Final Project

Code

Ben Hertzberg

2022-11-28

Introduction

In this project, I use a variety of machine learning models to predict the model of a used car listed for sale given general information about the vehicle.

This may seem like an odd relationship to investigate, but in my eyes that was kind of the point. Originally, I planned on predicting a used car's price given the listing information. I wanted to see how make, model, mileage, year, etc would effect a used cars price, which could then be used to see if a car for sale was a good deal. However, although such a prediction might have more purpose in the real world, I realized I wasn't particularly interested in the relationships I was analyzing. Some models are cheaper than others. Cars with low mileage will be more expensive than those with higher mileage. Most, if not all, of the relationships between the predictor variables and the response followed very straightforward logic. As I built my original recipe, I wasn't excited to see what my models would reveal. So I decided to switch things up.

Predicting a used car's model might not have a clear use in the real world, but as a newcomer to machine learning, it seemed far more interesting. There are not many common sense relationships between predictors and the response in this case. Multiple car models fall in the same price range. Any car can have any amount of miles driven. Given these obstacles, I wasn't sure it would even be possible. However, if the machine learning models could do it, I would have built a tool that learned relationships I don't understand myself, and that seemed a much more rewarding and captivating challenge.

First things first - I loaded in a bunch of R packages, which provide prewritten code functions that I can use later on.

Code

It All Starts With Data

My models were trained on used car data taken from a Kaggle dataset provided by Rupesh Raundal, which can be found using this link:

https://www.kaggle.com/datasets/3ea0a6a45dbd4713a8759988845f1a58038036d84515ded58f65a2ff2bd32e00?resource=download&select=us-dealers-used.csv (https://www.kaggle.com/datasets/3ea0a6a45dbd4713a8759988845f1a58038036d84515ded58f65a2ff2bd32e00?resource=download&select=us-dealers-used.csv)

*This file, and the folder it is contained in, are not in my github repo due to the file size.

Raundal provided a dataset for the US, as well as one for Canada, but I only used the US file. I read in the data, and cleaned up the variable names from the data set so they all follow snake case (no capitalization and words separated by underlines, which_looks_like_this).

Code

Importing the data was easier said than done - the file on US used cars was over 1.28 gigabytes and contained millions of data points. I had to immediately subset the data to prevent errors regarding memory allocation. I took a random sample of 50,000 cars to start off with. Prior to this, I set a random seed value, which allows the randomly chosen sample to be reproduced.

With the data in a more manageable size, I filled all blank spaces with 'NA', which lets R know that there is no data contained. Then I removed all observations containing 'NA'. After passing through this filter, the data set contained 43,551 observations.

Code

The result: a reasonably sized data set with standardized variable names and no missing data. I saved this into a new file so I wouldn't ever have to work with the original behemoth again.

Code

Right off the bat I removed variables I am not interested in. The original set had 21 variables(including model). I was not interested in the vehicles ID, VIN or stock number, and did not need information on the seller name and location.

I also made one adjustment to the outcome variable. Some of the model names contained spaces, which can be problematic when coding. So I replaced all spaces in the car model categories with underscores, following the snake case that I used for variable names.

Code

I removed 'trim' as well, because sometimes there are trim names that are unique to one specific car model. In those cases, the machine learning model wouldn't be predicting anything - just matching the model to the corresponding trim name.

Code

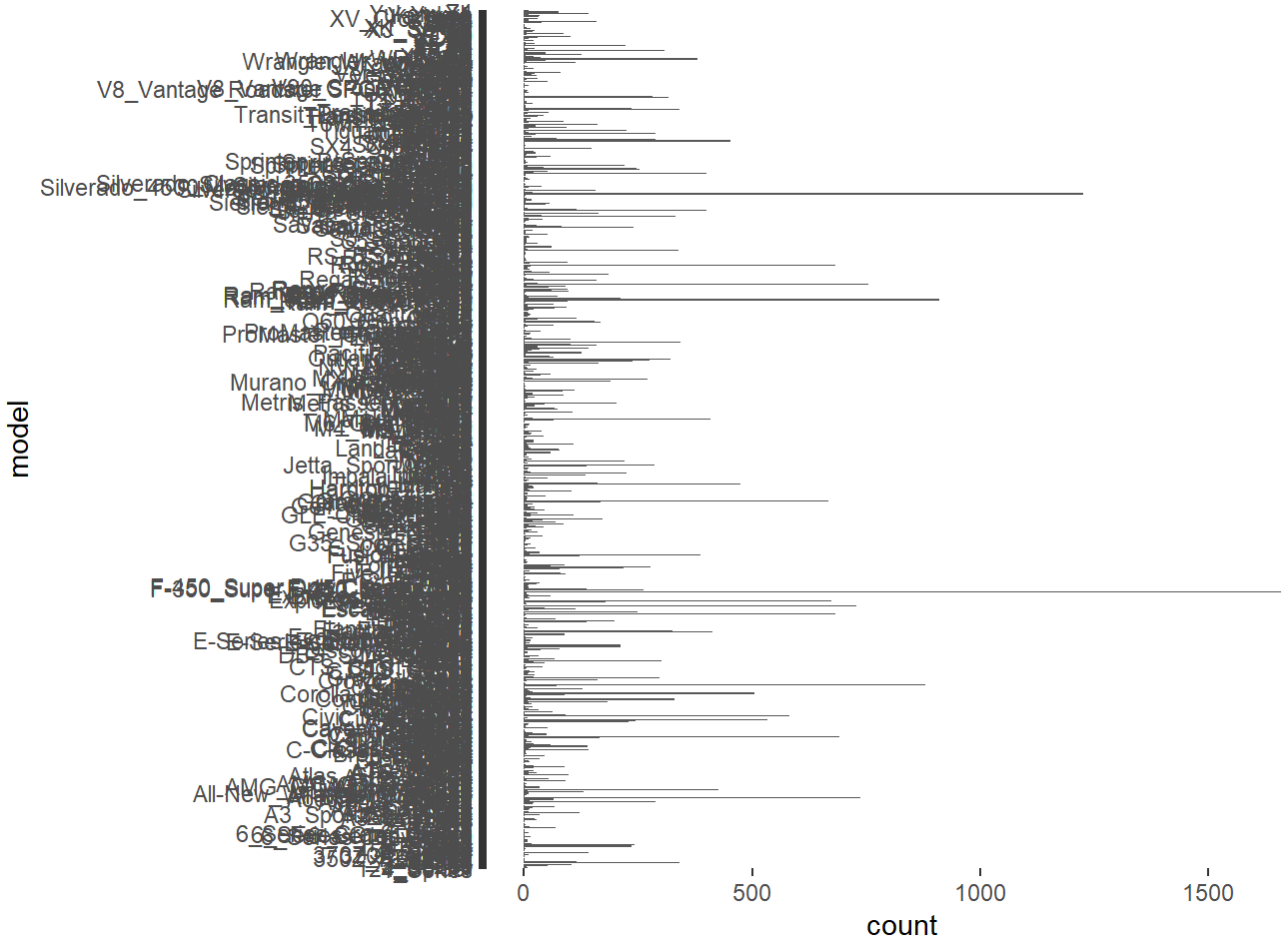
Now that I removed the variables I know I didn't want, I reworked the remaining ones into formats that are easier to work with. I turned every categorical variable into a factor, and every continuous variable into a numeric variable. These steps prevent hiccups in the recipe building process.

Code

Exploring the Data

The next step was taking a look at the data to get a better idea of what I was working with. I first wanted to know what the spread of my outcome variable, car model, looked like.

Code

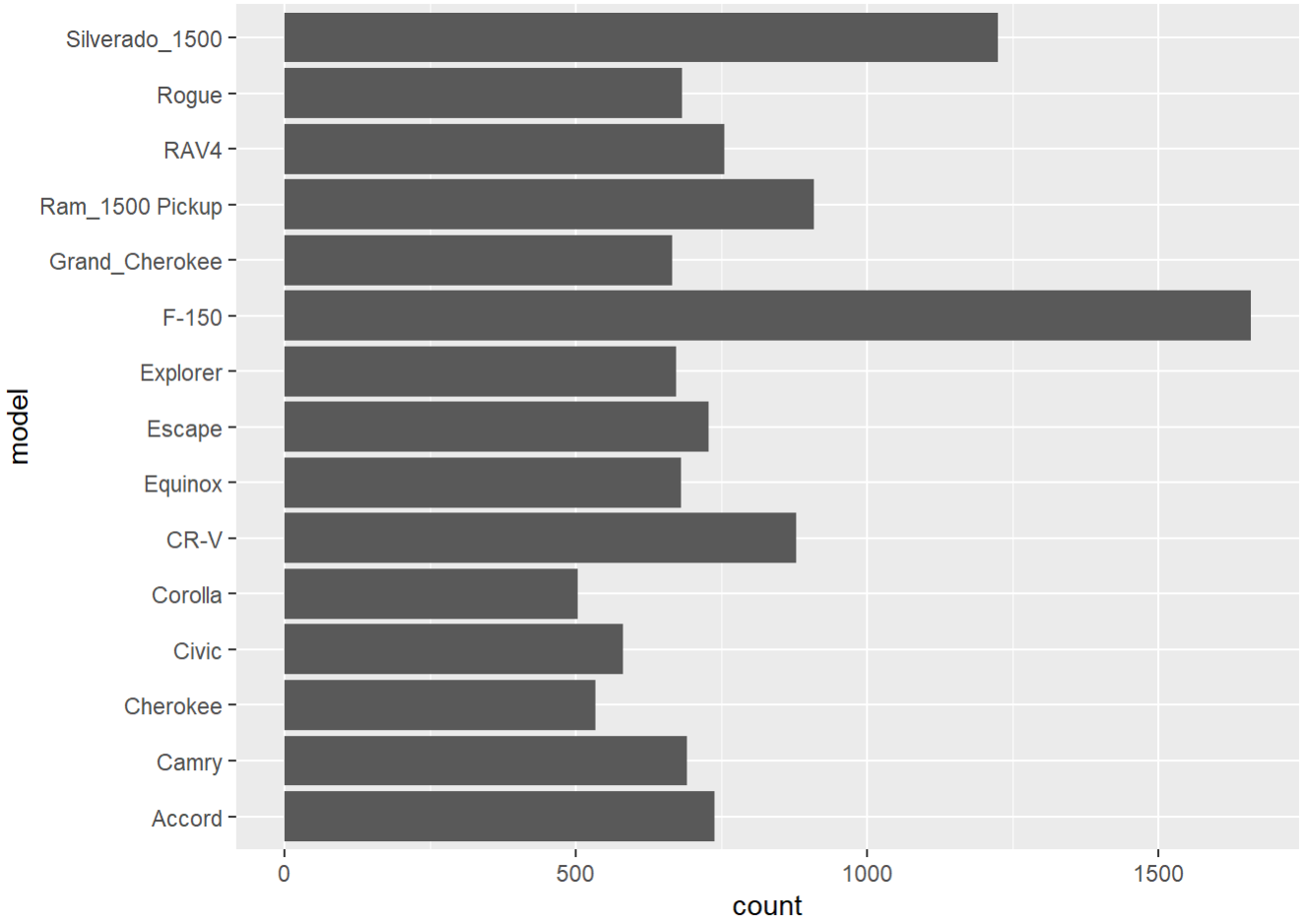


Yikes! The first thing you might notice is the jumble of model names attempting to serve as labels on the y-axis. But before I tried to fix this, I focused on the data displayed and saw that the majority of models contained very few data points. Statistical models tend to perform poorly if there is not an adequate amount of data, so I decided the best thing to do was limit my models to vehicles that appeared at least 500 times in the data set.

Code

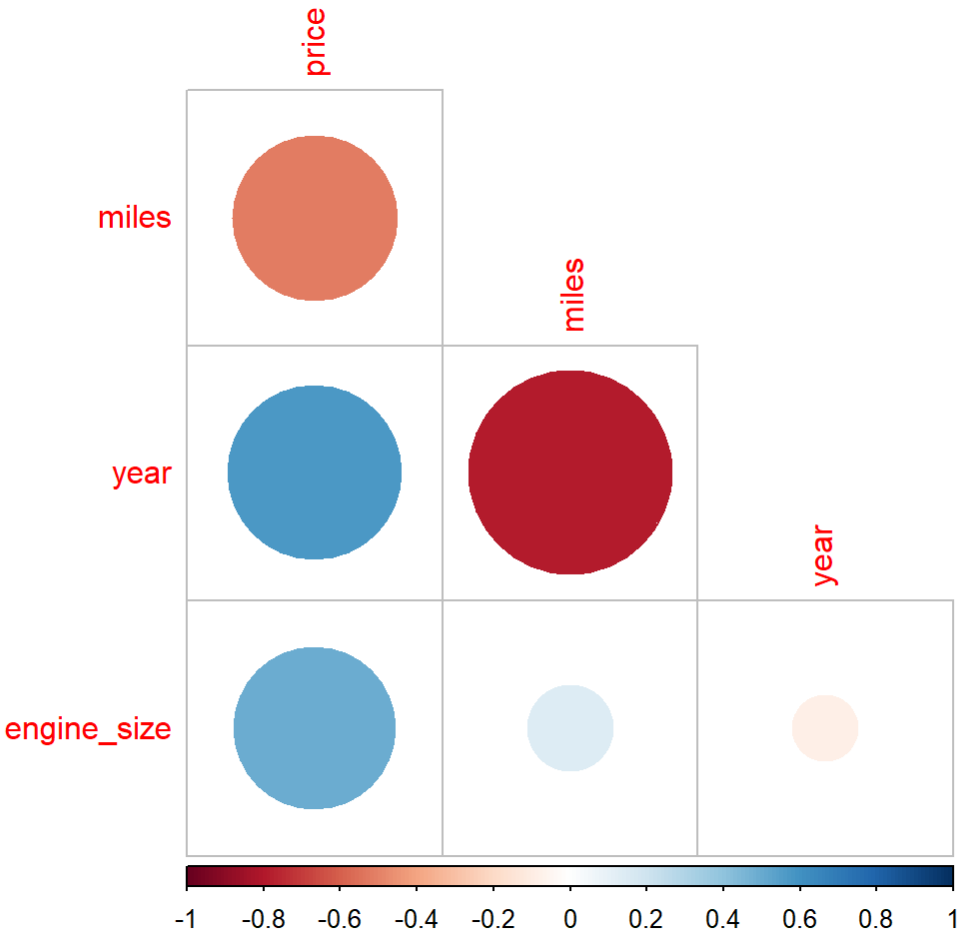
I saved these more popular vehicles in a data set called ‘common_models’, which contains 11,910 observations. Lets see what the spread of the car models looks like now.

Code



Much better. My next exploratory mission was to see how the continuous variables in my data set are correlated.

Code



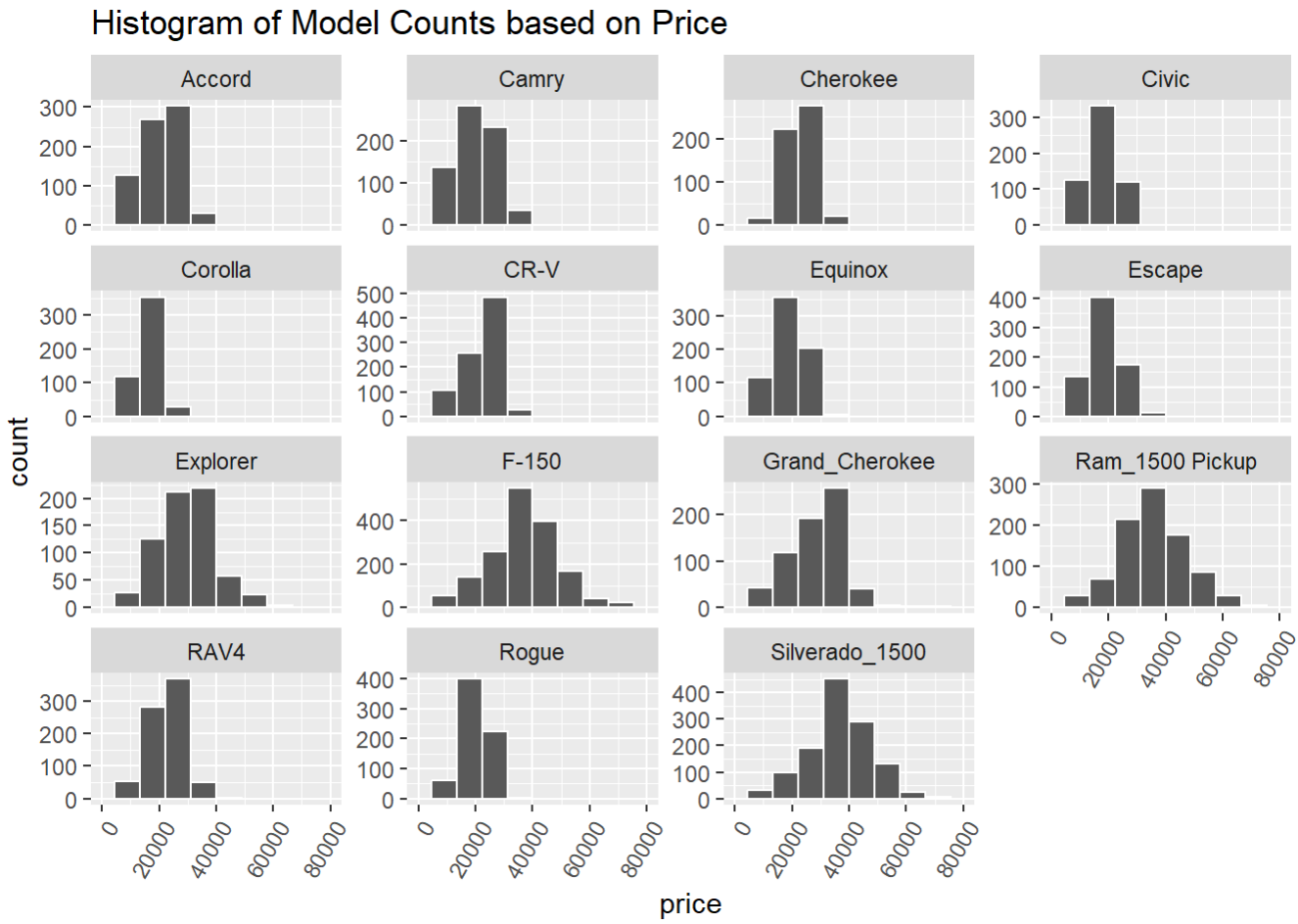
The correlation plot shows strong relationships between many of the continuous variables in this data set. Price has a strong negative correlation with miles, which makes sense - no car lasts forever, so people tend to prefer cars that have been driven less. There is a strong positive correlation between price and year, which means newer cars have a higher price. There is also a strong positive correlation between price and engine size; a car with a bigger engine will cost more. Car companies often tout the horsepower of a car, so assuming a larger engine means a more powerful vehicle, this relationship checks out too.

There is a very strong negative correlation between year and miles. This means newer cars (with a larger value for year) tend to have fewer miles driven on them. Sounds right to me.

There is a slight positive correlation between engine size and miles, and a slight negative correlation between engine size and year. These relationships are harder to explain. The first one is a mystery to me. As far as the second relationship, my best guess would be that cars have gotten smaller and more fuel efficient in recent years, which might mean engine sizes have decreased.

Finally, I took a look at how price and car model are related by plotting the counts for each car model in separate histograms. I set the range of prices to be from $0 to $80,000. Each histogram has 10 bins, each of which has a range of $8,000.

Code



The histograms for the Accord, CR-V, and Grand Cherokee are clearly skewed left. The histograms for the Civic, Equinox, Escape, F-150, Ram 1500, and Silverado are approximately normal. The other car models don't follow as clear distributions, but most are more or less normal or slighlty skewed left. The Explorer, the F-150, the Ram 1500, and the Silverado appear to have the widest range of prices, and are the only models that approach or surpass the $60,000 bin.

Splitting

The last thing standing between me and model training was splitting the data. I set my random seed to be able to reproduce the same split in the future, and divided my data in what is called the initial split.

Code

I chose to put 80% of the data into the training set and left the remaining 20% for the test set. I stratified the split on the outcome variable, model - this ensures that there are proportional amount of observations for every possible outcome in both sets. That way, even if there are models with fewer counts, they won't get accidentally overlooked during random split. The training set will be used to train all of the machine learning algorithms. It is necessary to have a separate testing set to see how well a model performs on data that it has never seen before. This is done to check for overfitting, where a model will specialize in predicting with the data it was trained on, but do a poor job making predictions with new data. The test set will only be used to check how well my best model performs. Until then, I will have no interaction with the test set to maintain model integrity.

That being said, it is almost always a good idea to test a model as you are working on it so you can tweak it and make improvements. How do you do that if you can't touch the test set? Split the training set once again!

Or, in this case, split it 5 times!

Code

```
## #  5-fold cross-validation using stratification
## # A tibble: 5 × 2
##   splits               id
##   <list>               <chr>
## 1 <split [7617/1905]> Fold1
## 2 <split [7617/1905]> Fold2
## 3 <split [7617/1905]> Fold3
## 4 <split [7618/1904]> Fold4
## 5 <split [7619/1903]> Fold5
```

This method is known as k-fold cross-validation. The general idea is to break the training set into ‘k’ smaller sets, known as folds. One fold will become a sort of mini testing set, known as an analysis set, and all the other folds are combined into a mini training set, known as the assessment set. Train a model on the assessment set, and test that model on the analysis set. Repeat this ‘k’ times, using a different fold as the analysis set each time, and take the model that performs the best on the analysis set. This effectively allows us to pick a model that avoids overfitting, before we take a crack at the test set from the initial split.

Time to Train Models

In R, every machine learning model is based on a recipe. In a recipe, you define what the outcome variable will be, what variables will be used as predictors, and what data to use. I knew my outcome variable is the car model, but I was a little uncertain as to which of the other variables I should use as predictors. For example, engine size might have useful information the model can use to make predictions in the future, but what if a specific engine size only showed up for one specific car model? My guess is that the machine learning model would predict that outcome every time that specific engine size was inputted, which undermines my goal of creating a tool that can make predictions based on relationships I don’t understand.

To ensure the machine learning models wouldn’t find any convenient shortcuts, I only used four predictor variables - price, miles, year, and transmission. I know for a fact that any car model can have any possible value for all of these predictors, so the ML models will have to dig a little deeper when making predictions.

In the recipe I dummy encoded all of the categorical predictors, which assigns specific values to each category for easier analysis. I then normalized all of the predictors, which prevents scaling issues from arising later on.

Code

Basic Model

Before I took a crack at training a more complicated ML model, I wanted to see how effective a very simple model would be. I decided to use Linear Discriminant Analysis, one of the most basic ways to make predictions in a classification problem with multiple possible outcomes. I trained it using the entire training set, because I was not planning on making any adjustments to this first model.

Code

I set up the model using a linear discriminant workflow, added my recipe, and fit it to the entire training set.

Code

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.245
```

Code

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.798
```

I checked two measures of model success, which are accuracy and ROC_AUC. Accuracy ranges from 0 (bad) to 1 (good), and ROC_AUC effectively ranges from 0.5 (bad) to 1 (good). This basic model had an accuracy of 0.245 and an ROC_AUC of 0.798 - not so good, as expected. Let's see if more complex models will do a better job.

First Model

Ok, time for the first serious model. I chose to use an elastic net, which is a linear combination of lasso and ridge regularization. These methods use a penalty value to decrease the variance of the ML model's prediction. Elastic net combines the two into a single model, and chooses the weight of each with the mixture value.

First, I created the elastic net model and set the penalty and mixture values equal to tune(). This allowed me to specify a range of possible values for those inputs. Every possible combination of those inputs are tried during training (and repeated across 5 different folds due to cross validation). This makes for some looooong training times, but the result can be a very effective model.

Code

Here I created a grid of possible ranges for the penalty and mixture inputs, and specified that I want 10 levels (equally spaced out values) chosen from those ranges.

Code

Finally, it's time to train a model using cross validation and tuning. In this case, I have 10 levels of 'penalty', 10 levels of 'mixture', and 5 folds, which means 10*10*5 = 500 models are about to get trained! As I said before, cross validation and tuning can take a long time, and this is the reason why. To avoid spending that time in the future, I saved the ML models to a separate file, where I can import the results directly rather than train them all again.

Code

Now that I saved the elastic net models, I simply read them in as shown below.

Code

I then viewed a table to see how the best elastic net models performed, based on their ROC_AUC.

Code

```
## # A tibble: 6 × 8
##   penalty mixture .metric .estimator  mean     n std_err .config
##     <dbl>   <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1 0.00001   0.667 roc_auc hand_till  0.817     5 0.00140 Preprocessor1_Model061
## 2 0.00001   0.556 roc_auc hand_till  0.817     5 0.00141 Preprocessor1_Model051
## 3 0.00001   0.778 roc_auc hand_till  0.817     5 0.00139 Preprocessor1_Model071
## 4 0.00001   0.889 roc_auc hand_till  0.817     5 0.00138 Preprocessor1_Model081
## 5 0.00001   0.444 roc_auc hand_till  0.817     5 0.00140 Preprocessor1_Model041
## 6 0.00001   1     roc_auc hand_till  0.817     5 0.00137 Preprocessor1_Model091
```

I selected the best elastic net model, added it to a new, finalized workflow, and fit it to the entire training set.

Code

Then I checked final elastic net model's performance on the training data.

Code

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.281
```

Code

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.820
```

Accuracy: 0.281 ROC_AUC: 0.820

Still not good. But both measures are higher than the basic model, so things are heading in the right direction.

Second Model

Up next is a random forest model. This is a very popular ML model, and I expected it to be the most effective for my purpose. Below, I set up a random forest workflow, and prepare to tune three hyperparameters: mtry, trees, and min_n.

Code

For tuning, I set up another grid specifying the ranges of each hyperparameter and the number of levels. With 3 hyperparameters, 8 levels, and 5 folds, there will be 8*8*5 = 2560 models trained!

Code

I tuned the model and saved the results so I wouldn't have to repeat the training process.

Code

Then I read in the results.

Code

Once again, I took a peak at the best performing models.

Code

```
## # A tibble: 512 × 9
##     mtry trees min_n .metric .estimator  mean     n  std_err .config
##    <int> <int> <int> <chr>   <chr>      <dbl> <int>    <dbl> <chr>
##  1     2   258    12 roc_auc hand_till  0.840     5 0.000706 Preprocessor1_Mode…
##  2     2   300    12 roc_auc hand_till  0.840     5 0.000677 Preprocessor1_Mode…
##  3     2   217    12 roc_auc hand_till  0.840     5 0.000791 Preprocessor1_Mode…
##  4     2   300    10 roc_auc hand_till  0.840     5 0.000769 Preprocessor1_Mode…
##  5     2    92    12 roc_auc hand_till  0.840     5 0.000790 Preprocessor1_Mode…
##  6     2   258    10 roc_auc hand_till  0.840     5 0.000604 Preprocessor1_Mode…
##  7     2   175    12 roc_auc hand_till  0.840     5 0.000822 Preprocessor1_Mode…
##  8     2   217    10 roc_auc hand_till  0.839     5 0.000882 Preprocessor1_Mode…
##  9     2   134    10 roc_auc hand_till  0.839     5 0.000639 Preprocessor1_Mode…
## 10     2   300     9 roc_auc hand_till  0.839     5 0.000629 Preprocessor1_Mode…
## # … with 502 more rows
```

Then I selected the best random forest model, fit it to a new workflow, and fit it to the entire training set.

Code

Time to check the model on the training set.

Code

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.726
```

Code

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.979
```

The random forest model got an accuracy of 0.725 and an ROC_ACU of 0.979. Not bad at all!

Third Model

The third model is a linear support vector machine, known as an SVM. I created a workflow, prepared to tune the hyperparameter cost, and create a grid of 10 values for that hyperparameter.

Code

Then I tuned the model and saved the results.

Code

I read the results from the saved file.

Code

Once again, I looked at all the best performing models.

Code

```
## # A tibble: 6 × 7
##       cost .metric .estimator  mean     n std_err .config
##      <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1  0.00310 roc_auc hand_till  0.750     5 0.00229 Preprocessor1_Model02
## 2  0.0312  roc_auc hand_till  0.749     5 0.00240 Preprocessor1_Model04
## 3  0.00984 roc_auc hand_till  0.749     5 0.00229 Preprocessor1_Model03
## 4  0.0992  roc_auc hand_till  0.747     5 0.00211 Preprocessor1_Model05
## 5  1       roc_auc hand_till  0.747     5 0.00226 Preprocessor1_Model07
## 6 10.1     roc_auc hand_till  0.747     5 0.00265 Preprocessor1_Model09
```

Then I selected the best linear SVM and fit it to the entire training set. Something about these SVMs really took a long time, so I saved the final fit to be read in.

Code

I saved the final linear SVM fit to save time while knitting.

Code

I did not run the accuracy/ROC_AUC code for any of the SVM models in my knitted file because it took a long time to run and made knitting difficult. I recorded the values in the text below.

Code

The linear SVM got an accuracy of 0.209 and an ROC_AUC of 0.753. This is worse than the most basic model.

The lectures for this course did not fully cover SVMs, so I couldn't analyze this result as well as I would have liked to. However, my best guess was that I used a linear SVM to try and model a non linear relationship, so I took another crack at it with a polynomial SVM. Ideally, I would have set the hyperparameter 'degree' to be tuned, but something about training the SVMs took an extremely long time, and my computer would not have been able to handle it. So I left 'cost' as the only hyperparameter to be tuned, and specified degree to be 4 (as opposed to 1, which was its value for the linear case). I chose 4 because there are four predictors in the recipe, but without more time to learn about SVMs in class, this choice was rather arbitrary.

Code

Once again, I trained and tuned some SVM models. The process is the same as for the linear case, with degree set to equal 4 instead of 1.

Code

I read in the saved data for the tuned polynomial SVMs

Code

I chose the best polynomial SVM based on ROC_AUC.

Code

I saved the final fit to save time in the knit process.

Code

Then I checked how the best polynomial SVM performed.

Code

The polynomial SVM got an accuracy of 0.262 and an ROC_AUC of 0.751. Barely better accuracy than the basic model, and slightly worse ROC_AUC than the basic model.

Changing the ‘degree’ value to 4 did not make much of a difference. Without the clearest understanding of SVMs, I was unable to figure out why this was the case. However, while researching and preparing these two SVM models, I saw that radial SVMs might be a better bet for non linear data. I decided against tuning dozens of radial SVMs with cross validation, but figured I would take a crack at one just to see what would happen.

I prepared a radial SVM and fit it to the entire training set. Although I was only training the one model and it was not too time intensive, I saved the model to a separate file to speed things up just slightly in the future.

Code

I read in the saved data for the radial SVM.

Code

The moment of SVM truth:

Code

The radial SVM got an accuracy of 0.327 and an ROC_AUC of 0.745. Slightly better accuracy than the basic model and the other SVMs, but slightly worse ROC_AUC those models too.

At this point I was quite fed up with SVMs and decided to move on.

Fourth Model

The last type of model I used was a boosted tree. I set up a boosted tree specification to be used in the workflow. The hyperparameters to be tuned were mtry, trees, and min_n, just like in the random forest models.

Code

In this case, I decided to increase the range of ‘trees’ to go up to 1000 (compared to the random forest tuning which went up to 300). These larger models would take longer to train, so I reduced the number of levels from 8 to 4 to reduce time consumption. This time around, I would only be training 4*4*5 = 320 models.

Code

I tuned the parameters across all the folds and saved the results.

Code

Then I read the results back in.

Code

The best models according to ROC_AUC are listed below.

Code

```
## # A tibble: 6 × 9
##    mtry trees min_n .metric .estimator  mean     n  std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>    <dbl> <chr>
## 1     1   340     8 roc_auc hand_till  0.836     5 0.000944 Preprocessor1_Model…
## 2     1   340    12 roc_auc hand_till  0.836     5 0.000689 Preprocessor1_Model…
## 3     1   340     5 roc_auc hand_till  0.835     5 0.00139  Preprocessor1_Model…
## 4     1   340     2 roc_auc hand_till  0.835     5 0.00126  Preprocessor1_Model…
## 5     1   670    12 roc_auc hand_till  0.833     5 0.000773 Preprocessor1_Model…
## 6     1   670     8 roc_auc hand_till  0.832     5 0.000874 Preprocessor1_Model…
```

It looks like the model with mtry = 1, trees = 340, and min_n = 8 performed the best. I then selected that model and used it to fit the entire training set.

Code

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.573
```

Code

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.942
```

The final boosted tree model had an accuracy of 0.575 and an ROC_AUC of 0.9432. Not amazing, and not as good as the random forest, but far better than any of the other options.

Best Model turned Test Model

Finally, it's time to pick the best model and try it on the test data - data the model has never seen. Based on both accuracy and ROC_AUC, the random forest model was the best by far.

Code

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.296
```

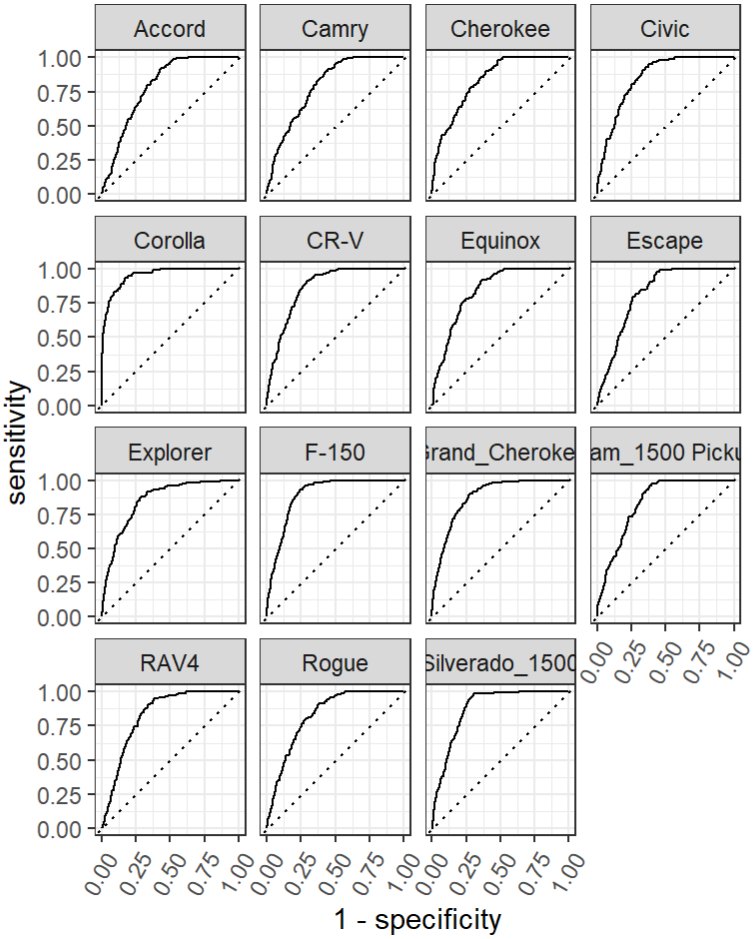
Code

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.841
```

When fit to the test data, the random forest model got an accuracy of 0.296 and an ROC_AUC of 0.840. This was a pretty substantial drop from the values the same random forest model got on the training set, which were 0.725 and 0.979 respectively. This suggests that despite the measures taken with k-fold cross validation, the model was overfit to the training data, and did not perform well when making predictions based on new data.

To dig a little deeper, I investigated the ROC_AUC curves for each possible value of the outcome variable. Ideally, each curve would approach the top left corner, which corresponds to an ROC_AUC of 1. A poorly performing model would have a curve along x=y, which corresponds to an ROC_AUC of 0.5

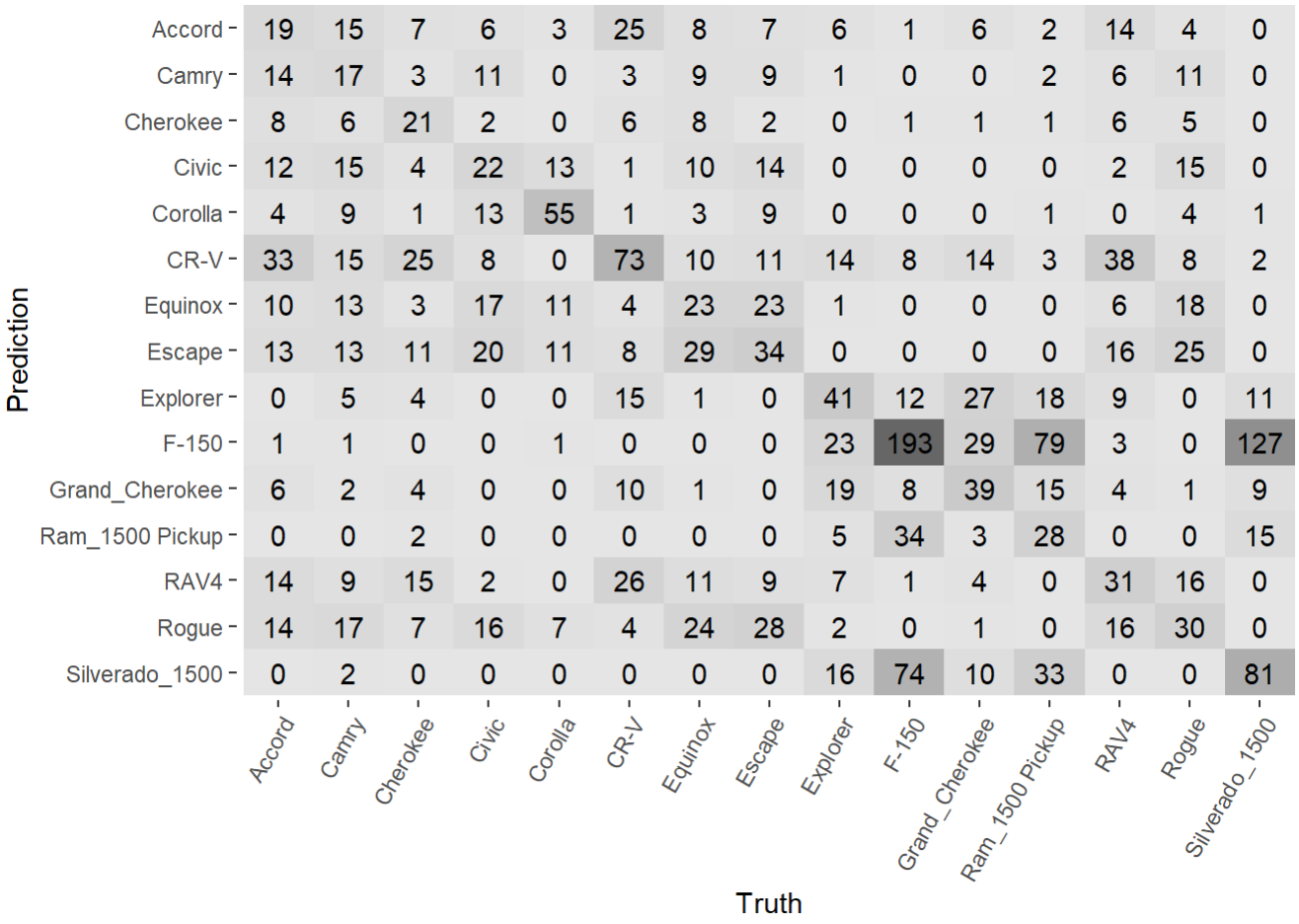
Code



For some car models, like the Corolla and the Ford F-150, the ROC curves look pretty good. However, this is not the case in all of the graphs - The Accord, the Camry, and the Escape are all pretty far off from that top left corner.

The last thing I checked was a confusion matrix, showing the model's predictions versus the true classification. A good model should have high numbers along the diagonal, which corresponds to the prediction being equivalent to the truth, and low numbers everywhere else.

Code



There is somewhat of a concentration along the diagonal, so it is clear the model was not blindly guessing. However, there are many scattered values across the board that are not insignificant. This matrix also agrees with the observations made from the ROC_AUC curves. The Corolla and the F-150 predictions seem more concentrated along the diagonal, but the Accord, the Camry, and the Escape are all over the place.

One thing that stood out to me was that the model made a lot of wrong predictions between the F-150, the Ram 1500, and the Silverado 1500 - it often chose one of these other two car models to predict the third. All three of these models are trucks, so there must be some relationship across the predictor variables that allow it to hone in on trucks in general, but struggle to make further distinctions.

Conclusion

In the end, I was unable to train a machine learning model that could effectively predict a used car's model given it's price, mileage, year, and transmission. The random forest model looked hopeful based on it's accuracy and ROC_AUC wen fit to the training data, but these values both dipped significantly when the model was fit to the testing data. This is a case of overfitting, where the model picked up on random patterns unique to the training data that do not generalize well to new data. I am a little shocked at the magnitude of the drop in performance, because cross validation is supposed to address this specific issue.

I am a bit disappointed, because I wanted to come away with a model that could effectively make predictions based on relationships that were not readily apparent. If I were to do this project over, I would try to incorporate more predictor variables in the recipe. I would also try to execute cross validation with more folds, to reduce the chance of overfitting.