Using FeatureExtraction

Martijn J. Schuemie

2024-04-18

Contents

1 Introduction

The `FeatureExtraction` package can be used to create features for a cohort, using the information stored in the Common Data Model. A cohort is defined a set of persons who satisfy one or more inclusion criteria for a duration of time. Features can for example be diagnoses observed prior to entering the cohort. Some people might also refer to such features as 'baseline characteristics', or to features in general as 'covariates', and we will use those terms interchangeably throughout this vignette.

This vignette describes how features can be constructed using the default covariate definitions embedded in the package. Although these definitions allow quite some customization through predefined parameters, it is possible that someone needs more customization. In this case, the reader is referred to the other vignettes included in this package that deal with constructing completely custom covariates.

This vignette will first describe how to specify which features to construct. In many situations, for example when using `FeatureExtraction` as part of another package such as `CohortMethod` or `PatientLevelPrediction`, that is all one needs to know about the `FeatureExtraction` package, as the actual calling of the package is done by the other package. However, it is also possible to use this package on its own, for example to create a descriptive characterization of a cohort to include in a paper.

2 Covariate settings

Users can specify which covariates to construct in three ways:

1. Choose the default set of covariates.
2. Choose from a set of prespecified analyses.
3. Create a set of custom analyses.

An **analysis** is a process that creates one or more similar covariates. For example, one analysis might create a binary covariate for each condition observed in the condition_occurrence table in the year prior to cohort start, and another analysis might create a single covariate representing the Charlson comorbidity index.

Note that it is always possible to specify a set of concept IDs that can or can't be used to construct features. When choosing the default set (option 1) or the prespecified analysis (option 2) this can only be done across all analysis. When creating custom analyses (option 3) this can be specified per analysis.

For **advanced users**: It is also possible to specify a set of covariate IDs that need to be constructed. A covariate ID identifies a specific covariate, for example the Charlson comorbidity index, or the occurrence of a specific condition concept in a specific time window. A covariate ID is therefore not to be confused with a concept ID. The typical scenario where one might want to specify covariate IDs to construct is when someone already constructed covariates in one population, found a subset of covariates to be of interest, and would like to have only those covariates constructed in another population.

2.1 Using the default set of covariates

Using the default set of covariates is straightforward:

```
settings <- createDefaultCovariateSettings()
```

This will create a wide array of features, ranging from demographics, through conditions and drugs, to several risk scores.

Note that we could specify a set of concepts that should not be used to create covariates, for example:

```
settings <- createDefaultCovariateSettings(
  excludedCovariateConceptIds = 1124300,
  addDescendantsToExclude = TRUE
)
```

This will create the default set of covariates, except those derived from concept 1124300 (the ingredient diclofenac) and any of its descendants (ie. all drugs containing the ingredient diclofenac).

2.2 Using prespecified analyses

The function `createCovariateSettings` allow the user to choose from a large set of predefined covariates. Type `?createCovariateSettings` to get an overview of the available options. For example:

```
settings <- createCovariateSettings(
  useDemographicsGender = TRUE,
  useDemographicsAgeGroup = TRUE,
  useConditionOccurrenceAnyTimePrior = TRUE
)
```

This will create binary covariates for gender, age (in 5 year age groups), and each concept observed in the `condition_occurrence` table any time prior to (and including) the cohort start date.

Many of the prespecified analyses refer to a short, medium, or long term time window. By default, these windows are defined as:

- Long term: 365 days prior up to and including the cohort start date.
- Medium term: 180 days prior up to and including the cohort start date.
- Short term: 30 days prior up to and including the cohort start date.

However, the user can change these values. For example:

```
settings <- createCovariateSettings(
  useConditionEraLongTerm = TRUE,
  useConditionEraShortTerm = TRUE,
  useDrugEraLongTerm = TRUE,
  useDrugEraShortTerm = TRUE,
  longTermStartDays = -180,
  shortTermStartDays = -14,
  endDays = -1
)
```

This redefines the long term window as 180 days prior up to (but not including) the cohort start date, and redefines the short term window as 14 days prior up to (but not including) the cohort start date.

Again, we can also specify which concept IDs should or should not be used to construct covariates:

```
settings <- createCovariateSettings(
  useConditionEraLongTerm = TRUE,
  useConditionEraShortTerm = TRUE,
  useDrugEraLongTerm = TRUE,
  useDrugEraShortTerm = TRUE,
  longTermStartDays = -180,
  shortTermStartDays = -14,
  endDays = -1,
  excludedCovariateConceptIds = 1124300,
  addDescendantsToExclude = TRUE
)
```

2.3 Creating a set of custom covariates

This option should only be used by **advanced users**. It requires one to understand that at the implementation level, an analysis is a combination of a piece of highly parameterized SQL together with a specification of the parameter values. The best way to understand the available options is to take a prespecified analysis as starting point, and convert it to a detailed setting object:

```
settings <- createCovariateSettings(useConditionEraLongTerm = TRUE)
settings2 <- convertPrespecSettingsToDetailedSettings(settings)
settings2$analyses[[1]]
```

```
## $analysisId
## [1] 202
```

```
##
## $sqlFileName
## [1] "DomainConcept.sql"
##
## $parameters
## $parameters$analysisId
## [1] 202
##
## $parameters$analysisName
## [1] "ConditionEraLongTerm"
##
## $parameters$startDay
## [1] -365
##
## $parameters$endDay
## [1] 0
##
## $parameters$subType
## [1] "all"
##
## $parameters$domainId
## [1] "Condition"
##
## $parameters$domainTable
## [1] "condition_era"
##
## $parameters$domainConceptId
## [1] "condition_concept_id"
##
## $parameters$domainStartDate
## [1] "condition_era_start_date"
##
## $parameters$domainEndDate
## [1] "condition_era_end_date"
##
## $parameters$description
## [1] "One covariate per condition in the condition_era table overlapping with any part of the long te
##
##
## $includedCovariateConceptIds
## list()
##
## $includedCovariateIds
## list()
##
## $addDescendantsToInclude
## [1] FALSE
##
## $excludedCovariateConceptIds
## list()
##
## $addDescendantsToExclude
## [1] FALSE
```

One can create a detailed analysis settings object from scratch, and use it to create a detailed settings object:

```
analysisDetails <- createAnalysisDetails(
  analysisId = 1,
  sqlFileName = "DemographicsGender.sql",
  parameters = list(
    analysisId = 1,
    analysisName = "Gender",
    domainId = "Demographics"
  ),
  includedCovariateConceptIds = c(),
  addDescendantsToInclude = FALSE,
  excludedCovariateConceptIds = c(),
  addDescendantsToExclude = FALSE,
  includedCovariateIds = c()
)

settings <- createDetailedCovariateSettings(list(analysisDetails))
```

2.4 Temporal covariates

Ordinarily, covariates are created for just a few time windows of interest, for example the short, medium, and long term windows described earlier. However, sometimes a more fine-grained temporal resolution is required, for example creating covariates for each day separately, in the 365 days prior to cohort start. We will refer to this type of covariates as *temporal covariates*. Temporal covariates share the same covariate ID across the time windows, and use a separate time ID to distinguish between time windows. There currently aren't many applications that are able to handle temporal covariates. For example, the `CohortMethod` package will break when provided with temporal covariates. However, there are some machine learning algorithms in the `PatientLevelPrediction` package that require temporal covariates.

Again, we can just choose to use the default settings:

```
settings <- createDefaultTemporalCovariateSettings()
```

Or, we can choose from a set of prespecified temporal covariates:

```
settings <- createTemporalCovariateSettings(
  useConditionOccurrence = TRUE,
  useMeasurementValue = TRUE
)
```

In this case we've chosen to create binary covariates for each concept in the `condition_occurrence` table, and continuous covariates for each measurement - unit combination in the `measurement` table in the CDM. By default, temporal covariates are created for each day separately in the 365 days before (but not including) the cohort start date. Different time windows can also be specified, for example creating 7 day intervals instead:

```
settings <- createTemporalCovariateSettings(
  useConditionOccurrence = TRUE,
  useMeasurementValue = TRUE,
  temporalStartDays = seq(-364, -7, by = 7),
  temporalEndDays = seq(-358, -1, by = 7)
)
```

Each time window includes the specified start and end day.

Similar to ordinary covariates, **advanced users** can also define custom analyses:

```
analysisDetails <- createAnalysisDetails(
  analysisId = 1,
  sqlFileName = "MeasurementValue.sql",
  parameters = list(
    analysisId = 1,
    analysisName = "MeasurementValue",
    domainId = "Measurement"
  ),
  includedCovariateConceptIds = c(),
  addDescendantsToInclude = FALSE,
  excludedCovariateConceptIds = c(),
  addDescendantsToExclude = FALSE,
  includedCovariateIds = c()
)

settings <- createDetailedTemporalCovariateSettings(list(analysisDetails))
```

3 Constructing covariates for a cohort of interest

Here we will walk through an example, creating covariates for two cohorts of interest: new users of diclofenac and new users of celecoxib.

3.1 Configuring the connection to the server

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"
resultsDatabaseSchema <- "my_results"
```

The last two lines define the `cdmDatabaseSchema` and `resultSchema` variables. We'll use these later to tell R where the data in CDM format live, and where we want to write intermediate and result tables. Note that for Microsoft SQL Server, database schemas need to specify both the database and the schema, so for example `cdmDatabaseSchema <- "my_cdm_data.dbo"`.

3.2 Creating a cohort of interest

FeatureExtraction requires the cohorts to be instantiated in the `cohort` table in the Common Data Model, or in a table that has the same structure as the `cohort` table. We could create cohorts using a cohort definition

tool, but here we'll just use some simple SQL to find the first drug era per person. Note that because we will be creating covariates based on data before cohort start, we are requiring 365 days of observation before the first exposure. FeatureExtraction will not check if a subject is observed during the specified time windows.

```
/**********************************
File cohortsOfInterest.sql
**********************************/

IF OBJECT_ID('@resultsDatabaseSchema.cohorts_of_interest', 'U') IS NOT NULL
DROP TABLE @resultsDatabaseSchema.cohorts_of_interest;

SELECT first_use.*
INTO @resultsDatabaseSchema.cohorts_of_interest
FROM (
SELECT drug_concept_id AS cohort_definition_id,
MIN(drug_era_start_date) AS cohort_start_date,
MIN(drug_era_end_date) AS cohort_end_date,
person_id
FROM @cdmDatabaseSchema.drug_era
WHERE drug_concept_id = 1118084-- celecoxib
OR drug_concept_id = 1124300 --diclofenac
GROUP BY drug_concept_id,
person_id
) first_use
INNER JOIN @cdmDatabaseSchema.observation_period
ON first_use.person_id = observation_period.person_id
AND cohort_start_date >= observation_period_start_date
AND cohort_end_date <= observation_period_end_date
WHERE DATEDIFF(DAY, observation_period_start_date, cohort_start_date) >= 365;
```

This is parameterized SQL which can be used by the `SqlRender` package. We use parameterized SQL so we do not have to pre-specify the names of the CDM and result schemas. That way, if we want to run the SQL on a different schema, we only need to change the parameter values; we do not have to change the SQL code. By also making use of translation functionality in `SqlRender`, we can make sure the SQL code can be run in many different environments.

```
library(SqlRender)
sql <- readSql("cohortsOfInterest.sql")
sql <- render(sql,
  cdmDatabaseSchema = cdmDatabaseSchema,
  resultsDatabaseSchema = resultsDatabaseSchema
)
sql <- translate(sql, targetDialect = connectionDetails$dbms)

connection <- connect(connectionDetails)
executeSql(connection, sql)
```

In this code, we first read the SQL from the file into memory. In the next line, we replace the two parameter names with the actual values. We then translate the SQL into the dialect appropriate for the DBMS we already specified in the `connectionDetails`. Next, we connect to the server, and submit the rendered and translated SQL.

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

```
sql <- paste(
  "SELECT cohort_definition_id, COUNT(*) AS count",
  "FROM @resultsDatabaseSchema.cohorts_of_interest",
  "GROUP BY cohort_definition_id"
)
sql <- render(sql, resultsDatabaseSchema = resultsDatabaseSchema)
sql <- translate(sql, targetDialect = connectionDetails$dbms)

querySql(connection, sql)
```

```
##   cohort_concept_id  count
## 1           1124300 240761
## 2           1118084  47293
```

3.3 Creating per-person covariates for a cohort of interest

We can create per-person covariates for one of the cohorts of interest, for example using the default settings:

```
covariateSettings <- createDefaultCovariateSettings()

covariateData <- getDbCovariateData(connectionDetails = connectionDetails,
                                    cdmDatabaseSchema = cdmDatabaseSchema,
                                    cohortDatabaseSchema = resultsDatabaseSchema,
                                    cohortTable = "cohorts_of_interest",
                                    cohortIds = c(1118084),
                                    rowIdField = "subject_id",
                                    covariateSettings = covariateSettings)
summary(covariateData)
```

3.3.1 Per-person covariate output format

The main component of the `covariateData` object is `covariates`:

```
covariateData$covariates
```

The columns are defined as follows:

- `rowId` uniquely identifies a cohort entry. When calling `getDbCovariateData` we defined `rowIdField = "subject_id"`, so in this case the `rowId` is the same as the `subject_id` in the cohort table. In cases where a single subject can appear in the cohort more than once it is up to the user to create a field in the cohort table that uniquely identifies each cohort entry, and use that as `rowIdField`.
- `covariateId` identifies the covariate, and definitions of covariates can be found in the `cohortData$covariateRef` object.
- `covariateValue` field provides the value.

3.3.2 Saving the data to file

Creating covariates can take considerable computing time, and it is probably a good idea to save them for future sessions. Because `covariateData` objects use `Andromeda`, we cannot use R's regular save function. Instead, we'll have to use the `saveCovariateData()` function:

```
saveCovariateData(covariateData, "covariates")
```

We can use the `loadCovariateData()` function to load the data in a future session.

3.3.3 Removing infrequent covariates, normalizing, and removing redundancy

One reason for generating per-person covariates may be to use them in some form of machine learning. In that case it may be necessary to tidy the data before proceeding. The `tidyCovariateData` function can perform three tasks:

1. **Remove infrequent covariates**: Oftentimes the majority of features have non-zero values for only one or a few subjects in the cohort. These features are unlikely to end up in any fitted model, but can increase the computational burden, so removing them could increase performance. By default, covariates appearing in less than .1% of the subjects are removed.
2. **Normalization**: Scales all covariate values to a value between 0 and 1 (by dividing by the max value for each covariate).
3. **Removal of redundancy**: If every person in the cohort has the same value for a covariate (e.g. a cohort that is restricted to women will have the same gender covariate value for all) that covariate is redundant. Redundant covariates may pose a problem for some machine learning algorithms, for example causing a simple regression to no longer have a single solution. Similarly, groups of covariates may be redundant (e.g. every person will belong to at least one age group, making one group redundant as it can be defined as the absence of the other groups).

```
tidyCovariates <- tidyCovariateData(covariateData,
  minFraction = 0.001,
  normalize = TRUE,
  removeRedundancy = TRUE
)
```

If we want to know how many infrequent covariates were removed we can query the `metaData` object:

```
deletedCovariateIds <- attr(tidyCovariates, "metaData")$deletedInfrequentCovariateIds
head(deletedCovariateIds)
```

Similarly, if we want to know which redundant covariates were removed we can also query the `metaData` object:

```
deletedCovariateIds <- attr(tidyCovariates, "metaData")$deletedRedundantCovariateIds
head(deletedCovariateIds)
```

If we want to know what these numbers mean, we can use the `covariateRef` object that is part of any `covariateData` object. Remember that the covariate data is stored in an `Andromeda` object, so we should use the proper syntax for querying these objects (see the `Andromeda` package documentation):

```
library(Andromeda)
covariateData$covariateRef %>%
  filter(covariateId %in% deletedCovariateIds, ) %>%
  collect()
```

3.4 Creating aggregated covariates for a cohort of interest

Often we do not need to have per-person covariates, but instead we are interested in aggregated statistics instead. For example, we may not need to know which persons are male, but would like to know what proportion of the cohort is male. We can aggregate per-person covariates:

```
covariateData2 <- aggregateCovariates(covariateData)
```

Of course, if all we wanted was aggregated statistics it would have been more efficient to aggregate them during creation:

```
covariateSettings <- createDefaultCovariateSettings()

covariateData2 <- getDbCovariateData(connectionDetails = connectionDetails,
                                     cdmDatabaseSchema = cdmDatabaseSchema,
                                     cohortDatabaseSchema = resultsDatabaseSchema,
                                     cohortTable = "cohorts_of_interest",
                                     cohortIds = c(1118084),
                                     covariateSettings = covariateSettings,
                                     aggregated = TRUE)
summary(covariateData2)
```

Note that we specified `aggregated = TRUE`. Also, we are no longer required to define a `rowIdField` because we will no longer receive per-person data.

3.4.1 Aggregated covariate output format

The two main components of the aggregated `covariateData` object are `covariates` and `covariatesContinuous`, for binary and continuous covariates respectively:

```
covariateData2$covariates
```

```
covariateData2$covariatesContinuous
```

The columns of `covariates` are defined as follows:

- `covariateId` identifies the covariate, and definitions of covariates can be found in the `covariateData$covariateRef` object.
- `sumValue` is the sum of the covariate values. Because these are binary features, this is equivalent to the number of people that have the covariate with a value of 1.
- `averageValue` is the average covariate value. Because these are binary features, this is equivalent to the proportion of people that have the covariate with a value of 1.

The columns of `covariatesContinuous` are defined as follows:

- `covariateId` identifies the covariate, and definitions of covariates can be found in the `cohortData$covariateRef` object.
- `countValue` is the number of people that have a value (for continuous variables).
- `minValue`, `maxValue`, `averageValue`, `standardDeviation`, `medianValue`, `p10Value`, `p25Value`, `p75Value`, and `p90Value` all inform on the distribution of covariate values. Note that for some covariates (such as the Charlson comorbidity index) a value of 0 is interpreted as the value 0, while for other covariates (Such as blood pressure) 0 is interpreted as missing, and the distribution statistics are only computed over non-missing values. To learn which continuous covariates fall into which category one can consult the `missingMeansZero` field in the `covariateData$analysisRef` object.

3.5 Creating a table 1

One task supported by the FeatureExtraction package is creating a table of overall study population characteristics that can be include in a paper. Since this is typically the first table in a paper we refer to such a table as 'table 1'. A default table 1 is available in the FeatureExtraction package:

```
result <- createTable1(
  covariateData1 = covariateData2
)
print(result, row.names = FALSE, right = FALSE)
```

Where applicable, these characteristics are drawn from analyses pertaining the 'long-term' windows, so concepts observed in the 365 days before up to and included the cohort start date.

The createTable1 function requires a simple specification of what variables to include in the table. The default specifications included in the package can be reviewed by calling the getDefaultTable1Specifications function. The specification reference analysis IDs and covariate IDs, and in the default specification these IDs refer to those in the default covariate settings. It is possible to create custom table 1 specifications and use those instead.

Here we based table 1 on a covariateData object containing all default covariates, even though only a small fraction of covariates are used in the table. If we only want to extract those covariates needed for the table, we can use the createTable1CovariateSettings function:

```
covariateSettings <- createTable1CovariateSettings()

covariateData2b <- getDbCovariateData(connectionDetails = connectionDetails,
                                      cdmDatabaseSchema = cdmDatabaseSchema,
                                      cohortDatabaseSchema = resultsDatabaseSchema,
                                      cohortTable = "cohorts_of_interest",
                                      cohortIds = c(1118084),
                                      covariateSettings = covariateSettings,
                                      aggregated = TRUE)
summary(covariateData2b)
```

4 Comparing two cohorts

Another task supported by the FeatureExtraction package is comparing two cohorts of interest. Suppose we want to compare two cohorts only on the variables included in the default table 1:

```
settings <- createTable1CovariateSettings(
  excludedCovariateConceptIds = c(1118084, 1124300),
  addDescendantsToExclude = TRUE
)

covCelecoxib <- getDbCovariateData(connectionDetails = connectionDetails,
                                   cdmDatabaseSchema = cdmDatabaseSchema,
                                   cohortDatabaseSchema = resultsDatabaseSchema,
                                   cohortTable = "cohorts_of_interest",
                                   cohortIds = c(1118084),
                                   covariateSettings = settings,
                                   aggregated = TRUE)
```

```
covDiclofenac <- getDbCovariateData(connectionDetails = connectionDetails,
                                    cdmDatabaseSchema = cdmDatabaseSchema,
                                    cohortDatabaseSchema = resultsDatabaseSchema,
                                    cohortTable = "cohorts_of_interest",
                                    cohortIds = c(1124300),
                                    covariateSettings = settings,
                                    aggregated = TRUE)
std <- computeStandardizedDifference(covCelecoxib, covDiclofenac)
```

In this example we have chosen to exclude any covariates derived from the two concepts that were used to define the two cohorts: celecoxib (1118084), and diclofenac (1124300). We compute the standardized difference between the remaining covariates.

```
head(std)
```

The `stdDiff` column contains the standardized difference. By default the data is ranked in descending order of the absolute value of the standardized difference, showing the covariate with the largest difference first.

We can also show the comparison as a standard table 1:

```
result <- createTable1(
  covariateData1 = covCelecoxib,
  covariateData2 = covDiclofenac,
  output = "two columns"
)
print(result, row.names = FALSE, right = FALSE)
```