sMTL Vignette

true true

2022-07-01

Contents

Introduction

`sMTL` is a fast toolkit for L0-constrained multi-task learning, multi-label learning and domain generalization. L0 constraints select the best subset of features in a variety of multi-dataset settigs. The toolkit can (approximately) solve the following two problems

$$\min_{\boldsymbol{z},\ \mathbb{B},\bar{\boldsymbol{\beta}}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}_k \boldsymbol{\beta}_k||_2^2 + \lambda \sum_{k=1}^{K} ||\boldsymbol{\beta}_k - \bar{\boldsymbol{\beta}}||_2^2 + \alpha ||\mathbb{B}||_2^2 \qquad (\textit{Common Support})$$

$$\text{subject to: } z_j \in \{0,1\} \ \ \forall\ j \in [p]$$

$$\beta_{k,j}(1 - z_j) = 0 \ \forall\ j \in [p], k \in [K]$$

$$\sum_{j=1}^{p} z_j \leq s$$

$$\min_{\boldsymbol{z},\ \mathbb{B},\bar{\boldsymbol{\beta}}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}_k \boldsymbol{\beta}_k||_2^2 + \lambda \sum_{k=1}^{K} ||\boldsymbol{\beta}_k - \bar{\boldsymbol{\beta}}||_2^2 + \alpha ||\mathbb{B}||_2^2 + \delta \sum_{k=1}^{K} ||\boldsymbol{z}_k - \bar{\boldsymbol{z}}||_2^2 \qquad (Heterogeneous\ Support)$$

$$\text{subject to: } z_{k,j} \in \{0,1\} \ \ \forall\ j \in [p]$$
$$\beta_{k,j}(1 - z_{k,j}) = 0\ \forall\ j \in [p], k \in [K]$$
$$\sum_{j=1}^{p} z_{k,j} \leq s \ \ \forall\ k \in [K]$$

where $\boldsymbol{\beta}_k$ is the vector of coefficients associated with dataset k, and $\boldsymbol{z}_k = \mathbb{1}_{(\boldsymbol{\beta}_k \neq \mathbf{1})}$, the "support'' of $\boldsymbol{\beta}_k$, i.e., a $p \times 1$ indicator vector of whether the entries of $\boldsymbol{\beta}_k$ are non-zero.

The Common Support problem ensures that each of the K task-specific (or dataset-specific) $\boldsymbol{\beta}_k$ has the same support. The Heterogeneous Support problem allows each task to have a different support put penalizes the $\boldsymbol{\beta}_k$ for having differing supports using the $\delta \sum_{k=1}^{K} ||\boldsymbol{z}_k - \bar{\boldsymbol{z}}||_2^2$ penalty.

The parameter λ shrinks the $\boldsymbol{\beta}_k$ towards a common $\bar{\boldsymbol{\beta}} = \frac{1}{K} \sum_{k=1}^{K} \boldsymbol{\beta}_k$. The parameter α controls the strength of ℓ_2 (Ridge) shrinkage where $\alpha ||\mathbb{B}||_2^2 = \alpha \sum_{k=1}^{K} ||\boldsymbol{\beta}_k||_2^2$. Adding either of these shrinkage terms can be effective in reducing the risk of overfitting and often results in better predictive models. We recommend using either but not both of these regularization terms. For Hetergeneous Support problems δ controls the degree to which the $\boldsymbol{z}$, the supports of the $\boldsymbol{\beta}_k$, are shrunk towards $\bar{\boldsymbol{z}} = \sum_{k=1}^{K} \boldsymbol{z}_k$. When δ is high, this results in solutions equivalent to the Common Support approach. The fitting is done over a grid of λ, δ and α values to generate a regularization path.

`sMTL` also has dedicated code to deal with the "Multi-Label Learning" (a special case of, and sometimes referred to as, "Multi-Task Learning") in which the design matrix is fixed across tasks (i.e., $\mathbb{X}_k = \mathbb{X}\ \forall\ k\ \in \{1, 2, ..., K\}$), but the outcome is multivaraite. While this can be seen as a special case of the above, we include dedicated code that yields gains in efficiency in terms of memory and solve time.

The algorithms provided in `sMTL` are based on block coordinate descent and local combinatorial search. Numerous computational tricks and heuristics are used to speed up the algorithms and improve the solution quality. These heuristics include warm starts and active set convergence. For more details on the algorithms used, please refer to our paper Fast Best Subset Selection: Coordinate Descent and Local Combinatorial Optimization Algorithms.

The toolkit is implemented in Julia along with an easy-to-use R interface. In this vignette, we provide a tutorial on using the R interface. Particularly, we will demonstrate how use `sMTL`'s main functions for fitting models and cross-validation.

Installation

`sMTL` can be installed directly from CRAN by executing:

```
install.packages("sMTL")
```

The very first time we use `sMTL` we have to keep in mind the following:

Package Setup

1. The programming language `Julia` is installed. We have a function to automatically install it if it is not already:

```
library(sMTL)
smtl_setup(installJulia = TRUE, installPackages = TRUE)
```

2. The `Julia` packages `TSVD` and `Statistics` must be installed. We also have a function to automatically install these if they are not already installed. If `Julia` is installed but the packages are not, the function can install them without re-installing `Julia`. If you ran the code for 1. above, you can ignore the rest of the set-up (1-3) and skip to fitting functions.

```
library(sMTL)
smtl_setup(installJulia = FALSE, installPackages = TRUE)
```

3. The `Julia` binary path must be set. If `Julia` is installed with our function (item 1), this should be set automatically. If `Julia` is already installed, simply open `Julia` and type "println(Sys.BINDIR)" and this will show the binary path.

The following code can be used where the string (instead of <>) after the code `path=` should be replaced with your computer's path for Julia's binary.

```
library(sMTL)
smtl_setup(path = "/Applications/Julia-1.5.app/Contents/Resources/julia/bin")
```

The `Julia` binary path is saved the first time the package is used after downloading, so one does not have to enter the path every time. Every time you load the package henceforth, one only need run the following.

```
library(sMTL)
library(LOLearn)
smtl_setup()
```

Failure to run this function above will leave R unable to locate `Julia` and the package functions will not work.

Tutorial

To illustrate the main `sMTL` functions, we start by generating synthetic datasets and proceed by fitting models. The package has three main machine learning problems it aims to solve:

1. Multi-Task Learning in which we have a collection of K separate training datasets: $\{\mathbb{X}_1, \mathbb{X}_2, ..., \mathbb{X}_K\}$, $\{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_K\}$ where $\mathbb{X}_k \in \mathbb{R}^{n_k \times p}$, $\mathbf{y}_k \in \mathbb{R}^{n_k}$. The goal is to make predictions on new observations from task k using model k.
2. Multi-Label Learning in which we have a collection of training data with a single common design matrix, $\mathbb{X}$, but a multivariate outcome for each observation, $\{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_K\}$ and $\mathbb{X} \in \mathbb{R}^{n \times p}, \mathbf{y}_k \in \mathbb{R}^n$. The goal is to make predictions on new observations from task k (i.e., to predict an observation of out come k) using model k. This can be viewed as a special case of the above multi-task framework (where $\mathbb{X}_k = \mathbb{X}\ \forall k \in \{1, 2, ..., K\}$ but we include specialized algorithms for this case.
3. Domain Generalization in which we have a collection of K separate training datasets: $\{\mathbb{X}_1, \mathbb{X}_2, ..., \mathbb{X}_K\}$ and $\{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_K\}$ where $\mathbb{X}_k \in \mathbb{R}^{n_k \times p}$, $\mathbf{y}_k \in \mathbb{R}^{n_k}$. The goal is to make predictions on new observations from an unseen "domain'' or dataset, $K+1$ using an ensemble of K models each trained on a single dataset combined with stacking ensemble weights.

Multi-Task Learning (Case 1)

We begin with case 1. We generate $K = 4$ synthetic datasets from a sparse linear model with the following:

- $\mathbb{X}_k$ is a 50×100 design matrix with iid standard normal entries

- $\boldsymbol{\beta}_k$ is a 50×1 vector with 10 entries set to 1 (plus some task-specific gaussian noise) and the rest are zeros. The indices of the non-zero entries differ slightly between tasks.
- $\boldsymbol{\epsilon}_k$ is a 100×1 vector with iid standard normal entries
- $\mathbf{y}_k$ is a 100×1 response vector such that $\mathbf{y}_k = \mathbb{X}_k \boldsymbol{\beta}_k + \boldsymbol{\epsilon}_k$

This dataset can be generated in R as follows:

```
set.seed(1) # fix the seed to get a reproducible result
K <- 4 # number of datasets
p <- 100 # covariate dimension
s <- 5 # support size
q <- 7 # size of subset of covariates that can be non-zero for any task
n_k <- 50 # task sample size
N <- n_k * p # full dataset samplesize
X <- matrix( rnorm(N * p), nrow = N, ncol=p) # full design matrix
B <- matrix(1 + rnorm(N * (p+1) ), nrow = p + 1, ncol = K) # betas before making sparse
Z <- matrix(0, nrow = p, ncol = K) # matrix of supports
y <- vector(length = N) # outcome vector

# randomly sample support to make betas sparse
for(j in 1:K)     Z[1:q, j] <- sample( c( rep(1,s), rep(0, q - s) ), q, replace = FALSE )
B[-1,] <- B[-1,] * Z # make betas sparse and ensure all models have an intercept

task <- rep(1:K, each = n_k) # vector of task labels (indices)

# iterate through and make each task specific dataset
for(j in 1:K){
    indx <- which(task == j) # indices of task
    e <- rnorm(n_k)
    y[indx] <- B[1, j] + X[indx,] %*% B[-1,j] + e
}
```

Note that the `task` vector is a $N \times 1$ vector where element i indicates the task index ($\{1, 2, ..., K\}$) of observation i.

First let's see the structure of the $\mathbb{B} \in \mathbb{R}^{p+1 \times K}$ where column k is equal to $\boldsymbol{\beta}_k$. As we can see there is heterogeneity across tasks both in the support (which elements are non-zero) and in the magnitude of the coefficients. But there is enough shared structure that borrowing strength across tasks would be expected to improve performance.

```
print(B[1:8,])
```

```
#>             [,1]       [,2]      [,3]      [,4]
#> [1,] -0.07979047 -0.6414113 0.3267767 0.7180235
#> [2,]  0.00000000  0.0000000 1.0559767 1.3721695
#> [3,]  0.00000000  2.2515126 1.8342540 2.0883395
#> [4,]  0.76785488  1.7435692 1.4888270 1.2915777
#> [5,]  1.22453355  0.3366593 0.0000000 0.0000000
#> [6,]  0.39290025  0.1788614 1.6207363 1.1085903
#> [7,]  0.08636348  0.0000000 0.6072575 1.8329919
#> [8,]  0.80643526  1.1063720 0.0000000 0.0000000
```

We will use `sMTL` to estimate $\mathbb{B}$ from the data.

We will start with fitting Common Support models.

Multi-Task Learning (Case 1): Fitting Common Support Regression Models

We start with the following model where we abbreviate the constraints above for conciseness.

$$\min_{\boldsymbol{z},\ \mathbb{B}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}_k \boldsymbol{\beta}_k||_2^2 + \lambda_1 ||\mathbb{B}||_2^2 \qquad (Common\ Support)$$
$$\text{subject to:} \quad ||\boldsymbol{\beta}_k||_0 \leq s, \ \forall\ k \in \{1, 2, ..., K\}$$

To fit a path of solutions for the Common Support Multi-Task model with 5 non-zero coefficients (i.e., $s = 5$) and a ridge penalty $\lambda_1 = 0.001$, we use the `smtl` function and then print out the coefficients stored in `fit`:

```
library(sMTL)
smtl_setup(path = "/Applications/Julia-1.5.app/Contents/Resources/julia/bin")

mod <- sMTL::smtl(y = y,
                  X = X,
                  study = task,
                  s = 5,
                  commonSupp = TRUE,
                  lambda_1 = 0.001)

print(mod$fit[1:8,])
```

```
#>            [,1]       [,2]      [,3]      [,4]
#> [1,] -0.3736541 -0.5943599 0.2273531 0.7587845
#> [2,] -0.2303298  0.1599438 0.6188391 1.3463789
#> [3,]  0.0103924  1.9894136 1.6556035 1.8058008
#> [4,]  0.8294115  1.9251914 1.4090303 1.2827704
#> [5,]  0.0000000  0.0000000 0.0000000 0.0000000
#> [6,]  0.6302355 -0.1557898 1.7822720 0.8641906
#> [7,]  0.2366567 -0.1520277 0.5094640 1.6759677
#> [8,]  0.0000000  0.0000000 0.0000000 0.0000000
```

As we can see the solutions have a common support, that is each row is either all zeros or all non-zeros.

Let's now try a model where we shrink the $\boldsymbol{\beta}_k$ towards each other:

$$\min_{\boldsymbol{z},\ \mathbb{B}, \bar{\boldsymbol{\beta}}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}_k \boldsymbol{\beta}_k||_2^2 + \lambda_2 \sum_{k=1}^{K} ||\boldsymbol{\beta}_k - \bar{\boldsymbol{\beta}}||_2^2 \qquad (Common\ Support)$$
$$\text{subject to:} \quad ||\boldsymbol{\beta}_k||_0 \leq s, \ \forall\ k \in \{1, 2, ..., K\}$$

```
mod <- smtl(y = y,
            X = X,
            study = task,
            s = 5,
            commonSupp = TRUE,
            lambda_2 = 0.1)

print(mod$fit[1:8,])
```

```
#>            [,1]        [,2]      [,3]      [,4]
#> [1,] -0.3171352 -0.62619968 0.2268748 0.7583876
#> [2,] -0.1681913  0.14101624 0.6113341 1.2802855
#> [3,]  0.1264364  1.91632422 1.6101000 1.8101617
```

```
#> [4,]  0.8936948  1.86725263 1.4092289 1.2655015
#> [5,]  0.0000000  0.00000000 0.0000000 0.0000000
#> [6,]  0.6213992 -0.07959537 1.6875496 0.8560316
#> [7,]  0.2137044 -0.07701170 0.5173653 1.5851054
#> [8,]  0.0000000  0.00000000 0.0000000 0.0000000
```

At times, we may not believe the supports are the same across the different $\boldsymbol{\beta}_k$. All we have to do is set `commonSupp = FALSE` and we end up with the Heterogeneous Support problem. Let's start with a simple model with a Ridge penalty:

$$\min_{\boldsymbol{z},\ \mathbb{B}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}_k \boldsymbol{\beta}_k||_2^2 + \lambda_1 ||\mathbb{B}||_2^2 \qquad (Heterogeneous\ Support)$$
$$\text{subject to:} \quad ||\boldsymbol{\beta}_k||_0 \leq s, \ \forall\ k \in \{1, 2, ..., K\}$$

```
mod <- smtl(y = y,
            X = X,
            study = task,
            s = 5,
            commonSupp = FALSE,
            lambda_1 = 0.001)

print(mod$fit[1:8,])
```

```
#>            [,1]       [,2]      [,3]      [,4]
#> [1,] -0.1289731 -0.6756479 0.2463807 0.7611506
#> [2,]  0.0000000  0.0000000 0.6040579 1.3494024
#> [3,]  0.0000000  2.1231651 1.7258090 1.8046491
#> [4,]  0.8506322  1.7494357 1.5165566 1.2826740
#> [5,]  1.2672647  0.0000000 0.0000000 0.0000000
#> [6,]  0.7379944  0.0000000 1.8365017 0.8676064
#> [7,]  0.0000000  0.0000000 0.0000000 1.6786831
#> [8,]  0.5575416  0.8479005 0.0000000 0.0000000
```

We can see that now many rows have a mix of zero and non-zero entries. With this model, however, we have essentially fit K independent sparse regression models. In order to borrow information across the supports of the models we add in a penalty and leave the full constraints in the formulation to emphasize the role of the $\boldsymbol{z}_k$:

$$\min_{\boldsymbol{z},\ \mathbb{B},\bar{\boldsymbol{\beta}}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}_k \boldsymbol{\beta}_k||_2^2 + \lambda_1 ||\mathbb{B}||_2^2 + \lambda_z \sum_{k=1}^{K} ||\boldsymbol{z}_k - \bar{\boldsymbol{z}}||_2^2 \qquad (Heterogeneous\ Support)$$
$$\text{subject to:} \quad z_{k,j} \in \{0, 1\} \quad \forall\ j \in [p]$$
$$\beta_{k,j}(1 - z_{k,j}) = 0\ \forall\ j \in [p], k \in [K]$$
$$\sum_{j=1}^{p} z_{k,j} \leq s \quad \forall\ k \in [K]$$

```
mod <- smtl(y = y,
            X = X,
            study = task,
            s = 5,
            commonSupp = FALSE,
            lambda_1 = 0.001,
            lambda_z = 0.25)
```

```
print(mod$fit[1:8,])
```

```
#>              [,1]       [,2]      [,3]      [,4]
#> [1,] -0.366273075 -0.6092721 0.2331170 0.7605889
#> [2,]  0.001851597  0.1482939 0.6292039 1.3476616
#> [3,]  0.011703748  2.0053410 1.6457597 1.7938859
#> [4,]  0.821286543  1.9176057 1.4213124 1.2833542
#> [5,]  1.002124674  0.0000000 0.0000000 0.0000000
#> [6,]  0.637081522 -0.1427352 1.7752111 0.8742219
#> [7,]  0.000000000  0.0000000 0.5136111 1.6748070
#> [8,]  0.000000000  0.8081371 0.0000000 0.0000000
```

As we can see, the additional penalty reduced the support heterogeneity. Most rows are now either all zeros or all non-zeros. Now let's turn the λ_z penalty up to show that the Common Support problem is a special case when $\lambda_z \to \infty$ (although in practice λ_z does not need to be very big to induce a solution with a common support).

```
mod <- smtl(y = y,
            X = X,
            study = task,
            s = 5,
            commonSupp = FALSE,
            lambda_1 = 0.001,
            lambda_z = 10)

print(mod$fit[1:8,])
```

```
#>              [,1]       [,2]      [,3]      [,4]
#> [1,] -0.36627307 -0.6092721 0.2331170 0.7605889
#> [2,] -0.21046269  0.1482939 0.6292039 1.3476616
#> [3,]  0.01170375  2.0053410 1.6457597 1.7938859
#> [4,]  0.82128654  1.9176057 1.4213124 1.2833542
#> [5,]  0.00000000  0.0000000 0.0000000 0.0000000
#> [6,]  0.63708152 -0.1427352 1.7752111 0.8742219
#> [7,]  0.22985555 -0.1465309 0.5136111 1.6748070
#> [8,]  0.00000000  0.0000000 0.0000000 0.0000000
```

Multi-Task Learning (Case 1): Predictions

Now that we have a model let's make predictions and see the output. We feed `predict()` a model object and some new data `X`, where here we use some of the training data for simplicity but in practice would likely be the covariates of new observations.

```
preds <- sMTL::predict(model = mod, X = X[1:5,])
head(preds)
```

We used each model to make predictions on the new data producing a $n^* \times K$ matrix where n^* is the sample size of the data we make predictions on. In practice we may only want the predictions associated with one of the tasks in which we would pick out the corresponding column.

Cross Validation

Fitting Whole paths

This will generate solutions for a sequence of λ values (chosen automatically by the algorithm). To view the sequence of λ along with the associated support sizes (i.e., the number of non-zeros), we use the `print` method as follows:

```
print(fit)
```

To extract the estimated B for particular values of λ and γ, we use the function `coef(fit,lambda,gamma)`. For example, the solution at $\lambda = 0.0325142$ (which corresponds to a support size of 10) can be extracted using

```
coef(fit, lambda=0.0325142, gamma=0)
```

The output is a sparse vector of type `dgCMatrix`. The first element in the vector is the intercept and the rest are the B coefficients. Aside from the intercept, the only non-zeros in the above solution are coordinates 1, 2, 3, ..., 10, which are the non-zero coordinates in the true support (used to generated the data). Thus, this solution successfully recovers the true support. Note that on some BLAS implementations, the `lambda` value we used above (i.e., `0.0325142`) might be slightly different due to the limitations of numerical precision. Moreover, all the solutions in the regularization path can be extracted at once by calling `coef(fit)`.

The sequence of λ generated by `L0Learn` is stored in the object `fit`. Specifically, `fit$lambda` is a list, where each element of the list is a sequence of λ values corresponding to a single value of γ. Since L0 has only one value of γ (i.e., 0), we can access the sequence of λ values using `fit$lambda[[1]]`. Thus, $\lambda = 0.0325142$ we used previously can be accessed using `fit$lambda[[1]][7]` (since it is the 7th value in the output of `print`). So the previous solution can also be extracted using `coef(fit,lambda=fit$lambda[[1]][7], gamma=0)`.

We can make predictions using a specific solution in the grid using the function `predict(fit,newx,lambda,gamma)` where newx is a testing sample (vector or matrix). For example, to predict the response for the samples in the data matrix X using the solution with $\lambda = 0.0325142$, we call the prediction function as follows:

```
predict(fit, newx=X, lambda=0.0325142, gamma=0)
```

We can also visualize the regularization path by plotting the coefficients of the estimated B versus the support size (i.e., the number of non-zeros) using the `plot(fit,gamma)` method as follows:

```
plot(fit, gamma=0)
```

The legend of the plot presents the variables in the order they entered the regularization path. For example, variable 7 is the first variable to enter the path, and variable 6 is the second to enter. Thus, roughly speaking, we can view the first k variables in the legend as the best subset of size k. To show the lines connecting the points in the plot, we can set the parameter `showLines=TRUE` in the `plot` function, i.e., call `plot(fit, gamma=0, showLines=TRUE)`. Moreover, we note that the output of the `plot` function above is a `ggplot` object, which can be further customized using the `ggplot2` package.

Fitting L0L2 and L0L1 Regression Models

We have demonstrated the simple case of using an L0 penalty. We can also fit more elaborate models that combine L0 regularization with shrinkage-inducing penalties like the L1 norm or squared L2 norm. Adding shrinkage helps in avoiding overfitting and typically improves the predictive performance of the models. Next, we will discuss how to fit a model using the L0L2 penalty for a two-dimensional grid of λ and γ values. Recall that by default, `L0Learn` automatically selects the λ sequence, so we only need to specify the γ sequence. Suppose we want to fit an L0L2 model with a maximum of 20 non-zeros and a sequence of 5 γ values ranging between 0.0001 and 10. We can do so by calling `L0Learn.fit` with `penalty="L0L2"`, `nGamma=5`, `gammaMin=0.0001`, and `gammaMax=10` as follows:

```
fit <- L0Learn.fit(X, y, penalty="L0L2", nGamma = 5, gammaMin = 0.0001, gammaMax = 10, maxSuppSize=20)
```

`L0Learn` will generate a grid of 5 γ values equi-spaced on the logarithmic scale between 0.0001 and 10. Similar to the case of L0, we can print a summary of the regularization path using the `print` function as follows:

```
print(fit)
```

The sequence of γ values can be accessed using `fit$gamma`. To extract a solution we use the `coef` method. For example, extracting the solution at $\lambda = 0.0011539$ and $\gamma = 10$ can be done using

```
coef(fit,lambda=0.0011539, gamma=10)
```

Similarly, we can predict the response at this pair of λ and γ for the matrix X using

```
predict(fit, newx=X, lambda=0.0011539, gamma=10)
```

The regularization path can also be plot at a specific γ using `plot(fit, gamma)`. Finally, we note that fitting an L0L1 model can be done by just changing the `penalty` to "L0L1" in the above (in this case `gammaMax` will be ignored since it is automatically selected by the toolkit; see the reference manual for more details.)

Higher-quality Solutions using Local Search

By default, `L0Learn` uses coordinate descent (CD) to fit models. Since the objective function is non-convex, the choice of the optimization algorithm can have a significant effect on the solution quality (different algorithms can lead to solutions with very different objective values). A more elaborate algorithm based on combinatorial search can be used by setting the parameter `algorithm="CDPSI"` in the call to `L0Learn.fit`. `CDPSI` typically leads to higher-quality solutions compared to CD, especially when the features are highly correlated. CDPSI is slower than CD, however, for typical applications it terminates in the order of seconds.

Cross-validation

We will demonstrate how to use K-fold cross-validation (CV) to select the optimal values of the tuning parameters λ and γ. To perform CV, we use the `L0Learn.cvfit` function, which takes the same parameters as `L0Learn.fit`, in addition to the number of folds using the `nFolds` parameter and a seed value using the `seed` parameter (this is used when randomly shuffling the data before performing CV).

For example, to perform 5-fold CV using the `L0L2` penalty (over a range of 5 `gamma` values between 0.0001 and 0.1) with a maximum of 50 non-zeros, we run:

```
cvfit = L0Learn.cvfit(X, y, nFolds=5, seed=1, penalty="L0L2", nGamma=5, gammaMin=0.0001, gammaMax=0.1, m
```

Note that the object `cvfit` has a member `fit` (accessed using `cvfit$fit`) which is output of running `L0Learn.fit` on (y,X). The cross-validation errors can be accessed using the `cvMeans` attribute of `cvfit`: `cvfit$cvMeans` is a list where the ith element, `cvfit$cvMeans[[i]]`, stores the cross-validation errors for the ith value of gamma (`cvfit$fit$gamma[i]`). To find the minimum cross-validation error for every `gamma`, we call the `min` function for every element in the list `cvfit$cvMeans`, as follows:

```
lapply(cvfit$cvMeans, min)
```

The above output indicates that the 4th value of gamma achieves the lowest CV error (`=0.9899542`). We can plot the CV errors against the support size for the 4th value of gamma, i.e., `gamma = cvfit$fit$gamma[4]`, using:

```
plot(cvfit, gamma=cvfit$fit$gamma[4])
```

The above plot is produced using the `ggplot2` package and can be further customized by the user. To extract the optimal λ (i.e., the one with minimum CV error) in this plot, we execute the following:

```
optimalGammaIndex = 4 # index of the optimal gamma identified previously
optimalLambdaIndex = which.min(cvfit$cvMeans[[optimalGammaIndex]])
optimalLambda = cvfit$fit$lambda[[optimalGammaIndex]][optimalLambdaIndex]
optimalLambda
```

To print the solution corresponding to the optimal gamma/lambda pair:

```
coef(cvfit, lambda=optimalLambda, gamma=cvfit$fit$gamma[4])
```

The optimal solution (above) selected by cross-validation correctly recovers the support of the true vector of coefficients used to generated the model.

Fitting Classification Models

All the commands and plots we have seen in the case of regression extend to classification. We currently support logistic regression (using the parameter `loss = "Logistic"`) and a smoothed version of SVM (using the parameter `loss="SquaredHinge"`). To give some examples, we first generate a synthetic classification dataset (similar to the one we generated in the case of regression):

```
set.seed(1) # fix the seed to get a reproducible result
X = matrix(rnorm(500*1000),nrow=500,ncol=1000)
B = c(rep(1,10),rep(0,990))
e = rnorm(500)
y = sign(X%*%B + e)
```

Advanced Options

User-specified Lambda Grids

By default, `L0Learn` selects the sequence of lambda values in an efficient manner to avoid wasted computation (since close λ values can typically lead to the same solution). Advanced users of the toolkit can change this default behavior and supply their own sequence of λ values. This can be done supplying the λ values through the parameter `lambdaGrid`. L0Learn versions before 2.0.0 would also require setting the `autoLambda` parameter to `FALSE`. This parameter remains in version 2.0.0 for backwards compatibility, but is no longer needed or used.

Specifically, the value assigned to `lambdaGrid` should be a list of lists of decreasing positive values (doubles). The length of `lambdaGrid` (the number of lists stored) specifies the number of gamma parameters that will fill between `gammaMin`, and `gammaMax`. In the case of L0 penalty, `lambdaGrid` must be a list of length 1. In case of L0L2/L0L1 `lambdaGrid` can have any number of sub-lists stored. The length of `lambdaGrid` (the number of lists stored) specifies the number of gamma parameters that will fill between `gammaMin`, and `gammaMax`. The ith element in `LambdaGrid` should be a **decreasing** sequence of positive lambda values which are used by the algorithm for the ith value of gamma. For example, to fit an L0 model with the sequence of user-specified lambda values: 1, 1e-1, 1e-2, 1e-3, 1e-4, we run the following:

```
userLambda <- list()
userLambda[[1]] <- c(1, 1e-1, 1e-2, 1e-3, 1e-4)
fit <- L0Learn.fit(X, y, penalty="L0", lambdaGrid=userLambda, maxSuppSize=1000)
```

To verify the results we print the fit object:

```
print(fit)
```

Note that the λ values above are the desired values. For L0L2 and L0L1 penalties, the same can be done where the `lambdaGrid` parameter.

```r
userLambda <- list()
userLambda[[1]] <- c(1, 1e-1, 1e-2, 1e-3, 1e-4)
userLambda[[2]] <- c(10, 2, 1, 0.01, 0.002, 0.001, 1e-5)
userLambda[[3]] <- c(1e-4, 1e-5)
# userLambda[[i]] must be a vector of positive decreasing reals.
fit <- L0Learn.fit(X, y, penalty="L0L2", lambdaGrid=userLambda, maxSuppSize=1000)
```

```r
print(fit)
```

References

Hussein Hazimeh and Rahul Mazumder. Fast Best Subset Selection: Coordinate Descent and Local Combinatorial Optimization Algorithms. Operations Research (2020).