Predicting University Admissions Rate from Directory and Financial Aid Data

Rohit Ravikumar

2/9/2021

Introduction

There are a wide variety of universities in the United States, from prestigious research institutions to local community colleges. They vary across a great many factors, but perhaps the most important differentiating criterion was a university's selectivity rate.

My question was simple: can a model be built using publicly available data that predicts the admission rate for a university using directory and financial aid information?

My model collects government data to answer this question, predicting a college's admissions rate with a residual mean squared error (RMSE) of only 10%.

The Data

```r
url_adm <- "https://raw.githubusercontent.com/rravikumar11/
gender_ratio_prediction/main/data/adm2018.csv"
url_hd <- "https://raw.githubusercontent.com/rravikumar11/
gender_ratio_prediction/main/data/hd2018.csv"
url_sfa <- "https://raw.githubusercontent.com/rravikumar11/
gender_ratio_prediction/main/data/sfa1718.csv"
adm <- read.csv(url_adm)
hd <- read.csv(url_hd)
sfa <- read.csv(url_sfa)
```

The data for this assignment were collected from the Integrated Postsecondary Education Data System (IPEDS), a national database of university data. To avoid inter-year complications, all data were selected from the year 2018. Specifically, I chose to focus on the following data sets:

SFA1718: Student Financial Aid and Net Price

HD2018: Institutional Characteristics (Directory information)

ADM2018: Admissions and Test Scores

For ADM2018, I choose to isolate only total applications and total admissions (to create my outcome variable for the model) and total enrollment (to check the distribution of schools by size to determine whether regularization was necessary).

For detailed information on all variables contained within these datasets, see the data dictionaries here: https://github.com/rravikumar11/admissions_prediction/tree/main/dictionaries

Cleaning the Data

One persistent problem with the IPEDS data was the proliferation of missing values that needed to be removed. I address this in full later on, but I begin my process of data cleaning and creating my outcome variable by removing its missing values, merging the datasets, and removing unnecessary variables:

```
edu <- adm %>% select(UNITID, APPLCN, ADMSSN, ENRLT) %>%
  inner_join(hd, by = "UNITID") %>% inner_join(sfa, by = "UNITID")

edu <- edu %>% filter(!is.na(APPLCN)) %>% filter(!is.na(ADMSSN))
adm_prob <- edu$ADMSSN/edu$APPLCN
edu$y <- adm_prob
edu <- edu %>% select(-UNITID, -APPLCN, -ADMSSN)



dim(edu)
```

```
## [1] 2006  720
```

The *edu* dataset has all the key predictor variables now, with 2006 observations and 720 variables, but it still needs a great deal of cleaning. First: the ZIP variable must be stripped of suffixes and converted to numeric:

```
edu <- edu %>% mutate(ZIP = as.numeric(substr(ZIP, 1, 5)))
```

Next, all unique identifiers for each institution, such as name, address, and EIN, as well as values like city that hardly vary at all (and therefore have limited value in a predictive model), need to be removed:

```
edu <- edu %>% select(-INSTNM, -IALIAS, -ADDR, -FIPS, -CHFNM, -GENTELE,
                      -EIN, -DUNS, -OPEID, -WEBADDR, -ADMINURL, -FAIDURL, -APPLURL,
                      -NPRICURL, -VETURL, -ATHURL, -DISAURL, -CITY, -STABBR, -CHFTITLE,
                      -F1SYSNAM, -COUNTYNM)

dim(edu)
```

```
## [1] 2006  698
```

Now we have 2006 observations of 698 variables.

Next, most variables in the IPEDS datasets have a corresponding variable tracking the imputation of a value for that observation: while useful for record-keeping purposes, these are entirely useless for predictive purposes. Conveniently, they all begin with the letter "X", and so can be easily removed:

```
edu <- edu %>% select(-contains("X"))

dim(edu)
```

```
## [1] 2006  375
```

Now we have 2006 observations of 375 variables.

Next, a crucial element of cleaning these data, as mentioned above, is removing the NAs. The SFA1718 dataset, in particular, has many variables mostly filled with NAs; this will inevitably weaken predictions if left in, as many observations would need to be dropped.

To address this, I count the number of NA values in each variable and, after some experimentation, drop any variable with more than 1300 NAs.

```r
nacount <- map(edu, ~sum(is.na(.)))
#count NAs per observation

fewna <- which(nacount > 1300)
#make index vector where NAs > 1300

edu <- na.omit(edu[,-fewna])
#remove corresponding vectors

dim(edu)
```

```
## [1] 814 233
```

Now we have 814 observations of 233 variables, with all NAs removed.

Finally, it is important to remove all remaining columns that were constant across all observations (and therefore, again, lack any predictive power):

```r
uniques <- sapply(edu, unique)
#all unique values in each variable

uniquelen <- sapply(uniques, length)
#number of unique values per variable

whichone <- which(uniquelen == 1)
#index vector from uniques

edu <- edu[,-whichone]
#removing uniques

dim(edu)
```

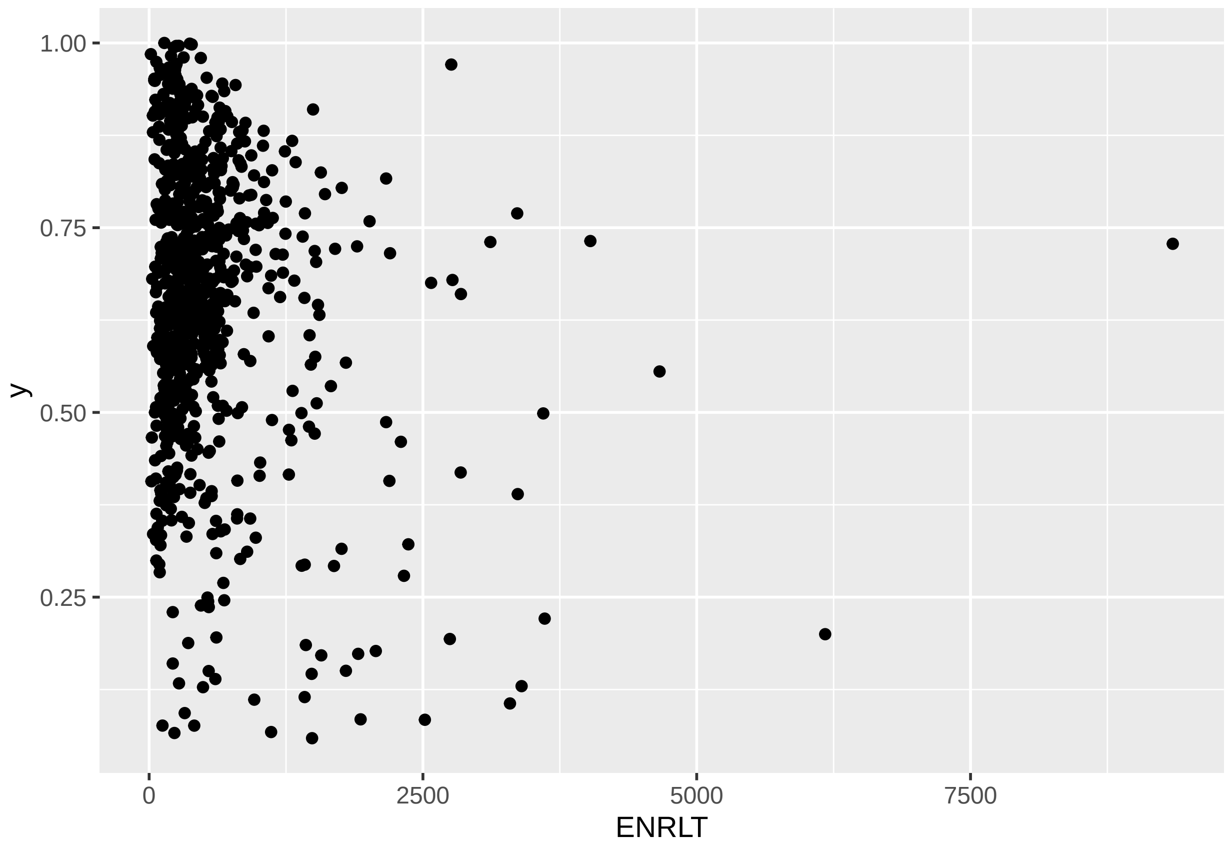
```
## [1] 814 220
```

The dataset is left with 814 observations of 220 variables: much smaller than when I began, but still with a sufficient number of observations to make a reasonably accurate prediction.

Regularization Check

Commonly, observations of different sizes will have higher variance at lower sizes, increasing the model's prediction of a small observation having an extreme value: regularization is a useful solution to this problem. To test whether regularization would be necessary for this analysis, I plotted the size of an institution (measured by its total enrollment) against its admissions rate:

```r
edu %>% ggplot(aes(ENRLT, y)) + geom_point()
```



As should be clear from the above graph, values are clustered around relatively low enrollment numbers: however, while a slight trend does exist between enrollment numbers and admission rates, no substantial increase in variance for smaller institutions is evident. As such, variables in this model are not regularized.

Splitting Data into Subsets

Three different subsets of the data are defined in this model:

edu_train: The dataset (the largest of the three) used to train the data and generate models. Contains 488 observations.

edu_test: The dataset used to compare different models and choose the best one. Contains 326 observations.

edu_validation: The final dataset, used only at the end when the model is fully determined to check accuracy. Contains 164 observations.

In order to ensure the training set has enough observations to accurately fit the data, I place more data in the training set, less in the test set, and even less in the validation set. The validation set in particular has only enough data to ensure a meaningful final validation test.

```r
set.seed(1, sample.kind = "Rounding")
validation_index <- createDataPartition(edu$y, times = 1, p = 0.2, list = FALSE)
edu_validation <- edu[validation_index,]
edu_model <- edu[-validation_index,]
#creating the validation subset
```

```
set.seed(1, sample.kind = "Rounding")
test_index <- createDataPartition(edu_model$y, times = 1, p = 0.4, list = FALSE)
edu_test <- edu[test_index,]
edu_train <- edu[-test_index,]
#creating the train and test subsets
```

Defining RMSE Calculation Function

Throughout this analysis, I will be calculating the residual mean squared error (RMSE) for various models: this is roughly the average error for a given prediction made by a model. A function to find it is defined here:

```
calc_RMSE <- function(y_hat, y) {
  sqrt(mean((y_hat - y)^2))
}
```

Establishing a Baseline with OLS

The most fundamental predictive algorithm for a continuous response variable is OLS (ordinary least squares) regression. While a model with so many predictors and so few observations is not entirely appropriate for OLS regression, I include it here for completeness and to establish a baseline RMSE that a more sophisticated model should beat.

Using the built-in lm function in R, I regress all predictors on the outcome y using the edu_train set and calculate the RMSE using the edu_test set.

```
train_ols <- lm(y ~ ., data = edu_train)
y_hat_ols <- predict(train_ols, edu_test)
#creating a vector of predicted values from model

RMSE_ols <- calc_RMSE(y_hat_ols, edu_test$y)
#comparing predicted values to actual
```

model	rmse
OLS	0.2449899

Our baseline RMSE is roughly 0.245: in other words, when using an OLS model to predict the admissions rate of a university using directory and financial aid information, the result will be about 24.5% off on average. This is already considerably better than simply guessing, but this model can clearly be improved significantly.

Using Random Forest to Identify Important Predictors

A model with 220 predictors is not appropriate for many useful algorithms, including simple ones like OLS and more sophisticated ones like *k*-nearest neighbors. Many of the variables in this model have little predictive power on our outcome y and can be safely removed to improve the model's accuracy.

In such a situation, one powerful tool is the random forest algorithm, which averages out the results of many regression trees to provide a smoothed model.

To identify the most important variables in the model, I use the random forest algorithm as a parameter of the train function, with all variables as predictors:

```
set.seed(1, sample.kind = "Rounding")
train_rf <- train(y ~ ., method = "rf", data = edu_train)
y_hat_rf <- predict(train_rf, edu_test)
#creating a vector of predicted values from model

RMSE_rf <- calc_RMSE(y_hat_rf, edu_test$y)
#comparing predicted values to actual
```

model	rmse	tune
OLS	0.2449899	–
RF	0.1408370	mtry = 110

As we can see, our model is already substantially improved, with an RMSE of 0.1408. It is also worth noting, as seen above, that the model selected $mtry = 110$ (the number of variables randomly sampled at each split of each tree).

Our primary interest here, though, is in identifying the most important variables in the model. Here are the top 20:

```
imp <- varImp(train_rf)$importance
imp <- imp %>% arrange(desc(Overall))
#extracting and ordering important variables

kable(head(imp, 20))
```

	Overall
GRN4A22	100.00000
ANYAIDP	81.08910
AIDFSIP	74.81520
AGRNT_P	53.59096
LATITUDE	42.58324
GRN4A32	34.46546
GRN4A21	31.91629
NPT452	26.78012
GRN4A12	26.21974
GRN4A20	19.59560
IGRNT_P	18.91179
ZIP	18.24729
FLOAN_P	16.02866
NPT451	15.79836
GRN4A31	14.46218
C18UGPRF	13.63312
OLOAN_A	13.24023
LONGITUD	13.07086
NPT450	10.75950
LOAN_P	10.58921

The variables are defined below:

GRN4A22: Average amount of grant and scholarship aid awarded, income level (30,001-48,000), 2017-18.
ANYAIDP: Percent of full-time first-time undergraduates awarded any financial aid.
AIDFSIP: Percent of full-time first-time undergraduates awarded any loans to students or grant aid from federal state/local government or the institution.
AGRNT_P: Percent of full-time first-time undergraduates awarded federal, state, local or institutional grant aid.
LATITUDE: Latitude location of institution.
GRN4A32: Average amount of grant and scholarship aid awarded, income level (48,001-75,000), 2017-18.
GRN4A21: Average amount of grant and scholarship aid awarded, income level (30,001-48,000), 2016-17.
NPT452: Average net price (income over 110,000)-students awarded Title IV federal financial aid, 2017-18.
GRN4A12: Average amount of grant and scholarship aid awarded, income level (0-30,000), 2017-18.
GRN4A20: Average amount of grant and scholarship aid awarded, income level (30,001-48,000), 2015-16.
IGRNT_P: Percent of full-time first-time undergraduates awarded institutional grant aid.
ZIP: ZIP code.
FLOAN-P: Percent of full-time first-time undergraduates awarded federal student loans.
NPT452: Average net price (income over 110,000)-students awarded Title IV federal financial aid, 2016-17.
GRN4A31: Average amount of grant and scholarship aid awarded, income level (48,001-75,000), 2016-17.
C18UGPRF: Carnegie Classification 2018: Undergraduate Profile.
OLOAN_A: Average amount of other student loans awarded to full-time first-time undergraduates.
LONGITUD: Longitude location of institution.
NPT452: Average net price (income over 110,000)-students awarded Title IV federal financial aid, 2017-18.
LOAN_P: Percent of full-time first-time undergraduates awarded student loans.

Many of the above variables are more or less what one would expect; unsurprisingly, financial aid availability and distribution strongly predicts a school's admissions rate. More surprising is the predictive power of the school's location, measured in latitude, longitude, and zip code.

Also of note is the Carnegie Classification Undergraduate Profile, a measure of the proportion of students attending full-time vs. part-time and how many students have transferred from another institution. This also makes sense as a predictor of admissions rate: one would assume a more competitive school would have students more willing to attend full-time .

Filtering Out Unimportant Predictors

I proceed by removing all predictors other than the 20 most important from the dataset. Doing so will reduce the risk of overfitting to the training data and allow the use of a wider variety of machine learning algorithms. For consistency, I remove these variables from all three datasets (validation, test, and training), creating new *select* datasets with only 20 predictors:

```r
edu_validation_select <- edu_validation %>%
  select(GRN4A22,ANYAIDP,AIDFSIP,AGRNT_P,LATITUDE,GRN4A32,GRN4A21,
         NPT452,GRN4A12,GRN4A20,IGRNT_P,ZIP,FLOAN_P,NPT452,GRN4A31,
         C18UGPRF,OLOAN_A,LONGITUD,NPT452,LOAN_P,y)

edu_test_select <- edu_test %>%
  select(GRN4A22,ANYAIDP,AIDFSIP,AGRNT_P,LATITUDE,GRN4A32,GRN4A21,
         NPT452,GRN4A12,GRN4A20,IGRNT_P,ZIP,FLOAN_P,NPT452,GRN4A31,
         C18UGPRF,OLOAN_A,LONGITUD,NPT452,LOAN_P,y)

edu_train_select <- edu_train %>%
  select(GRN4A22,ANYAIDP,AIDFSIP,AGRNT_P,LATITUDE,GRN4A32,GRN4A21,
         NPT452,GRN4A12,GRN4A20,IGRNT_P,ZIP,FLOAN_P,NPT452,GRN4A31,
         C18UGPRF,OLOAN_A,LONGITUD,NPT452,LOAN_P,y)
```

Now the process of trying out a wider variety of algorithms can begin.

OLS and RF Revisited

With significantly fewer predictors in the model, OLS is more appropriate as a method of prediction. I apply a simple linear regression model and compare the results to the previous ones here:

```r
train_ols_select <- lm(y ~ ., data = edu_train_select)
y_hat_ols_select <- predict(train_ols_select, edu_test_select)
#creating a vector of predicted values from model

RMSE_ols_select <- calc_RMSE(y_hat_ols_select, edu_test_select$y)
#comparing predicted values to actual
```

model	rmse	tune
OLS	0.2449899	–
RF	0.1408370	mtry = 110
OLS (select)	0.1536187	–

The OLS model has improved substantially after removing most of the model's predictors, with an RMSE of 0.1536; however, it still falls short of the full random forest model.

The next step is to revisit the random forest algorithm with the reduced predictor dataset. This time, I use the tuneGrid parameter to tune the model for different values of the *mtry* parameter: I used the values 1 to 10.

The results are as follows:

```r
set.seed(17, sample.kind = "Rounding")
train_rf_select <- train(y ~ ., method = "rf", data = edu_train_select,
                         tuneGrid = data.frame(mtry = c(1,10)))

y_hat_rf_select <- predict(train_rf_select, edu_test_select)
#creating a vector of predicted values from model

RMSE_rf_select <- calc_RMSE(y_hat_rf_select, edu_test_select$y)
#comparing predicted values to actual
```

model	rmse	tune
OLS	0.2449899	–
RF	0.1408370	mtry = 110
OLS (select)	0.1536187	–
RF (select)	0.1536187	mtry = 1

With an RMSE of 0.1536, the reduced parameter random forest model has the same calculated accuracy as the reduced parameter OLS model; notably, it is considerably inferior to the all-predictor random forest model. Notably, 1 was selected as the optimal *mtry* value.

Next, we move onto two other algorithms.

k-nearest neighbors Algorithm

The *k*-nearest neighbors algorithm takes the average value of the *k* nearest values to each object to create a predictive model. While the *k*-NN algorithm struggles with large numbers of predictors due to the curse of dimensionality, it is entirely appropriate for a model with only 20 predictors.

I apply the *k*-NN algorithm to the reduced-predictor dataset, tuning *k* for values between 20 and 30. The results are as follows:

```r
set.seed(28, sample.kind = "Rounding")
train_knn_select <- train(y ~ ., method = "knn", data = edu_train_select,
                          tuneGrid = data.frame(k = seq(20, 30, 1)))

y_hat_knn_select <- predict(train_knn_select, edu_test_select)
#creating a vector of predicted values from model

RMSE_knn_select <- calc_RMSE(y_hat_knn_select, edu_test_select$y)
#comparing predicted values to actual
```

model	rmse	tune
OLS	0.2449899	–
RF	0.1408370	mtry = 110
OLS (select)	0.1536187	–
RF (select)	0.1536187	mtry = 1
k-NN (select)	0.1490925	k = 22

Identifying an optimal number of neighbors at 22, the *k*-NN model has an RMSE of 0.149; notably, it is superior to all models thus far except for the full-predictor random forest model.

Cubist Algorithm

The Cubist algorithm is another tree-based algorithm, dividing data into subsets and applying linear regressions to each branch to smooth the regression prediction at the tree's terminal node. The Cubist algorithm has two parameters that can be tuned:

neighbors: Similarly to *k*-NN, this identifies n most similar cases to an object in the training data, predicting a value based on the average of the nearest neighbors.
committees: This creates n rule-based models, each of which predicts a value; the values are then averaged for the model's prediction.

I begin by applying the Cubist algorithm to the full dataset:

```r
set.seed(5, sample.kind = "Rounding")
train_cubist <- train(y ~ ., method = "cubist", data = edu_train,
                             tuneGrid = data.frame(
                               committees = seq(16,24,2), neighbors = c(0:4)))

y_hat_cubist <- predict(train_cubist, edu_test)
#creating a vector of predicted values from model

RMSE_cubist <- calc_RMSE(y_hat_cubist, edu_test$y)
#comparing predicted values to actual
```

model	rmse	tune
OLS	0.2449899	–
RF	0.1408370	mtry = 110
OLS (select)	0.1536187	–
RF (select)	0.1536187	mtry = 1
k-NN (select)	0.1490925	k = 22
Cubist	0.1441380	committees = 16 / neighbors = 0

With an RMSE of 0.144, the Cubist model surpasses the OLS and *k*-NN models in accuracy, but falls short of the random forest models.

We now apply the same algorithm to the select data:

```r
set.seed(5, sample.kind = "Rounding")
train_cubist_select <- train(y ~ ., method = "cubist", data = edu_train_select,
                             tuneGrid = data.frame(
                               committees = seq(16,24,2), neighbors = c(0:4)))

y_hat_cubist_select <- predict(train_cubist_select, edu_test_select)
#creating a vector of predicted values from model

RMSE_cubist_select <- calc_RMSE(y_hat_cubist_select, edu_test_select$y)
#comparing predicted values to actual
```

model	rmse	tune
OLS	0.2449899	–
RF	0.1408370	mtry = 110
OLS (select)	0.1536187	–
RF (select)	0.1536187	mtry = 1
k-NN (select)	0.1490925	k = 22
Cubist	0.1441380	committees = 16 / neighbors = 0
Cubist (select)	0.1446417	committees = 16 / neighbors = 0

Using the select data has reduced our RMSE; however, even with the reduced data, the Cubist algorithm is still superior to everything except for the full predictor random forest model.

Ensemble Algorithm

Of the algorithms used so far, the random forest algorithm with reduced predictors has the lowest RMSE. Before simply concluding that random forest is the best algorithm to use, though, it is worth determining whether a combination of the above models has superior predictive power.

To ensemble the previous algorithms, I use the caretEnsemble package; this package can fit multiple different models to the same dataset and generate a variety of meta-models using those predictions.

Here I utilize the select predictor version of the dataset to ensemble the OLS, random forest, *k*-NN, and Cubist algorithms. I also input all parameters from the above models that were found through tuning.

```r
set.seed(16, sample.kind = "Rounding")
ensemble_select_model_list <- caretList(y ~ ., data = edu_train_select, tuneList = list(
  rf = caretModelSpec(method = "rf", tuneGrid = data.frame(mtry = 1)),
  ols = caretModelSpec(method = "lm"),
  knn = caretModelSpec(method = "knn", tuneGrid = data.frame(k = 22)),
  cubist = caretModelSpec(method = "cubist", tuneGrid = data.frame(
    committees = 16, neighbors = 0))
 )
)
#Generate an ensemble list of models

y_hat_ensemble_select <- predict(ensemble_select_model_list, edu_test_select)
#creating a vector of predicted values from ensemble

RMSE_ensemble_select <- calc_RMSE(y_hat_ensemble_select, edu_test_select$y)
```

model	rmse	tune
OLS	0.2449899	–
RF	0.1408370	mtry = 110
OLS (select)	0.1536187	–
RF (select)	0.1536187	mtry = 1
k-NN (select)	0.1490925	k = 22
Cubist	0.1441380	committees = 16 / neighbors = 0
Cubist (select)	0.1446417	committees = 16 / neighbors = 0
Ensemble (select)	0.1467551	–

Though the ensemble, with an RMSE of 0.1467, is superior to many of the above models, its RMSE is higher than the Cubist and full predictor random forest models.

I conclude that the random forest full predictor model is the most accurate of the above, and proceed to the final step.

Testing Final Accuracy Against Validation Data Set

To conclude the analysis, I test my final model, the full predictor random forest model, against the validation data set kept separate until this stage. The results are as follows:

```
y_hat_final <- predict(train_rf, edu_validation)

RMSE_final <- calc_RMSE(y_hat_final, edu_validation$y)
```

RMSE
0.0982946

Surprisingly, the accuracy when using the validation set is higher than with the test set: an RMSE of 0.098, or about 10%. In other words, when using this predictive model to estimate the admission rate of a university, the prediction will be about 10% off reality on average. This is a substantial improvement over the initial OLS model.

Conclusions and Further Research to Do

While certainly a pleasant surprise, it cannot be ignored that the RMSE from the validation set was significantly lower than that of the test set. Modifying the model in response would invite the possibility of p-hacking, so my model stands as constructed; nevertheless, the discrepancy must be addressed.

I suspect that the discrepancy is largely due to the relatively small nature of my dataset. While I tried to feed the validation set a small amount of data so as to keep my training set robust, it seems possible that I overcorrected and put too little data in the validation set; this led to non-representative datasets.

The issue of insufficient data is a common one that I have encountered before when conducting independent research. If given more time, access to better records, and perhaps funding in the future, I feel that this model could be strengthened considerably.

In particular, many possibly insightful predictors had to be removed entirely due to having too many omitted values. Further digging could reveal resources that could be used to fill in the gaps here.

In addition, while the limited availability of data to just a few years made a time series component seem largely unnecessary in this case, more years available could allow for a robust predictive model that includes the year in question on top of everything else. By taking the year as a possible additional input, this could even allow projecting admissions rates into the future.

Despite the issues that arose in the course of the analysis, I am pleased with the results I have obtained here. Education is an important field, and being able to predict results about the admissions process could certainly be valuable to potential college students, especially if I am able to address the above issues in the future.