Predicting Airbnb prices

24 nov 2019

Michael Thomsen and Emma Munk

Table of Contents

Link to Colab:
https://colab.research.google.com/drive/1zcGtft6Fo6y2RVnlSGfyKZfAQdrk5QUt?fbclid=IwAR3B--gL6wWkM_pNVnYaPeg9OVJ7jn8cjlxwMVxnLucb2t9n1QX64VndI6U

Introduction

In this project, we'll test six different models to see which one performs better, and we espcially want to analyse whether advanced deep learning models perform better than simple models. We'll analyse Airbnb prices in Stockholm, Sweden, where we'll include numeric, categorical and text data.

Data desciption

We got our data over Airbnb rooms from https://www.kaggle.com/liubacuzacov/stockholm-sweden-airbnb-listings. They got their data from http://insideairbnb.com/get-the-data.html. Here we load two different files, one for listings details and one for reviews. The one with listings details contains around 7800 observations and 106 variables. The one with reviews has around 119000 observations, and therefore we had to cut it down, but more on that later.

Data cleaning

Before starting working on the models, we have to do some data cleaning.

Load libaries

We start by cleaning the environment.

```
#Cleaning the environment
rm(list=ls())
```

And then we'll import Keras, which is essential for the anaysis. You have to use "install_keras" if you're installing Keras for the first time.

```
#devtools::install_github("rstudio/keras", force = TRUE)
#library(keras)
#install_keras()
```

And then we'll load a bunch of other packages.

```
#Loading packages
if (!require("pacman")) install.packages("pacman")

## Loading required package: pacman

pacman::p_load(knitr,
               readr,
               rmarkdown,
               tidyverse,
               tidytext,
               dplyr,
               broom,
               keras,
               drat,
               reticulate,
               caret,
               textstem,
               recipes,
               MLmetrics,
               e1071,
               GGally,
               ranger,
               nnet
               )
```

Data

<<<<<<< HEAD And now we can download the data from our own Github ======= Now we can download the data from the Github. >>>>>>> 733676b31d6153d3bfc0cf177023361535cbf5bb

```
listings = read_csv("https://raw.githubusercontent.com/emmamunk/M3-miniproject/master/listings_detailed.csv")

reviews = read_csv("https://raw.githubusercontent.com/emmamunk/M3-miniproject/master/reviews_detailed.csv")
```

After loading reviews, we had to cut it, as we had problems running some models with this much data. To get a samlpe, which wouldn't be biased, we choose the year 2015, as this year was around 10000 observations. We have to assume that this is representative for all reviews, as the models cannot run if we do it with all years.

```
reviews = reviews %>% mutate(year = str_detect(date, pattern = "2015")) %>% filter(year == TRUE) %>% select(c(-year))
```

Data preprocessing

In the variable listings, there's a lof of variables. We have choosen some of them, here id, zipcode, room type, accommodates, bathrooms, bedrooms, number of reviews, guist included and review score rating. We have choosen these, as we think these will help with the understanding of how price is defined. We have to make them all numeric and binary, as we'll have problems further in the assignment if we have categorically variables.

```
listings = listings %>% select(c(id, price, zipcode, room_type, accommodates, bathrooms, bedrooms, number_of_reviews, guests_included, review_scores_rating))
```

Airbnb price

Our dependent variables is Airbnb price. We have choosen to look at a Classification problem to see how good the models will perform in predicting the price of a room, where price is categorized into six classes.

The first thing, we'll do is making the price numeric and removing the dollar sign in front of it.

```
listings$price = as.numeric(gsub('[$,]', '', listings$price))
```

In this project, we want to look at catogorical variables. Here we set the max price to 5500 dollars and split by 500, so we'll get 11 intervals. We had some outliers with higher prices, but they only had 1 or 2 observations, so we decided to remove them. We removed outliers, where the maximum price is 5500 dollars.

```
listings$price_intervals = cut(listings$price, c(0, seq(500, 5500, by=500)))
```

Adn then we'll make the price intervals numeric.

```
listings$price_intervals = as.numeric(listings$price_intervals)
```

And now we have to change the names, as the models will not run to names, unfortunately. therefore we made two different new variables, one where they are classified as letters, running from A to F. And after that we made six classes running from number 1 to 6. Here everything above 3000 dollars is classified as F and 6, as the classes got smaller and smaller. We still have a very biased data set, where class 2 or B hav around 41 percent of the data, which may cause our models to perform worse compared to if it was equally distributed. First the price interval as letters.

```
#Price intervals as letters
listings$price_intervals_ml[listings$price_intervals == "1"] = "A"

## Warning: Unknown or uninitialised column: 'price_intervals_ml'.
```

```
listings$price_intervals_ml[listings$price_intervals == "2"] = "B"
listings$price_intervals_ml[listings$price_intervals == "3"] = "C"
listings$price_intervals_ml[listings$price_intervals == "4"] = "D"
listings$price_intervals_ml[listings$price_intervals == "5"] = "E"
listings$price_intervals_ml[listings$price_intervals == "6"] = "F"
listings$price_intervals_ml[listings$price_intervals == "7"] = "F"
listings$price_intervals_ml[listings$price_intervals == "8"] = "F"
listings$price_intervals_ml[listings$price_intervals == "9"] = "F"
listings$price_intervals_ml[listings$price_intervals == "10"] = "F"
listings$price_intervals_ml[listings$price_intervals == "11"] = "F"
```

And then as numbers.

```
#Price intervals as numbers
listings$price_intervals_nlp[listings$price_intervals == "1"] = 1

## Warning: Unknown or uninitialised column: 'price_intervals_nlp'.

listings$price_intervals_nlp[listings$price_intervals == "2"] = 2
listings$price_intervals_nlp[listings$price_intervals == "3"] = 3
listings$price_intervals_nlp[listings$price_intervals == "4"] = 4
listings$price_intervals_nlp[listings$price_intervals == "5"] = 5
listings$price_intervals_nlp[listings$price_intervals == "6"] = 6
listings$price_intervals_nlp[listings$price_intervals == "7"] = 6
listings$price_intervals_nlp[listings$price_intervals == "8"] = 6
listings$price_intervals_nlp[listings$price_intervals == "9"] = 6
listings$price_intervals_nlp[listings$price_intervals == "10"] = 6
listings$price_intervals_nlp[listings$price_intervals == "11"] = 6
```

And now we can remove price intervals and price.

```
listings = listings %>% select(c(-price_intervals, -price))
```

Zipcode

As with price, we'll make the zipcodes numeric.

```
listings$zipcode = as.numeric(gsub('[ ]', '', listings$zipcode))

## Warning: NAs introduced by coercion
```

And then we can remove NA's as we will not do more data manipulation for now. Then we can describe the variables more presicely.

```
listings = na.omit(listings)
```

Room type

There is all in all four different types of rooms, but shared rooms and hotel rooms were very small combined, so we decided only to look at entire home/apartments or private room

combined with shared rooms and hotel rooms, and define the variable as a binary variable. Here entire home/apartment is equal to 1 and private room, hotel room or shared room is equal to 0. And now we can define the binary room type.

```
listings$room_type = ifelse(listings$room_type == "Entire home/apt", 1, 0)
```

And look at the distibution.

```
table(listings$room_type)

##
##   0   1
## 1035 4707
```

Here we can see that 4707 of the rooms are entire houses or apartments, while 979 are

```
listings = na.omit(listings)
```

Accommodates

Accomoodates describe how many people the place is for.

```
table(listings$accommodates)

##
##   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
## 341 2293 964 1103 462 353  97  75  19  14   6   4   1   3   2   5
```

It goes from 1 to 16, where most have 2-4 people.

Bathrooms

Bathrooms describe the number of bathrooms available.

```
table(listings$bathrooms)

##
##   0 0.5   1 1.5   2 2.5   3 3.5   4 4.5   5
##   8  22 4452 687 448  68  44   6   1   2   4
```

Number of bathrooms goes from 0 to 5, where most have 1 bathrooms.

Bedrooms

Bedrooms describe the number of bedrooms available.

```
table(listings$bedrooms)

##
##   0   1   2   3   4   5   6  10
## 603 3365 958 519 234  51  10   2
```

Number of bathrooms goes from 0 to 10, where most have 1 bedrooms.

Number of reviews

Number of reviews describe the number of reviews given.

```
table(listings$number_of_reviews)
```

```
##
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
## 606 549 449 366 338 281 260 197 176 161 154 153 115 106  82  92  83  84  60  59
##  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40
##  57  58  53  42  57  31  39  43  34  26  37  29  22  26  21  18  23  24  18  25
##  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
##  14  14  18  17  17  16  12  10  23   8  12  14  17  11  11   7   9  10   6  10
##  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
##  16   9  11   5   5   6  10   6   5   4   8   7   9   6  10   8   6   9   5   5
##  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
##   8   2   2   6   5   5   8   4   2   6   5   5   3   2   7   2   3   2   5   1
## 101 102 103 104 105 106 107 108 109 110 111 112 115 116 117 118 119 121 122 123
##   2   2   7   6   3   5   3   1   5   2   2   1   1   3   5   1   5   3   5   3
## 125 126 127 129 132 133 134 135 136 137 139 141 142 145 146 147 148 150 152 153
##   2   1   6   2   1   2   1   4   1   4   2   3   2   1   2   2   1   3   1   1
## 155 157 160 161 163 164 165 166 167 168 170 171 173 174 175 176 177 178 183 184
##   1   2   1   1   1   2   1   2   1   1   1   1   3   1   1   1   1   1   2   1
## 186 189 190 192 193 195 196 203 204 205 206 207 208 209 211 212 224 226 227 229
##   1   1   1   2   2   2   2   1   2   1   1   1   1   1   1   1   1   2   1   1
## 230 236 240 241 242 247 249 254 255 257 262 266 269 278 279 283 284 286 287 290
##   1   1   1   1   1   1   1   1   1   2   1   2   1   2   1   1   1   1   1   1
## 297 301 302 304 306 309 310 348 367 376 398 404 421 425 435 473 508
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

As seen the number of reviews goes from 1 to 508, where it slowly decreases from 1 and forward.

Guest included

Guest included describes number of guest included in the price.

```
table(listings$guests_included)
```

```
##
##    1    2    3    4    5    6    7    8    9   10   16
## 4329  908  146  223   70   40    9   12    2    2    1
```

Number of bathrooms goes from 1 to 16, where most have 1 guest included.

Review scores rating

Review scores rating describes a score rating going from 0 to 100, where people could rate the rented rooms/houses.

```
table(listings$review_scores_rating)
```

```
##
##  20  40  55  60  63  66  67  70  71  73  74  75  76  77  78  79
##   4   5   1  30   1   1   1  13   1  12   1   4   4   4   6   2
##  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
## 179   1   6  20  31  39  28  72  76  71 244 122 128 306 229 332
##  96  97  98  99 100
## 378 433 494 228 2235
```

Number of bathrooms goes from 40 to 100, where 100 is the most common, which is a bit odd.

Id

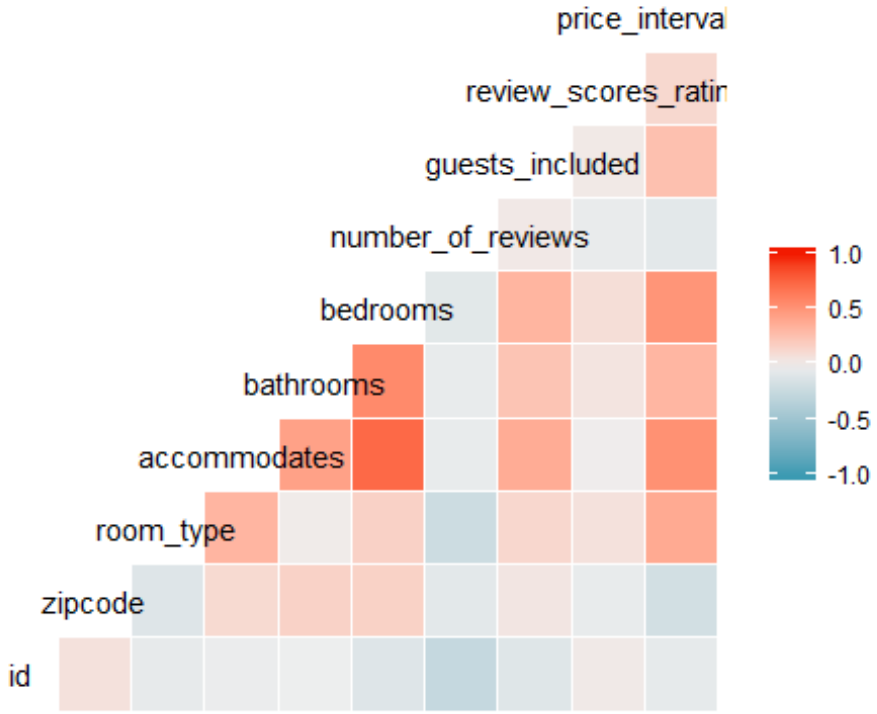
Here the number of identifikation numbers for the Airbnb rooms/apartments. Before any data cleaning and manipulations, there's 7854 identifation numbers. In the end there's 5742 back.

Correlation

Now, we want to run a correlation matrix of the different variables to see how much the correlate.

```
ggcorr(listings)
```

```
## Warning in ggcorr(listings): data in column(s) 'price_intervals_ml' are not
## numeric and were ignored
```



Here, it's especially interesting to see what variables price is correlated with. Here, especially accommodates and bedrooms are highly positive correlated and zipcode and number of reviews are slightly negative correlated. And now we can run how the listings look like. Here there's 5686 observatiosna and 13 variables, where some will be removed later on, but this is simple the easiest thing for us to keep them.

```
listings
```

```
## # A tibble: 5,742 x 11
##       id zipcode room_type accommodates bathrooms bedrooms number_of_revie~
##    <dbl>   <dbl>    <dbl>        <dbl>     <dbl>    <dbl>           <dbl>
##  1  42808   11347        0            2         1        1              64
##  2  53895   12838        0            2         1        1               7
##  3 145320   11853        0            2         1        1              72
##  4 155685   11739        1            2         1        2              22
##  5 164448   11864        0            2         1        1             304
##  6 170651   11737        1            4         1        1              32
##  7 206221   11639        0            2         1        1              78
##  8 220851   11341        0            1         1        1              45
##  9 242188   11864        0            1         1        1             286
## 10 259025   16371        0            1         1        1              60
## # ... with 5,732 more rows, and 4 more variables: guests_included <dbl>,
## #   review_scores_rating <dbl>, price_intervals_ml <chr>,
## #   price_intervals_nlp <dbl>
```

Supervised machine learning

We'll start by applying supervised machine learning from M1. We really want to see how well simple models perform against more complex models, and therefore it's interesting to start with the decision tree and the random forest to see how good they perform. Our benchmark is 41 percent, as that is what we get if you guess everything in class 2 or class B.

Training and test data set

Here, we want to look at the listing and we're choosing not to look at id and price intrvals for nlp.

```
listings_ml = listings %>% select(c(-id, -price_intervals_nlp))
```

We'll start by splitting the data into test and training. Here, we're doing a 75 percent split for the training data and a 25 percent split for the test data.

```
index = createDataPartition(y = listings_ml$price_intervals_ml, p = 0.75, list = FALSE)

training = listings_ml[index,]
test = listings_ml[-index,]
```

Here, we're using the recipes package. It lets you conveniently define a recipe of standard ML preprocessing tasks. Afterwards, we can just can use this recipe to bake our data, meaning performing all the steps in the recipe.

```
reci = recipe(price_intervals_ml ~ ., data = training) %>%
 step_center(all_numeric(), -all_outcomes()) %>%
 step_scale(all_numeric(), -all_outcomes()) %>%
 step_zv(all_predictors())

reci = reci %>% prep(data = training)
```

Now we just split again in predictors and outcomes, bake it all, and we are good to go.

```
x_train = bake(reci, new_data = training) %>% select(-price_intervals_ml)
y_train = training %>% pull(price_intervals_ml) %>% as.factor()

x_test = bake(reci, new_data = test) %>% select(-price_intervals_ml)
y_test =  test %>% pull(price_intervals_ml) %>% as.factor()
```

We now define a trainControl() object.

```
ctrl = trainControl(method = "cv",
                    number = 10,
                    classProbs = TRUE,
                    savePredictions = TRUE,
                    summaryFunction = multiClassSummary,
                    verboseIter = FALSE,
                    adaptive = list(min = 3,
```

```
                        alpha = 0.05,
                        method = "gls",
                        complete = TRUE),
              search = "random" )

metric = "Accuracy"
n_tune = 10
```

And then we can train our models

Decision tree

We start with a decision tree, where we're applying our trainObject().

```
fit_dt = train(x = x_train,
           y = y_train,
           trControl = ctrl,
           metric = metric,
           tuneLength = n_tune,
           method = "rpart")
```

And then printing it for

$$n_{tune} = 10$$

.

```
fit_dt
```

```
## CART
##
## 4308 samples
##    8 predictor
##    6 classes: 'A', 'B', 'C', 'D', 'E', 'F'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 3877, 3878, 3876, 3876, 3876, 3878, ...
## Resampling results across tuning parameters:
##
##   cp          logLoss   AUC       prAUC      Accuracy   Kappa
##   0.0000000000  3.608627  0.7389378  0.34644277  0.4786342  0.27290763
##   0.0004483551  2.799002  0.7476383  0.34074763  0.4876927  0.28159257
##   0.0004903884  2.799002  0.7476383  0.34074763  0.4876927  0.28159257
##   0.0005884661  2.684457  0.7482026  0.34079285  0.4944191  0.28719506
##   0.0007846214  2.451604  0.7540920  0.34093624  0.5006907  0.29370157
##   0.0025107885  1.360958  0.7603573  0.30755631  0.5076469  0.27874873
##   0.0047077285  1.312556  0.7402388  0.28156000  0.5046387  0.26546082
##   0.0094154570  1.274175  0.7345773  0.20418022  0.4886121  0.24885082
```

```
##  0.0145154963 1.277075 0.7298926 0.15933456 0.4772475 0.23591142
##  0.0784621420 1.438606 0.5618964 0.02256142 0.4303762 0.07530415
##  Mean_F1   Mean_Sensitivity Mean_Specificity Mean_Pos_Pred_Value
##  0.3852514 0.3594841       0.8778500        0.4020940
##  0.3776557 0.3642929       0.8790554        0.4011407
##  0.3776557 0.3642929       0.8790554        0.4011407
##  0.3765871 0.3651673       0.8799131        0.4033260
##  0.3756766 0.3678101       0.8809159        0.4055701
##  0.3750707 0.3404395       0.8779774        0.4204441
##  0.3617768 0.3215076       0.8757063        0.4955410
##       NaN  0.3006919       0.8734020              NaN
##       NaN  0.2883202       0.8711985              NaN
##       NaN  0.2107598       0.8432696              NaN
##  Mean_Neg_Pred_Value  Mean_Precision  Mean_Recall  Mean_Detection_Rate
##  0.8796684            0.4020940       0.3594841    0.07977237
##  0.8818768            0.4011407       0.3642929    0.08128212
##  0.8818768            0.4011407       0.3642929    0.08128212
##  0.8837203            0.4033260       0.3651673    0.08240319
##  0.8855488            0.4055701       0.3678101    0.08344845
##  0.8902090            0.4204441       0.3404395    0.08460782
##  0.8895879            0.4955410       0.3215076    0.08410646
##  0.8849673                  NaN       0.3006919    0.08143536
##  0.8802895                  NaN       0.2883202    0.07954124
##  0.8544103                  NaN       0.2107598    0.07172937
##  Mean_Balanced_Accuracy
##  0.6186671
##  0.6216741
##  0.6216741
##  0.6225402
##  0.6243630
##  0.6092084
##  0.5986070
##  0.5870470
##  0.5797593
##  0.5270147
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.002510789.
```

For $n_{tune} = 10$ we can see that the accuracy is between 0.47 and 0.507, which isn't good, but is alright. Our goal is to beat 0.41.

Random Forest

Let's now run a Random Forest.

```
fit_rf <- train(x = x_train,
          y = y_train,
```

```
          trControl = ctrl,
          metric = metric,
          tuneLength = n_tune,
          method = "ranger",
          importance = "impurity",
          num.trees = 25
          )
```

And print it.

```
fit_rf
```

```
## Random Forest
##
## 4308 samples
##    8 predictor
##    6 classes: 'A', 'B', 'C', 'D', 'E', 'F'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 3877, 3876, 3878, 3877, 3877, 3878, ...
## Resampling results across tuning parameters:
##
##   min.node.size  mtry  splitrule   logLoss   AUC       prAUC     Accuracy
##    2             2     extratrees  1.217142  0.7918279  0.3900967  0.5044240
##    3             5     gini        2.277244  0.7758509  0.3810789  0.4983716
##    8             4     gini        1.618687  0.7889622  0.3943604  0.5027810
##   12             1     gini        1.192715  0.8000897  0.3887409  0.5062774
##   17             7     extratrees  1.424271  0.7943768  0.3931011  0.5041839
##   18             6     gini        1.430489  0.7972177  0.3955844  0.5053278
##   18             8     extratrees  1.444485  0.7955334  0.3967212  0.5060298
##   20             5     gini        1.336729  0.7989785  0.4017759  0.5162553
##   20             7     extratrees  1.396776  0.7973369  0.3965958  0.5109054
##   20             8     gini        1.443464  0.7958791  0.3957133  0.5034787
##   Kappa     Mean_F1   Mean_Sensitivity  Mean_Specificity  Mean_Pos_Pred_Value
##   0.2819461  0.3838428  0.3572520        0.8784246        0.4284999
##   0.3017848  0.4018261  0.3875825        0.8825909        0.4247131
##   0.3000102  0.3958706  0.3807449        0.8820595        0.4594244
##   0.2641097       NaN  0.3225217        0.8749698        0.4275974
##   0.2971056  0.3928095  0.3709722        0.8812557        0.4439987
##   0.3031571  0.3911190  0.3780479        0.8826748        0.4475499
##   0.3009028  0.4047460  0.3753501        0.8820667        0.4380379
##   0.3164366  0.4031322  0.3829712        0.8848503        0.4292389
##   0.3065405  0.4087843  0.3779932        0.8829416        0.4283182
##   0.3019837  0.3921770  0.3744208        0.8826465        0.4286479
##   Mean_Neg_Pred_Value  Mean_Precision  Mean_Recall  Mean_Detection_Rate
##   0.8874319            0.4284999       0.3572520    0.08407066
##   0.8849971            0.4247131       0.3875825    0.08306193
```

```
##  0.8857334       0.4594244    0.3807449   0.08379684
##  0.8902591       0.4275974    0.3225217   0.08437957
##  0.8858260       0.4439987    0.3709722   0.08403065
##  0.8864170       0.4475499    0.3780479   0.08422130
##  0.8865152       0.4380379    0.3753501   0.08433830
##  0.8892054       0.4292389    0.3829712   0.08604256
##  0.8876973       0.4283182    0.3779932   0.08515091
##  0.8859278       0.4286479    0.3744208   0.08391311
##  Mean_Balanced_Accuracy
##  0.6178383
##  0.6350867
##  0.6314022
##  0.5987457
##  0.6261139
##  0.6303613
##  0.6287084
##  0.6339107
##  0.6304674
##  0.6285336
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 5, splitrule = gini
##  and min.node.size = 20.
```

Here the accuracy is between 0.477 and 0.507, which is much like the Decision Tree.

Evaluation of Supervised Machine Learning via final out-of-sample prediction

Now we have to test how well is does on the test data. First for the Decision Tree.

```
pred_dt = predict(fit_dt, newdata = x_test)
```

And let's print a confusion matrix.

```
confusionMatrix(pred_dt, y_test)
```

```
## Confusion Matrix and Statistics
##
##          Reference
## Prediction   A   B   C   D   E   F
##          A 137  33   5   1   1   2
##          B 108 506 234  63  14   9
##          C   1  31  45  41  17  11
##          D   0  14  27  37  14  18
##          E   0   0   0   0   0   0
##          F   0   2   7  23  10  23
##
```

```
## Overall Statistics
##
##               Accuracy : 0.5216
##                 95% CI : (0.4954, 0.5478)
##    No Information Rate : 0.4086
##    P-Value [Acc > NIR] : < 2.2e-16
##
##                  Kappa : 0.2955
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E Class: F
## Sensitivity          0.55691   0.8635  0.14151  0.22424  0.00000  0.36508
## Specificity          0.96465   0.4953  0.90950  0.94247  1.00000  0.96937
## Pos Pred Value       0.76536   0.5418  0.30822  0.33636      NaN  0.35385
## Neg Pred Value       0.91315   0.8400  0.78804  0.90332  0.96095  0.97078
## Prevalence           0.17155   0.4086  0.22176  0.11506  0.03905  0.04393
## Detection Rate       0.09554   0.3529  0.03138  0.02580  0.00000  0.01604
## Detection Prevalence  0.12483   0.6513  0.10181  0.07671  0.00000  0.04533
## Balanced Accuracy     0.76078   0.6794  0.52550  0.58336  0.50000  0.66722
```

Here the accuracy is slightly better than the training, going up to 0.526. Let's chech the Random Forest.

```
pred_rf = predict(fit_rf, newdata = x_test)
```

Adn again print a confusion matrix.

```
confusionMatrix(pred_rf, y_test)
```

```
## Confusion Matrix and Statistics
##
##          Reference
## Prediction   A   B   C   D   E   F
##          A 165  57   6   2   1   1
##          B  77 408 184  41   9   6
##          C   3 105  81  46  19  12
##          D   1  16  41  55  18  22
##          E   0   0   0   3   1   3
##          F   0   0   6  18   8  19
##
## Overall Statistics
##
##               Accuracy : 0.5084
##                 95% CI : (0.4821, 0.5346)
##    No Information Rate : 0.4086
##    P-Value [Acc > NIR] : 1.628e-14
```

```
##
##                Kappa : 0.308
##
##  Mcnemar's Test P-Value : 1.845e-12
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D  Class: E Class: F
## Sensitivity          0.6707   0.6962  0.25472  0.33333 0.0178571  0.30159
## Specificity          0.9436   0.6262  0.83423  0.92277 0.9956459  0.97666
## Pos Pred Value       0.7112   0.5628  0.30451  0.35948 0.1428571  0.37255
## Neg Pred Value       0.9326   0.7489  0.79709  0.91413 0.9614576  0.96819
## Prevalence           0.1715   0.4086  0.22176  0.11506 0.0390516  0.04393
## Detection Rate       0.1151   0.2845  0.05649  0.03835 0.0006974  0.01325
## Detection Prevalence   0.1618   0.5056  0.18550  0.10669 0.0048815  0.03556
## Balanced Accuracy      0.8072   0.6612  0.54447  0.62805 0.5067515  0.63912
```

A bit worse, bu much like the Decision Tree.

Simple Neural Network

Now we want to check how well a Simple Neural Network will do. This is the simplest neural network, you can do.

```
fit.nnet = train(price_intervals_ml ~ ., training,
          method='nnet',
          trace = FALSE)
```

Adn let's print it.

```
fit.nnet
```

```
## Neural Network
##
## 4308 samples
##    8 predictor
##    6 classes: 'A', 'B', 'C', 'D', 'E', 'F'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 4308, 4308, 4308, 4308, 4308, 4308, ...
## Resampling results across tuning parameters:
##
##   size  decay  Accuracy   Kappa
##   1     0e+00  0.4103798  0.000000e+00
##   1     1e-04  0.4103798  0.000000e+00
##   1     1e-01  0.4435146  1.096845e-01
##   3     0e+00  0.4103798  0.000000e+00
```

```
##   3    1e-04  0.4103798   0.000000e+00
##   3    1e-01  0.4588194   1.628032e-01
##   5    0e+00  0.4103546  -3.316069e-05
##   5    1e-04  0.4135437   1.071965e-02
##   5    1e-01  0.4700498   1.992501e-01
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were size = 5 and decay = 0.1.
```

Overall, this is really performing well.

Evaluation of Simple Neural Network via final out-of-sample prediction

Let's see how well it does on the test data.

```
pred_nnet = predict(fit.nnet, newdata = x_test)
```

Adn we'll again print a confusion matrix.

```
confusionMatrix(pred_nnet, y_test)
```

```
## Confusion Matrix and Statistics
##
##          Reference
## Prediction   A   B   C   D   E   F
##          A 120 163  70  40   8   4
##          B   4  37  52  46  11  19
##          C   0   8   3   7  11   6
##          D 122 378 192  72  26  34
##          E   0   0   1   0   0   0
##          F   0   0   0   0   0   0
##
## Overall Statistics
##
##               Accuracy : 0.1618
##                 95% CI : (0.1431, 0.1819)
##    No Information Rate : 0.4086
##    P-Value [Acc > NIR] : 1
##
##                  Kappa : -0.0077
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D  Class: E Class: F
## Sensitivity          0.48780  0.06314 0.009434  0.43636 0.0000000  0.00000
```

```
## Specificity         0.76010  0.84434 0.971326  0.40741 0.9992743  1.00000
## Pos Pred Value      0.29630  0.21893 0.085714  0.08738 0.0000000      NaN
## Neg Pred Value      0.87755  0.56601 0.774839  0.84754 0.9609211  0.95607
## Prevalence          0.17155  0.40865 0.221757  0.11506 0.0390516  0.04393
## Detection Rate      0.08368  0.02580 0.002092  0.05021 0.0000000  0.00000
## Detection Prevalence  0.28243  0.11785 0.024407  0.57462 0.0006974  0.00000
## Balanced Accuracy     0.62395  0.45374 0.490380  0.42189 0.4996372  0.50000
```

On the test data, it is performing slightly better, but not as good as the Random Forest or Decision Tree.

NLP

Data preprocessing

As the data is very raw and messy, we now want to do some cleaning. We remove everything that isn't normal letters. Which is special characters, numbers and etc. Furthermore we will set all letters from the comments column to lower case.

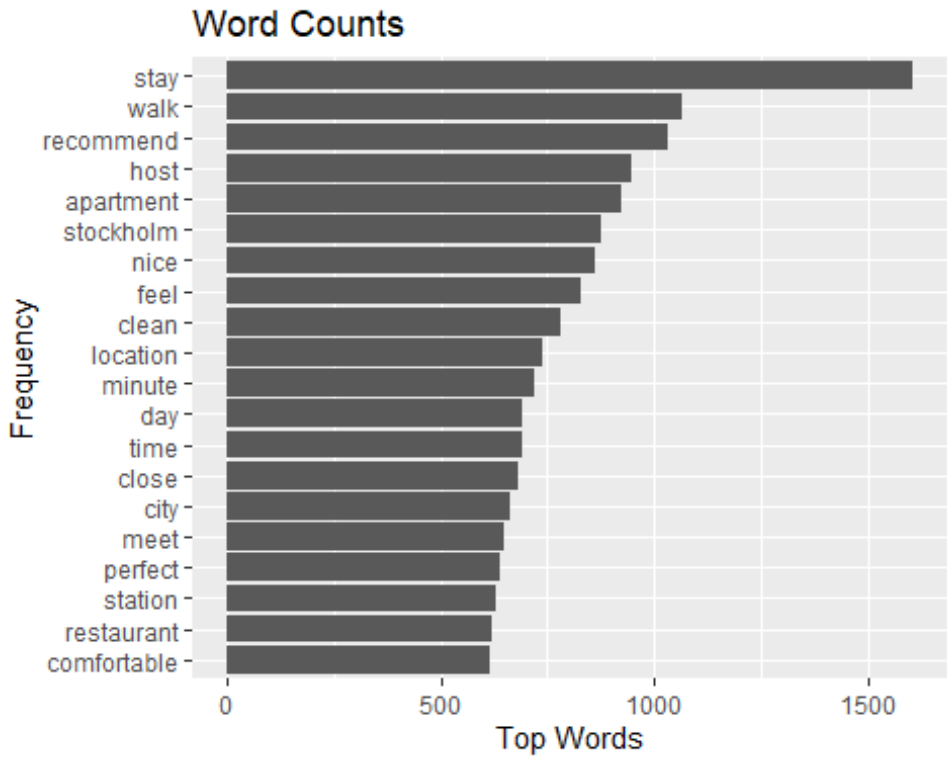
To clean up the data we are using lemmatization. The purpose of this is to not only analyze the exact word strings in the reviews, as this would include several possible forms of the words used. F. ex. think and thought. Instead we want to merge all possible forms of a word into it's root word. Lemmatization try and do so, by using detailed dictionaries which the algorithm looks trough to link a given word string back to it's root word. This is a more advanced method than stemming and should be beneficial in this report.

```
#Unnest the comments and lemmatize the words
reviews_tidy <- reviews %>%
 unnest_tokens(word, comments) %>%
 count(listing_id, word, sort = TRUE) %>%
 mutate(word = lemmatize_words(word))

#Defining the number of times a word is used the comments
reviews_tidy %>%
 count(word, sort = TRUE)

## # A tibble: 22,861 x 2
##    word        n
##    <chr>    <int>
##  1 be        4066
##  2 have      1874
##  3 good      1760
##  4 a         1650
##  5 stay      1606
##  6 the       1096
##  7 and       1073
```

```
##  8 walk      1065
##  9 place     1047
## 10 recommend  1030
## # ... with 22,851 more rows

#Defining own stopwords, that isn't relevant for the analysis
own_stopwords <- tibble(c("2","10","5","7","3","15"),
                        lexicon = "OWN")
#Removing theese homemade stopwords, but also cleaning for general stopwords such as the, as, a and  etc.
reviews_tidy = reviews_tidy %>%
 anti_join(stop_words %>% bind_rows(own_stopwords), by = "word")

#Removing all numbers and special characters and removing words who only contain one letter
reviews_tidy = reviews_tidy %>%
 mutate(word = word %>% str_remove_all("[^[:alnum:]]") ) %>%
  mutate(word = word %>% str_remove_all("[^a-zA-Z]")) %>%
 filter(str_length(word) > 1)
```

After cleaning up the data, we can now look at which words are most represented in the comments. To do this we define a variable called topwords, which will be used to visualize it through a table.

```
#Defining the the topwords after cleaning
topwords <- reviews_tidy %>%
 count(word, sort = TRUE)

#Table of topwords
topwords %>%
 top_n(20, n) %>%
 ggplot(aes(x = word %>% fct_reorder(n), y = n)) +
 geom_col() +
 coord_flip() +
 labs(title = "Word Counts",
     x = "Frequency",
     y = "Top Words")
```



We see that the individual tokens that are most frequent are words such as “apartment”, “stockholm” or names such as “clean”, “comfortable”, which describes how the Airbnb home is.

Sentiment analysis

Sentiment analysis refers to a use of text analysis to extract and identify subjective information, where it analyzises whether the words are positive or negative. In this section, we will be doing two sentiment analysis, first by identifying positive and negative words using the bing lexicon and after this using the afinn lexicon.

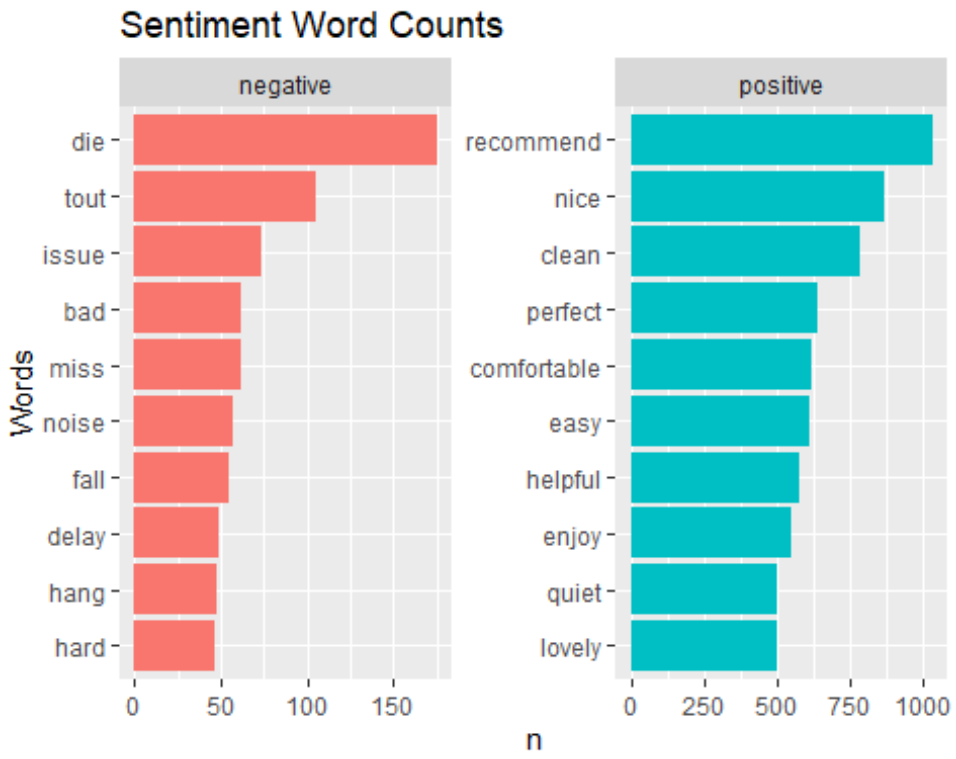
Bing

We wil start with the Bing lexicon. The Bing lexicon categorizes words in a binary fashion as positive or negative with no weighting. Here, we are using the function get_sentiment to get a specific sentiment lexicon and inner_join to join the lexicon with tokenized data.

Now we are plotting a word count, grouped by sentiment, showing the 10 most frequent negative and positive words.

```
sentiment_bing = reviews_tidy %>% inner_join(get_sentiments("bing"))

## Joining, by = "word"

sentiment_analysis <- sentiment_bing %>%
  filter(sentiment %in% c("positive", "negative"))
```

```
#Calculating the number of words in each sentiment
word_counts <- sentiment_analysis %>%
count(word, sentiment) %>%
group_by(sentiment) %>%
top_n(10, n) %>%
ungroup() %>%
mutate(
word2 = fct_reorder(word, n))

#Plotting the two sentiment with their respective words
ggplot(word_counts, aes(x = word2, y = n, fill = sentiment)) +
geom_col(show.legend = FALSE) +
facet_wrap(~ sentiment, scales ="free") +
coord_flip() +
labs(title ="Sentiment Word Counts",x ="Words")
```



We see that in general that negative words aren't represented so much as the positive words in the Airbnb comments. Which seems kind odd, then comparing to the expectation beforehand.

A Simple Recurrent Neural Network model

We start by changing the name of id in listings

```
listings = listings %>% rename(listing_id = id)
```

And then we can leftjoin the two data sets, here reviews and listings.

```
data = left_join(reviews, listings, by = "listing_id")
```

And then we can select to lok at listing id, price intervals and comments.

```
data = data %>% select(comments, price_intervals_nlp, listing_id) %>% na.omit()
```

Then we can split the datset into test and training, here 25 percent and 75 percent.

```
index1 = createDataPartition(y = data$price_intervals_nlp, p = 0.75, list = FALSE)

training = data[index1,]
test = data[-index1,]
```

As the data is very raw and messy, we now want to do some cleaning. We remove everything that isn't normal letters. Which is special characters, numbers and etc. Furthermore we will set all letters from the comments column to lower case.

To clean up the data we are using lemmatization. The purpose of this is to not only analyze the exact word strings in the reviews, as this would include several possible forms of the words used. F. ex. think and thought. Instead we want to merge all possible forms of a word into it's root word. Lemmatization try and do so, by using detailed dictionaries which the algorithm looks trough to link a given word string back to it's root word. This is a more advanced method than stemming and should be beneficial in this report.

```
# Training data
training_tidy = training %>%
 unnest_tokens(word, comments) %>%
 count(listing_id, word, sort = TRUE) %>%
 mutate(word = lemmatize_words(word)) %>%
  mutate(word = word %>% str_remove_all("[^[:alnum:]]") ) %>%
  mutate(word = word %>% str_remove_all("[^a-zA-Z]")) %>%
 filter(str_length(word) > 1) %>%
 anti_join(stop_words %>% bind_rows(own_stopwords), by = "word")

#Test data
test_tidy = test %>%
 unnest_tokens(word, comments) %>%
 count(listing_id, word, sort = TRUE) %>%
 mutate(word = lemmatize_words(word)) %>%
  mutate(word = word %>% str_remove_all("[^[:alnum:]]") ) %>%
  mutate(word = word %>% str_remove_all("[^a-zA-Z]")) %>%
 filter(str_length(word) > 1) %>%
 anti_join(stop_words %>% bind_rows(own_stopwords), by = "word")
```

We set max feautres to 10000, so it selects the first 10000 words.

```
max_features = 10000
```

And then we tokenize the text by max numbers, here 10000.

```
tokenizer_train = text_tokenizer(num_words = max_features)
```

And then we fit it.

```
tokenizer_train %>% fit_text_tokenizer(training_tidy$word)
```

And then to a sequence.

```
text_seqs_train = texts_to_sequences(tokenizer_train, training$comments)
```

And then for text.

```
tokenizer_test = text_tokenizer(num_words = max_features)
```

And again fit it.

```
tokenizer_test %>% fit_text_tokenizer(test_tidy$word)
```

And again to sequence.

```
text_seqs_test = texts_to_sequences(tokenizer_test, test$comments)
```

Then max length of words is 100.

```
maxlen = 100
```

And then we pad the sequences.

```
train_pad = pad_sequences(text_seqs_train, maxlen = maxlen)
test_pad = pad_sequences(text_seqs_test, maxlen = maxlen)
```

And look at the dimensions.

```
dim(train_pad)

## [1] 6838  100
```

And then we can run the model. Here a simple RNN with some dropouts to compensate for overfitting.

```
model = keras_model_sequential() %>%
 layer_embedding(input_dim = 10000, output_dim = 100) %>%
 layer_simple_rnn(units = 100) %>%
 layer_dropout(0.2) %>%
 layer_dense(units = 7, activation = "softmax")
```

And then we choose optimizer, loss and metrics.

```
model %>% compile(
 optimizer = "rmsprop",
```

```
  loss = "sparse_categorical_crossentropy",
  metrics = "accuracy")
```

And then we have to tranform the price intervals to look at it as numeric.

```
y = as.numeric(training$price_intervals_nlp)
y1 = as.numeric(test$price_intervals_nlp)
```

And then we can run the model. Batch size is how many samples to pass to our model at a time where epochs is how many times we look at the whole dataset. Here we choose 256 and 10 for it to not run forever.

```
trained_model = model %>% fit(
   x = train_pad,
   y = y,
   batch_size = 256,
   epochs = 10,
   validation_split = 0.25)
```

Adn then we can look at how it performs.

```
trained_model

## Trained on 5,128 samples (batch_size=256, epochs=10)
## Final epoch (plot to see history):
##        loss: 0.4039
##    accuracy: 0.8918
##    val_loss: 2.36
## val_accuracy: 0.2643
```

It does not perform well on the validation test set, and therefore it may be overfitting. We have tried to play around with the dropout, but it still doesn't perform better. If we had more time, it may be performing better, but only looking at reviews, can not predict the price well into six classes. At least it performs worse than our other models and it cannot beat our bench model of 41 percent.

Evaluation of a Recurrent Neural Network via final out-of-sample prediction

Let's look at how it performs on out-of-sample prediction.

```
metrics = model %>% evaluate(test_pad, y1)
```

And then we can print the metrics.

```
metrics

## $loss
## [1] 2.427557
```

```
##
## $accuracy
## [1] 0.2296882
```

As expected it performs badly on the test data as well, here around 33 percent.

Mixed Input Model

We wanted to experiment with a mixed input model, which is a model that draws from different data sources. Here, we're using numeric and categorical data from listings and text data from reviews. We want to make a model that combines those two. We start by defining a new data set with the variables we need. We have been using https://keras.rstudio.com/articles/functional_api.html?fbclid=IwAR2hqL5vif6bGDF_NddwvPE5_QhXlM-bUMk75yFLAq676ipuP35sRd44_LI as inspiration.

```
listings_mim = listings
data = left_join(reviews, listings_mim, by = "listing_id")

data_mim = data %>%
 select(c(-listing_id, -id, -date, -reviewer_name, -reviewer_id, -price_intervals_ml)) %>%
 na.omit()
```

Adn then we split the data set into test and training, here 25 percent and 75 percent. We also had to remove NA's again.

```
index2 = createDataPartition(y = data_mim$price_intervals_nlp, p = 0.75, list = FALSE)

training = data[index2,]
test = data[-index2,]

training = na.omit(training)
test = na.omit(test)

# Price intervals (y)
training_price = training$price_intervals_nlp
test_price = test$price_intervals_nlp

# Comments intervals
training_comments = training$comments
test_comments = test$comments
```

And now we're going in to bake the training data for the numeric and categorical variables.

```
reci_mim = recipe(price_intervals_ml ~ zipcode + room_type + accommodates + bathrooms + bedrooms +
number_of_reviews + guests_included + review_scores_rating, data = training)

reci_mim = reci %>% prep(data = training)
```

Now we just split again in predictors and outcomes, bake it all, and we are good to go.

```
x_train = bake(reci, new_data = training) %>% select(-price_intervals_ml)
y_train = training %>% pull(price_intervals_ml) %>% as.factor()

x_test = bake(reci, new_data = test) %>% select(-price_intervals_ml)
y_test =  test %>% pull(price_intervals_ml) %>% as.factor()
```

We now define a trainControl() object.

```
ctrl = trainControl(method = "cv",
                    number = 10,
                    classProbs = TRUE,
                    savePredictions = TRUE,
                    summaryFunction = multiClassSummary,
                    verboseIter = FALSE,
                    adaptive = list(min = 3,
                                    alpha = 0.05,
                                    method = "gls",
                                    complete = TRUE),
                    search = "random" )

metric = "Accuracy"
n_tune = 10
```

Unfortunately, we did not have time to apply the tokenization from the NLP, which we guessed would have optimzed the model, so we're just using a simple text tokenizer from the Keras package with max features of 10000.

```
max_features = 10000
tokenizer = text_tokenizer(num_words = max_features,
  filters = "!\"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n",
  lower = TRUE, split = " ", char_level = FALSE)
```

Here we update the tokenizer internal vocabulary based on training comments

```
keras::fit_text_tokenizer(tokenizer, training_comments)
```

Adnt hen making it a sequence.

```
tweet_tokenizer = keras::texts_to_sequences(tokenizer, training_comments)
```

Adn then the same for the test data.

```
keras::fit_text_tokenizer(tokenizer, test_comments)
```

And again, we're making it into a sequence

```
tweet_tokenizer2 = keras::texts_to_sequences(tokenizer, test_comments)
```

And then we're padding the sequence, here with a max length of 100 words.

```
x_train_nlp = pad_sequences(tweet_tokenizer, maxlen = 100)
x_test_nlp = pad_sequences(tweet_tokenizer2, maxlen = 100)
```

And now we're defining the model. Here there's two input models, the numeric and categorical data from the listings data set with eight variables, and the text data from the reviews data set with a max length of 100. the text data will be trained a bit before in a LSTM, while the numeric data won't. We have played around with number of dense layers and dropouts to compensate for overfitting and performance. With more time, we're sure you could make a model that'll perform better. For the exam, we'll bring an image to show of the mixed input model, we didn't have to time to make it for this.

```
# Numeric and categorical model
bi_input = layer_input(shape = c(8))



# Text model
main_input = layer_input(shape = c(100))

text_out = main_input %>%
 layer_embedding(input_dim = 10000, output_dim = 512, input_length = 100) %>%
 layer_lstm(units = 32)

# Mix layer
main_output = layer_concatenate(c(text_out, bi_input)) %>%
 layer_dropout(0.5) %>%
 layer_dense(units = 64, activation = 'relu') %>%
 layer_dense(units = 64, activation = 'relu') %>%
 layer_dense(units = 64, activation = 'relu') %>%
 layer_dropout(0.5) %>%
 layer_dense(units = 7, activation = 'softmax')

# Model
model = keras_model(
 inputs = c(main_input, bi_input),
 outputs = c(main_output)
)
```

Then we'll run a summary to show.

```
summary(model)

## Model: "model"
## _____
## Layer (type)          Output Shape     Param #  Connected to
## ==========================================================================
=
```

```
## input_2 (InputLayer)     [(None, 100)]     0
## _____________________________________________________________________
## embedding_1 (Embedding)   (None, 100, 512)  5120000  input_2[0][0]
## _____________________________________________________________________
## lstm (LSTM)               (None, 32)       69760    embedding_1[0][0]
## _____________________________________________________________________
## input_1 (InputLayer)      [(None, 8)]      0
## _____________________________________________________________________
## concatenate (Concatenate) (None, 40)       0       lstm[0][0]
##                                                    input_1[0][0]
## _____________________________________________________________________
## dropout_1 (Dropout)       (None, 40)       0       concatenate[0][0]
## _____________________________________________________________________
## dense_1 (Dense)           (None, 64)       2624    dropout_1[0][0]
## _____________________________________________________________________
## dense_2 (Dense)           (None, 64)       4160    dense_1[0][0]
## _____________________________________________________________________
## dense_3 (Dense)           (None, 64)       4160    dense_2[0][0]
## _____________________________________________________________________
## dropout_2 (Dropout)       (None, 64)       0       dense_3[0][0]
## _____________________________________________________________________
## dense_4 (Dense)           (None, 7)        455     dropout_2[0][0]
## =====================================================================
=
## Total params: 5,201,159
## Trainable params: 5,201,159
## Non-trainable params: 0
## _____________________________________________________________________
```

Adn choosing loss, optimizer and metrics.

```
model %>% compile(loss = 'sparse_categorical_crossentropy',
                  optimizer = 'RMSprop',
                  metrics = c('accuracy'))
```

And last we had to transform our x_train into a matrix.

```
x_train_ma = as.matrix(x_train)
x_test_ma = as.matrix(x_test)
```

And now we can run the model.

```
model_mim = model %>% fit(
  x = list(x_train_nlp, x_train_ma),
  y = training_price,
  epochs = 10,
  batch_size = 256,
  validation_split = 0.25
)
```

And then let's print it

```
model_mim
```

```
## Trained on 4,987 samples (batch_size=256, epochs=10)
## Final epoch (plot to see history):
##        loss: 0.8105
##    accuracy: 0.6972
##    val_loss: 1.477
## val_accuracy: 0.371
```

As with the last model, it also looks like it's overfitting. the accuracy is high, but it cannot replicate the good results for the validation, which lies at 0.43 and therefore only slightly beats out model.

Evaluation of a Mixed Input Model via final out-of-sample prediction

And then we can test how well it performs on out test data.

```
metrics = model %>% evaluate(list(x_test_nlp, x_test_ma), test_price)
```

And then we can print the metrics.

```
metrics
```

```
## $loss
## [1] 1.474213
##
## $accuracy
## [1] 0.4085193
```

It performs better than the only text based model, but cannot beat simpler models. With some tuning, it might perform better.

Conclusion

Overall the models perform with an accuracy spanding from 36 percent to 52 percent. Our goal was to beat an accuracy of 41 percent as that would be the results if you classified everything as class 2.

Unfortunately the models didn't perform as well as we thought, with the Decision Tree and Random Forest beating more complex models. Other models or finetuning of layers may change the result, but unfortunately we dind't have time to test every model our tune our models further. It often looks like it's overfitting, so if we could find a solution for this, the text based and deep learning models may perform better and beat out the simpler models.

We had some problems with how the distributions are. First of all, the price intervals are not equally distrbuted with class 2 being significat bigger than the rest of the groups. Ideally it should be at 16,6 percent but it takes up 41 percent. If the data was more equally distributed it may perform better. Secondly, we had to only look at the year 2015 as out models could not run, because Reviews had too much data.

Other variables may also have made out model perform better. Overall, we though it would perform better, but predicting half right for six different classes is alright.