For some problems you may want to take a traditional regression or classification based approach[3] 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[4].

```
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` datasets[5]
- Filter out extreme outliers in `SCORE` (likely data entry errors)
- Filter to only locations of `TYPE` restaurant[6]
- Filter out potential duplicate entries[7]
- 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 excluded[8].

```
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

Discussion on issue #168 suggests that some features (those depending on prior observations) should be created before the data is split[9]. 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`[10].

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"[11]) sets. This is because calculation of these features may depend on observations in prior splits[12]. 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),
                                                    .after =
-lubridate::days(1))
         ) %>%
  group_by(HSISID) %>%
  mutate(SCORE_lag = lag(SCORE),
         SCORE_recent = slider::slide_index_dbl(SCORE,
                                                date,
                                                mean,
                                                na.rm = TRUE,
                                                .before =
lubridate::days(365*3),
                                                .after =
-lubridate::days(1),
                                                .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 calculation[13] [14].

Data Splits

Additional Filtering:

We will presume that the model is only intended for restaurants that have previous inspections on record[15] 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-validation[16]. This entails creating multiple ordered subsets of the training data where each set has a different assignment of observations into “analysis” or “assessment” data[17].

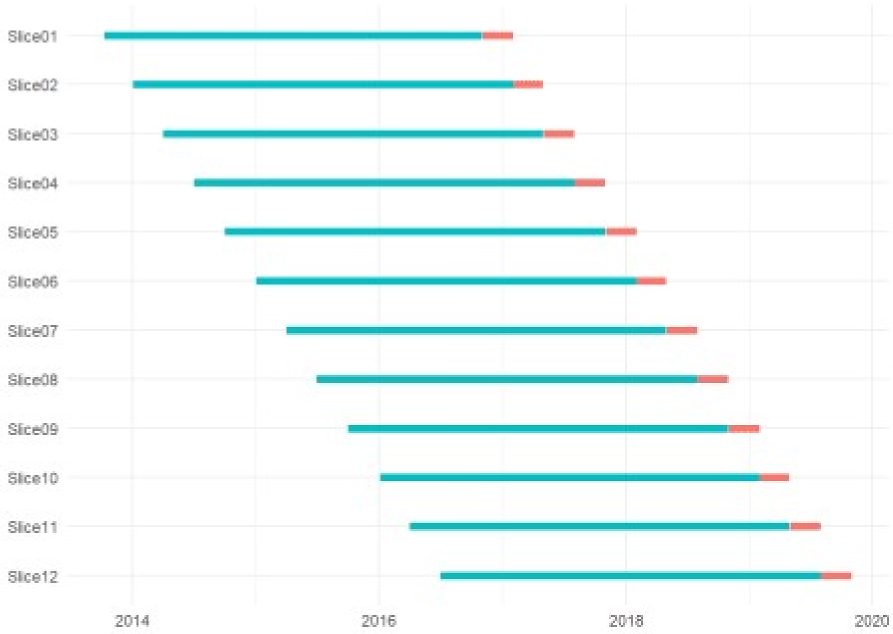
Ideally the resampling scheme used for model evaluation mirrors how the model will be built and evaluated in production. For example, if the production model will be updated once every three months it makes sense that the “assessment” sets be this length. We can use `rsample::sliding_period()` to set 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[18].

```
devtools::source_gist("https://gist.github.com/brshallo/
7d180bde932628a151a4d935ffa586a5")

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)[19] [20].

Other Features (After Split, use `recipes`)

Where possible, features should be created using the recipes package[21]. `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 steps[22].

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))
```

See code in Model Building with Hyperparameter Tuning for more sophisticated examples that include hyperparameter tuning for `glmnet`[23] and `ranger` models.

Model Evaluation

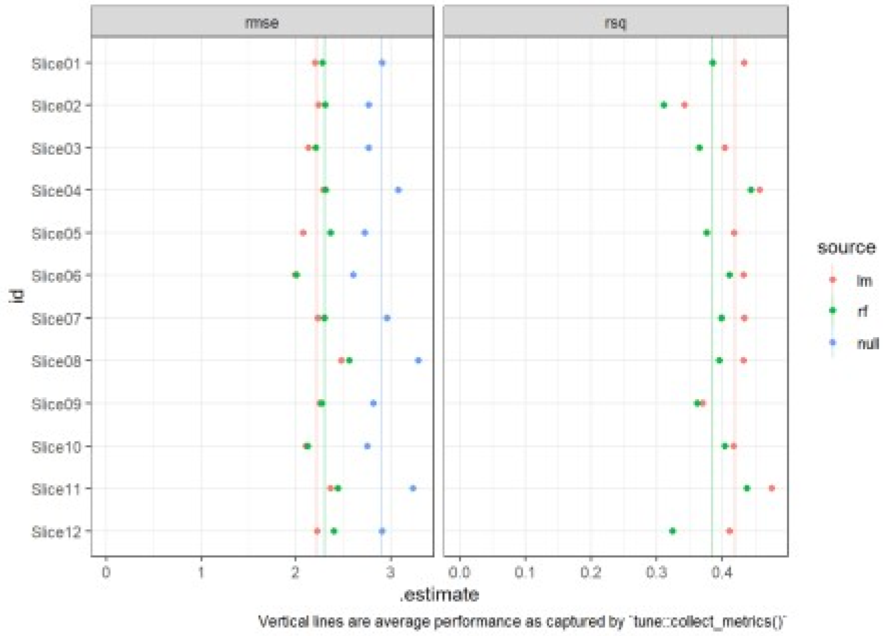
The next several code chunks extract the *average* performance across “assessment” sets[24] or extract the performance across each of the individual “assessment” sets.

```
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[26]. 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, see tidymodels.org or Tidy Modeling with R[27].

Appendix

Model Building with Hyperparameter Tuning

Below is code for tuning a `glmnet` linear regression model (use `tune` to optimize the L1/L2 penalty)[28].

```
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` parameters[29].

```
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)
```