# Model Creation: Social Vulnerability Index Training Random Forest and Geographically Weighted Random Forest Models

Thesis for a Master of Public Health, Epidemiology

Nathan Garcia-Diaz

Brown University, School of Public Health

August 13, 2024

## Statement of Purpose

The purpose of the file is to build two final models: a traditional random forest model (RF) and a geographically weighted random forest model (GWFRF). The following two sentence provide a overarching description of the two models. In a RF model, each tree in the forest is built from a different bootstrap sample of the training data, and at each node, a random subset of predictors (features) is considered for splitting, rather than the full set of predictors. A GWRF model expands on this concept by incorporating spatial information by weighting the training samples based on their geographic proximity to the prediction location. The splitting process in a RF model is determined by the mean squared error and in a GWRF is influenced by the spatial weights (i.e., weighted mean squared error), which adjust the contribution of each sample based on its geographic distance.

### Overview of Hyperparameters Definitions

In James et al 2021, Ch 8.2.2 Random Forests, James et al 2023, Ch 15.2 Definition of Random Forests and Garson 2021, Ch 5 Random Forest, the others highlight shared parameters between the RF and GWRF models:

- **Number of randomly selected predictors**: This is the number of predictors (p) considered for splitting at each node. It controls the diversity among the trees. A smaller m leads to greater diversity, while a larger m can make the trees more similar to each other.
  - for regression this defaults to $p/3$, where $p$ is the total of predictor variables
- **Number of trees**: This is the total number of decision trees in the forest (m). More trees generally lead to a more stable and accurate model, but at the cost of increased computational resources and time.
  - for the `randomForest::randomForest()`, this defaults to 500

Additionally, GWRF involves an extra tuning spatial parameters:

- **Bandwidth parameter**: This controls the influence of spatial weights, determining how quickly the weight decreases with distance. A smaller bandwidth means only very close samples have significant influence, while a larger bandwidth allows more distant samples to also contribute to the model.

### Outline of Hypereparameter Tuning Process

4 RF models will be built, and they differ based on the different hyperparameters: (1) default settings, (2) first tune $p$, and subsequently tune, then $m$ while keeping $p$ constant, (3) simultaneously tune $m$ and $p$ with a grid search, (4) tuned with Out of Bag MSE Error Rates as described by Garson 2021. Two metrics will be implemented in the tuning process: Root Mean Squared Error and Out of Bag Error Rate.

In Garson 2021, Ch 5 Random Forest, Garson teaches Random Forest Models by using `randomForest::randomForest()`, and in chapter 5.5.9 (pg. 267), he provides methods for tuning both of these parameters simultaneously using the Out of Bag MSE Error Rates. This value is a measure of the prediction error for data points that were not used in training each tree, and it can be written as OOB Error Rate $= \frac{1}{n}\Sigma_{i=1}^{N}(y_i - \hat{y}_i^{\text{OOB}})^2$ . $\hat{y}_i^{\text{OOB}}$ is the OOB prediction for the i-th observation, which is obtained by averaging the predictions from only those trees that did not include i in their bootstrap sample. To provide a high-level summary, since each tree in a Random Forest is trained on a bootstrap sample (a random sample with replacement) of the

data, approximately one-third of the data is not used for training each tree. This subset of data is referred to as the "out-of-bag" data for that tree, and this value is calculated using the data points that were not included in the bootstrap sample used to build each tree.

Georganos et al (2019) created the `package(SpatialML)`, and subsequently the tuning is made possible by the `SpatialML::grf.bw()` function. The function uses an exhaustive approach (i.e., it tests sequential nearest neighbor bandwidths within a range and with a user defined step, and returns a list of goodness of fit statistics).

4 RF models will be built, and they differ based on the different hyperparameters: (1) default settings; (2) tuned by first tuning *mtry*, with *ntrees* set to default, and subsequently tuning then *ntrees* while keeping the newly defined *mtry* constant and both methods use RMSE as the metric; (3) tuned both *mtry* and *ntrees* with an Exhaustive Grid Search and both methods use RMSE as the metric, (4) tune tuned with Out of Bag MSE Error Rates as described by Garson 2021. For each model, MAE, MSE, RMSE, and $R^2$ will be calculated and the hyperparameters of the best model will continue onto the GWRF. To provide points of comparison in the GWRF, two additional models will be created. Thus, three GWRF models will be created: (1) default *mtry* and *ntrees* with optimized *bandwidth parameter*, (2) using the previously defined best hyperparameters, (3) using the optimized *bandwidth parameter* in step one, then tuning *mtry*, with *ntrees* set to default. The method for GWRF Model 3 uses Out of Bag Error Rate as the Metric. The same model evaluation metrics will be compared in addition to calculating the residual autocorrelation.

Lastly, the feature importance plots will be generated for the final, and local feature importance plots will also be created.

## Preparation

```
### importing packages
# define desired packages
library(tidyverse)     # general data manipulation
library(knitr)         # Rmarkdown interactions
library(here)          # define top level of project folder
                           # this allows for specification of where
                           # things live in relation to the top level
library(foreach)       # parallel execution
# spatial tasks
library(tigris)        # obtain shp files
library(spdep)         # exploratory spatial data analysis
# random forest
library(caret)         # machine learning model training
library(rsample)       # splitting testing/training data
library(randomForest)  # traditional RF model
library(SpatialML)     # spatial RF model
# others
library(foreach)       # parrallel processing
library(ggpubr)        # arrange multiple graphs

### setting seed
set.seed(926)

### loading data
svi_df = read_csv(here::here("01_Data", "svi_df.csv")) %>%
  mutate(fips = as.character(fips)) %>%
  select(-...1)

### obtaining SPH files for RI tracts
tracts = tracts(state = "RI", year = 2022, cb = TRUE)

### joining data
svi_df = inner_join(tracts, svi_df, by = c("GEOID" = "fips"))
```

## Training and Testing Split

The split that will be preformed is a 70-30 split. The code creates 4 objects: `train_data`, `test_data`, `train_coords` and `test_coords`.

```
### data splits for RF models
# Create a partition for training (70% of the data)
set.seed(926)  # Set seed for reproducibility
train_index = caret::createDataPartition(svi_df$rpl_themes, p = 0.7, list = FALSE)
# Split the data into training and test sets
train_data = svi_df[train_index, ]
```

```r
test_data = svi_df[-train_index, ]

### defining coordinates for centroid and drop the geometry object type
train_coords = train_data %>%
  mutate(
    # redefines geometry to be the centroid of the polygon
    geometry = st_centroid(geometry),
    # pulls the lon and lat for the centroid
    lon = map_dbl(geometry, ~st_point_on_surface(.x)[[1]]),
    lat = map_dbl(geometry, ~st_point_on_surface(.x)[[2]])) %>%
  # removes geometry, coerce to data.frame
  st_drop_geometry() %>%
  # only select the lon and lat
  select(lon, lat)
# only obtain response and predictor variables
train_data = train_data %>%
  st_drop_geometry() %>%
  select(rpl_themes, starts_with("e_"))

test_coords = test_data %>%
  mutate(
    geometry = st_centroid(geometry),
    lon = map_dbl(geometry, ~st_point_on_surface(.x)[[1]]),
    lat = map_dbl(geometry, ~st_point_on_surface(.x)[[2]])) %>%
  st_drop_geometry() %>%
  select(lon, lat)

test_data = test_data %>%
  st_drop_geometry() %>%
  select(rpl_themes, starts_with("e_"))
```

# Traditional Random Forest Model

## Model Training and Hyperparameter Tuning

Models will be created and compared at the end of the section.

### RF Model 1 - Default Settings

The default settings for the RF model is $mtry = \text{p}/3 = 5$, and ntrees = 500, where $p$ is the number of predictors.

```r
### setting seed
set.seed(926)

# obtain the number of predictors
pred_num = svi_df %>%
  st_drop_geometry() %>%
```

```
  select(starts_with("e_")) %>%
  colnames() %>%
  length()
# determine the default number of predictors
mtry = round(pred_num / 3)

# creating the first model
rf_mod1 = randomForest::randomForest(rpl_themes ~.,
                                     data = train_data,
                                     mtry = mtry,
                                     ntree = 500,
                                     importance = TRUE)

# Print the results
print(rf_mod1)
```

```
##
## Call:
##  randomForest(formula = rpl_themes ~ ., data = train_data, mtry = mtry,      ntree = 500, i
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 0.02010956
##                     % Var explained: 76.06
```

### RF Model 2 - Sequential Processing With RMSE Metric

This model training process uses a combination of sequential processing and cross-validation. First, tuning the `mtry` parameter by using cross-validation to find the best value for each iteration. The model runs 10 times (i.e., the for loop) because given the nature of the building random forest models, the value of m within the loop changes. Therefore, preforming the function 10 times and taking the average of the most optimal mtry value it calculates and prints the average of the best mtry values. During the second step, the ntree is changing and cross-validated while *mtry* is held constant. The second model hyperparameters have been set to $mtry = 4$, and $ntrees = 550$.

```
### setting seed
set.seed(926)

### Step 1: Find the best `mtry` value
# Create an empty list to store the results
results_list = vector("list", 10)
# Loop to repeat the code 10 times
for (i in 1:10) {
  # Train the random forest model with 10-fold cross-validation
  rf_mod2 = train(rpl_themes ~ ., data = train_data, method = "rf",
                  ntree = 500, # Start with a default number of trees
                  trControl = trainControl(method = "cv", number = 10),
```

```
                  tuneGrid = expand.grid(mtry = c(3:8)))

  # print model results
  # print(rf_mod2)
  plot(rf_mod2)

  # Extract the best number of predictors (mtry) from the model
  m = rf_mod2$bestTune$mtry

  # Store the result in the list
  results_list[[i]] = m
}

mean_mtry = round(mean(unlist(results_list)))
print(mean_mtry)
```

```
## [1] 4
```

```
# Step 2: Find the best `ntree` value using cross-validation with the optimal `mtry`
store_maxtrees = list()
ntree_values = c(250, 300, 350, 400, 450, 500, 550, 600, 800, 1000) # List of `ntree` values t

for (ntree in ntree_values) {
    rf_maxtrees = train(rpl_themes ~ ., data = train_data, method = "rf",
                        tuneGrid = expand.grid(mtry = mean_mtry), # Use the fixed best `mtry`
                        trControl = trainControl(method = "cv", number = 10),
                        ntree = ntree)
    # print model results
    # print(rf_maxtrees)

    store_maxtrees[[as.character(ntree)]] <- rf_maxtrees
}

# 550 subjectively made
summary(resamples(store_maxtrees))
```

```
##
## Call:
## summary.resamples(object = resamples(store_maxtrees))
##
## Models: 250, 300, 350, 400, 450, 500, 550, 600, 800, 1000
## Number of resamples: 10
##
## MAE
##            Min.    1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 250  0.08242959 0.09939579 0.1134425 0.1168035 0.1367525 0.1477000    0
## 300  0.09925257 0.10951210 0.1166846 0.1181826 0.1283057 0.1380937    0
## 350  0.10197736 0.10353097 0.1216793 0.1188706 0.1276951 0.1448007    0
```

```
## 400  0.09378122 0.10574817 0.1173629 0.1200012 0.1264735 0.1549378    0
## 450  0.10258299 0.10560957 0.1182663 0.1177393 0.1251197 0.1424980    0
## 500  0.08784191 0.10931228 0.1206556 0.1193515 0.1338765 0.1489123    0
## 550  0.09606454 0.10565270 0.1177844 0.1183032 0.1298803 0.1402303    0
## 600  0.09763836 0.11175673 0.1210995 0.1197400 0.1321589 0.1355047    0
## 800  0.08412788 0.10955696 0.1161000 0.1168393 0.1258567 0.1482968    0
## 1000 0.08606974 0.10400699 0.1210171 0.1176695 0.1286993 0.1488905    0
##
## RMSE
##           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 250  0.1017674 0.1214815 0.1381986 0.1406572 0.1654931 0.1705079    0
## 300  0.1199442 0.1276712 0.1358007 0.1411935 0.1499061 0.1708046    0
## 350  0.1193367 0.1276486 0.1437809 0.1424804 0.1544937 0.1677310    0
## 400  0.1219718 0.1296214 0.1390872 0.1447786 0.1526881 0.1899729    0
## 450  0.1180564 0.1305948 0.1424042 0.1426324 0.1556118 0.1680307    0
## 500  0.1033448 0.1280917 0.1463723 0.1419252 0.1527854 0.1855545    0
## 550  0.1101120 0.1324290 0.1395902 0.1426412 0.1504197 0.1793398    0
## 600  0.1257977 0.1303845 0.1457564 0.1440770 0.1526889 0.1760067    0
## 800  0.1010706 0.1219275 0.1355472 0.1397896 0.1551987 0.1870169    0
## 1000 0.1095564 0.1268560 0.1436883 0.1424877 0.1598796 0.1679032    0
##
## Rsquared
##           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 250  0.6688857 0.6919905 0.7687824 0.7744250 0.8517341 0.9102818    0
## 300  0.6144158 0.7697999 0.8163390 0.7876018 0.8333925 0.8973803    0
## 350  0.6376250 0.7318562 0.7834310 0.7720673 0.8365699 0.8691612    0
## 400  0.6986293 0.7511294 0.8053687 0.7843142 0.8219672 0.8314683    0
## 450  0.6426321 0.7030926 0.7941211 0.7837714 0.8600774 0.8991179    0
## 500  0.5110343 0.7598893 0.8199929 0.7818938 0.8607187 0.8888501    0
## 550  0.6418803 0.7437289 0.7874566 0.7769717 0.7931903 0.9018571    0
## 600  0.6055090 0.7345325 0.7864024 0.7755174 0.8087727 0.8911132    0
## 800  0.6262958 0.7566033 0.8068211 0.7923817 0.8560529 0.8889661    0
## 1000 0.6579984 0.7137128 0.7949993 0.7816814 0.8259920 0.9043983    0
```

```r
rf_mod2 = randomForest::randomForest(rpl_themes ~., data = train_data,
                                     mtry = mean_mtry,
                                     ntree = 550, importance = TRUE)
```

### Model 3 - Exhaustive Grid Search with RMSE as Metric

To preform an exhaustive Grid Search, Brownlee (2020) created a custom function that preforms the grid search. This function checks every combination of *mtry* and *ntree* values determines the final values with RMSE. The final values used for the model were $mtry = 4$ and $ntree = 450$. Note that the variation between each of the combinations is minimal.

```r
### setting seed
set.seed(926)

# Define the tuned parameter
```

```r
grid = expand.grid(.mtry = c(3:8),
                   .ntree = c(250, 300, 350, 400, 450, 500, 550, 600, 800, 1000) )

ctrl = trainControl(method = "cv", number = 10)

# create custom
customRF <- list(type = "Regression", library = "randomForest", loop = NULL)
customRF$parameters <- data.frame(parameter = c("mtry", "ntree"), class = rep("numeric", 2), la
customRF$grid <- function(x, y, len = NULL, search = "grid") {}
customRF$fit <- function(x, y, wts, param, lev, last, weights, classProbs, ...) {
  randomForest(x, y, mtry = param$mtry, ntree=param$ntree, ...)
}
customRF$predict <- function(modelFit, newdata, preProc = NULL, submodels = NULL)
   predict(modelFit, newdata)
customRF$prob <- function(modelFit, newdata, preProc = NULL, submodels = NULL)
   predict(modelFit, newdata, type = "prob")
customRF$sort <- function(x) x[order(x[,1]),]
customRF$levels <- function(x) x$classes

rf_mod3 = train(rpl_themes ~ ., data = train_data, method = customRF,
                  trControl = ctrl,
                  tuneGrid = grid)
#plot(custom)
#custom$finalModel

rf_mod3 = randomForest::randomForest(rpl_themes ~., data = train_data,
                                     mtry = 4,
                                     ntree = 450, importance = TRUE)
```

### Model 4

This code snippet is designed to optimize the hyperparameters mtry and ntree in a Random Forest model and by examining the OOB MSE across these combinations, the code identifies which parameters yield the lowest error, helping to optimize the Random Forest model. Here's how the code meets this objective:

- Iterative Search for mtry: The mtry_iter function generates an iterable sequence of mtry values, starting from 1 up to the number of predictors, incremented by a step factor. This allows the code to explore different numbers of predictors used at each split in the trees.
- Specification of ntree Values: A predefined vector vntree contains different values for the number of trees to be grown in the forest. This allows the code to assess how the number of trees impacts the model performance.
- Error Calculation Across Hyperparameter Combinations: The tune function performs a grid search over the specified mtry values and the maximum number of trees specified in vntree. For each combination, the function trains a Random Forest model and calculates the OOB error rate (MSE if y is continuous).
- Parallel Processing: The foreach loop with the .dopar argument allows for parallel execution of the grid search, which speeds up the computation.

- Result Aggregation: The results are combined into a data frame, which can then be analyzed to identify the optimal combination of mtry and ntree that minimizes the OOB error rate.

This approach ensures that both hyperparameters are tuned simultaneously, leading to a more efficient model optimization process. The final model hyperparameters have been set to $m = 9$, and $ntrees = 501$. The graph below illustrates that the errors across the hyperparameters used with this method are very similar.

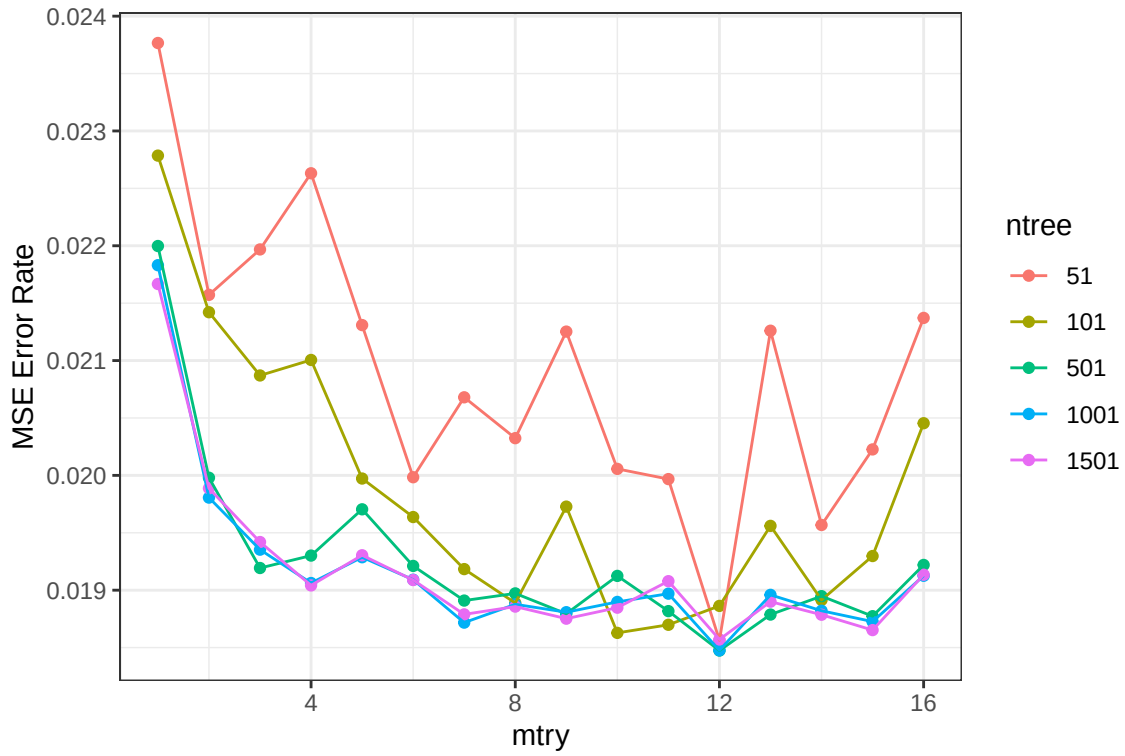
```
# create an interaction function to search over different values of mtry
mtry_iter = function(from, to, stepFactor = 1.05){
  nextEl = function(){
    if (from > to) stop('StopIteration')
    i = from
    from <<- ceiling(from * stepFactor)
    i
  }
  obj = list(nextElem = nextEl)
  class(obj) = c('abstractiter', 'iter')
  obj
}

# create a vector of ntree values of interest
vntree = c(51, 101, 501, 1001, 1501)

# specify the predictor (x) and outcome (y) object
x = svi_df %>% select(starts_with("e_")) %>% st_drop_geometry()
y = svi_df %>% pull(rpl_themes)

# Create a function to get random forest error information for different mtry values
tune = function(x, y, ntree = vntree, mtry = NULL, keep.forest = FALSE, ...) {

  # Define the combination function to aggregate results
  comb = function(a, b) {
    if (is.null(a)) return(b)
    rbind(a, b)
  }
  results = foreach(mtry = mtry_iter(1, ncol(x)), .combine = comb, .packages = 'randomForest')
    model = randomForest::randomForest(x, y, ntree = max(ntree), mtry = mtry, keep.forest = FA
    if (is.factor(y)) {
      errors = data.frame(ntree = ntree, mtry = mtry, error = model$err.rate[ntree, 1])
    } else {
      errors = data.frame(ntree = ntree, mtry = mtry, error = model$mse[ntree])
    }
    return(errors)
  }
  return(results)
}

# running the tuning
```

```r
results = tune(x,y) %>%
  mutate(MSE = error) %>%
  select(-error)

# examinations of other hyperparameters
# table
results %>%
  arrange(MSE) %>%
  head() %>%
  kable(caption = "Model 4 Preformance Metrics", digits = 4, align = c("l", "l", "c"))
```

Table 1: Model 4 Preformance Metrics

| ntree | mtry | MSE |
|---|---|---|
| 501 | 12 | 0.0185 |
| 1001 | 12 | 0.0185 |
| 51 | 12 | 0.0186 |
| 1501 | 12 | 0.0186 |
| 101 | 10 | 0.0186 |
| 1501 | 15 | 0.0187 |

```r
# plot
ggplot(results, aes(y = MSE, x = mtry,
                    color = as.factor(ntree))) +
  geom_point() +
  geom_line() +
  theme_bw() +
  labs(color = "ntree", y = "MSE Error Rate")
```




```
rf_mod4 = randomForest::randomForest(rpl_themes ~., data = train_data,
                                     mtry = 15,
                                     ntree = 501, importance = TRUE)
```

### RF Model Evaluation

Despite variations in the $m$ and *ntrees* parameters across different models, the overall prediction performance remains consistent. The relatively low MSE and RMSE values across the models indicate that the predictions are generally close to the actual values. The high R-Squared values suggest that each model explains a significant portion of the variance in the target variable. However, since model 4 produced code that is lowest MSE and RMSE, and highest R-squared value, these are the parameters that will be head contains for the GWRF.

- Mean Absolute Error (MAE):$\frac{1}{n}\Sigma_{i=1}^{n}|y_i - \hat{y}_i|$
- Mean Squared Error (MSE): $\frac{1}{n}\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2$
- Root Mean Squared Error (RMSE):$\sqrt{\frac{1}{n}\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2}$
- R-Squared Value: $\frac{\Sigma(y-\hat{y})^2}{\Sigma(y-\bar{y})^2}$

```
# Model 1
predictions1 = predict(rf_mod1, newdata = test_data)
mse1 = mean((test_data$rpl_themes - predictions1)^2)
rmse1 = sqrt(mse1)
mae1 = sum(abs(test_data$rpl_themes - predictions1))/length(predictions1)
r_squared1 = 1 - sum((test_data$rpl_themes - predictions1)^2) / sum((test_data$rpl_themes - mea

# Model 2
```

```r
predictions2 = predict(rf_mod2, newdata = test_data)
mse2 = mean((test_data$rpl_themes - predictions2)^2)
rmse2 = sqrt(mse2)
mae2 = sum(abs(test_data$rpl_themes - predictions2))/length(predictions2)
r_squared2 = 1 - sum((test_data$rpl_themes - predictions2)^2) / sum((test_data$rpl_themes - mea

# Model 3
predictions3 = predict(rf_mod3, newdata = test_data)
mse3 = mean((test_data$rpl_themes - predictions3)^2)
rmse3 = sqrt(mse3)
mae3 = sum(abs(test_data$rpl_themes - predictions3))/length(predictions3)
r_squared3 = 1 - sum((test_data$rpl_themes - predictions3)^2) / sum((test_data$rpl_themes - mea

# Model 4
predictions4 = predict(rf_mod4, newdata = test_data)
mse4 = mean((test_data$rpl_themes - predictions4)^2)
rmse4 = sqrt(mse4)
mae4 = sum(abs(test_data$rpl_themes - predictions4))/length(predictions4)
r_squared4 = 1 - sum((test_data$rpl_themes - predictions4)^2) / sum((test_data$rpl_themes - mea


# Create a data frame with the results
results_df = data.frame(
  Model = c("Model 1", "Model 2", "Model 3", "Model 4"),
  mtry = c(5,4,4,9),
  ntree = c(500,550,540,501),
  MAE = c(mae1, mae2, mae3, mae4),
  MSE = c(mse1, mse2, mse3, mse4),
  RMSE = c(rmse1, rmse2, rmse3, rmse4),
  R_Squared = c(r_squared1, r_squared2, r_squared3, r_squared4)
)

# Print the results using kable
kable(results_df, caption = "Performance Metrics for Each Model",
      digits = 3, align = c("l", "c", "c", "c", "c", "c", "c"))
```

Table 2: Performance Metrics for Each Model

| Model | mtry | ntree | MAE | MSE | RMSE | R_Squared |
|---|---|---|---|---|---|---|
| Model 1 | 5 | 500 | 0.109 | 0.020 | 0.140 | 0.767 |
| Model 2 | 4 | 550 | 0.109 | 0.020 | 0.140 | 0.767 |
| Model 3 | 4 | 540 | 0.109 | 0.020 | 0.142 | 0.759 |
| Model 4 | 9 | 501 | 0.104 | 0.018 | 0.135 | 0.783 |

## Training a Geographically Weighted Random Forest Model

### GWRF Model 1

This model has hyperparameters defined with mtry and trees by the default: *bandwith* = 50, trees = 500 and mtry = 5.

```
# testing for optimal bandwidth
temp = SpatialML::grf.bw(rpl_themes ~ e_pov150 + e_unemp + e_hburd + e_nohsdp +
                           e_uninsur + e_age65 + e_age17 + e_disabl +
                           e_sngpnt + e_limeng + e_minrty + e_munit +
                           e_mobile + e_crowd + e_noveh + e_groupq,
                         dataset = train_data,
                         kernel = "adaptive",
                         bw.min = 20,
                         bw.max = 50,
                         coords = train_coords,
                         trees = 500,
                         mtry = mtry,
                         step = 1, importance = "impurity")
best.bw_gwrf_mod1 = temp$Best.BW
best.bw_gwrf_mod1
```

```
## [1] 50
```

```
# defining the spatial model with prior model hyparameters
gwrf_mod1 = SpatialML::grf(rpl_themes ~ e_pov150 + e_unemp + e_hburd + e_nohsdp +
                           e_uninsur + e_age65 + e_age17 + e_disabl +
                           e_sngpnt + e_limeng + e_minrty + e_munit +
                           e_mobile + e_crowd + e_noveh + e_groupq,
                         dframe = train_data,
                         kernel = "adaptive",
                         coords = train_coords,
                         bw = best.bw_gwrf_mod1,
                         ntree = 500,
                         mtry = mtry,
                         importance = "impurity")
```

```
## Ranger result
##
## Call:
##  ranger(rpl_themes ~ e_pov150 + e_unemp + e_hburd + e_nohsdp +      e_uninsur + e_age65 + e
##
## Type:                             Regression
## Number of trees:                  500
## Sample size:                      174
## Number of independent variables:  16
## Mtry:                             5
## Target node size:                 5
## Variable importance mode:         impurity
```

```
## Splitrule:                          variance
## OOB prediction error (MSE):         0.02094584
## R squared (OOB):                    0.7520913
##   e_pov150    e_unemp    e_hburd   e_nohsdp  e_uninsur    e_age65    e_age17
## 1.73059616 0.28844157 0.30714937 1.55946227 0.26501948 0.72135521 0.35324871
##   e_disabl   e_sngpnt   e_limeng   e_minrty    e_munit   e_mobile    e_crowd
## 0.23946662 0.67786720 2.50338754 2.08375646 0.66017926 0.04838752 0.91237540
##    e_noveh   e_groupq
## 1.72037478 0.23538726
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -0.411990 -0.103723 -0.006432 -0.001380  0.105177  0.511217
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -5.359e-02 -5.308e-03 -9.718e-05 -3.763e-05  6.129e-03  4.137e-02
##                    Min        Max       Mean        StD
## e_pov150  0.0452443401 0.87407008 0.28943757 0.15304247
## e_unemp   0.0375479731 0.38508271 0.09680800 0.04403796
## e_hburd   0.0314999601 1.00911641 0.16713216 0.18437147
## e_nohsdp  0.0530124702 0.70025902 0.23576809 0.13946395
## e_uninsur 0.0225393735 0.36766016 0.08178669 0.05947693
## e_age65   0.0182680766 0.48217108 0.12729031 0.13306313
## e_age17   0.0161139677 0.29875735 0.07739529 0.05452568
## e_disabl  0.0205378815 0.21722369 0.08151400 0.04858315
## e_sngpnt  0.0393322502 0.34326776 0.11948708 0.05384066
## e_limeng  0.0276436736 0.99679275 0.29438893 0.19753976
## e_minrty  0.0566454665 0.90652689 0.29715676 0.18913646
## e_munit   0.0209786988 0.77326488 0.21514435 0.18959004
## e_mobile  0.0009241709 0.04493175 0.01292015 0.00986898
## e_crowd   0.0089834166 0.90074791 0.21727341 0.16694824
## e_noveh   0.0302185827 0.77422081 0.29282285 0.17089854
## e_groupq  0.0179034470 0.14001379 0.06111753 0.02580908
```

### GWRF Model 2

This model contains the hyperparameters defined in the RF building section: *bandwidth* = 44, *trees* = 501 and *mtry* = 15.

```
# testing for optimal bandwidth
temp = SpatialML::grf.bw(rpl_themes ~ e_pov150 + e_unemp + e_hburd + e_nohsdp +
                           e_uninsur + e_age65 + e_age17 + e_disabl +
                           e_sngpnt + e_limeng + e_minrty + e_munit +
                           e_mobile + e_crowd + e_noveh + e_groupq,
                         dataset = train_data,
                         kernel = "adaptive",
                         bw.min = 20,
                         bw.max = 50,
                         coords = train_coords,
                         trees = 501,
                         mtry = 15,
```

```
                              step = 1, importance.mode = "impurity")

best.bw_gwrf_mod2 = temp$Best.BW
best.bw_gwrf_mod2
```

```
## [1] 44
```

```
# defining the spatial model with prior model hyparameters
gwrf_mod2 = SpatialML::grf(rpl_themes ~ e_pov150 + e_unemp + e_hburd + e_nohsdp +
                             e_uninsur + e_age65 + e_age17 + e_disabl +
                             e_sngpnt + e_limeng + e_minrty + e_munit +
                             e_mobile + e_crowd + e_noveh + e_groupq,
                           dframe = train_data,
                           kernel = "adaptive",
                           coords = train_coords,
                           bw = best.bw_gwrf_mod2,
                           ntree = 501,
                           mtry = 9,
                           importance.mode = "impurity") # this is a ranger argument
```

```
## Ranger result
##
## Call:
##  ranger(rpl_themes ~ e_pov150 + e_unemp + e_hburd + e_nohsdp +      e_uninsur + e_age65 + e
##
## Type:                             Regression
## Number of trees:                  501
## Sample size:                      174
## Number of independent variables:  16
## Mtry:                             9
## Target node size:                 5
## Variable importance mode:         impurity
## Splitrule:                        variance
## OOB prediction error (MSE):       0.02103485
## R squared (OOB):                  0.7510378
##   e_pov150    e_unemp    e_hburd   e_nohsdp  e_uninsur    e_age65    e_age17
## 1.79776734 0.23691443 0.21896029 1.42663739 0.19593241 0.77869448 0.32884224
##   e_disabl   e_sngpnt   e_limeng   e_minrty    e_munit   e_mobile    e_crowd
## 0.22801725 0.33878506 3.15731429 2.29050627 0.65828105 0.03743623 0.72385124
##    e_noveh   e_groupq
## 1.77235048 0.17788135
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -0.466611 -0.112992 -0.005233  0.002736  0.119215  0.539698
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -5.291e-02 -4.831e-03 -2.515e-05  4.046e-04  6.549e-03  4.108e-02
##                    Min        Max       Mean         StD
## e_pov150  0.0294989492 0.95080742 0.23442161 0.167445641
## e_unemp   0.0152193745 0.43225195 0.06515446 0.042740308
```

```
## e_hburd   0.0212451652 1.45822437 0.15321704 0.233074408
## e_nohsdp  0.0235613874 0.89706984 0.20537341 0.171701910
## e_uninsur 0.0089193415 0.43130531 0.05787435 0.064228731
## e_age65   0.0079961613 0.52262670 0.11051020 0.132555905
## e_age17   0.0059905723 0.23761107 0.05578441 0.048381362
## e_disabl  0.0079282195 0.19316936 0.05630476 0.043479541
## e_sngpnt  0.0145501694 0.38129583 0.08742160 0.062874311
## e_limeng  0.0132140750 1.27325992 0.26751402 0.240552329
## e_minrty  0.0176982965 1.14250578 0.27155826 0.246774758
## e_munit   0.0079108339 1.05712242 0.21932978 0.252446507
## e_mobile  0.0004545065 0.06277922 0.00891469 0.009167348
## e_crowd   0.0045472830 1.24714341 0.19185855 0.211320860
## e_noveh   0.0141840692 0.92860698 0.26242320 0.185388972
## e_groupq  0.0094762325 0.12712572 0.04555757 0.022471690
```

```
                                                    # specification of this value
                                                    # corrected errors that previously appe
                                                    # no importance value specified
```

### GWRF Model Evaluation

The models both preform nearly identically because the hyperparameters preform nearly identically. Therefore, the model define by the previous traditional random forest model.

```
test_data = cbind(test_data, test_coords)
# Model 5
predictions5 = predict.grf(gwrf_mod1, test_data, x.var.name="lon", y.var.name="lat", local.w=1
mse5 = mean((test_data$rpl_themes - predictions5)^2)
rmse5 = sqrt(mse5)
mae5 = sum(abs(test_data$rpl_themes - predictions5))/length(predictions5)
r_squared5 = 1 - sum((test_data$rpl_themes - predictions5)^2) / sum((test_data$rpl_themes - mea

# Model 6
predictions6 = predict.grf(gwrf_mod1, test_data, x.var.name="lon", y.var.name="lat", local.w=1
mse6 = mean((test_data$rpl_themes - predictions6)^2)
rmse6 = sqrt(mse6)
mae6 = sum(abs(test_data$rpl_themes - predictions6))/length(predictions6)
r_squared6 = 1 - sum((test_data$rpl_themes - predictions6)^2) / sum((test_data$rpl_themes - mea

# Create a data frame with the results
results_df = data.frame(
  Model = c("Model 5", "Model 6"),
  bw = c(50, 44),
  mtry = c(5, 9),
  ntree = c(500,501),
  MAE = c(mae5, mae6),
  MSE = c(mse5, mse6),
  RMSE = c(rmse5, rmse6),
  R_Squared = c(r_squared5, r_squared6)
```

```
)

# Print the results using kable
kable(results_df, caption = "Performance Metrics for Each Model",
      digits = 3, align = c("l", "c", "c", "c", "c", "c", "c"))
```

Table 3: Performance Metrics for Each Model

| Model | bw | mtry | ntree | MAE | MSE | RMSE | R_Squared |
|---|---|---|---|---|---|---|---|
| Model 5 | 50 | 5 | 500 | 0.132 | 0.029 | 0.17 | 0.655 |
| Model 6 | 44 | 9 | 501 | 0.132 | 0.029 | 0.17 | 0.655 |

## Random Forest Model Comparisons

Model 4 shows a lower MAE and MSE, indicating that it generally makes smaller errors in prediction. The RMSE is also relatively low, and the high $R^2$ value (0.791) suggests that this model explains a significant portion of the variance in the SVI. Overall, Model 4 performs well and is effective in predicting SVI using the selected predictors.

Model 6 has higher MAE and MSE values, indicating that it makes larger errors on average compared to Model 4. The RMSE is also higher, and the $R^2$ is lower (0.638), suggesting that Model 6 explains less variance in the SVI. This could mean that while the Geographically Weighted Random Forest accounts for spatial autocorrelation, it may not perform as well in terms of overall prediction accuracy as the traditional Random Forest.

Model 4 (Traditional Random Forest) outperforms Model 6 (Geographically Weighted Random Forest) in terms of accuracy and explained variance. However, Model 6 is still valuable because it accounts for spatial dependencies. Model 6 might be more appropriate despite its lower overall performance metrics, particularly if the goal is to understand regional variations in the SVI. However, if the focus is purely on predictive accuracy, Model 4 appears to be the better choice.

```
test_data = test_data %>% select(-lon, -lat)

# Create a data frame with the results
results_df = data.frame(
  Model = c("Model 4", "Model 6"),
  bw = c(NA, 44),
  mtry = c(15, 9),
  ntree = c(501,501),
  MAE = c(mae4, mae6),
  MSE = c(mse4, mse6),
  RMSE = c(rmse4, mse6),
  R_Squared = c(r_squared4, r_squared6)
)

# Print the results using kable
kable(results_df, caption = "Performance Metrics for Each Model",
      digits = 3, align = c("l", "c", "c", "c", "c", "c", "c"))
```

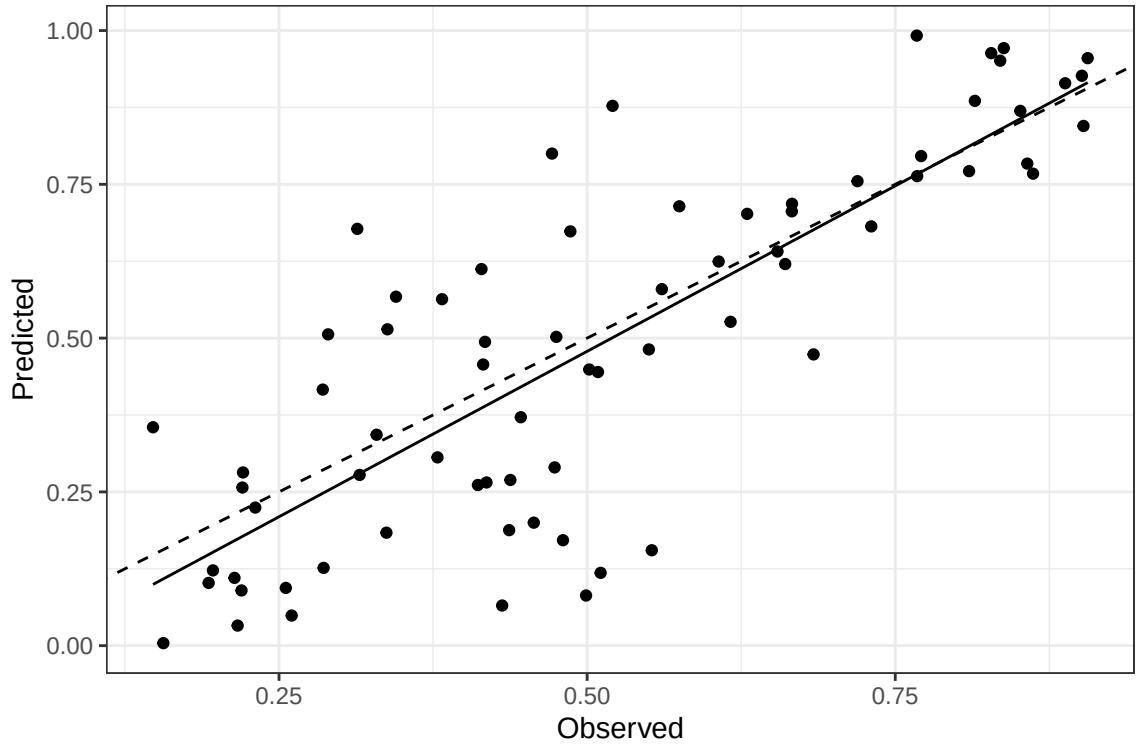
Table 4: Performance Metrics for Each Model

| Model | bw | mtry | ntree | MAE | MSE | RMSE | R_Squared |
|---|---|---|---|---|---|---|---|
| Model 4 | NA | 15 | 501 | 0.104 | 0.018 | 0.135 | 0.783 |
| Model 6 | 44 | 9 | 501 | 0.132 | 0.029 | 0.029 | 0.655 |

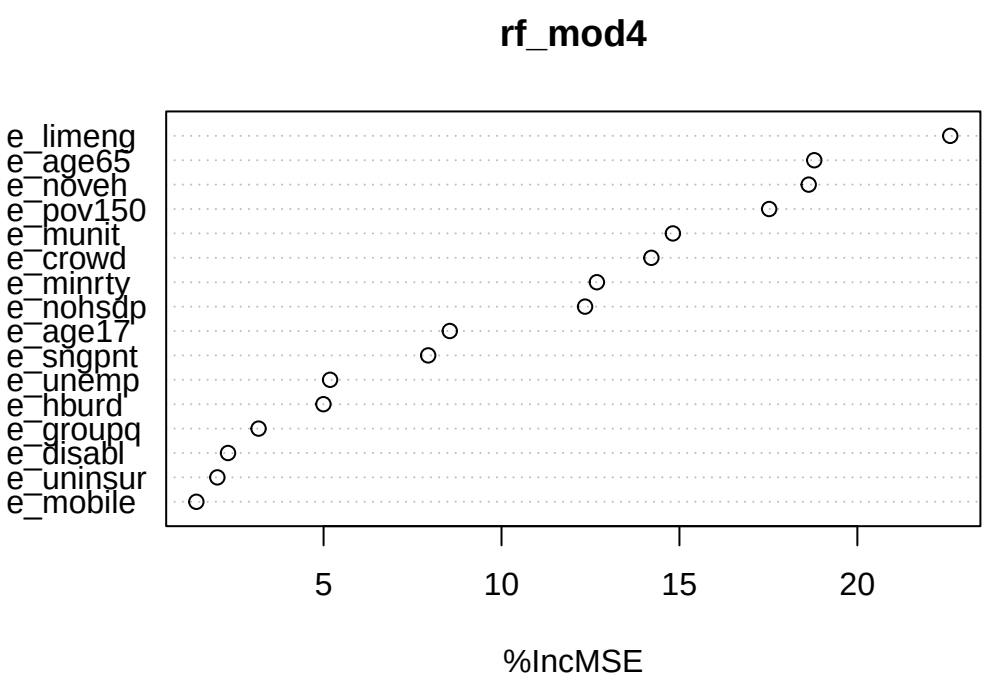
## Graphs for the Final Models

### Traditional Random Forest

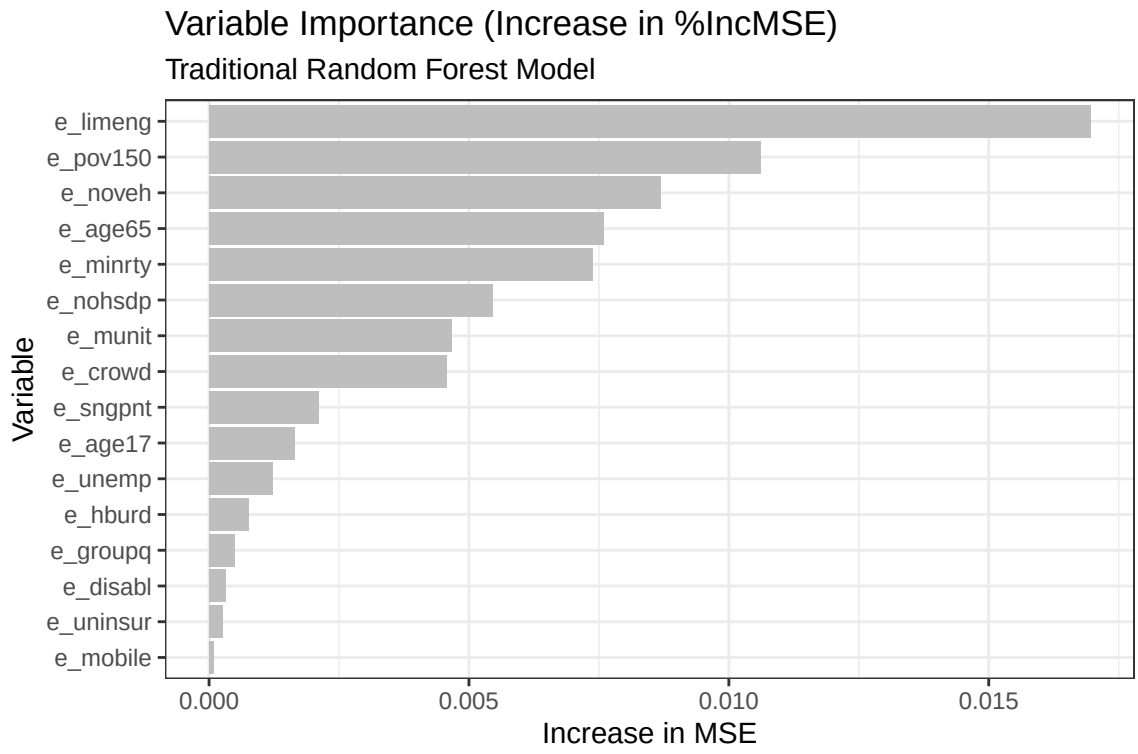
```
df = cbind(test_data, predictions5)

# Predicted 1:1 Plot
ggplot(df, aes(x =predictions5, y = rpl_themes)) +
  geom_point() +
  theme_bw() +
  geom_abline(slope=1, intercept=0,linetype="dashed",size=0.5) +
  geom_smooth(method = "lm", se = FALSE, colour="black",size=0.5) +
  labs(x="Observed", y = "Predicted")
```



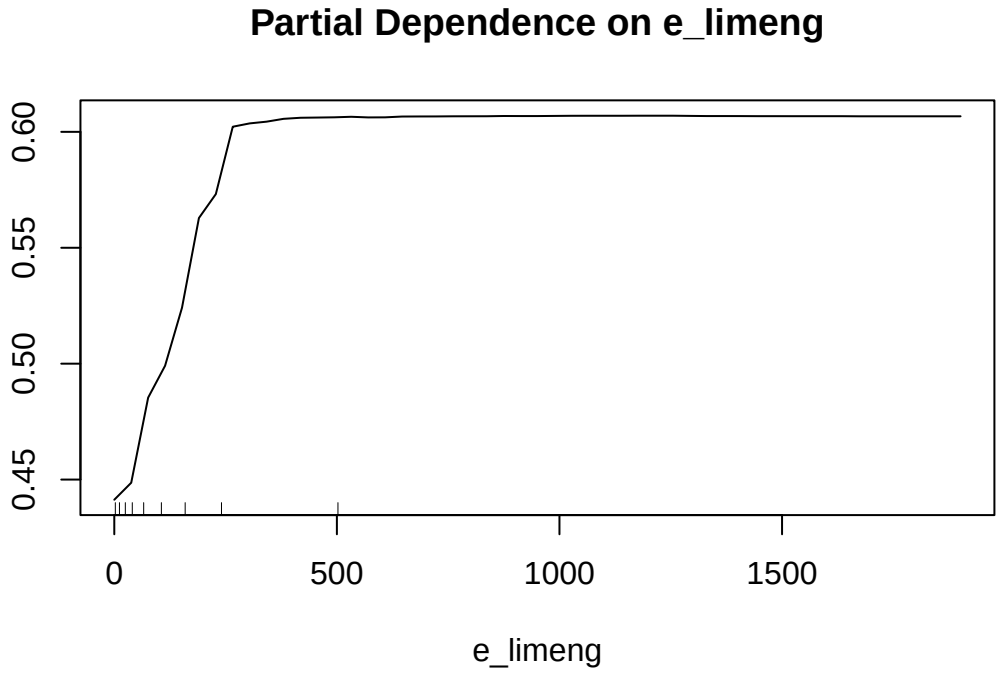

```
# Variable Importance
varImpPlot(rf_mod4, type = 1)
```



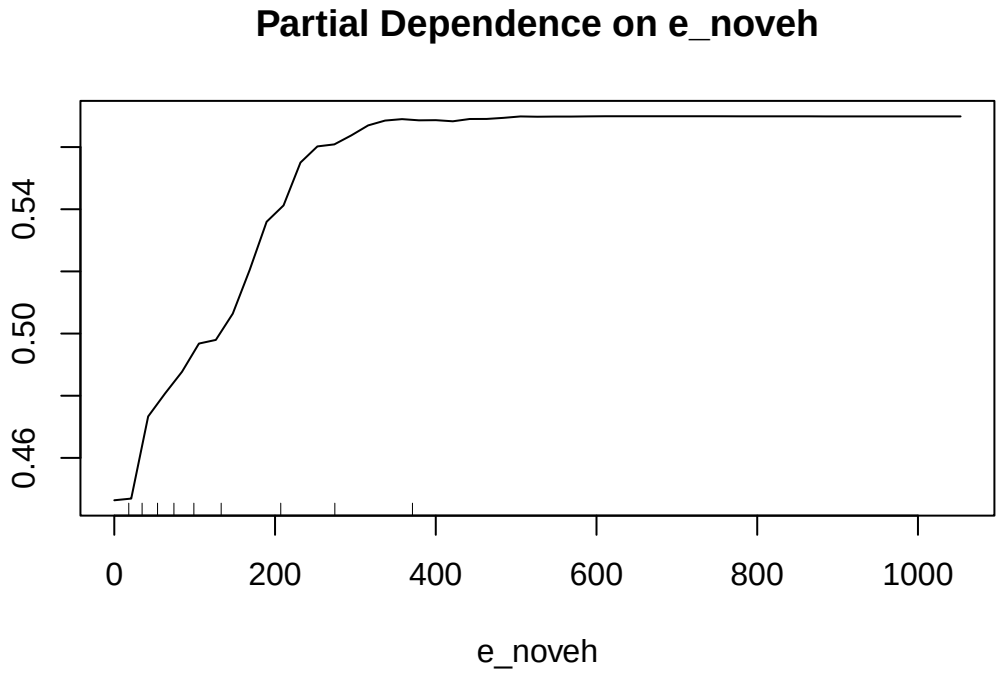

```r
temp = as.data.frame(rf_mod4$importance)
colnames(temp) = c("IncMSE", "IncNodePurity")
temp = tibble::rownames_to_column(temp, "Variable")

ggplot(temp, aes(x = fct_reorder(Variable, IncMSE), y = IncMSE)) +
  geom_bar(stat = "identity", fill = "grey") +
  coord_flip() +  # Flip coordinates for better readability
  labs(title = "Variable Importance (Increase in %IncMSE)",
       subtitle = "Traditional Random Forest Model",
       x = "Variable",
       y = "Increase in MSE") +
  theme_bw()
```




```
# Partial Dependence Plots
randomForest::partialPlot(rf_mod4, train_data, e_limeng)
```

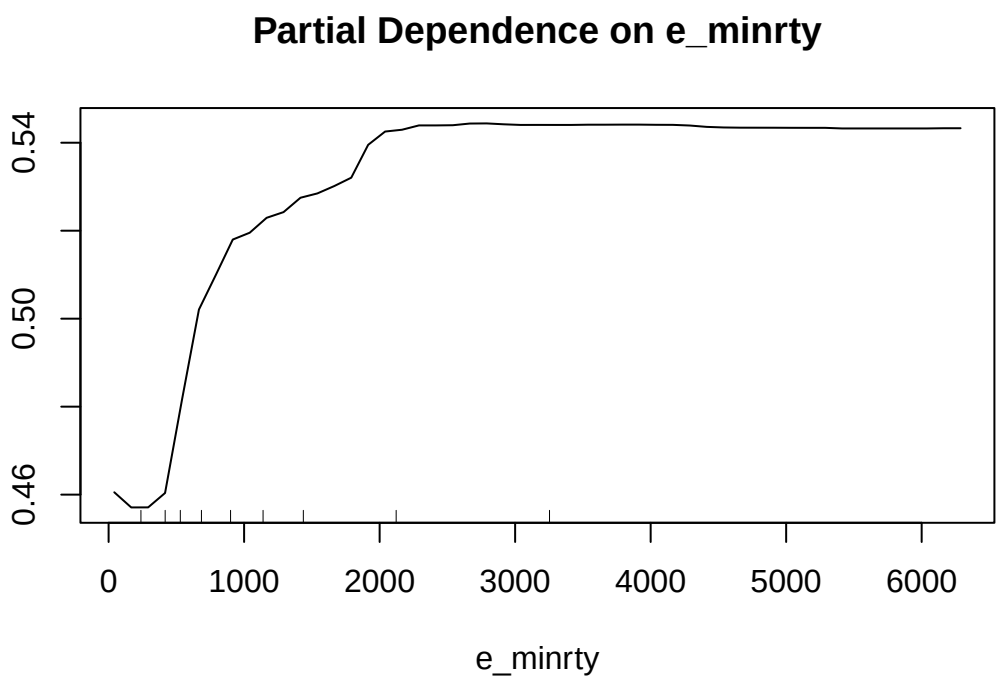


```
randomForest::partialPlot(rf_mod4, train_data, e_noveh)
```




```
randomForest::partialPlot(rf_mod4, train_data, e_age65)
```

```
randomForest::partialPlot(rf_mod4, train_data, e_minrty)
```

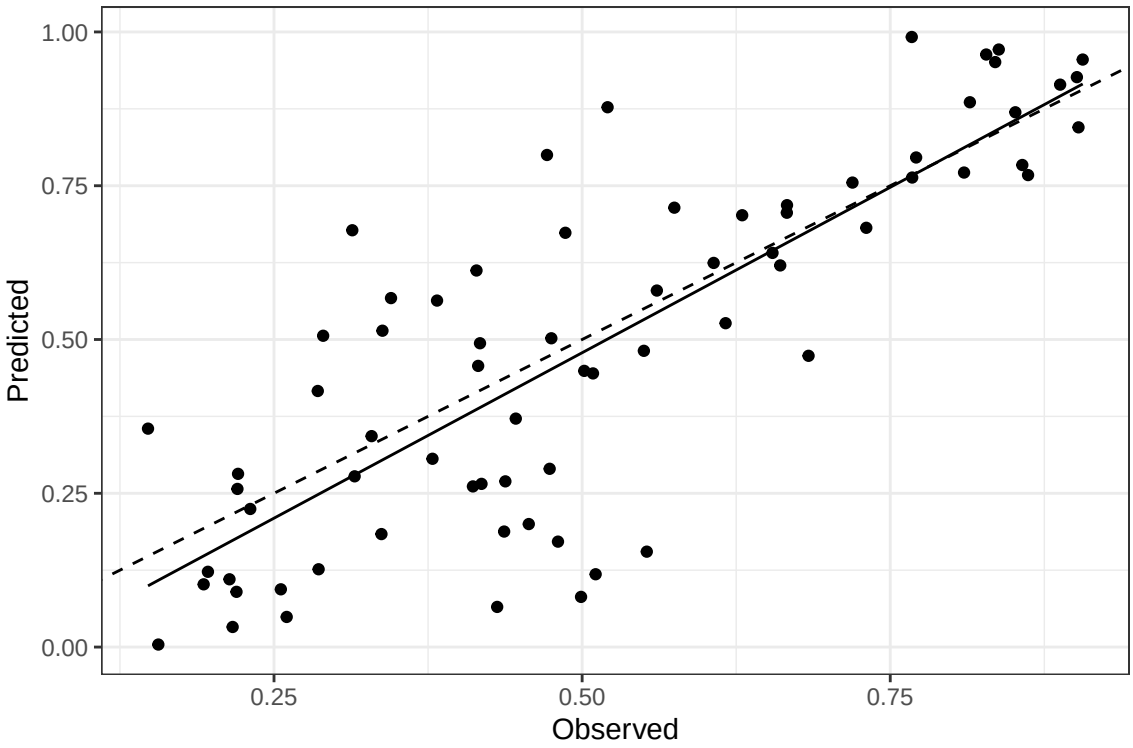



### Geographically Weighted Random Forest

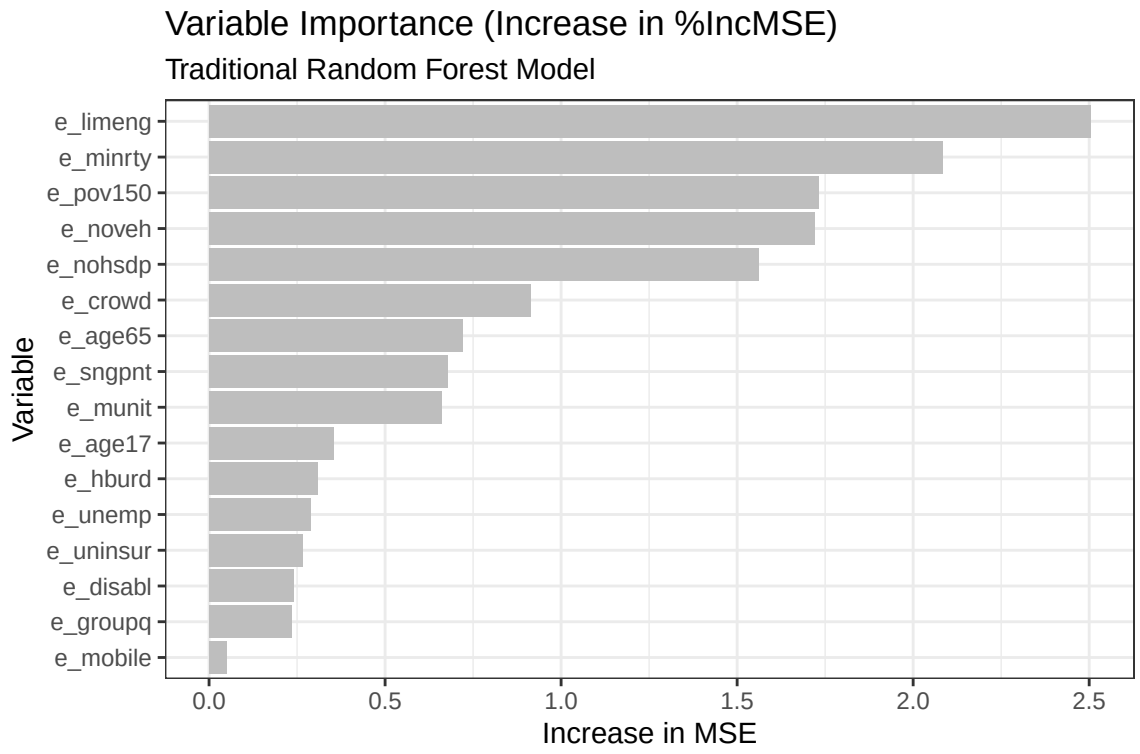
I am unsure as to how to obtain local variable importance maps. Currently I have local variable importance for 174 observations or census tracts. This aligns with the number of observations/census tracts in the training data sets, therefore, making a map of the entire state is not possible as I do not have values of for those census tracts in the testing data set.

To remedy this situation, I propose changing the analytically plan such that the model training and testing occurs on two different years of the same data (i.e., train on 2022, and test on 2021). This would allow me to obtain variable importance values across all values of the SVI.

```
df = cbind(test_data, predictions6)

# Predicted 1:1 Plot
ggplot(df, aes(x = predictions6, y = rpl_themes)) +
  geom_point() +
  theme_bw() +
  geom_abline(slope=1, intercept=0,linetype="dashed",size=0.5) +
  geom_smooth(method = "lm", se = FALSE, colour="black",size=0.5) +
  labs(x="Observed", y = "Predicted")
```




```
# Global Variable Importance
temp = as.data.frame(gwrf_mod1$Global.Model$variable.importance)
colnames(temp) = c("IncMSE")
temp = tibble::rownames_to_column(temp, "Variable")
ggplot(temp, aes(x = reorder(Variable, IncMSE), y = IncMSE)) +
  geom_bar(stat = "identity", fill = "grey") +
  coord_flip() +  # Flip coordinates for better readability
  labs(title = "Variable Importance (Increase in %IncMSE)",
       subtitle = "Traditional Random Forest Model",
       x = "Variable",
       y = "Increase in MSE") +
  theme_bw()
```




# Appendix A: Additional Codes

## Cross Validation of Performance Metrics

I am attempting to create code that preformed cross-validation when calculating performance metrics.

```
# Train the model
model1 = train(rpl_themes ~ ., data = train_data,
               method = "rf",
               trControl = trainControl(method = "cv", number = 10),
               tuneGrid = expand.grid(mtry = 5),
               ntree = 500)  # Set ntree directly

model2 = train(rpl_themes ~ ., data = train_data,
               method = "rf",
               trControl = trainControl(method = "cv", number = 10),
               tuneGrid = expand.grid(mtry = 4),
               ntree = 550)

model3 = train(rpl_themes ~ ., data = train_data,
               method = "rf",
               trControl = trainControl(method = "cv", number = 10),
               tuneGrid = expand.grid(mtry = 4),
               ntree = 450)
```

```r
model4 = train(rpl_themes ~ ., data = train_data,
               method = "rf",
               trControl = trainControl(method = "cv", number = 10),
               tuneGrid = expand.grid(mtry = 9),
               ntree = 501)

# Create a data frame with the results
results_df = data.frame(
  Model = c("Model 1", "Model 2", "Model 3", "Model 4"),
  mtry = c(5,4,4,9),
  ntree = c(500,550,450,501),
  MAE = c(model1$results$MAE, model2$results$MAE, model3$results$MAE, model4$results$MAE),
  RMSE = c(model1$results$RMSE, model2$results$RMSE, model3$results$RMSE, model4$results$RMSE)
  R_Squared = c(model1$results$Rsquared, model2$results$Rsquared, model3$results$Rsquared, mode
)

# Print the results using kable
kable(results_df, caption = "Performance Metrics for Each Model",
      digits = 3, align = c("l", "c", "c", "c", "c", "c", "c"))
```

Table 5: Performance Metrics for Each Model

| Model | mtry | ntree | MAE | RMSE | R_Squared |
|---|---|---|---|---|---|
| Model 1 | 5 | 500 | 0.119 | 0.143 | 0.759 |
| Model 2 | 4 | 550 | 0.116 | 0.140 | 0.780 |
| Model 3 | 4 | 450 | 0.119 | 0.141 | 0.779 |
| Model 4 | 9 | 501 | 0.121 | 0.145 | 0.767 |