Model Building Module

Robert Duc Bui

11/13/2021

```r
# Initialising packages and data import
library(dplyr)
library(tidyverse)
library(tidytext)

resReviewData <- read.csv2('yelpRestaurantReviews_sample_s21b.csv')
rrData <- resReviewData %>%
    filter(str_detect(postal_code, "^[0-9]{1,5}"))
```

Model Development

The original data frame is large, and can cause performance issues for training. Therefore, we will train our model on only a sampled subset of the data. We will be sampling without replacement for a subset of size 10000 total, and then splitting that subset into training and testing sets at a 75:25 ratio.

Note that for the sake of readable code and unified syntactic structure across disparate models, we will be using the `tidymodels` ecosystem of packages, specifically the `textrecipes` package for text preprocessing. The code below also outputs the dimensions of the train *and* test sets.

To retain our focus on text mining, we will not be including any data other than the review text itself as the sole independent variable, and the number of stars given in a review as the dependent variable.

```r
###############     MODEL PREPARATIONS: SAMPLING & SPLITTING

library(tidymodels)
library(textrecipes)

set.seed(8675309)
df <- sample_n(rrData, 10000) %>%
  transmute(stars = as.factor(starsReview),
            text = as.character(text))

set.seed(8675309)
splitkey <- initial_split(df, strata = stars)

df_train <- training(splitkey)
df_test <- testing(splitkey)

dim(df_train)
```

```
## [1] 7498    2
```

```
dim(df_test)
```

```
## [1] 2502    2
```

Now, we create a preprocessing workflow with `tidymodels` and `textrecipes`. Disregarding the dictionaries for now, we will perform the following preprocessing steps:

- `textrecipes::step_tokenize()` to tokenise the text column.
- `textrecipes::step_stem()` to perform stemming on the tokens. We would have preferred to use lemmatisation, but some back-end issues prevented us from calling `textrecipes::step_lemma()`'s necessary dependent libraries.
- `textrecipes::step_stopwords()` to remove stop-words.
- `textrecipes::step_tokenfilter()` to filter out token by certain criteria, listed below:
  - `min_times = 0.01` to filter out tokens that only appear in less than 1% of the corpus.
  - `percentage = T` to set `min_times` and `max_times` (where applicable) as percentages rather than nominal values.
  - `max_tokens = tune::tune()` to only keep a certain amount of most frequently-occurring tokens. Here we actually fix it to 1000, as during experimentation, we attempted to perform a grid search for the best max_tokens value, but ran into issues with `glmnet`'s FORTRAN layer, whose error codes are not well documented. Instead, we have decided to choose the overall best value from other models' grid search instead.
- `textrecipes::step_tfidf()` is the final preprocessing step, which outputs the term freq-inverse document frequency of each token. We are using TF*IDF to account for the fact that some words will inevitably appear very frequently across all star levels ("food", "lunch", "dinner" are an obvious examples).

```
###############     MODEL PREP: CREATING TIDYMODELS RECIPE + WORKFLOW

recipe <- recipe(
  stars ~ text,
  data = df_train
)

recipe <- recipe %>%
  step_tokenize(text) %>%
  step_stem(text) %>%
  step_stopwords(text) %>%
  step_tokenfilter(text,
                   min_times = 0,
                   max_times = .75,
                   percentage = T,
                   max_tokens = 1000) %>%
  step_tfidf(text)

wf <- workflow() %>%
  add_recipe(recipe)
```

After preprocessing, we have a document-term matrix of 7498 documents by n terms, n being whatever the result of our hyperparameter tuning process is. Below is a sample `dim()` call of the preprocessed data, where the `max_tokens` option has been set to 100, which results in a matrix of 7498 by 101 - with one column retaining the dependent variable.

```
## [1] 7498  101
```

In order to prevent our hyperparameter tuning process from resulting in overfitting, the final step of model-building is to create k-fold cross validation objects. For the sake of time and simplicity, we limit this to a 5-fold CV.

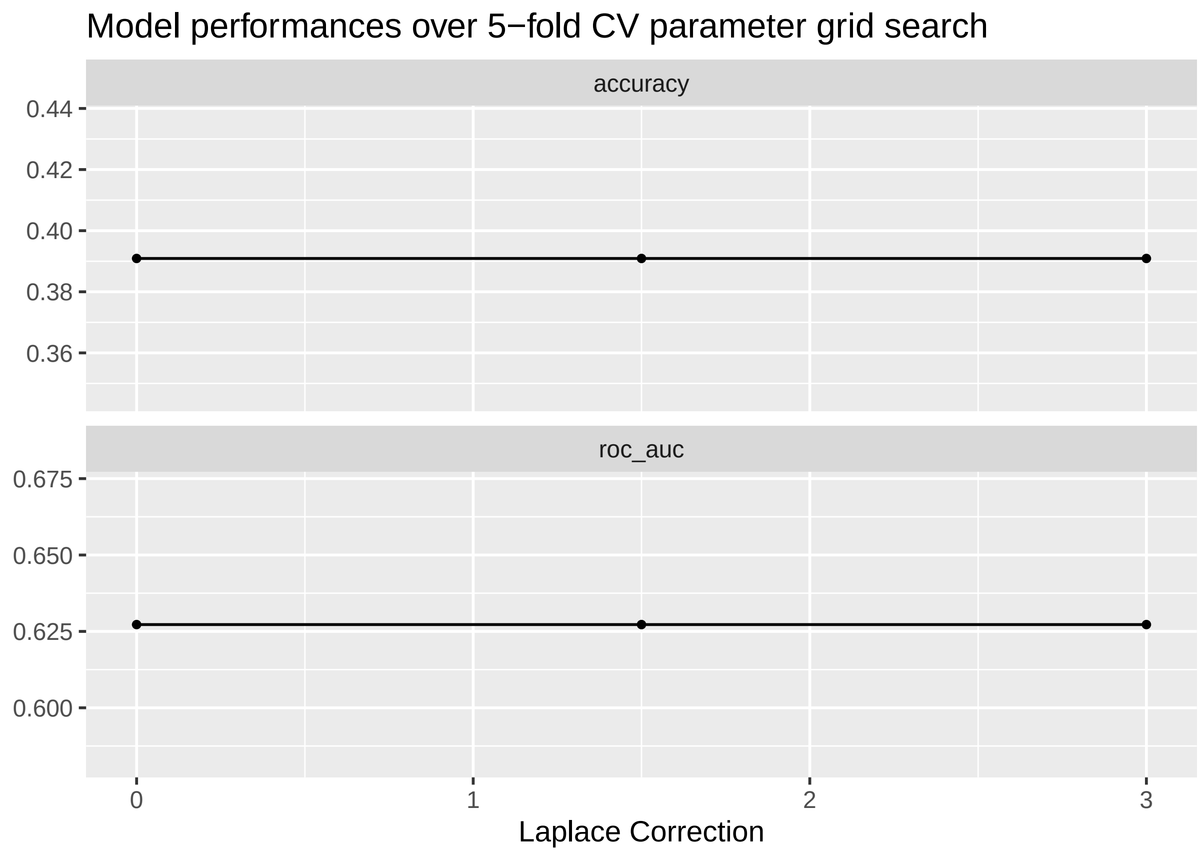
```r
set.seed(8675309)
folds <- vfold_cv(df_train, v = 5)
```

We now start fitting and tuning the three models. For this purpose, we will fit a Naive-Bayes, a lasso-GLM model, and an xGBoost model to the data - all for their widespread application in real-world text mining application. The lasso-regularised GLM is especially suitable thanks to its regularisation method, which also performs variable selection, ie. the penalty applied to certain features here means that less useful predictor tokens are penalised in favour of those with more predictive power.

Tuning Naive-Bayes Here we tune for one parameter: Laplace smoothing and the number of tokens to keep in training data. From the cross-validated results, we can extract the values with the highest AUC for the final model.

```r
###############     MODEL PREP: FITTING NB SPECS

library(discrim)

nb_spec <- naive_Bayes(Laplace = tune()) %>%
  set_mode("classification") %>%
  set_engine("naivebayes")

nb_grid <- grid_regular(
  Laplace()
)

nb_wf <- wf %>% add_model(nb_spec)
```

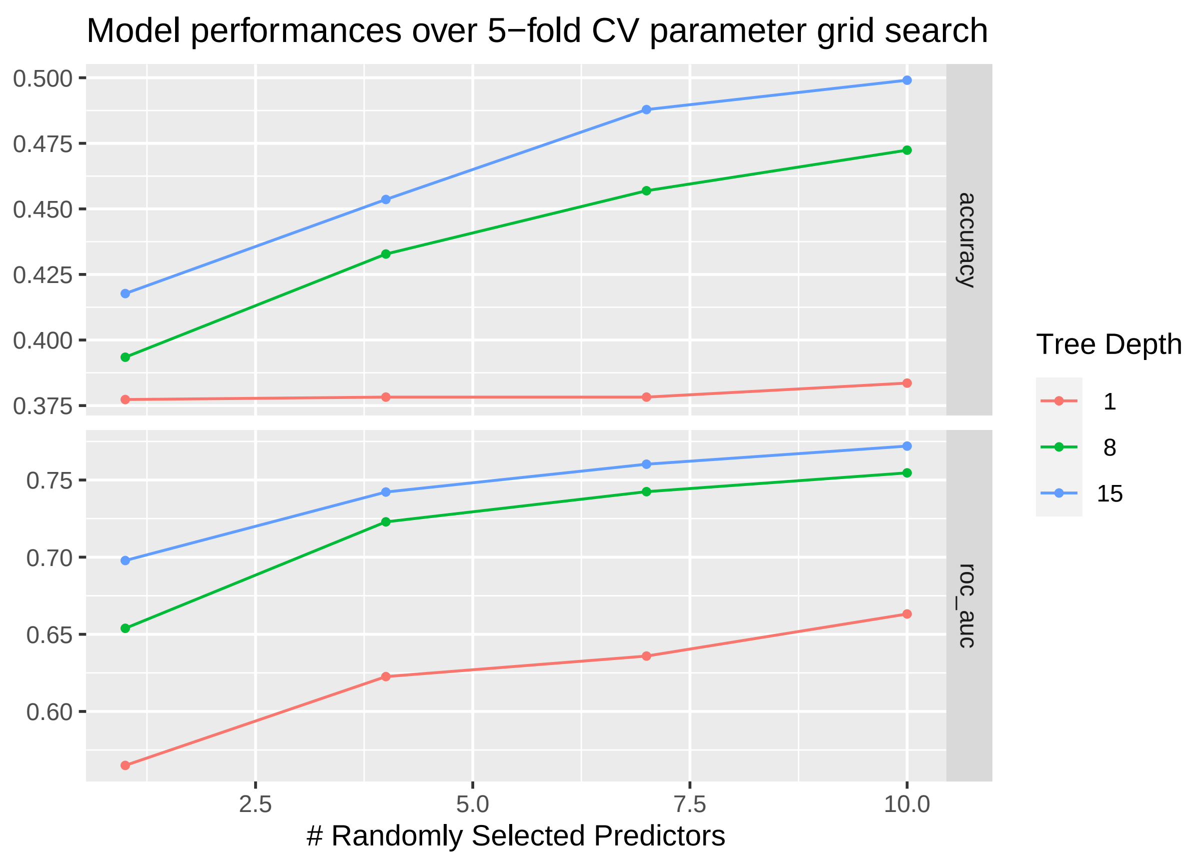
```r
set.seed(8675309)
nb_tune <- tune_grid(
  nb_wf,
  resamples = folds,
  grid = nb_grid,
  metrics = metric_set(accuracy,roc_auc)
)
```



Tuning xGB For the xGBoost model, we tune the mtry function. Again, we keep the parameters with the highest AUC, as a balance between overall accuracy and real-world performance.

```
###############     MODEL PREP: FITTING xGBoost SPECS
xg_spec <- boost_tree(mtry = tune(),
                      tree_depth = tune()) %>%
  set_mode("classification") %>%
  set_engine("xgboost")

xg_grid <- grid_regular(
  mtry(c(1L, 10L)),
  tree_depth(),
  levels = 4:3
)

xg_wf <- wf %>% add_model(xg_spec)
```

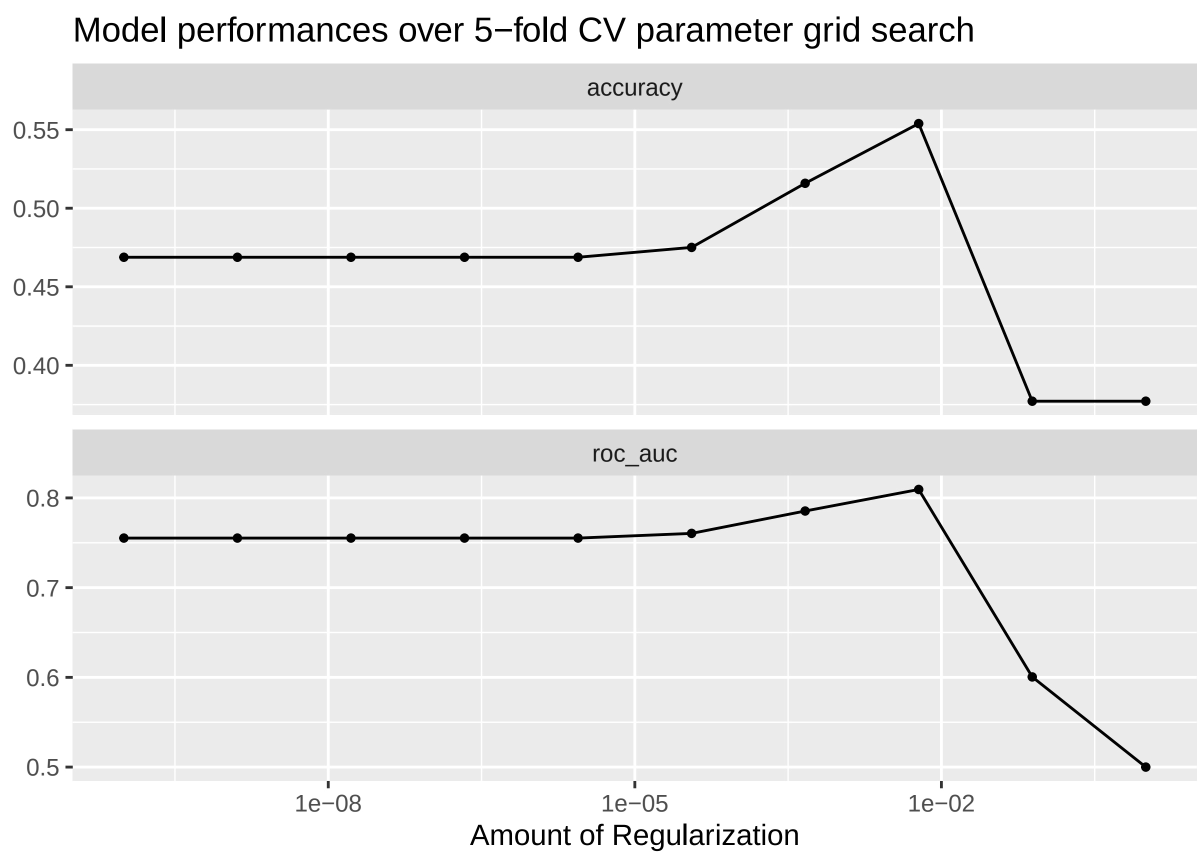
```
set.seed(8675309)
xg_tune <- tune_grid(
  xg_wf,
  resamples = folds,
  grid = xg_grid,
  metrics = metric_set(accuracy,roc_auc)
)
```



Tuning LASSO For our LASSO GLM model, we tune the penalty function. We will keep the parameter with the highest AUC. Here, note that our workflow is slightly modified compared to the other two models: instead of training directly on the preprocessed data, we add an extra conversion step through `hardhat` (part of the `tidymodels` ecosystem) to turn the data into a `dgCMatrix`-format sparse matrix, since glmnet is more optimised for sparse matrices.

```
###############     MODEL PREP: FITTING LASSO-GLM SPECS
library(hardhat)

ls_spec <- multinom_reg(penalty = tune(), mixture = 1) %>%
  set_mode("classification") %>%
  set_engine("glmnet")

ls_sparse_wf <- workflow() %>%
  add_recipe(recipe, blueprint = default_recipe_blueprint(composition = "dgCMatrix")) %>%
  add_model(ls_spec)

ls_grid <- grid_regular(
  penalty(),
  levels = 10
)
```

```
set.seed(8675309)
ls_tune <- tune_grid(
  ls_sparse_wf,
  folds,
  grid = ls_grid,
  control = control_resamples(save_pred = TRUE),
  metrics = metric_set(accuracy,roc_auc)
)
```



Finalising and Evaluating Models:

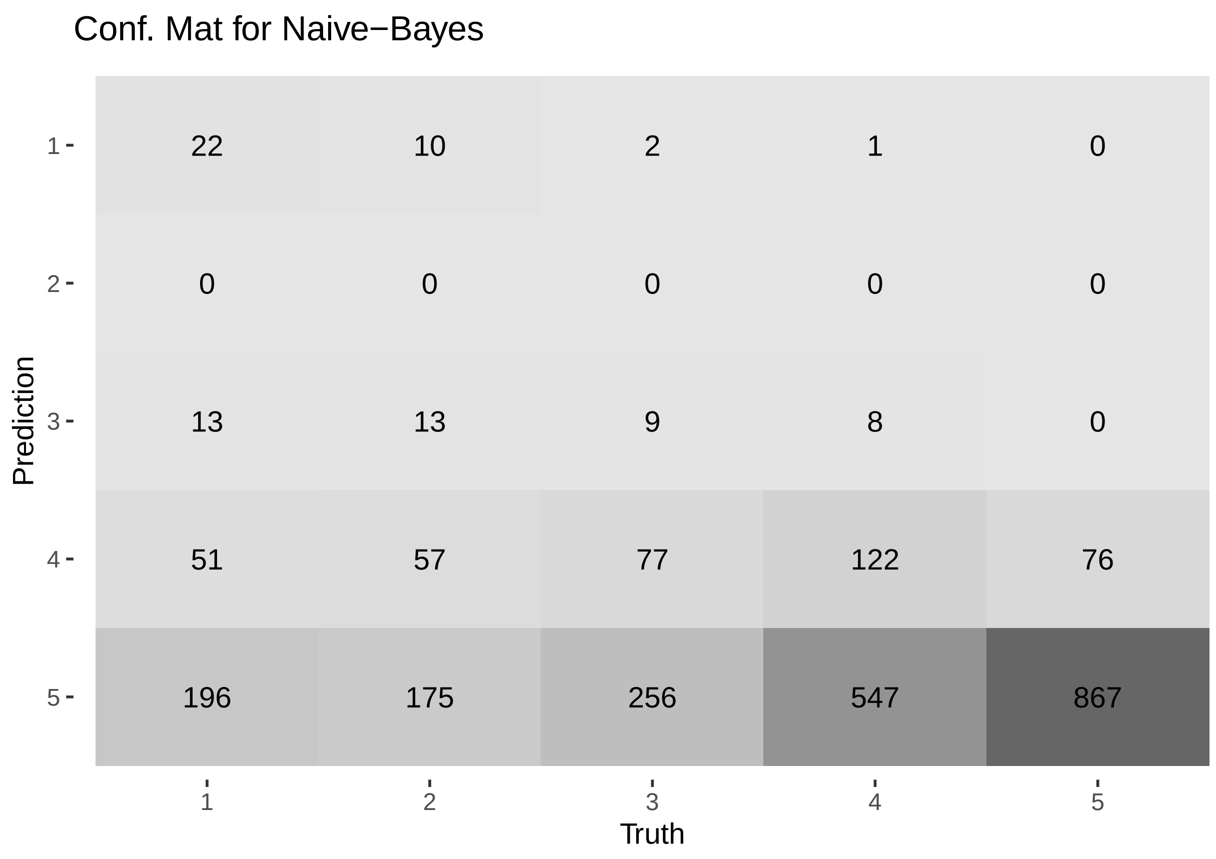
From the tuning results, instead of simply choosing the parameter set with the best AUC or best accuracy, we will choose the least complex model that falls within 1 standard deviation of the lowest AUC (Breiman et al. 1984). This allows for a model that has good performance but is not extremely complex, which prevents overfitting.

```
###############     FINALIZING MODELS: SELECTING BY BEST AUC

# Choosing parameters with AUC within 1sd of optimal, with lowest model complexity
nb_chosen <- nb_tune %>% select_by_one_std_err(metric = "roc_auc", -Laplace)
xg_chosen <- xg_tune %>% select_by_one_std_err(metric = "roc_auc", -tree_depth)
ls_chosen <- ls_tune %>% select_by_one_std_err(metric = "roc_auc", -penalty)

# Finalizing workflow with tuned params
nb_final <- finalize_workflow(nb_wf, nb_chosen)
xg_final <- finalize_workflow(xg_wf, xg_chosen)
ls_final <- finalize_workflow(ls_sparse_wf, ls_chosen)
```

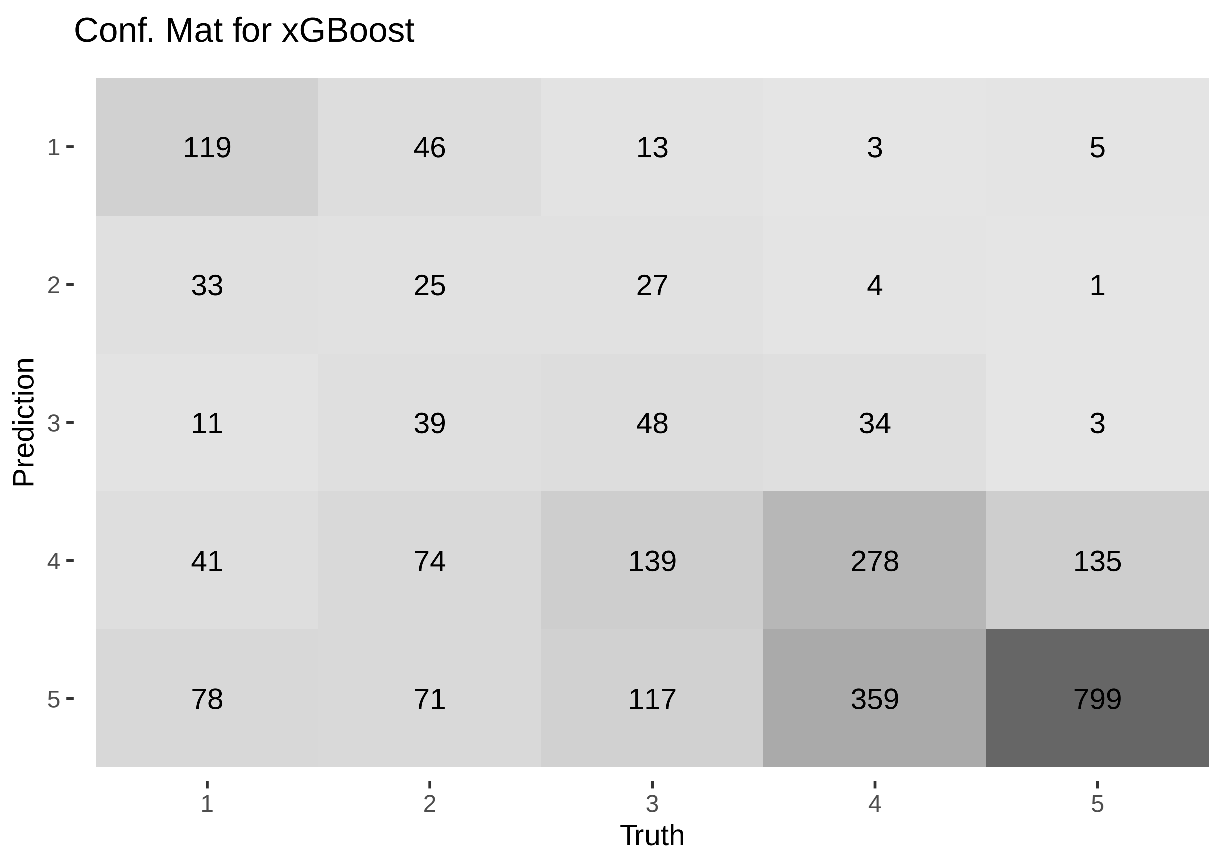
```r
# Fitting the finalized models on data (test set)
nb_final_fit <- last_fit(nb_final, splitkey)
xg_final_fit <- last_fit(xg_final, splitkey)
ls_final_fit <- last_fit(ls_final, splitkey)
```

With the test fits, we can now derive the confusion matrices and specific metrics for each model. Since this is a multiclass classification model, we will be using a 1-vs-all AUC average as the main evaluator, with overall accuracy as a secondary metric.

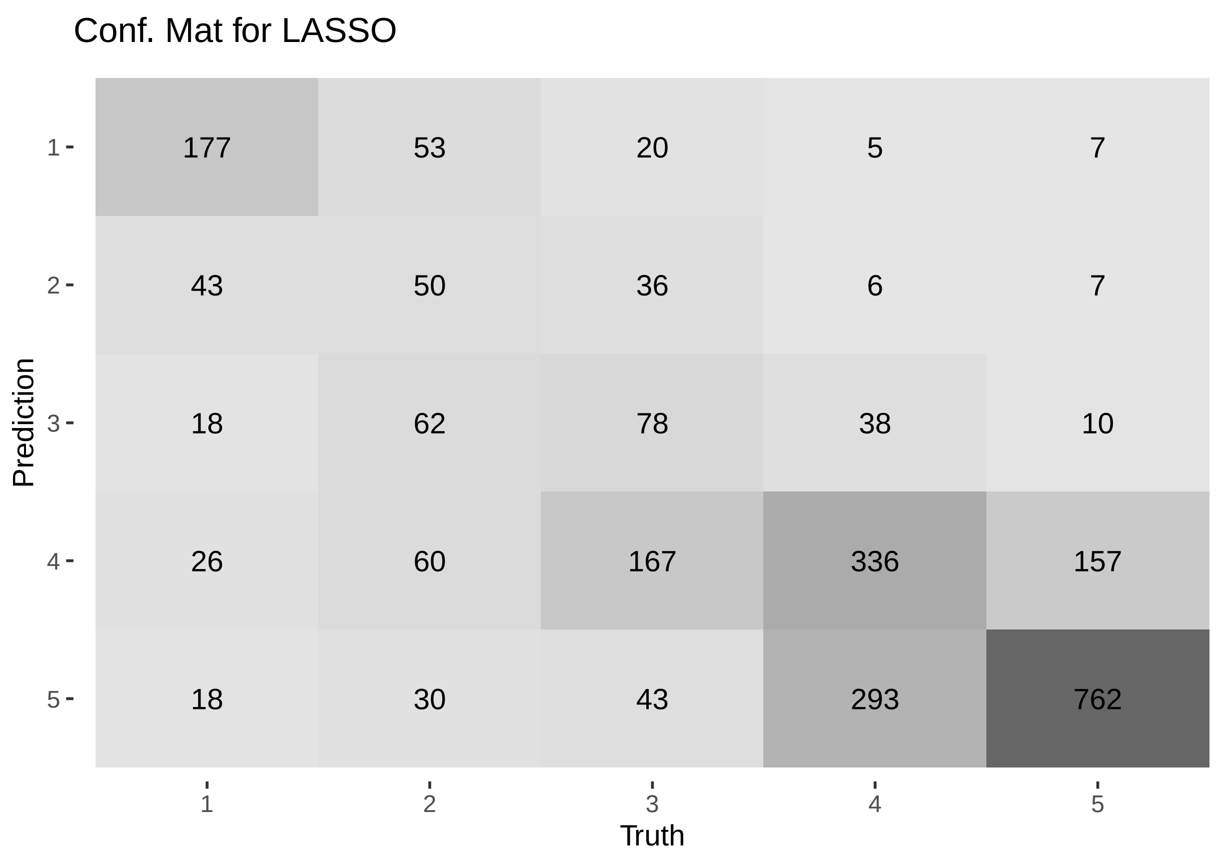
```r
# Confusion Matrix for models (on test set)
collect_predictions(nb_final_fit) %>%
  conf_mat(truth = stars, estimate = .pred_class) %>%
  autoplot(type = "heatmap") +
  labs(
    title = "Conf. Mat for Naive-Bayes"
  )
```



```r
collect_predictions(xg_final_fit) %>%
  conf_mat(truth = stars, estimate = .pred_class) %>%
  autoplot(type = "heatmap") +
  labs(
    title = "Conf. Mat for xGBoost"
  )
```



```r
collect_predictions(ls_final_fit) %>%
  conf_mat(truth = stars, estimate = .pred_class) %>%
  autoplot(type = "heatmap") +
  labs(
    title = "Conf. Mat for LASSO"
  )
```



```
# Average of Accuracies and Pairwise AUCs (on test set)
nb_metrics <- collect_metrics(nb_final_fit) %>%
  transmute(model = "Naive-Bayes", metric = .metric, value = .estimate) %>%
  pivot_wider(id_cols = model, values_from = value, names_from = metric)

xg_metrics <- collect_metrics(xg_final_fit) %>%
  transmute(model = "xGBoost", metric = .metric, value = .estimate) %>%
  pivot_wider(id_cols = model, values_from = value, names_from = metric)

ls_metrics <- collect_metrics(ls_final_fit) %>%
  transmute(model = "LASSO", metric = .metric, value = .estimate) %>%
  pivot_wider(id_cols = model, values_from = value, names_from = metric)

rbind(nb_metrics, xg_metrics, ls_metrics) %>% print()
```

```
## # A tibble: 3 x 3
##   model       accuracy roc_auc
##   <chr>          <dbl>   <dbl>
## 1 Naive-Bayes    0.408   0.630
## 2 xGBoost        0.507   0.785
## 3 LASSO          0.561   0.818
```