Choose Your Own Project

Steve Kruit

2022-11-21

Sign language classification

1. Overview

Goal

The goal of this project is to train a model to classify images of hand signals for different letters in American Sign Language. Different classification algorithms will be considered, with the final model chosen based on the accuracy of predictions and computing time required.

Dataset

The dataset used is 'Sign Language MNIST', hosted by Kaggle at the following link:

https://www.kaggle.com/datasets/datamunge/sign-language-mnist?resource=download

The dataset format is patterned to match closely with the classic MNIST. Each training and test case represents a label (0-25) as a one-to-one map for each alphabetic letter A-Z (and no cases for 9=J or 25=Z because of gesture motions). The data is similar to the standard MNIST, with a header row of label, pixel1,pixel2....pixel784 which represent a single 28x28 pixel image with grayscale values between 0-255.

A sample of the original images for each of the letters considered is shown below.

And the below image shows what they look like after being converted to 28x28 pixel images with grayscale values between 0-255.

I have downloaded the train and test data files from Kaggle and uploaded to my GitHub repo for this project (the code below will not automatically download from Kaggle itself - due to my employer's IT restrictions

on automated downloads it was not possible for me to do it this way!)

The dataset is available already partitioned into train and test sets of 27,455 and 7,172 rows respectively. For my analysis I combine these two sets into one larger dataset and then re-partition randomly into **training** and **validation** datasets such that the **validation** dataset consists of 10% of the total.

Key steps

As described above, we divide the dataset into **training_data** and **validation_data** sets. The **training_data** set is used for all exploratory data analysis and model training, while the **validation_data** set is set aside and only used for the final assessment of the model's accuracy.

To train the model, we further partition the **training_data** set into **test_set** and **train_set**, this time using a 50/50 split which ensures that there are around 500 different images tested for each of the 24 labels, and also reduces the number of rows in **train_set** to a level that allows for model testing without blowing out calculation times to an unworkable level.

Given the large number of predictor variables, we investigate different methods for dimension reduction, including principal component analysis (PCA), to make the problem more manageable and try to contain computation time. We utilise dimension reduction with 2 different classification algorithms, k-nearest neighbours (kNN) and Random Forest, and adjust model parameters to try to optimise for prediction accuracy and calculation time.

The following models are tested:

- kNN using different numbers of principal components (PCs) identified by PCA
- kNN using a given number of PCs and different values of the parameter k
- Random Forest with differing number of trees used
- Random Forest with a set number of trees and differing values of the nodesize parameter
- Random Forest using the most important predictors only
- Random Forest using different numbers of principal components (PCs) identified by PCA

All of the models above are trained using **train_set** and then tested with **test_set**. The best performing model (as determined by the accuracy of predictions and model run time) is then trained using the entire **training_data** set and used to predict the labels in the **validation_data** set.

2. Methods and analysis

Loading the data and creating training and testing sets

First we load the libraries we will use in our analysis, ensuring we download any missing packages.

```
if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")
if(!require(matrixStats)) install.packages("matrixStats", repos = "http://cran.us.r-project.org")
if(!require(lubridate)) install.packages("lubridate", repos = "http://cran.us.r-project.org")
if(!require(randomForest)) install.packages("randomForest", repos = "http://cran.us.r-project.org")

library(tidyverse)
library(caret)
library(matrixStats)
library(lubridate)
library(randomForest)
```

Then we will upload the train and test sets provided at Kaggle (renaming to **training_data** and **validation_data**) and determine the number of rows in each set, as well as the total number of labels.

```
# import training and validation data
training_data <- read_csv("sign_mnist_train.csv")
validation_data <- read_csv("sign_mnist_test.csv")

# get number of labels and number of rows in training and validation data sets
print(length(unique(training_data$label)))
```

```
## [1] 24
```

```
print(nrow(training_data))
```

```
## [1] 27455
```

```
print(nrow(validation_data))
```

```
## [1] 7172
```

The **validation_data** as provided makes up roughly 20% of the total data. For this exercise I'd like to have a **training_data** set with 90% of the overall data so we will combine both the provided datasets into one larger dataset and then randomly generate new **training_data** and **validation_data**. The **validation_data** will still have around 3,500 rows, equating to more than 100 rows for each label.

```
# combine validation and training data into one dataframe and then randomly partition into
# new validation and training sets with validation making up 10% of the total
all_data <- bind_rows(training_data, validation_data)
set.seed(95, sample.kind = "Rounding")
test_index <- createDataPartition(y = all_data$label, times = 1, p = 0.1, list = FALSE)
training_data <- all_data[-test_index, ]
validation_data <- all_data[test_index, ]
```

We will further partition the **training_data** into a **train_set** and **test_set** for model evaluation, this time using a 50/50 split which ensures that there are around 500 different images tested for each of the 24 labels, and also ensures the number of rows in **train_set** is not so large that calculation times will be too lengthy for testing. We will then convert the tibble of predictor variables (pixel1,. . . ,pixel784) into matrices, and the label data into vectors of factor variables.

We will not use k-fold cross-validation for our analysis as we will be using PCA, and we would need to re-run the PCA calculation (which takes around a minute to complete) for each new partition of the data, which would be time-consuming. Instead we stick with the fairly simple 50/50 partitioning into a **train_set** and **test_set**.

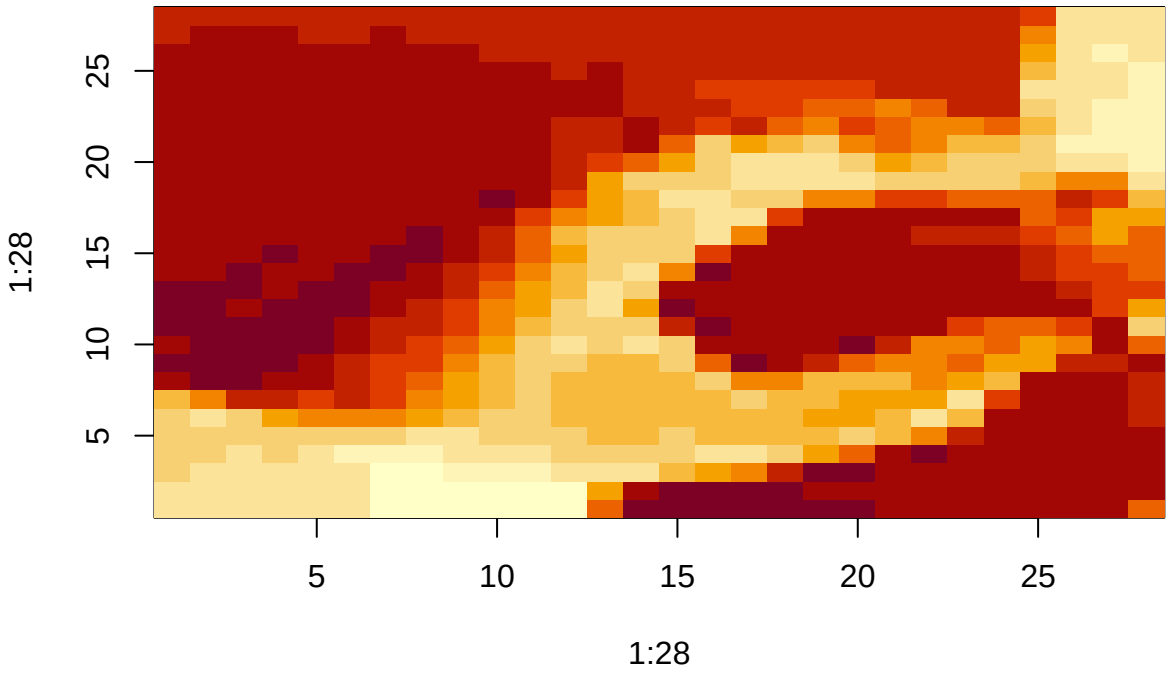
```
# separate training data into train and test sets for model testing
set.seed(9365, sample.kind = "Rounding")
test_index <- createDataPartition(y = training_data$label, times = 1, p = 0.5, list = FALSE)
train_set <- training_data[-test_index,]
test_set <- training_data[test_index,]

# ensure both train and test sets include all possible labels
all(sort(unique(train_set$label)) == sort(unique(test_set$label)))
```

```
## [1] TRUE
```

```
# convert predictors to matrices for analysis
train_x <- train_set %>%
  select(-label) %>%
  as.matrix()
test_x <- test_set %>%
  select(-label) %>%
  as.matrix()
# convert labels to factors
train_y <- factor(train_set$label)
test_y <- factor(test_set$label)
```

To quickly test that the data looks correct, we will display one of the predictor rows as an image.

```
grid <- matrix(train_x[2,], 28, 28) # the 784 predictors represent a 28x28 grid
image(1:28, 1:28, grid[, 28:1]) # image was inverted
```

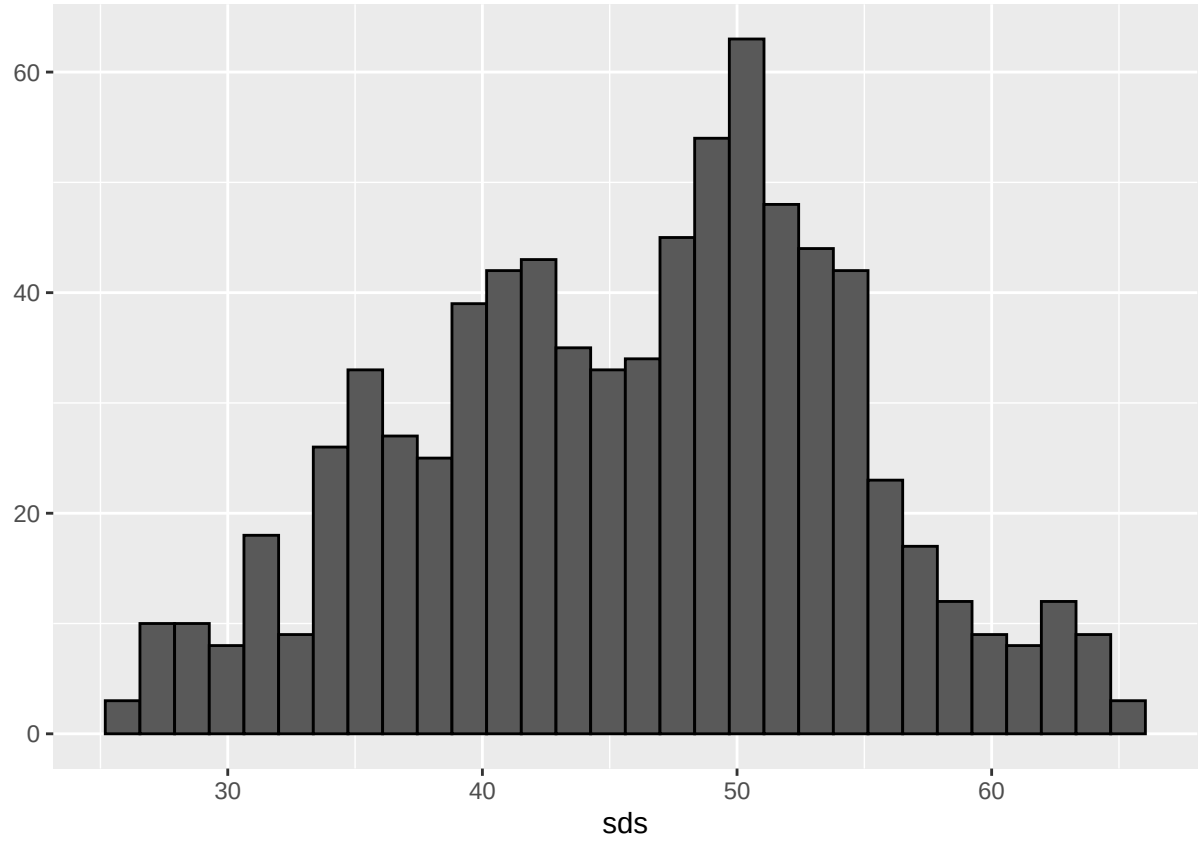


That definitely looks like a hand gesture!

Dimension reduction

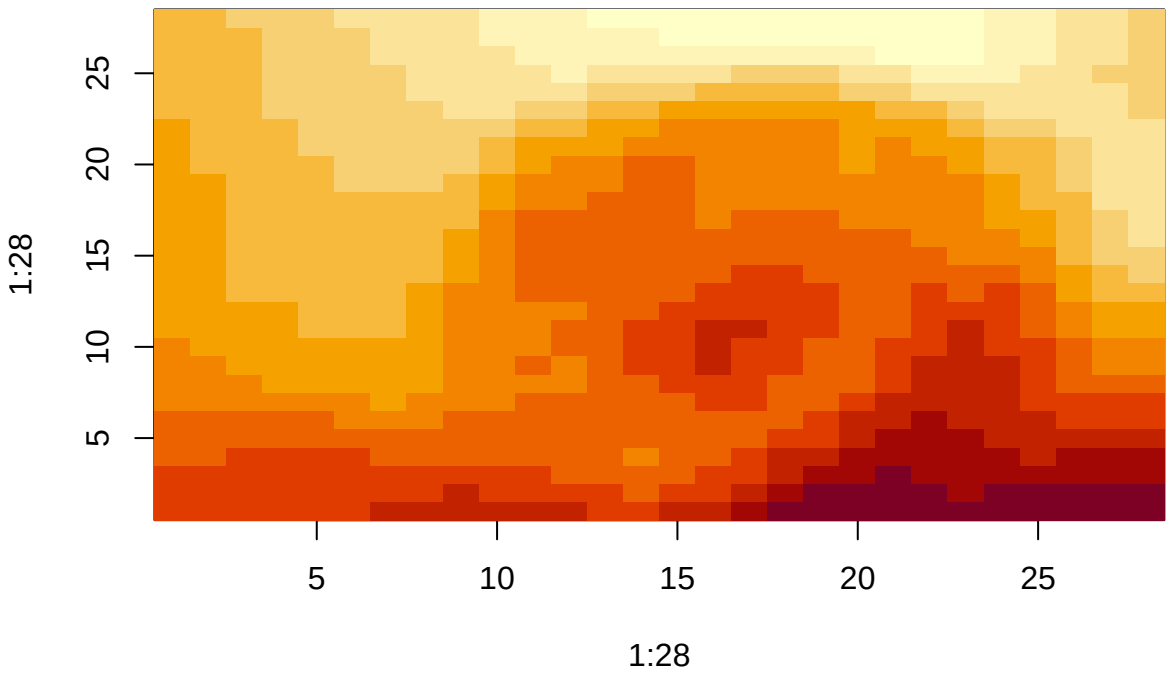
As the hand signals do not take up the entire image, we will analyse the variance (technically the standard deviation, but for simplicity we will refer to variance here) for each pixel to determine whether there are pixels that have very low variance and can therefore be omitted without materially reducing the accuracy of our predictions.

```
sds <- colSds(train_x)
qplot(sds, bins = "30", color = I("black"))
```



This is not a very encouraging result! Unlike previous exercises we have done during this course there is not a large number of low-variance pixels we can rule out to begin with. Instead we have something resembling a Normal distribution. We look at an image of the variance.

```
image(1:28, 1:28, matrix(sds, 28, 28)[, 28:1])
```



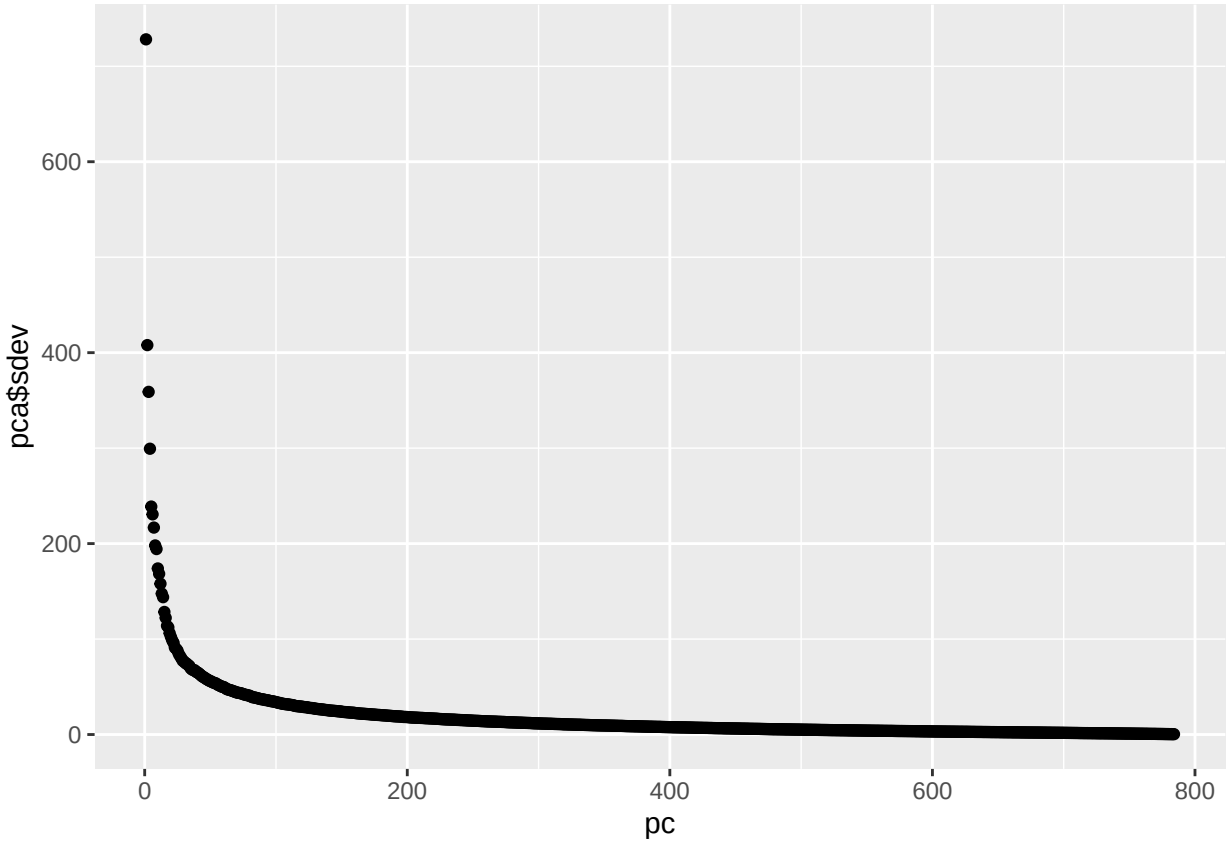
There is a small area at the top centre of the image that has relatively low variance. But generally speaking, there are not any pixels we can easily identify as being removable for our analysis. Using the caret package's nearZeroVar() function confirms that it does not recommend any predictors for removal.

```
nearZeroVar(train_x)
```

```
## integer(0)
```

Since the previous simple variance observation wasn't useful, we will try dimension reduction using principal component analysis (PCA) on **train_set** to allow for exploratory data analysis.

```
pca <- prcomp(train_x)
pc <- 1:784
qplot(pc, pca$sdev)
```



We can see that a relatively small number of predictors account for a large proportion of the overall variance. We look at the 10 highest variance principal components (PCs):

```
summary(pca)$importance[,1:10]
```

```
##                             PC1       PC2       PC3       PC4       PC5
## Standard deviation     728.20068 407.92433 358.92702 299.36696 238.77000
## Proportion of Variance   0.31068   0.09749   0.07548   0.05251   0.03340
## Cumulative Proportion    0.31068   0.40818   0.48366   0.53617   0.56957
##                             PC6       PC7       PC8       PC9      PC10
## Standard deviation     230.63473 216.77895 198.04895 194.28510 173.96175
## Proportion of Variance   0.03116   0.02753   0.02298   0.02212   0.01773
## Cumulative Proportion    0.60073   0.62827   0.65125   0.67336   0.69109
```

10 out of the 784 PCs account for nearly 70% of the variance. The table below shows the number of PCs required to account for different proportions of the overall variance:

```
props <- seq(0.1, 1, 0.1)
tibble(`Proportion of variance` = props,
       `PCs required` = sapply(props, function(p){
         x <- summary(pca)$importance[3,]
         sum(x < p) + 1
       }))
```

```
## # A tibble: 10 x 2
```

```
##    `Proportion of variance` `PCs required`
##                       <dbl>          <dbl>
##  1                      0.1              1
##  2                      0.2              1
##  3                      0.3              1
##  4                      0.4              2
##  5                      0.5              4
##  6                      0.6              6
##  7                      0.7             11
##  8                      0.8             22
##  9                      0.9             59
## 10                      1              761
```

As we saw in the previous table, 1 predictor accounts for more than 30% of the total variability, and 6 predictors account for more than 60%. This table shows we can actually account for 80-90% of variability using only 22-59 PCs! We will fit models using different numbers of PCs to see what trade-offs exist between calculation time and the accuracy of predictions.

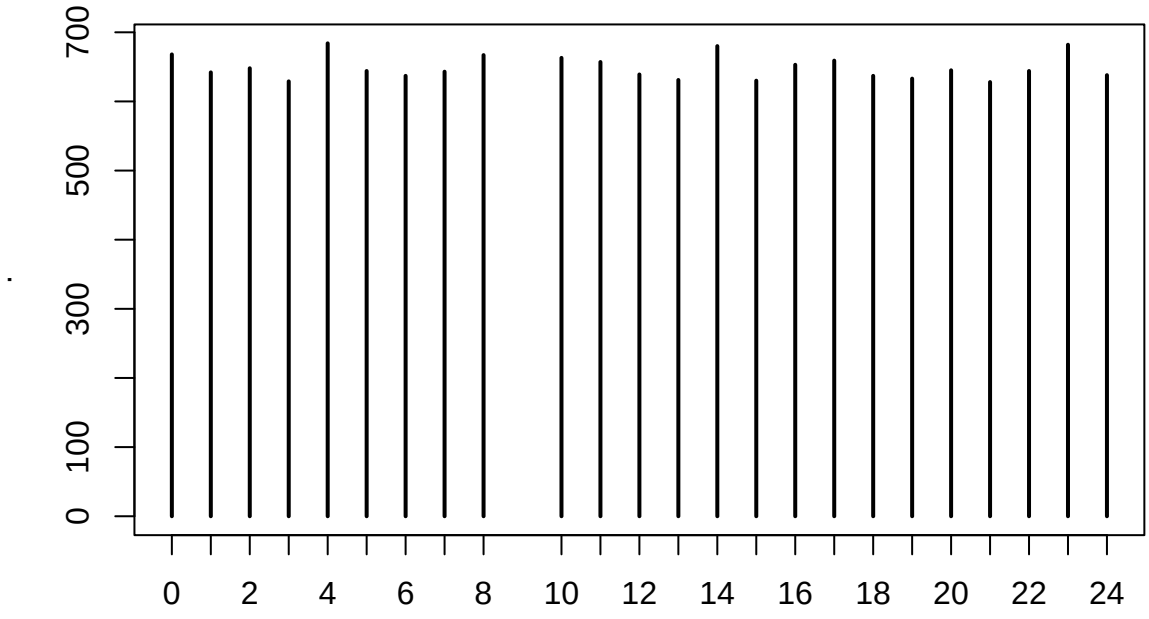
Model training and testing

First we will need to transform the **test_set** in line with the PCA performed on the **train_set** by subtracting the column means and then applying the PCA rotation.

```
col_means <- colMeans(test_x)
transformed_x_test <- sweep(test_x, 2, col_means) %*% pca$rotation
```

To ensure that prediction accuracy is a suitable test of model efficacy, we need to use a balanced dataset.

```
train_y %>%
  table() %>%
  plot()
```



The chart above shows that we have a pretty even distribution of labels (please note that label 9=J has been removed from the data as it includes gesture motions that wouldn't translate to a static image). As the data is balanced, we feel comfortable using prediction accuracy to test the performance of our model.

Model 1: kNN using different numbers of principal components (PCs) identified by PCA
The first model we will try to fit will use the k-nearest neighbours algorithm, which calculates the distance between predictor sets and predicts the classification of data points based on the most common classification of the k nearest data points in the training data. We will try fitting a kNN model using 22 and 59 PCs (corresponding with 80% and 90% of the total predictor variance). To simplify our code, we create a function to calculate model accuracy and calculation time for different numbers of PCs.

```
accuracy_and_time <- function(PCs, y_train = train_y, y_test = test_y,
                              x_test = transformed_x_test, x_train = pca$x){
  # function to calculate model accuracy for a given value of n PCs
  accuracy_by_PCs <- function(n){
    x_train_subset <- x_train[,1:n]
    fit <- knn3(x_train_subset, y_train)
    x_test_subset <- x_test[,1:n]
    y_hat <- predict(fit, x_test_subset, type = "class")
    confusionMatrix(y_hat, y_test)$overall["Accuracy"]
  }
  accs <- c()
  times <- c()
  # iterate through each value of n, calculate model accuracy and calculation time,
  # and add to lists
  for(n in PCs){
```

```
    time1 <- now()
    accs <- c(accs, accuracy_by_PCs(n))
    time2 <- now()
    time_elapsed <- difftime(time2, time1, units = "secs")
    times <- c(times, time_elapsed)
  }
  # return tibble of model accuracy and calc times for each k
  tibble(PCs = PCs, `Model accuracy` = accs, `Calculation time (seconds)` = times)
}
```

Running the function using 22 and 59 PCs gives us the following:

```
accuracy_and_time(c(22, 59))
```

```
## # A tibble: 2 x 3
##     PCs `Model accuracy` `Calculation time (seconds)`
##   <dbl>            <dbl>                        <dbl>
## 1    22            0.939                         11.0
## 2    59            0.973                         32.8
```

We can achieve a model accuracy of 0.97 in only around 25 seconds using 59 PCs. We may be able to get an even higher accuracy using larger numbers of PCs without blowing out calculation times too much. The table below shows the number of PCs required to account for more than 90% of predictor variance.

```
props <- seq(0.91, 1, 0.01)
tibble(`Proportion of variance` = props,
       `PCs required` = sapply(props, function(p){
         x <- summary(pca)$importance[3,]
         sum(x < p) + 1
       }))
```

```
## # A tibble: 10 x 2
##    `Proportion of variance` `PCs required`
##                       <dbl>          <dbl>
##  1                     0.91             66
##  2                     0.92             75
##  3                     0.93             85
##  4                     0.94             98
##  5                     0.95            114
##  6                     0.96            136
##  7                     0.97            166
##  8                     0.98            211
##  9                     0.99            291
## 10                     1               761
```

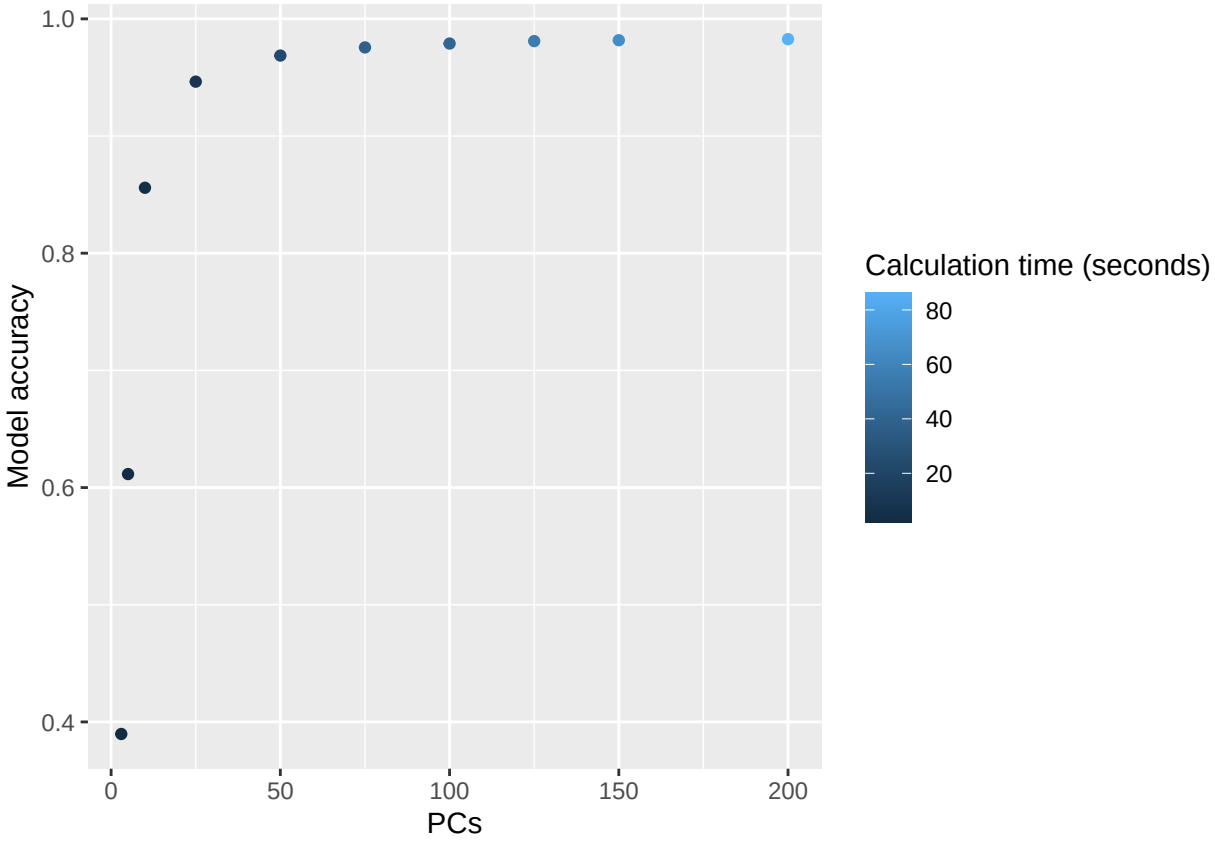
We can explain 95% of the variance using roughly double the number of PCs (114). We will now test the accuracy and runtime for several larger numbers of PCs used in the kNN model (this calculation takes several minutes).

```
# print table and plot accuracy by number of PCs used
PC_accuracy <- accuracy_and_time(c(3, 5, 10, 25, 50, 75, 100, 125, 150, 200))
PC_accuracy
```

```
## # A tibble: 10 x 3
##      PCs `Model accuracy` `Calculation time (seconds)`
##    <dbl>            <dbl>                        <dbl>
##  1     3            0.390                         1.95
##  2     5            0.612                         2.41
##  3    10            0.856                         4.95
##  4    25            0.946                         8.67
##  5    50            0.969                        23.5
##  6    75            0.976                        38.4
##  7   100            0.979                        41.6
##  8   125            0.981                        55.4
##  9   150            0.982                        65.7
## 10   200            0.983                        86.3
```

The table shows that once we have more than 75-100 PCs, we start to get big increases in runtime for very little improvement in model accuracy. We can show this visually:

```
PC_accuracy %>%
  ggplot(aes(PCs, `Model accuracy`, color = `Calculation time (seconds)`)) +
  geom_point()
```



Model 2: kNN using a given number of PCs and different values of the parameter k Next we will test whether we can improve the accuracy from Model 1 by optimising the value of k used (the number of nearest neighbours). We write a new function to calculate model accuracy and runtime using different values of k. We will test this on a model using the first 22 PCs as they explain 80% of the predictor variance,

and using 22 PCs in Model 1 gave us relatively high accuracy of 0.94 with calculation time of only a few seconds.

```
accuracy_and_time_by_k <- function(ks, n = 22, y_train = train_y, y_test = test_y,
                                   x_test = transformed_x_test, x_train = pca$x){
  # function to calculate model accuracy for given values of k
  accuracy_by_k <- function(k, n){
    x_train_subset <- x_train[,1:n]
    fit <- knn3(x_train_subset, y_train, k=k)
    x_test_subset <- x_test[,1:n]
    y_hat <- predict(fit, x_test_subset, type = "class")
    confusionMatrix(y_hat, y_test)$overall["Accuracy"]
  }
  accs <- c()
  times <- c()
  # iterate through each value of k, calculate model accuracy and calculation time,
  # and add to lists
  for(k in ks){
    time1 <- now()
    accs <- c(accs, accuracy_by_k(k, n))
    time2 <- now()
    time_elapsed <- difftime(time2, time1, units = "secs")
    times <- c(times, time_elapsed)
  }
  # return tibble of model accuracy and calc times for each k
  tibble(k = ks, `Model accuracy` = accs, `Calculation time (seconds)` = times)
}
```

We run the function to test different values of k.

```
k_accuracy <- accuracy_and_time_by_k(c(1, 3, 5, 7, 9))
k_accuracy
```

```
## # A tibble: 5 x 3
##       k `Model accuracy` `Calculation time (seconds)`
##   <dbl>            <dbl>                        <dbl>
## 1     1            0.977                         6.30
## 2     3            0.955                         6.14
## 3     5            0.939                         5.93
## 4     7            0.925                         5.85
## 5     9            0.910                         6.18
```

Using k=1 seems to be best, as the model accuracy decreases as k gets larger. This means that only one nearest neighbour will be used in the kNN algorithm, which could risk over-fitting the model, but given that the model is being trained on **train_set** and then tested on **test_set** and we are still being told that k=1 is best, we assume that our predictions are not being harmed by over-fitting here.

Model 3: Random Forest with differing number of trees used Next we will try fitting a Random Forest model. As covered in the course material, Random Forests improve upon the accuracy of decision trees by averaging multiple decision trees. Using the randomForest package, we would usually try the following code:

```
fit_rf <- randomForest(label ~ ., data = training_data)
```

However, due to the size of our dataset the runtime of this calculation would be enormous. One way we can reduce the runtime is to use a matrix input instead of formula input (e.g. randomForest(y = example[, i], x = example[, j:k]) instead of randomForest(y ~ ., data = example)). This alone will not reduce the runtime to suitable levels, though. We will need to adjust model parameters too. To start with, we write a formula to test the Random Forest model using different numbers of trees.

```
# create a formula to test run time and accuracy for multiple values of ntree
accuracy_and_time_by_ntree <- function(ns, x_train=train_x, y_train=train_y, x_test=test_x, y_test=test_
  # function to calculate accuracy for a given value of ntree
  accuracy_by_n <- function(n){
    set.seed(421, sample.kind = "Rounding") # we will use random seed of 421 for all random forest mode
    fit_rf <- randomForest(x = x_train, y = y_train, ntree=n)
    y_hat <- predict(fit_rf, newdata = x_test)
    confusionMatrix(y_hat, y_test)$overall["Accuracy"]
  }
  accs <- c()
  times <- c()
  for(n in ns){
    time1 <- now()
    accs <- c(accs, accuracy_by_n(n))
    time2 <- now()
    times <- c(times, difftime(time2, time1, units = "secs"))
  }
  # return tibble of model accuracy and calc times for each ntree
  tibble(ntree = ns, `Model accuracy` = accs, `Calculation time (seconds)` = times)
}
```

We use the formula with different values of the parameter ntree.

```
ntree_accuracy <- accuracy_and_time_by_ntree(c(4, 8, 16, 32, 64))
ntree_accuracy
```

```
## # A tibble: 5 x 3
##   ntree `Model accuracy` `Calculation time (seconds)`
##   <dbl>            <dbl>                        <dbl>
## 1     4            0.869                         9.42
## 2     8            0.939                        14.8
## 3    16            0.970                        27.1
## 4    32            0.982                        55.5
## 5    64            0.988                       109.
```

We can achieve slightly better accuracy with this Random Forest model than we could with kNN - albeit with a longer calculation time.

Model 4: Random Forest with a set number of trees and differing values of the nodesize parameter We will also try to optimise by the nodesize parameter (using ntree=8 as this gave us relatively high accuracy of 0.94 with a short calculation time). Again we will write a formula to test the model with different values of nodesize.

```
accuracy_and_time_by_nodesize <- function(nodesizes, n=8, x_train=train_x, y_train=train_y, x_test=test
  accuracy_by_nodesize <- function(nodesize){
    set.seed(421, sample.kind = "Rounding") # we will use random seed of 421 for all random forest mode
    fit_rf <- randomForest(x = x_train, y = y_train, ntree=n, nodesize = nodesize)
    y_hat <- predict(fit_rf, newdata = x_test)
    confusionMatrix(y_hat, y_test)$overall["Accuracy"]
  }
  accs <- c()
  times <- c()
  for(nodesize in nodesizes){
    time1 <- now()
    accs <- c(accs, accuracy_by_nodesize(nodesize))
    time2 <- now()
    times <- c(times, difftime(time2, time1, units = "secs"))
  }
  # return tibble of model accuracy and calc times for each ntree
  tibble(nodesize = nodesizes, `Model accuracy` = accs, `Calculation time (seconds)` = times)
}
```

We test model accuracy and runtime with different values of the nodesize parameter.

```
nodesize_accuracy <- accuracy_and_time_by_nodesize(seq(1, 51, 10))
nodesize_accuracy
```

```
## # A tibble: 6 x 3
##   nodesize `Model accuracy` `Calculation time (seconds)`
##      <dbl>            <dbl>                        <dbl>
## 1        1            0.939                         16.8
## 2       11            0.907                         14.8
## 3       21            0.880                         14.9
## 4       31            0.838                         15.1
## 5       41            0.817                         14.7
## 6       51            0.783                         14.8
```

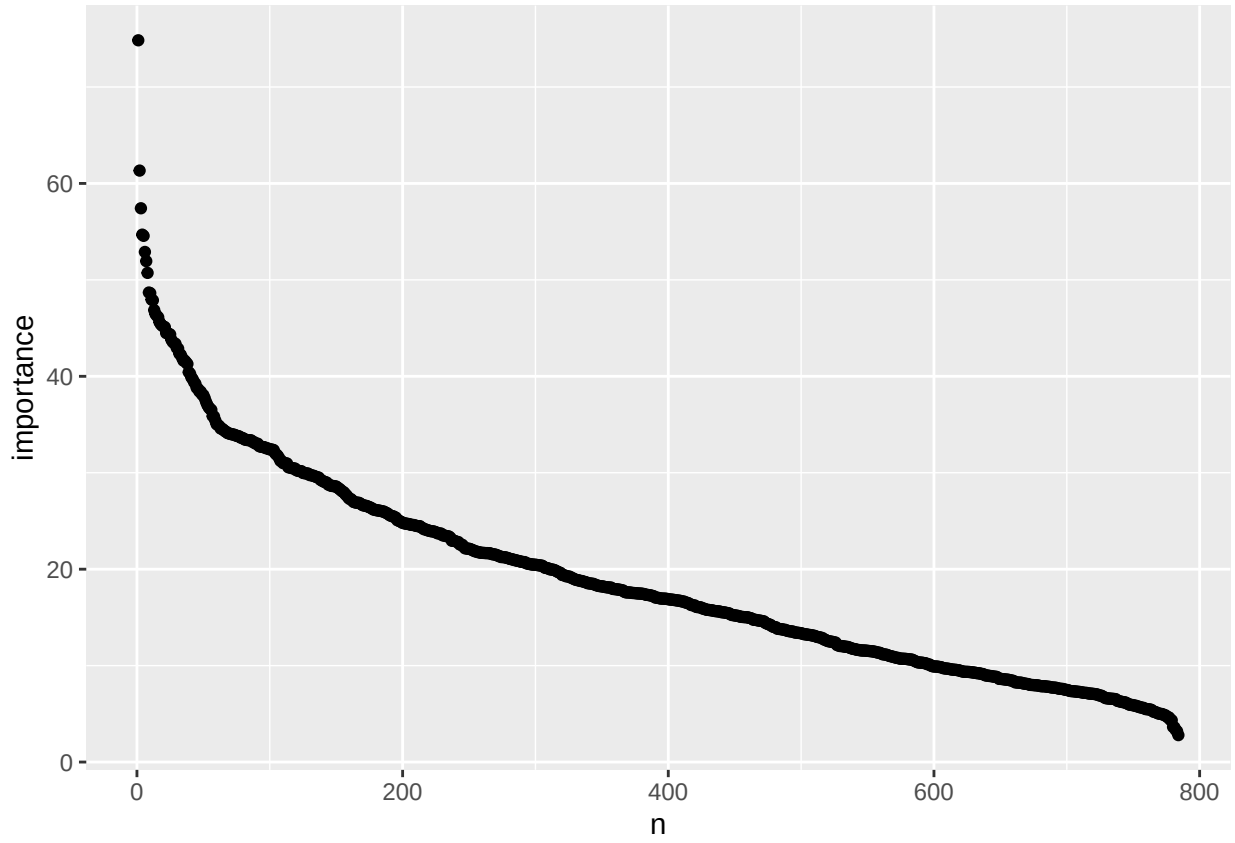
Model accuracy diminishes as we increase nodesize above 1 - so we will leave it as is.

Model 5: Random Forest using the most important predictors only The Random Forest models we have used so far have included all 784 predictors, leading to relatively long calculation times. This has meant that we will need to use models with a relatively low number of trees if we want to contain runtimes. However, the randomForest package also includes the 'importance' function, which determines which predictors have the greatest effect on the model. If we can determine the most important predictors, then we can exclude the less important predictors and hopefully not reduce model accuracy too substantially. A model with fewer predictors will be faster, meaning we may be able to use more trees without runtimes blowing out.

We will fit a Random Forest model, this time using ntree=16, which gave us accuracy of 0.97 (with higher accuracy, we can be more confident that the the most important predictors chosen by the model are correct). We will then use the 'importance' function to determine the predictors with the largest effect on the model (in terms of total decrease in node impurities as measured by the Gini Index) and plot from highest to lowest.

```
set.seed(421, sample.kind = "Rounding") # we will use random seed of 421 for all random forest models w
train_rf <- randomForest(x = train_x, y = train_y, ntree=16) # ntree=16 gave us 97% accuracy
```

```
imp <- tibble(pixel = 1:784, importance = importance(train_rf)) %>%
  arrange(desc(importance))
# visualise the importance of each predictor
imp %>%
  mutate(n = 1, n = cumsum(n)) %>%
  ggplot(aes(n, importance)) +
  geom_point()
```



From looking at the visualisation, we will make the arbitrary choice to select the top 100 most important predictors, as there appears to be a drop in importance after this point. We will then test the accuracy of a Random Forest model using only the top 100 predictors, for various values of ntree.

```
top100 <- imp %>%
  top_n(100, importance) %>%
  pull(pixel)
# test the calculation times for different values of ntree using the top 100
top100_accuracy <- accuracy_and_time_by_ntree(c(4, 8, 16, 32, 64, 128, 256), x_train = train_x[, top100]
top100_accuracy
```

```
## # A tibble: 7 x 3
##   ntree `Model accuracy` `Calculation time (seconds)`
##   <dbl>            <dbl>                        <dbl>
## 1     4            0.866                         1.33
## 2     8            0.936                         2.18
## 3    16            0.965                         3.66
```

```
## 4    32            0.974                      6.87
## 5    64            0.982                     13.4
## 6   128            0.984                     26.3
## 7   256            0.986                     52.3
```

By using the top 100 predictors only we can increase ntree and achieve prediction accuracy of 0.986 with less than a minute of runtime (not including the time taken to calculate the most important predictors).

Model 6: Random Forest using different numbers of principal components (PCs) identified by PCA The approach used for Model 5 picks the most important predictors, but the PCA approach used earlier potentially improved on this by using linear transformations of multiple predictors in each transformed predictor - meaning that the effect of many of the original predictors could potentially be captured using relatively few transformed predictors.

To test whether the use of PCA could improve our Random Forest model, we will write a formula to assess the accuracy and runtime of a Random Forest model fitted to the first n principal components instead of the original predictors. We pick ntree=8 due to relatively high accuracy and speedy calculation time.

```
accuracy_and_time_by_PCs <- function(PCs, ntree=8, y_train = train_y, y_test = test_y,
                                     x_test=transformed_x_test, x_train=pca$x){
  # function to calculate model accuracy for a given value of n PCs
  accuracy_by_PCs <- function(n){
    x_train_subset <- x_train[, 1:n]
    set.seed(421, sample.kind = "Rounding") # we will use random seed of 421 for all random forest mode
    fit_rf <- randomForest(x = x_train_subset, y = y_train)
    x_test_subset <- x_test[, 1:n]
    y_hat <- predict(fit_rf, x_test_subset)
    confusionMatrix(y_hat, y_test)$overall["Accuracy"]
  }
  accs <- c()
  times <- c()
  # iterate through each value of n, calculate model accuracy and calculation time,
  # and add to lists
  for(n in PCs){
    time1 <- now()
    accs <- c(accs, accuracy_by_PCs(n))
    time2 <- now()
    time_elapsed <- difftime(time2, time1, units = "secs")
    times <- c(times, time_elapsed)
  }
  # return tibble of model accuracy and calc times for each k
  tibble(PCs = PCs, `Model accuracy` = accs, `Calculation time (seconds)` = times)
}
```

We test the accuracy and runtime of the PCA Random Forest model for various numbers of PCs.

```
PC_rf_accuracy <- accuracy_and_time_by_PCs(c(5, 10, 15, 20, 25))
PC_rf_accuracy
```

```
## # A tibble: 5 x 3
##     PCs `Model accuracy` `Calculation time (seconds)`
##   <dbl>            <dbl>                        <dbl>
## 1     5            0.817                         15.2
```

```
## 2    10           0.985                    16.9
## 3    15           0.999                    20.4
## 4    20           1.00                     24.4
## 5    25           1.00                     28.6
```

Accuracy (and runtime) appears to be astonishingly good - we have ~100% accuracy with runtime of only around 25 seconds using the first 20 PCs. This actually made me suspicious that I must have somehow trained the model using testing data (or tested the model on the training data) but the PCA and model fitting was completed using only **train_set**, and the predictions and testing were performed by applying the trained model to the transformed **test_set**. The PCA Random Forest approach really does seem to be that accurate!

Given the relative success of this PCA Random Forest model, we select Model 6 (using the first 20 PCs) as our final model for training on the full **training_data** and testing on the **validation_data** set.

3. Model results and performance

For the final test of our prediction model, we will perform PCA on the full **training_data** set and fit a Random Forest model to the top 20 PCs.

First we conduct PCA on the full **training_data** set (after converting the predictors to a matrix). This took around 90 seconds on my computer.

```
training_x <- training_data %>%
  select(-label) %>%
  as.matrix()
final_pca <- prcomp(training_x)
```

We then fit a Random Forest model to the first 20 PCS. This took around a minute on my computer.

```
training_y <- factor(training_data$label)
set.seed(421, sample.kind = "Rounding")
fit_rf <- randomForest(x = final_pca$x[, 1:20], y = training_y)
```

With our model fitted using **training_data**, we will now use this model to predict values in the **validation_data** set.

First we need to prepare the **validation_data** by converting the predictors to a matrix, subtracting the column means, and applying the rotation from the PCA performed on the **training_data**.

```
validation_x <- validation_data %>%
  select(-label) %>%
  as.matrix()
validation_y <- factor(validation_data$label)
validation_col_means <- colMeans(validation_x)
validation_x_transformed <- sweep(validation_x, 2, validation_col_means) %*% final_pca$rotation
```

With the **validation_data** prepared, we use the PCA Random Forest modeltrained on the **training_data** to predict the label values in the **validation_data** set.

```
y_hat <- predict(fit_rf, validation_x_transformed[, 1:20])
confusionMatrix(y_hat, validation_y)$overall["Accuracy"]
```

```
## Accuracy
##        1
```

This approach has once again given us ~100% prediction accuracy!

While our model is very accurate, the one potential issue we may have with this model is the runtime. When accounting for the PCA and then fitting the Random Forest model to the **training_data** we used around 2 and a half minutes on my computer (around 90 seconds for PCA and another minute for model fitting - the subsequent transformation of the **validation_data** and making the predictions used negligible runtime). The project instructions do not explicitly request that we consider runtime in our analysis, but in practice the trade-off between accuracy and runtime will often be front of mind for a machine learning project. We will therefore also consider a model that does not include PCA as an option to decrease the runtime (while hopefully keeping any reduction in prediction accuracy to a minimum).

We saw earlier that Model 5 (using the top 100 most important predictors) reduced runtimes substantially - although actually determining the identity of those top 100 predictors took some time as we needed to first fit a Random Forest model (using ntree=16). To speed up this process, we will instead fit a model using ntree=1 to the **training_data**, hoping that the reduction in runtime will make up for any reduced accuracy in our determination of the top 100 most important predictors.

```
time1 <- now()
train_rf <- randomForest(x = training_x, y = training_y, ntree=1)
top100 <- tibble(pixel = 1:784, importance = importance(train_rf)) %>%
  arrange(desc(importance)) %>%
  top_n(100, importance) %>%
  pull(pixel)
time2 <- now()
difftime(time2, time1, units = "secs")
```

```
## Time difference of 8.575174 secs
```

This was much faster than the previous calculation. We will now test the accuracy and runtime of predictions made on the **validation_data** for different values of ntree.

```
set.seed(421, sample.kind = "Rounding")
fast_ntree_accuracy <- accuracy_and_time_by_ntree(c(4, 8, 16, 32, 64), x_train = training_x[, top100], 
                           y_train = training_y, y_test = validation_y)
fast_ntree_accuracy
```

```
## # A tibble: 5 x 3
##   ntree `Model accuracy` `Calculation time (seconds)`
##   <dbl>            <dbl>                        <dbl>
## 1     4            0.931                         3.32
## 2     8            0.975                         5.57
## 3    16            0.992                         8.31
## 4    32            0.996                        14.7
## 5    64            0.997                        30.2
```

Using this approach we could achieve prediction accuracy of 0.94 with around 10 seconds of total runtime (identifying the most important predictors and then fitting the Random Forest model to those predictors), or accuracy of 0.99 in around 15 seconds. While our Model 6 prediction achieved the highest accuracy, it is helpful to have other faster options in case a particular use case requires faster turnaround and is willing to accept a slightly lower level of accuracy.

4. Conclusion

The goal of this project was to design a model that could read images of different American Sign Language gestures representing letters of the alphabet and correctly classify them. Through testing different methods of dimension reduction along with 2 popular classification algorithms, we built:

1. A model using PCA and Random Forest that could classify the images with near-perfect accuracy; and
2. A model that determined the top 100 most important classifiers and fit a Random Forest model to these classifiers to maximise calculation speed while still delivering a very high level of prediction accuracy.

For dimension reduction, we considered variance analysis, identifying the most important predictors, and PCA. We found that variance analysis did not allow us to identify any obvious predictors to remove, but using the most important predictors could improve our model accuracy and reduce runtimes, while PCA provided the best prediction accuracy overall.

We tested different types of dimension reduction by fitting kNN and Random Forest models to training data and testing the prediction accuracy and runtime when the models were used to predict labels in testing data. The prediction accuracy of the Random Forest models appeared to outperform the kNN models in our testing, and our final model that delivered the highest prediction accuracy fit a Random Forest model to training data that had undergone PCA - although we also examined the potential to use the top 100 predictors instead to fit a Random Forest model that had much shorter runtime (with only a small reduction in prediction accuracy).

Machine learning projects will often need to consider trade-offs between model accuracy and runtimes, which is why I felt it was important to explore faster models. The information presented here would allow an analyst to determine their preferred choice of prediction models that best fit their expectations around model accuracy and runtime.