For some problems you may want to take a traditional regression or classification based approach3 while still accounting for the date/time-sensitive components of your data. In this post I will use the tidymodels suite of packages to:

build lag based and non-lag based features

set-up appropriate time series cross-validation windows

evaluate performance of linear regression and random forest models on a regression problem

For my example I will use data from Wake County food inspections. I will try to predict the

SCORE for upcoming restaurant food inspections.

# Load data

You can use Wake County’s open API (does not require a login/account) and the httr and jsonlite packages to load in the data. You can also download the data directly from the Wake County website.

library(tidyverse) library(lubridate) library(httr) library(jsonlite) library(tidymodels)

*Get food inspections data:*

r\_insp <- GET("https://opendata.arcgis.com/datasets/ebe3ae7f76954fad81411612d7c4fb 17\_1.geojson")

inspections <- content(r\_insp, "text") %>% fromJSON() %>%

.$features %>%

.$properties %>% as\_tibble()

inspections\_clean <- inspections %>%

mutate(date = ymd\_hms(DATE\_) %>% as.Date()) %>% select(-c(DATE\_, DESCRIPTION, OBJECTID))

*Get food locations data:*

r\_rest <- GET("https://opendata.arcgis.com/datasets/124c2187da8c41c59bde04fa67eb28 72\_0.geojson") #json

restauraunts <- content(r\_rest, "text") %>% fromJSON() %>%

.$features %>%

.$properties %>% as\_tibble() %>% select(-OBJECTID)

restauraunts <- restauraunts %>%

mutate(RESTAURANTOPENDATE = ymd\_hms(RESTAURANTOPENDATE) %>%

as.Date())

*Further prep:*

Join the inspections and restaurants datasets5

Filter out extreme outliers in SCORE (likely data entry errors) Filter to only locations of TYPE restaurant6

Filter out potential duplicate entries7

It’s important to consider which fields should be excluded for ethical reasons. For our problem, we will say that any restaurant name or location information must be excluded8.

inspections\_restaurants <- inspections\_clean %>% left\_join(restauraunts, by = c("HSISID", "PERMITID")) %>% filter(SCORE > 50, FACILITYTYPE == "Restaurant") %>% distinct(HSISID, date, .keep\_all = TRUE) %>%

select(-c(FACILITYTYPE, PERMITID)) %>%

select(-c(NAME, contains("ADDRESS"), CITY, STATE, POSTALCODE, PHONENUMBER, X, Y, GEOCODESTATUS))

inspections\_restaurants %>% glimpse()

## Rows: 24,294

## Columns: 6

## $ HSISID "04092017542", "04092017542", "04092017542",

"04...

## $ SCORE 94.5, 92.0, 95.0, 93.5, 93.0, 93.5, 92.5,

94.0, ...

## $ TYPE "Inspection", "Inspection", "Inspection", "Inspe...

## $ INSPECTOR "Anne-Kathrin Bartoli", "Laura McNeill", "Laura ...

## $ date 2017-04-07, 2017-11-08, 2018-03-23,

2018-09-07,...

## $ RESTAURANTOPENDATE 2017-03-01, 2017-03-01, 2017-03-01,

2017-03-01,...

# Feature Engineering & Data Splits

The first and last sub-sections:

Lag Based Features (Before Split, use dplyr or similar) Other Features (After Split, use recipes)

provide examples of the types of features that should be created before and after splitting your data respectively. Lag based features can, in some ways, be thought of as ‘raw inputs’ as they should be created prior to building a recipe.

## Lag Based Features (Before Split, use dplyr or similar)

Lag based features should generally be computed prior to splitting your data into “training” /

“testing” (or “analysis” / “assessment”) sets. This is because calculation of these features may depend on observations in prior splits. Let’s build a few features where this is the case:

Prior SCORE for HSISID

Average of prior 3 years of SCORE for HSISISD

Overall recent (year) prior average SCORE (across HSISISD) Days since RESTAURANTOPENDATE

Days since last inspection date

data\_time\_feats <- inspections\_restaurants %>% arrange(date) %>%

mutate(SCORE\_yr\_overall = slider::slide\_index\_dbl(SCORE,

.i = date,

.f = mean, na.rm = TRUE,

.before =

lubridate::days(365),

-lubridate::days(1))

) %>%

group\_by(HSISID) %>% mutate(SCORE\_lag = lag(SCORE),

.after =

SCORE\_recent = slider::slide\_index\_dbl(SCORE,

date, mean,

na.rm = TRUE,

.before =

lubridate::days(365\*3),

-lubridate::days(1),

.after =

.complete = FALSE),

days\_since\_open = (date - RESTAURANTOPENDATE) / ddays(1), days\_since\_last = (date - lag(date)) / ddays(1)) %>%

ungroup() %>% arrange(date)

The use of .after = -lubridate::days(1) prevents data leakage by ensuring that this feature does not include information from the current day in its calculation13 14.

## Data Splits

### Additional Filtering:

We will presume that the model is only intended for restaurants that have previous inspections on record15 and will use only the most recent seven years of data.

data\_time\_feats <- data\_time\_feats %>%

filter(date >= (max(date) - years(7)), !is.na(SCORE\_lag))

### Initial Split:

After creating our lag based features, we can split our data into training and testing splits.

initial\_split <- rsample::initial\_time\_split(data\_time\_feats, prop =

.8)

train <- rsample::training(initial\_split) test <- rsample::testing(initial\_split)

### Resampling (Time Series Cross-Validation):

For this problem we should evaluate our models using time series cross-validation16. This entails creating multiple ordered subsets of the training data where each set has a different assignment of observations into “analysis” or “assessment” data17.

Ideally the resampling scheme used for model evaluation mirrors how the model will be built and evaluated in production. For example, if the production model will be updated once every three months it makes sense that the “assessment” sets be this length. We can use rsample::sliding\_period() to set things up.

For each set, we will use three years of “analysis” data for training a model and then three months of “assessment” data for evaluation.

resamples <- rsample::sliding\_period(train,

index = date, period = "month", lookback = 36,

assess\_stop = 3,

step = 3)

I will load in some helper functions I created for reviewing the dates of our resampling windows.

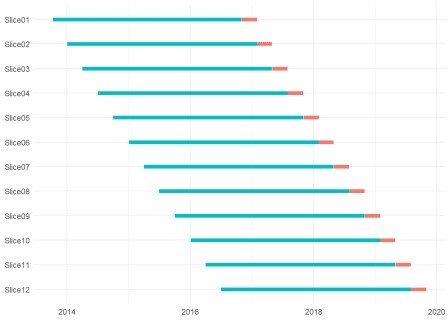
resamples %>% extract\_dates\_rset() %>% print() %>% plot\_dates\_rset()

## # A tibble: 12 x 6

## splits id analysis\_min analysis\_max assessment\_min assessment\_max

##

## 1



For purposes of overall Model Evaluation, performance across each period will be weighted equally (regardless of number of observations in a period).

## Other Features (After Split, use recipes)

Where possible, features should be created using the recipes package. recipes makes pre- processing convenient and helps prevent data leakage.

It is OK to modify or transform a previously created lag based feature in a recipes step. Assuming that you created the lag based input as well as your resampling windows in an appropriate manner, you should be safe from data leakage issues when modifying the variables during later feature engineering steps22.

*Some features / transformations I’ll make with recipes:*

collapse rare values for INSPECTOR and TYPE

log transform days\_since\_open and days\_since\_last

add calendar based features

rec\_general <- recipes::recipe(SCORE ~ ., data = train) %>% step\_rm(RESTAURANTOPENDATE) %>%

update\_role(HSISID, new\_role = "ID") %>% step\_other(INSPECTOR, TYPE, threshold = 50) %>% step\_string2factor(one\_of("TYPE", "INSPECTOR")) %>% step\_novel(one\_of("TYPE", "INSPECTOR")) %>% step\_log(days\_since\_open, days\_since\_last) %>% step\_date(date, features = c("dow", "month")) %>% update\_role(date, new\_role = "ID") %>% step\_zv(all\_predictors())

Let’s peak at the features we will be passing into the model building step:

prep(rec\_general, data = train) %>% juice() %>%

glimpse()

## Rows: 17,048

## Columns: 12

## $ HSISID 04092016152, 04092014520, 04092014483,

04092012102...

## $ TYPE Inspection, Inspection, Inspection, Inspection, In...

## $ INSPECTOR David Adcock, Naterra McQueen, Andrea Anover, othe...

## $ date 2013-10-09, 2013-10-09, 2013-10-09, 2013-10-09,

2...

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ## $ SCORE\_yr\_overall | 96.22766, 96.22766, 96.22766, | | | 96.22766, |
| 96.22766, ... |  | | |  |
| ## $ SCORE\_lag | 96.0, 95.5, 97.0, 94.5, 97.5, | | | 99.0, 96.0, 96.0, |
| 10... |  | | |  |
| ## $ SCORE\_recent | 96.75000, | 95.75000, | 97.50000, | 95.25000, |
| 96.75000, ... |  |  |  |  |
| ## $ days\_since\_open | 6.410175, | 7.926964, | 7.959276, | 8.682029, |
| 8.970432, ... |  |  |  |  |
| ## $ days\_since\_last | 4.709530, | 4.941642, | 4.934474, | 4.875197, |
| 5.117994, ... |  |  |  |  |

## $ SCORE 98.5, 96.0, 96.0, 93.0, 95.0, 93.5, 95.0, 92.0,

98...

## $ date\_dow Wed, Wed, Wed, Wed, Wed, Wed, Wed, Wed, Wed, Thu, ...

## $ date\_month Oct, Oct, Oct, Oct, Oct, Oct, Oct, Oct, Oct, Oct, ...

# Model Specification and Training

### Simple linear regression model:

lm\_mod <- parsnip::linear\_reg() %>% set\_engine("lm") %>% set\_mode("regression")

lm\_workflow\_rs <- workflows::workflow() %>% add\_model(lm\_mod) %>% add\_recipe(rec\_general) %>% fit\_resamples(resamples,

control = control\_resamples(save\_pred = TRUE))

### ranger Random Forest model (using defaults):

rand\_mod <- parsnip::rand\_forest() %>% set\_engine("ranger") %>% set\_mode("regression")

set.seed(1234)

rf\_workflow\_rs <- workflow() %>% add\_model(rand\_mod) %>%

add\_recipe(rec\_general) %>% fit\_resamples(resamples,

control = control\_resamples(save\_pred = TRUE))

### parsnip::null\_model:

The NULL model will be helpful as a baseline Root Mean Square Error (RMSE) comparison.

null\_mod <- parsnip::null\_model(mode = "regression") %>% set\_engine("parsnip")

null\_workflow\_rs <- workflow() %>% add\_model(null\_mod) %>% add\_formula(SCORE ~ NULL) %>% fit\_resamples(resamples,

control = control\_resamples(save\_pred = TRUE))

# Model Evaluation

The next several code chunks extract the *average* performance across “assessment” sets24 or extract the performance across each of the individual “assessment” sets.

mod\_types <- list("lm", "rf", "null"

avg\_perf <- map(list(lm\_workflow\_rs, rf\_workflow\_rs, null\_workflow\_rs), collect\_metrics) %>%

map2(mod\_types, ~mutate(.x, source = .y)) %>% bind\_rows()

extract\_splits\_metrics <- function(rs\_obj, name){

rs\_obj %>%

select(id, .metrics) %>% unnest(.metrics) %>% mutate(source = name)

}

splits\_perf <- map2(list(lm\_workflow\_rs, rf\_workflow\_rs, null\_workflow\_rs),

mod\_types, extract\_splits\_metrics) %>%

bind\_rows()

The overall performance as well as the performance across splits suggests that both models were better than the baseline (the mean within the analysis set)25 and that the linear model outperformed the random forest model.

splits\_perf %>%

mutate(id = forcats::fct\_rev(id)) %>%

ggplot(aes(x = .estimate, y = id, colour = source))+ geom\_vline(aes(xintercept = mean, colour = fct\_relevel(source,

c("lm", "rf", "null"))),

alpha = 0.4, data = avg\_perf)+

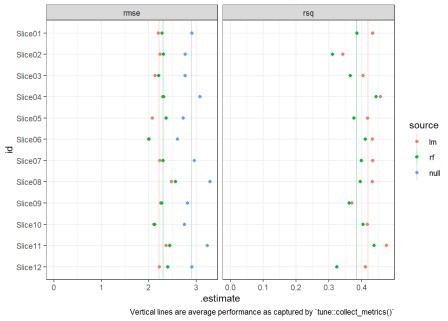
geom\_point()+

facet\_wrap(~.metric, scales = "free\_x")+ xlim(c(0, NA))+

theme\_bw()+

labs(caption = "Vertical lines are average performance as captured by

`tune::collect\_metrics()`")



We could use a paired sample t-test to formally compare the random forest and linear models’ out-of-sample RMSE performance.

t.test(

filter(splits\_perf, source == "lm", .metric == "rmse") %>% pull(.estimate),

filter(splits\_perf, source == "rf", .metric == "rmse") %>% pull(.estimate),

paired = TRUE

) %>%

broom::tidy() %>% mutate(across(where(is.numeric), round, 4)) %>% knitr::kable()

### estimate statistic p.value parameter conf.low conf.high method alternative

-0.0839 -3.7277 0.0033 11 -0.1334 -0.0343 Paired t-test two.sided

This suggests the better performance by the linear model *is* statistically significant.

### Other potential steps:

There is lots more we could do from here. However the purpose of this post was to provide a short tidymodels example that incorporates window functions from rsample and slider on a regression problem. For more resources on modeling and the tidymodels framework

# Appendix

## Model Building with Hyperparameter Tuning

Below is code for tuning a glmnet linear regression model (use tune to optimize the L1/L2 penalty).

rec\_glmnet <- rec\_general %>% step\_dummy(all\_predictors(), -all\_numeric()) %>% step\_normalize(all\_predictors(), -all\_nominal()) %>% step\_zv(all\_predictors())

glmnet\_mod <- parsnip::linear\_reg(penalty = tune(), mixture = tune())

%>%

set\_engine("glmnet") %>% set\_mode("regression")

glmnet\_workflow <- workflow::workflow() %>% add\_model(glmnet\_mod) %>% add\_recipe(rec\_glmnet)

glmnet\_grid <- tidyr::crossing(penalty = 10^seq(-6, -1, length.out = 20), mixture = c(0.05,

0.2, 0.4, 0.6, 0.8, 1))

glmnet\_tune <- tune::tune\_grid(glmnet\_workflow,

resamples = resamples,

control = control\_grid(save\_pred = TRUE), grid = glmnet\_grid)

And code to tune a ranger Random Forest model, tuning the mtry and min\_n parameters29.

rand\_mod <- parsnip::rand\_forest(mtry = tune(), min\_n = tune(), trees = 1000) %>%

set\_engine("ranger") %>% set\_mode("regression")

rf\_workflow <- workflow() %>% add\_model(rand\_mod) %>% add\_recipe(rec\_general)

cores <- parallel::detectCores()

set.seed(1234)

rf\_tune <- tune\_grid(rf\_workflow,

resamples = resamples,

control = control\_grid(save\_pred = TRUE), grid = 25)