demo_optic

OPTIC Core

This project contains the simulation code for OPTIC. This README's purpose is to tell a user how to install the package and get it up and running with the following simulation scenarios:

1. Concurrent Simulations
2. No Confounding Simulations

Installing Package

Clone Clone the repository to your machine. Once you have the project on your local machine, the first thing you want to do is build and install the package. Open the project in RStudio - easiest to just open the optic.Rproj file. Make sure you have `devtools` installed.

```
# fresh build of the package documentation and source code
devtools::document()
devtools::build()

# install optic package and dependencies
devtools::install()
```

1. Running Concurrent Simulation

The example below uses the `example_data` dataset provided with the package, which is derived from data from the US Bureau of Labor Statistics and the Centers for Disease Control and Prevention.

The following can be used to setup any of the simulations (Concurrent, No Confounding), but throughout this section we will use the concurrent simulations as an example.

Configure Simulations The first step is to configure the simulation or set of simulations you wish to run. There four main components to the configuration object:

1. **Basic Parameters**: this includes straightforward arguments like the data, number of iterations, and verbosity
2. **Models**: models are provided to the configuration method in a nested list where each model you want to test/use is a list element with the necessary named values based on the simulation being run (e.g., concurrent or selection bias)
3. **Methods**: the configuration object requires at least three methods - `method_sample`, `method_model`, and `method_results`. These "slots", if you will, are intended for the user to provide their sampling, treatment identification, and common data transformation code; modeling code; and code to extract results, fit information, summary statistics, etc. respectively. There are also two optional slots - `method_premodel` and `method_postmodel` that if provided will be applied to the simulation run immediately before and immediately after the `method_model`. See the section on "How an Iteration is Run" below.
4. **Iterable and Global Parameters**: The `params` argument is where you provide the named values that will be expanded into a full set of all unique combinations of all named elements in the provided list. The `globals` argument should also be a list of named elements that are statically stored for access by any of the methods and will not be expanded with the `params` values.

Here we will walk through an example of setting up the configuration object, exploring the configuration object prior to running the job, and looking at how a single simulation is defined. For our example, we will look at evaluating a linear 2-way fixed effects model compared to a linear autoregressive model under the context of having two co-occurring policies.

```
data(example_data)
x <- example_data

# we will define two scenarios for different effect magnitudes for
# the policies using 5, 10, and 15 percent changes in the outcome
linear5 <- 690 / ((sum(x$population) / length(unique(x$year))) / 100000)
linear10 <- 1380 / ((sum(x$population) / length(unique(x$year))) / 100000)
linear15 <- 2070 / ((sum(x$population) / length(unique(x$year))) / 100000)

scenario1 <- c(linear10, linear10)
scenario2 <- c(linear5, linear15)
```

Let's first set up our models object. Note that the methods applied here are already defined in `R/concurrent-methods.R`. For each element in the list, we will define the same named attributes: - name: a name for the model to identify it in the output - type: this is used in some of our methods to transform the data correctly for the type of model being run (e.g. if set to "autoreg" the run will control for a single lag of the outcome) - model_call: the R function name - model_formula: the formula to run with the specified model_call - model_args: will be passed as additional arguments when running model - se_adjust: in this case our post-processing can perform some standard error adjustments and we only want two here

```
my_models <- list(
  list(
    name="fixedeff_linear",
    type="reg",
    model_call="lm",
    model_formula=crude.rate ~ unemploymentrate + as.factor(year) + as.factor(state) + treatment1_level
    model_args=list(weights=as.name('population')),
    se_adjust=c("none", "cluster")
  ),
  list(
    name="autoreg_linear",
    type="autoreg",
    model_call="lm",
    model_formula=deaths ~ unemploymentrate + as.factor(year) + treatment1_change + treatment2_change,
    model_args=list(weights=as.name('population')),
    se_adjust=c("none", "cluster")
  )
)
```

Now we will setup the configuration object that will be used to dispatch each of the simulations. In this case we will tell the configuration function our data, the models we specified above, and the number of iterations we want for each simulation. In a production or real simulation we probably want 5,000 iterations, but we are using 10 here for illustrative purposes. We pass each of the models defined in `R/concurrent-methods.R` to their respective arguments, then define the `params` that will be expanded and we will compare with this simulation. This simulation does not have any global variables.

```
sim_config <- configure_simulation(
  x=x,
  models=my_models,
  iters=10,
```

```
  method_sample=concurrent_sample,
  method_pre_model=concurrent_premodel,
  method_model=concurrent_model,
  method_post_model=concurrent_postmodel,
  method_results=concurrent_results,

  params=list(
    unit_var="state",
    time_var="year",
    effect_magnitude=list(scenario1, scenario2),
    n_units=c(30),
    effect_direction=c("null", "neg"),
    policy_speed=c("instant", "slow"),
    n_implementation_periods=c(3),
    rhos=c(0, 0.25, 0.5, 0.75, 0.9),
    years_apart=2,
    ordered=TRUE
  )
)
```

Once the configuration object is created, you can take a look at a couple of key things: 1. Did the `params` expand as you expected? 1. Setup a single simulation to get a sense for how it works.

```
# if we look at object, it is an R6 class object and has a number of named elements
> class(sim_config)
[1] "SimConfig" "R6"

> names(sim_config)
 [1] ".__enclos_env__"         "method_results"          "method_post_model"
 [4] "method_model"            "method_pre_model"        "method_sample"
 [7] "simulation_params"       "globals"                 "params"
[10] "iters"                   "models"                  "data"
[13] "clone"                   "print"                   "setup_single_simulation"
[16] "initialize"

# the "simulation_params" element is the expanded data frame
> head(sim_config$simulation_params)
  unit_var time_var effect_magnitude1 effect_magnitude2 n_units effect_direction policy_speed n_impleme
1    state     year         0.4550925         0.4550925      30             null      instant
2    state     year         0.2275463         0.6826388      30             null      instant
3    state     year         0.4550925         0.4550925      30              neg      instant
4    state     year         0.2275463         0.6826388      30              neg      instant
5    state     year         0.4550925         0.4550925      30             null         slow
6    state     year         0.2275463         0.6826388      30             null         slow

# The "setup_single_simulation" element is an internal method for this class.
# It takes an integer in 1:nrow(simulation_params), pulls the values from that row
# in the table and all the static values in the object and creates a single
# simulation object:
example_single_sim <- sim_config$setup_single_simulation(1)

> names(example_single_sim)
 [1] "unit_var"                "time_var"                "effect_magnitude1"
 [4] "effect_magnitude2"       "n_units"                 "effect_direction"
```

```
 [7] "policy_speed"             "n_implementation_periods" "rhos"
[10] "years_apart"              "ordered"                  "data"
[13] "models"                   "iters"                    "method_sample"
[16] "method_pre_model"         "method_model"             "method_post_model"
[19] "method_results"
```

Dispatch Concurrent Simulation To run a simulation, use the `dispatch_simulation` method, which will iterate over the rows in the `"simulation_params"` element and run the iterations for each simulation either in a loop or in parallel depending on how you choose to dispatch the job.

```
# use_future will run the iterations in parallel. In this case we are also showing
# how you may need to specify certain environment inclusions like a specific method
# or package to ensure it is used in each of the threads the future library will
# create.
# You can also set a seed, which will work for both single-threaded and multi-
# threaded runs to ensure replication is possible.
lm_results <- dispatch_simulations(
    sim_config,
    use_future=TRUE,
    seed=9782,
    verbose=2,
    future.globals=c("cluster_adjust_se"),
    future.packages=c("dplyr", "optic")
)

# we get back a list with a data.frame of results for each of the 40 unqiue
# combinations of parameters we provided in the configuration object
> length(lm_results)
[1] 40

# take a look at one of the data.frames if you would like
head(lm_results[[1]])
```

How an Iteration is Run When constructing methods, it is important to understand how they are going to be implemented. It's relatively straightforward and you can check out the `R/run-iteration.R` program for a quick look. Essentially, each method you write should take a single list object and return a single list object that has all the information needed for subsequent methods. Only the `method_results` function should return the "final product" - in the above case a data.frame. Here is the logic:

A `single_simulation` object (list) is created and provided to the `run_iteration` function. The `run_iteration` function goes through the methods provided by the user in the following order:

1. The `method_sample` function is applied to the list and the returned object overwrites `single_simulation`. This is the sampling and common data transformation step. Everything after this step will be looped and performed once per model element provided in the models argument - in our example above there are two models elements.
2. Then, the `run_iteration` method enters a loop - looping over each of the models provided, creates a new `model_simulation` object that is a copy of the `single_simulation` object. This isolates each model to start from the same initial sampling and data transformation so any further data altering or changes to the object by other methods are not retained/impacting subsequent models' runs. The rest of the methods are then run in order:
    1. If provided, runs `method_pre_model` passing `model_simulation` as the sole argument, the return object is stored in `model_simulation`
    2. Runs the `method_model` function passing `model_simulation` as the sole argument and overwriting

the object with the returned object
    3. If provided, runs the `method_post_model`, the return object is assigned to a list whose named element is the model name
    4. If no post model method is provided, the return object from the `method_model` function is assigned to the list directly
3. The list of independent model results is provided as the argument to `method_results` and the return object from this final function is returned by the `run_iteration` function.

2. No Confounding Simulations

You can use the same setup as shown above to run the No Confounding Simulations. The following is example code to run the Two-Way Fixed Effects model for the no confounding simulations. This example again uses the `example_data` dataset provided with the package.

The first snippet of code displays variable cleanup to prepare for running the simulations for the opioid prescribing outcome. A user will have their own startup code to get up and running with the simulation package.

```
x <- x %>%
  arrange(state, year) %>%
  group_by(state) %>%
  mutate(lag1 = lag(opioid_rx, n=1L),
         lag2 = lag(opioid_rx, n=2L),
         lag3 = lag(opioid_rx, n=3L)) %>%
  ungroup() %>%
  rowwise() %>%
  # code in moving average and trend versions of prior control
  mutate(prior_control_mva3_OLD = mean(c(lag1, lag2, lag3)),
         prior_control_trend_OLD = lag1 - lag3) %>%
  ungroup() %>%
  dplyr::select(-lag1, -lag2, -lag3) %>%
  mutate(state = factor(as.character(state)))

linear0 <- 0
linear5 <- .05*mean(x$opioid_rx, na.rm=T)
linear15 <- .15*mean(x$opioid_rx, na.rm=T)
linear25 <- .25*mean(x$opioid_rx, na.rm=T)
```

The linear0, linear5, linear15, and linear25 represent the effect magnitudes for each simulation. The next step is to specify the 2-way fixed effects model, the model call associated with your model, model formula, and other model arguments.

```
linear_models <- list(
  list(
    name="fixedeff_linear",
    type="reg",
    model_call="lm",
    model_formula=opioid_rx ~ treatment_level + unemploymentrate + as.factor(year) + as.factor(state),
    model_args=list(weights=as.name("population")),# NULL
    se_adjust=c("none", "cluster")
  )
)
```

We now setup our configuration object for the simulation. This configuration object requires the data, model list, and the number of iterations. The list of parameters are unique to the no confounding simulation. The `unit_var` represents the unit of analysis and the `treat_var` represents the treatment unit. In these

simulations, we have state-level data and the treatment assignment is generated at the treatment-level. In addition, we have specified the `time_var` to be the variable in `x` that indicates the time object. In this case it is year. The effect magnitudes were specified in the previous code snippet and the number of units `n_units` needs to be the number of treated units generated. The remaining parameters include the effect direction, policy speed, the number of implementation periods, and the prior control (moving average and trend).

```
linear_fe_config <- configure_simulation(
  # data and models required
  x=x,
  models=list(linear_models[[1]]),
  # iterations
  iters=5000,

  # specify functions or S3 class of set of functions
  method_sample=noconf_sample,
  method_pre_model=noconf_premodel,
  method_model=noconf_model,
  method_post_model=noconf_postmodel,
  method_results=noconf_results,

  globals=NULL,#list(), # no globals as of now.

  # parameters that will be expanded and added
  params=list(
    unit_var="state",
    treat_var="state",
    time_var="year",
    effect_magnitude=list(linear0, linear5, linear15, linear25),
    n_units=c(1, 5, 15, 30), #2%, 10%, 30%, 60%
    effect_direction=c("neg"),
    policy_speed=list("instant"),
    n_implementation_periods=list(0),
    prior_control=c("mva3", "trend")
  )
)
```

The above will produce the following automatically:

```
Number of Simulations: 32
Number of Models: 1
Iteration per Simulation : 5000
Total number of Iterations to Run: 160000
hey, that's a lot of iterations! we recommend using the parallel options when dispatching this job.
```

You can see that the outcome prints to screen the number of simulations your specification plans to run, the number of models (i.e. 2-way FE is considered one model but if you were to also run the autoregressive model the number would be 2 reported here), the number of iterations per simulation, and the total number of iterations to run. The package will print a message telling you to utilize the parallel options if you have a lot of iterations to run.

Next, we will double check to make sure the simulations were setup correctly:

```
class(linear_fe_config)
[1] "SimConfig" "R6"

> names(linear_fe_config)
```

```
 [1] ".__enclos_env__"         "method_results"          "method_post_model"
 [4] "method_model"            "method_pre_model"        "method_sample"
 [7] "simulation_params"       "globals"                 "params"
[10] "iters"                   "models"                  "data"
[13] "clone"                   "print"                   "setup_single_simulation"
[16] "initialize"

> head(linear_fe_config$simulation_params)
  unit_var treat_var time_var effect_magnitude n_units effect_direction policy_speed
1    state     state     year         0.000000       1              neg      instant
2    state     state     year         4.091773       1              neg      instant
3    state     state     year        12.275318       1              neg      instant
4    state     state     year        20.458864       1              neg      instant
5    state     state     year         0.000000       5              neg      instant
6    state     state     year         4.091773       5              neg      instant
  n_implementation_periods prior_control
1                        0          mva3
2                        0          mva3
3                        0          mva3
4                        0          mva3
5                        0          mva3
6                        0          mva3

# Look at the setup for a single simulation:
example_single_sim_NoConf <- linear_fe_config$setup_single_simulation(1)

> names(example_single_sim_NoConf)
 [1] "unit_var"                 "treat_var"                "time_var"
 [4] "effect_magnitude"         "n_units"                  "effect_direction"
 [7] "policy_speed"             "n_implementation_periods" "prior_control"
[10] "data"                     "models"                   "iters"
[13] "method_sample"            "method_pre_model"         "method_model"
[16] "method_post_model"        "method_results"
```

Everything looks correct so we now can dispatch our simulations, using the future package, with the following code:

```
# dispatch with the same seed (want the same sampled data each run)
linear_fe_r <- dispatch_simulations(linear_fe_config,
                                    use_future=TRUE,
                                    seed=89721,
                                    verbose=2,
                                    future.globals=c("cluster_adjust_se"),
                                    future.packages=c("MASS", "dplyr", "optic"))

# clean up results
linear_fe_results <- do.call(rbind, linear_fe_r)
rownames(linear_fe_results) <- NULL
```