ITR 0.3.1 User Guide

Haoda Fu[1]
Jie Xue[2]
Bo Zhang[3]

[1]fu_haoda@lilly.com
[2]xue_jie@lilly.com
[3]zhang_bo3@lilly.com

Contents

Introduction v

Implementation vii

Installation ix

Usage xi

Introduction

Precision medicine is a medical model that proposes the customization of healthcare, with medicial decisions, practices, and/or products being tailored to the individual patient. In this model, diagnostic testing is often employed for selecting appropriate and optimal therapies based on the context of a patient's genetic content or other molecular or cellular analysis. Tools employed in precision medicine can include molecular diagnostics, imaging, and analytics/software.

With new treatments and novel technology available, personalized medicine has become an important piece in the new era of medical product development. Traditional statistics methods for personalized medicine and subgroup identification primarily focus on single treatment or two arm randomized control trials. Motivated by the recent development of outcome weighted learning framework, we propose an alternative algorithm to search treatment assignments which has a connection with subgroup identification problems. Our method focuses on applications from clinical trials to generate easy to interpret results. This framework is able to handle two or more than two treatments from both randomized control trials and observational studies.

Assume that there are N subjects from a large population. For sample i, A_i is the treatment assignment, Y_i is the response where larger value is better, and X_i is vector of covariates. Let (Y, A, X) be the generic random variable $\{(Y_i, A_i, X_i)\}$, $\mathcal{P}$ be the corresponding distribution, and E be the expectation with respect to $\mathcal{P}$. Let $\mathcal{D}(\cdot)$ be a treatment recommendation, i.e,

$$\mathcal{D}(\cdot) : \chi \to A, \quad X_i \in \chi,$$

and $\mathcal{P}^{\mathcal{D}}$ be the distribution of (Y, A, X) given $A = \mathcal{D}(X)$. Define

$$E^{\mathcal{D}}(Y) = \int Y \, d\mathcal{P}^{\mathcal{D}} == \int Y \frac{d\mathcal{P}^{\mathcal{D}}}{d\mathcal{P}} = E\left[\frac{I_{A=\mathcal{D}(X)}}{p(A|X)} Y\right],$$

where

$$\frac{d\mathcal{P}^{\mathcal{D}}}{d\mathcal{P}} = \frac{p(y|x,a) I_{a=\mathcal{D}(x)} p(x)}{p(y|x,a) p(a|x) p(x)} = \frac{I_{a=\mathcal{D}(x)}}{p(a|x)}$$

The quantity $E^{\mathcal{D}}(Y)$ can be used to compare different treatment recommendations. For instance, assume that we have two doctors, Adam and Barry, each having a treatment rule. Doctor Adam gives patients treatment 1 if $X \geq 2$ and treatment 0 otherwise, denoted by $\mathcal{D}_A$, and doctor Barry give patients treatment 1 if $X \geq 3$ and treatment 0 otherwise, denoted by $\mathcal{D}_B$. Over the following population, we have

$$E^{\mathcal{D}_A} = \frac{1}{10} \times \frac{1}{0.5} (1 \times 1 + 0 \times 2 + 0 \times 3 + 0 \times 4 + 0 \times 5 + 0 \times 3 + 1 \times 3 + 1 \times 3 + 1 \times 3 + 1 \times 3) = 2.6$$

and

$$E^{\mathcal{D}_B} = \frac{1}{10} \times \frac{1}{0.5} (1 \times 1 + 1 \times 2 + 0 \times 3 + 0 \times 4 + 0 \times 5 + 0 \times 3 + 0 \times 3 + 1 \times 3 + 1 \times 3 + 1 \times 3) = 2.4$$

Table 1.1: Comparison of treatment recommendations between Doctors Adam and Barry

ID	Y	A	X	$P(A\|X)$	$\mathcal{D}_A$	$\mathcal{D}_B$	$\mathcal{D}_A = A$	$\mathcal{D}_B = A$
1	1	0	1	0.5	0	0	1	1
2	2	0	2	0.5	1	0	0	1
3	3	0	3	0.5	1	1	0	0
4	4	0	4	0.5	1	1	0	0
5	5	0	5	0.5	1	1	0	0
6	3	1	1	0.5	0	0	0	0
7	3	1	2	0.5	1	0	1	0
8	3	1	3	0.5	1	1	1	1
9	3	1	4	0.5	1	1	1	1
10	3	1	5	0.5	1	1	1	1

Therefore, we conclude doctor Adam's treatment recommendation is better than doctor Barry's treatment recommendation. When there are more than two treatment rules, the objective is to find $\mathcal{D}_o$ such that

$$\mathcal{D}_o \in \arg\max_{\mathcal{D}\in R} E^{\mathcal{D}}(Y) = \arg\max_{\mathcal{D}\in R} E\left[\frac{I_{A=\mathcal{D}(X)}}{p(A|X)}Y\right]$$

where R is the space of possible treatment recommendations.

Implementation

In the toy example in Chapter 1, doctors Adam and Barry use different threshold values on covariate X when making their treatment recommendations. Such threshold value is referred to a *cut* in the implementation. For each covariate X and a particular cut value c, there are two treatment recommendations,

Treatment 1 Give treatment 1 if $X < c$ and treatment 0 otherwise.

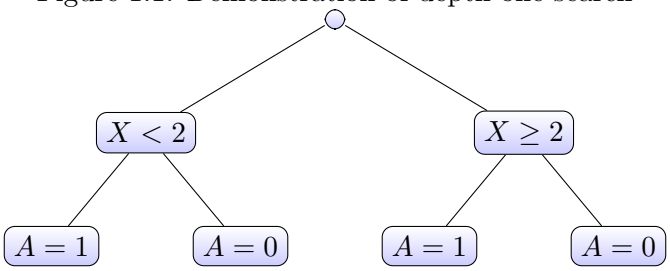
Treatment 2 Give treatment 1 if $X \geq c$ and treatment 0 otherwise.

It is easy to show that $I_{X<c} = I_{A=(X<c)}$ and $I_{X\geq c} = I_{A=(X\geq c)}$ In the implementation, the bitmask $I_{X<c}$ for each cut is saved. The search can then be implemented as follows

```
// Depth one
double v[2] = {0.0};
for (size_t i = 0; i < N; ++i) {
  v[0] += y[i] * (a[i] == m[i]);       // X < c
  v[1] += y[i] * (a[i] == 1 - m[i]);   // X >= c
}

// Depth two
double v[4] = {0.0};
for (size_t i = 0; i < N; ++i) {
  // X1 < c1 and X2 < c2
  v[0] += y[i] * (a[i] == m1[i]) * (a[i] == m2[i]);

  // X1 < c1 and X2 >= c2
  v[1] += y[i] * (a[i] == m1[i]) * (a[i] == 1 - m2[i]);

  // X1 >= c1 and X2 < c2
  v[2] += y[i] * (a[i] == 1 - m1[i]) * (a[i] == m2[i]);

  // X1 >= c1 and X2 >= c2
  v[3] += y[i] * (a[i] == 1 - m1[i]) * (a[i] == 1 - m2[i]);
}
```

As the depth of search increases, the above implementation becomes more cumbersome. However, if the package is to be deployed on GPUs, the above implementation can be revised to utilize vectorization. Currently, the package is intended to run on multicore computers, and the computation is organized differently which we explain next.

Figure 1.1: Demonstration of depth one search



Consider the depth one case shown in Figure 1.1. Recall that the bitmask of $X < 2$ is stored. For each sample i being processed, there are four combinations depending on the value of A and the bitmask $X(i) < c$. This means, the corresponding $Y(i)$ can be sorted into four buckets. For instance, the far left bucket corresponds to $X(i) < c$ and $A = 1$ and can therefore be indexed as bucket 3 ($(11)_2$). The far right bucket can be indexed as bucket 0 ($(00)_2$). Let T_0 be

$$T_0 = \sum_{i, A(i)=0} Y(i)$$

and B_i, $i = 0, 1, 2, 3$ denote the value accumulated in each bucket. Notice that

$$B_3 - B_2 + T_0 = \{A = 1 \cap X < 2\} - \{A = 0 \cap X < 2\} + \{A = 0\} = \{A = 1 \cap X < 2\} + \{A = 0 \cap X \geq 2\}$$

If $X(i) < 2$, the recommendation is treatment 1, matching $A(i)$. If $X(i) \geq 2$, the recommendation is treatment 0, matching $A(i)$. In other words, the quantity $B_3 - B_2 + T_0$ gives the result for treatment $X < c$. Similarly, one can verify that $B_1 - B_0 + T_0$ gives the result for treatment $X \geq c$.

Installation

The package is implemented in C++ and can be built with `cmake` version 3.12 or later. The package has been tested with `gcc` version 8.1.0 and `clang` version 9.1.0.

Assume that the source code has been unpacked into directory `/path/to/ITR`, the following steps will build the library in `/path/to/ITR-build` and install the library in `/path/to/ITR-install`.

```
> cd /path/to/ITR-build
> cmake /path/to/ITR -DCMAKE_INSTALL_PREFIX=/path/to/ITR-install
> make
```

The user is welcome to specify a particular compiler by prefixing `CC=/path/to/c_compiler CXX=/path/to/cpp_compiler` to the `cmake` configuration line.

The package uses `googletest` for unit testing, which is downloaded automatically during the configuration phase. The user is recommended to verify the build before the final installation, and this can be done as follows

```
> cd /path/to/ITR-build/test
> make test
> ./test
```

Once the build is verified, one can complete the installation

```
> cd /path/to/ITR-build
> make install
```

This will install header file `ITR.h` to `/path/to/ITR-install/include/itr` and `libitr.a` to `path/to/ITR-install/lib`.

Usage

The package comes with a demonstration program in the `demo` directory showing how to use the `ITR` library. The example can be built as follows

```
> cd /path/to/ITR-build/demo
> make demo
> ./demo
```

Generally, one needs to `-I/path/to/ITR-install/include/itr` and `-L/path/to/ITR-install/lib -litr` to the compile command.

To use the library, one needs to include the header file `ITR.h`. Inside `main()`, an `ITR::ITR` instance is created by passing two arguments to its constructor. The first argument is a `std::string` object specifying the path to the input csv file. The second argument is an unsigned integer specifying the search depth.

For the csv input file, the first line must be a header. The first column is the subject identifier, followed by continuous variables (labeled as `cont*`), ordinal variables (labeled as `ord*`), nominal variables (labeled as `nom*`), actions (labeled as `A*`), responses (labeled as `Y*`), and condition probablity $P(A = 1|X)$. The ITR library is case insensitive to these labels.

For the search depth, the valid values are 1, 2, and 3.

The constructor of 'ITR::ITR' will throw an exception if the input file does not exist or the search depth is invalid.

After the instance is created, one performs the search by invoking the `run` method. Here, one has the choice of running the search sequentially or in parallel by passing an `unsigned` integer `nThreads` to the method. If `nThreads` is 1, the search is done sequentially. If `nThreads` is greater than 1, the search is done in parallel.

Once the search completes, one can query the top `n` results by calling the `report` method. An sample output of the `demo` program looks like the following

```
> ./demo --thread=8
Loading input data ...
Creating search engine with depth 3
Searching 86131 choices ...
Completed in 1.102359e-02 seconds using 8 threads
Best 5 results: ...
Value = 6.884978e+01
  X1 < 5.094425e+01
  X2 >= 5.253060e+01
  X7 not in {0, 2}
Value = 6.787278e+01
  X1 < 5.942735e+01
```

```
   X6 not in {2, 3}
   X7 not in {2, 3}
 Value = 6.744488e+01
   X1 < 5.094425e+01
   X7 not in {0, 2}
   X8 not in {2}
 Value = 6.742991e+01
   X1 < 4.372625e+01
   X5 < 3
   X6 not in {2, 3}
 Value = 6.740870e+01
   X1 < 5.094425e+01
   X6 not in {0, 2}
   X8 not in {2}
```