Code Workflow Example (Draft)

Pre-processing Notes

This document provides an overview of how to estimate causal impacts on long-T panel data by stringing together functions in this repository. The input data to estimate should be a long form tibble or dataframe, meaning each row contains information about a unique unit-time combination and each unit's series should be complete so that every unit has an observed value at every period in the data. Some pre-processing might be needed to get data into this form – namely removing NAs as well as time periods that only contain data from a subset of the units. See ?pivot_longer in R for helpful documentation in how to take wide data to long.

Once the data is in the correct form, there are a few more convenience items to optionally take care of. The analysis requires variables (columns in the tibble) identifying the unit ID (by defauly, searches for "entry"), the time ID ("period", which would ideally be recoded to range from 1 to T), a treatment indicator for each unit by time combination (by default, "treatperiod_0"), an outcome variable (the observed outcome, default "target"), and if it exists, a counterfactual outcome for that unit time combination ("counter_factual"). Should you decide to change the column names to these defaults, you will not need to specify the names when calling the pre-estimation plotting and estimation functions. Because the first set of estimator functions output a long-form tibble with point predictions and CI bands for each unit-time combination, the names of newly created columns in these output tibbles will automatically be the default for functions called later in the workflow.

The set of estimators implemented in this repository share a common structure in that they find a set of control units (sometimes called donors) that emulate a particular treatment unit, and use this donors to answer "what would have happened to this treatment unit had it not been treated"; this is known as imputing the unobserved potential outcome of the treated unit. Then, by comparing the actual outcome of the treated unit to this predicted counterfactual outcome (the imputed value based on donors), we calculate a treatment effect – how much and in which direction did the treatment impact the unit relative to where we estimate it otherwise would've been.

In order to convince ourselves that the estimators are able to adequately impute this missing outcome, we must carefully consider a set of practical assumptions underlying this analysis[1]: 1) Do the treated units have similar outcome series as the control units? 2) Do we expect the treatment effect to be large enough to detect, given the variability of the series? 3) Do we have any reason to believe that some of the controls have suffered large idiosyncratic shocks during the relevant periods? 4) Could the treated units have anticipated the intervention? 5) Could there be spillover effects from treated units onto others? 6) Are there enough pre-treatment periods to be confident that our donors will provide a good approximation? Reflecting on these questions prior to estimating the models is recommended, and the following sub-section provides some concrete steps in that direction.

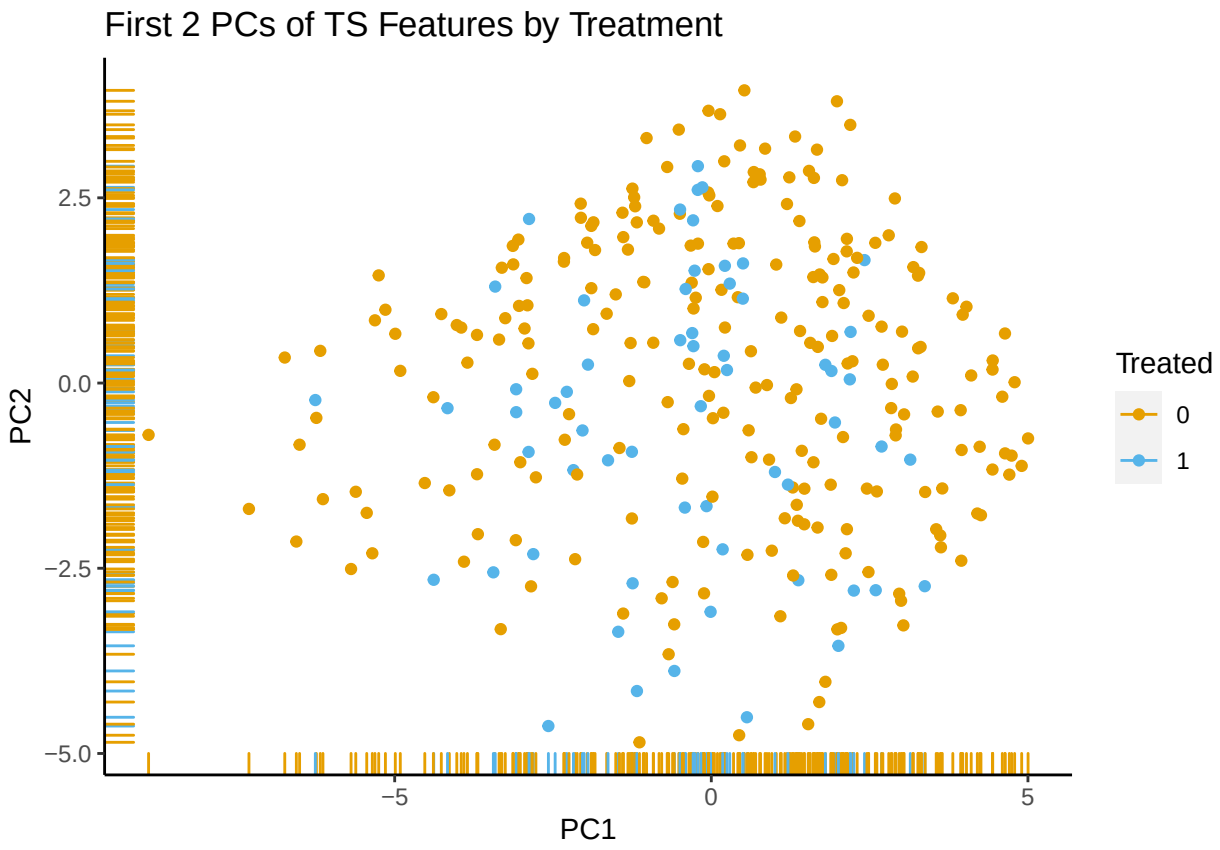
Sanity Checks

Although the particular answers to the above questions will vary by context, there are a few commands we recommend plotting prior to proceeding with estimation. Borrowing from Hyndman (TS Features), *TSFeaturesPlot* and *TSFeatureTest* provide complemtary approaches for understanding how different the

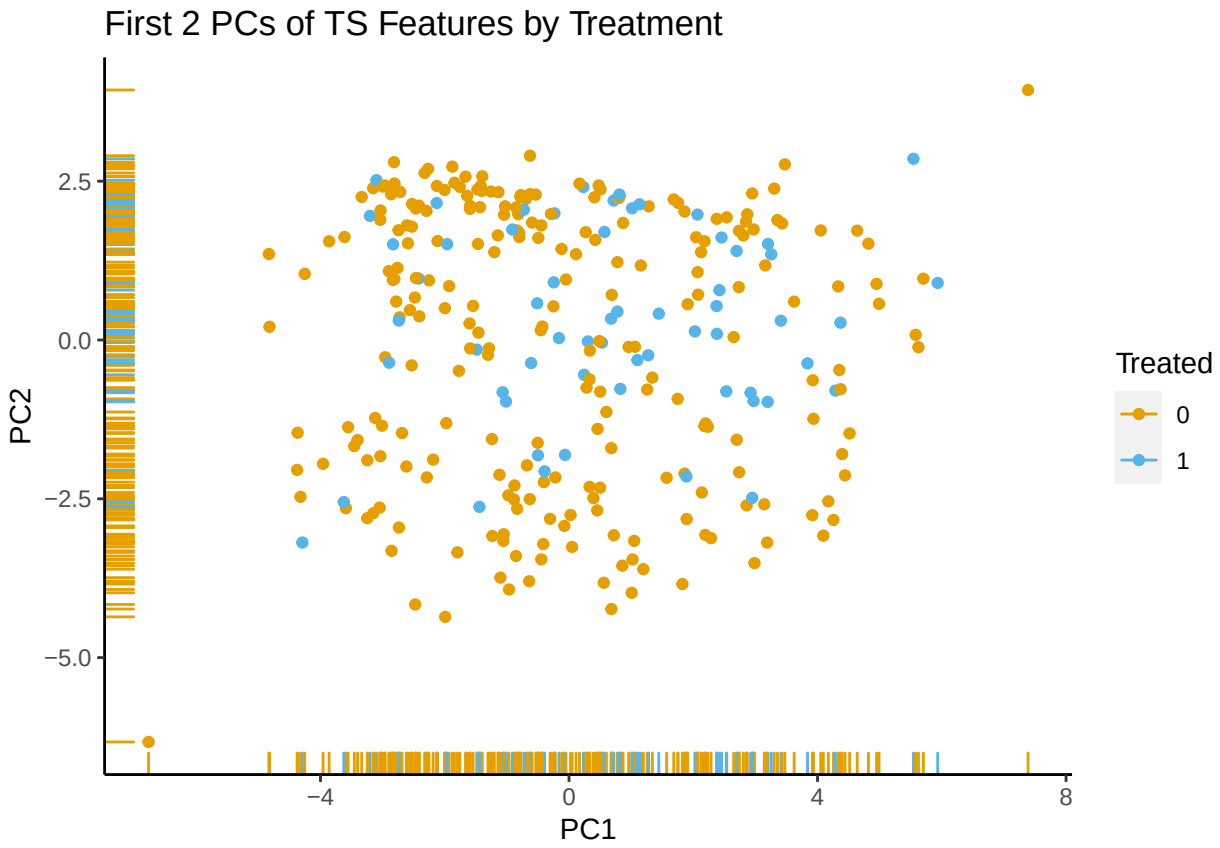
[1]For technical assumptions, please review our sister paper on benchmarking the methods in this repository (LINK).

treatment and control time series are by feature. There are a large number of time series features (e.g. autocorrelation, seasonality, entropy) that are computed within these functions, and *TSFeaturesPlot* produces a scatter plot of the first two principle components of these features by treatment status while *TSFeatureTest* produces a table of t-tests on the null hypothesis that the features in the two groups are the same. To examine the distribution of a particular feature (perhaps to get a sense of whether to trim extreme values), the *FeatureDensity* function takes the relevant feature and dataset as input and returns a group density. The code below produces an example of these plots in two different dataset from our *SyntheticDGP* method – one with selection and one without. The plots demonstrate that the features of the treatment group seem fairly well covered despite the selection problem in the second data, but the hypothesis tests suggest we can reject the null of no difference between certain features among the two groups.

```
TSFeaturesPlot(sim_data)
```




```
TSFeaturesPlot(selection_data)
```




```
TSFeatureTest(sim_data) %>% dplyr::select(-c(group1, group2))
```

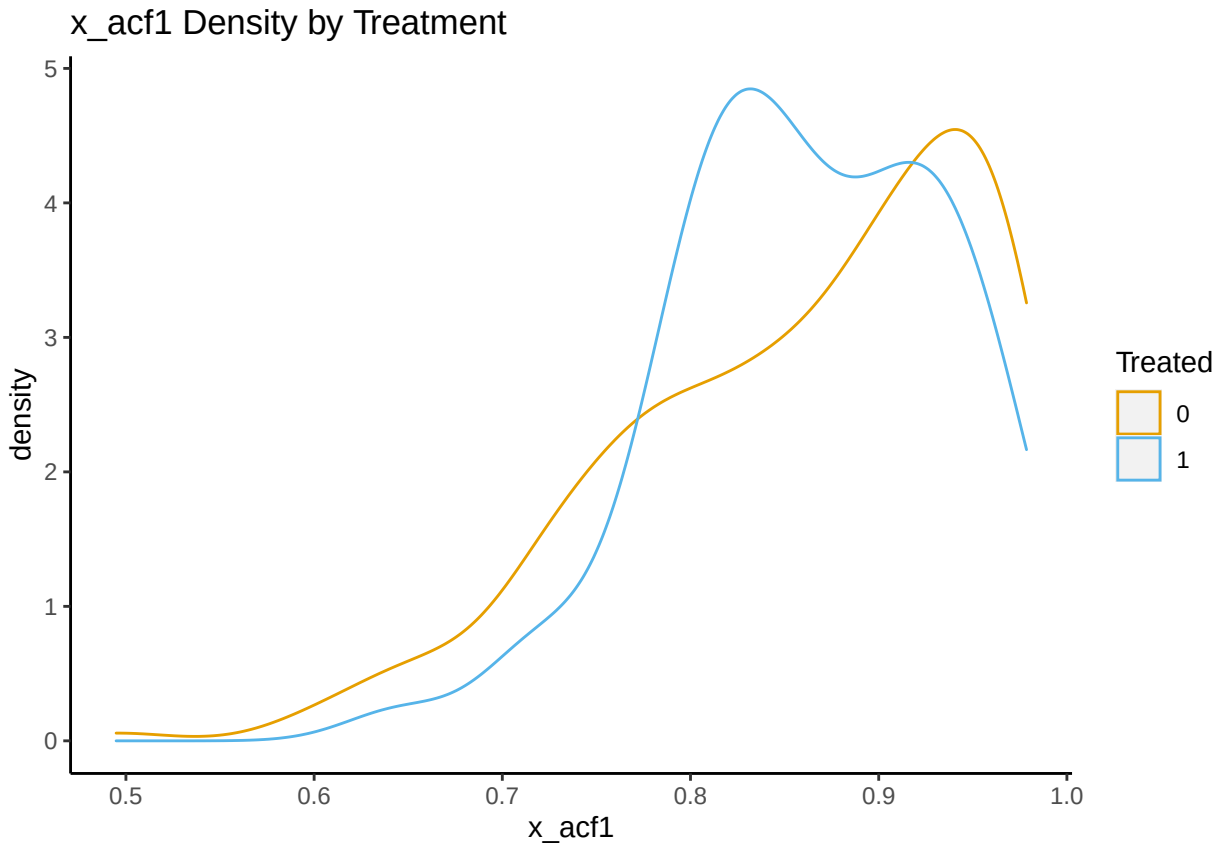
```
## # A tibble: 13 x 8
##    vars           n1    n2 statistic    df        p   p.adj p.adj.signif
##    <chr>       <int> <int>     <dbl> <dbl>    <dbl>   <dbl> <chr>
##  1 curvature     240    60   -0.0120  98.1 0.99     0.99    ns
##  2 diff1_acf1    240    60    3.29   109.  0.00136  0.00884 **
##  3 diff1_acf10   240    60    3.79   107.  0.000248 0.00322 **
##  4 diff2_acf1    240    60    2.38   107.  0.0193   0.0502  ns
##  5 diff2_acf10   240    60   -2.30    87.8 0.0237   0.0513  ns
##  6 e_acf1        240    60    3.09   102.  0.00257  0.0111  *
##  7 e_acf10       240    60    2.79    91.0 0.00647  0.0210  *
##  8 entropy       240    60   -0.187  100.  0.852    0.953   ns
##  9 linearity     240    60   -2.10   119.  0.038    0.0706  ns
## 10 spike         240    60    1.18   114.  0.239    0.388   ns
## 11 trend         240    60   -0.152  108.  0.88     0.953   ns
## 12 x_acf1        240    60   -0.213  115.  0.832    0.953   ns
## 13 x_acf10       240    60    0.459  102.  0.647    0.935   ns
```

```
TSFeatureTest(selection_data) %>% dplyr::select(-c(group1, group2))
```

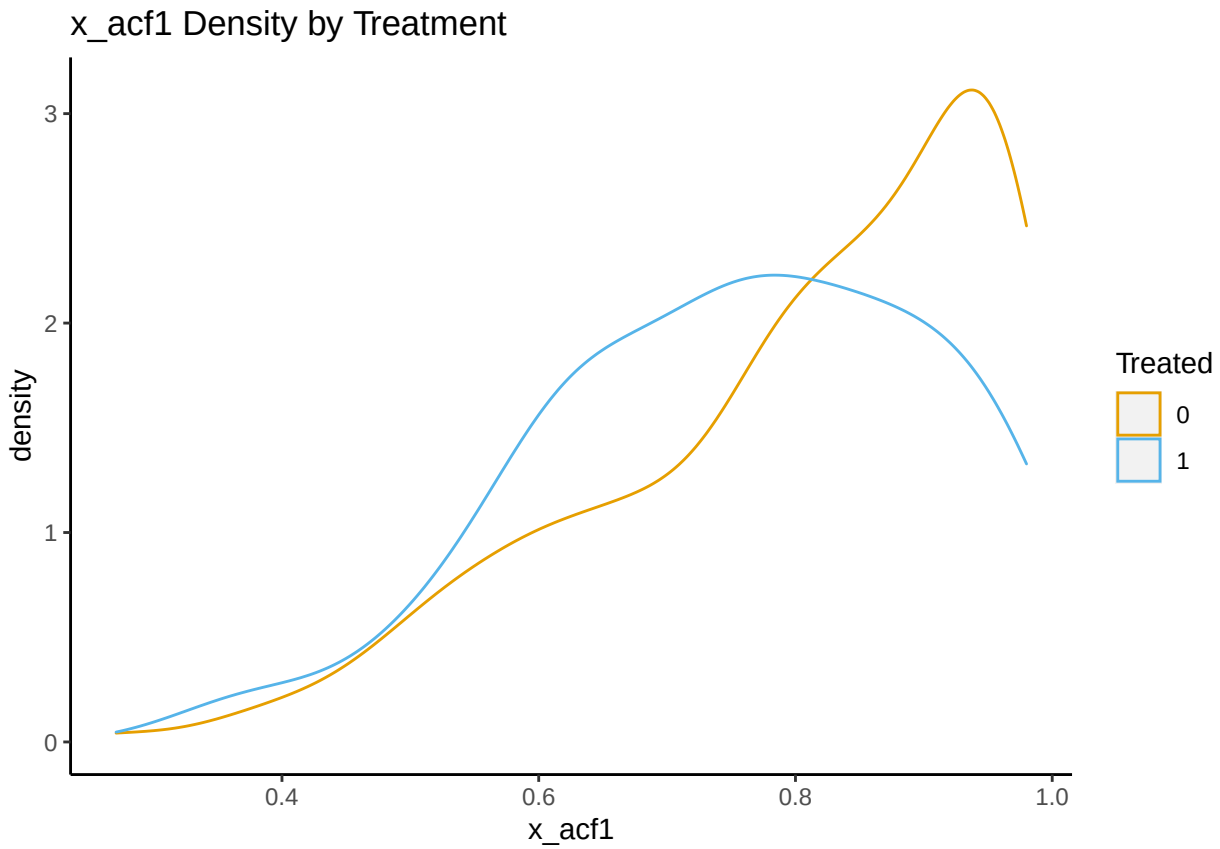
```
## # A tibble: 13 x 8
##    vars           n1    n2 statistic    df         p    p.adj p.adj.signif
##    <chr>       <int> <int>     <dbl> <dbl>     <dbl>    <dbl> <chr>
##  1 curvature     240    60  -1.01    166.  0.314     0.408    ns
```

```
##  2 diff1_acf1    240    60   4.11    120.  0.0000739 0.000961 ***
##  3 diff1_acf10   240    60  -0.327   135.  0.744     0.841    ns
##  4 diff2_acf1    240    60   2.71    121.  0.0078    0.0338   *
##  5 diff2_acf10   240    60  -1.18    110.  0.241     0.348    ns
##  6 e_acf1        240    60   3.73    103.  0.000317  0.00206  **
##  7 e_acf10       240    60  -0.00944 111.  0.992     0.992    ns
##  8 entropy       240    60  -2.55    103.  0.0122    0.0397   *
##  9 linearity     240    60  -0.285   136.  0.776     0.841    ns
## 10 spike         240    60  -1.42     97.7 0.157     0.255    ns
## 11 trend         240    60   2.33     94.9 0.0222    0.0577   ns
## 12 x_acf1        240    60   2.11     92.4 0.0378    0.0819   ns
## 13 x_acf10       240    60   1.84     98.2 0.0688    0.128    ns
```

```
FeatureDensity(sim_data)
```

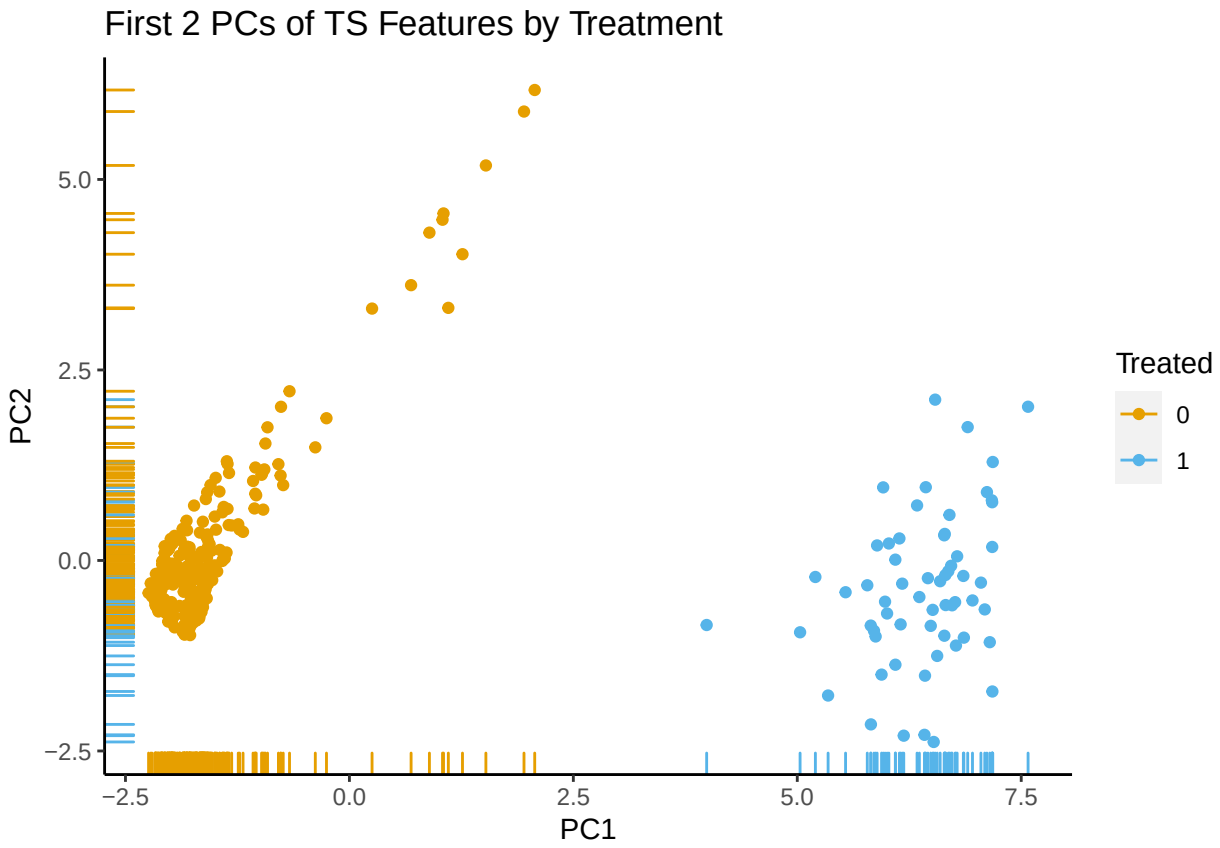



```
FeatureDensity(selection_data)
```



A more extreme example is run below, displaying a case where these methods are not advisable without modifications. Here, the autocorrelation of the control units is drawn from a distribution quite separate from that of the treatment group (as is the distribution of the factor loading – the same selection as above.)

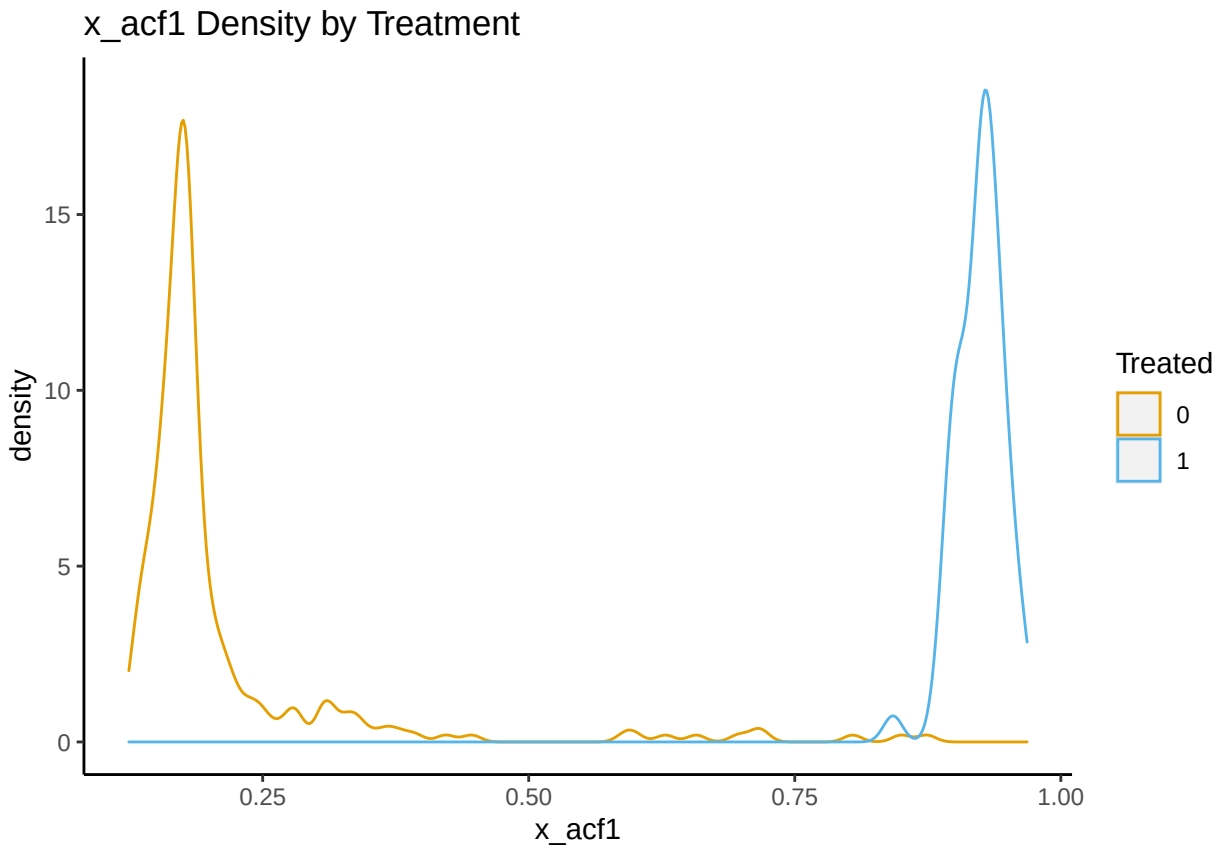
```
inappropriate_data <-
  SyntheticDGP(num_entries = units, prop_treated = 0.2,
               loading_scale = 0.9, rho_shift=0.05, rho_scale=0) %>%
  FormatForEst()
TSFeaturesPlot(inappropriate_data)
```




```
TSFeatureTest(inappropriate_data) %>% dplyr::select(-c(group1, group2))
```

```
## # A tibble: 13 x 8
##    vars           n1    n2 statistic    df         p     p.adj p.adj.signif
##    <chr>       <int> <int>     <dbl> <dbl>     <dbl>     <dbl> <chr>
##  1 curvature     240    60      9.22  60.5 3.98e- 13 3.98e- 13 ****
##  2 diff1_acf1    240    60   -338.    63.1 1.38e-104 2.99e-104 ****
##  3 diff1_acf10   240    60     58.3  107.  8.26e- 83 1.34e- 82 ****
##  4 diff2_acf1    240    60   -145.   120.  6.55e-137 1.70e-136 ****
##  5 diff2_acf10   240    60     46.9  166.  1.29e- 97 2.40e- 97 ****
##  6 e_acf1        240    60   -123.   155.  3.15e-156 1.36e-155 ****
##  7 e_acf10       240    60    -31.3   59.7 9.62e- 39 1.25e- 38 ****
##  8 entropy       240    60     23.0   88.7 3.69e- 39 5.33e- 39 ****
##  9 linearity     240    60    -15.0   91.7 2.31e- 26 2.50e- 26 ****
## 10 spike         240    60     58.8  243.  1.22e-145 3.96e-145 ****
## 11 trend         240    60    -71.8  293.  6.29e-188 4.09e-187 ****
## 12 x_acf1        240    60    -86.5  288.  4.16e-208 5.41e-207 ****
## 13 x_acf10       240    60    -23.0   73.4 2.97e- 35 3.51e- 35 ****
```

```
FeatureDensity(inappropriate_data)
```

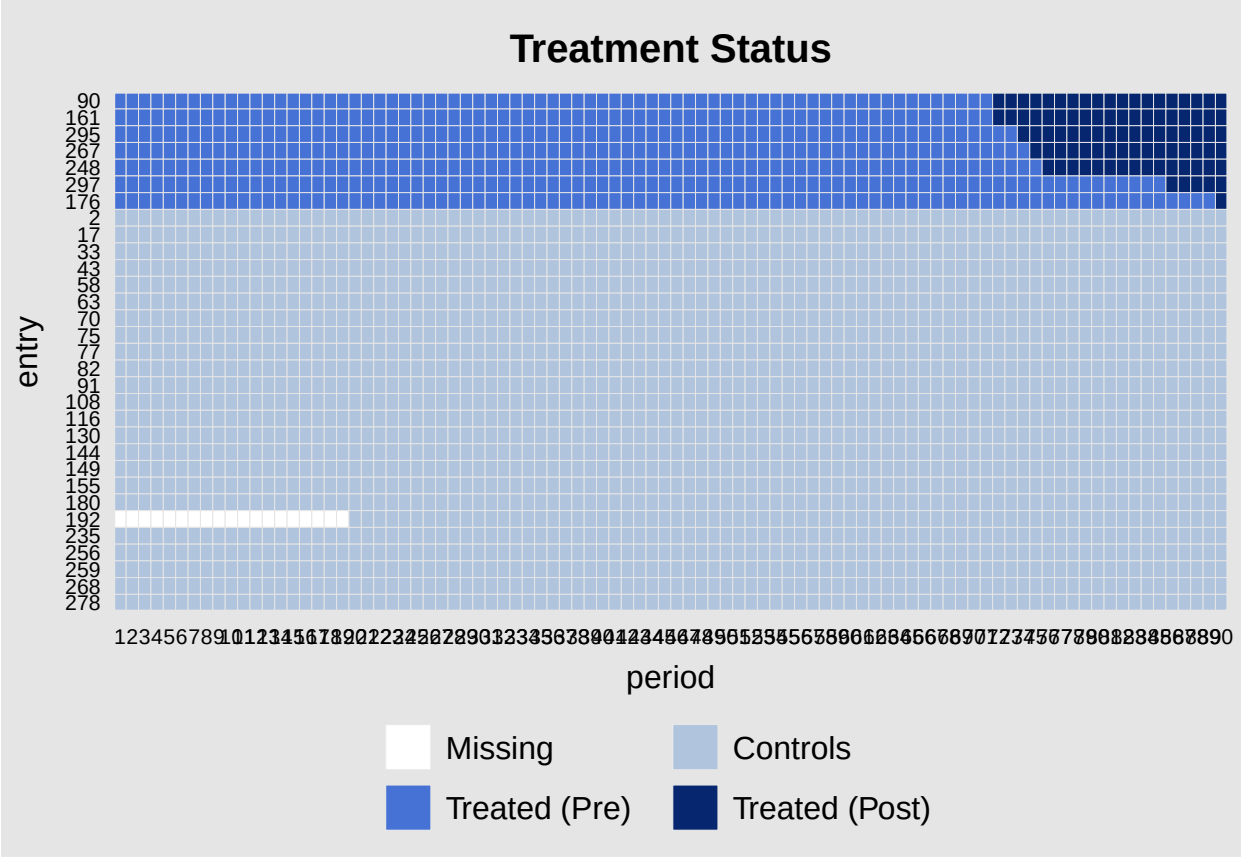


Another couple of visualizations are borrowed from the panelView library – which is loaded in with gsynth. The panelView function has an option to create a grid of dates and units to highlight the number of treated units as well as the number of pre and post treatment periods. Alternatively, the output can be specified as a plot of the raw time series, highlighting the treatment observations. This can be useful for visualizing whether the treatment units seem systematically different than the donors. Both are shown below, again demonstrated for the advisable and inadvisable cases.

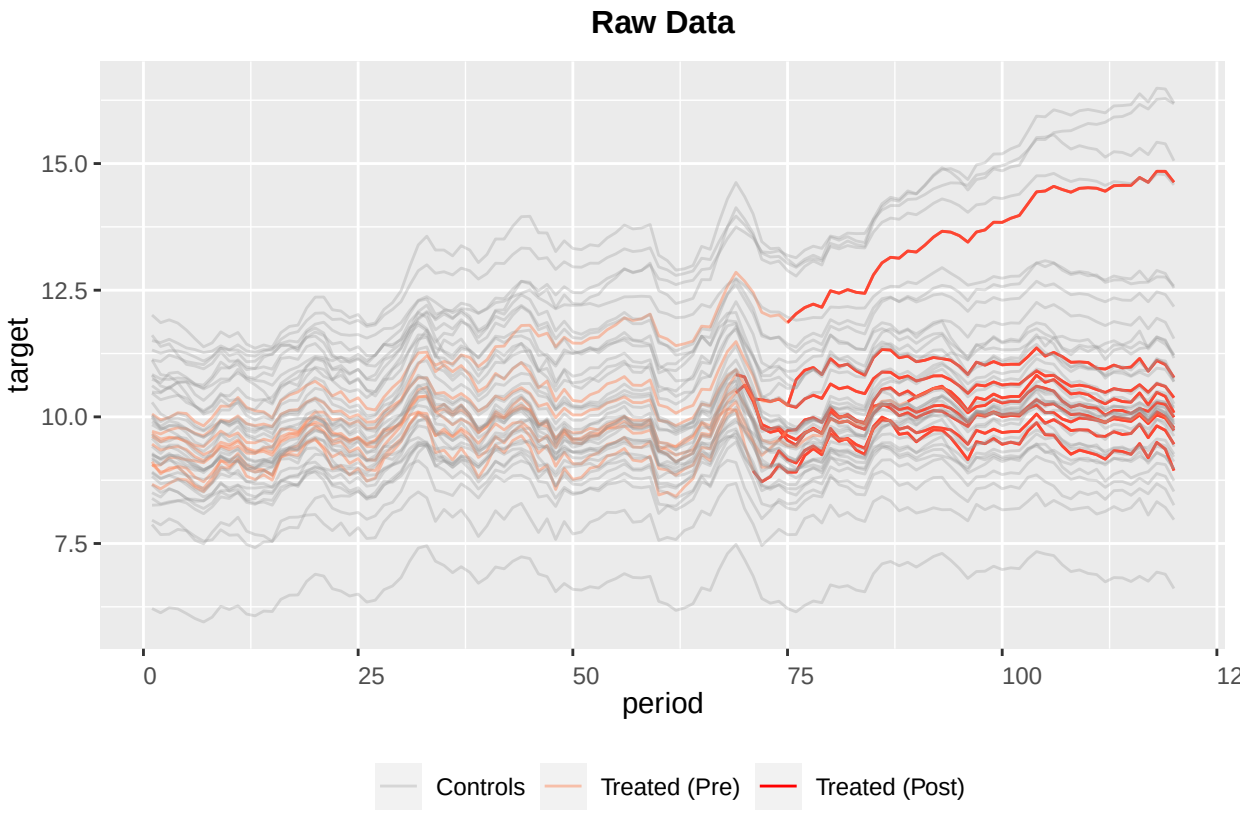
```
require(panelView)
```

```
## Loading required package: panelView
```

```
panelView(target~treatperiod_0,
          data=sim_data %>% filter(entry %in% c(sample(1:units, 30),192),
                                   period %in% 1:90) %>%
            mutate(target=ifelse(entry==192 & period<20, NA, target)) %>%
            as.data.frame(),
          index=c("entry", "period"),
          pre.post = TRUE, by.timing = TRUE)
```




```
panelView(target~treatperiod_0, data=sim_data %>%
            filter(entry %in% sample(1:units, 45)) %>%
            as.data.frame(),
          index=c("entry", "period"),
          type = "outcome")
```




```
panelView(target~treatperiod_0, data=inappropriate_data %>%
            filter(entry %in% sample(1:units, 45)) %>%
            as.data.frame(), index=c("entry", "period"),
          type = "outcome")
```

The panelView grid output displays that there are a large number of pre-treatment periods among our treated units, and can also be used to display missing data if desired (example shown for entry 486). The raw outcome series plots show that, while there are some treated series with larger magnitude than any of the controls in the advisable scenario, the trend and variance levels seem fairly similar across the groups, so our data seem suitable for these methodologies.[2] In the second case, despite the fact that the treated series have similar magnitudes as the donor series, we observe that they are much more volatile, which could lead to imputation problems as the series appear fundamentally different by treatment.[3]

Imputation of Unobserved Potential Outcomes

Once we have convinced ourselves that this suite of methods is appropriate in the context of our data, we proceed to the next step in the workflow: estimating the optimal combination of donor units to predict the outcome of our treated units had they not been treated. Each of the methods in the repository do just that, with their own distinct flavor.[4] The advantage of our repo, as opposed to loading and calling the functions from the packages directly (which exists for Gsynth/MC, SDID and Causal Impact - both of which currently only handle one treated observation, and SDID has further restrictions at the time of writing) is that our wrapper functions standardize the input and output across these methods, allowing for easier comparison and less pre-processing time.

The Gsynth Interactive Fixed Effects (IFE) estimator, one of the favored methods in the benchmarking study, is implemented in the code below. The wrapper to "gsynth()" implemented below can handle a large number of unnamed arguments (examples include the number of factors and the information criteria to

[2] In the original Synthetic Control Method approach, the outcome series would have to fall into the convex hull of the donor units, so that being everywhere larger would be problematic. However, the use of an intercept term in the models relaxes this assumption. See Doudchenko and Imbens (2017) and Abadie (2019) for a discussion.

[3] NOTE TO SELF: do we have a recommended action in this scenario?

[4] To learn more about the methods themselves, consider reading our sister paper (LINK).

select among factors); for more information on which arguments exist, please review the ?gsynth documentation. The calls to other estimation methods follow a similar form, and are included in the snippet below. Several of the estimators have options for additional parameters: CausalImpact can in principle handle user-specified additions in the form of Bayesian Structural Time Series parameters, as well as training period input (neither is currently supported in our wrapper, though it's on the TODO list); SDID has options for constrained vs unconstrained optimization of the unit weights (which can be useful for taking advantage of negative correlations between series with negative weights) as well as arguments for the number of donors to consider (by Euclidean distance), and the number of pre-periods to train on.[5] The SCM implementation in our repository by default adopts the most flexible approach discussed in Doudchenko and Imbens (2017), employing an intercept term and elastic net penalty (CITE TIBSHIRANI ET AL) over a speedy grid search (implemeted by ?bigstatsr) to determine the optimal donor weights for each unit.

```
# Sequential estimation of Gsynth IFE on two data sets.
gsynth_demo <- EstimateGsynthSeries(sim_data)
```

```
## Cross-validating ...
##  r = 0; sigma2 = 0.44485; IC = -0.72445; PC = 0.44114; MSPE = 0.17723
##  r = 1; sigma2 = 0.06094; IC = -2.58472; PC = 0.06301; MSPE = 0.05280
##  r = 2; sigma2 = 0.01345; IC = -3.96845; PC = 0.01448; MSPE = 0.00968
##  r = 3; sigma2 = 0.00666; IC = -4.54487; PC = 0.00746; MSPE = 0.00452
##  r = 4; sigma2 = 0.00413; IC = -4.89868; PC = 0.00479; MSPE = 0.00285
##  r = 5; sigma2 = 0.00156; IC = -5.74371; PC = 0.00188; MSPE = 0.00106
##
##  r* = 5

## Note: Using an external vector in selections is ambiguous.
## i Use `all_of(outcome_var)` instead of `outcome_var` to silence this message.
## i See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.
## This message is displayed once per session.
```

```
gsynth_inapp_demo <- EstimateGsynthSeries(inappropriate_data)
```

```
## Cross-validating ...
##  r = 0; sigma2 = 0.00585; IC = -5.05645; PC = 0.00580; MSPE = 0.08962
##  r = 1; sigma2 = 0.00071; IC = -7.03408; PC = 0.00074; MSPE = 0.07542*
##  r = 2; sigma2 = 0.00013; IC = -8.57922; PC = 0.00014; MSPE = 0.07659
##  r = 3; sigma2 = 0.00000; IC = -14.63315; PC = 0.00000; MSPE = 0.05878*
##  r = 4; sigma2 = 0.00000; IC = -69.51441; PC = 0.00000; MSPE = 0.58506
##  r = 5; sigma2 = 0.00000; IC = -69.38396; PC = 0.00000; MSPE = 0.57481
##
##  r* = 3
```

```
#To parallelize over several data sets:
# sdid_list <- furrr::future_map(.x=list(sim_data, inappropriate_data),
#                                .f=EstimateSDIDSeries)
sdid_demo <- EstimateSDIDSeries(sim_data)
sdid_uncon_demo <- EstimateSDIDSeries(sim_data, constrained = F)
scm_demo <- EstimateSCMSeries(sim_data)
causalimpact_demo <- EstimateCausalImpactSeries(sim_data)

head(gsynth_demo)
```

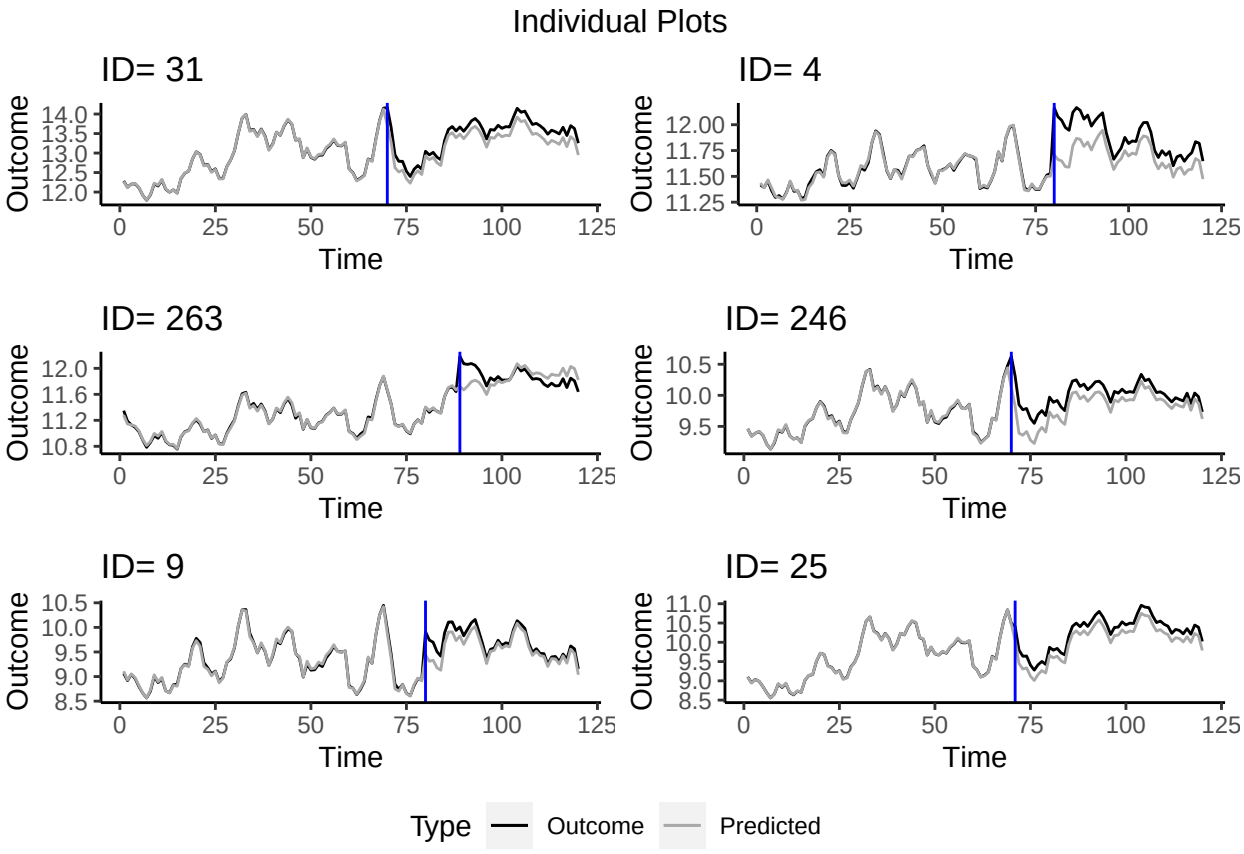
[5] See Doudchenko and Imbens, 2017 for further discussion on the trade-offs between constrained and unconstrained weights in SCM methods more generally.

```
## # A tibble: 6 x 10
##   period entry response point.pred point.effect pct.effect Treatment_Period
##    <dbl> <dbl>    <dbl>      <dbl>        <dbl>      <dbl>            <dbl>
## 1      1     4    11.4       11.4      -0.0195   -0.00171               80
## 2      1     7     8.43       8.39      0.0399    0.00476               77
## 3      1     9     9.07       9.11     -0.0416   -0.00457               80
## 4      1    20    10.0       10.0       0.0313    0.00312               83
## 5      1    25     9.10       9.10      0.00313   0.000344              71
## 6      1    31    12.3       12.3       0.0116    0.000944              70
## # ... with 3 more variables: counter_factual <dbl>, cf_point.effect <dbl>,
## #   cf_pct.effect <dbl>
```

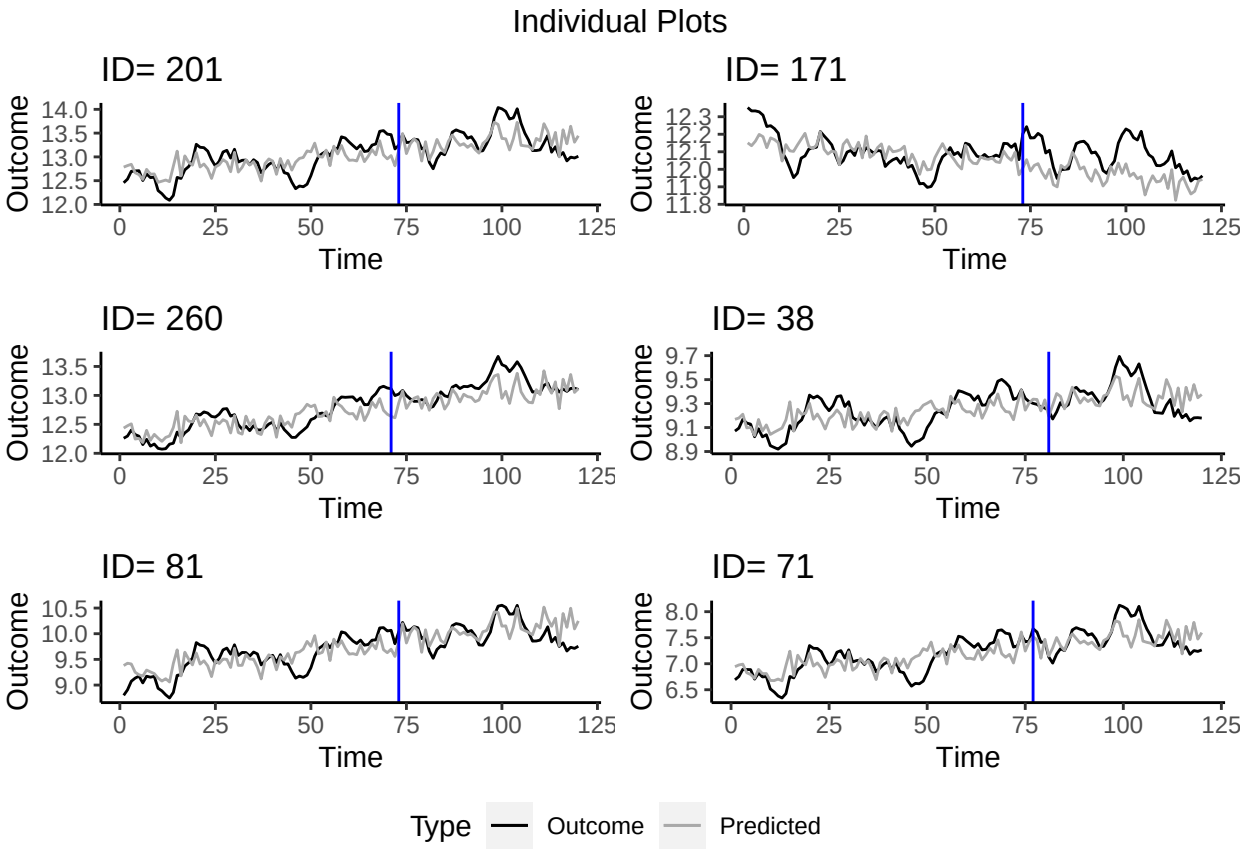
A subset of the resulting tibble is shown below, with reassigned column names of *response* for the actually observed outcome (was *target* before), the *point.pred* as our imputed estimate, *Treatment_Period* for the time in which the treatment is impemented for the given unit. There are also columns for the *point.effect* and *pct.effect*, and in this example (because we have the true counterfactual outcome we are hoping to impute) we have columns for *counter_factual*, *cf_point.effect*, and *cf_pct.effect* which have the true point-wise treatment effects.

Individual imputations in hand, we can plot the true outcome series atop the predicted outcome series for a sampling of our treated units to get a sense of well the donor-combined predictions are tracking the outcomes in the pre-period. The code below takes the estimated series and plots the 3 largest treated series (by first period outcome) and a random sample of 3 additional treated series to get a visual sense. As demonstrated in the plots, the Gsynth IFE estimates on the data that is deemed appropriate for these methods track the outcome quite closely in the pre-treatment periods whereas the estimates on the data less-adviseable for these methods struggles to impute the values because the series between treated and control have quite distinct patterns.

```
AssortedSeriesPlot(gsynth_demo)
```




```
AssortedSeriesPlot(gsynth_inapp_demo)
```

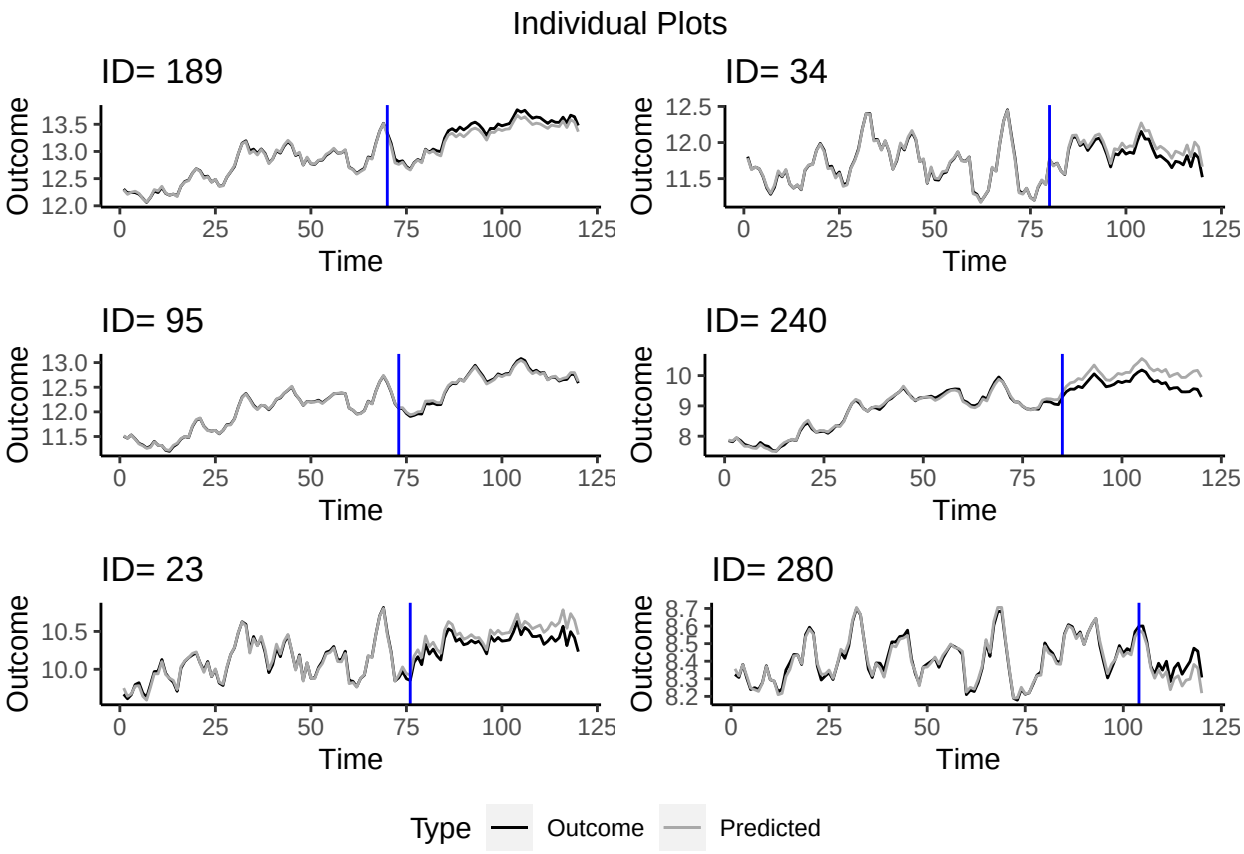


While the pre-period tracking is a good first sign, it's also important to ensure we are not overfitting in these pre-treatment periods. To get a sense of this, one approach is to create a placebo data set and generate estimates of the pre and post *placebo-treatment* periods. Our *CreatePlaceboData* function finds a set of donor units that are most similar to our actually treated units (in terms of the Euclidean distance between the time series), and assigns those units a *placebo-treatment* taking place in the same time period as the intervention for the actual treated unit it is matched to. The creation of an appropriate placebo is particularly important because we rely on the placebo dataset to understand whether we are overfitting, and later, to get a sense of how biased our estimators might be. With this data set formed and subsequently estimated, we can examine the individual series plots and see whether the model accurately imputes the outcome in both the pre and post treatment periods (because the true treatment effect here is known to be 0). The code below outlines this process, with resulting plots that reassure us (at least for the first several post-treatment periods) that we are not simply overfitting to the pre-intervention data.

```
sim_data_placebo <- CreatePlaceboData(sim_data)
gsynth_placebo_demo <- EstimateGsynthSeries(sim_data_placebo)
```

```
## Cross-validating ...
##  r = 0; sigma2 = 0.52462; IC = -0.56191; PC = 0.52025; MSPE = 0.16531
##  r = 1; sigma2 = 0.06456; IC = -2.51928; PC = 0.06697; MSPE = 0.05301
##  r = 2; sigma2 = 0.01428; IC = -3.89109; PC = 0.01547; MSPE = 0.01000
##  r = 3; sigma2 = 0.00710; IC = -4.45467; PC = 0.00801; MSPE = 0.00486
##  r = 4; sigma2 = 0.00424; IC = -4.83503; PC = 0.00498; MSPE = 0.00310
##  r = 5; sigma2 = 0.00166; IC = -5.64155; PC = 0.00202; MSPE = 0.00103
##
##  r* = 5
```

```
AssortedSeriesPlot(gsynth_placebo_demo)
```



Estimation of Average Treatment Effects

The next set of functions in the workflow estimate the treatment effect of interest (typically the Average Treatment Effect on the Treated in post-treatment period t, though we also handle the median). These functions essentially take in the individual point-effects for each unit, map them to a post-treatment period, and average them over these post-treatment periods using jackknife resampling – giving us confidence bounds (CITE JACKKNIFE PAPER).[6]

```
gsynth_demo_att <- ComputeTreatmentEffect(gsynth_demo)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
sdid_demo_att <- ComputeTreatmentEffect(sdid_demo)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

[6]We are aware that the jackknife method is not ideal for this type of inference as the estimate treatment effects across each of the units are not realistically independent. We are planning to implement a conformal inference procedure following Chernozhukov et al (2019), which has a number of enticing properties and seems to be the best and most broadly applicable approach to estimating CIs/p-values in this framework.

```
sdid_uncon_demo_att <- ComputeTreatmentEffect(sdid_uncon_demo)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
scm_demo_att <- ComputeTreatmentEffect(scm_demo)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
causalimpact_demo_att <- ComputeTreatmentEffect(causalimpact_demo)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
head(gsynth_demo_att %>% filter(post_period_t>=0))
```

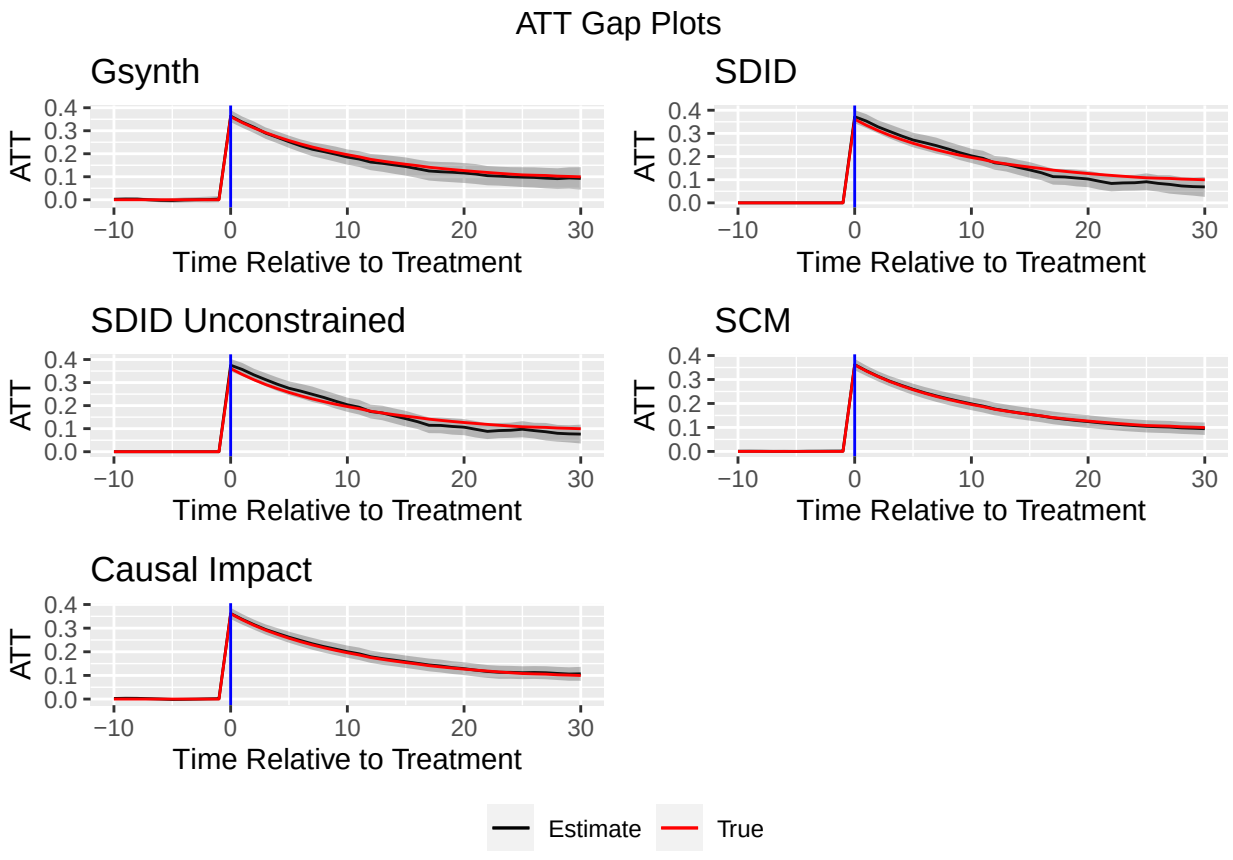
```
## # A tibble: 6 x 12
##   post_period_t treated_n jackknife_lb_me~ jackknife_ub_me~ jackknife_mean_~
##           <dbl>     <int>            <dbl>            <dbl>            <dbl>
## 1             0        60            0.338            0.390            0.364
## 2             0        60            0.338            0.390            0.364
## 3             0        60            0.338            0.390            0.364
## 4             0        60            0.338            0.390            0.364
## 5             0        60            0.338            0.390            0.364
## 6             0        60            0.338            0.390            0.364
## # ... with 7 more variables: observed_mean_abs_att <dbl>,
## #   jackknife_lb_mean_pct_att <dbl>, jackknife_ub_mean_pct_att <dbl>,
## #   jackknife_mean_pct_att <dbl>, observed_mean_pct_att <dbl>,
## #   mean_abs_cf_att <dbl>, mean_pct_cf_att <dbl>
```

The output tibble of from *ComputeTreatmentEffect* contains columns for the time relative to treatment (*post_period_t*, which can be specified but by default ranges from large negative to large positive values); the number of treated units in that period (*treated_n* – this informs us as to how many units we are estimating the treatment effect off of); jackknifed estiamtes of the lower and upper CI bounds (95% default) on the treatment effect of interest alongside the jackknifed and observed treatment effect (these should be equal in means, but may differ for medians). These jackknifed estimates are computed for both absolute treatment effects and percent treatment effects. Lastly, if the data exist (e.g. in the placebo data), the tibble will contain a column for the true treatment effect by comparing the observed treated outcome to the (unobservable) counterfactual outcome.

This treatment effect tibble is more easily understood in graphical form, which can easily be created using the *GapPlot* function within the repo.[7] Because we generally cannot observe the counterfactual (potential untreated) outcome of the treated units unless we have synthetic data (which we are using here), we rely on the placebo ATT estimates to give us a sense of whether our method is biased. The plots below depict a grid of ATT plots for the several methods we have discussed; note that in this case, we make use of the true (typically unobservable) counterfactual to demonstrate that each method is able to accurately recover the treatment effect. However, the plot for the Gsynth estimates of the placebo ATT, which we would typically use to reassure ourselves of limited bias, are also presented.

[7]There's also *IndividualPlotter*, which works much like the *AssortedSeriesPlot* above but takes as input the particular ID number of the unit of interest.

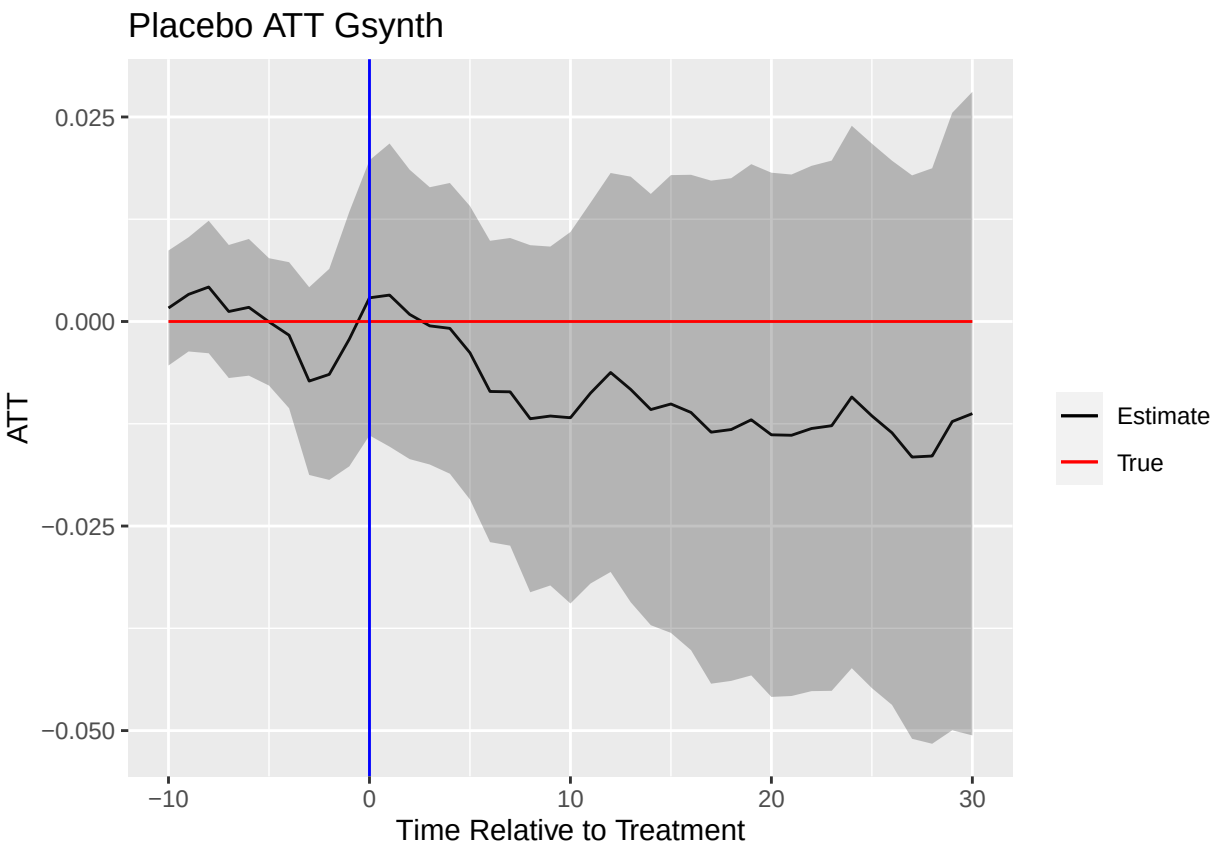
```
gap_plot_list <- furrr::future_map2(.x=list(gsynth_demo_att,sdid_demo_att,
                                            sdid_uncon_demo_att,scm_demo_att,
                                            causalimpact_demo_att),
                                    .y=list("Gsynth", "SDID",
                                            "SDID Unconstrained", "SCM",
                                            "Causal Impact"),
                                    .f=~GapPlot(att_tib=.x,
                                                plot_title=.y,
                                                plot_y_lab="ATT"))


att_grid_out <- ggpubr::ggarrange(
  plotlist = gap_plot_list, ncol = 2, nrow = 3,
  common.legend = TRUE, legend = "bottom"
)
ggpubr::annotate_figure(att_grid_out, top = text_grob(
    paste("ATT Gap Plots")
  )) %>% print()
```




```
gsynth_placebo_demo %>%
  ComputeTreatmentEffect() %>%
  GapPlot(plot_title="Placebo ATT Gsynth",
          plot_y_lab="ATT")
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```



Ensembling the Estimators

An additional feature in this repository is the functionality for an ensemble estimator. In particular, this can be implemented after estimating the imputed series (*EstimateXSeries* above) by calling *EstimateEnsemble*. This function works by first creating a placebo data set from the raw data,[8] applying each of the methods to this placebo data and extracting the point predictions for each treated series, then finding the optimal combination of weights that minimize the Mean Squared Error between the actual post-treatment outcome and the combined prediction (with options for an intercept term, constraints so that the weights both sum to 1 and are non-negative, and the choice of whether to find a set weights for each unit or one set of weights for the whole data).[9] Once this ensemble is estimated, it can be treated just like the EstimateXSeries output for any of the other methods, so that ensemble treatment effects and CIs can be computed and plotted.

Shown next is the code to estimate the ensemble. It takes as arguments the name of the methods (must be specified as the X in EstimateXSeries – e.g. "Gsynth"), the raw data, and a list combining the estimated series (ideally, in the order that the methods are named).

```
# Bug: When num_methods > num_cores, it loses the definition of the function.
plan(sequential, workers=availableCores()-2)
EstimateEnsemble(method_names = c("Gsynth", "SDID", "SCM"),
                 true_data = sim_data,
```

[8]The placebo is currently not independent of the donor pool for the treated units, which can cause issues in estimating the CI for the ensemble.

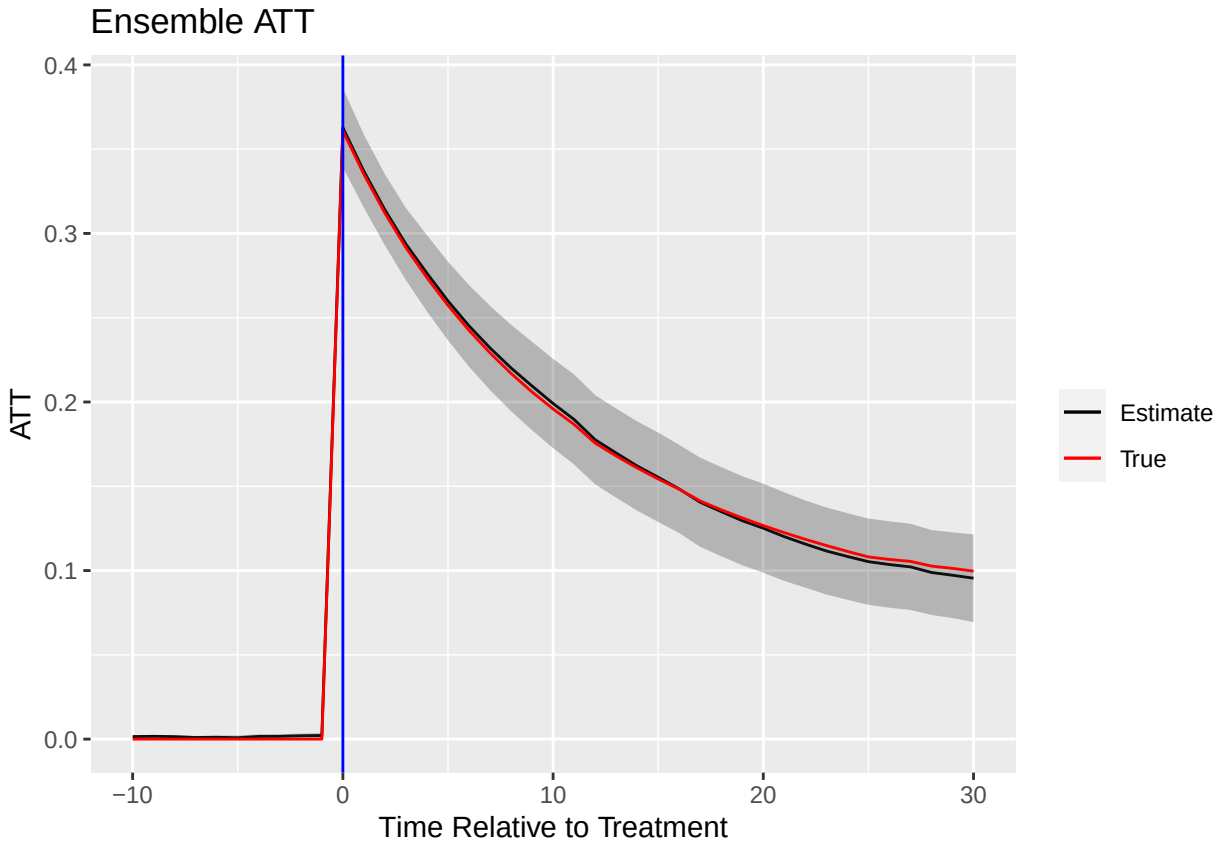
[9]Potential extension – would using 50 draws from the placebo help at all? Would that allow us to get better inference/variance? Would this help with reducing the noise from the weights, if we average the weights over the 50 draws??

```
                pred_list = list(
                  gsynth_demo, sdid_demo, scm_demo
                )) %>%
  ComputeTreatmentEffect() %>%
  GapPlot(plot_title = "Ensemble ATT", plot_y_lab = "ATT")
```

```
## Cross-validating ...
##  r = 0; sigma2 = 0.52403; IC = -0.56303; PC = 0.51966; MSPE = 0.16862
##  r = 1; sigma2 = 0.06392; IC = -2.52927; PC = 0.06630; MSPE = 0.05637
##  r = 2; sigma2 = 0.01402; IC = -3.90982; PC = 0.01518; MSPE = 0.01116
##  r = 3; sigma2 = 0.00708; IC = -4.45743; PC = 0.00799; MSPE = 0.00492
##  r = 4; sigma2 = 0.00423; IC = -4.83624; PC = 0.00497; MSPE = 0.00315
##  r = 5; sigma2 = 0.00161; IC = -5.66728; PC = 0.00197; MSPE = 0.00118
##
##  r* = 5
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```



Debiased ATT

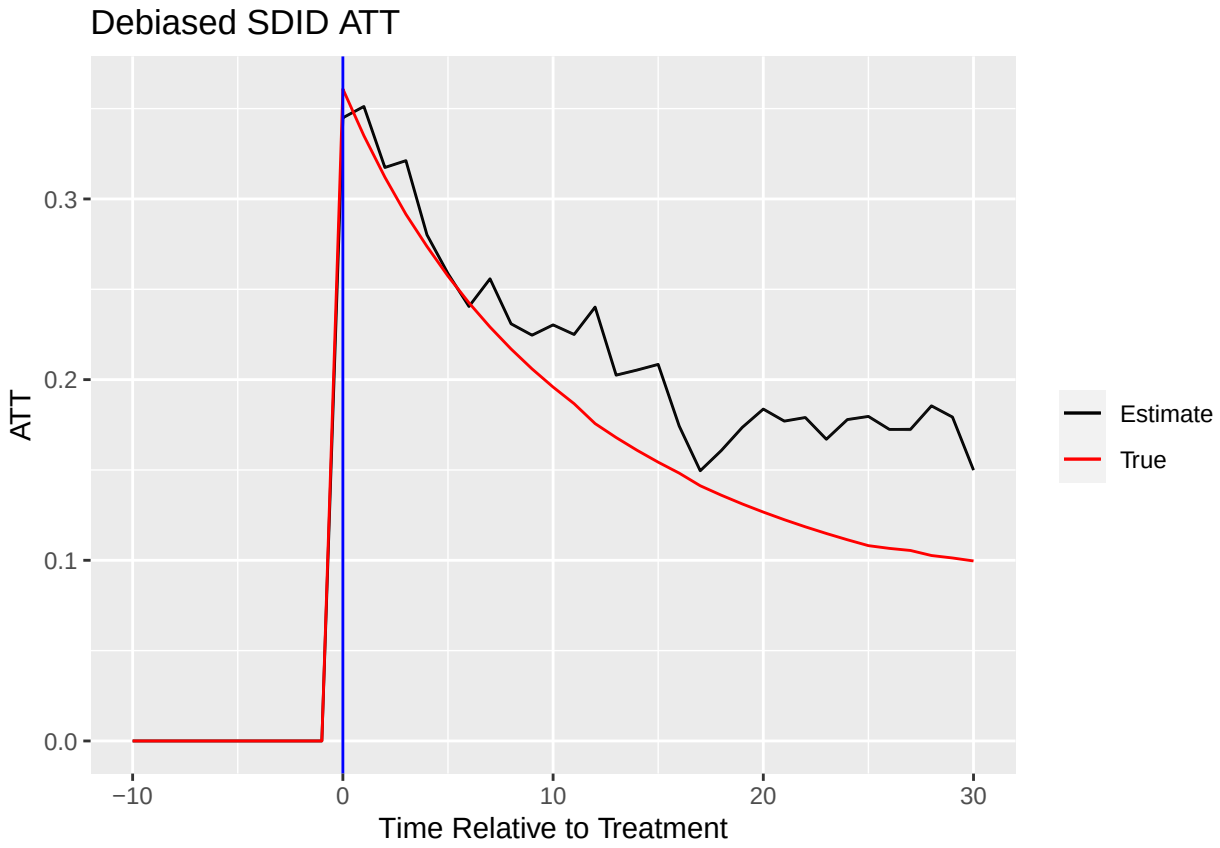
We implement a debiased ATT estimator by taking advantage of the ATT estimate among the placebo group and using that to adjust the ATT estimates on our true data.[10] This is implemented by identifying the set of data we will use for our placebo experiment, shocking these data with noise over a user-inputted number of seeds (e.g. 50 seeds will create 50 placebo data sets, each with slightly different outcome processes due to the iid noise), and estimating the ATT for each of these placebo data. Then, after estimating the ATT for the true data, we can compute a debiased ATT series by computing the difference between the true ATT and the placebo ATT for each data set (where placebo ATT is really the estimator bias in that data). We then find the mean of this debiased ATT over the data sets, and compute confidence intervals by jackknifing.

```
plan(sequential, workers=availableCores()-2)
DebiasedATT(raw_data = sim_data, method_name = "SDID", num_placebos = 10) %>%
  GapPlot(plot_title = "Debiased SDID ATT", plot_y_lab = "ATT")
```

```
## Joining, by = "entry"

## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)

## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)

## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)

## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)

## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)

## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)

## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)
```

[10] Importantly, to achieve proper CI and de-biasing, we would ideally separate the placebo set from the donor set prior to estimating either. This may not always be computationally feasible, especially in cases where there are only a few donor units relative to treated units. In these cases, we can use the same set of data for both, but we must recognize that this may not provide us with much of a correction, and could potentially lead to worse estimates because the of the noise in the placebo modelling.

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)

## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)

## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)

## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)

## `summarise()` ungrouping output (override with `.groups` argument)

## Note: Using an external vector in selections is ambiguous.
## i Use `all_of(abs_predvar)` instead of `abs_predvar` to silence this message.
## i See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.
## This message is displayed once per session.

## Note: Using an external vector in selections is ambiguous.
## i Use `all_of(pct_predvar)` instead of `pct_predvar` to silence this message.
## i See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.
## This message is displayed once per session.
```



Auto Estimate

For convenience, we provide a function that automatically determines the optimal method given the particular user inputted data (and the desired methods to try). Once again, this method relies heavily on the placebo treatment assignment generated based on the true data – if the two are very similar, we can be more confident that the optimal method on the placebo data is likely the optimal method on the true data. In particular, taking a string of method names, the function generates a large number of placebo data and estimates (in parallel, for computational benefit) each method on each placebo dataset. Currently, the average mean squared error across the data sets (for a given horizon) is the criteria by which we determine the best method.[11] Future iterations will consider bias and perhaps even coverage, as this is often quite important in applied settings. Moreover, we plan to break the execution of the auto-estimator if certain diagnostic tests – introduced in the first stages of the workflow – are not adequately met (e.g. density of several features is quite different; inadequate coverage with respect to the PC plots; too many t-test are failed). [12]

The output of this function is a predicted series of imputations (which can be passed on to the treatment effect or plots – it's the equivalent of *gsynth_demo*) as well as a message specifying the best method given the parameters (number of placebos, horizon, metric – RMSE for now). An example of a call to this function is shown below.

```
AutoEstimator(raw_data = sim_data, method_names = c("Gsynth", "SDID"), num_placebos = 10)
```

```
## Joining, by = "entry"

## Cross-validating ...
##  r = 0; sigma2 = 0.47910; IC = -0.66760; PC = 0.47511; MSPE = 0.16610
##  r = 1; sigma2 = 0.09571; IC = -1.94154; PC = 0.11633; MSPE = 0.08964
##  r = 2; sigma2 = 0.05397; IC = -2.18229; PC = 0.07770; MSPE = 0.05220
##  r = 3; sigma2 = 0.04649; IC = -2.00409; PC = 0.07738; MSPE = 0.04966
##  r = 4; sigma2 = 0.04369; IC = -1.74318; PC = 0.08256; MSPE = 0.04887*
##  r = 5; sigma2 = 0.04158; IC = -1.47407; PC = 0.08798; MSPE = 0.04920
##
##  r* = 4
##
## Cross-validating ...
##  r = 0; sigma2 = 0.48127; IC = -0.66309; PC = 0.47726; MSPE = 0.17017
##  r = 1; sigma2 = 0.09644; IC = -1.93394; PC = 0.11721; MSPE = 0.09000
##  r = 2; sigma2 = 0.05312; IC = -2.19813; PC = 0.07648; MSPE = 0.05672
##  r = 3; sigma2 = 0.04607; IC = -2.01308; PC = 0.07668; MSPE = 0.05619
##  r = 4; sigma2 = 0.04242; IC = -1.77253; PC = 0.08018; MSPE = 0.05595
##  r = 5; sigma2 = 0.04039; IC = -1.50316; PC = 0.08546; MSPE = 0.05413
##
##  r* = 5
##
## Cross-validating ...
##  r = 0; sigma2 = 0.48187; IC = -0.66183; PC = 0.47786; MSPE = 0.16718
##  r = 1; sigma2 = 0.09792; IC = -1.91868; PC = 0.11902; MSPE = 0.08704
##  r = 2; sigma2 = 0.05348; IC = -2.19152; PC = 0.07699; MSPE = 0.05224
##  r = 3; sigma2 = 0.04590; IC = -2.01674; PC = 0.07640; MSPE = 0.05129*
##  r = 4; sigma2 = 0.04254; IC = -1.76978; PC = 0.08040; MSPE = 0.05131
##  r = 5; sigma2 = 0.04076; IC = -1.49405; PC = 0.08624; MSPE = 0.04974
##
```

[11] We could also think about bias, though naively averaging the bias can be problematic as bias of -10 in time 1 and 10 in time 2 would look quite good. Perhaps take the average of the squared or absolute bias.

[12] Thanks Jarod M for the suggestion.

```
##  r* = 5
##
## Cross-validating ...
##  r = 0; sigma2 = 0.47956; IC = -0.66665; PC = 0.47556; MSPE = 0.16303
##  r = 1; sigma2 = 0.09489; IC = -1.95018; PC = 0.11533; MSPE = 0.08677
##  r = 2; sigma2 = 0.05196; IC = -2.22034; PC = 0.07480; MSPE = 0.05104
##  r = 3; sigma2 = 0.04537; IC = -2.02848; PC = 0.07551; MSPE = 0.04850
##  r = 4; sigma2 = 0.04231; IC = -1.77509; PC = 0.07997; MSPE = 0.04827
##  r = 5; sigma2 = 0.04019; IC = -1.50810; PC = 0.08504; MSPE = 0.04770
##
##  r* = 5
##
## Cross-validating ...
##  r = 0; sigma2 = 0.48428; IC = -0.65685; PC = 0.48025; MSPE = 0.16706
##  r = 1; sigma2 = 0.09843; IC = -1.91354; PC = 0.11963; MSPE = 0.09085
##  r = 2; sigma2 = 0.05404; IC = -2.18110; PC = 0.07780; MSPE = 0.05386
##  r = 3; sigma2 = 0.04631; IC = -2.00790; PC = 0.07708; MSPE = 0.05244*
##  r = 4; sigma2 = 0.04296; IC = -1.75987; PC = 0.08120; MSPE = 0.05252
##  r = 5; sigma2 = 0.04015; IC = -1.50915; PC = 0.08495; MSPE = 0.05166
##
##  r* = 5
##
## Cross-validating ...
##  r = 0; sigma2 = 0.48659; IC = -0.65210; PC = 0.48253; MSPE = 0.16319
##  r = 1; sigma2 = 0.09901; IC = -1.90769; PC = 0.12033; MSPE = 0.08932
##  r = 2; sigma2 = 0.05364; IC = -2.18854; PC = 0.07722; MSPE = 0.04951
##  r = 3; sigma2 = 0.04550; IC = -2.02555; PC = 0.07573; MSPE = 0.04758*
##  r = 4; sigma2 = 0.04188; IC = -1.78545; PC = 0.07915; MSPE = 0.04771
##  r = 5; sigma2 = 0.03972; IC = -1.51985; PC = 0.08405; MSPE = 0.04727
##
##  r* = 5
##
## Cross-validating ...
##  r = 0; sigma2 = 0.48128; IC = -0.66307; PC = 0.47727; MSPE = 0.16501
##  r = 1; sigma2 = 0.09894; IC = -1.90833; PC = 0.12025; MSPE = 0.09072
##  r = 2; sigma2 = 0.05488; IC = -2.16560; PC = 0.07901; MSPE = 0.05566
##  r = 3; sigma2 = 0.04765; IC = -1.97928; PC = 0.07932; MSPE = 0.05402*
##  r = 4; sigma2 = 0.04469; IC = -1.72038; PC = 0.08447; MSPE = 0.05405
##  r = 5; sigma2 = 0.04241; IC = -1.45448; PC = 0.08973; MSPE = 0.05200
##
##  r* = 5
##
## Cross-validating ...
##  r = 0; sigma2 = 0.48398; IC = -0.65748; PC = 0.47995; MSPE = 0.16487
##  r = 1; sigma2 = 0.09697; IC = -1.92847; PC = 0.11786; MSPE = 0.08931
##  r = 2; sigma2 = 0.05440; IC = -2.17436; PC = 0.07832; MSPE = 0.05248
##  r = 3; sigma2 = 0.04688; IC = -1.99567; PC = 0.07803; MSPE = 0.05089*
##  r = 4; sigma2 = 0.04336; IC = -1.75079; PC = 0.08194; MSPE = 0.05108
##  r = 5; sigma2 = 0.04058; IC = -1.49843; PC = 0.08587; MSPE = 0.04965
##
##  r* = 5
##
## Cross-validating ...
##  r = 0; sigma2 = 0.48462; IC = -0.65616; PC = 0.48058; MSPE = 0.16690
```

```
##  r = 1; sigma2 = 0.09694; IC = -1.92874; PC = 0.11783; MSPE = 0.08882
##  r = 2; sigma2 = 0.05492; IC = -2.16481; PC = 0.07907; MSPE = 0.05164
##  r = 3; sigma2 = 0.04662; IC = -2.00122; PC = 0.07760; MSPE = 0.05138
##  r = 4; sigma2 = 0.04369; IC = -1.74306; PC = 0.08257; MSPE = 0.05115
##  r = 5; sigma2 = 0.04158; IC = -1.47408; PC = 0.08798; MSPE = 0.05015
##
##  r* = 5
##
## Cross-validating ...
##  r = 0; sigma2 = 0.48207; IC = -0.66143; PC = 0.47805; MSPE = 0.16435
##  r = 1; sigma2 = 0.09709; IC = -1.92718; PC = 0.11801; MSPE = 0.08773
##  r = 2; sigma2 = 0.05332; IC = -2.19437; PC = 0.07677; MSPE = 0.05057
##  r = 3; sigma2 = 0.04638; IC = -2.00627; PC = 0.07721; MSPE = 0.04970
##  r = 4; sigma2 = 0.04399; IC = -1.73619; PC = 0.08314; MSPE = 0.04951
##  r = 5; sigma2 = 0.04162; IC = -1.47317; PC = 0.08806; MSPE = 0.04796
##
##  r* = 5
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` regrouping output by 'post_period_t' (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## Cross-validating ...
##  r = 0; sigma2 = 0.44485; IC = -0.72445; PC = 0.44114; MSPE = 0.17723
##  r = 1; sigma2 = 0.06094; IC = -2.58472; PC = 0.06301; MSPE = 0.05280
##  r = 2; sigma2 = 0.01345; IC = -3.96845; PC = 0.01448; MSPE = 0.00968
##  r = 3; sigma2 = 0.00666; IC = -4.54487; PC = 0.00746; MSPE = 0.00452
##  r = 4; sigma2 = 0.00413; IC = -4.89868; PC = 0.00479; MSPE = 0.00285
##  r = 5; sigma2 = 0.00156; IC = -5.74371; PC = 0.00188; MSPE = 0.00106
##
##  r* = 5
```

```
## Gsynth was RMSE minimizing estimator on 10 draws of the placebo given a horizon from 0 to 4 relative
```

```
## # A tibble: 7,200 x 10
##    period entry response point.pred point.effect pct.effect Treatment_Period
##     <dbl> <dbl>    <dbl>      <dbl>        <dbl>      <dbl>             <dbl>
##  1      1     4    11.4       11.4      -0.0195   -0.00171                80
##  2      1     7     8.43       8.39      0.0399    0.00476                77
##  3      1     9     9.07       9.11     -0.0416   -0.00457                80
##  4      1    20    10.0       10.0       0.0313    0.00312                83
##  5      1    25     9.10       9.10      0.00313   0.000344               71
##  6      1    31    12.3       12.3       0.0116    0.000944               70
##  7      1    35     9.54       9.51      0.0282    0.00297                72
##  8      1    38     8.79       8.73      0.0610    0.00699                95
##  9      1    46    11.2       11.1       0.0575    0.00518                71
## 10      1    55    11.0       11.0      -0.0186   -0.00168                77
## # ... with 7,190 more rows, and 3 more variables: counter_factual <dbl>,
## #   cf_point.effect <dbl>, cf_pct.effect <dbl>
```