Lecture 12

Multiple regression, ensemble methods

Introduction to Ensemble Methods in Machine Learning
Ensemble methods are powerful techniques in machine learning that combine multiple models to improve overall performance. The basic idea is that by aggregating predictions from several models, you can often achieve better generalization and accuracy than you would with a single model. Ensemble methods are especially useful in situations where individual models may suffer from high variance, bias, or may not be robust enough to capture the complexities of the data.

Three of the most popular ensemble methods are **bagging**, **boosting**, and **stacking**. Each of these techniques approaches model combination in a different way, with its own set of advantages and use cases.

Bagging (Bootstrap Aggregating)
Concept:
Bagging involves creating multiple versions of a model by training each on a different random subset of the training data, generated through bootstrapping (random sampling with replacement). Each model is then used to make predictions, and these predictions are averaged (in the case of regression) or voted upon (in the case of classification) to produce the final output.

Advantages:

- *Reduction in Variance:* By averaging multiple models, bagging reduces the variance of the prediction and helps prevent overfitting.
- *Parallelization:* Each model is trained independently, making it easy to parallelize.
- *Robustness:* Since each model sees only a subset of the data, it is less likely to overfit specific quirks in the training set.

Example Algorithms: Random Forest (a type of bagged decision tree).

Example Workflow:

- Create multiple subsets of the data by bootstrapping.
- Train a model on each subset.
- Aggregate the predictions (mean for regression, majority vote for classification).

Key Considerations:

- Bagging works well when you have models that are prone to overfitting, such as decision trees.
- It may not significantly improve models that are already low in variance.

Boosting
Concept:
Boosting is an iterative technique where models are trained sequentially. Each new model tries to correct the errors made by the previous models. The final model is a weighted sum of the individual models, where the weights depend on the accuracy of each model. Unlike bagging, which treats all models equally, boosting gives more weight to models that perform better on the data.

Advantages:

- *Reduction in Bias:* Boosting can significantly reduce bias and is particularly useful for creating strong learners from weak ones.
- *Handling Imbalanced Data:* Boosting can be effective in handling imbalanced datasets by focusing on harder-to-classify instances.
- *Customization:* The sequential nature allows for flexible adjustments in how errors are weighted and corrected.

Example Algorithms: AdaBoost, Gradient Boosting Machines (GBM), XGBoost, LightGBM, CatBoost.

Example Workflow:
- Train the first model on the dataset.
- Evaluate the model and identify the misclassified instances.
- Train a new model, giving more weight to the instances that were misclassified in the previous step.
- Repeat this process for a specified number of iterations or until the model reaches a satisfactory level of performance.
- Aggregate the predictions by weighting them according to each model's accuracy.

Key Considerations:
- Boosting can be sensitive to noisy data and outliers because it focuses on difficult-to-classify examples.
- Tuning is more complex than bagging, as it involves setting learning rates, number of iterations, and other hyperparameters.

Stacking (Stacked Generalization)
Concept:
Stacking involves training multiple different types of models and then using another model, often called a meta-learner or stacking model, to learn how to best combine their predictions. Unlike bagging and boosting, stacking typically involves using a diverse set of models (e.g., decision trees, SVMs, neural networks) rather than variations of a single model type.

Advantages:
- *Leverage Diversity:* By combining models of different types, stacking can exploit the strengths of each model.
- *Customizability:* The choice of base models and the meta-learner can be highly customized to the problem at hand.
- *Flexibility:* Stacking allows you to incorporate a wide variety of models, including those that might not perform well individually.

Example Workflow:
- Train multiple different models (e.g., a linear regression, a random forest, and an SVM) on the training data.
- Generate predictions from each model on a validation dataset.
- Train a meta-learner model on the validation predictions to learn how to best combine the base models' outputs.
- Use the meta-learner to make final predictions on new data.

Key Considerations:

- Stacking requires careful validation, usually done with k-fold cross-validation, to prevent overfitting.
- The choice of meta-learner is crucial; common choices are linear regression or logistic regression, but more complex models can also be used.

Comparison and Summary

- *Bagging* is typically used when you have high-variance models and want to reduce overfitting, such as with decision trees.
- *Boosting* is used when you want to correct for model bias and improve the overall accuracy by focusing on harder-to-predict instances.
- *Stacking* allows for the combination of multiple different models, which can be advantageous if you have access to diverse models that may capture different aspects of the data.

Each of these methods has its place in machine learning, and the choice of which to use depends on the specific problem, the data, and the types of models you are working with.

Before we break this down, let’s look at some examples using packages.

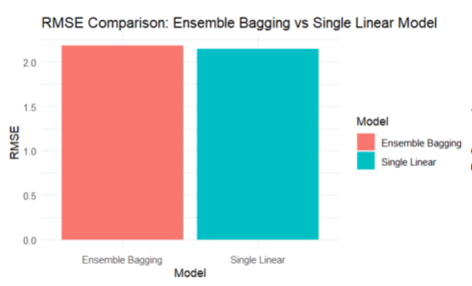
```
# Load necessary libraries
install.packages("caretEnsemble") #install other packages as needed
library(caretEnsemble)
library(randomForest)  # For bagging with Random Forest
library(xgboost)       # For boosting with XGBoost
library(caret)         # For stacking with the 'caretEnsemble' package

# Example dataset
data(mtcars)

# Bagging with Random Forest
rf_model <- randomForest(mpg ~ ., data = mtcars, ntree = 100)
rf_preds <- predict(rf_model, mtcars)

# Boosting with XGBoost
dtrain <- xgb.DMatrix(data = as.matrix(mtcars[, -1]), label = mtcars$mpg)
params <- list(objective = "reg:squarederror", max_depth = 3, eta = 0.1)
xgb_model <- xgboost(params = params, data = dtrain, nrounds = 100)
xgb_preds <- predict(xgb_model, as.matrix(mtcars[, -1]))

# Stacking with caretEnsemble
model_list <- caretList(
  mpg ~ ., data = mtcars,
  trControl = trainControl(method = "cv", number = 5),
  methodList = c("rf", "lm")
)
stack_model <- caretStack(model_list, method = "glm")
stack_preds <- predict(stack_model, mtcars)

# Compare performance
```

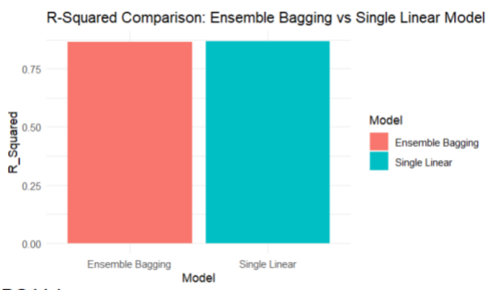
```
cat("Bagging RMSE:", sqrt(mean((rf_preds - mtcars$mpg)^2)), "\n")
cat("Boosting RMSE:", sqrt(mean((xgb_preds - mtcars$mpg)^2)), "\n")
cat("Stacking RMSE:", sqrt(mean((stack_preds - mtcars$mpg)^2)), "\n")
```

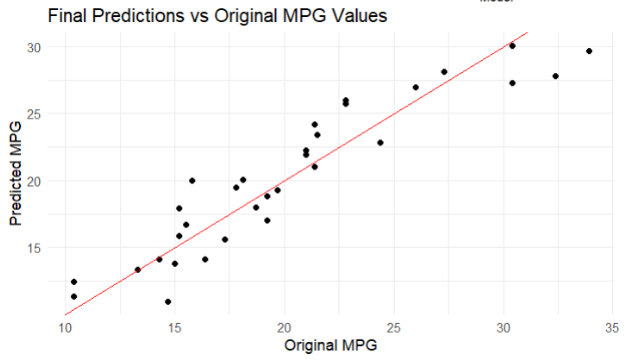
In order to dig into these methods more, we'll create a bagging ensemble of multiple linear regression models. The basic process is to create a bootstrap sample, then make a prediction model, make predictions, and then aggregate those predictions. I've also included some visualizations so that we can see the comparisons.

```
#bagging with regression
# Load the dataset
data(mtcars)

# Number of bootstrap samples
n_bootstraps <- 100
n <- nrow(mtcars)
set.seed(123)  # For reproducibility

# Function to fit a linear model on a bootstrap sample
fit_bootstrap_model <- function(data, indices) {
  bootstrap_sample <- data[indices, ]
  lm(mpg ~ ., data = bootstrap_sample)
}

# Store the models and predictions
models <- list()
predictions <- matrix(NA, nrow = n, ncol = n_bootstraps)

# Perform bagging
for (i in 1:n_bootstraps) {
  # Generate bootstrap sample
  indices <- sample(1:n, size = n, replace = TRUE)

  # Fit model on bootstrap sample
  model <- fit_bootstrap_model(mtcars, indices)
  models[[i]] <- model

  # Make predictions on the original data
  predictions[, i] <- predict(model, newdata = mtcars)
}

# Aggregate predictions by averaging
final_predictions <- rowMeans(predictions)

# Calculate performance metrics
mse <- mean((final_predictions - mtcars$mpg)^2)
rmse <- sqrt(mse)
```

```r
r_squared <- 1 - sum((final_predictions - mtcars$mpg)^2) / sum((mtcars$mpg -
mean(mtcars$mpg))^2)

# Output the results
cat("Ensemble Bagging Model Performance Metrics:\n")
cat("RMSE:", rmse, "\n")
cat("R-Squared:", r_squared, "\n")

# Optional: Compare with a single linear model
single_model <- lm(mpg ~ ., data = mtcars)
single_model_predictions <- predict(single_model, newdata = mtcars)
single_model_rmse <- sqrt(mean((single_model_predictions - mtcars$mpg)^2))
single_model_r_squared <- summary(single_model)$r.squared

cat("\nSingle Linear Model Performance Metrics:\n")
cat("RMSE:", single_model_rmse, "\n")
cat("R-Squared:", single_model_r_squared, "\n")

# Visualization: Comparison Metrics
metrics <- data.frame(
  Model = c("Ensemble Bagging", "Single Linear"),
  RMSE = c(rmse, single_model_rmse),
  R_Squared = c(r_squared, single_model_r_squared)
)

# Plot the comparison of RMSE and R-Squared
library(ggplot2)

# RMSE Comparison
ggplot(metrics, aes(x = Model, y = RMSE, fill = Model)) +
  geom_bar(stat = "identity") +
  theme_minimal() +
  ggtitle("RMSE Comparison: Ensemble Bagging vs Single Linear Model")

# R-Squared Comparison
ggplot(metrics, aes(x = Model, y = R_Squared, fill = Model)) +
  geom_bar(stat = "identity") +
  theme_minimal() +
  ggtitle("R-Squared Comparison: Ensemble Bagging vs Single Linear Model")

# Visualization: Final Predictions vs Original Points
ggplot(mtcars, aes(x = mpg, y = final_predictions)) +
  geom_point() +
  geom_abline(intercept = 0, slope = 1, col = "red") +
  theme_minimal() +
  ggtitle("Final Predictions vs Original MPG Values") +
  xlab("Original MPG") +
  ylab("Predicted MPG")
```







This method is particularly useful when dealing with high-variance models, though in this case, linear models tend to have lower variance compared to more complex models. However, the principles apply broadly across different types of models.

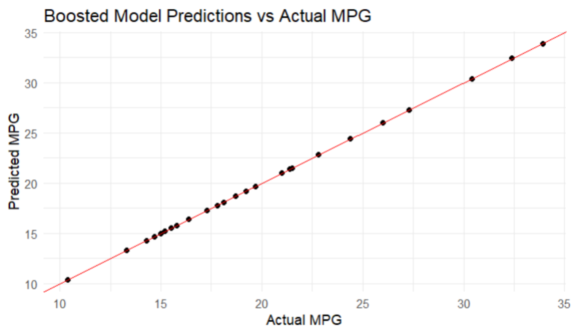
We'll look at bagging again later in another lecture with decision trees and building a random forest algorithm.

Let's now look at boosting, using the same dataset and regression models. The general approach: Start with a simple prediction, which is the mean of the target variable (mpg). Calculate the initial residuals as the difference between the actual values and the initial predictions. In each iteration, a new linear model is trained on the residuals from the previous model. The predictions are updated by adding the scaled predictions of the new model to the previous predictions. The residuals are recalculated after each iteration. The learning rate controls how much each model contributes to the final prediction. A smaller learning rate requires more iterations.

```
# Load the necessary dataset
data(mtcars)

# Set the number of boosting iterations
n_boosting_iterations <- 100
learning_rate <- 0.1  # Controls the contribution of each model

# Initialize the model predictions (starting with the mean of the target variable)
initial_prediction <- mean(mtcars$mpg)
predictions <- rep(initial_prediction, nrow(mtcars))

# Initialize the residuals (difference between actual values and predictions)
residuals <- mtcars$mpg - predictions
```

```r
# Store all models
models <- list()

# Boosting iterations
for (i in 1:n_boosting_iterations) {
  # Fit a linear model to the residuals
  model <- lm(residuals ~ ., data = mtcars)
  models[[i]] <- model

  # Predict the residuals using the new model
  model_predictions <- predict(model, newdata = mtcars)

  # Update the predictions with the learning rate
  predictions <- predictions + learning_rate * model_predictions

  # Update the residuals
  residuals <- mtcars$mpg - predictions
}

# Calculate final performance metrics
mse <- mean((predictions - mtcars$mpg)^2)
rmse <- sqrt(mse)
r_squared <- 1 - sum((predictions - mtcars$mpg)^2) / sum((mtcars$mpg -
mean(mtcars$mpg))^2)

# Output the final performance metrics
cat("Boosting Model Performance Metrics:\n")
cat("RMSE:", rmse, "\n")
cat("R-Squared:", r_squared, "\n")

# Plot the results
library(ggplot2)

# Plot: Predicted MPG vs Actual MPG
ggplot(mtcars, aes(x = mpg, y = predictions)) +
  geom_point() +
  geom_abline(intercept = 0, slope = 1, col = "red") +
  theme_minimal() +
  ggtitle("Boosted Model Predictions vs Actual MPG") +
  xlab("Actual MPG") +
  ylab("Predicted MPG")

# Optional: Comparing to a single linear model for reference
single_model <- lm(mpg ~ ., data = mtcars)
single_model_predictions <- predict(single_model, newdata = mtcars)
single_model_rmse <- sqrt(mean((single_model_predictions - mtcars$mpg)^2))
single_model_r_squared <- summary(single_model)$r.squared
```

```
cat("\nSingle Linear Model Performance Metrics:\n")
cat("RMSE:", single_model_rmse, "\n")
cat("R-Squared:", single_model_r_squared, "\n")

# Comparison plot: Boosting vs Single Linear Model
comparison_data <- data.frame(
  Model = c("Boosting", "Single Linear"),
  RMSE = c(rmse, single_model_rmse),
  R_Squared = c(r_squared, single_model_r_squared)
)

# Plot the comparison of RMSE and R-Squared
ggplot(comparison_data, aes(x = Model, y = RMSE, fill = Model)) +
  geom_bar(stat = "identity") +
  theme_minimal() +
  ggtitle("RMSE Comparison: Boosting vs Single Linear Model")

ggplot(comparison_data, aes(x = Model, y = R_Squared, fill = Model)) +
  geom_bar(stat = "identity") +
  theme_minimal() +
  ggtitle("R-Squared Comparison: Boosting vs Single Linear Model")
```



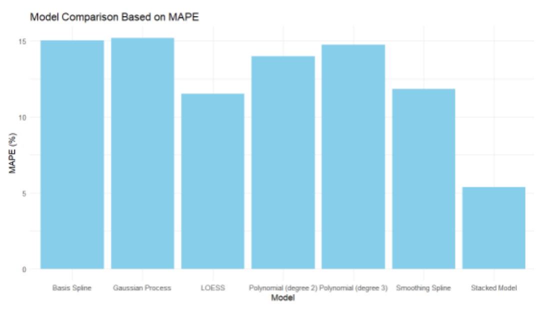
In this case, we ended up with an $R^2 = 1$, which is particularly unlikely in any real-world scenario. While boosting is a powerful technique, it is not that good. This is one reason why cross-validation of models is essential. This technique is more realistic to use on much larger datasets. With the power of 100 iterations, and only 32 observations in the original dataset, it’s very easy to overfit. Compare the results of this algorithm to a much larger dataset, or to one that uses fewer iterations (<<n) for a more realistic sense of what boosting can do.

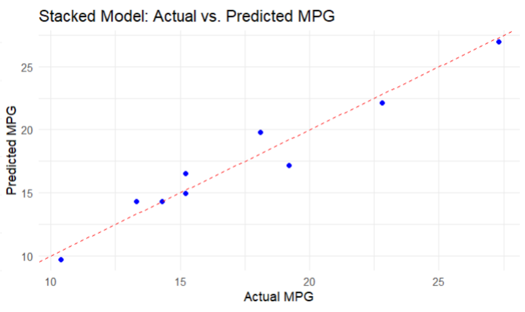
For the stacking example, rather than use linear regression models as the basis for the ensemble, we’ll build our stack from nonlinear models. One of the issues we have is that different models can handle data in different ways, so combining different types of models can be particularly problematic. Here, we need to scale variables for some models, but not other, and LOESS in particular does not support extrapolation, so we’ll need to ensure that the minimum and maximum values are included in the training set so that the test set will not be out of bounds. It may require some experimentation as to which predictor or predictors will do the best job of ensuring this. Or, you may only want to work from similar models that avoid these issues.

```
# Load required libraries
library(GPfit)
library(splines)
library(ggplot2)

# Load the mtcars dataset
data(mtcars)

# Split the data into training and testing sets
set.seed(42)
min_row <- which.min(mtcars$hp)
max_row <- which.max(mtcars$hp)

# Remaining indices after including the extremes
remaining_indices <- setdiff(1:nrow(mtcars), c(min_row, max_row))

# Sample the rest of the data for training and testing
train_indices <- sample(remaining_indices, size = (0.7 * length(remaining_indices)), replace =
FALSE)
train_indices <- c(train_indices, min_row, max_row)  # Ensure extremes are included

test_indices <- setdiff(1:nrow(mtcars), train_indices)

# Step 2: Create training and testing datasets
train_data <- mtcars[train_indices, ]
test_data <- mtcars[test_indices, ]

# Polynomial model (degree 2)
poly2_model <- function(train_data, test_data) {
  model <- lm(mpg ~ poly(hp, 2), data = train_data)
  predict(model, newdata = test_data)
}

# Polynomial model (degree 3)
poly3_model <- function(train_data, test_data) {
  model <- lm(mpg ~ poly(hp, 3), data = train_data)
  predict(model, newdata = test_data)
}

# LOESS model
loess_model <- function(train_data, test_data) {
  model <- loess(mpg ~ hp, data = train_data, span = 0.5)
  predict(model, newdata = test_data)
}

# Smoothing spline model
spline_model <- function(train_data, test_data) {
  model <- smooth.spline(train_data$hp, train_data$mpg)
```

```r
  predict(model, x = test_data$hp)$y
}

# Basis spline model (with penalty)
bspline_model <- function(train_data, test_data) {
  knots <- quantile(train_data$hp, probs = seq(0.2, 0.8, by = 0.2))
  model <- lm(mpg ~ bs(hp, knots = knots), data = train_data)
  predict(model, newdata = test_data)
}

# Gaussian Process model
gp_model <- function(train_data, test_data) {
  # Scaling the 'hp' variable to [0, 1]
  hp_min <- min(train_data$hp)
  hp_max <- max(train_data$hp)
  train_data$hp_scaled <- (train_data$hp - hp_min) / (hp_max - hp_min)
  test_data$hp_scaled <- (test_data$hp - hp_min) / (hp_max - hp_min)

  # Fit the GP model
  model <- GP_fit(train_data$hp_scaled, train_data$mpg)

  # Predict using the fitted GP model
  predict(model, test_data$hp_scaled)$Y_hat
}

# Get predictions from all models
stacked_predictions <- data.frame(
  poly2 = poly2_model(train_data, test_data),
  poly3 = poly3_model(train_data, test_data),
  loess = loess_model(train_data, test_data),
  spline = spline_model(train_data, test_data),
  bspline = bspline_model(train_data, test_data),
  gp = gp_model(train_data, test_data)
)

# Combine the predictions into a new data frame for stacking
stacking_data <- data.frame(
  test_data$mpg, stacked_predictions
)
colnames(stacking_data)[1] <- "mpg"

# Fit a linear model to combine the predictions
stacked_model <- lm(mpg ~ ., data = stacking_data)

# Get the final predictions
final_predictions <- predict(stacked_model, newdata = stacked_predictions)

# Calculate the Mean Absolute Percentage Error (MAPE)
```

```r
mape <- function(actual, predicted) {
  mean(abs((actual - predicted) / actual)) * 100
}

# Compare the MAPE of the models
model_performance <- data.frame(
  Model = c("Polynomial (degree 2)", "Polynomial (degree 3)", "LOESS",
            "Smoothing Spline", "Basis Spline", "Gaussian Process", "Stacked Model"),
  MAPE = c(mape(test_data$mpg, stacked_predictions$poly2),
           mape(test_data$mpg, stacked_predictions$poly3),
           mape(test_data$mpg, stacked_predictions$loess),
           mape(test_data$mpg, stacked_predictions$spline),
           mape(test_data$mpg, stacked_predictions$bspline),
           mape(test_data$mpg, stacked_predictions$gp),
           mape(test_data$mpg, final_predictions))
)

print(model_performance)

# Plot comparison of the model performances
ggplot(model_performance, aes(x = Model, y = MAPE)) +
  geom_bar(stat = "identity", fill = "skyblue") + theme_minimal() +
  labs(title = "Model Comparison Based on MAPE", x = "Model", y = "MAPE (%)")

# Plot the final predictions against the original points
comparison_plot <- data.frame(
  Actual = test_data$mpg, Predicted = final_predictions
)

ggplot(comparison_plot, aes(x = Actual, y = Predicted)) +
  geom_point(color = "blue") +
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "red") +
  theme_minimal() +
  labs(title = "Stacked Model: Actual vs. Predicted MPG",
       x = "Actual MPG", y = "Predicted MPG")
```





We can see that, at least on this particular metric, the stacked model performs much better than any of the individual models alone.

These types of ensemble algorithms are most frequently used in classification models, so we'll be seeing them again as we move on to classification.

Resources:

1. https://daviddalpiaz.github.io/r4sl/ensemble-methods.html