Crime and Communities

Group Member 1 Name: Khang V. Tran **Group Member 1 SID:** 25181590

Group Member 2 Name: Christian Philip Hoeck **Group Member 2 SID:** 3035385003

The crime and communities dataset contains crime data from communities in the United States. The data combines socio-economic data from the 1990 US Census, law enforcement data from the 1990 US LEMAS survey, and crime data from the 1995 FBI UCR. More details can be found at https://archive.ics.uci.edu/ml/datasets/Communities+and+Crime+Unnormalized.

The dataset contains 125 columns total; $p = 124$ predictive and 1 target (ViolentCrimesPerPop). There are $n = 1994$ observations. These can be arranged into an $n \times p = 1994 \times 127$ feature matrix $\mathbf{X}$, and an $n \times 1 = 1994 \times 1$ response vector $\mathbf{y}$ (containing the observations of ViolentCrimesPerPop).

Once downloaded (from bCourses), the data can be loaded as follows.

```
library(readr)
CC <- read_csv("../data_files/crime_and_communities_data.csv")
print(dim(CC))
```

```
## [1] 1994  125
```

```
y <- CC$ViolentCrimesPerPop
X <- subset(CC, select = -c(ViolentCrimesPerPop))
```

Part 1) Dataset Exploration - Feature Creating and Engineering

In this section, you should provide a thorough exploration of the features of the dataset. Things to keep in mind in this section include:

- Which variables are categorical versus numerical?
- What are the general summary statistics of the data? How can these be visualized?
- Is the data normalized? Should it be normalized?
- Are there missing values in the data? How should these missing values be handled?
- Can the data be well-represented in fewer dimensions?

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Missing Data Processing

check if target y contains missing data

```
any(is.na(y))
```

```
## [1] FALSE
```

check if any of the features contains missing data

```
any(is.na(X))
```

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

Now, as we have detected that there exist feature(s) in X that contain NA, the next step is to find and remove features with high proportion of NA (50% or above) and remove them all together before process those that have an acceptable number of NA

```
get_na_perc <- function(x, n){
  return(sum(is.na(x))/n)
}

get_na_perc_all_features <- function(features){
  n = nrow(features)
  perc <- apply(X = features, MARGIN = 2, FUN = function(x) sum(is.na(x))/n)
  return(perc)
}

get_feature_high_na <- function(features, threshold){
  perc <- get_na_perc_all_features(features)
  return(names(perc[perc >= threshold]))
}

feature_high_na <- get_feature_high_na(X, .5)
print(feature_high_na)
```

```
##  [1] "LemasSwornFT"          "LemasSwFTPerPop"       "LemasSwFTFieldOps"
##  [4] "LemasSwFTFieldPerPop" "LemasTotalReq"         "LemasTotReqPerPop"
##  [7] "PolicReqPerOffic"      "PolicPerPop"           "RacialMatchCommPol"
## [10] "PctPolicWhite"         "PctPolicBlack"         "PctPolicHisp"
## [13] "PctPolicAsian"         "PctPolicMinor"         "OfficAssgnDrugUnits"
## [16] "NumKindsDrugsSeiz"     "PolicAveOTWorked"      "PolicCars"
## [19] "PolicOperBudg"         "LemasPctPolicOnPatr"   "LemasGangUnitDeploy"
## [22] "PolicBudgPerPop"
```

```
# X <- X %>% dplyr::select(-feature_high_na)
X <- X %>% dplyr::select(-feature_high_na)
```

Now that we have only feature with none or a low number of NA left, let's replace the mising values with the median

```
X <- X %>% mutate_all(function(x) ifelse(is.na(x), median(x, na.rm = TRUE), x))
any(is.na(X))
```

```
## [1] FALSE
```

Examine Categorical vs. Quantitative data

```
str(X)
```

By examine the structure of the data using str() (result not shown due to exessive printing) we can explore the datatype of each feature. So far, we are be able to see that there is no factor type, which means there is no string categorical feature. Neither str() nor apply(class) shows any factor. Just to be certain, I examine the documentation from the source (UC Irvine): https://archive.ics.uci.edu/ml/datasets/Communities+and+Crime+Unnormalized and did not find any character nor factor class. However, more examination is needed since there might be numerical catergorical data.

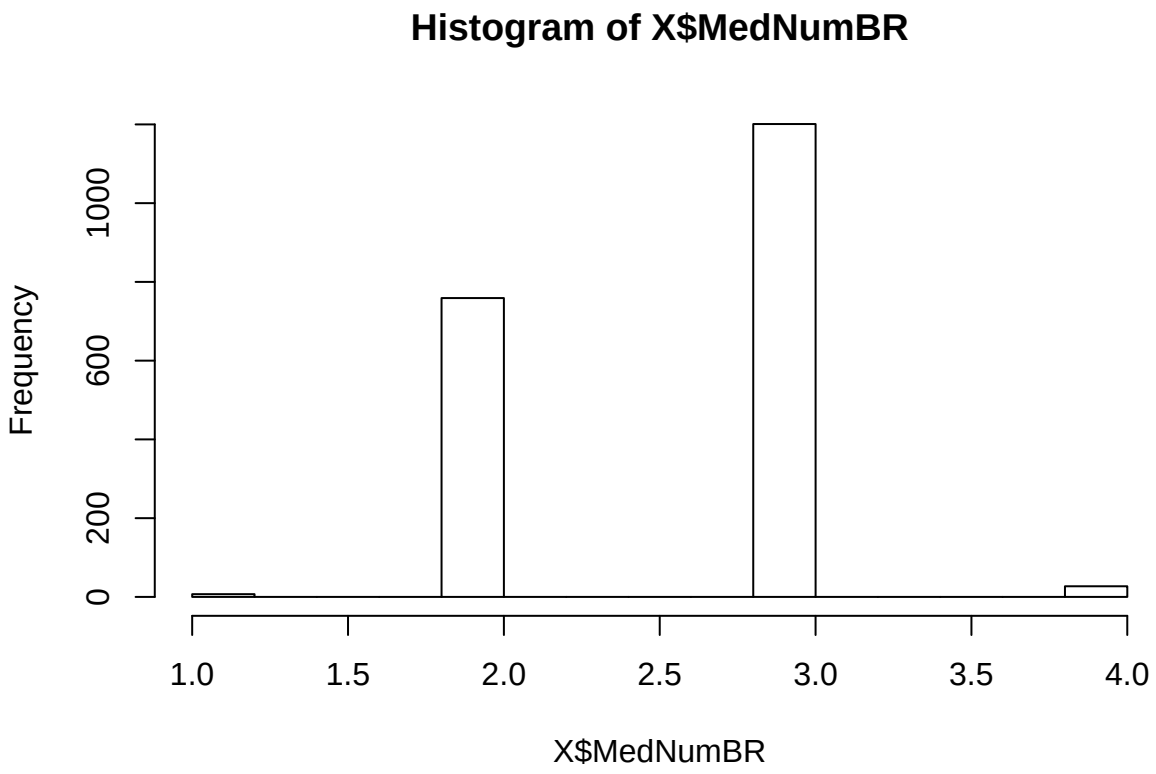
```
hist(X$householdsize)
```


```
print(head(X$householdsize, n = 20))
```

```
##  [1] 3.10 2.82 2.43 2.40 2.45 2.60 2.45 2.46 2.62 2.54 2.89 2.54 2.74 2.85
## [15] 2.62 2.67 2.60 2.60 3.34 2.36
```

By plotting histogram, we can pick out some categorical-likely feature. One of them is *householdsize*. However, when verifying with the documentation, we see that *householdsize* is actually the mean people per household (numeric - decimal). By printing out the first few values, we are able to confirm this. Therefore, it is not categorical

```
hist(X$MedNumBR)
```




```
print(head(X$MedNumBR))
```

```
## [1] 3 3 3 3 2 3
```

The next candidate is $MedNumBR$. This is, indeed, categorical and we can confirm this by once again using histogram and by printing out the first few values. However, since this is numerical, it does not matter if this is categorical. We leave it as is.

There exist in the original data the feature of states, county code, and community code, which are catergorical. However, they are not included in the given data. On the other hads, all other quantitative features in the original data are. We can say that the data set is entirely quantitative.

Summary Statistics

When speaking about crimes, there are factors that need to be taken into consideration. They are Unemployment, Children of Single Parent, Homelessness/Rent, and Poverty. Due to the high number of feature, we will only select a few feature that will give a very general idea about some of these factors

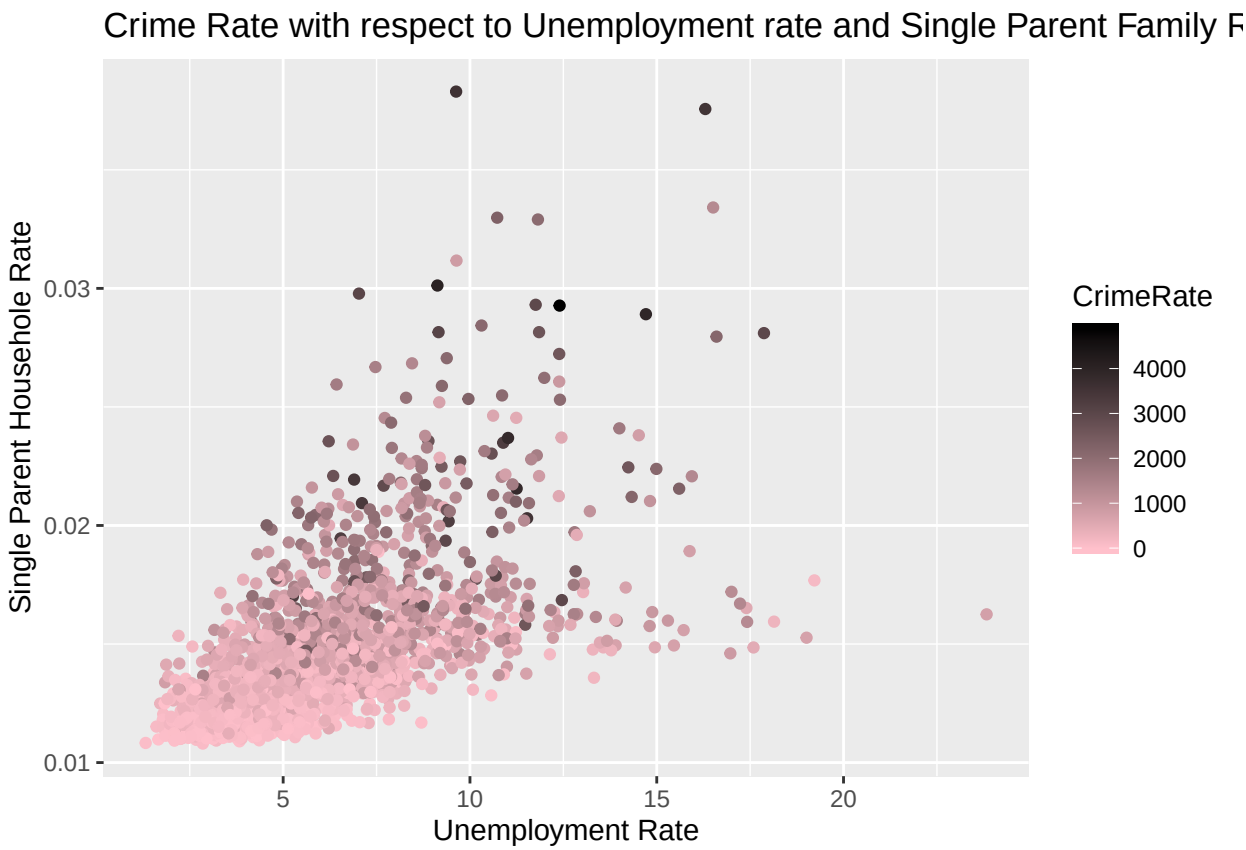
```
interesting_features <- c("PctUnemployed", "NumKidsBornNeverMar", "MedRentPctHousInc", "PctPopUnderPov")

summary_stat <- function(features, selection){
  stats <- apply(X = features[, selection], MARGIN = 2, FUN = summary)
  return(stats)
}

summary_stat(X, interesting_features)
```

```
##         PctUnemployed NumKidsBornNeverMar MedRentPctHousInc PctPopUnderPov
## Min.         1.320000               0.00          14.90000        0.64000
## 1st Qu.      4.090000             146.25          24.30000        4.69250
## Median       5.485000             361.00          26.20000        9.65000
## Mean         6.023862            2041.45          26.32803       11.79593
## 3rd Qu.      7.430000            1070.25          28.10000       17.07750
## Max.        23.830000          527557.00          35.10000       48.82000
```

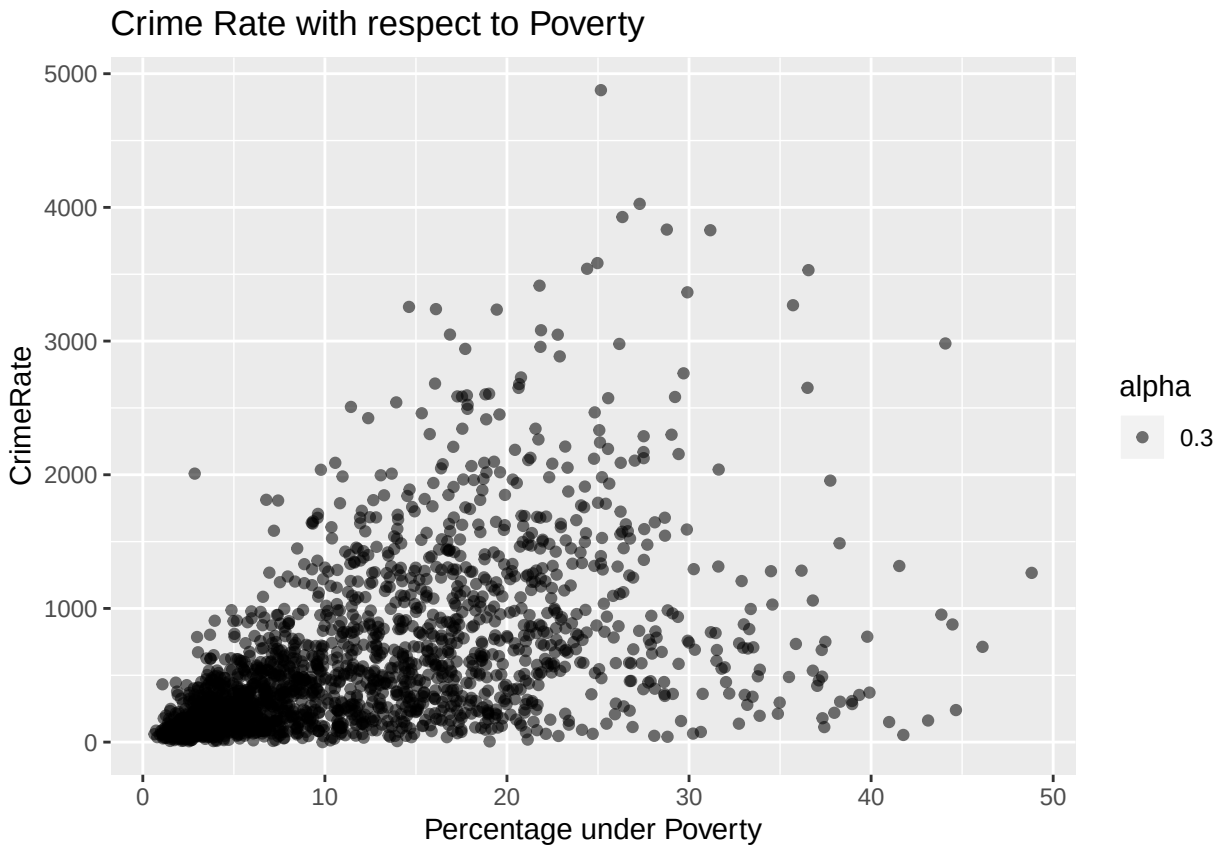
Visualization

Due to such as massive number of feature, there is no way to visualize data from every feature without dimentionality reduction. In the next coming graphs, we only examine some groups of feature that will hopefully tell us something about the data.

```
ggplot(data = temp_df) +
  geom_point(aes(x = PctUnemployed, color = CrimeRate, y = 1/PctKids2Par)) +
  scale_color_gradient(low = "pink", high = "black") +
  xlab("Unemployment Rate") +
  ylab("Single Parent Household Rate") +
  ggtitle("Crime Rate with respect to Unemployment rate and Single Parent Family Rate")
```



```
ggplot(data = temp_df) +
  geom_point(aes(x = PctPopUnderPov, y = CrimeRate, alpha = 0.3)) +
  xlab("Percentage under Poverty") +
  ggtitle("Crime Rate with respect to Poverty")
```



As we can see, there is a correlation between crime and poverty as well as crime and unemployment. However, these relartionship are not exactly linear.

Data Normalization - Scaling

After the previous step of examination, it is obvious that many features are different in nature. For example, some feautures are Percentage (PctForeignBorn, PctBornSameState). Some are counts (NumInShelters, population). Some are in US Dollars (MedRent, . . .). Each of the features have different range, scale, and unit. Such condition will affect how much each of the feature influence the predition later on .Therefore, it is highly crucial that we normalize the features.

```
X <- scale(X)
```

Dimensionality reduction - Principal Component Analysis

Apply PCA

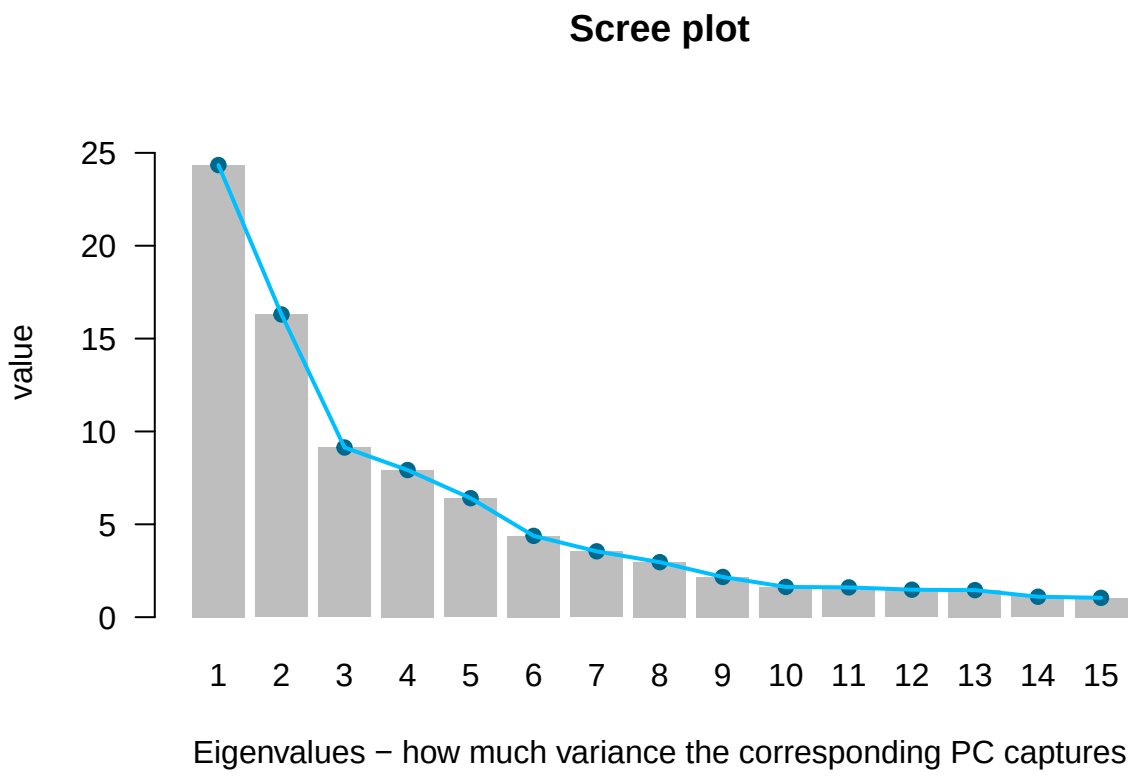
```
res.pca <- PCA(X = X,graph = F,ncp = 10)
```

Plot Screeplot for Eigenvalues. Due to the very high number of components (125), we only pick out the first 20

```
eig <- res.pca$eig
```

Visualize Eigenvalue

```
eigvalue <- eig[1:15, "eigenvalue"]

barchart <- barplot(eigvalue, las = 1, border = NA,
                    names.arg = 1:length(eigvalue),
                    ylim = c(0, 1.1 * ceiling(max(eigvalue))),
                    ylab = "value",
                    xlab = "Eigenvalues - how much variance the corresponding PC captures",
                    main = "Scree plot")

points(barchart, eigvalue, pch = 19, col = "deepskyblue4")
lines(barchart, eigvalue, lwd = 2, col = "deepskyblue")
```



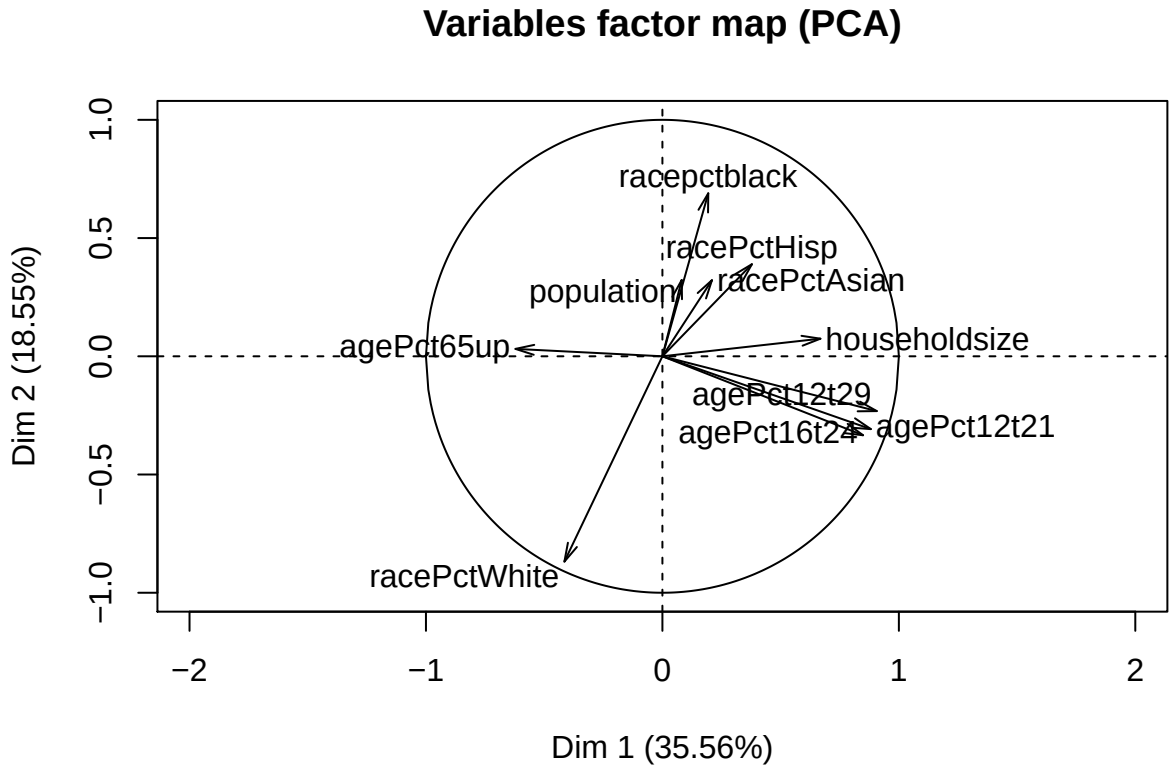
As you can see, each of the eigencalue represents the amount of variance in the dataset that was captured by the corresponding PC. Also, let's examine the eigen value result overall

```
head(eig, n = 7)
```

```
##        eigenvalue percentage of variance cumulative percentage of variance
## comp 1  24.343603              23.866278                           23.86628
## comp 2  16.298258              15.978684                           39.84496
## comp 3   9.134802               8.955689                           48.80065
## comp 4   7.922395               7.767053                           56.56770
## comp 5   6.406744               6.281121                           62.84883
## comp 6   4.377330               4.291500                           67.14033
## comp 7   3.544954               3.475445                           70.61577
```

With the given information above we can choose the number of component based on:

- Elbow method: the first 6 PCs
- Kaiser's rule $\lambda_k > 1$: the first 20 PCs
- Jollie's rule $\lambda_k > 0.7$: the first 30 PCs
- If we wish to keep the number of PCs that accumulatively capture 77% of the variance in the data, we can keep the first 10 PCs




```
# names(PCs) # "coord"   "cor"     "cos2"    "contrib"
PCs <- res.pca$ind$coord
print(head(PCs, 3))
```

```
##       Dim.1     Dim.2     Dim.3     Dim.4       Dim.5     Dim.6      Dim.7
## 1 11.139818  2.351282 -1.789758  1.807596  0.03109947  3.094648  1.2789742
## 2  6.344709 -1.673081 -1.897192  2.255860 -0.05696305  2.352737 -1.3071426
## 3  2.555890 -1.228925  1.888838 -2.156084  0.51900905 -3.296810 -0.9798917
##         Dim.8      Dim.9     Dim.10
## 1 -1.16483210  1.0360963 -0.4945979
## 2  0.41199724 -0.1417800 -1.0425311
## 3  0.07096829  0.2027569 -1.0705218
```

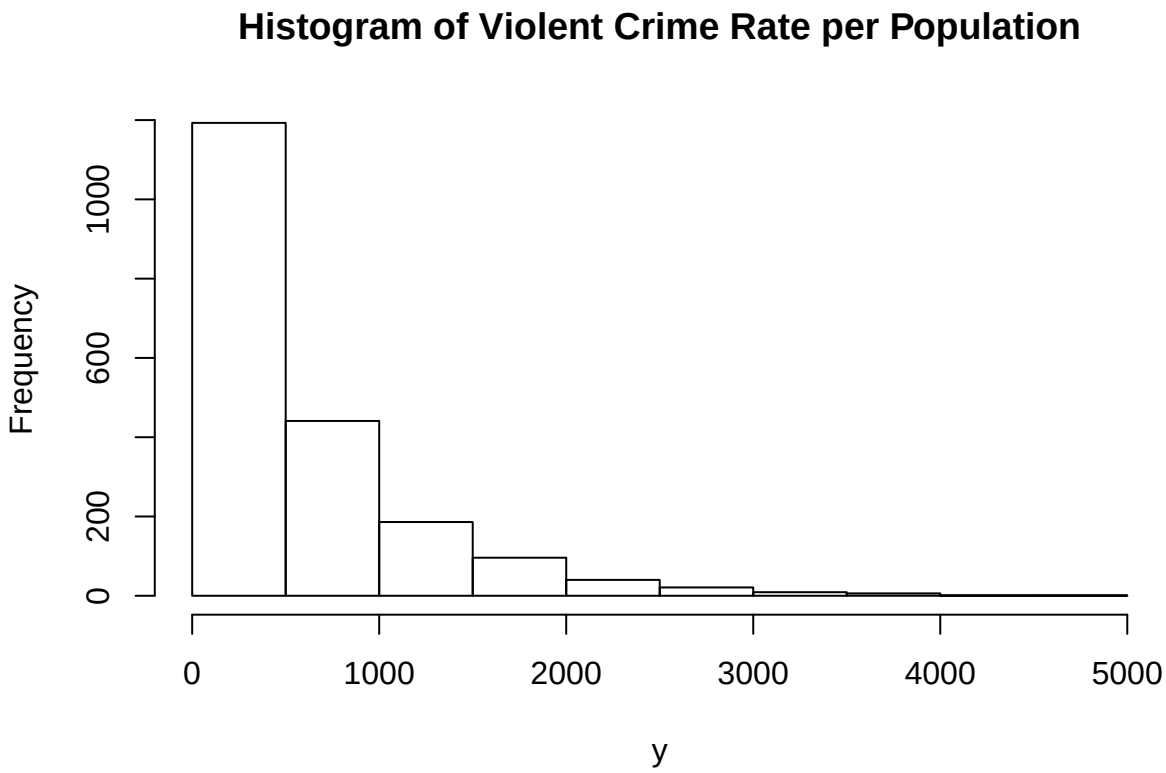
Explore the target: Crime Rate per Population

After exploring the feature, we should also take the target Crime Rate per Population into consideration. First, let's take a look at the summary and the density plot.

```
summary(y)
```

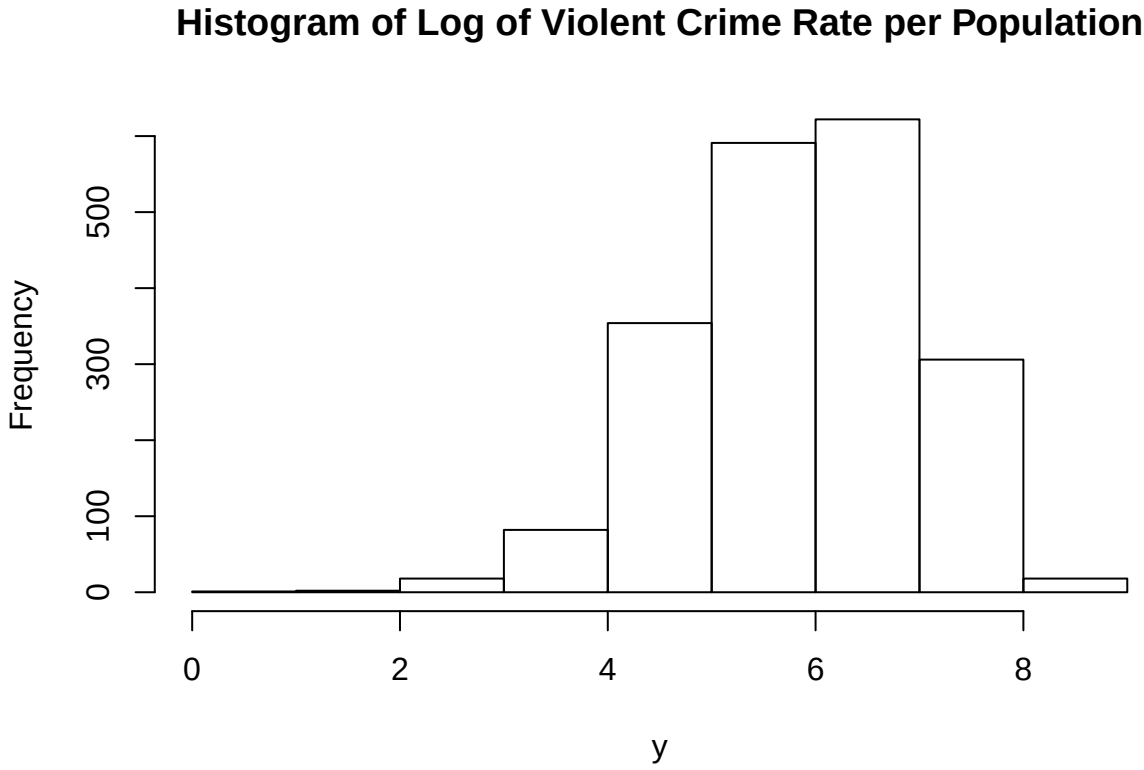
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0   161.7   374.1   589.1   794.4  4877.1
```

```r
hist(y, main = "Histogram of Violent Crime Rate per Population")
```



As we can see, Crime rate per population is heavily left skewed. In order to archieve a better distribution, let's try taking log transformation of the target

```r
y <- log(y)
y[is.infinite(y)] = 0
hist(y, main = "Histogram of Log of Violent Crime Rate per Population")
```



And now, we have a much better distribution. In the next steps, all operations will be done on this data.

Part 2) Regression task

In this section, we use the techniques learned in class to develop a model to predict ViolentCrimesPerPop using the 124 features (or some subset of them) stored in **X**. We try several different methods, and use model selection methods to determine which model is best. We keep a held-out test set to evaluate the performance of our model.

Part 2.1) Train - Test - Validation Data Randomization

In order to perform hyperparameter tuning within a model and model selecting, we will apply three-way hold-out method. Meaning, the data will be randomized into:

- Train: 60% of the data
- Validation: 20% of the data
- Test: 20% of the data

The train set will be used in two steps. The first step is to perform intial training and visualization. The next one is to perform k-folds cross validation for hyperparameter tuning. We will apply multiple regression algorithms and determine the best configuration (hyper parameters) for each of them

The Validation set will be used for model selection. Meaning, we apply all of the algorithms in the previous step with theit optimal hyperparameters, then compare the result together to determine which one perform the best

Finally, we take the winning model the regress it onto the test set. This is equivalent to proceed into production

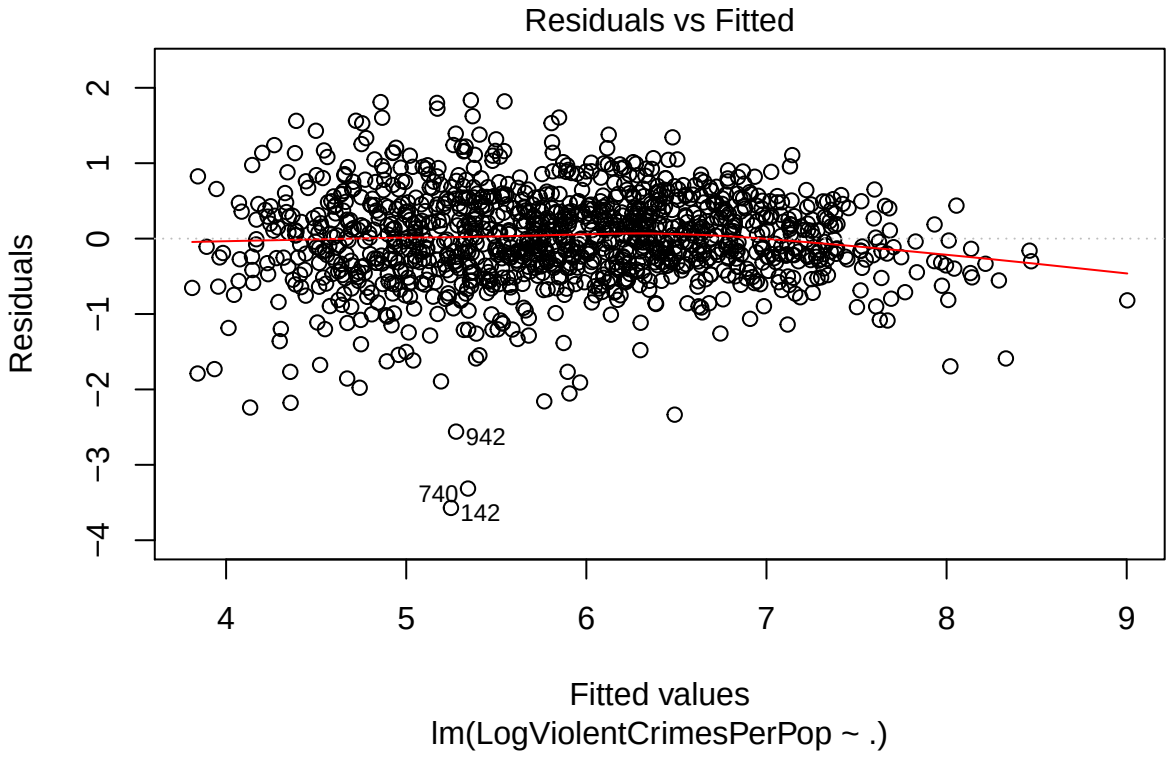
```
n <- nrow(X)
indices <- 1:n

# randomize 60% of the original data to be the train set
train_indices <- sample(x = indices, size = round(.6*n))
train_pcs <- PCs[train_indices, ]
train_features <- X[train_indices, ]
train_target <- y[train_indices]
train_data <- data.frame(cbind(train_features, train_target))
names(train_data)[ncol(train_data)] <- "LogViolentCrimesPerPop"

# randomize 20% of the original data to be the validation test
# (50% of the remaining data after sampling the train set)
validation_indices <- sample(x = indices[-train_indices],
                  size = round(.5*length(indices[-train_indices])))
validation_pcs <- PCs[validation_indices, ]
validation_features <- X[validation_indices, ]
validation_target <- y[validation_indices]

# use the rest of the data (20% of the original dataset) to be the final dataset
test_indices <- sample(x = indices[c(-train_indices,
                                     -validation_indices)])
validation_pcs <- PCs[validation_indices, ]
validation_features <- X[validation_indices, ]
validation_target <- y[validation_indices]
```

Part 2.2) Training Phase

We take this step to give an overall view of each of the regression techniques used in this project. We will fit the model, visualize the result. Some algorithms will involves regularization.

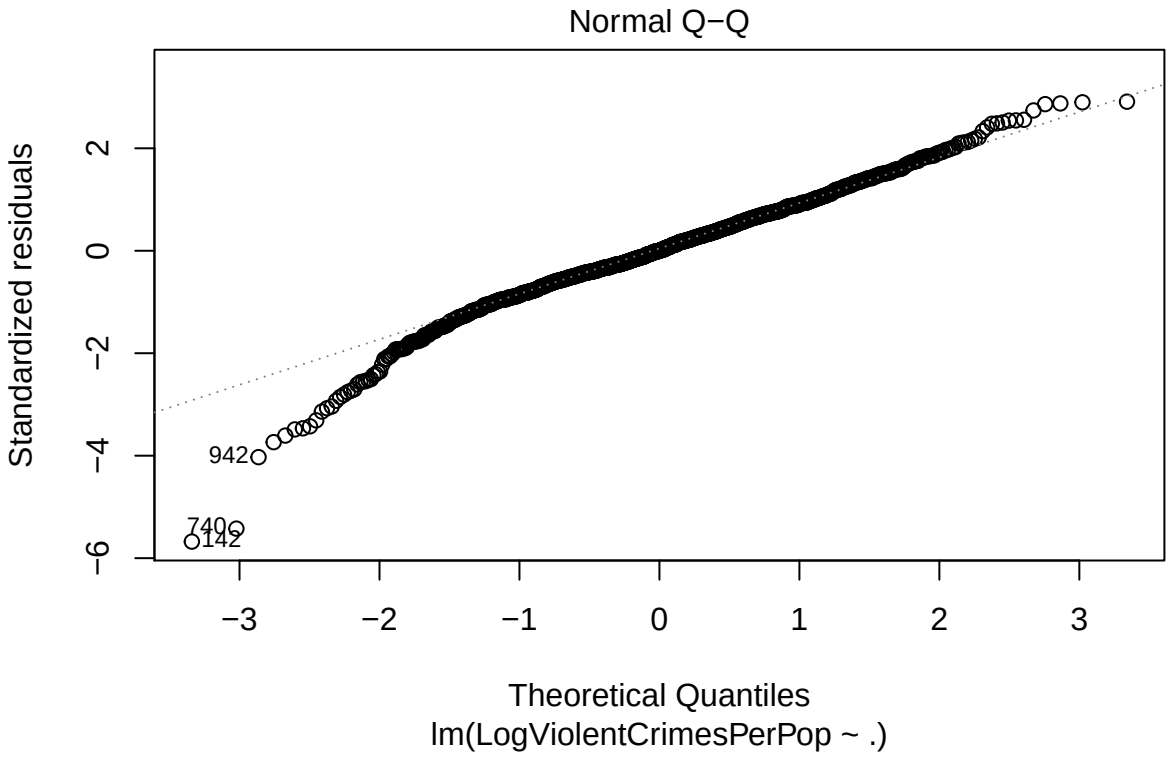
OLS Regression

```
linear_model <- lm(formula = LogViolentCrimesPerPop ~ . ,
                   data = train_data)
```



The residuals vs fitted value plot above tests whether there is a linear relationship between features, and whether there is equal varianve across the regression line. The plot aboves shows that most data points are assymetrical around the 0 line, with the right tail being less densedn compared to the left one. However, there are outliers spotted. Hence, we conclude there is a weak linear relation between variables

```
plot(linear_model, which = 2)
```



The QQ plot shown above helps us examine whether the actual distribution has the same shape as our theortical distribution. The data lies on QQ line mostly, which is consistent with our observation of gaussian from the historogram of log Crime Rate per Population. However, there are outliers exist at the two ends. We conclude the same as above

Principal Componant Regression (PCR)

```
# pcr_model <- pcr(LogViolentCrimesPerPop ~ .,
#                  m = 10,
#                  data = train_data)
#
# names(pcr_model)
```

```
# coeff <- pcr_model$coefficients
# dim(coeff)
```

Part 2.3) Hyper Parameter Tuning

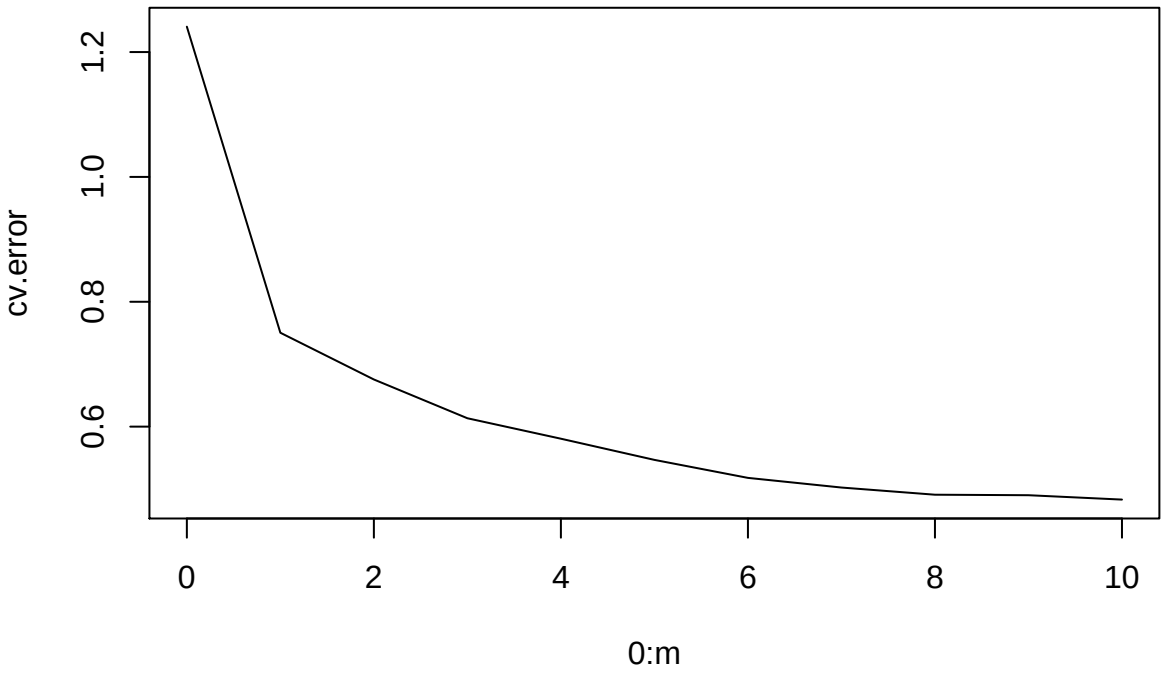
OLS Regression

Due to the nature of OLS, we get exactly one solution. There is no hyperparameter tuning

Principal Componant Analysis: tune number of component

We runs k-folds cross validation with k = 5 on hyperparameter m (number of principal componants) from 1 to 30. We will see what is the optimal number of PCs for our regression

```
set.seed(seed)
pcr_cv <- pcr.cv(train_features, train_target, k=num_folds,
       m = 10, groups=NULL, scale=F, eps=0.000001,
       plot.it = T)
```




```
(optima_num_cp <- unname(which(pcr_cv$cv.error == min(pcr_cv$cv.error))-1))
```

```
## [1] 10
```

As we see above, the number of principal component that leads to the minimal error is 10

Ridge Regression: tune Lambda term

```
set.seed(seed)
(ridge_cv <- cv.glmnet(x = train_features, y = train_target, alpha = 0,
          nfolds = num_folds, parallel = T ))
```

```
## Warning: executing %dopar% sequentially: no parallel backend registered
```

```
##
## Call:  cv.glmnet(x = train_features, y = train_target, nfolds = num_folds,      parallel = T, alpha =
##
## Measure: Mean-Squared Error
##
##     Lambda Measure      SE Nonzero
## min 0.0811  0.4547 0.02597     102
## 1se 1.4513  0.4788 0.02898     102
```

```
(optimal_lambda_ridge <- ridge_cv$lambda.min)
```

```
## [1] 0.08113913
```

Lasso Regression: tune Lambda term

```
set.seed(seed)
(lasso_cv <- cv.glmnet(x = train_features, y = train_target, alpha = 1,
          nfolds = num_folds, parallel = T ))
```

```
##
## Call:  cv.glmnet(x = train_features, y = train_target, nfolds = num_folds,      parallel = T, alpha =
```

```
##
## Measure: Mean-Squared Error
##
##      Lambda Measure      SE Nonzero
## min 0.00368  0.4496 0.02558      61
## 1se 0.06581  0.4721 0.02750      15
```

```
(optimal_lambda_lasso <- lasso_cv$lambda.min)
```

```
## [1] 0.003679561
```

K- Nearest Neighbors: tune number of neighbors

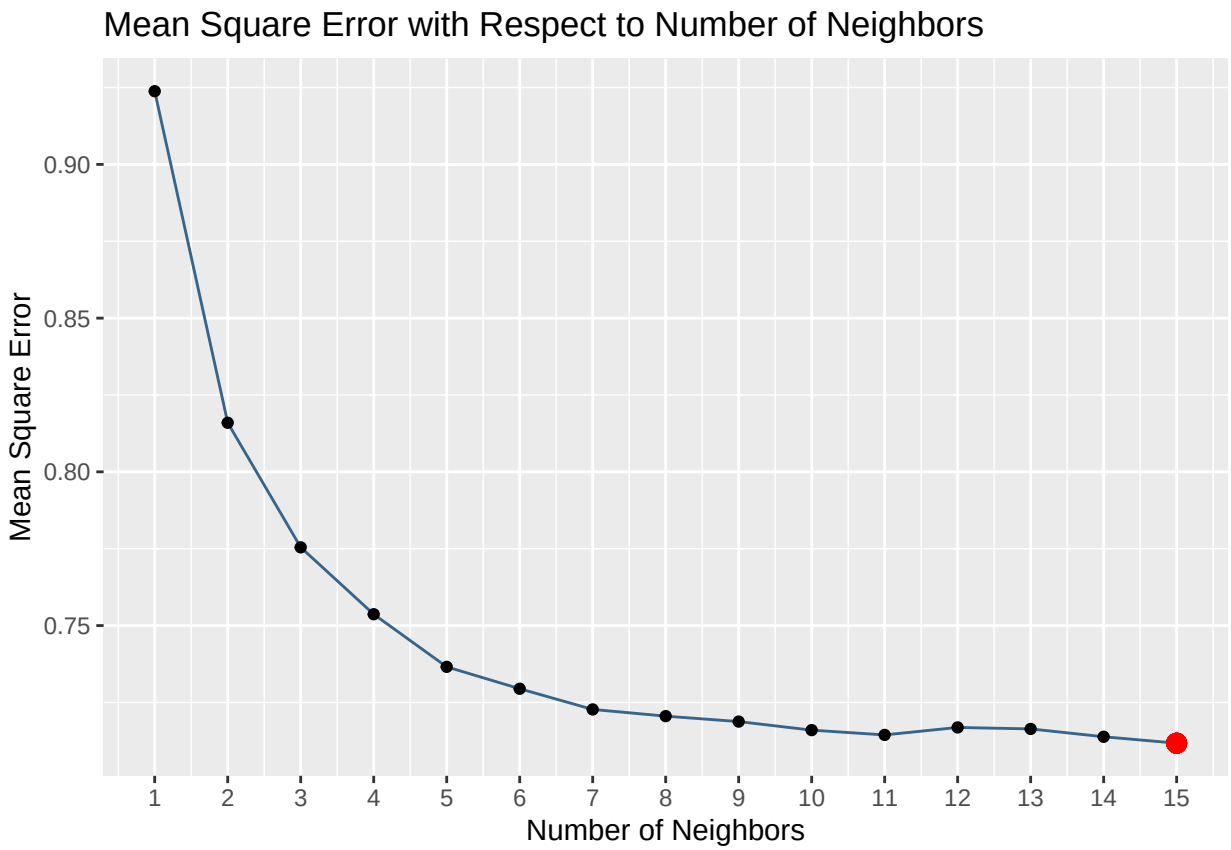
```
set.seed(seed)

train_control <- trainControl(method = "cv",
                              number = num_folds)
num_neighbors = 1:15

tune_grid = expand.grid(k = num_neighbors)

(knn_cv <- train(x = train_pcs, y = train_target, method = "knn",
      trControl = train_control, metric = "RMSE",
      tuneGrid = tune_grid))
```

```
## k-Nearest Neighbors
##
## 1196 samples
##   10 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 1076, 1076, 1076, 1077, 1076, 1077, ...
## Resampling results across tuning parameters:
##
##   k   RMSE       Rsquared   MAE
##    1  0.9238024  0.4298499  0.7032936
##    2  0.8159621  0.4979156  0.6223304
##    3  0.7754425  0.5271592  0.5872355
##    4  0.7536849  0.5482184  0.5722012
##    5  0.7365785  0.5657451  0.5554219
##    6  0.7294456  0.5732043  0.5515486
##    7  0.7227188  0.5802466  0.5458575
##    8  0.7205421  0.5825367  0.5464484
##    9  0.7187709  0.5844234  0.5432712
##   10  0.7159894  0.5877773  0.5400094
##   11  0.7144378  0.5894953  0.5372539
##   12  0.7168649  0.5865267  0.5396705
##   13  0.7163849  0.5873626  0.5391747
##   14  0.7138342  0.5912437  0.5373815
##   15  0.7117416  0.5939927  0.5360510
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was k = 15.
```

```
(temp <- knn_cv$results %>% filter(RMSE == min(RMSE)))
```

```
##    k      RMSE  Rsquared      MAE     RMSESD RsquaredSD     MAESD
## 1 15 0.7117416 0.5939927 0.536051 0.04613226 0.02963307 0.0396799
```

```
optimal_num_neighbors <- temp[, "k"]

optimal_rmse <- temp[, "RMSE"]
```



Decision Tree

Method "rpart" is only capable of tuning the cp, method "rpart2" is used for maxdepth. There is no tuning for minsplit or any of the other rpart controls. If you want to tune on different options you can write a custom model to take this into account.

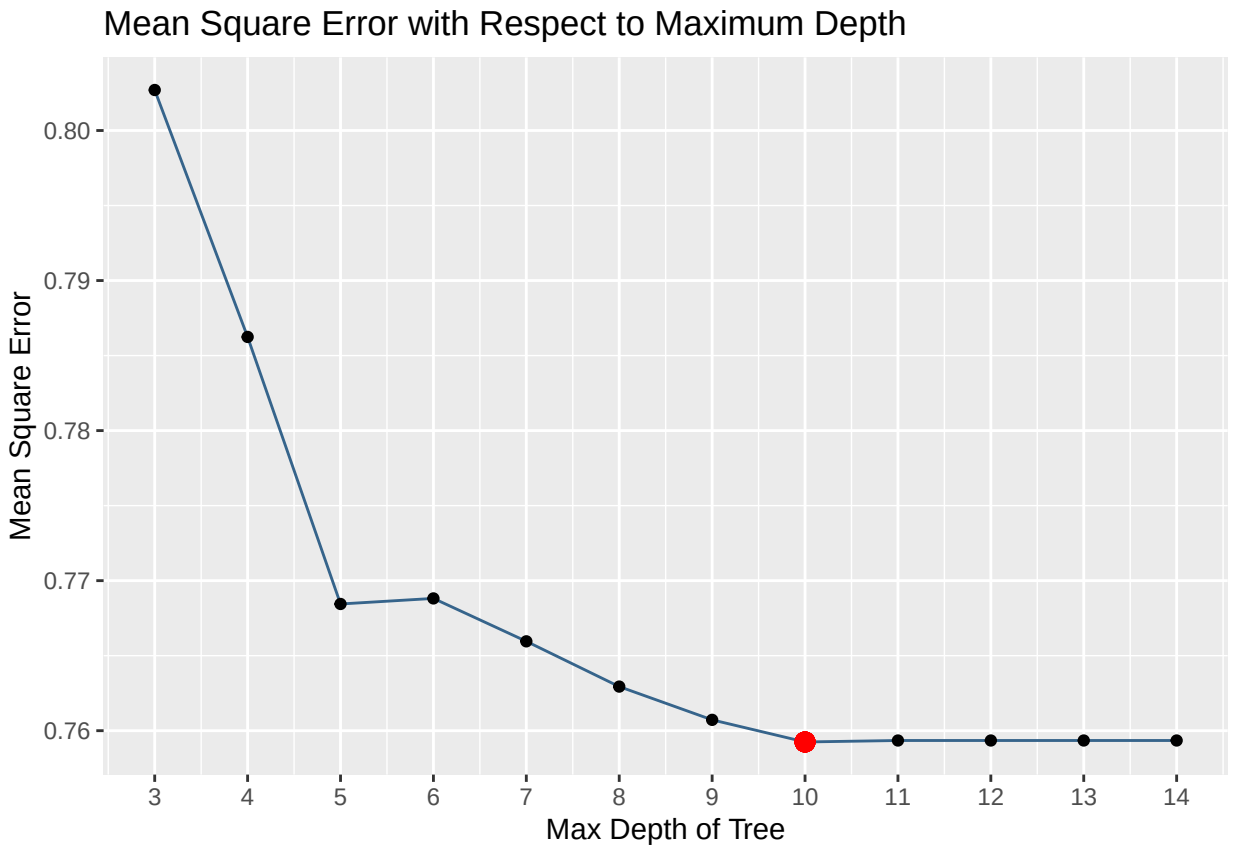
Click here for more info on how to do this. Also read this answer about how to use the rpart control within the train function.

```
train_control <- trainControl(method = "cv",
                              number = num_folds)

max_depth = 3:14

tune_grid = expand.grid(maxdepth = max_depth)

(tree_cv <- train(x = train_features, y = train_target, method = "rpart2",
      trControl = train_control, metric = "RMSE", tuneGrid = tune_grid))
```

```
## CART
```

```
##
## 1196 samples
##  102 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 1076, 1076, 1076, 1078, 1076, 1077, ...
## Resampling results across tuning parameters:
##
##   maxdepth  RMSE       Rsquared   MAE
##    3        0.8026995  0.4814167  0.6173830
##    4        0.7862416  0.5066074  0.6061447
##    5        0.7684443  0.5276239  0.5915784
##    6        0.7688156  0.5274352  0.5910638
##    7        0.7659554  0.5318177  0.5887634
##    8        0.7629391  0.5354522  0.5883072
##    9        0.7607223  0.5382707  0.5860965
##   10        0.7592477  0.5394486  0.5841080
##   11        0.7593463  0.5391912  0.5833984
##   12        0.7593463  0.5391912  0.5833984
##   13        0.7593463  0.5391912  0.5833984
##   14        0.7593463  0.5391912  0.5833984
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was maxdepth = 10.
```

```
##   maxdepth      RMSE  Rsquared      MAE     RMSESD RsquaredSD      MAESD
## 1       10 0.7592477 0.5394486 0.584108 0.08355467 0.07944174 0.05687596
```



Random Forest

```
set.seed(seed)
train_control <- trainControl(method = "cv",
                              number = num_folds)

sqrt_p <- round(sqrt(ncol(train_features)))
mtry <- (sqrt_p - 2) : (sqrt_p + 2)
tune_grid = expand.grid(mtry = mtry)

(rf_cv <- train(x = train_features, y = train_target, method = "rf",
      trControl = train_control, metric = "RMSE",
      tuneGrid = tune_grid))
```

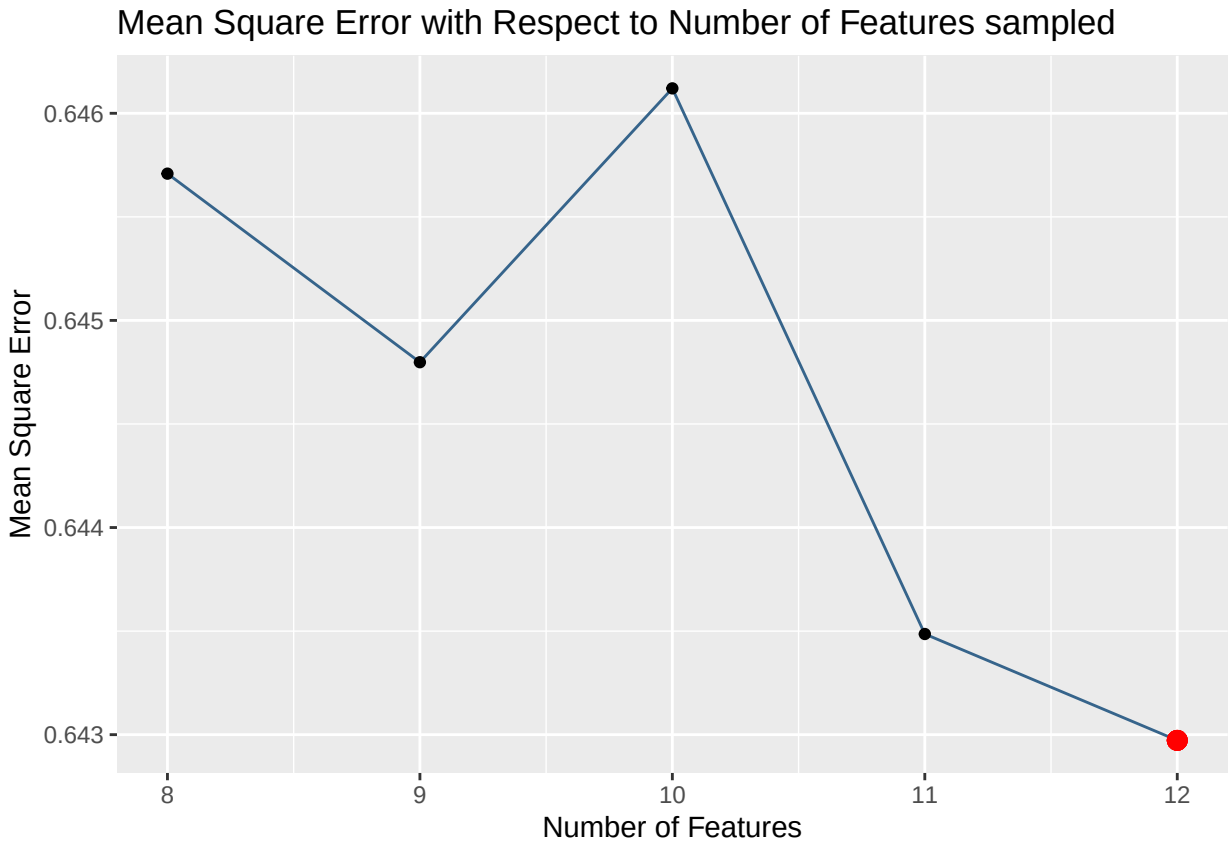
```
## Random Forest
##
## 1196 samples
##  102 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 1076, 1076, 1076, 1077, 1076, 1077, ...
## Resampling results across tuning parameters:
##
##   mtry  RMSE       Rsquared   MAE
##    8    0.6457087  0.6677062  0.4799637
```

```
##    9    0.6447982  0.6683803  0.4786929
##   10    0.6461203  0.6668724  0.4794820
##   11    0.6434860  0.6693311  0.4779322
##   12    0.6429725  0.6694850  0.4783011
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was mtry = 12.
```

```
(temp <- rf_cv$results %>% filter(RMSE == min(RMSE)))
```

```
##   mtry      RMSE Rsquared       MAE     RMSESD RsquaredSD      MAESD
## 1   12 0.6429725 0.669485 0.4783011 0.05128346  0.0396553 0.03498691
```

```
optimal_mtry<- temp[, "mtry"]

optimal_rmse <- temp[, "RMSE"]
```




```
tune_grid = expand.grid(mtry = 12)

for (num_tree in c(100, 500, 600)){
 rf_cv_temp <- train(x = train_features, y =train_target,
        method = "rf", metric = "RMSE",
        tree = num_tree,
        tuneGrid = tune_grid,
        trControl = train_control)

  print(rf_cv_temp$results %>% filter(RMSE == min(RMSE)))
```

```
}
```

```
##   mtry      RMSE  Rsquared       MAE     RMSESD RsquaredSD      MAESD
## 1   12 0.6430626 0.6680945 0.4781534 0.05226513 0.06369943 0.02874279
##   mtry      RMSE  Rsquared      MAE     RMSESD RsquaredSD      MAESD
## 1   12 0.6444158 0.6684539 0.478065 0.03368936 0.02796689 0.02526577
##   mtry     RMSE  Rsquared       MAE     RMSESD RsquaredSD     MAESD
## 1   12 0.644164 0.6690943 0.4804673 0.06944519 0.04366539 0.0411056
```

Part 2.4) Model Selection

Part 2.5) Final Model