0.07742 -0.00802 -0.06741 0.1737 -0.12538 ...
 $ dim23: num  -0.109027 0.007008 0.000556 -0.057942 -0.034231 ...
 $ dim24: num  -0.05323 0.00737 0.00135 0.01436 -0.05806 ...
 $ dim25: num  0.0815 -0.0887 0.0303 0.1225 -0.0028 ...
 $ dim26: num  -0.0851 0.0693 0.0642 0.0631 0.0718 ...
 $ dim27: num  -0.0214 -0.0216 0.0164 0.1089 -0.041 ...
 $ dim28: num  0.0478 -0.0038 -0.0992 -0.0125 0.0536 ...
 $ dim29: num  -0.04607 0.00187 -0.02084 -0.03893 -0.03676 ...
 $ dim30: num  0.04878 0.00316 0.0223 0.043 -0.06424 ...
[1] 3431   31
```

Let's prepare, train and test the dataset from the reduced dimension dataset as well.

```
# Prepare train and test data from the revised PCA Train dataset.
  sample_size = floor(0.80*nrow(imagePCATrainDF))
  # set seed to ensure you always have same random numbers generated #324 has 100% training accuracy
  train_index = sample(seq_len(nrow(imagePCATrainDF)),size = sample_size)
  # train_index
  pca_train_set =imagePCATrainDF[train_index,] #creates the training dataset with row numbers stored in train_index
  # pca_train_set
  # # table(train_data$author)
  pca_test_set=imagePCATrainDF[-train_index,]  # creates the test dataset excluding the row numbers mentioned in train_index
  # # table(test_data$author)
  cat("\nImages by Labels:")
  table(imagePCATrainDF$label)
  cat("\nTrain_set - Images by Labels:")
  table(pca_train_set$label)
  cat("\nTest_set - Images by Labels:")
  table(pca_test_set$label)
```

```
Images by Labels:
 20409  83144 113209 126637
   879    870    567   1115

Train_set - Images by Labels:
 20409  83144 113209 126637
   707    697    450    890

Test_set - Images by Labels:
 20409  83144 113209 126637
   172    173    117    225
```

Now that we have datasets from both PCA (30 predictor variables) and actual dataset (784 predictor variables) - it's time to build models based on both datasets.

Multiclass Classifiers

Model 1: Decision Trees (DT)

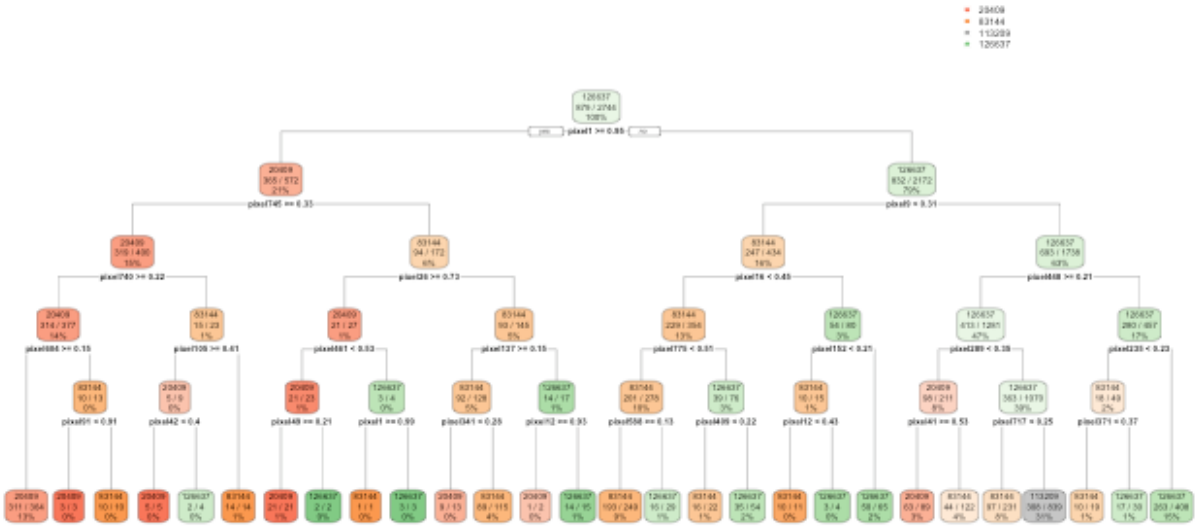
The DT model is built with control measures including cp, minsplit and maxdepth using the rpart library from R Studio. Initial case let's use the original data set split with 784 predictor variables and parameters as cp=0, minsplit=1 and maxdepth=5. This process involves, building the tree, plotting and summarizing it with cross validation results.

```
# Section 2: Build and tune decision tree models
    # grow tree
  rtree <- rpart(label~. ,data=train_set, method='class', cp=0,minsplit = 1, maxdepth = 5)
  #summarize rtree values
  summary(rtree)
  plotcp(rtree) # plot cross-validation results
  printcp(rtree) # plot cross-validation results
  # Plot tree | lets Plot decision trees
  rpart.plot(rtree,main="Image Classification Tree", extra= 102) # plot decision tree
  rsq.rpart(rtree) # plot approximate R-squared and relative error for different splits (2 plots)
```

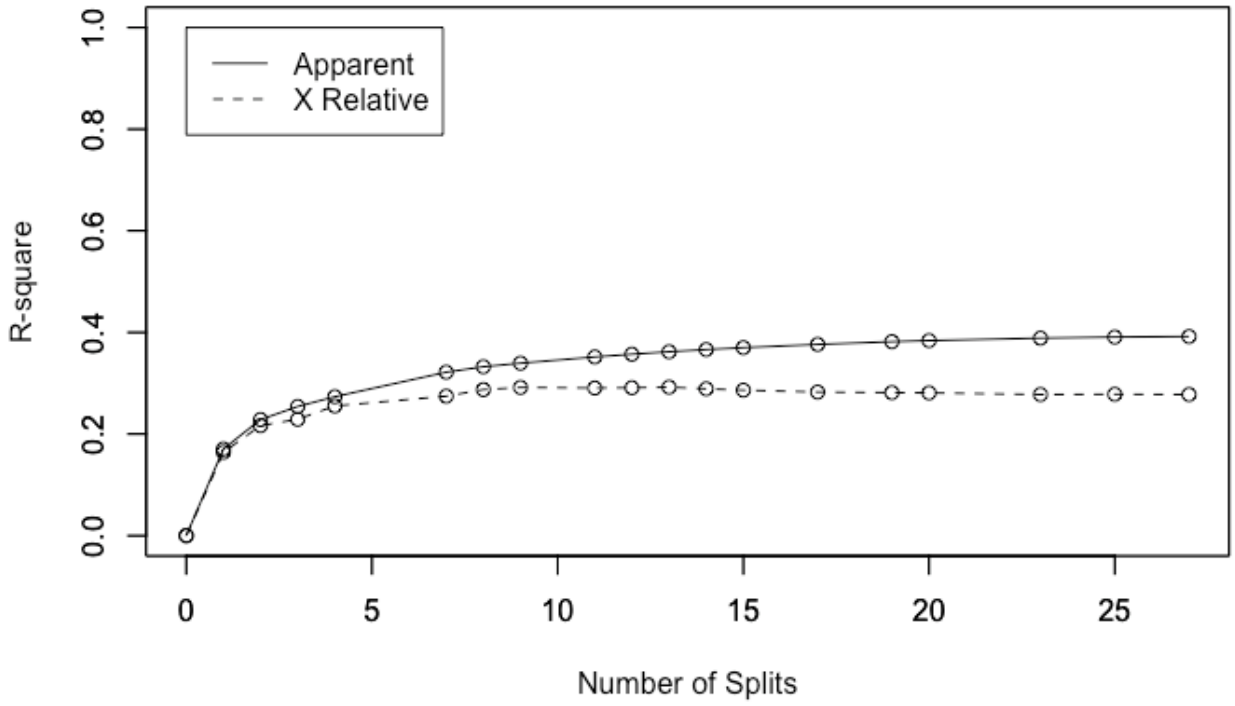
```
Call:
rpart(formula = label ~ ., data = train_set, method = "class",
    cp = 0, minsplit = 1, maxdepth = 5)
  n= 2744

            CP nsplit rel error    xerror       xstd
1  0.170509383      0 1.0000000 1.0000000 0.01310578
2  0.057908847      1 0.8294906 0.8359249 0.01391270
3  0.025737265      2 0.7715818 0.7833780 0.01401418
4  0.019302949      3 0.7458445 0.7710456 0.01402750
5  0.012868633      4 0.7265416 0.7447721 0.01404268
6  0.010723861      7 0.6782842 0.7254692 0.01404241
7  0.006970509      8 0.6675603 0.7126005 0.01403686
8  0.006166220      9 0.6605898 0.7083110 0.01403405
9  0.005361930     11 0.6482574 0.7093834 0.01403479
10 0.004825737     12 0.6428954 0.7088472 0.01403443
11 0.004289544     13 0.6380697 0.7077748 0.01403366
12 0.003753351     14 0.6337802 0.7109920 0.01403586
13 0.003217158     15 0.6300268 0.7136729 0.01403748
14 0.002680965     17 0.6235925 0.7174263 0.01403944
15 0.002144772     19 0.6182306 0.7184987 0.01403993
16 0.001608579     20 0.6160858 0.7190349 0.01404017
17 0.001072386     23 0.6112601 0.7222520 0.01404142
18 0.000536193     25 0.6091153 0.7217158 0.01404123
19 0.000000000     27 0.6080429 0.7222520 0.01404142
```

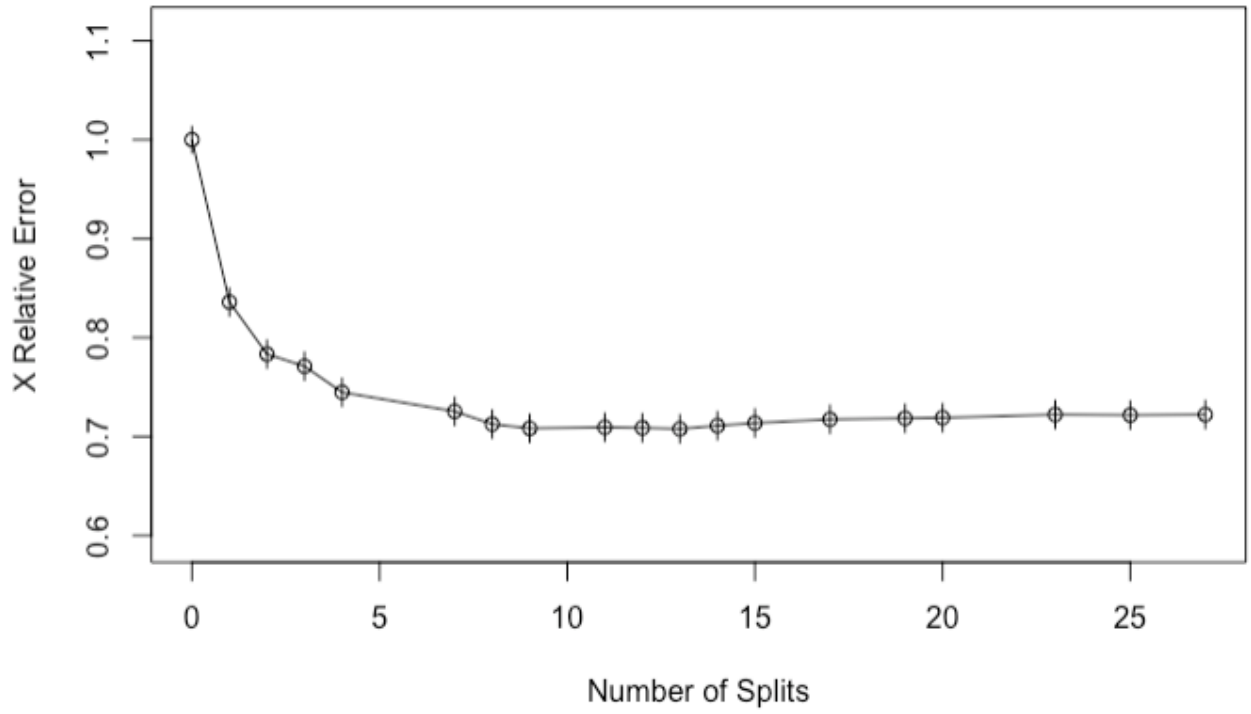
```
Variable importance
  pixel1   pixel2  pixel29  pixel30  pixel57   pixel3   pixel9 pixel745 pixel448 pixel716
       7        6        6        6        6        6        3        3        2        2
pixel744 pixel772 pixel773  pixel16 pixel420 pixel447 pixel475 pixel392  pixel26 pixel137
       1        1        1        1        1        1        1        1        1        1
  pixel8  pixel10  pixel37 pixel289 pixel717   pixel7 pixel746  pixel36 pixel476
       2        2        1        1        1        1        1        1        1
pixel740 pixel775  pixel25 pixel689  pixel41 pixel233  pixel42 pixel235 pixel588
       1        1        1        1        1        1        1        1        1
pixel718 pixel604
       1        1
```

Image Classification Tree



```
Node number 1: 2744 observations,    complexity param=0.1705094
  predicted class=126637  expected loss=0.6796647  P(node) =1
    class counts:   709   692   464   879
   probabilities: 0.258 0.252 0.169 0.320
  left son=2 (572 obs) right son=3 (2172 obs)
  Primary splits:
      pixel1  < 0.9529412  to the right, improve=158.8902, (0 missing)
      pixel2  < 0.9618647  to the right, improve=155.4911, (0 missing)
      pixel29 < 0.9453548  to the right, improve=149.6338, (0 missing)
      pixel3  < 0.9599865  to the right, improve=148.8699, (0 missing)
      pixel30 < 0.9592037  to the right, improve=146.1075, (0 missing)
  Surrogate splits:
      pixel2  < 0.9527544  to the right, agree=0.973, adj=0.869, (0 split)
      pixel29 < 0.9529412  to the right, agree=0.973, adj=0.869, (0 split)
      pixel30 < 0.9513255  to the right, agree=0.964, adj=0.825, (0 split)
      pixel57 < 0.9562842  to the right, agree=0.960, adj=0.808, (0 split)
      pixel3  < 0.9496499  to the right, agree=0.952, adj=0.767, (0 split)

Node number 2: 572 observations,    complexity param=0.02573727
  predicted class=20409   expected loss=0.3618881  P(node) =0.2084548
    class counts:   365   142    18    47
   probabilities: 0.638 0.248 0.031 0.082
  left son=4 (400 obs) right son=5 (172 obs)
  Primary splits:
      pixel745 < 0.3271325  to the right, improve=57.28967, (0 missing)
      pixel744 < 0.3146509  to the right, improve=51.59881, (0 missing)
      pixel772 < 0.3208717  to the right, improve=48.05360, (0 missing)
      pixel743 < 0.250396   to the right, improve=46.99264, (0 missing)
      pixel770 < 0.2848231  to the right, improve=45.65515, (0 missing)
  Surrogate splits:
      pixel744 < 0.29385    to the right, agree=0.836, adj=0.453, (0 split)
      pixel746 < 0.2801996  to the right, agree=0.832, adj=0.442, (0 split)
      pixel772 < 0.2932056  to the right, agree=0.832, adj=0.442, (0 split)
      pixel773 < 0.3313575  to the right, agree=0.830, adj=0.436, (0 split)
      pixel716 < 0.2970755  to the right, agree=0.825, adj=0.419, (0 split)
```


Apparent
X Relative
R-square
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25
Number of Splits


X Relative Error
0.6
0.7
0.8
0.9
1.0
1.1
0
5
10
15
20
25
Number of Splits

```r
# Section 3: Prediction | Test Phase
predict_unseen <- predict(rtree, test_set, type = 'class')
# predict_unseen
table_test_matrix <- table(test_set$label, predict_unseen)
cat("\n\nPrediction results : Confusion Matrix \n\n")
# table_mat
confusionMatrix(table_test_matrix)
```

```
Prediction results : Confusion Matrix

Confusion Matrix and Statistics

        predict_unseen
         20409 83144 113209 126637
  20409    105    22     36      7
  83144     14    97     32     35
  113209     5    11     70     17
  126637    15    58     86     77

Overall Statistics

               Accuracy : 0.508
                 95% CI : (0.4699, 0.546)
    No Information Rate : 0.3261
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.3545

 Mcnemar's Test P-Value : < 2.2e-16

Statistics by Class:

                     Class: 20409 Class: 83144 Class: 113209 Class: 126637
Sensitivity                0.7554       0.5160        0.3125         0.5662
Specificity                0.8814       0.8377        0.9287         0.7114
Pos Pred Value             0.6176       0.5449        0.6796         0.3263
Neg Pred Value             0.9342       0.8212        0.7363         0.8692
Prevalence                 0.2023       0.2737        0.3261         0.1980
Detection Rate             0.1528       0.1412        0.1019         0.1121
Detection Prevalence       0.2475       0.2591        0.1499         0.3435
Balanced Accuracy          0.8184       0.6768        0.6206         0.6388
```

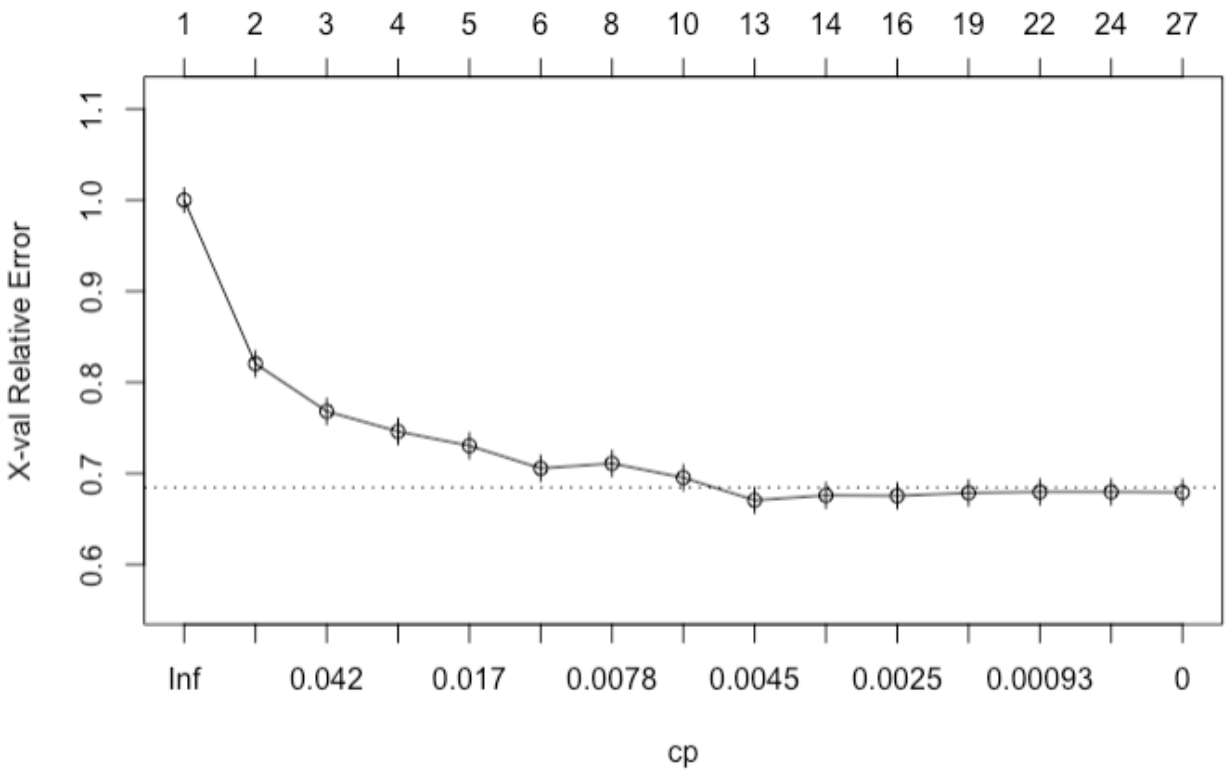
Model 2: Decision Trees (DT) with PCA

The DT model was built with control measures including cp, minsplit and maxdepth using the rpart library from R Studio. In this case, the PCA dataset would be split with 30 predictor variables and parameters as cp=0, minsplit=1 and maxdepth=5. This process involved, building the tree, plotting and summarizing it with cross validation results.

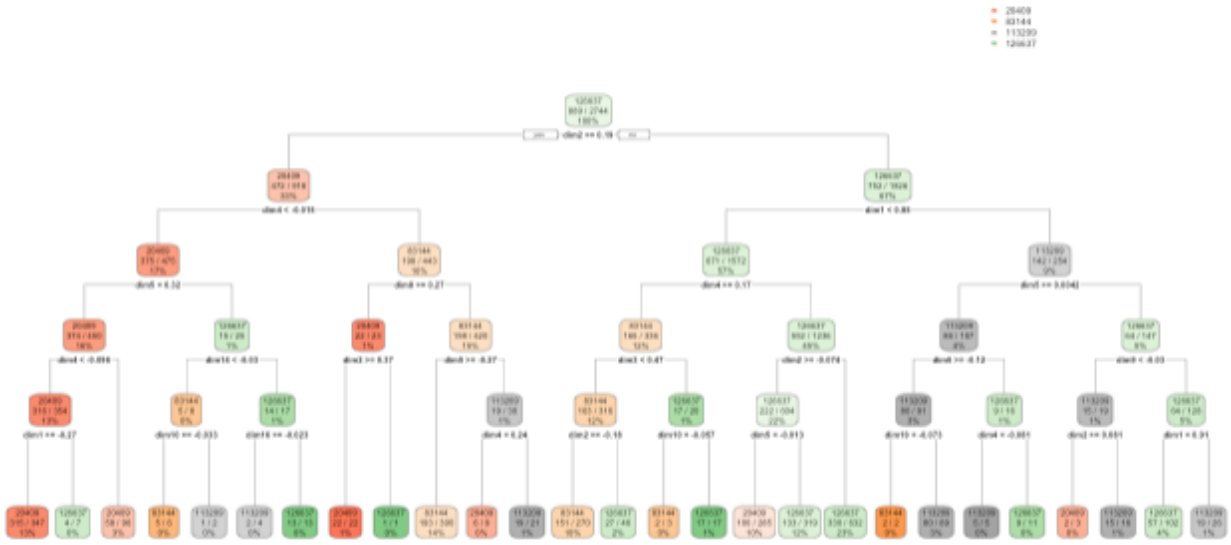
```
# Section 2: Build and tune decision tree models with PCA
  # grow tree
  pca_rtree <- rpart(label~. ,data=pca_train_set, method='class', cp=0,minsplit = 1, maxdepth = 5)
  #summarize rtree values
  summary(pca_rtree)
  plotcp(pca_rtree) # plot cross-validation results
  printcp(pca_rtree) # plot cross-validation results
  # Plot tree | lets Plot decision trees
  rpart.plot(pca_rtree,main="Classification Tree for Images after PCA", extra= 102) # plot decision tree
  rsq.rpart(pca_rtree) # plot approximate R-squared and relative error for different splits (2 plots)
```

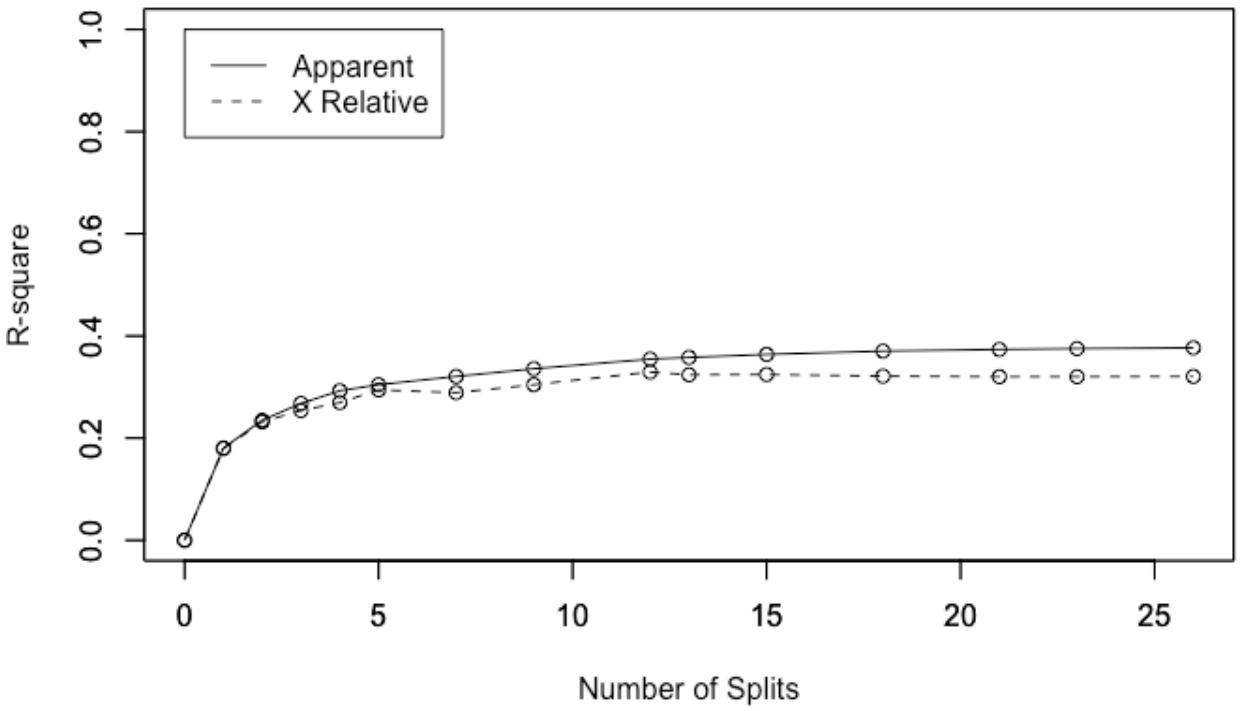
```
Call:
rpart(formula = label ~ ., data = pca_train_set, method = "class",
    cp = 0, minsplit = 1, maxdepth = 5)
  n= 2744

             CP nsplit rel error    xerror       xstd
1  0.1805929919      0 1.0000000 1.0000000 0.01321560
2  0.0544474394      1 0.8194070 0.8204852 0.01403482
3  0.0328840970      2 0.7649596 0.7681941 0.01410892
4  0.0247978437      3 0.7320755 0.7460916 0.01411892
5  0.0118598383      4 0.7072776 0.7304582 0.01411838
6  0.0080862534      5 0.6954178 0.7056604 0.01410457
7  0.0075471698      7 0.6792453 0.7110512 0.01410892
8  0.0061994609      9 0.6641509 0.6954178 0.01409422
9  0.0032345013     12 0.6452830 0.6706199 0.01405788
10 0.0029649596     13 0.6420485 0.6760108 0.01406714
11 0.0021563342     15 0.6361186 0.6754717 0.01406625
12 0.0010781671     18 0.6296496 0.6787062 0.01407149
13 0.0008086253     21 0.6264151 0.6797844 0.01407318
14 0.0005390836     23 0.6247978 0.6797844 0.01407318
15 0.0000000000     26 0.6231806 0.6792453 0.01407234

Variable importance
 dim2  dim4  dim5  dim1  dim8  dim9  dim6  dim7 dim10  dim3 dim14 dim11 dim19 dim20 dim13 dim23 dim16
   25    24    10     9     5     4     4     4     3     3     1     1     1     1     1     1     1
```

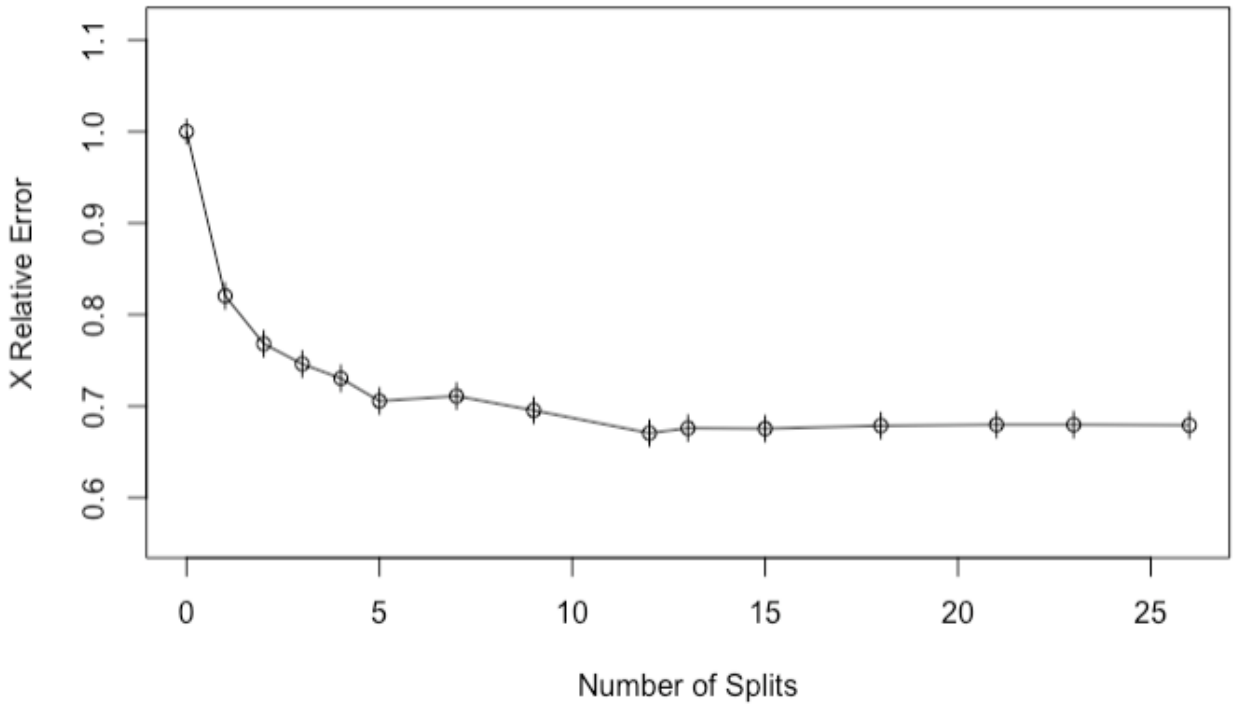
```
Node number 1: 2744 observations,    complexity param=0.180593
  predicted class=126637  expected loss=0.6760204  P(node) =1
    class counts:   714   677   464   889
   probabilities: 0.260 0.247 0.169 0.324
  left son=2 (918 obs) right son=3 (1826 obs)
  Primary splits:
      dim2  < 0.1920394    to the right, improve=145.64180, (0 missing)
      dim4  < -0.1820366   to the left,  improve=114.28550, (0 missing)
      dim1  < 0.8508237    to the left,  improve= 62.99443, (0 missing)
      dim15 < 0.06059013   to the right, improve= 23.79906, (0 missing)
      dim20 < 0.1143907    to the right, improve= 22.98311, (0 missing)
  Surrogate splits:
      dim4  < -0.2596543   to the left,  agree=0.693, adj=0.082, (0 split)
      dim8  < 0.2484494    to the right, agree=0.675, adj=0.028, (0 split)
      dim9  < -0.2265304   to the left,  agree=0.669, adj=0.010, (0 split)
      dim5  < -0.4537964   to the left,  agree=0.668, adj=0.008, (0 split)
      dim11 < 0.2737261    to the right, agree=0.668, adj=0.008, (0 split)

Node number 2: 918 observations,    complexity param=0.05444744
  predicted class=20409   expected loss=0.4858388  P(node) =0.3345481
    class counts:   472   246    63   137
   probabilities: 0.514 0.268 0.069 0.149
  left son=4 (475 obs) right son=5 (443 obs)
  Primary splits:
      dim4 < -0.01540942  to the left,  improve=107.90190, (0 missing)
      dim2 < 0.3340554    to the right, improve= 19.61587, (0 missing)
      dim6 < -0.04367292  to the right, improve= 15.70118, (0 missing)
      dim1 < -0.179588    to the right, improve= 15.58040, (0 missing)
      dim5 < 0.3229042    to the left,  improve= 14.19351, (0 missing)
  Surrogate splits:
      dim7  < -0.04439555  to the right, agree=0.595, adj=0.160, (0 split)
      dim6  < -0.0967257   to the right, agree=0.585, adj=0.140, (0 split)
      dim10 < 0.02898234   to the left,  agree=0.581, adj=0.131, (0 split)
      dim9  < 0.01585291   to the right, agree=0.578, adj=0.126, (0 split)
      dim8  < 0.04878242   to the left,  agree=0.569, adj=0.106, (0 split)
```



Classification Tree for Images after PCA




Apparent
X Relative
R-square
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25
Number of Splits


X Relative Error
0.6
0.7
0.8
0.9
1.0
1.1
0
5
10
15
20
25
Number of Splits

```r
# Section 3: Prediction | Test Phase
pca_predict_unseen <- predict(pca_rtree, pca_test_set, type = 'class')
# predict_unseen
pca_table_test_matrix <- table(pca_test_set$label, pca_predict_unseen)
cat("\n\nPrediction results : Confusion Matrix \n\n")
# table_mat
confusionMatrix(pca_table_test_matrix)
```

```
Prediction results : Confusion Matrix

Confusion Matrix and Statistics

        pca_predict_unseen
         20409 83144 113209 126637
  20409     96    31      1     37
  83144     29    92      2     70
  113209     3    17     24     59
  126637    25    53      6    142

Overall Statistics

               Accuracy : 0.5153
                 95% CI : (0.4772, 0.5532)
    No Information Rate : 0.4483
    P-Value [Acc > NIR] : 0.0002502

                  Kappa : 0.3201

 Mcnemar's Test P-Value : 3.101e-11

Statistics by Class:

                     Class: 20409 Class: 83144 Class: 113209 Class: 126637
Sensitivity                0.6275       0.4767       0.72727        0.4610
Specificity                0.8708       0.7955       0.87920        0.7784
Pos Pred Value             0.5818       0.4767       0.23301        0.6283
Neg Pred Value             0.8908       0.7955       0.98459        0.6399
Prevalence                 0.2227       0.2809       0.04803        0.4483
Detection Rate             0.1397       0.1339       0.03493        0.2067
Detection Prevalence       0.2402       0.2809       0.14993        0.3290
Balanced Accuracy          0.7491       0.6361       0.80324        0.6197
```

Model 3: k-Nearest Neighbors (kNN) with PCA

Next a kNN model was built using the earlier PCA data set, that is with 4 categorical values and 30 predictor variables. kNN models are termed as lazy learning models as it does not construct a general internal model rather simply stores instances of the training data.

```
# kNN - Training the KNN model
pca_training_kNN <- trainControl(method = "repeatedcv", number = 10, repeats = 3)
set.seed(3033)
pca_knn_fit <- train(label ~ ., data = pca_train_set, method = "knn", trControl=pca_training_kNN
                     , preProcess = c("center", "scale")
                     , tuneLength = 10)
pca_knn_fit
```

```
k-Nearest Neighbors

2744 samples
  30 predictor
   4 classes: '20409', '83144', '113209', '126637'

Pre-processing: centered (30), scaled (30)
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 2470, 2469, 2470, 2470, 2468, 2468, ...
Resampling results across tuning parameters:

  k   Accuracy   Kappa
   5  0.5773613  0.4400201
   7  0.5738421  0.4361845
   9  0.5638988  0.4237844
  11  0.5624438  0.4228346
  13  0.5595276  0.4192064
  15  0.5534599  0.4121099
  17  0.5462986  0.4037145
  19  0.5416823  0.3980295
  21  0.5309890  0.3852299
  23  0.5279521  0.3807847

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was k = 5.
```

```
#kNN - Prediction
pca_knn_test_pred <- predict(pca_knn_fit, newdata =pca_test_set)
View(data.frame(pca_knn_test_pred, pca_test_set$label))
summary(pca_knn_test_pred)
```

	pca_knn_test_pred	pca_test_set.label
1	20409	20409
2	20409	20409
3	113209	20409
4	20409	20409
5	20409	20409
6	20409	20409
7	20409	20409
8	20409	20409
9	20409	20409
10	20409	20409
11	20409	20409
12	20409	20409
13	20409	20409
14	20409	20409
15	20409	20409
16	83144	20409
17	20409	20409
18	20409	20409
19	83144	20409
20	20409	20409
21	83144	20409
22	20409	20409
23	20409	20409
24	20409	20409
25	20409	20409
26	20409	20409

Model 4: Support Vector Machines (SVM) with Linear Kernel

Let's build a SVM model using the PCA data set on the Linear kernel, that is with 4 categorical values and 30 predictor variables. SVM models are a representation of training data as points in space separated into categories by a clear gap that is as wide as possible. Number of support vectors identified defines how accurately the model can predict variables.

```
#SVM - Building the Model | Linear Kernel

pca_svm_model <- svm(label ~ ., data = pca_train_set, kernel= 'linear', cost =100, scale = FALSE
                     , Probability = TRUE, Cross = 3,type = 'C')
summary(pca_svm_model)
```

```
Call:
svm(formula = label ~ ., data = pca_train_set, kernel = "linear", cost = 100,
Probability = TRUE, Cross = 3, type = "C", scale = FALSE)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  100

Number of Support Vectors:  2174

 ( 592 392 732 458 )


Number of Classes:  4

Levels:
 20409 83144 113209 126637
```

```r
#SVM - Prediction | Linear Kernel
pca_svmPred <- predict(pca_svm_model, pca_test_set, type = 'prob')
pca_svmPred <- as.data.frame(pca_svmPred)
colnames(pca_svmPred) <- 'results'


svmResults <- pca_test_set %>% select(label) %>% bind_cols(pca_svmPred) %>%
  mutate(real = factor(as.character(str_remove(label, 'V')))
         , prediction = factor(as.character(str_remove(results, 'V'))))

confusionMatrix(svmResults$real, svmResults$prediction) #54.88%
```

```
Confusion Matrix and Statistics

          Reference
Prediction 113209 126637 20409 83144
    113209     40     42     3    18
    126637     26    139    14    47
    20409       5     29   104    27
    83144      10     60    29    94

Overall Statistics

               Accuracy : 0.5488
                 95% CI : (0.5107, 0.5864)
    No Information Rate : 0.393
    P-Value [Acc > NIR] : < 2e-16

                  Kappa : 0.3772

 Mcnemar's Test P-Value : 0.03664

Statistics by Class:

                     Class: 113209 Class: 126637 Class: 20409 Class: 83144
Sensitivity                0.49383        0.5148       0.6933       0.5054
Specificity                0.89604        0.7914       0.8864       0.8024
Pos Pred Value             0.38835        0.6150       0.6303       0.4870
Neg Pred Value             0.92979        0.7158       0.9119       0.8138
Prevalence                 0.11790        0.3930       0.2183       0.2707
Detection Rate             0.05822        0.2023       0.1514       0.1368
Detection Prevalence       0.14993        0.3290       0.2402       0.2809
Balanced Accuracy          0.69493        0.6531       0.7899       0.6539
```

Model 5: Support Vector Machines (SVM) with Radial Kernel

Let's build an SVM model using the PCA data set on the radial kernel. i.e. 4 categorical values and 30 predictor variables. SVM models are representation of training data as points in space separated into categories by a clear gap that is as wide as possible. Number of support vectors identified defines how accurately the model can predict variables.

```
#SVM - Building the Model | Radial Kernel | cost 10
pca_svm_model_radial <- svm(label ~ ., data = pca_train_set, kernel= 'radial', cost =100
                            , scale = FALSE, Probability = TRUE, Cross = 3,type = 'C')
summary(pca_svm_model_radial)
```

```
Call:
svm(formula = label ~ ., data = pca_train_set, kernel = "radial", cost = 100, Probability = TRUE,
Cross = 3,
    type = "C", scale = FALSE)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  100

Number of Support Vectors:  2172

 ( 582 395 745 450 )


Number of Classes:  4

Levels:
 20409 83144 113209 126637
```

```
#SVM - Prediction
pca_svmPred_radial <- predict(pca_svm_model_radial, pca_test_set, type = 'prob')

#Plot the results
pca_svmPred_radial <- as.data.frame(pca_svmPred_radial)
colnames(pca_svmPred_radial) <- 'results'
svmResults_radial <- pca_test_set %>% select(label) %>% bind_cols(pca_svmPred_radial) %>%
mutate(real = factor(as.character(str_remove(label, 'V'))) , prediction =
factor(as.character(str_remove(results, 'V'))))
#Build Confusion matrix
pca_SVM_radialMatrix <- confusionMatrix(svmResults_radial$real
                                        , svmResults_radial$prediction) #94.48 %
pca_SVM_radialMatrix
pca_SVM_radialMatrix$overall
```

```
Confusion Matrix and Statistics

          Reference
Prediction 113209 126637 20409 83144
    113209     47     38     2    16
    126637     28    141    16    41
    20409       5     28   108    24
    83144       7     57    24   105

Overall Statistics

               Accuracy : 0.5837
                 95% CI : (0.5458, 0.6209)
    No Information Rate : 0.3843
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.4267

 Mcnemar's Test P-Value : 0.0575

Statistics by Class:

                     Class: 113209 Class: 126637 Class: 20409 Class: 83144
Sensitivity                0.54023        0.5341       0.7200       0.5645
Specificity                0.90667        0.7991       0.8939       0.8244
Pos Pred Value             0.45631        0.6239       0.6545       0.5440
Neg Pred Value             0.93151        0.7332       0.9195       0.8360
Prevalence                 0.12664        0.3843       0.2183       0.2707
Detection Rate             0.06841        0.2052       0.1572       0.1528
Detection Prevalence       0.14993        0.3290       0.2402       0.2809
Balanced Accuracy          0.72345        0.6666       0.8069       0.6944
```

Model 6: Support Vector Machines (SVM) with Polynomial Kernel

Let's build an SVM model using the PCA data set on the polynomial kernel. i.e. 4 categorical values and 30 predictor variables. SVM models are representation of training data as points in space separated into categories by a clear gap that is as wide as possible. Number of support vectors identified defines how accurately the model can predict variables.

```
#SVM - Building the Model | Ploynomial Kernel | cost 100
pca_svm_model_poly <- svm(label ~ ., data = pca_train_set, kernel= 'polynomial', cost =100
                          , scale = FALSE, Probability = TRUE, Cross = 3,type = 'C')
summary(pca_svm_model_poly)
```

```
Call:
svm(formula = label ~ ., data = pca_train_set, kernel = "polynomial", cost = 100
    Probability = TRUE, Cross = 3, type = "C", scale = FALSE)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  100
     degree:  3
     coef.0:  0

Number of Support Vectors:  2729

 ( 677 714 874 464 )


Number of Classes:  4

Levels:
 20409 83144 113209 126637
```

```
#SVM - Prediction
pca_svmPred_poly <- predict(pca_svm_model_poly, pca_test_set, type = 'prob')

summary(pca_svmPred_poly)
```

```
20409   83144  113209  126637
    0       0       0     687
```

Model 7: Random Forest (RF) with 100 trees

Let's build a random forest model using the PCA data set on the polynomial kernel. i.e. 4 categorical values and 30 predictor variables. SVM models are representation of training data as points in space separated into categories by a clear gap that is as wide as possible. Number of support vectors identified defines how accurately the model can predict variables.

```
# Random Forest prediction of PCA data
  rf_fit <- randomForest(y=pca_train_set$label, x=pca_train_set[2:ncol(pca_train_set)],
data=pca_train_set, ntree=100 , keep.forest=FALSE, importance=TRUE)
```

```
Call:
 randomForest(x = pca_train_set[2:ncol(pca_train_set)], y = pca_train_set$label,
ntree = 100, importance = TRUE, keep.forest = FALSE, data = pca_train_set)
               Type of random forest: classification
                     Number of trees: 100
No. of variables tried at each split: 5

        OOB estimate of  error rate: 37.5%
Confusion matrix:
       20409 83144 113209 126637 class.error
20409    525    59     11    119   0.2647059
83144     80   382     16    199   0.4357459
113209    25    47    217    175   0.5323276
126637    73   156     69    591   0.3352081
```

	X20409 <dbl>	X83144 <dbl>	X113209 <dbl>	X126637 <dbl>	MeanDecreaseAccuracy <dbl>	MeanDecreaseGini <dbl>
dim1	11.5804459	7.42814741	17.721272	-0.06676139	17.857287	112.98281
dim2	28.2172975	5.17227203	13.646266	19.66052113	28.006408	190.53779
dim3	5.8980376	7.40593353	4.287873	4.09208315	9.217755	73.17246
dim4	28.8486273	15.06063747	11.416778	7.06055851	24.879798	190.29013
dim5	9.4224108	2.49121902	12.381095	6.96918782	12.676467	85.65394
dim6	9.7982749	4.45208344	6.446922	5.49230870	11.505764	71.80820
dim7	7.7205226	4.96494867	6.322877	4.40501294	10.159144	60.82395
dim8	8.2808192	2.62165523	5.522450	3.75218187	8.975621	63.78033
dim9	4.3627266	1.01498689	5.657895	5.06423032	8.455902	61.58105
dim10	5.3251472	1.52768691	6.640370	3.60227830	7.780939	63.81761
dim11	3.8651963	0.14104813	4.242418	1.63533518	4.881682	57.94760
dim12	5.4200952	2.35113072	7.605564	5.34802325	9.703087	67.52137
dim13	4.0322740	2.55005854	5.393341	1.08347729	5.296344	55.26954
dim14	5.0814515	2.83879906	5.095199	3.17276591	7.361375	58.26932
dim15	6.1496505	-0.09942668	6.537755	3.71720187	6.737086	59.26148
dim16	3.6332715	3.44008487	4.571976	2.54717599	6.677459	49.24575
dim17	2.3998708	3.63995127	1.496614	3.30318575	6.047491	45.82192
dim18	2.9855977	-1.33429636	3.230557	2.08312912	3.635766	49.13004
dim19	2.0134547	1.23349470	7.129853	0.77112552	4.738822	54.70977
dim20	5.1966583	0.65648321	6.737234	1.31457020	6.222931	59.69016
dim21	3.3617495	2.85753776	3.667741	1.75941640	4.925454	50.57983
dim22	3.2472608	-0.42191551	5.180438	2.07801629	5.395330	50.92815
dim23	2.3481211	0.43312185	4.995309	1.56719415	4.693222	49.18898
dim24	1.2970354	0.73894499	6.074260	0.66926504	4.166517	49.52945
dim25	2.5308085	0.99433576	6.455794	0.81453327	4.226335	51.00267
dim26	0.6460289	-1.42852158	3.544913	-0.12176190	1.374375	46.11069
dim27	1.5867727	1.15673451	1.263251	2.41838458	3.158946	48.35782
dim28	2.2317249	1.84205392	6.685225	-0.26240513	4.792039	53.65248
dim29	0.9237997	0.68915673	1.512335	2.66036952	3.048272	45.13575
dim30	2.4074996	0.65807494	4.418315	2.42959087	4.735113	47.73733

30 rows

Binary Classifier

In this section, the goal was to classify and predict the images with 2 labels using traditional classifiers such as Decision trees, k-Nearest Neighbors(kNN), Supply Vector Machines (SVM) and Random Forest Trees. In addition, this experiment also uses a binary and multi-class classifier approach to solve and compare the accuracy among conventional vs deep learning models.

- 20409 - Tombs
- 83144 - Huts

Data Preprocessing

Used the same dataset as that of experiments with *Traditional Model: Multiclass classifiers* earlier; however, just that the labels 20409 and 83144 would be retained in the new dataframe.

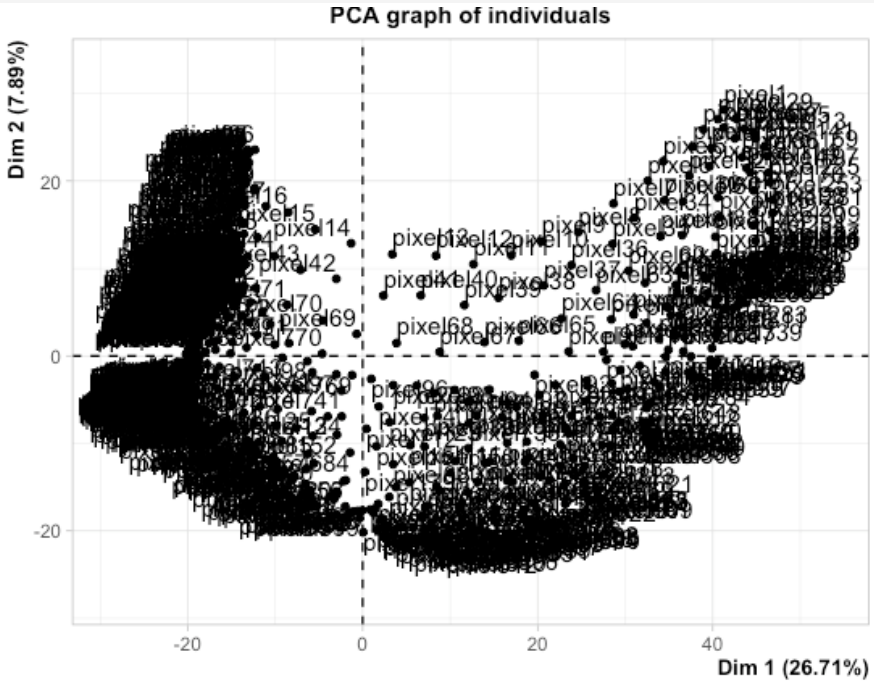
```
## Bind rows in a single dataset
trainImages2DF <- rbind(train_data_20409, train_data_83144)
dim(trainImages2DF) #[1] 3431  785

#convert label column to factor
trainImages2DF$label <- as.factor(trainImages2DF$label)
dim(trainImages2DF)
str(trainImages2DF)
```

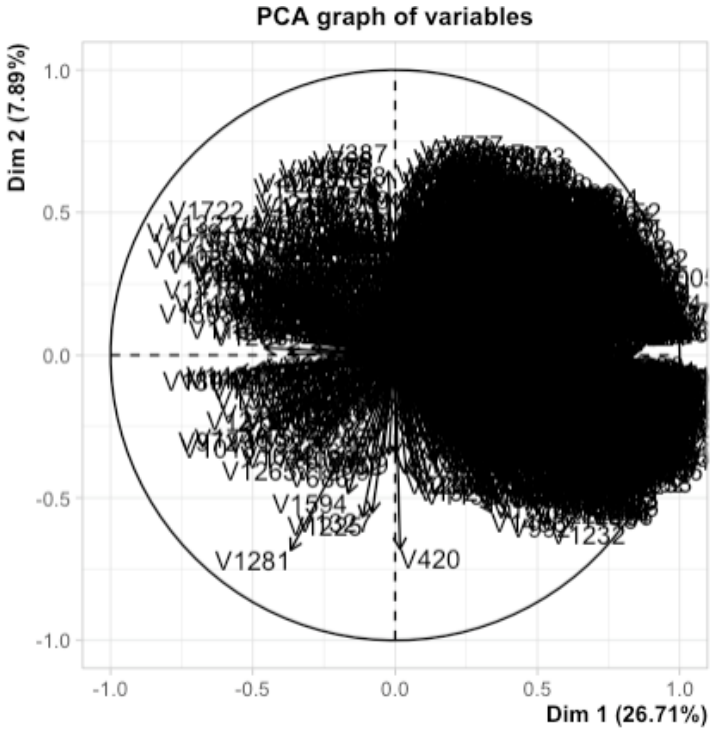
```
'data.frame':   1749 obs. of  785 variables:
 $ label   : Factor w/ 2 levels "20409","83144": 1 1 1 1 1 1 1 1 1 1 ...
 $ pixel1  : num  1 1 1 0.67 0.599 ...
 $ pixel2  : num  1 1 1 0.81 0.628 ...
 $ pixel3  : num  1 1 1 0.703 0.642 ...
 $ pixel4  : num  1 1 1 0.739 0.644 ...
 $ pixel5  : num  1 1 1 0.628 0.66 ...
 $ pixel6  : num  1 1 1 0.763 0.677 ...
 $ pixel7  : num  1 0.983 1 0.756 0.686 ...
 $ pixel8  : num  1 0.935 1 0.767 0.693 ...
 $ pixel9  : num  1 0.524 1 0.739 0.718 ...
 $ pixel10 : num  1 0.482 1 0.749 0.725 ...
```

Data Preprocessing - Dimensionality reduction using Principal Component Analysis (PCA)

As the number of predictor variables are still large (784) - dimensionality reduction method is applied to enhance the dataset. Principal Component Analysis (PCA) is used to reduce the number of dimensions to 30.

```{r}
#Principal Component Analysis
pca_2digits <- PCA(t(select(trainImages2DF,-label)),ncp=30)
```





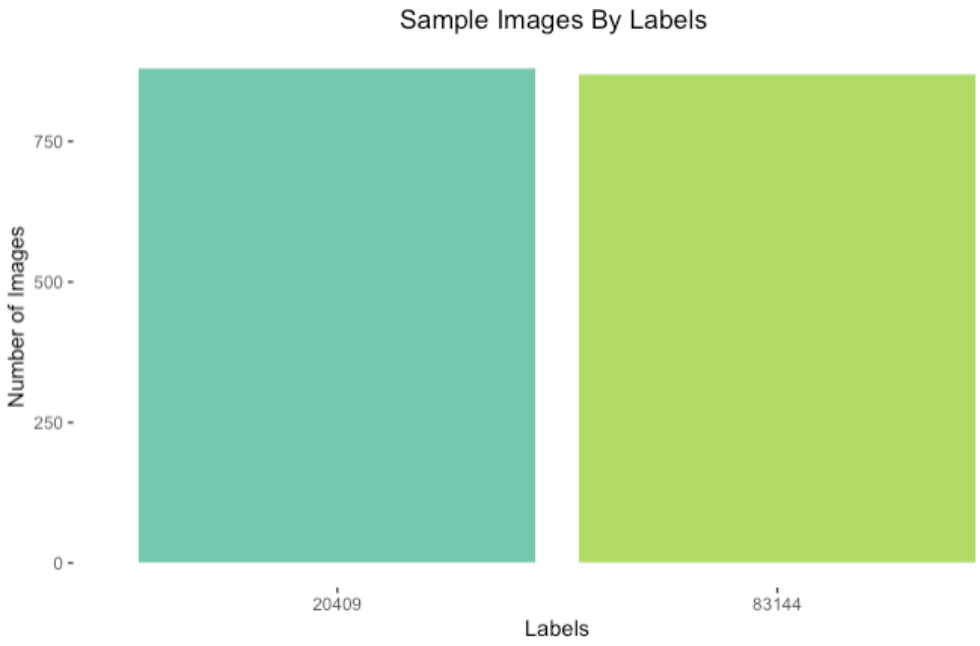
```
#Prepare dataframe from PCA Output
trainPCAImages2DF <- data.frame(trainImages2DF$label,pca_2digits$var$coord)
dim(trainPCAImages2DF) #[1] 3431   31

str(trainPCAImages2DF)
```

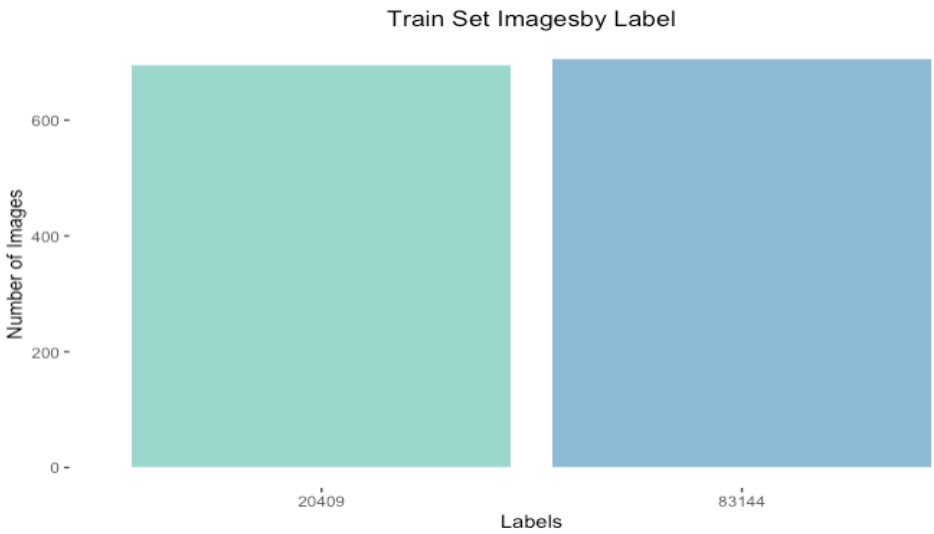
```
[1] 1749   31
'data.frame':   1749 obs. of  31 variables:
 $ trainImages2DF.label: Factor w/ 2 levels "20409","83144": 1 1 1 1 1 1 1 1 1 1 ...
 $ Dim.1               : num  0.6319 0.5093 0.5656 -0.0813 0.3066 ...
 $ Dim.2               : num  0.406 0.385 0.463 0.161 -0.293 ...
 $ Dim.3               : num  -0.0488 0.1007 -0.018 -0.0948 -0.19 ...
 $ Dim.4               : num  -0.0411 0.0629 -0.1445 -0.065 -0.1614 ...
 $ Dim.5               : num  -0.1094 0.2616 0.1904 -0.1227 0.0256 ...
 $ Dim.6               : num  -0.0202 0.2353 -0.0598 0.1827 -0.0479 ...
 $ Dim.7               : num  -0.0386 0.1191 0.0872 0.0338 0.1697 ...
 $ Dim.8               : num  0.1851 -0.0506 -0.0812 0.0192 -0.1213 ...
 $ Dim.9               : num  0.0793 -0.0174 -0.0757 0.0425 0.0945 ...
 $ Dim.10              : num  -0.1157 0.0194 0.0488 0.1053 0.2753 ...
 $ Dim.11              : num  0.0545 0.0104 0.1207 0.1983 -0.0115 ...
 $ Dim.12              : num  0.0912 0.1483 0.0466 -0.2459 -0.0793 ...
 $ Dim.13              : num  0.00572 -0.01459 0.06642 -0.02223 -0.04702 ...
 $ Dim.14              : num  -0.0115 0.1275 -0.2074 0.0767 -0.1526 ...
 $ Dim.15              : num  -0.192 -0.0426 -0.0531 0.0623 -0.131 ...
 $ Dim.16              : num  -0.02 -0.0107 -0.0521 0.0443 -0.0537 ...
 $ Dim.17              : num  -0.014 -0.0815 0.0418 0.1008 0.0643 ...
 $ Dim.18              : num  -0.1955 -0.0629 -0.0162 -0.1201 -0.0784 ...
 $ Dim.19              : num  0.0394 -0.0616 -0.0642 0.0836 -0.0796 ...
 $ Dim.20              : num  0.0261 -0.0652 -0.0448 0.1134 0.1168 ...
 $ Dim.21              : num  -0.0747 -0.0644 -0.0793 0.0194 0.043 ...
 $ Dim.22              : num  0.1007 -0.0753 -0.1704 0.0617 0.051 ...
 $ Dim.23              : num  -0.05302 0.00501 -0.02799 -0.14204 0.1601 ...
 $ Dim.24              : num  0.0216 -0.0619 -0.037 0.0427 -0.0303 ...
 $ Dim.25              : num  -0.02368 0.0251 -0.10337 -0.06766 -0.00594 ...
 $ Dim.26              : num  0.03252 -0.037 -0.07584 0.00533 -0.0082 ...
 $ Dim.27              : num  0.0861 -0.0149 -0.0163 -0.0377 -0.0255 ...
 $ Dim.28              : num  -0.0601 0.0152 0.0934 0.0638 -0.0413 ...
 $ Dim.29              : num  0.127 -0.0611 -0.0419 0.133 0.0161 ...
 $ Dim.30              : num  0.0379 -0.0082 0.0526 0.0811 -0.1188 ...
```

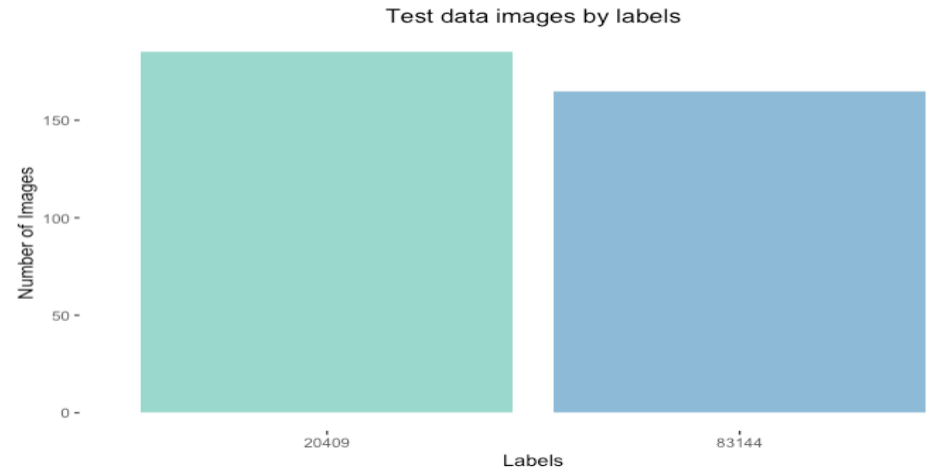
```
#Rename Columns
colnames(trainPCAImages2DF) <- c("label","dim1","dim2","dim3","dim4","dim5","dim6","dim7","dim8","dim9","dim10","dim11","dim12","dim13"
                                 ,"dim14","dim15","dim16","dim17","dim18","dim19","dim20","dim21","dim22","dim23","dim24","dim25"
                                 ,"dim26","dim27","dim28","dim29","dim30")
str(trainPCAImages2DF)
dim(trainPCAImages2DF)
```

```r
# Prepare train and test data from the revised PCA Train dataset.
  sample_size = floor(0.80*nrow(trainPCAImages2DF))
  # set seed to ensure you always have same random numbers generated #324 has 100% training accuracy
  train_index = sample(seq_len(nrow(trainPCAImages2DF)),size = sample_size)
  # train_index
  pca_train_set_binary =trainPCAImages2DF[train_index,] #creates the training dataset with row numbers stored in train_index
  # pca_train_set
  # # table(train_data$author)
  pca_test_set_binary=trainPCAImages2DF[-train_index,]  # creates the test dataset excluding the row numbers mentioned in train_index
  # # table(test_data$author)
  cat("\nImages by Labels:")
  table(trainPCAImages2DF$label)
  cat("\nTrain_set - Images by Labels:")
  table(pca_train_set_binary$label)
  cat("\nTest_set - Images by Labels:")
  table(pca_test_set_binary$label)
```

```
Images by Labels:
20409 83144
  879   870

Train_set - Images by Labels:
20409 83144
  694   705

Test_set - Images by Labels:
20409 83144
  185   165
```







Model 1: Decision Trees (DT) with PCA and Binary Classifier

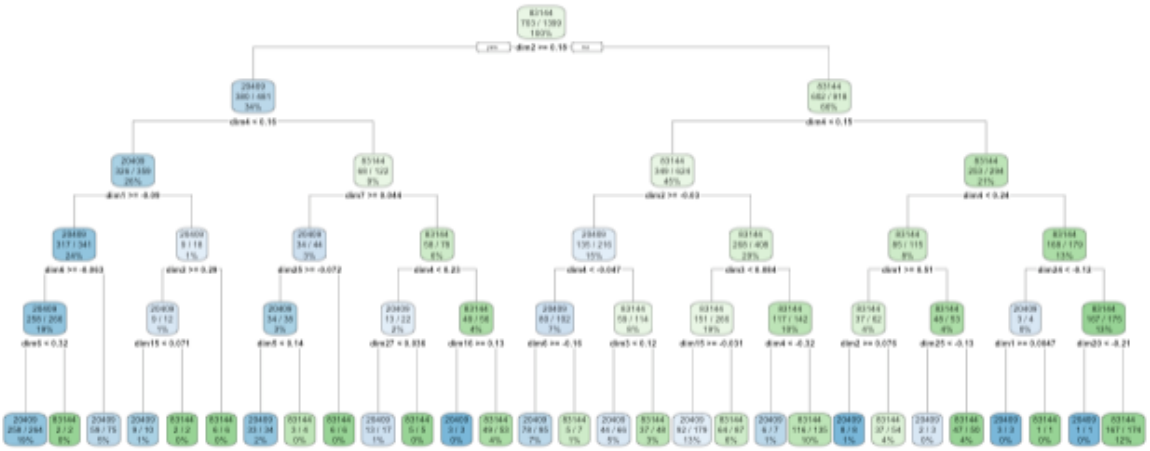
The DT model is built with control measures including cp, minsplit and maxdepth using the rpart library from R Studio. Initial case used the PCA data set split with 30 predictor variables and parameters as cp=0, minsplit=1 and maxdepth=5. This process involves, building the tree, plotting and summarizing it with cross validation results.

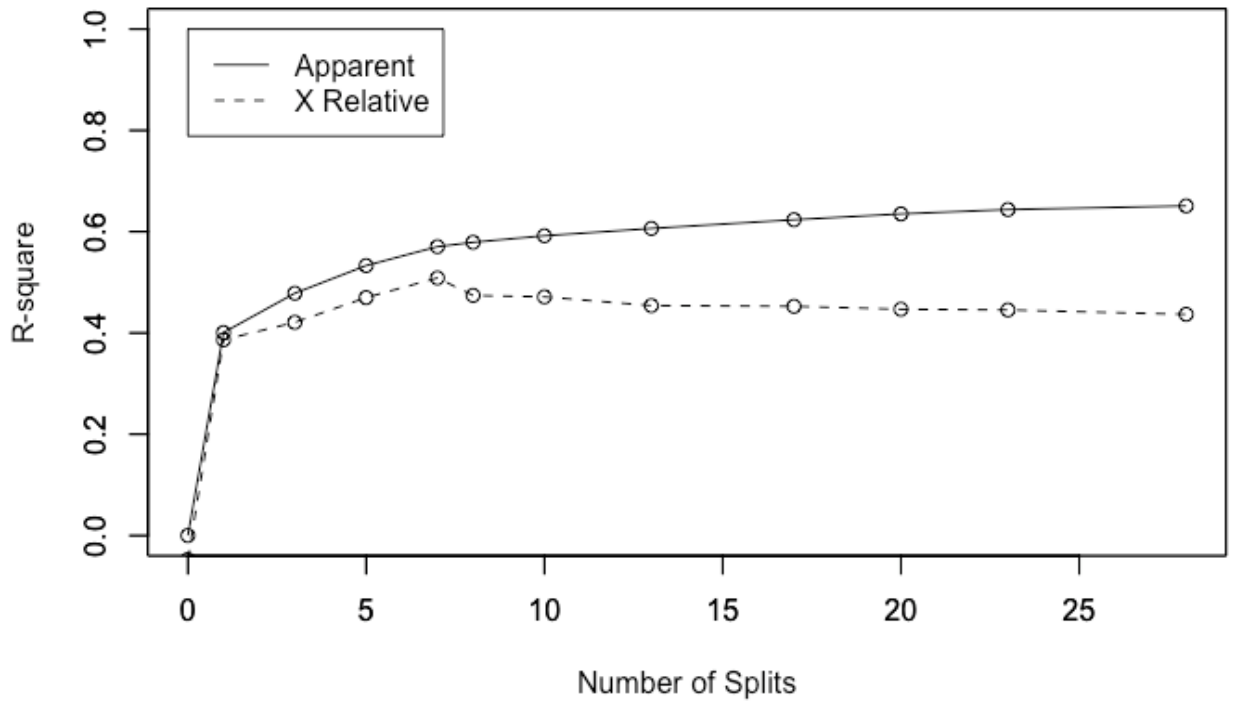
```
# Section 2: Build and tune decision tree models with PCA
  # grow tree
  pca_rtree_binary <- rpart(label~. ,data=pca_train_set_binary, method='class', cp=0,minsplit = 1, maxdepth = 5)
  #summarize rtree values
  summary(pca_rtree_binary)
  plotcp(pca_rtree_binary) # plot cross-validation results
  printcp(pca_rtree_binary) # plot cross-validation results
  # Plot tree | lets Plot decision trees
  rpart.plot(pca_rtree_binary,main="Classification Tree for Images after PCA", extra= 102) # plot decision tree
  rsq.rpart(pca_rtree_binary) # plot approximate R-squared and relative error for different splits (2 plots)
```

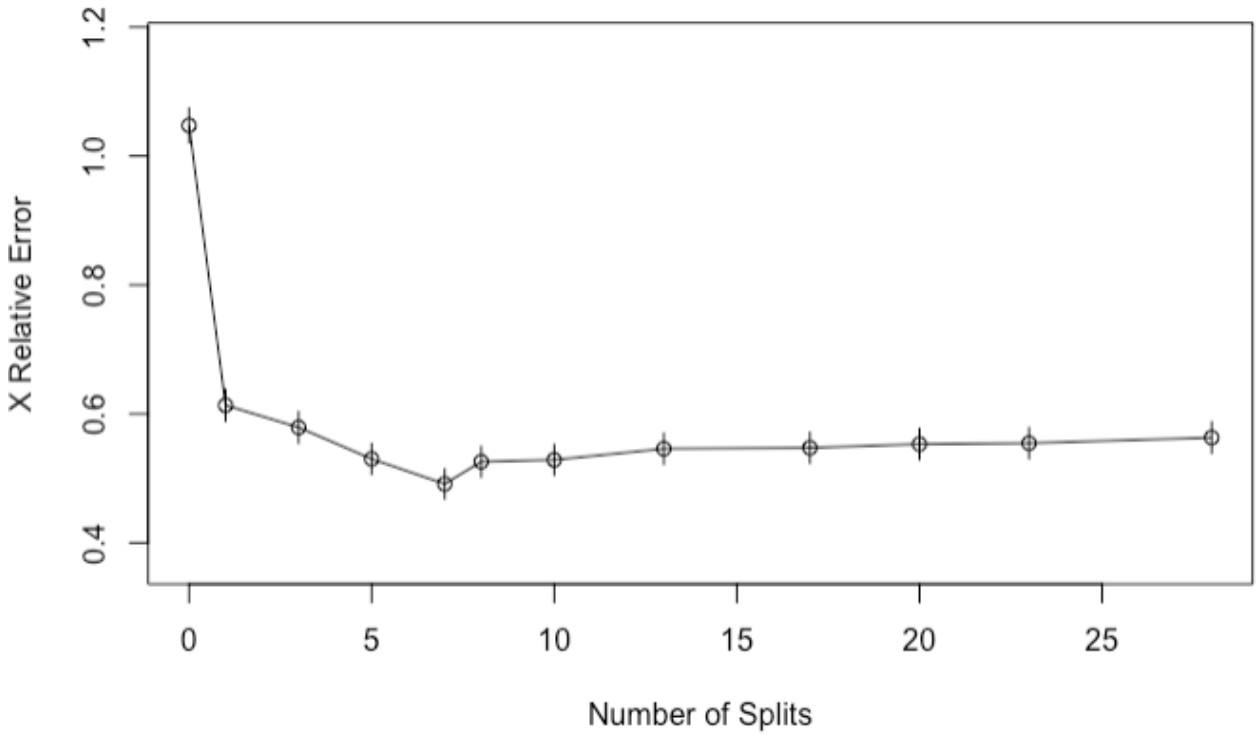
```
Call:
rpart(formula = label ~ ., data = pca_train_set_binary, method = "class",
    cp = 0, minsplit = 1, maxdepth = 5)
  n= 1399

            CP nsplit rel error    xerror       xstd
1  0.400862069      0 1.0000000 1.0474138 0.02684622
2  0.038793103      1 0.5991379 0.6135057 0.02474737
3  0.027298851      3 0.5215517 0.5790230 0.02433683
4  0.018678161      5 0.4669540 0.5301724 0.02368174
5  0.008620690      7 0.4295977 0.4913793 0.02309577
6  0.006465517      8 0.4209770 0.5258621 0.02361960
7  0.004789272     10 0.4080460 0.5287356 0.02366111
8  0.004310345     13 0.3936782 0.5459770 0.02390346
9  0.003831418     17 0.3764368 0.5474138 0.02392315
10 0.002873563     20 0.3649425 0.5531609 0.02400111
11 0.001436782     23 0.3563218 0.5545977 0.02402040
12 0.000000000     28 0.3491379 0.5632184 0.02413458

Variable importance
 dim2  dim4  dim3  dim7  dim1 dim15  dim6 dim25 dim11 dim10 dim14 dim16
   31    23     7     5     4     4     3     3     2     2     2     1
dim22
    1
dim27  dim5 dim21 dim13 dim24 dim23  dim9 dim20  dim8 dim19 dim12
    1     1     1     1     1     1     1     1     1     1     1
```

```
Classification tree:
rpart(formula = label ~ ., data = pca_train_set_binary, method = "class",
    cp = 0, minsplit = 1, maxdepth = 5)

Variables actually used in tree construction:
 [1] dim1  dim15 dim16 dim2  dim20 dim24 dim25 dim27 dim3  dim4  dim5  dim6  dim7

Root node error: 696/1399 = 0.4975

n= 1399

          CP nsplit rel error  xerror     xstd
1  0.4008621      0   1.00000 1.04741 0.026846
2  0.0387931      1   0.59914 0.61351 0.024747
3  0.0272989      3   0.52155 0.57902 0.024337
4  0.0186782      5   0.46695 0.53017 0.023682
5  0.0086207      7   0.42960 0.49138 0.023096
6  0.0064655      8   0.42098 0.52586 0.023620
7  0.0047893     10   0.40805 0.52874 0.023661
8  0.0043103     13   0.39368 0.54598 0.023903
9  0.0038314     17   0.37644 0.54741 0.023923
10 0.0028736     20   0.36494 0.55316 0.024001
11 0.0014368     23   0.35632 0.55460 0.024020
12 0.0000000     28   0.34914 0.56322 0.024135
```

Classification Tree for Images after PCA




Apparent
X Relative
R-square
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25
Number of Splits


X Relative Error
0.4
0.6
0.8
1.0
1.2
0
5
10
15
20
25
Number of Splits

```r
# Section 3: Prediction | Test Phase
  pca_predict_unseen_binary <- predict(pca_rtree_binary, pca_test_set_binary, type = 'class')
  # predict_unseen
  pca_table_test_matrix_binary <- table(pca_test_set_binary$label, pca_predict_unseen_binary)
  cat("\n\nPrediction results : Confusion Matrix \n\n")
  # table_mat
  confusionMatrix(pca_table_test_matrix_binary)
```

```
Prediction results : Confusion Matrix

Confusion Matrix and Statistics

       pca_predict_unseen_binary
        20409 83144
  20409   150    33
  83144    51   116

               Accuracy : 0.76
                 95% CI : (0.7117, 0.8038)
    No Information Rate : 0.5743
    P-Value [Acc > NIR] : 2.979e-13

                  Kappa : 0.5167

 Mcnemar's Test P-Value : 0.06362

            Sensitivity : 0.7463
            Specificity : 0.7785
         Pos Pred Value : 0.8197
         Neg Pred Value : 0.6946
             Prevalence : 0.5743
         Detection Rate : 0.4286
   Detection Prevalence : 0.5229
      Balanced Accuracy : 0.7624

       'Positive' Class : 20409
```

Model 2: k-Nearest Neighbors (kNN) with PCA

Let's build a kNN model using the PCA data set. i.e. 2 categorical values and 30 predictor variables. kNN models are termed as lazy learning models as it does not construct a general internal model rather simply stores instances of the training data.

```r
# kNN - Training the KNN model
pca_training_kNN_binary <- trainControl(method = "repeatedcv", number = 10, repeats = 3)
set.seed(3033)
pca_knn_fit_binary <- train(label ~ ., data = pca_train_set_binary, method = "knn", trControl=pca_training_kNN_binary
                    , preProcess = c("center", "scale")
                    , tuneLength = 10)
pca_knn_fit_binary
```

```
k-Nearest Neighbors

1399 samples
  30 predictor
   2 classes: '20409', '83144'

Pre-processing: centered (30), scaled (30)
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 1260, 1258, 1260, 1259, 1258, 1259, ...
Resampling results across tuning parameters:

  k   Accuracy   Kappa
   5  0.7808653  0.5620945
   7  0.7837223  0.5678197
   9  0.7868090  0.5740223
  11  0.7953995  0.5911376
  13  0.7951307  0.5905979
  15  0.7951239  0.5906093
  17  0.7877359  0.5758668
  19  0.7901221  0.5806470
  21  0.7972685  0.5949095
  23  0.8015374  0.6034392

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was k = 23.
```

```
#kNN - Prediction
pca_knn_test_pred_binary <- predict(pca_knn_fit_binary, newdata =pca_test_set_binary)
View(data.frame(pca_knn_test_pred_binary, pca_test_set_binary$label))
summary(pca_knn_test_pred_binary)
```

	pca_knn_test_pred_binary	pca_test_set_binary.label
1	20409	20409
2	20409	20409
3	20409	20409
4	20409	20409
5	20409	20409
6	20409	20409
7	20409	20409
8	20409	20409
9	20409	20409
10	20409	20409
11	20409	20409
12	20409	20409
13	20409	20409
14	20409	20409
15	20409	20409
16	20409	20409
17	20409	20409
18	20409	20409
19	20409	20409
20	20409	20409
21	20409	20409
22	20409	20409
23	20409	20409
24	20409	20409
25	20409	20409

Model 3: Support Vector Machines (SVM) with Linear Kernel & Binary Classifier

Let's build an SVM model using the PCA data set on the Linear kernel. i.e. 2 categorical values and 30 predictor variables. SVM models are representation of training data as points in space separated into categories by a clear gap that is as wide as possible. Number of support vectors identified defines how accurately the model can predict variables.

```
#SVM - Building the Model | Linear Kernel | Binary Classifier

pca_svm_model_binary <- svm(label ~ ., data = pca_train_set_binary, kernel= 'linear'
                            , cost =100, scale = FALSE, Probability = TRUE
                            , Cross = 3,type = 'C')
summary(pca_svm_model_binary)
```

```
Call:
svm(formula = label ~ ., data = pca_train_set_binary, kernel = "linear", cost = 100,
Probability = TRUE, Cross = 3,
    type = "C", scale = FALSE)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  100

Number of Support Vectors:  696

 ( 348 348 )


Number of Classes:  2

Levels:
 20409 83144
```

```r
#SVM - Prediction | Linear Kernel | Binary Classifier
pca_svmPred_binary <- predict(pca_svm_model_binary, pca_test_set_binary, type = 'prob')
pca_svmPred_binary <- as.data.frame(pca_svmPred_binary)
colnames(pca_svmPred_binary) <- 'results'


svmResults_binary <- pca_test_set_binary %>% select(label) %>% bind_cols(pca_svmPred_binary)
%>% mutate(real = factor(as.character(str_remove(label, 'V')))
         , prediction = factor(as.character(str_remove(results, 'V'))))

confusionMatrix(svmResults_binary$real, svmResults_binary$prediction) #73.71%
```

```
Confusion Matrix and Statistics

          Reference
Prediction 20409 83144
     20409   126    59
     83144    33   132

               Accuracy : 0.7371
                 95% CI : (0.6877, 0.7825)
    No Information Rate : 0.5457
    P-Value [Acc > NIR] : 1.235e-13

                  Kappa : 0.477

 Mcnemar's Test P-Value : 0.009149

            Sensitivity : 0.7925
            Specificity : 0.6911
         Pos Pred Value : 0.6811
         Neg Pred Value : 0.8000
             Prevalence : 0.4543
         Detection Rate : 0.3600
   Detection Prevalence : 0.5286
      Balanced Accuracy : 0.7418

       'Positive' Class : 20409
```

Model 4: Support Vector Machines (SVM) with Radial Kernel and Binary Classifier

Let's build an SVM model using the PCA data set on the radial kernel. i.e. 2 categorical values and 30 predictor variables. SVM models are representation of training data as points in space separated into categories by a clear gap that is as wide as possible. Number of support vectors identified defines how accurately the model can predict variables.

```
#SVM - Building the Model | Radial Kernel | cost 100 | Binary Classifier
pca_svm_model_radial_binary <- svm(label ~ ., data = pca_train_set_binary, kernel= 'radial',
cost =100
                           , scale = FALSE, Probability = TRUE, Cross = 3,type = 'C')
summary(pca_svm_model_radial_binary)
```

```
Call:
svm(formula = label ~ ., data = pca_train_set_binary, kernel = "radial",
    cost = 100, Probability = TRUE, Cross = 3, type = "C", scale = FALSE)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  100

Number of Support Vectors:  701

 ( 355 346 )


Number of Classes:  2

Levels:
 20409 83144
```

```
#SVM - Prediction
pca_svmPred_radial_binary <- predict(pca_svm_model_radial_binary, pca_test_set_binary, type =
'prob')

#Plot the results
pca_svmPred_radial_binary <- as.data.frame(pca_svmPred_radial_binary)
colnames(pca_svmPred_radial_binary) <- 'results'
svmResults_radial_binary <- pca_test_set_binary %>% select(label) %>%
bind_cols(pca_svmPred_radial_binary) %>% mutate(real = factor(as.character(str_remove(label,
'V'))) , prediction = factor(as.character(str_remove(results, 'V'))))
#Build Confusion matrix
pca_SVM_radialMatrix_binary <- confusionMatrix(svmResults_radial_binary$real
                                    , svmResults_radial_binary$prediction) #76.57%
pca_SVM_radialMatrix_binary
pca_SVM_radialMatrix_binary$overall
```

```
Call:
svm(formula = label ~ ., data = pca_train_set_binary, kernel = "radial",
    cost = 100, Probability = TRUE, Cross = 3, type = "C", scale = FALSE)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  100

Number of Support Vectors:  682

 ( 344 338 )


Number of Classes:  2

Levels:
 20409 83144



Confusion Matrix and Statistics

          Reference
Prediction 20409 83144
     20409   130    55
     83144    27   138

               Accuracy : 0.7657
                 95% CI : (0.7178, 0.8091)
    No Information Rate : 0.5514
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.5342

 Mcnemar's Test P-Value : 0.002867

            Sensitivity : 0.8280
            Specificity : 0.7150
         Pos Pred Value : 0.7027
         Neg Pred Value : 0.8364
             Prevalence : 0.4486
         Detection Rate : 0.3714
   Detection Prevalence : 0.5286
      Balanced Accuracy : 0.7715

       'Positive' Class : 20409

      Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull AccuracyPValue
  7.657143e-01   5.341665e-01   7.177654e-01   8.091034e-01   5.514286e-01   6.657536e-17
 McnemarPValue
  2.866984e-03
```

Model 5: Support Vector Machines (SVM) with Polynomial Kernel and Binary Classifier

Let's build an SVM model using the PCA data set on the polynomial kernel. i.e. 2 categorical values and 30 predictor variables. SVM models are representation of training data as points in space separated into categories by a clear gap that is as wide as possible. Number of support vectors identified defines how accurately the model can predict variables.

```
 #SVM - Building the Model | Ploynomial Kernel | cost 100 | Binary Classifier
pca_svm_model_poly_binary <- svm(label ~ ., data = pca_train_set_binary, kernel= 'polynomial',
cost =100
                          , scale = FALSE, Probability = TRUE, Cross = 3,type = 'C')
summary(pca_svm_model_poly_binary)
```

```
Call:
svm(formula = label ~ ., data = pca_train_set_binary, kernel = "polynomial",
    cost = 100, Probability = TRUE, Cross = 3, type = "C", scale = FALSE)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  100
     degree:  3
     coef.0:  0

Number of Support Vectors:  1389

 ( 695 694 )


Number of Classes:  2

Levels:
 20409 83144
```

```
#SVM - Prediction
pca_svmPred_poly_binary <- predict(pca_svm_model_poly_binary, pca_test_set_binary, type =
'prob')

summary(pca_svmPred_poly_binary)
```

```
20409 83144
    0   350
```

Model 6: Random Forest (RF) with 100 trees and Binary Classifier

Let's build a random forest model using the PCA data set on the polynomial kernel. i.e. 2 categorical values and 30 predictor variables models are representation of training data as points in space separated into categories by a clear gap that is as wide as possible. Number of support vectors identified defines how accurately the model can predict variables.

```
# Random Forest prediction of PCA data | Binary Classifier
  rf_fit_binary <- randomForest(y=pca_train_set_binary$label,
x=pca_train_set_binary[2:ncol(pca_train_set_binary)], data=pca_train_set_binary, ntree=100 ,
keep.forest=FALSE, importance=TRUE)

  print(rf_fit_binary) # view results
  importance(rf_fit_binary) # importance of each predictor

  rf_importance_binary <- data.frame(importance(rf_fit_binary)) # importance of each predictor
  rf_importance_binary
```

```
Call:
 randomForest(x = pca_train_set_binary[2:ncol(pca_train_set_binary)],      y =
pca_train_set_binary$label, ntree = 100, importance = TRUE,      keep.forest = FALSE, data =
pca_train_set_binary)
               Type of random forest: classification
                     Number of trees: 100
No. of variables tried at each split: 5

        OOB estimate of  error rate: 19.51%
Confusion matrix:
      20409 83144 class.error
20409   528   166   0.2391931
83144   107   598   0.1517730
```

	X20409 <dbl>	X83144 <dbl>	MeanDecreaseAccuracy <dbl>	MeanDecreaseGini <dbl>
dim1	9.504320123	4.46215010	9.4892805	32.59979
dim2	21.056316650	20.24912329	22.9804363	116.64602
dim3	7.791839635	3.79897882	8.6377730	26.83194
dim4	17.714556903	14.16148508	19.7345711	85.35037
dim5	3.986908927	5.32890024	6.5176678	20.28569
dim6	8.120394799	3.54934249	7.5588289	24.33311
dim7	2.872902016	4.39029305	4.5394300	20.41108
dim8	5.843155869	2.41857432	5.3226577	20.57015
dim9	5.688162403	0.80659842	4.1823018	20.73064
dim10	6.058649575	2.26484175	5.4850251	18.60400
dim11	3.675688947	0.23254202	2.5217990	15.92450
dim12	5.670109003	2.88870397	6.2456871	25.26395
dim13	1.605124648	0.71526590	1.6183341	17.14376
dim14	6.964115635	0.52011317	4.6027526	18.45545
dim15	4.457610424	2.87563835	4.5002255	21.63737
dim16	2.128454359	1.09763158	2.5034240	15.24377
dim17	2.158981434	1.42220163	2.2100641	13.83786
dim18	0.558576515	0.19130772	0.4756285	13.27986
dim19	2.107078184	0.43382507	1.7088520	14.33795
dim20	2.282095461	0.19486721	1.9527896	15.05354
dim21	2.939457485	-1.55573126	1.0836099	14.84135
dim22	4.680647320	0.83973304	3.6409994	16.63634
dim23	0.003308768	-0.31873596	-0.1836399	12.06214
dim24	1.077886320	1.70507261	2.2139396	15.64105
dim25	1.846016173	0.19081957	1.3573072	13.62108
dim26	2.995011922	1.39028885	3.0408185	11.23175
dim27	1.948537679	-0.66151227	0.8736547	15.25466
dim28	0.200474691	0.07072954	0.1894169	12.39472
dim29	3.744025595	2.22487008	4.0900530	20.23048
dim30	1.212615281	1.69380094	1.9679487	10.46567

30 rows

Results

Deep Learning: Multiclass Classifiers (large dataset)

This section captures the results around the various experiments outlined earlier:

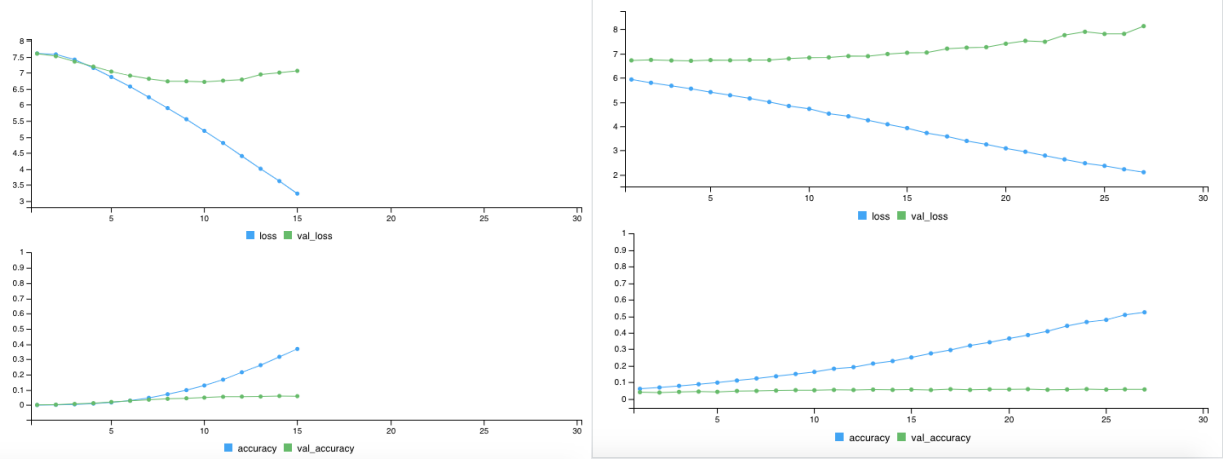


Fig <>: Results showing poor accuracy with Model-1 params

Most experiments took several hours to complete and yielded poor intermediate validation accuracy as seen above. Some of the references indicate running the training for upto 100 epochs so it is possible the training was not done for long enough.

The final experiment with VGG16 was so intensive that it repeatedly starved resources and crashed RStudio. For experiments further, some of the considerations are:

- Setup GPUs for intensive image processing
- Loan or buy a deep-learning VM on Google Cloud Platform (some of the landmark retrieval problems take up to a day to complete at any reasonable scale even on deep-learning VMs)
- Possibly make efficient the known bottleneck layers in VGG16

Deep Learning: Binary/Multiclass Classifiers (reduced dataset)

The second experiment investigated using convolutional neural networks to solve a binary classification problem and a multiclass classification problem. After rounds of testing with hyperparameter-tuning, three binary classification models gave high test accuracy between approximately 97.3% and 97.7%. The same network architecture (except for the output layer unit and activation function) was then used to build the multiclass classifier. Three models of the 4-labels classifier reached test accuracy between approximately 89.8% and 93.1%.

Test Accuracy Comparison

Model	Accuracy (4 labels)	Accuracy (2 labels)
CNN 1 (base model)	91%	97.6%
CNN 2 (augmentation + dropout)	89.8%	97.7%
CNN 3 (augmentation + dropout + LR)	93.1%	97.3%

In comparing the accuracy results between the binary classification models and the 4-label classification models, the former had much higher accuracy. It is understandable because a binary classification task is easier than a multi-classification task.

For both binary and multiclass classifiers —

- The first models had an overfitting issue, which is common with models that are trained on small sample data. It is interesting to note that even with an overfitting issue, both models still reached above 90% test accuracy. The binary model reached a high test accuracy of approximately 97.6%; the multiclass model had approximately 1.2% higher test accuracy than the better-fitted model two.

- The second models used data augmentation and dropout techniques and mostly resolved the overfitting issue. However, the test accuracy did not improve. The binary model has about the same accuracy as the first model and the multiclass model had slightly lower accuracy than the first model. One potential issue could be the dropout rate of 50% being a bit high. Further testing could be done with lower dropout rates.

- The third models used RMSProp's adaptive learning rate (instead of the manual setting of the learning rate as 0.0001 in the previous two models). The test accuracy reduced approximately 0.4% for the binary model but improved approximately 3.3% for the multiclass model. As the learning rate is an important hyperparameter, further tuning of this parameter could be tested to see if an optimal learning rate can be found to improve accuracy. Furthermore, different optimizers, such as Adam (Adaptive Moment Estimation), which is an update to the RMSProp, can be explored and compared the accuracy results with RMSProp.

Activation Maps: What Does a Model See

Activation maps are useful for understanding what the network sees and learns from the decomposition of the input on each of its intermediate layers. It is interesting to note that as the image flows through deeper layers, the activations become increasingly abstract. A look into binary classification model two provided insights on how the network distills the input information, filters out irrelevant visual details, and magnifies useful information about the class of the image. In this sense, it seems to behave like a human brain, which learns to transform visual input into high-level visual concepts, while filtering out irrelevant visual details.

Traditional Models: Binary/Multiclass Classifiers (reduced dataset)

Multi-Class Classifier

From this experiment on traditional classifiers using multi-class/4-labels,

- Dimensionality reduction –number of predictor variables were reduced to 30 with **principal component analysis (PCA)**. Train and test dataset size as below,

```
Images by Labels:
 20409  83144 113209 126637
   879    870    567   1115

Train_set - Images by Labels:
 20409  83144 113209 126637
   707    697    450    890

Test_set - Images by Labels:
 20409  83144 113209 126637
   172    173    117    225
```

  Training data set:

```
2744 samples
  30 predictor
   4 classes: '20409', '83144', '113209', '126637'
```

- **Decision Tree** models – decision tree with actual train and test data sets yielded an accuracy of 50.8%; Whereas, with PCA dataset the accuracy has increased to 51.53%
- **k-Nearest Neighbors (KNN)** – on PCA dataset had better results at k value 5 as 57.73%.
- **Supply Vector Machine (SVM)** models– 3 versions of SVM model built on PCA dataset had slightly better results than DT models.
    - SVM with linear Kernel performed better than DT models at 54.88% accuracy
    - SVM with radial Kernel performed at 58.37%
    - SVM with polynomial all images assigned to label 126637.
- **Random Forest** model predicted with 63.5% on the actual dataset with number of trees set as 100

Overall, experiments on Conventional classifiers with 4 labels had consistent accuracy over 50% with Random Forest with highest followed by SVM's radial kernel.

Binary Classifier

From this experiment on traditional classifiers using binary/2-labels

- Dimensionality reduction –number of predictor variables were reduced to 30 with **principal component analysis** (**PCA**). Train and test dataset size s below,

```
Images by Labels:
20409 83144
  879   870

Train_set - Images by Labels:
20409 83144
  694   705

Test_set - Images by Labels:
20409 83144
  185   165
```

Training data set:

```
1399 samples
  30 predictor
   2 classes: '20409', '83144'
```

- **Decision Tree** models – decision tree with actual train and test data sets yielded an accuracy of 76%; Whereas, with PCA dataset the accuracy has increased to 76%
- **k-Nearest Neighbors (KNN)** – on PCA dataset had better results at k value 23 as 80.15%. **Supply Vector Machine (SVM)** models– 3 versions of SVM model built on PCA dataset had slightly better results than DT models.
    - SVM with linear Kernel performed better than DT models at 73.71% accuracy
    - SVM with radial Kernel performed at 76.57%
    - SVM with polynomial all images assigned to label 83144.

- **Random Forest** model predicted with 81.49% on the actual dataset with number of trees set as 100

Overall, experiments on Conventional classifiers with 2 labels had consistent accuracy over 70% with Random Forest with highest followed by kNN.

Conclusions

The landmark retrieval problem is a paradigm in itself. While there is a steep learning curve to image processing with understanding the architecture of Convolutional Neural Networks (CNN) there are deeper aspects to experimentation at scale. Given the problem complexity at the outset and the resource limitations (lack of compute resources or GPUs), the Team dissected the project into three parts:

- Focusing on exploring CNNs, generators to sift through images at scale and experimentation with convnet parameters to improve prediction
- Deeper understanding of image processing at each layer and other aspects of CNNs while limiting the data-set classifications to binary or multi-class at most
- Exploring traditional classification and prediction algorithms for image processing

Details of datasets, excerpts of code, meaningful outputs and analysis has been presented in an attempt to fully explain the problem at hand and tools that were used to solve it. The findings from the study are:

- For image classification, convolutional neural networks give much higher accuracy in making predictions than traditional classification algorithms, as shown in the table below.
- CNNs can even work well with very small sample sets.
- When dealing with large image data sets, like Google Landmark data, CNNs need much computing power to carry out their “learning”.
- Keras library with TensorFlow backend provides an easy-to-use framework that allows fast prototyping.
- Hyperparameter-tuning of CNNs can get very complex and time-consuming.

Accuracy Comparison for All Models

Model	Accuracy (4 labels)	Accuracy (2 labels)
Decision Tree	53.71%	70%
Random Forest	63.54%	83.70%
KNN	56.73% (k=5)	80.34% (k=17)
SVM Radial	61.43%	78.86%
CNN	93.1%	97.7%

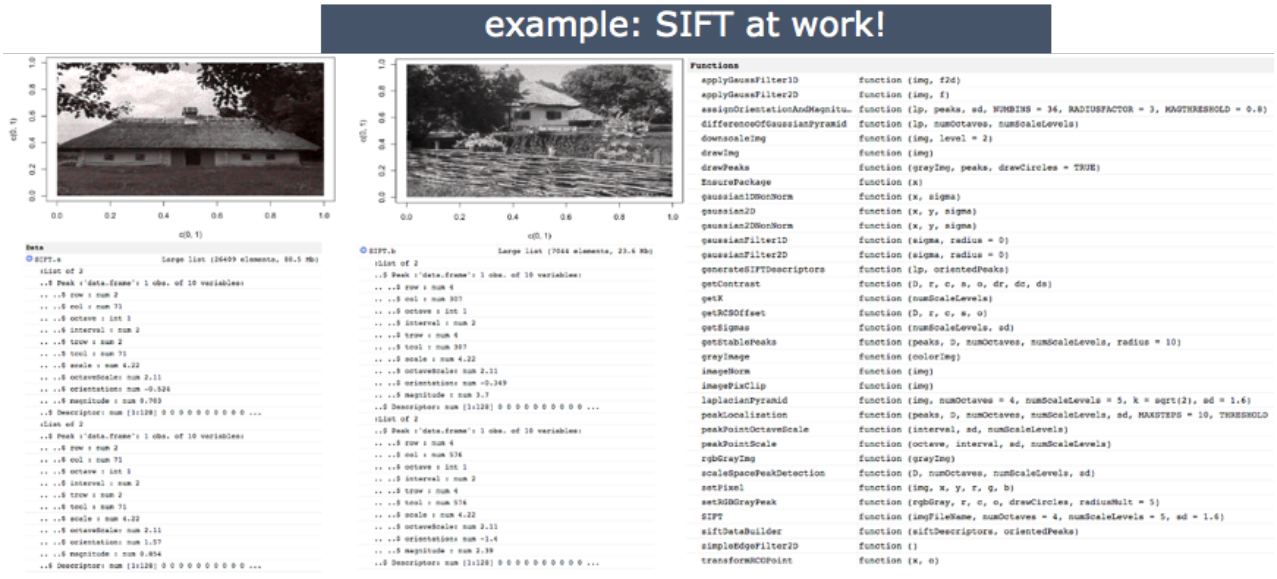
Extended Research & Future work

In addition, our team also looked into image classification using the SIFT algorithm written by D. Lowe from University of British Columbia. The SIFT algorithm is invariant to image scale and rotation and focuses on steps below,

- Scale-space peak selection: Potential location for finding features using difference of Gaussians/scale-space pyramid techniques
- Keypoint Localization: Accurately locating the feature keypoints
- Orientation Assignment: Assigning orientation to keypoints
- Keypoint descriptor: Describing the keypoints as a high dimensional vector

Our study also suggested that the advantages of using SIFT for its,

- Locality - features are local; Hence, robust to occlusion and clutter and no prior segmentation needed.
- Distinctiveness - Often, the features can be stored and matched against a large database of objects.
- Quantity - Many features can be generated for even smaller objects
- Efficiency -With high Compute – results can be close to real-time performance.
- Extensibility - Features can be extended to a wide range of different feature types.



However, considering the short-duration and limited resources; Our team decided to extend this research further by procuring GPUs or utilizing other cloud resources (GCP offers deep-learning VMs) to experiment with models and measure its efficiency at scale.

References

Chollet, François & Allaire, J.J. *Deep Learning with R.* Shelter Island, NY: Manning Publications, 2018. https://www.manning.com/books/deep-learning-with-r.

Kaggle. “Google Landmark Retrieval 2020.” Accessed July 26, 2020. https://www.kaggle.com/c/landmark-retrieval-2020

Kaggle. “Deep Learning.” Accessed August 2020. https://www.kaggle.com/learn/deep-learning.

TensorFlow for R. “Basic Image Classification.” Accessed August 2020. https://tensorflow.rstudio.com/tutorials/beginners/basic-ml/tutorial_basic_classification.

Weyand, Tobias, Andre Araujo, Bingyi Cao, and Jack Sim. “Google Landmarks Dataset v2 - A Large-Scale Benchmark for Instance-Level Recognition and Retrieval.” CVPR’20 (April 2020) https://arxiv.org/abs/2004.01804.