Running multiple analyses at once using the CohortMethod package

Martijn J. Schuemie, Marc A. Suchard and Patrick Ryan

2024-05-31

Contents

1 Introduction **1**

2 General approach **1**

3 Preparation for the example **2**

4 Specifying hypotheses of interest **5**

5 Specifying analyses **6**
5.1 Covariate balance 9

6 Executing multiple analyses **9**
6.1 Restarting 10

7 Retrieving the results **10**
7.1 Empirical calibration and negative control distribution 11

8 Exporting to CSV **17**

9 View results in a Shiny app **18**

10 Acknowledgments **18**

1 Introduction

In this vignette we focus on running several different analyses on several target-comparator-outcome combinations. This can be useful when we want to explore the sensitivity to analyses choices, include controls, or run an experiment similar to the OMOP experiment to empirically identify the optimal analysis choices for a particular research question.

This vignette assumes you are already familiar with the `CohortMethod` package and are able to perform single studies. We will walk through all the steps needed to perform an exemplar set of analyses, and we have selected the well-studied topic of the effect of coxibs versus non-selective nonsteroidal anti-inflammatory drugs (NSAIDs) on gastrointestinal (GI) bleeding-related hospitalization. For simplicity, we focus on one coxib – celecoxib – and one non-selective NSAID – diclofenac. We will execute various variations of an analysis for the primary outcome and a large set of negative control outcomes.

2 General approach

The general approach to running a set of analyses is that you specify all the function arguments of the functions you would normally call, and create sets of these function arguments. The final outcome models as

well as intermediate data objects will all be saved to disk for later extraction.

An analysis will be executed by calling these functions in sequence:

1. `getDbCohortMethodData()`
2. `createStudyPopulation()`
3. `createPs()` (optional)
4. `trimByPs()` or `trimByPsToEquipoise()` (optional)
5. `matchOnPs()`, `matchOnPsAndCovariates()`, `stratifyByPs()`, or `stratifyByPsAndCovariates()` (optional)
6. `computeCovariateBalance()` (optional)
7. `fitOutcomeModel()` (optional)

When you provide several analyses to the `CohortMethod` package, it will determine whether any of the analyses have anything in common, and will take advantage of this fact. For example, if we specify several analyses that only differ in the way the outcome model is fitted, then `CohortMethod` will extract the data and fit the propensity model only once, and re-use this in all the analyses.

The function arguments you need to define have been divided into four groups:

1. **Hypothesis of interest**: arguments that are specific to a hypothesis of interest, in the case of the cohort method this is a combination of target, comparator, and outcome.
2. **Analyses**: arguments that are not directly specific to a hypothesis of interest, such as the washout window, whether to include drugs as covariates, etc.
3. Arguments that are the output of a previous function in the `CohortMethod` package, such as the `cohortMethodData` argument of the `createPs` function. These cannot be specified by the user.
4. Arguments that are specific to an environment, such as the connection details for connecting to the server, and the name of the schema holding the CDM data.

There are a two arguments (`excludedCovariateConceptIds`, and `includedCovariateConceptIds` of the `getDbCohortMethodData()` function) that can be argued to be part both of group 1 and 2. These arguments are therefore present in both groups, and when executing the analysis the union of the two lists of concept IDs will be used.

3 Preparation for the example

We need to tell R how to connect to the server where the data are. `CohortMethod` uses the `DatabaseConnector` package, which provides the `createConnectionDetails` function. Type `?createConnectionDetails` for the specific settings required for the various database management systems (DBMS). For example, one might connect to a PostgreSQL database using this code:

```
connectionDetails <- createConnectionDetails(dbms = "postgresql",
                                             server = "localhost/ohdsi",
                                             user = "joe",
                                             password = "supersecret")

cdmDatabaseSchema <- "my_cdm_data"
cohortDatabaseSchema <- "my_results"
cohortTable <- "my_cohorts"
options(sqlRenderTempEmulationSchema = NULL)
```

The last few lines define the `cdmDatabaseSchema`, `cohortDatabaseSchema`, and `cohortTable` variables. We'll use these later to tell R where the data in CDM format live, and where we want to write intermediate tables. Note that for Microsoft SQL Server, databaseschemas need to specify both the database and the schema, so for example `cdmDatabaseSchema <- "my_cdm_data.dbo"`. For database platforms that do not support temp tables, such as Oracle, it is also necessary to provide a schema where the user has write access that can be used to emulate temp tables. PostgreSQL supports temp tables, so we can set

`options(sqlRenderTempEmulationSchema = NULL)` (or not set the `sqlRenderTempEmulationSchema` at all.)

We need to define the exposures and outcomes for our study. Here, we will define our exposures using the OHDSI `Capr` package. We define two cohorts, one for celecoxib and one for diclofenac. For each cohort we require a prior diagnosis of 'osteoarthritis of knee', and 365 days of continuous prior observation. we restrict to the first exposure per person:

```
library(Capr)

osteoArthritisOfKneeConceptId <- 4079750
celecoxibConceptId <- 1118084
diclofenacConceptId <- 1124300
osteoArthritisOfKnee <- cs(
  descendants(osteoArthritisOfKneeConceptId),
  name = "Osteoarthritis of knee"
)
attrition = attrition(
  "prior osteoarthritis of knee" = withAll(
    atLeast(1, conditionOccurrence(osteoArthritisOfKnee),
            duringInterval(eventStarts(-Inf, 0)))
  )
)
celecoxib <- cs(
  descendants(celecoxibConceptId),
  name = "Celecoxib"
)
diclofenac  <- cs(
  descendants(diclofenacConceptId),
  name = "Diclofenac"
)
celecoxibCohort <- cohort(
  entry = entry(
    drugExposure(celecoxib, firstOccurrence()),
    observationWindow = continuousObservation(priorDays = 365)
  ),
  attrition = attrition,
  exit = exit(endStrategy = drugExit(celecoxib,
                                     persistenceWindow = 30,
                                     surveillanceWindow = 0))
)
diclofenacCohort <- cohort(
  entry = entry(
    drugExposure(diclofenac, firstOccurrence()),
    observationWindow = continuousObservation(priorDays = 365)
  ),
  attrition = attrition,
  exit = exit(endStrategy = drugExit(diclofenac,
                                     persistenceWindow = 30,
                                     surveillanceWindow = 0))
)
# Note: this will automatically assign cohort IDs 1 and 2, respectively:
exposureCohorts <- makeCohortSet(celecoxibCohort, diclofenacCohort)
```

We'll pull the outcome definition from the OHDSI `PhenotypeLibrary`:

```
library(PhenotypeLibrary)
outcomeCohorts <- getPlCohortDefinitionSet(77) # GI bleed
```

In addition to the outcome of interest, we also want to include a large set of negative control outcomes:

```
negativeControlIds <- c(29735, 140673, 197494,
                        198185, 198199, 200528, 257315,
                        314658, 317376, 321319, 380731,
                        432661, 432867, 433516, 433701,
                        433753, 435140, 435459, 435524,
                        435783, 436665, 436676, 442619,
                        444252, 444429, 4131756, 4134120,
                        4134454, 4152280, 4165112, 4174262,
                        4182210, 4270490, 4286201, 4289933)
negativeControlCohorts <- tibble(
  cohortId = negativeControlIds,
  cohortName = sprintf("Negative control %d", negativeControlIds),
  outcomeConceptId = negativeControlIds
)
```

We combine the exposure and outcome cohort definitions, and use `CohortGenerator` to generate the cohorts:

```
allCohorts <- bind_rows(outcomeCohorts,
                        exposureCohorts)

library(CohortGenerator)
cohortTableNames <- getCohortTableNames(cohortTable = cohortTable)
createCohortTables(connectionDetails = connectionDetails,
                   cohortDatabaseSchema = cohortDatabaseSchema,
                   cohortTableNames = cohortTableNames)
generateCohortSet(connectionDetails = connectionDetails,
                  cdmDatabaseSchema = cdmDatabaseSchema,
                  cohortDatabaseSchema = cohortDatabaseSchema,
                  cohortTableNames = cohortTableNames,
                  cohortDefinitionSet = allCohorts)
generateNegativeControlOutcomeCohorts(
  connectionDetails = connectionDetails,
  cdmDatabaseSchema = cdmDatabaseSchema,
  cohortDatabaseSchema = cohortDatabaseSchema,
  cohortTable = cohortTable,
  negativeControlOutcomeCohortSet = negativeControlCohorts
)
```

If all went well, we now have a table with the cohorts of interest. We can see how many entries per cohort:

```
connection <- DatabaseConnector::connect(connectionDetails)
sql <- "SELECT cohort_definition_id, COUNT(*) AS count FROM @cohortDatabaseSchema.@cohortTable GROUP BY
DatabaseConnector::renderTranslateQuerySql(
  connection = connection,
  sql = sql,
  cohortDatabaseSchema = cohortDatabaseSchema,
  cohortTable = cohortTable
)
DatabaseConnector::disconnect(connection)
```

```
##    COHORT_DEFINITION_ID    COUNT
```

```
## 1                    2   176675
## 2               432867 31548471
## 3                   77   733601
## 4               442619      772
## 5               317376    67020
## 6                    1   109307
## 7               257315   918842
## 8              4182210  3650875
## 9              4174262   910846
## 10              380731    80707
## 11              444429        6
## 12              140673  6344090
## 13               29735   288675
## 14              436665   623024
## 15             4134454    15103
## 16              321319   971627
## 17              435140     1337
## 18              432661      990
## 19              197494   104253
## 20              433701    81191
## 21              435783   154047
## 22              433753   410743
## 23              314658  3032511
## 24              200528   878778
## 25              435524   308750
## 26             4286201   202230
## 27              436676   368572
## 28             4134120    43760
## 29              198199   215695
## 30              433516   214450
## 31              435459   140988
```

4 Specifying hypotheses of interest

The first group of arguments define the target, comparator, and outcome. Here we demonstrate how to create one set, and add that set to a list:

```
outcomeOfInterest <- createOutcome(outcomeId = 77,
                                   outcomeOfInterest = TRUE)
negativeControlOutcomes <- lapply(
  negativeControlIds,
  function(outcomeId) createOutcome(outcomeId = outcomeId,
                                    outcomeOfInterest = FALSE,
                                    trueEffectSize = 1)
)
tcos <- createTargetComparatorOutcomes(
  targetId = 1,
  comparatorId = 2,
  outcomes = append(list(outcomeOfInterest),
                    negativeControlOutcomes)
)
targetComparatorOutcomesList <- list(tcos)
```

We first define the outcome of interest (GI-bleed, cohort ID 77), explicitly stating this is an outcome of

interest (`outcomeOfInterest = TRUE`), meaning we want the full set of artifacts generated for this outcome. We then create a set of negative control outcomes. Because we specify `outcomeOfInterest = FALSE`, many of the artifacts will not be saved (like the matched population), or even not generated at all (like the covariate balance). This can save a lot of compute time and disk space. We also provide the true effect size for these controls, which will be used later for empirical calibration. We set the target to be celecoxib (cohort ID 1), and the comparator to be diclofenac (cohort ID 2).

A convenient way to save `targetComparatorOutcomesList` to file is by using the `saveTargetComparatorOutcomesList` function, and we can load it again using the `loadTargetComparatorOutcomesList` function.

5 Specifying analyses

The second group of arguments are not specific to a hypothesis of interest, and comprise the majority of arguments. For each function that will be called during the execution of the analyses, a companion function is available that has (almost) the same arguments. For example, for the `trimByPs()` function there is the `createTrimByPsArgs()` function. These companion functions can be used to create the arguments to be used during execution:

```
covarSettings <- createDefaultCovariateSettings(
  excludedCovariateConceptIds = c(1118084, 1124300),
  addDescendantsToExclude = TRUE
)

getDbCmDataArgs <- createGetDbCohortMethodDataArgs(
  washoutPeriod = 183,
  restrictToCommonPeriod = FALSE,
  firstExposureOnly = TRUE,
  removeDuplicateSubjects = "remove all",
  studyStartDate = "",
  studyEndDate = "",
  covariateSettings = covarSettings
)

createStudyPopArgs <- createCreateStudyPopulationArgs(
  removeSubjectsWithPriorOutcome = TRUE,
  minDaysAtRisk = 1,
  riskWindowStart = 0,
  startAnchor = "cohort start",
  riskWindowEnd = 30,
  endAnchor = "cohort end"
)

fitOutcomeModelArgs1 <- createFitOutcomeModelArgs(modelType = "cox")
```

Any argument that is not explicitly specified by the user will assume the default value specified in the function. We can now combine the arguments for the various functions into a single analysis:

```
cmAnalysis1 <- createCmAnalysis(
  analysisId = 1,
  description = "No matching, simple outcome model",
  getDbCohortMethodDataArgs = getDbCmDataArgs,
  createStudyPopArgs = createStudyPopArgs,
  fitOutcomeModelArgs = fitOutcomeModelArgs1
)
```

Note that we have assigned an analysis ID (1) to this set of arguments. We can use this later to link the results back to this specific set of choices. We also include a short description of the analysis.

We can easily create more analyses, for example by using matching, stratification, inverse probability of treatment weighting, or by using more sophisticated outcome models:

```
createPsArgs <- createCreatePsArgs() # Use default settings only

matchOnPsArgs <- createMatchOnPsArgs(maxRatio = 100)

computeSharedCovBalArgs <- createComputeCovariateBalanceArgs()

computeCovBalArgs <- createComputeCovariateBalanceArgs(
  covariateFilter = getDefaultCmTable1Specifications()
)

fitOutcomeModelArgs2 <- createFitOutcomeModelArgs(
  modelType = "cox",
  stratified = TRUE
)

cmAnalysis2 <- createCmAnalysis(
  analysisId = 2,
  description = "Matching",
  getDbCohortMethodDataArgs = getDbCmDataArgs,
  createStudyPopArgs = createStudyPopArgs,
  createPsArgs = createPsArgs,
  matchOnPsArgs = matchOnPsArgs,
  computeSharedCovariateBalanceArgs = computeSharedCovBalArgs,
  computeCovariateBalanceArgs = computeCovBalArgs,
  fitOutcomeModelArgs = fitOutcomeModelArgs2
)

stratifyByPsArgs <- createStratifyByPsArgs(numberOfStrata = 5)

cmAnalysis3 <- createCmAnalysis(
  analysisId = 3,
  description = "Stratification",
  getDbCohortMethodDataArgs = getDbCmDataArgs,
  createStudyPopArgs = createStudyPopArgs,
  createPsArgs = createPsArgs,
  stratifyByPsArgs = stratifyByPsArgs,
  computeSharedCovariateBalanceArgs = computeSharedCovBalArgs,
  computeCovariateBalanceArgs = computeCovBalArgs,
  fitOutcomeModelArgs = fitOutcomeModelArgs2
)

fitOutcomeModelArgs3 <- createFitOutcomeModelArgs(
  modelType = "cox",
  inversePtWeighting = TRUE
)

cmAnalysis4 <- createCmAnalysis(
  analysisId = 4,
  description = "Inverse probability weighting",
```

```
  getDbCohortMethodDataArgs = getDbCmDataArgs,
  createStudyPopArgs = createStudyPopArgs,
  createPsArgs = createPsArgs,
  fitOutcomeModelArgs = fitOutcomeModelArgs3
)
```

```
## Note: Using propensity scores but not computing covariate balance
```

```
fitOutcomeModelArgs4 <- createFitOutcomeModelArgs(
  useCovariates = TRUE,
  modelType = "cox",
  stratified = TRUE
)

cmAnalysis5 <- createCmAnalysis(
  analysisId = 5,
  description = "Matching plus full outcome model",
  getDbCohortMethodDataArgs = getDbCmDataArgs,
  createStudyPopArgs = createStudyPopArgs,
  createPsArgs = createPsArgs,
  matchOnPsArgs = matchOnPsArgs,
  fitOutcomeModelArgs = fitOutcomeModelArgs4
)
```

```
## Note: Using propensity scores but not computing covariate balance
```

```
interactionCovariateIds <- c(
  8532001, # Female
  201826210, # T2DM
  21600960413 # concurrent use of antithrombotic agents
)

fitOutcomeModelArgs5 <- createFitOutcomeModelArgs(
  modelType = "cox",
  stratified = TRUE,
  interactionCovariateIds = interactionCovariateIds
)

cmAnalysis6 <- createCmAnalysis(
  analysisId = 6,
  description = "Stratification plus interaction terms",
  getDbCohortMethodDataArgs = getDbCmDataArgs,
  createStudyPopArgs = createStudyPopArgs,
  createPsArgs = createPsArgs,
  stratifyByPsArgs = stratifyByPsArgs,
  fitOutcomeModelArgs = fitOutcomeModelArgs5
)
```

```
## Note: Using propensity scores but not computing covariate balance
```

These analyses can be combined in a list:

```
cmAnalysisList <- list(cmAnalysis1,
                       cmAnalysis2,
                       cmAnalysis3,
                       cmAnalysis4,
```

```
                          cmAnalysis5,
                          cmAnalysis6)
```

A convenient way to save `cmAnalysisList` to file is by using the `saveCmAnalysisList` function, and we can load it again using the `loadCmAnalysisList` function.

5.1 Covariate balance

In our code, we specified that covariate balance must be computed for some of our analysis. For computational reasons, covariate balance has been split into two: We can compute covariate balance for each target-comparator-outcome-analysis combination, and we can compute covariate balance for each target-comparator-analysis, so across all outcomes. The latter is referred to as 'shared covariate balance'. Since there can be many outcomes, it is often not feasible to recompute (or store) balance for all covariates for each outcome. Moreover, the differences between study populations for the various outcomes are likely very small; the only differences will arise from removing those having the outcome prior, which will exclude different people from the study population depending on the outcome. We therefore typically compute the balance for all covariates across all outcomes (shared balance), and only for a small subset of covariates for each outcome. In the code above, we use all covariates for the shared balance computation, which we typically use to evaluate whether our analysis achieved covariate balance. We limit the covariates for the per-outcome balance computations to only those used for the standard 'table 1' definition used in the `getDefaultCmTable1Specifications()` function, which we can use to create a 'table 1' for each outcome.

6 Executing multiple analyses

We can now run the analyses against the hypotheses of interest using the `runCmAnalyses()` function. This function will run all specified analyses against all hypotheses of interest, meaning that the total number of outcome models is `length(cmAnalysisList) * length(targetComparatorOutcomesList)` (if all analyses specify an outcome model should be fitted). Note that we do not want all combinations of analyses and hypothesis to be computed, we can can skip certain analyses by using the `analysesToExclude` argument of the `runCmAnalyses()`.

```
multiThreadingSettings <- createDefaultMultiThreadingSettings(parallel::detectCores())

result <- runCmAnalyses(
  connectionDetails = connectionDetails,
  cdmDatabaseSchema = cdmDatabaseSchema,
  exposureDatabaseSchema = cohortDatabaseSchema,
  exposureTable = cohortTable,
  outcomeDatabaseSchema = cohortDatabaseSchema,
  outcomeTable = cohortTable,
  outputFolder = folder,
  cmAnalysisList = cmAnalysisList,
  targetComparatorOutcomesList = targetComparatorOutcomesList,
  multiThreadingSettings = multiThreadingSettings
)
```

In the code above, we first specify how many parallel threads `CohortMethod` can use. Many of the computations can be computed in parallel, and providing more than one CPU core can greatly speed up the computation. Here we specify `CohortMethod` can use all the CPU cores detected in the system (using the `parallel::detectCores()` function).

We call `runCmAnalyses()`, providing the arguments for connecting to the database, which schemas and tables to use, as well as the analyses and hypotheses of interest. The `folder` specifies where the outcome models and intermediate files will be written.

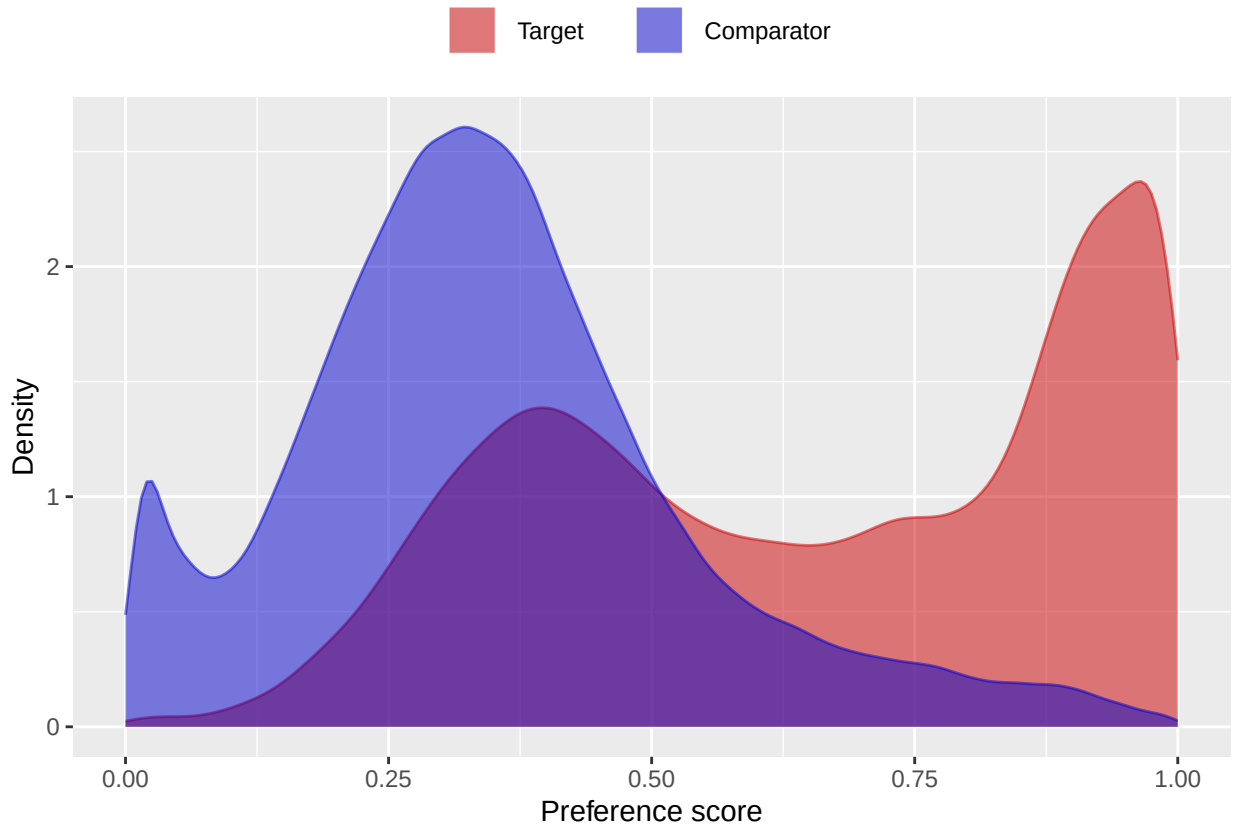
6.1 Restarting

If for some reason the execution was interrupted, you can restart by re-issuing the `runCmAnalyses()` command. Any intermediate and final products that have already been completed and written to disk will be skipped.

7 Retrieving the results

The result of the `runCmAnalyses()` is a data frame with one row per target-target-outcome-analysis combination. It provides the file names of the intermediate and end-result files that were constructed. For example, we can retrieve and plot the propensity scores for the combination of our target, comparator, outcome of interest, and last analysis:

```
psFile <- result %>%
  filter(targetId == 1,
         comparatorId == 2,
         outcomeId == 77,
         analysisId == 5) %>%
  pull(psFile)
ps <- readRDS(file.path(folder, psFile))
plotPs(ps)
```



Note that some of the file names will appear several times in the table. For example, analysis 3 and 5 only differ in terms of the outcome model, and will share the same propensity score and stratification files.

We can always retrieve the file reference table again using the `getFileReference()` function:

```
result <- getFileReference(folder)
```

We can get a summary of the results using `getResultsSummary()`:

```
resultsSum <- getResultsSummary(folder)
resultsSum
```

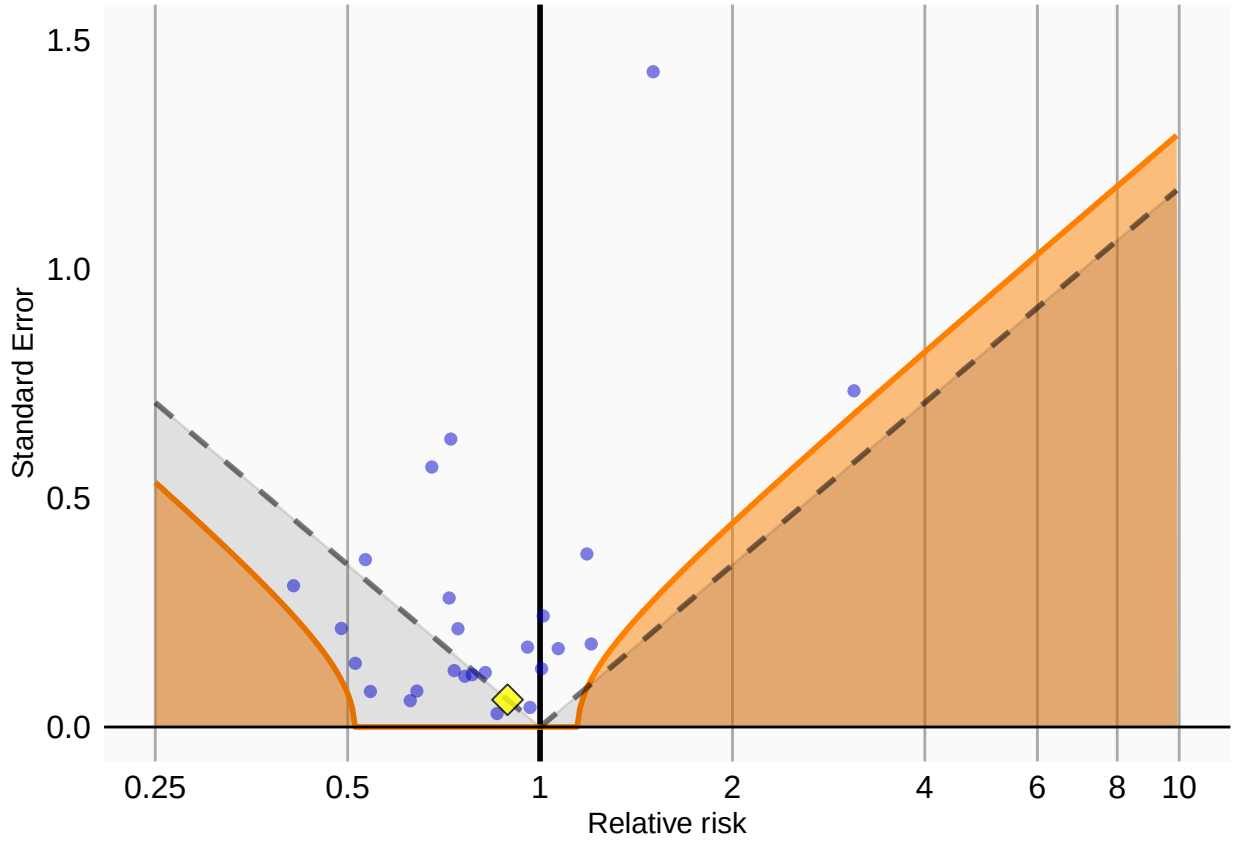
```
## # A tibble: 216 x 29
##    analysisId targetId comparatorId outcomeId trueEffectSize targetSubjects comparatorSubjects targe
##         <int>    <int>        <int>     <int>          <dbl>          <int>              <int>      
##  1          1        1            2        77             NA          84392             147052   1164
##  2          1        1            2     29735              1          86648             152055   1203
##  3          1        1            2    140673              1          81984             140789   1156
##  4          1        1            2    197494              1          87506             154512   1215
##  5          1        1            2    198185              1          87665             155007   1217
##  6          1        1            2    198199              1          87123             153565   1208
##  7          1        1            2    200528              1          87003             153149   1207
##  8          1        1            2    257315              1          86649             152232   1201
##  9          1        1            2    314658              1          78225             133312   1068
## 10          1        1            2    317376              1          87534             154542   1215
## # i 206 more rows
## # i 13 more variables: logRr <dbl>, seLogRr <dbl>, llr <dbl>, mdrr <dbl>, targetEstimator <chr>, cali
## #   calibratedOneSidedP <dbl>, calibratedLogRr <dbl>, calibratedSeLogRr <dbl>, ease <dbl>
```

This tells us, per target-comparator-outcome-analysis combination, the estimated relative risk and 95% confidence interval, as well as the number of people in the treated and comparator group (after trimming and matching if applicable), and the number of outcomes observed for those groups within the specified risk windows.

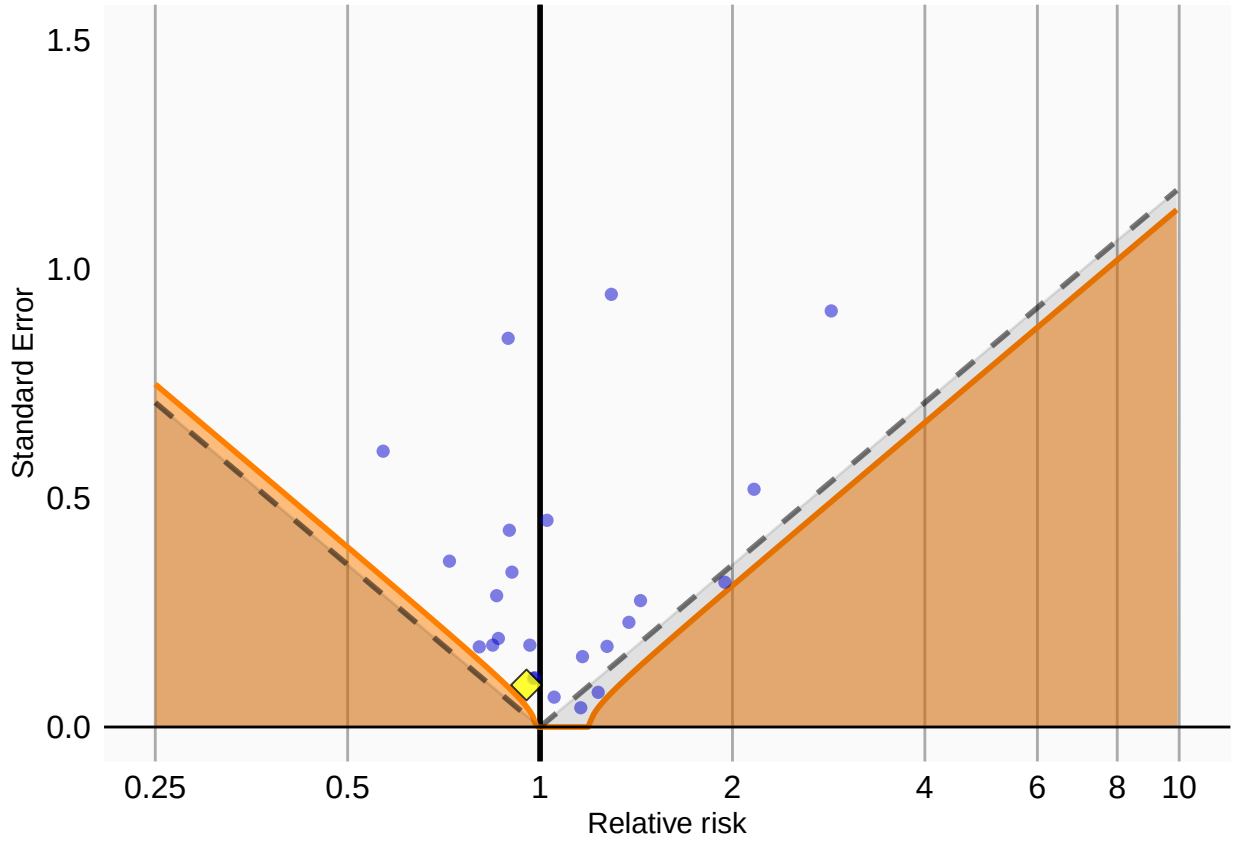
7.1 Empirical calibration and negative control distribution

Because our study included negative control outcomes, our analysis summary also contains calibrated confidence intervals and p-values. We can also create the calibration effect plots for every analysis ID. In each plot, the blue dots represent our negative control outcomes, and the yellow diamond represents our health outcome of interest: GI bleed. An unbiased, well-calibrated analysis should have 95% of the negative controls between the dashed lines (ie. 95% should have $p > .05$).

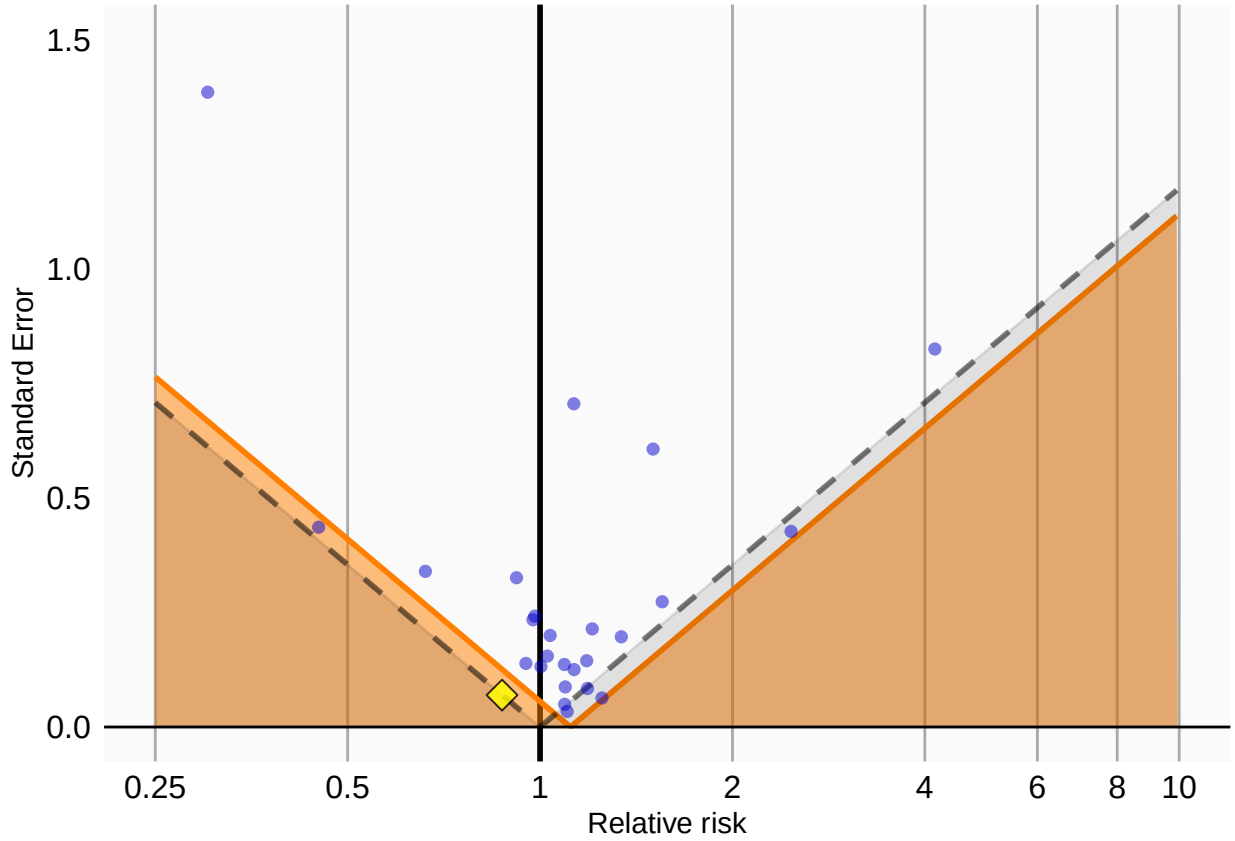
```
install.packages("EmpiricalCalibration")
library(EmpiricalCalibration)

# Analysis 1: No matching, simple outcome model
ncs <- resultsSum %>%
  filter(analysisId == 1,
         outcomeId != 77)
hoi <- resultsSum %>%
  filter(analysisId == 1,
         outcomeId == 77)
null <- fitNull(ncs$logRr, ncs$seLogRr)
plotCalibrationEffect(logRrNegatives = ncs$logRr,
                      seLogRrNegatives = ncs$seLogRr,
                      logRrPositives = hoi$logRr,
                      seLogRrPositives = hoi$seLogRr, null)
```



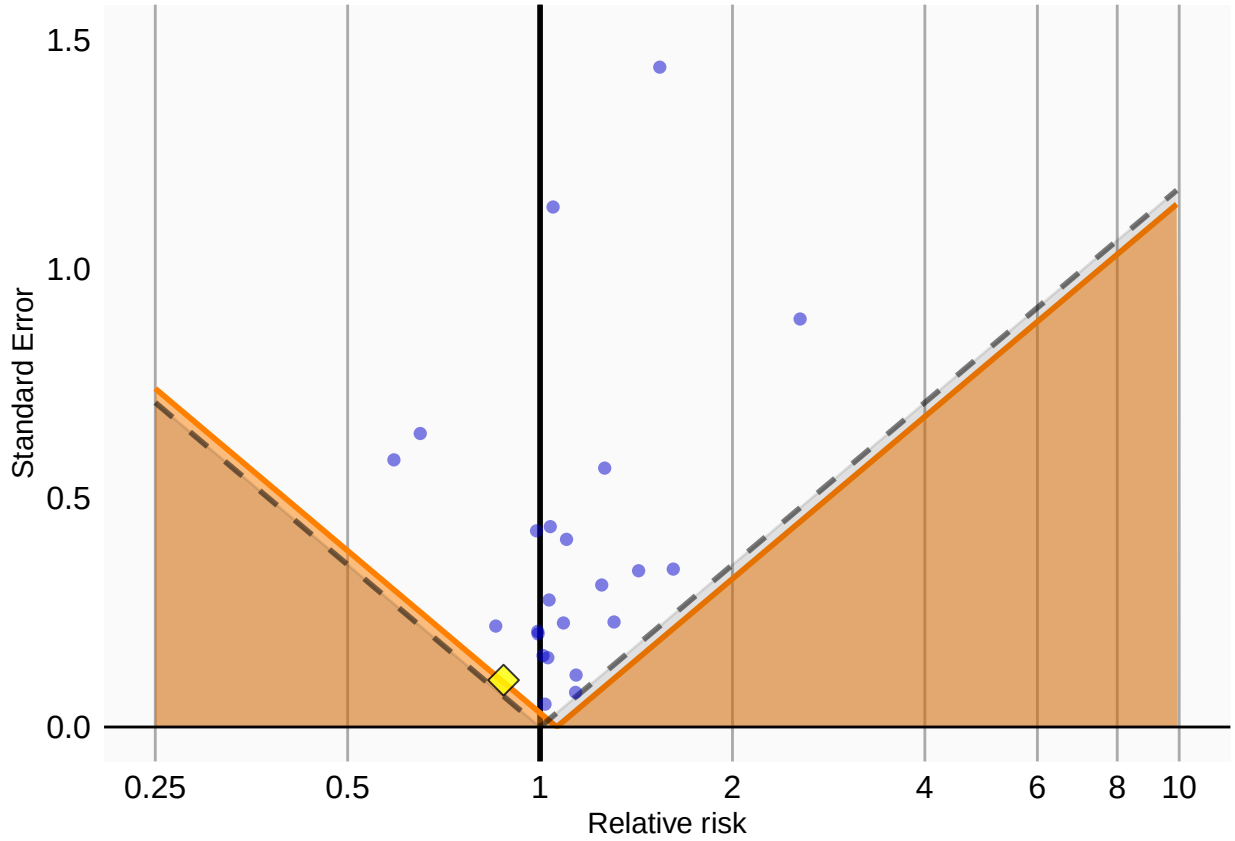

```
# Analysis 2: Matching
ncs <- resultsSum %>%
  filter(analysisId == 2,
         outcomeId != 77)
hoi <- resultsSum %>%
  filter(analysisId == 2,
         outcomeId == 77)
null <- fitNull(ncs$logRr, ncs$seLogRr)
plotCalibrationEffect(logRrNegatives = ncs$logRr,
                      seLogRrNegatives = ncs$seLogRr,
                      logRrPositives = hoi$logRr,
                      seLogRrPositives = hoi$seLogRr, null)
```



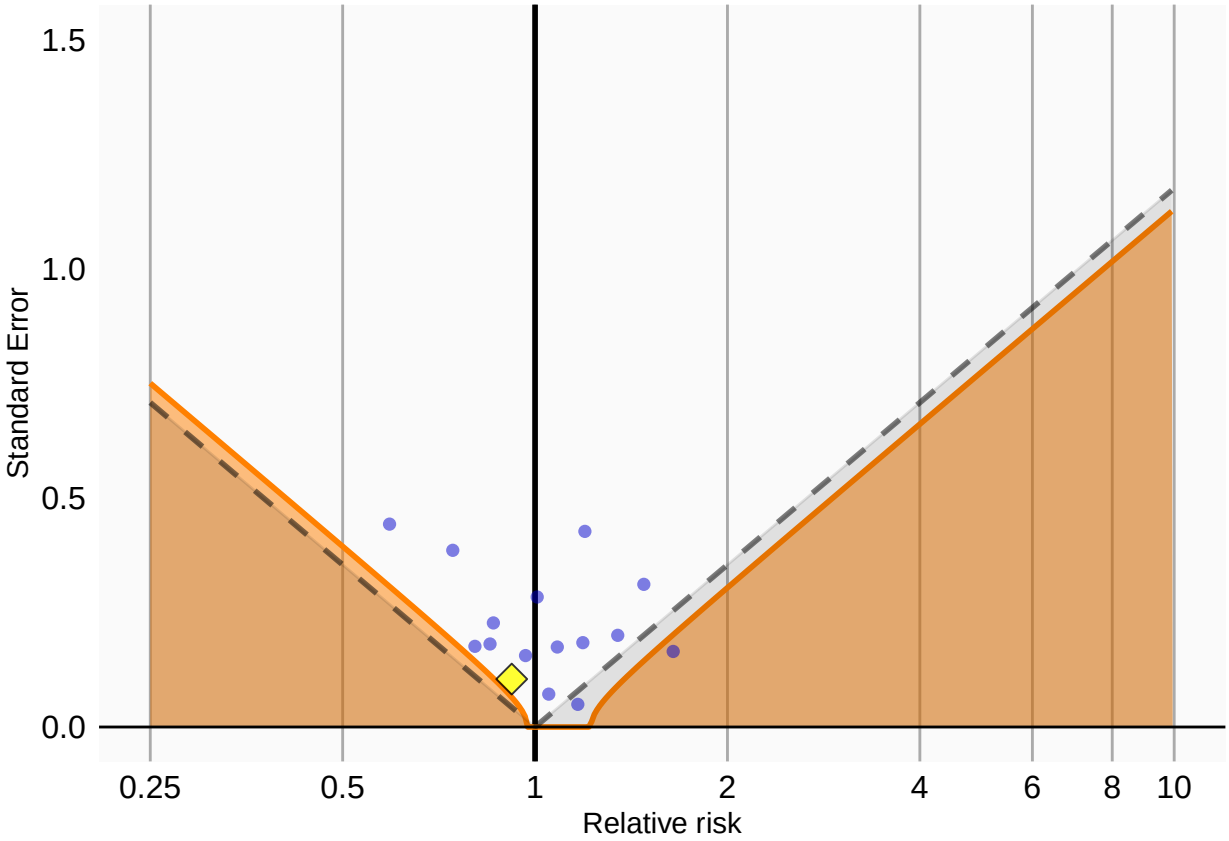

```
# Analysis 3: Stratification
ncs <- resultsSum %>%
  filter(analysisId == 3,
         outcomeId != 77)
hoi <- resultsSum %>%
  filter(analysisId == 3,
         outcomeId == 77)
null <- fitNull(ncs$logRr, ncs$seLogRr)
plotCalibrationEffect(logRrNegatives = ncs$logRr,
                      seLogRrNegatives = ncs$seLogRr,
                      logRrPositives = hoi$logRr,
                      seLogRrPositives = hoi$seLogRr, null)
```




```
# Analysis 4: Inverse probability of treatment weighting
ncs <- resultsSum %>%
  filter(analysisId == 4,
         outcomeId != 77)
hoi <- resultsSum %>%
  filter(analysisId == 4,
         outcomeId == 77)
null <- fitNull(ncs$logRr, ncs$seLogRr)
plotCalibrationEffect(logRrNegatives = ncs$logRr,
                      seLogRrNegatives = ncs$seLogRr,
                      logRrPositives = hoi$logRr,
                      seLogRrPositives = hoi$seLogRr, null)
```




```
# Analysis 5: Stratification plus full outcome model
ncs <- resultsSum %>%
  filter(analysisId == 5,
         outcomeId != 77)
hoi <- resultsSum %>%
  filter(analysisId == 5,
         outcomeId == 77)
null <- fitNull(ncs$logRr, ncs$seLogRr)
plotCalibrationEffect(logRrNegatives = ncs$logRr,
                      seLogRrNegatives = ncs$seLogRr,
                      logRrPositives = hoi$logRr,
                      seLogRrPositives = hoi$seLogRr, null)
```

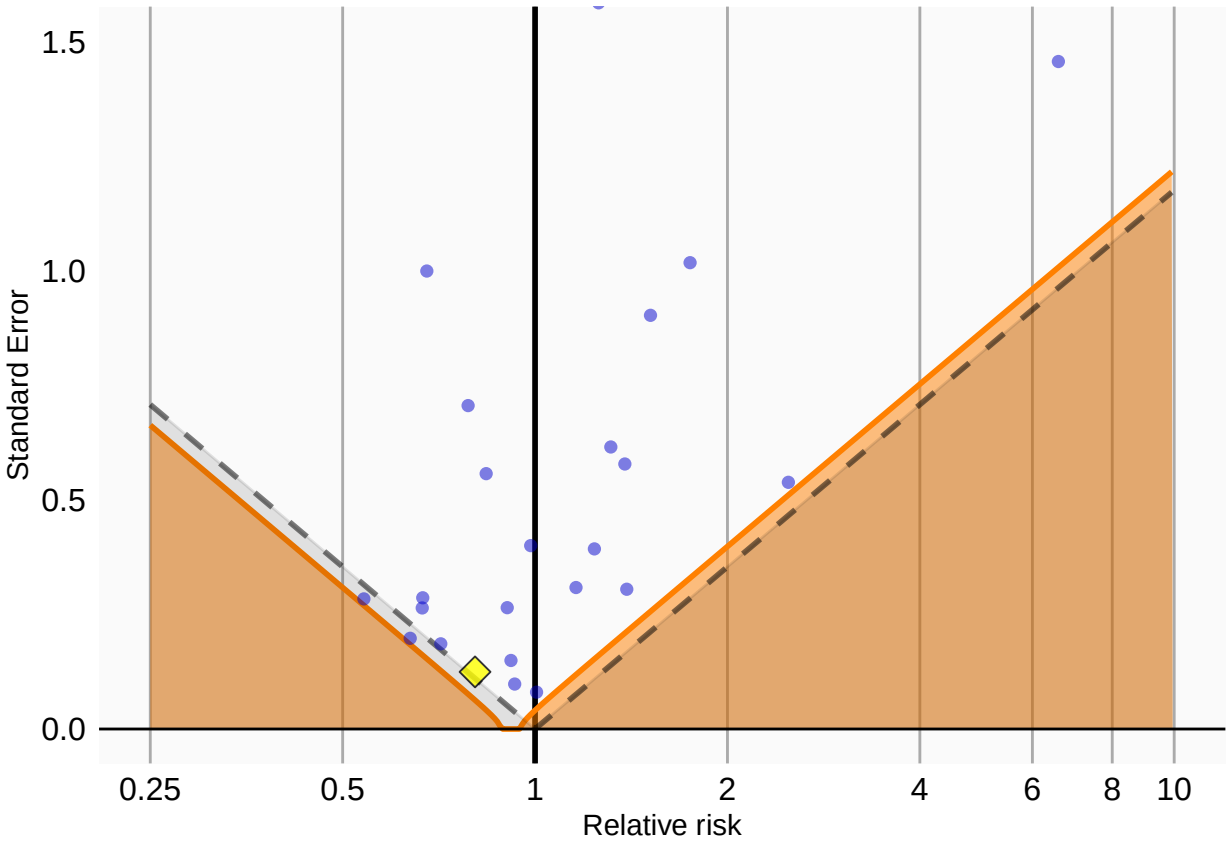


Analysis 6 explored interactions with certain variables. The estimates for these interaction terms are stored in a separate results summary. We can examine whether these estimates are also consistent with the null. In this example we consider the interaction with 'concurrent use of antithrombotic agents' (covariate ID 21600960413):

```
interactionResultsSum <- getInteractionResultsSummary(folder)

# Analysis 6: Stratification plus interaction terms
ncs <- interactionResultsSum %>%
  filter(analysisId == 6,
         interactionCovariateId == 21600960413,
         outcomeId != 77)
hoi <- interactionResultsSum %>%
  filter(analysisId == 6,
         interactionCovariateId == 21600960413,
         outcomeId == 77)
null <- fitNull(ncs$logRr, ncs$seLogRr)
plotCalibrationEffect(logRrNegatives = ncs$logRr,
                      seLogRrNegatives = ncs$seLogRr,
                      logRrPositives = hoi$logRr,
                      seLogRrPositives = hoi$seLogRr, null)
```

```
## Warning in checkWithinLimits(yLimits, c(seLogRrNegatives, seLogRrPositives), : Values are outside pl
```

```
## Warning: Removed 1 rows containing missing values (`geom_vline()`).
```



8 Exporting to CSV

The results generated so far all reside in binary object on your local file system, mixing aggregate statistics such as hazard ratios with patient-level data including propensity scores per person. How could we share our results with others, possibly outside our organization? This is where the `exportToCsv()` function comes in. This function exports all results, including diagnostics to CSV (comma-separated values) files. These files only contain aggregate statistics, not patient-level data. The format is CSV files to enable human review.

```
exportToCsv(
  folder,
  exportFolder = file.path(folder, "export"),
  databaseId = "My CDM",
  minCellCount = 5,
  maxCores = parallel::detectCores()
)
```

Any person counts in the results that are smaller than the `minCellCount` argument will be blinded, by replacing the count with the negative `minCellCount`. For example, if the number of people with the outcome is 3, and `minCellCount = 5`, the count will be reported to be -5, which in the Shiny app will be displayed as '<5'.

Information on the data model used to generate the CSV files can be retrieved using `getResultsDataModelSpecifications()`:

```
getResultsDataModelSpecifications()
```

```
## # A tibble: 188 x 8
##    tableName         columnName      dataType isRequired primaryKey minCellCount deprecated descripti
```

```
##    <chr>             <chr>           <chr>    <chr>      <chr>      <chr>        <chr>      <chr>
##  1 cm_attrition      sequence_number int      Yes        Yes        No           No         "The pla
##  2 cm_attrition      description     varchar  Yes        No         No           No         "A descr
##  3 cm_attrition      subjects        int      Yes        No         Yes          No         "The numb
##  4 cm_attrition      exposure_id     bigint   Yes        Yes        No           No         "The iden
##  5 cm_attrition      target_id       bigint   Yes        Yes        No           No         "The iden
##  6 cm_attrition      comparator_id   bigint   Yes        Yes        No           No         "The iden
##  7 cm_attrition      analysis_id     int      Yes        Yes        No           No         "The iden
##  8 cm_attrition      outcome_id      bigint   Yes        Yes        No           No         "Foreign
##  9 cm_attrition      database_id     varchar  Yes        Yes        No           No         "Foreign
## 10 cm_follow_up_dist target_id       bigint   Yes        Yes        No           No         "The iden
## # i 178 more rows
```

9 View results in a Shiny app

Finally, we can view the results in a Shiny app. For this we must first load the CSV files produced by `exportToCsv()` into a database. We could use the `uploadExportedResults()` function for this. However, if we just want to view the results ourselves we can create a small SQLite database ourselves without having to set up a database server. In any case we need to specify the names of the exposure and outcome cohorts we used in our study. We can create the SQLite database using:

```
cohorts <- data.frame(
  cohortId = c(
    1,
    2,
    77),
  cohortName = c(
    "Celecoxib",
    "Diclofenac",
    "GI Bleed"
  )
)

insertExportedResultsInSqlite(
  sqliteFileName = file.path(folder, "myResults.sqlite"),
  exportFolder = file.path(folder, "export"),
  cohorts = cohorts
)
```

Next we launch the Shiny app using:

```
launchResultsViewerUsingSqlite(
  sqliteFileName = file.path(folder, "myResults.sqlite")
)
```

10 Acknowledgments

Considerable work has been dedicated to provide the `CohortMethod` package.

```
citation("CohortMethod")
```

```
## To cite package 'CohortMethod' in publications use:
##
##   Schuemie M, Suchard M, Ryan P (2024). _CohortMethod: New-User Cohort Method with Large Scale Proper
##   https://github.com/OHDSI/CohortMethod.
```

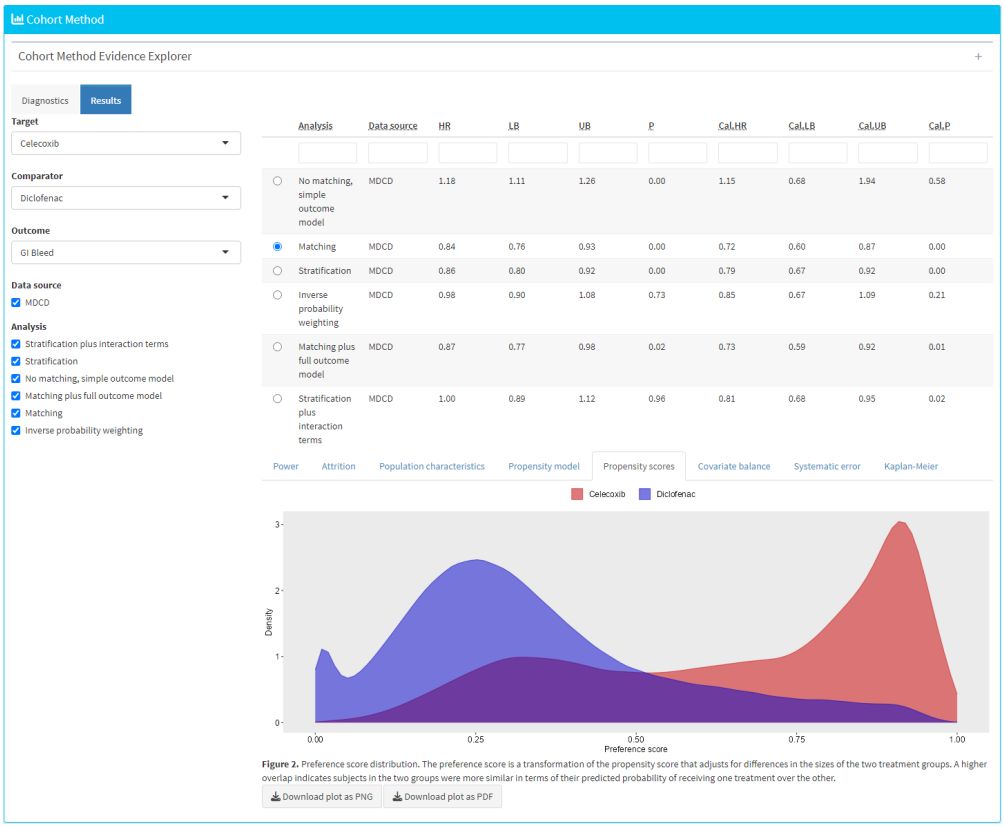


Figure 1: CohortMethod Shiny app

```
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {CohortMethod: New-User Cohort Method with Large Scale Propensity and Outcome
## Models},
##     author = {Martijn Schuemie and Marc Suchard and Patrick Ryan},
##     year = {2024},
##     note = {https://ohdsi.github.io/CohortMethod,
## https://github.com/OHDSI/CohortMethod},
##   }
```

Further, `CohortMethod` makes extensive use of the `Cyclops` package.

```
citation("Cyclops")
```

```
## To cite Cyclops in publications use:
##
##   Suchard MA, Simpson SE, Zorych I, Ryan P, Madigan D (2013). "Massive parallelization of serial inf
##   Modeling and Computer Simulation_, *23*, 10. <https://dl.acm.org/doi/10.1145/2414416.2414791>.
##
## A BibTeX entry for LaTeX users is
##
##   @Article{,
##     author = {M. A. Suchard and S. E. Simpson and I. Zorych and P. Ryan and D. Madigan},
##     title = {Massive parallelization of serial inference algorithms for complex generalized linear m
##     journal = {ACM Transactions on Modeling and Computer Simulation},
##     volume = {23},
##     pages = {10},
##     year = {2013},
##     url = {https://dl.acm.org/doi/10.1145/2414416.2414791},
##   }
```

This work is supported in part through the National Science Foundation grant IIS 1251151.