Final Project

AUTHOR
Conor McLaughlin, Kristian Olsson, Michael Vaden

PUBLISHED
December 11, 2023

An Introduction to TidyModels through Spam Detection

Data Introduction

Our Spambase data comes from the UCI Machine Learning Repository and details a classification task for identifying spam emails from a variety of factors. There are 4601 observations representing emails in this dataset, and 57 different continuous features to assist us in our classification. The majority of our features represent the word frequency of certain words that are often common in spam emails, as well as a few features that measure the average, maximum, and total numbers and sequences of capital letters in a given email. There are also no missing values.

TidyModels Methodology

The TidyModels framework is a collection of packages that builds upon tidyverse principles for machine learning and modeling use.

This framework offers an opportunity to standardize many of the approaches we covered in DS 6030, ranging from concepts like data preprocessing and feature engineering to model implementation. Specifically, in this project, we will show how *TidyModels* can be used to preprocess data, tune hyperparameters with various approaches, implement **random forest, xgboost, and penalized logistic regression** models, view feature importance, and examine model results with the *Spambase* data.

In particular, we want to focus on how hyperparameters can be tuned in different ways for our model using this library. Throughout our project, we have hyperlinked the *TidyModels* documentation and other learning resources for each function that we use for the reader's convenience.

Lastly, linked here is the Tidy Modeling with R textbook by Max Kuhn and Julia Silge- this is a great introductory resource to tidymodels and can be used for reference. Let's get started.

Data Preprocessing

One of the first steps in our model building process that we can implement with *TidyModels* is the splitting of our data into a *train* and *test* set using the initial_split function. As we know from class, the *train* set will be used to fit the data while the *test* data will be used to evaluate our model performance

```
set.seed(73)
data_split <- initial_split(spam_data, prop = 4/5)
# data_split <- initial_split(spam_data, strata= spam_label)
x_train <- training(data_split)
x_test <- testing(data_split)
```

After setting our seed, we split our data as shown above. There are a few arguments we can consider for this function, and we wanted to highlight two in particular:

- **prop** indicates the proportion of our total data that should be used for training and fitting our models. In this case, due to our large sample size, we feel comfortable using 80% of the data for training and 20% for testing our results.
- **strata** allows us to pick a variable in our data to conduct stratified sampling. Numeric strata are binned into quartiles. We do not implement this here as our data is not too imbalanced (roughly ~2800 non-spam and 1800 spam observations), but will later for cross-validation.

Finally, the *training()* and *testing()* methods allow us to create our *train* and *test* sets from the split parameters.

Building a Recipe

In general, the *TidyModels* library builds upon the *tidyverse* concept of piping, where we can use the %>% syntax to emphasize a sequence of actions or steps.

TidyModels introduces a concept known as a recipe, which is a description of steps to be applied to the data for preprocessing and analysis.

Recipes essentially turn the data preprocessing used on training data into a single function that can later be applied to the test data. This concept improves code reusability to help make the model evaluation process less cumbersome.

```
random_forest_rec <- recipe(formula=spam_label~., data=x_train) %>%
  step_mutate(spam_label = factor(spam_label))

xgboost_rec <- recipe(formula=spam_label~., data=x_train) %>%
  step_mutate(spam_label = factor(spam_label))
```

```
logistic_reg_rec <- recipe(formula=spam_label~., data=x_train) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_mutate(spam_label = factor(spam_label))
```

For our models, the two most important components of our recipes are the **formula** and the **data**. The formula tells our models what our response variable should be, which in this case is the binary *spam_label*, as well as what predictors we should include. The data indicates what data should be used to train our model, which we got from our split function above.

For our random forest and xgboost models, there are no requirements for our features being normalized as the ensembles of decision trees for our bagging and boosting models split on data quantiles. However, our logistic regression will perform better with our predictors being on the same scale. As a result, we include a normalization step in the logistic regression recipe.

There are many different **step** functions within *TidyModels* found here that can be used in recipes. Step functions allow us to preprocess, impute, and transform our data in different ways. However, we wanted to touch on five basic ones in particular:

- *step_normalize* (used above) allows us to normalize all numeric data to have a standard deviation of one and a mean of zero.
- *step_mutate* (used above) allows us to mutate a feature like the tidyverse *mutate* function. In this case, we use it to make our response variable a factor for our classification approach.
- *step_dummy* allows us to one-hot-encode any character or factor features in our dataset.
- *step_impute_mean* allows us to impute numeric data using the mean for any missing values.
- *step_impute_mode* allows us to impute nominal data using the most common value for any missing values.

In the case of our data, we are very fortunate to have only categorical predictors with no missing values and only need to normalize the data for the logistic regression. However, these step functions above are common preprocessing techniques that could be used for most model-building processes.

Building Model Specifications

The next step in the *TidyModels* process is the initial model creation. For each of our **random forest, xgboost, and penalized logistic regression** models, we want to create specifications about which model parameters to include and eventually tune.

We consider the **mode** of the model which dictates modeling context (commonly "*classification*" or"*regression*"), as well as the **engine** of the model. There are many different engine options, each with their own R package implementation of how the model should be fit. For example if you are building a random forest model you can specify the engine to be either "*ranger*" or "*randomforest*" depending on which package you want to use. When we set the engine, we can add the parameter 'importance = "impurity"', which will provide variable importance scores for the last random forest model.

One of the biggest advantages to the *TidyModels* framework that we want to focus on is its intuitive approach for tuning models. The **tune()** function here acts as a placeholder which marks the parameter for future optimization. Later on, we will find the best value for each of the hyperparameters specified in the models below.

```
random_forest_spec <- rand_forest(
  mtry = tune(),
  trees = 1000,
  min_n = tune()
) %>%
  set_mode("classification") %>%
  set_engine("ranger", importance = 'impurity')

xgboost_spec <- boost_tree(
  trees = 1000,
```

```
  tree_depth = tune(),
  min_n = tune(),
  loss_reduction = tune(),
  sample_size = tune(),
  mtry = tune(),
  learn_rate = tune()
) %>%
  set_mode("classification") %>%
  set_engine("xgboost")

logistic_reg_spec <- logistic_reg(
  penalty = tune(),
  mixture = tune()
) %>%
  set_engine("glmnet")
```

We can see each of the model specifications above built using *TidyModels*. Specifically, we set the model mode to *"classification"* for our random forest and xgboost approaches given our spam detection problem, and build a logistic regression. You may recognize the "*ranger"* and"*glmnet"* libraries from class which we have traditionally used to implement random forest and logistic regression, and this approach allows us to easily incorporate these libraries as our engines in a larger model-building process.

Additionally, we list many of the hyperparameters we have been exposed to in class for these different models, using **tune()** to specify that we will find a good combination of these hyperparameters to optimize each of our models.

As a refresher, some of the important hyperparameters for both random forest and xgboost are:

- **mtry** is the number of predictors that will be randomly sampled at each split when creating tree models.
- **min_n** is the minimum number of data points in a node that are required for the node to be split further within a tree.

And parameters for penalized logistic regression:

- **penalty** specifies the penalty term that shrinks the coefficients in the regression towards zero using regularization.
- **mixture** specifies which regression type should be used, with a value of 1 using lasso regression and a value of 0 using ridge regression.

Tuning Hyperparameters with Grid Search

Now, after selecting which model parameters should be tuned and optimized for the best results, we can specify which ranges of these values should be tested. In particular, we want to focus on the grid search approach, which is when we evaluate a predefined set of parameters. In this project, we will focus on three different ways within *TidyModels* to implement a grid search to optimize hyperparameters.

For each of our models below, we can see the parameters we need to tune.

```
rf_params <- extract_parameter_set_dials(random_forest_spec)
rf_params
```

```
Collection of 2 parameters for tuning

 identifier  type    object
       mtry  mtry nparam[?]
      min_n min_n nparam[+]

Model parameters needing finalization:
   # Randomly Selected Predictors ('mtry')

See `?dials::finalize` or `?dials::update.parameters` for more information.
```

```
xg_params <- extract_parameter_set_dials(xgboost_spec)
xg_params
```

```
Collection of 6 parameters for tuning

     identifier           type    object
           mtry           mtry nparam[?]
```

```
          min_n          min_n nparam[+]
     tree_depth     tree_depth nparam[+]
     learn_rate     learn_rate nparam[+]
 loss_reduction loss_reduction nparam[+]
    sample_size    sample_size nparam[+]

Model parameters needing finalization:
   # Randomly Selected Predictors ('mtry')

See `?dials::finalize` or `?dials::update.parameters` for more information.
```

```
lg_params <- extract_parameter_set_dials(logistic_reg_spec)
lg_params
```

```
Collection of 2 parameters for tuning

 identifier    type    object
    penalty penalty nparam[+]
    mixture mixture nparam[+]
```

The three approaches that we implement here **grid_regular, grid_random, and grid_latin_hypercube**.

Grid Selection and Characteristics

Both Regular and Random Grids can be created for any number of parameter objects. The difference in these two approaches is that:

- **Regular Grid** uses all combinations of given parameter sets
- **Random Grid** generates independent uniform random numbers across the parameter set ranges

```
lg_params %>% extract_parameter_dials("mixture")
```

```
Proportion of Lasso Penalty (quantitative)
Range: [0.05, 1]
```

```
lg_params %>% extract_parameter_dials("penalty")
```

```
Amount of Regularization (quantitative)
Transformer: log-10 [1e-100, Inf]
Range (transformed scale): [-10, 0]
```

These grids approaches automatically leverage the default values for potential model parameters (shown above) from the engine that we've selected, which in the case of logistic regression is *glmnet*.

```
lg_params %>% grid_regular(levels = c(mixture = 3, penalty = 3))
```

```
# A tibble: 9 × 2
       penalty mixture
         <dbl>   <dbl>
1 0.0000000001   0.05
2 0.00001        0.05
3 1              0.05
4 0.0000000001   0.525
5 0.00001        0.525
6 1              0.525
# ℹ 3 more rows
```

Here, we can see how grid_regular leverages the default values for penalty and mixture from the *glmnet* engine. We use the **levels** argument to specify the number of levels per parameter to create (in this case 3), and the regular grid then matches each combination of parameters for optimization.

```
rf_params %>% extract_parameter_dials("mtry")
```

```
# Randomly Selected Predictors (quantitative)
Range: [1, ?]
```

```
rf_params %>% extract_parameter_dials("min_n")
```

```
Minimal Node Size (quantitative)
Range: [2, 40]
```

One issue that we run into with both our random forest (shown above) and xgboost approaches is that the **mtry** parameter does not have a default upper bound, meaning it contains unknown() parameter values shown by the "?" in the range. One way that *TidyModels* allows us to address this issue is by overriding the default range given for a parameter, such as **mtry** from the *ranger* engine:

```
grid_regular(
  mtry(range = c(6, 8)),
  min_n()
  )
```

```
# A tibble: 9 × 2
   mtry min_n
  <int> <int>
1     6     2
2     7     2
3     8     2
4     6    21
5     7    21
6     8    21
# ℹ 3 more rows
```

The regular grid above is an example of the utility provided by these functions, where we can determine a range of values for one or more parameters, while keeping default engine values for others.

One issue with the regular grid approach as implemented here is that because all possible values of each parameter set are compared, this operation quickly becomes costly when working with a model such as xgboost which has 6 different parameters to combine and tune.

The random grid is an example of an *irregular grid*, and solves this issue by generating random samples of the parameter values from the given ranges.

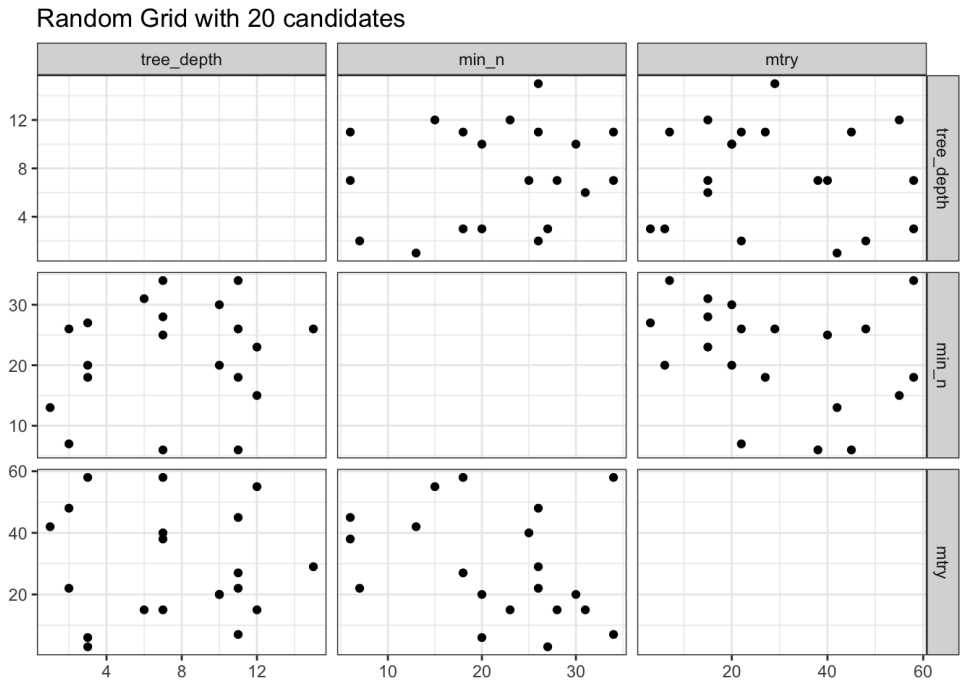
```
xg_example_random_grid <- grid_random(
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(),
  finalize(mtry(), x_train),
  learn_rate(),
  size = 20
)
xg_example_random_grid
```

```
# A tibble: 20 × 6
   tree_depth min_n loss_reduction sample_size  mtry learn_rate
        <int> <int>          <dbl>       <dbl> <int>      <dbl>
1          12    15       5.78e- 7       0.934    55   7.58e- 7
2           1    13       7.24e- 4       0.830    42   3.27e- 8
3           7    34       9.15e- 1       0.718    58   1.80e- 9
4          10    20       3.61e+ 0       0.960    20   5.90e- 3
5           7     6       6.14e- 4       0.957    38   8.12e-10
6          11     6       7.89e-10       0.584    45   9.99e- 6
# ℹ 14 more rows
```

As shown above, the random grid allows us to take random value combinations from each of our parameter sets for efficiency. In this example, the **size** argument allows us to specify the total combination of parameters to get from the grid. Additionally, the finalize() function here is a different approach to solving our unknown range issue for **mtry**: *finalize()* modifies the unknown parts of ranges based on a data set for our grid.

However, although the random grid acts as a more efficient method for hyperparameter optimization than the regular grid, it introduces a new issue: we want our sampled values to cover the whole parameter space, but this requires us to increase the number of parameter combinations substantially.

```
xg_example_random_grid %>%
  ggplot(aes(x = .panel_x, y = .panel_y)) +
  geom_point() +
  geom_blank() +
  facet_matrix(vars(tree_depth, min_n, mtry), layer.diag = 2) +
  labs(title = "Random Grid with 20 candidates")
```



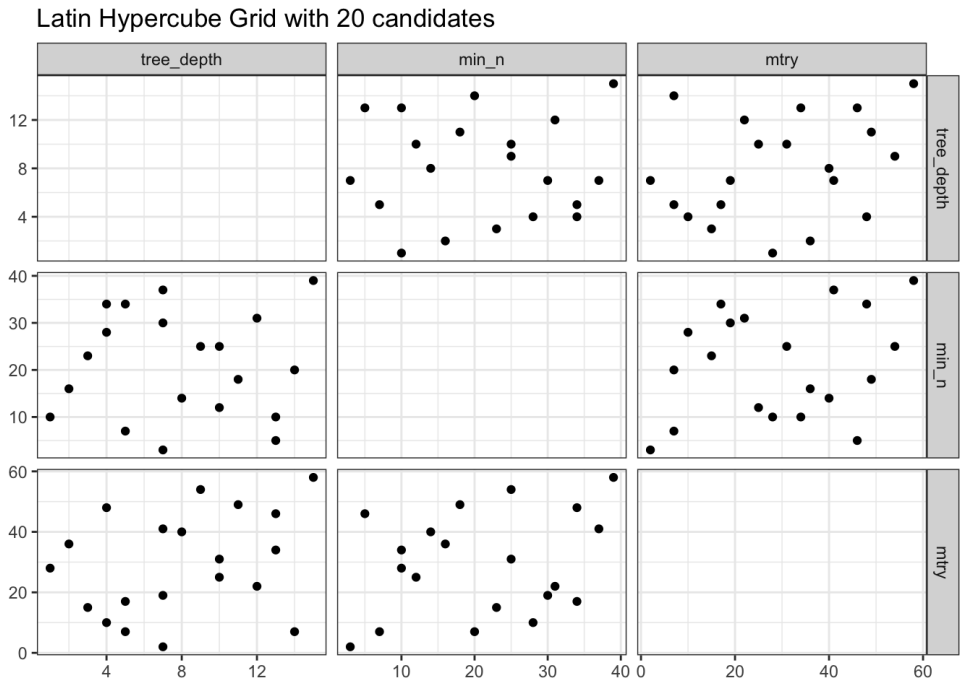
*plotting code taken from https://www.tmwr.org/grid-search

We can see that our randomly selected parameters for the xgboost model do not necessarily cover the entire parameter space and some of our parameters combinations overlap.

This brings us to our third and final approach to parameter optimization with Grid Search: The Latin Hypercube Grid.

The *Latin Hypercube grid* is a space-filling parameter grid that attempts to cover the parameter space equally (you could also substitute the Maximum Entropy Grid, which produces similar results although we will not cover it in this project). This grid functions as a compromise of sorts between regular and random grid approaches, where it uses random but stratified sampling where each parameter is split into intervals of equal probability before sampling each interval. More on Latin Hypercubes here.

```
grid_latin_hypercube(
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(),
  finalize(mtry(), x_train),
  learn_rate(),
  size = 20
) %>%
  ggplot(aes(x = .panel_x, y = .panel_y)) +
  geom_point() +
  geom_blank() +
  facet_matrix(vars(tree_depth, min_n, mtry), layer.diag = 2) +
  labs(title = "Latin Hypercube Grid with 20 candidates")
```



Here, we can see that the *Latin Hypercube* approach does a better job of covering the parameter space and increasing distance between parameter combination samples. Now, let's actually implement these different grids for our models.

Grid Implementation

```
random_forest_grid <- grid_random(
  finalize(mtry(), x_train),
  min_n(),
  size = 30
)

xgboost_grid <- grid_latin_hypercube(
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(),
  finalize(mtry(), x_train),
  learn_rate(),
  size = 30
)

logistic_reg_grid <- grid_regular(
  penalty(),
  mixture(range = c(0, 1)),
  levels = c(mixture = 5, penalty = 20)
)
```

For our random forest model, we are only concerned about two model parameters. We use *finalize()* to make the **mtry** values fit the range in our train set, and the default range of **min_n**. Because there are many potential values for both of these parameters but only two of them, we feel comfortable using a random grid to tune this model.

For our xgboost model, however, we implement the *Latin Hypercube grid* because we have to find the an optimal combination of 6 parameters and are concerned about sampling well from the parameter space. However, this approach still allows our sampling to not be too costly by using elements of random sampling.

Finally, a regular grid is appropriate for our logistic regression model. There are only a few values of **mixture** that we are concerned with, and we modify the range so that both full lasso regression (1) and full ridge regression (0) are included in our tuning. We then use a higher level of **penalty** to compare with these **mixture** values to optimize for our eventual results.

For each of these models, we do not want to excessively sample combinations of parameters and *overfit* our models. As a result, 30 combinations for each model is appropriate.

Building Model Workflows

Now that we have used the *TidyModels* library to split our data, make recipes that specify and preprocess our data, create model templates that specify hyperparameters, and build parameter grids to optimize our model hyperparameters, it is time to put it all together.

Workflows allow us to bundle together our data preprocessing and model structures for convenient implementation. More on workflows here.

```
random_forest_workflow <- workflow() %>%
  add_model(random_forest_spec) %>%
  add_recipe(random_forest_rec)

xgboost_workflow <- workflow() %>%
  add_model(xgboost_spec) %>%
  add_recipe(xgboost_rec)

logistic_reg_workflow <- workflow() %>%
  add_model(logistic_reg_spec) %>%
  add_recipe(logistic_reg_rec)
```

The workflows for each of the **random forest, xgboost, and logistic regression** models include their model specifications and recipes respectively.

Train and Tune Models

Now that we have our workflow to define our models and our different parameter grids to optimize our model parameters, we use the tune_grid() function to put this into practice. *Tune Grid* allows us to combine our predefined workflows and our parameter grids to tune each model, find our best combinations of parameters, and compute a set of performance metrics for each combination.

Tune Grid also allows us to incorporate k-fold cross validation into our models through the **resamples** argument.

```
kfold = vfold_cv(x_train, strata = spam_label, v = 5)
```

The *vfold_cv* function takes a dataframe as an argument, and there are a few other arguments we can consider:

- **strata** allows us to conduct a stratified split on a variable, which in our case ensures that each split of our data has similar proportions of 'spam' and 'not spam' observations
- **v** dictates the number of equally-sized folds to partition our data into for cross-validation. Although the default value is 10, we use a 5-fold cross validation here due to the size of our data and number of parameters we are tuning.

More info on resampling with *TidyModels* here, Below is our implementation using *Tune Grid* and *vfold_cv* created above.

```
doParallel::registerDoParallel()

random_forest_result <- tune_grid(
  random_forest_workflow,
  resamples = kfold,
  grid = random_forest_grid,
  control = control_grid(save_pred = TRUE, save_workflow = TRUE)#,
  #metrics = metric_set(rmse, rsq)
)
show_notes(.Last.tune.result)
```

```
  Great job! No notes to show.
```

```
doParallel::registerDoParallel()

xgboost_result <- tune_grid(
  xgboost_workflow,
  resamples = kfold,
  grid = xgboost_grid,
  control = control_grid(save_pred = TRUE, save_workflow = TRUE)#,
  #metrics = metric_set(rmse, rsq)
```

```
)
show_notes(.Last.tune.result)
```

```
Great job! No notes to show.
```

```
doParallel::registerDoParallel()

logistic_reg_result <- tune_grid(
  logistic_reg_workflow,
  resamples = kfold,
  grid = logistic_reg_grid,
  control = control_grid(save_pred = TRUE, save_workflow = TRUE)#,
  #metrics = metric_set(rmse, rsq)
)
show_notes(.Last.tune.result)
```

```
Great job! No notes to show.
```

After our *tune grids* for each model are finished running, our result variables will contain the parameter combinations for our tuning variables that perform best for a given metric. In this case, we use the show_best function to display the best-performing sub models with their parameters and metric of choice. The results for each fold of the cross-validation is shown.

```
show_best(random_forest_result, 'roc_auc')
```

```
# A tibble: 5 × 8
   mtry min_n .metric .estimator  mean     n  std_err .config
  <int> <int> <chr>   <chr>      <dbl> <int>    <dbl> <chr>
1     4     6 roc_auc binary     0.986     5 0.000960 Preprocessor1_Model11
2     3    22 roc_auc binary     0.986     5 0.00102  Preprocessor1_Model18
3     2    12 roc_auc binary     0.986     5 0.00109  Preprocessor1_Model06
4     6    14 roc_auc binary     0.986     5 0.00106  Preprocessor1_Model03
5     5    25 roc_auc binary     0.985     5 0.000990 Preprocessor1_Model20
```

```
show_best(xgboost_result, 'roc_auc')
```

```
# A tibble: 5 × 12
   mtry min_n tree_depth learn_rate loss_reduction sample_size .metric
  <int> <int>      <int>      <dbl>          <dbl>       <dbl> <chr>
1    45     4         13    0.0337     0.000000145       0.654 roc_auc
2    47    16          7    0.0139     0.0756            0.717 roc_auc
3    13    28          5    0.0112     0.000000114       0.941 roc_auc
4    27    36          3    0.00516    0.00000699        0.994 roc_auc
5     7    13          5    0.00167   16.5               0.804 roc_auc
# ℹ 5 more variables: .estimator <chr>, mean <dbl>, n <int>, std_err <dbl>,
#   .config <chr>
```

```
show_best(logistic_reg_result, 'roc_auc')
```

```
# A tibble: 5 × 8
   penalty mixture .metric .estimator  mean     n std_err .config
     <dbl>   <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1 6.16e- 5       1 roc_auc binary     0.972     5 0.00296 Preprocessor1_Model092
2 1   e-10       1 roc_auc binary     0.972     5 0.00299 Preprocessor1_Model081
3 3.36e-10       1 roc_auc binary     0.972     5 0.00299 Preprocessor1_Model082
4 1.13e- 9       1 roc_auc binary     0.972     5 0.00299 Preprocessor1_Model083
5 3.79e- 9       1 roc_auc binary     0.972     5 0.00299 Preprocessor1_Model084
```

Above we can see the best combinations of model parameters for each fold of the cross-validation by maximizing the metric of Area under the receiver operator curve, also known as *roc_auc*.

Model Predictions

Now that we have our tuned parameter combinations that maximize our model performance on the *train* set, we need to update our workflow with these parameters and apply it to the *test* set to evaluate our models. Specifically, we use the finalize_workflow() function

to update our original model workflows with the optimal combinations of parameters produced by our tuning grid processes.

Additionally, we can use the *TidyModels* last_fit() function here, which takes our finalized workflow and uses our *data_split* object with the *train* and *test* data to first fit the entirety of the training set before evaluating on the testing set. In essence, this function takes our tuned model and automates the process of evaluating our model with our split data.

Lastly, we use the collect_predictions() function to get a dataframe of our prediction probabilities for spam classes, our binary prediction resulting from these probabilities, and the true values for each email observation in the *test* data.

```
final_random_forest_workflow <-
  random_forest_workflow %>%
  finalize_workflow(select_best(random_forest_result, "roc_auc"))

final_random_forest_results = final_random_forest_workflow %>% last_fit(data_split)

head(collect_predictions(final_random_forest_results))
```

```
  # A tibble: 6 × 7
    id               .pred_0 .pred_1  .row .pred_class spam_label .config
    <chr>              <dbl>   <dbl> <int> <fct>       <fct>      <chr>
  1 train/test split 0.00609   0.994     1 1           1          Preprocessor1_M…
  2 train/test split 0.00934   0.991     2 1           1          Preprocessor1_M…
  3 train/test split 0.109     0.891     3 1           1          Preprocessor1_M…
  4 train/test split 0.109     0.891     4 1           1          Preprocessor1_M…
  5 train/test split 0.547     0.453     7 0           1          Preprocessor1_M…
  6 train/test split 0.0595    0.940     9 1           1          Preprocessor1_M…
```

```
final_xgboost_workflow <-
  xgboost_workflow %>%
  finalize_workflow(select_best(xgboost_result, "roc_auc"))

final_xgboost_results = final_xgboost_workflow %>% last_fit(data_split)

head(collect_predictions(final_xgboost_results))
```

```
  # A tibble: 6 × 7
    id                .pred_0 .pred_1  .row .pred_class spam_label .config
    <chr>               <dbl>   <dbl> <int> <fct>       <fct>      <chr>
  1 train/test split 0.000276   1.00      1 1           1          Preprocessor1_…
  2 train/test split 0.00129    0.999     2 1           1          Preprocessor1_…
  3 train/test split 0.00523    0.995     3 1           1          Preprocessor1_…
  4 train/test split 0.00531    0.995     4 1           1          Preprocessor1_…
  5 train/test split 0.157      0.843     7 1           1          Preprocessor1_…
  6 train/test split 0.00429    0.996     9 1           1          Preprocessor1_…
```

```
final_logistic_reg_workflow <-
  logistic_reg_workflow %>%
  finalize_workflow(select_best(logistic_reg_result, "roc_auc"))

final_logistic_reg_results = final_logistic_reg_workflow %>% last_fit(data_split)

head(collect_predictions(final_logistic_reg_results))
```

```
  # A tibble: 6 × 7
    id                  .pred_0 .pred_1  .row .pred_class spam_label .config
    <chr>                 <dbl>   <dbl> <int> <fct>       <fct>      <chr>
  1 train/test split 0.0133       0.987     1 1           1          Preprocessor…
  2 train/test split 0.00000173   1.00      2 1           1          Preprocessor…
  3 train/test split 0.222        0.778     3 1           1          Preprocessor…
  4 train/test split 0.222        0.778     4 1           1          Preprocessor…
  5 train/test split 0.328        0.672     7 1           1          Preprocessor…
  6 train/test split 0.128        0.872     9 1           1          Preprocessor…
```

Model Evaluation

Train and Predict with Base Models

We also used base models for each of our **random forest, xgboost, and logistic regression** approaches so that we can compare the results to our tuned and cross-validated models. For the base models, we use a similar workflow to our tuned approach, but without tuning and optimizing our parameters for model performance.

```
base_random_forest = workflow() %>% add_model(
  rand_forest() %>%
  set_mode("classification") %>%
  set_engine("ranger", importance = 'impurity')
) %>%
  add_recipe(random_forest_rec) %>%
  last_fit(data_split)


base_xgboost = workflow() %>% add_model(
  boost_tree() %>%
  set_mode("classification") %>%
  set_engine("xgboost")
) %>%
  add_recipe(xgboost_rec) %>%
  last_fit(data_split)

base_logistic_reg <- workflow() %>% add_model(
  logistic_reg(penalty=.001) %>%
  set_engine("glmnet")
) %>%
  add_recipe(logistic_reg_rec) %>%
  last_fit(data_split)
```

Model Metrics and ROC Curves

Tidy Models also provides a function called collect_metrics() that easily allows us to find the model evaluation metrics for our best models. We will use this to quickly compare the performance of our tuned models with the base models.

```
# tuned
collect_metrics(final_random_forest_results)
```

```
# A tibble: 2 × 4
  .metric  .estimator .estimate .config
  <chr>    <chr>          <dbl> <chr>
1 accuracy binary         0.941 Preprocessor1_Model1
2 roc_auc  binary         0.982 Preprocessor1_Model1
```

```
# base
collect_metrics(base_random_forest)
```

```
# A tibble: 2 × 4
  .metric  .estimator .estimate .config
  <chr>    <chr>          <dbl> <chr>
1 accuracy binary         0.943 Preprocessor1_Model1
2 roc_auc  binary         0.982 Preprocessor1_Model1
```

We see that the tuned random forest model has slightly worse performance than the model that had no tuning. This could be due to the tuned model overfitting to the training data causing it to perform worse on the out of sample observations in the test set.

```
#tuned
collect_metrics(final_xgboost_results)
```

```
# A tibble: 2 × 4
  .metric  .estimator .estimate .config
  <chr>    <chr>          <dbl> <chr>
1 accuracy binary         0.948 Preprocessor1_Model1
2 roc_auc  binary         0.983 Preprocessor1_Model1
```

```
# base
collect_metrics(base_xgboost)
```

```
# A tibble: 2 × 4
  .metric  .estimator .estimate .config
  <chr>    <chr>          <dbl> <chr>
1 accuracy binary         0.935 Preprocessor1_Model1
2 roc_auc  binary         0.975 Preprocessor1_Model1
```

```
#tuned
collect_metrics(final_logistic_reg_results)
```

```
# A tibble: 2 × 4
  .metric  .estimator .estimate .config
  <chr>    <chr>          <dbl> <chr>
1 accuracy binary         0.921 Preprocessor1_Model1
2 roc_auc  binary         0.971 Preprocessor1_Model1
```

```
# base
collect_metrics(base_logistic_reg)
```

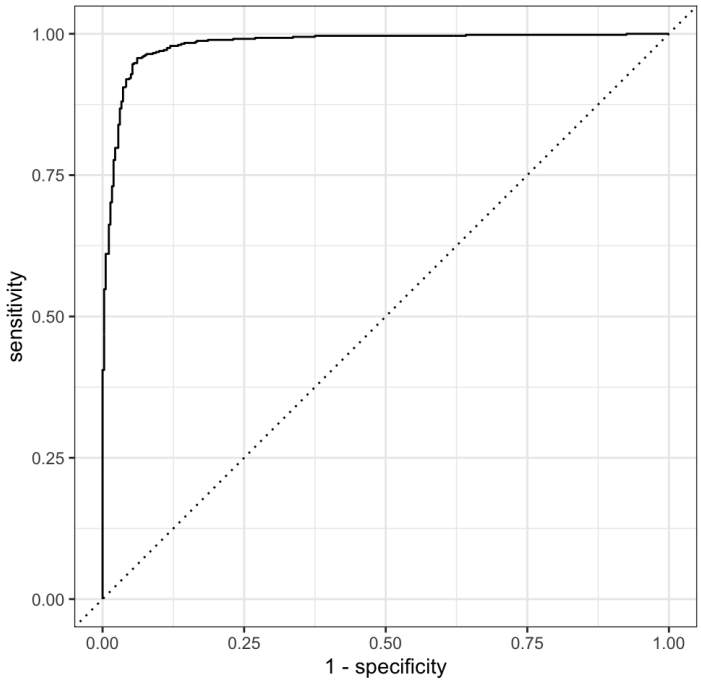
```
# A tibble: 2 × 4
  .metric  .estimator .estimate .config
  <chr>    <chr>          <dbl> <chr>
1 accuracy binary         0.913 Preprocessor1_Model1
2 roc_auc  binary         0.969 Preprocessor1_Model1
```

We see that the tuned xgboost and logistic regression models performed better than their non-tuned counterparts. This is expected and shows our tuning worked!

We can also use the results of *collect_predictions()* function to easily plot the ROC Curves using the Yardstick package we used in class.

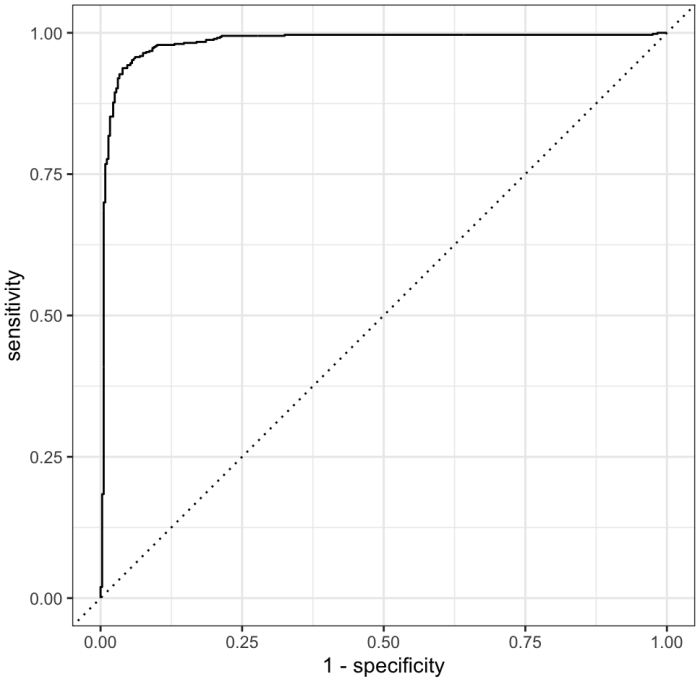
```
rf_auc <-
  collect_predictions(final_random_forest_results)%>%
  roc_curve(spam_label, .pred_0)%>%
  mutate(model = "Random Forest")

autoplot(rf_auc)
```



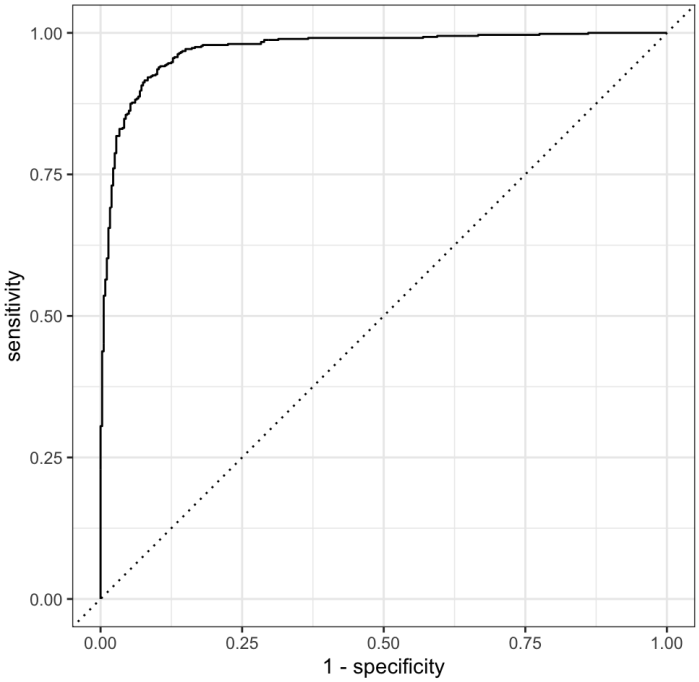
```
xgboost_auc <-
  collect_predictions(final_xgboost_results)%>%
  roc_curve(spam_label, .pred_0) %>%
  mutate(model = "xgboost")

autoplot(xgboost_auc)
```

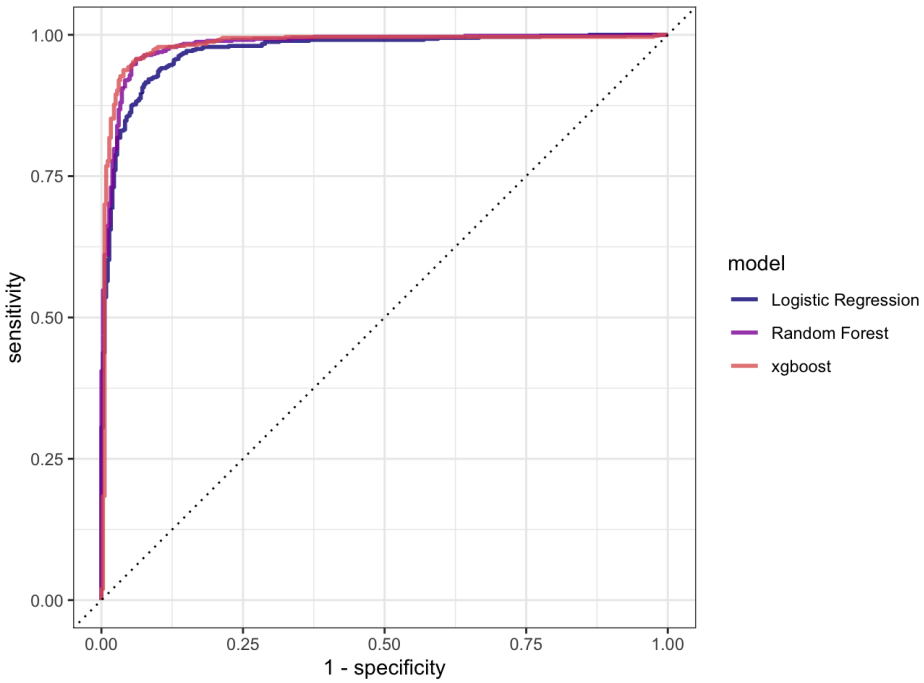



```
lr_auc <-
  collect_predictions(final_logistic_reg_results)%>%
  roc_curve(spam_label, .pred_0) %>%
  mutate(model = "Logistic Regression")

autoplot(lr_auc)
```



```
bind_rows(rf_auc, xgboost_auc, lr_auc) %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity, col = model)) +
  geom_path(lwd = 1, alpha = 0.8) +
  geom_abline(lty = 3) +
  coord_equal() +
  scale_color_viridis_d(option = "plasma", end = .6)
```



Looking at the above ROC curves and the respective AUC values, we find that xgboost performed the best of our three models, although the results are similar to and just slightly better than the random forest model. The logistic regression model performs the worst of the three. Overall, these models perform well with regards to sensitivity and specificity. Next, we look at the confusion matrices and calculate the False Positive rate, as this measures how many non-spam emails we classify as spam, which we want to minimize.

Confusion Matrices and False Positive Rate

We can also use confusion matrices to get a more comprehensive view of the model's predictions by breaking down the results into:

- True Negatives: Observations the model predicted were not spam (0) that were actually not spam (0)
- True Positives: Observations the model predicted were spam (1) that were actually spam (1)
- False Positives: Observations the model predicted were spam (1) that were actually not spam (0)
  - Type I Error
- False Negatives: Observations the model predicted were not spam (0) that were actually spam (1)
  - Type II Error

```
rf_conf_mat <- conf_mat(collect_predictions(final_random_forest_results)
                        , truth = spam_label
                        , estimate = .pred_class)

rf_conf_mat
```

```
          Truth
Prediction   0   1
         0 542  36
         1  18 324
```

```
xg_conf_mat <- conf_mat(collect_predictions(final_xgboost_results)
                        , truth = spam_label
                        , estimate = .pred_class)

xg_conf_mat
```

```
           Truth
Prediction   0   1
         0 539  27
         1  21 333
```

```
lr_conf_mat <- conf_mat(collect_predictions(final_logistic_reg_results)
                        , truth = spam_label
                        , estimate = .pred_class)
lr_conf_mat
```

```
           Truth
Prediction   0   1
         0 536  49
         1  24 311
```

Since we care most about false positives (Type I Error), we can use the confusion matrices to calculate the false positive rate. The false positive rate is the probability that a positive result is given when the true value is negative. It is calculated using the below formula:

[FPR =]

```
paste("Random Forest FPR: ", round(rf_conf_mat$table[2] / (rf_conf_mat$table[1] + rf_conf_mat$table[2]),4))
```

```
[1] "Random Forest FPR:  0.0321"
```

```
paste("XGBoost FPR: ", round(xg_conf_mat$table[2] / (xg_conf_mat$table[1] + xg_conf_mat$table[2]),4))
```

```
[1] "XGBoost FPR:  0.0375"
```

```
paste("Logistic Regression FPR: ",round(lr_conf_mat$table[2] / (lr_conf_mat$table[1] + lr_conf_mat$table[2]),4))
```

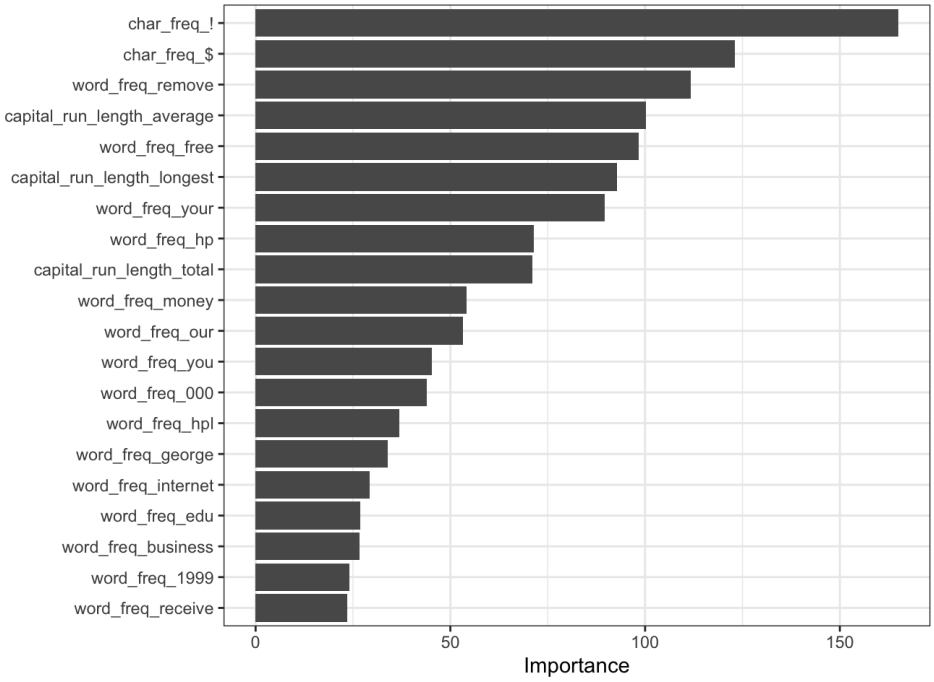
```
[1] "Logistic Regression FPR:  0.0429"
```

The confusion matrices show that the random forest had the least number of false positives (18) with a false positive rate of 0.0321. The xgboost model had 21 false positives with a false positive rate of 0.0375. Finally, the logistic regression model had 24 false positives with a false positive rate of 0.0429. Thus, the random forest performs best in terms of minimizing the number of false positive predictions.

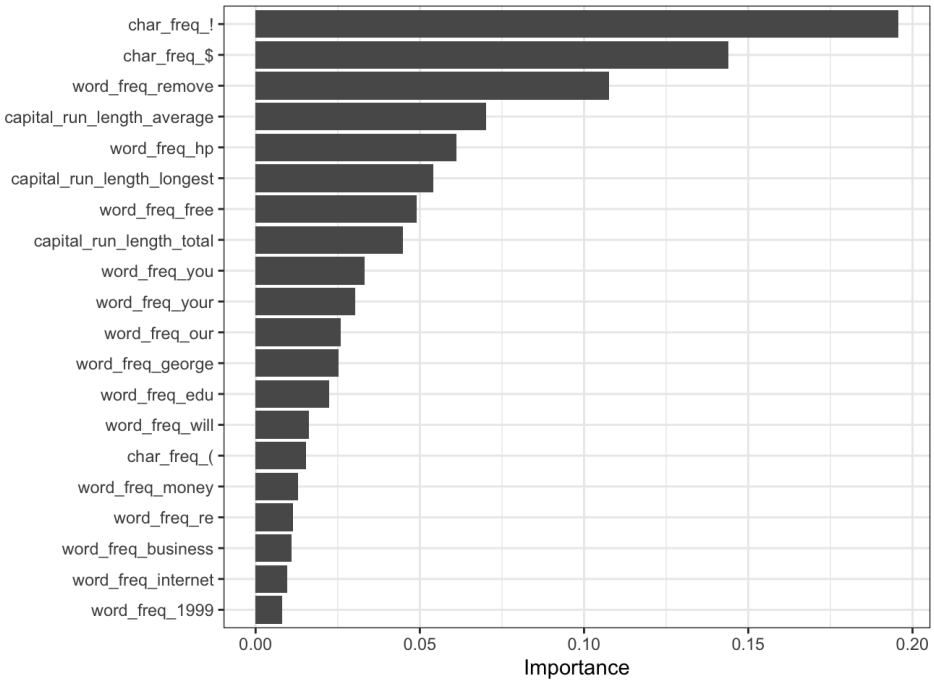
Variable Importance

The variable importance plots show the relative importance of the predictors in predicting spam or not spam for each model. Below we show the top 20 most important variables for each model.

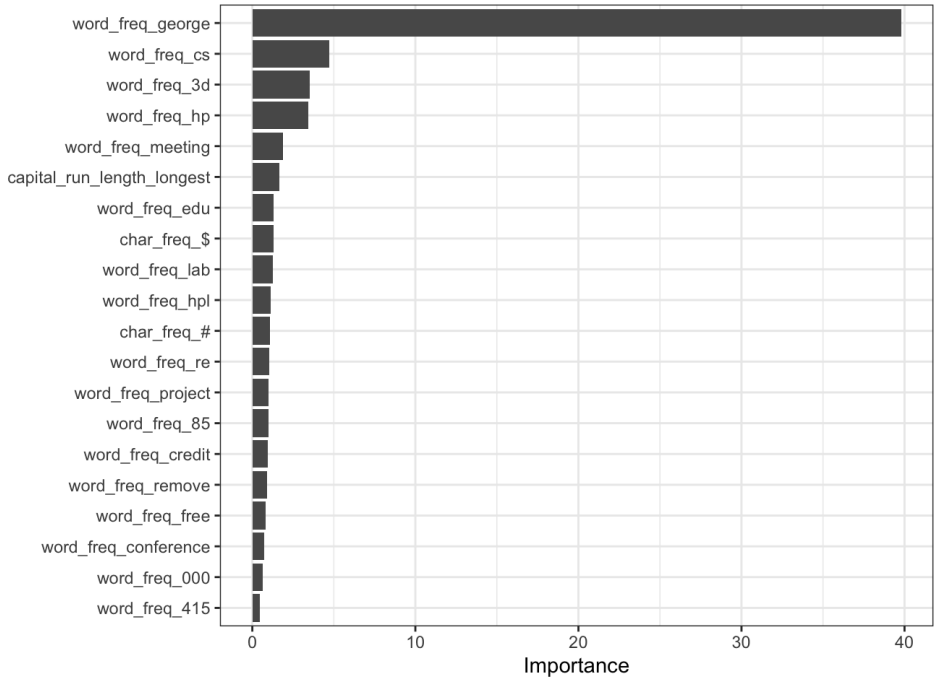
```
final_random_forest_results %>%
  extract_fit_parsnip() %>%
  vip(num_features = 20)
```




```
final_xgboost_results %>%
  extract_fit_parsnip() %>%
  vip(num_features = 20)
```




```
final_logistic_reg_results %>%
  extract_fit_parsnip() %>%
  vip(num_features = 20)
```



In the variable importance plots for the random forest and xgboost model, we see that the top four most important variables are the same for each. These variables were: *char_freq_!*, *char_freq_$*, *word_freq_remove*, and *capital_run_average_length*. For the logistic regression model, the most important variables were: *word_freq_george*, *word_freq_cs*, *word_freq_3d*,and *word_freq_hp*. By looking at these plots, we are able to understand what predictors are most important in predicting whether an email was spam or not.

Conclusion

In this project, we were able to apply Tidy Models to a common classification problem of spam filtering. Using Tidy Models, we are able to predict spam emails correctly with a high success rate while presenting methods to tune parameters, which is an essential component of model building. Moreover, we are able to analyze the difference in performance between models through the ROC curves, Type 1 Error, and variable importance plots.

Tidy Models is a very useful package that has simplified the process of model building and tuning through its recipe, workflow, and engine parameters. In addition, Tidy Models can be extrapolated to many model types as shown in this project. We believe this project can be used as a demonstration to those who are interested in learning about and applying Tidy Models to data science problems.

References

https://www.tmwr.org/

https://tidymodels.org

https://tidyverse.org

https://www.jstor.org/stable/1268522?seq=1