sMTL Vignette

true true

2022-12-14

Contents

Introduction **1**

Installation **2**

Package Setup **3**

Tutorial Guide **3**

Multi-Task Learning (Case 1) **4**
1) Common Support Models with Ridge Penalty . . . 5
1) Common Support Models with Cross-Task Coefficient Shrinkage . . . 5
1) Heterogeneous Support Models with Ridge Penalty . . . 6
1) Heterogeneous Support Models with Support Heterogeneity Regularization . . . 7
1) Predictions . . . 8

Multi-Label Learning (Case 2) **8**
2) Common Support Models with Ridge Penalty . . . 9
2) Heterogeneous Support Models . . . 10

Domain-Generalization (Case 3) **10**
3) Heterogeneous Support Model with Ridge Penalty . . . 11

Cross Validation **12**
User Specified Grids . . . 13

Advanced Options **14**
Local Search Options for Cross Validation . . . 14
Two Stage Cross Validation for Solution Quality and Speed . . . 14
Fitting Whole Paths . . . 16
Predictions on Whole Path Fits . . . 16

References **16**

Introduction

`sMTL` is a fast toolkit for ℓ_0-constrained Multi-Task Learning, Multi-Label Learning and Domain Generalization. ℓ_0 constraints select the best subset of features in a variety of multi-dataset settigs. The toolkit can (approximately) solve the following two problems

$$\min_{\mathbb{B},\bar{\boldsymbol{\beta}}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}_k \boldsymbol{\beta}_k||_2^2 + \lambda_1 ||\mathbb{B}||_2^2 + \lambda_2 \sum_{k=1}^{K} ||\boldsymbol{\beta}_k - \bar{\boldsymbol{\beta}}||_2^2 \qquad \text{(Common Support)}$$
$$\text{subject to: } \text{Supp}(\boldsymbol{\beta}_k) = \text{Supp}(\boldsymbol{\beta}) \qquad \text{for } k \in \{1, 2, ..., K\}$$
$$||\boldsymbol{\beta}_k||_0 \leq s \qquad \text{for } k \in \{1, 2, ..., K\}$$

$$\min_{\boldsymbol{z},\ \mathbb{B},\bar{\boldsymbol{\beta}}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}_k \boldsymbol{\beta}_k||_2^2 + \lambda_1 ||\mathbb{B}||_2^2 + \lambda_2 \sum_{k=1}^{K} ||\boldsymbol{\beta}_k - \bar{\boldsymbol{\beta}}||_2^2 + \lambda_z \sum_{k=1}^{K} ||\boldsymbol{z}_k - \bar{\boldsymbol{z}}||_2^2 \qquad \text{(Heterogeneous Support)}$$
$$\text{subject to: } \boldsymbol{z}_k = \text{Supp}(\boldsymbol{\beta}_k) \qquad \text{for } k \in \{1, 2, ..., K\}$$
$$||\boldsymbol{\beta}_k||_0 \leq s \qquad \text{for } k \in \{1, 2, ..., K\}$$

where $\boldsymbol{\beta}_k$ is the vector of coefficients associated with dataset k, and $\text{Supp}(\boldsymbol{\beta}_k) = \boldsymbol{z}_k = \mathbb{1}(\boldsymbol{\beta}_k \neq \mathbf{1})$, 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. Note $||\boldsymbol{\beta}_k||_0 = \sum_{j=1}^{p} z_{k,j}$, where the ℓ_0 norm, $||\boldsymbol{u}||_0$, equals the number of non-zero entries in the vector $\boldsymbol{u}$. For example,

$$\boldsymbol{\beta} = \begin{bmatrix} 1.23 \\ 0 \\ -0.71 \\ 0 \end{bmatrix} \qquad \text{Supp}(\boldsymbol{\beta}) = \boldsymbol{z} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

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

The parameter λ_2 shrinks the $\boldsymbol{\beta}_k$ towards a common $\bar{\boldsymbol{\beta}} = \frac{1}{K} \sum_{k=1}^{K} \boldsymbol{\beta}_k$. The parameter λ_1 controls the strength of ℓ_2 (Ridge) shrinkage where $\lambda_1 ||\mathbb{B}||_2^2 = \lambda_1 \sum_{k=1}^{K} ||\boldsymbol{\beta}_k||_2^2$. 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 λ_z controls the degree to which the $\boldsymbol{z}$, the supports of the $\boldsymbol{\beta}_k$, are shrunk towards a common $\bar{\boldsymbol{z}} = \frac{1}{K} \sum_{k=1}^{K} \boldsymbol{z}_k$. When λ_z is high, this results in solutions equivalent to the Common Support approach. The fitting is done over a grid of λ_1, λ_2 and λ_z values to generate a regularization path.

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 Multi-Task Learning for Sparsity Pattern Heterogeneity: A Discrete Optimization Approach.

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. We 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 (after this initial setup, subsequent uses of the `sMTL` package do not require any additional steps beyond the standard package load):

Package Setup

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

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

2. A few `Julia` packages are required (e.g., `TSVD`, `LinearAlgebra`, `Statistics`). We include a function to automatically install these if they are not already. 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)
smtl_setup()
```

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

Tutorial Guide

To illustrate the main `sMTL` functions, we start by generating synthetic datasets and proceed by fitting models. We use terminology from Zhang & Yang, 2021. 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 defined by their design (feature) matrices, $\{\mathbb{X}_1, \mathbb{X}_2, ..., \mathbb{X}_K\}$, and outcome vectors, $\{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_K\}$, where $\mathbb{X}_k \in \mathbb{R}^{n_k \times p}$, $\mathbf{y}_k \in \mathbb{R}^{n_k}$. We jointly train K models, one to each of the K datasets with the goal of making predictions on new observations from task k using model k.

2. Multi-Label Learning is a special case of Multi-Task Learning (1) in which we have a collection of tasks with a single common design matrix, $\mathbb{X} \in \mathbb{R}^{n \times p}$, but separate task-specific outcome vectors, $\mathbf{y}_k \in \mathbb{R}^n$. One can think of each observation having a K dimensional multivariate outcome. We jointly train K models, one with each of the K outcome vectors, $\mathbf{y}_k$, with the goal of making predictions on new observations from task k (i.e., to predict an observation of outcome k) using model k. While this is equivalent to (1) for the case that $\mathbb{X}_k = \mathbb{X}\ \forall k \in \{1, 2, ..., K\}$ but we include specialized algorithms for this case that yield gains in efficiency in terms of memory and solve time.

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 $\mathbb{X}_k \in \mathbb{R}^{n_k \times p}$, $\mathbf{y}_k \in \mathbb{R}^{n_k}$. Unlike in (1) and (2), 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. We calculate ensemble weights based upon multi-study stacking (Patil and Parmigiani, 2018) and simple average weights. The code to fit models is identical to Case 1 (Multi-Task Learning). The only difference comes in how one tunes the models and makes predictions.

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(K * (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
}
colnames(B) <- paste0("beta_", 1:K)
rownames(B) <- paste0("X_", 1:(p+1))
```

Note that the `task` vector is a $N \times 1$ vector where element i indicates the task index ($\in \{1, 2, ..., K\}$) 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(round(B[1:8,],2))
```

```
#>     beta_1 beta_2 beta_3 beta_4
#> X_1  -0.08  -0.64   0.33   0.72
```

```
#> X_2   1.61   0.00   1.06   1.37
#> X_3  -0.50   2.25   1.83   0.00
#> X_4   0.00   1.74   1.49   1.29
#> X_5   1.22   0.34   0.00   0.81
#> X_6   0.39   0.00   1.62   1.11
#> X_7   0.00   2.03   0.61   0.00
#> X_8   0.81   1.11   0.00   2.26
```

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

We will start with fitting Common Support models.

1) Common Support Models with Ridge Penalty

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

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

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 `beta`:

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

print(round(mod$beta[1:8,],2))
```

```
#>           beta_1 beta_2 beta_3 beta_4
#> Intercept  -0.37  -0.60   0.36   1.13
#> V1          1.51  -0.11   1.16   1.49
#> V2         -0.43   2.28   1.74   0.27
#> V3          0.29   1.52   1.65   1.41
#> V4          0.00   0.00   0.00   0.00
#> V5          0.00   0.00   0.00   0.00
#> V6          0.46   2.21   0.80  -0.25
#> V7          0.45   1.02  -0.25   2.23
```

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

1) Common Support Models with Cross-Task Coefficient Shrinkage

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

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

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

print(round(mod$beta[1:8,],2))
```

```
#>           beta_1 beta_2 beta_3 beta_4
#> Intercept  -0.35  -0.59   0.34   1.11
#> V1          1.45  -0.04   1.13   1.44
#> V2         -0.30   2.19   1.69   0.29
#> V3          0.37   1.53   1.65   1.40
#> V4          0.00   0.00   0.00   0.00
#> V5          0.00   0.00   0.00   0.00
#> V6          0.47   2.07   0.76  -0.15
#> V7          0.52   1.00  -0.11   2.10
```

1) Heterogeneous Support Models with Ridge Penalty

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:

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

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

print(round(mod$beta[1:8,],2))
```

```
#>           beta_1 beta_2 beta_3 beta_4
#> Intercept  -0.16  -0.57   0.25   0.88
#> V1          1.74   0.00   0.98   1.42
#> V2         -0.61   2.30   1.89   0.00
#> V3          0.00   1.48   1.78   1.34
#> V4          1.21   0.00   0.00   0.77
#> V5          0.35   0.00   1.51   1.10
#> V6          0.00   2.17   0.49   0.00
#> V7          0.91   1.09   0.00   2.27
```

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 the next section, we borrow information across the supports of the models by adding in a penalty.

1) Heterogeneous Support Models with Support Heterogeneity Regularization

We now shrink the supports of the $\boldsymbol{\beta}_k$, $\boldsymbol{z}_k$, towards each other. We leave the full constraints in the formulation to emphasize the role of the $\boldsymbol{z}_k$:

$$\min_{\boldsymbol{z},\ \mathbb{B},\boldsymbol{\beta}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}_k\boldsymbol{\beta}_k||_2^2 + \lambda_1||\mathbb{B}||_2^2 + \lambda_2 \sum_{k=1}^{K} ||\boldsymbol{\beta}_k - \bar{\boldsymbol{\beta}}||_2^2 + \lambda_z \sum_{k=1}^{K} ||\boldsymbol{z}_k - \bar{\boldsymbol{z}}||_2^2 \qquad \text{(Heterogeneous Support)}$$

$$\text{subject to:} \quad \boldsymbol{z}_k = \text{Supp}(\boldsymbol{\beta}_k) \qquad \text{for } k \in \{1, 2, ..., K\}$$

$$||\boldsymbol{\beta}_k||_0 \leq s \qquad \text{for } k \in \{1, 2, ..., K\}$$

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

print(round(mod$beta[1:8,], 2))
```

```
#>           beta_1 beta_2 beta_3 beta_4
#> Intercept  -0.36  -0.60   0.37   1.10
#> V1          1.50  -0.12   1.18   1.48
#> V2         -0.44   2.27   1.75   0.00
#> V3          0.28   1.52   1.67   1.40
#> V4          0.96   0.00   0.00   0.76
#> V5          0.00   0.00   1.42   0.95
#> V6          0.00   2.20   0.00   0.00
#> V7          0.48   1.02  -0.08   2.23
```

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(round( mod$beta[1:8,], 2))
```

```
#>           beta_1 beta_2 beta_3 beta_4
#> Intercept  -0.36  -0.60   0.37   1.10
#> V1          1.50  -0.12   1.18   1.48
#> V2         -0.44   2.27   1.75   0.25
#> V3          0.28   1.52   1.67   1.40
#> V4          0.00   0.00   0.00   0.00
#> V5          0.00   0.00   0.00   0.00
#> V6          0.43   2.20   0.79  -0.24
#> V7          0.48   1.02  -0.26   2.23
```

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`. Here we make predictions on the training data for simplicity but in practice we would likely be more interested in predictions on new observations.

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

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.

Multi-Label Learning (Case 2)

Multi-Label Learning can be cast as a special case of Multi-Task Learning (Case 1). The main difference here is that we can imagine that we have a single $\mathbb{X} \in \mathbb{R}^{n \times p}$ that is common to all tasks. Put another way, we have one $\mathbb{X}$ and we store all the $\mathbf{y}_k$ as columns in a matrix (i.e., a multivariate outcome, $\mathbb{Y} \in \mathbb{R}^{p \times K}$). We set $K = 4$ and generate data from a sparse linear model with the following:

- $\mathbb{X}$ 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 $\mathbf{y}_k = \mathbb{X}\boldsymbol{\beta}_k + \boldsymbol{\epsilon}_k$

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 <- 50 # full dataset samplesize
X <- matrix( rnorm(N * p), nrow = N, ncol=p) # full design matrix
B <- matrix(1 + rnorm(K * (p+1) ), nrow = p + 1, ncol = K) # betas before making sparse
Z <- matrix(0, nrow = p, ncol = K) # matrix of supports
y <- matrix(nrow = N, ncol = K) # 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

# iterate through and make each task specific dataset
for(j in 1:K){
    e <- rnorm(N)
    y[,j] <- B[1, j] + X %*% B[-1,j] + e
}
colnames(B) <- paste0("beta_", 1:K)
rownames(B) <- paste0("X_", 1:(p+1))
```

Note that we do not have a `task` vector here. Instead we concatenate the outcomes $\boldsymbol{y}_k$ into a matrix, $\mathbb{Y} \in \mathbb{R}^{n \times K}$.

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(round(B[1:8,],2))
```

```
#>     beta_1 beta_2 beta_3 beta_4
#> X_1  -0.52   2.38   2.31   1.08
#> X_2   1.63   2.34   0.00   0.00
#> X_3  -0.68   0.24   0.30   1.73
#> X_4   0.00   0.00   1.87   0.00
#> X_5   2.12   2.00   0.21  -0.22
#> X_6  -0.24   0.00   1.55   0.89
#> X_7   0.00   0.69   0.00   0.32
#> X_8   1.60   1.72   0.18   1.49
```

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

We will start with fitting Common Support models.

2) Common Support Models with Ridge Penalty

We start with the following model where we abbreviate the constraints above for conciseness. Notice that there is a single $\mathbb{X}$ for all tasks.

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

To fit a 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 `beta`. Notice here the y object in `R` is a matrix, and that we do not include the `study=` syntax that we use for Multi-Task learning or Domain Generalization. Instead the columns of the y object are treated as separate "labels'' (akin to separate tasks). Otherwise the syntax is the exact same.

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

print(round(mod$beta[1:8,],2))
```

```
#>           beta_1 beta_2 beta_3 beta_4
#> Intercept  -0.53   2.29   2.31   1.42
#> V1          1.69   2.07  -0.03  -0.16
#> V2          0.00   0.00   0.00   0.00
#> V3          0.10   0.07   2.16  -0.01
#> V4          2.06   2.02   0.04  -0.25
#> V5         -0.39   0.14   1.55   0.87
#> V6          0.00   0.00   0.00   0.00
#> V7          1.58   1.61   0.21   1.39
```

2) Heterogeneous Support Models

We next show code for the Heterogeneous Support case. As we have already introduced predictions above, we include the prediction code in the same block.

$$\min_{\boldsymbol{z},\ \mathbb{B},\bar{\boldsymbol{\beta}}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}\boldsymbol{\beta}_k||_2^2 + \lambda_1 ||\mathbb{B}||_2^2 + \lambda_2 \sum_{k=1}^{K} ||\boldsymbol{\beta}_k - \bar{\boldsymbol{\beta}}||_2^2 + \lambda_z \sum_{k=1}^{K} ||\boldsymbol{z}_k - \bar{\boldsymbol{z}}||_2^2 \qquad \text{(Heterogeneous Support)}$$

$$\text{subject to:} \quad \boldsymbol{z}_k = \text{Supp}(\boldsymbol{\beta}_k) \qquad \text{for } k \in \{1, 2, ..., K\}$$
$$||\boldsymbol{\beta}_k||_0 \leq s \qquad \text{for } k \in \{1, 2, ..., K\}$$

```
mod <- sMTL::smtl(y = y,
                  X = X,
                  s = 5,
                  commonSupp = TRUE,
                  lambda_1 = 0.001,
                  lambda_z = 0.1)

print(round(mod$beta[1:8,],2))
```

```
#>           beta_1 beta_2 beta_3 beta_4
#> Intercept  -0.53   2.29   2.31   1.42
#> V1          1.69   2.07  -0.03  -0.16
#> V2          0.00   0.00   0.00   0.00
#> V3          0.10   0.07   2.16  -0.01
#> V4          2.06   2.02   0.04  -0.25
#> V5         -0.39   0.14   1.55   0.87
#> V6          0.00   0.00   0.00   0.00
#> V7          1.58   1.61   0.21   1.39
```

Notice that the predictions are labeled as "task_1",...,"task_K." Since this is a "Multi-Label'' learning problem, we would interpret "task_j" to mean "label_j" (or "otucome_j").

Domain-Generalization (Case 3)

The syntax and setup here is basically identical to Case 1. The main difference comes in how we tune the models and make predictions. For that reason, we include an example involving tuning, fitting and predictions.

We again 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
```

```r
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(K * (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
}
colnames(B) <- paste0("beta_", 1:K)
rownames(B) <- paste0("X_", 1:(p+1))
```

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

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 domain-specific regression coefficients 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 domains would be expected to improve performance.

```r
print(round(B[1:8,],2))
```

```
#>     beta_1 beta_2 beta_3 beta_4
#> X_1  -0.08  -0.64   0.33   0.72
#> X_2   1.61   0.00   1.06   1.37
#> X_3  -0.50   2.25   1.83   0.00
#> X_4   0.00   1.74   1.49   1.29
#> X_5   1.22   0.34   0.00   0.81
#> X_6   0.39   0.00   1.62   1.11
#> X_7   0.00   2.03   0.61   0.00
#> X_8   0.81   1.11   0.00   2.26
```

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

We will start with fitting Common Support models.

3) Heterogeneous Support Model with Ridge Penalty

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

$$\min_{\boldsymbol{z},\ \mathbb{B}} \sum_{k=1}^{K} \frac{1}{n_k} ||\mathbf{y}_k - \mathbb{X}_k \boldsymbol{\beta}_k||_2^2 + \lambda_1 ||\mathbb{B}||_2^2 \qquad \text{(Heterogeneous Support)}$$

$$\text{subject to: } \ ||\boldsymbol{\beta}_k||_0 \leq s, \ \forall\ k \in \{1, 2, ..., K\}$$

Here we use the same syntax from Case 1 except for 1) in the `cv.smtl()` function we set `multiTask = FALSE` and 2) when making predictions, we set `stack = TRUE` in the `predict()` function to use multi-study stacking to generate ensemble weights. The `predict()` function then produces aggregate predictions for some new dataset or "domain'' by either taking a weighted average of the model-specific predictions: $\hat{f}(\boldsymbol{x}) = \sum_{k=1}^{K} w_k \boldsymbol{x}^T \hat{\boldsymbol{\beta}}_k$. We provide the output using average weights, w_k, (i.e., $w_k = 1/K, \forall\ k\ \in \{1, 2, ..., K\}$), or multi-study stacking weights, w_k.

```
# tuning grid
grid <- data.frame(s = c(4, 4, 5, 5),
                   lambda_1 = c(0.01, 0.1, 0.01, 0.1),
                   lambda_2 = rep(0, 4),
                   lambda_z = c(0.01, 0.1, 0.01, 0.1)
                   )

# cross validation
tn <- cv.smtl(y = y,
              X = X,
              study = task,
              commonSupp = FALSE,
              grid = grid,
              nfolds = 5,
              multiTask = FALSE)

# model fitting
mod <- sMTL::smtl(y = y,
                  X = X,
                  study = task,
                  s = tn$best.1se$s,
                  commonSupp = TRUE,
                  lambda_1 = tn$best.1se$lambda_1,
                  lambda_z = tn$best.1se$lambda_z)
```

Cross Validation

We can use our cross validation function to tune models. The syntax is similar to the `smtl()` function except one can specify an optional tuning grid, and the number of folds (`nfolds`). For Domain Generalization problems, one needs to switch `multiTask=FALSE` to change the cross validation to a hold-one-domain-out cross validation approach appropriate for the problem.

Here we show a Multi-Task Learning problem in which we let the `cv.smtl()` function generate a grid of hyperparameters for tuning internally, tune and then we fit a final model with the tuned hyperparameters.

```
# cross validation with internally generated grid
tn <- cv.smtl(y = y,
              X = X,
              study = task,
              commonSupp = FALSE,
              lambda_1 = TRUE,
              lambda_2 = FALSE,
              lambda_z = TRUE,
              nfolds = 5)

# fit final model
mod <- smtl(y = y,
```

```
            X = X,
            study = task,
            s = tn$best$s,
            commonSupp = FALSE,
            lambda_1 = tn$best$lambda_1,
            lambda_z = tn$best$lambda_z)
```

We see that instead of providing a grid, we chose which hyperparameters we wanted to tune by setting them to `TRUE`.

User Specified Grids

We highly recommend specifying your own grids as good ranges for hyperparameter values may be very problem-specific. Next we show the same Multi-Task Learning problem with a custom grid and then fit a final model with the tuned hyperparameters.

```
# generate grid
grid <- data.frame(s = c(4,4,5,5),
                   lambda_1 = c(0.01, 0.1,0.01, 0.1),
                   lambda_2 = 0,
                   lambda_z = c(0.01, 0.1,0.01, 0.1))

# cross validation with custom grid
tn <- cv.smtl(y = y,
              X = X,
              study = task,
              commonSupp = FALSE,
              grid = grid,
              nfolds = 5)

# fit final model
mod <- smtl(y = y,
            X = X,
            study = task,
            s = tn$best$s,
            commonSupp = FALSE,
            lambda_1 = tn$best$lambda_1,
            lambda_z = tn$best$lambda_z)
```

Now we show a Domain-Generalization problem with a custom grid and then fit a final model with the tuned hyperparameters. We switch `multiTask=FALSE` and proceed as normal.

For illustrative purposes, we also choose the `best.1se` parameter values which chooses the smallest s that achieves an average cross validated prediction error within 1 standard error of the parameters with the best cross validated (and the best regularization parameters for that s). This can result in sparser models and is an alternative option.

```
# generate grid
grid <- data.frame(s = c(4,4,5,5),
                   lambda_1 = c(0.01, 0.1,0.01, 0.1),
                   lambda_2 = 0,
                   lambda_z = c(0.01, 0.1,0.01, 0.1))

# cross validation with custom grid
tn <- cv.smtl(y = y,
```

```
              X = X,
              study = task,
              commonSupp = FALSE,
              grid = grid,
              nfolds = 5,
              multiTask = FALSE)

# fit final model
mod <- smtl(y = y,
            X = X,
            study = task,
            s = tn$best$s,
            commonSupp = FALSE,
            lambda_1 = tn$best$lambda_1,
            lambda_z = tn$best$lambda_2)
```

Advanced Options

Local Search Options for Cross Validation

There are a few advanced options for users looking to adjust the speed of the algorithms and quality of solutions. The `maxIter` argument can be increased to improve the quality of the coordinate descent algorithm or reduced to improve speed (at a cost to the solution quality). Similarly, `LocSrch_maxIter` argument specifies the number of local search iterations and increasing cit an improve the quality of the solutions, but can slow down the algorithm substantially. Setting `LocSrch_maxIter=0` tells the package to use no local search and may even be preferable for some problems. Finally, increasing `LocSrch_skip` to an integer above 1 will increase the speed of the algorithm by only conducting local search on a subset of parameter values on the solution path. For example, `LocSrch_skip=3` only uses local search on every 3^{rd} parameter value. If some local search is desirable, but one wishes to avoid using it at every parameter value because it is too slow, setting `LocSrch_skip` to somewhere between 2 and 5 could speed up cross-validation substantially while still retaining some of the benefits of local search.

```
# cross validation with custom grid using
tn <- cv.smtl(y = y,
              X = X,
              study = task,
              commonSupp = FALSE,
              grid = grid,
              LocSrch_skip = 3,
              LocSrch_maxIter = 5,
              nfolds = 5,
              multiTask = FALSE)
```

Two Stage Cross Validation for Solution Quality and Speed

Discrete optimization offers modeling richness at the cost of additional tuning parameters. For example, tuning a Heterogeneous Support model with a Zbar penalty has 3 hyperparameters (s, λ_a and λ_z) and thus tuning across a full tuning grid may incur a large computational cost. One solution to tune hyperparameters with a lower computational burden is to tune in two stages. First, in stage 1 we tune across a smaller grid with just s and λ_z and a small, fixed $lambda_1$ for numerical reasons. Then in stage 2, we take the optimal hyperparameter values and create a second grid built around a neighborhood of the optimal hyperparameter

values from stage 1 and perform a second cross validation. This way we avoid making a large grid with all three hyperparameters s, λ_a and λ_z.

```
####################################
# generate initial tuning grid
####################################
s <- c(5, 10, 15, 20)
lambda_1 <- c(1e-6) # small fixed ridge penalty for regularization
lambda_2 <- 0
lambda_z <- sort( unique( c(0, 1e-6, 1e-5, 1e-4, 1e-3,
                            exp(-seq(0,5, length = 8)),  1, 3) ),  decreasing = TRUE )

grid <- expand.grid(s, lambda_1, lambda_2, lambda_z)
colnames(grid) <- c("s", "lambda_1", "lambda_2", "lambda_z")

# order correctly
grid <- grid[  order(-grid$s,
                     grid$lambda_1,
                     -grid$lambda_z,
                     decreasing=TRUE),     ]
#####################################################
# stage 1: cross validation with initial grid
#####################################################
tn <- cv.smtl(y = y,
              X = X,
              study = task,
              commonSupp = FALSE,
              grid = grid,
              nfolds = 5,
              multiTask = FALSE)

#############################################################################
# create second grid with optimal hyperparameters from stage 1
#############################################################################
# new sparsity grid
s <- tn$best$s
s <- c(s-2, s, s+2)
s <- s[s >= 1]

# ridge penalty grid
lambda_1 <- c(1e-7, 1e-5, 1e-3, 1e-1)

# new lambda_z grid
lambda_z <- tn$best$lambda_z
lambda_z <- sort( c( seq(1.5, 10, length = 5), seq(0.1, 1, length = 5) ) * lambda_z, decreasing = TRUE)
lambda_z <- lambda_z[lambda_z <= 10] # make sure this is below a threshold to prevent numerical issues

grid <- as.data.frame(  expand.grid( lambda_1, lambda_2, lambda_z, s) )
colnames(grid) <- c("lambda_1", "lambda_2", "lambda_z", "s")

# order correctly
grid <- grid[  order(-grid$s,
                     grid$lambda_1,
                     -grid$lambda_z,
```

```
                      decreasing=TRUE),     ]

#####################################################
# stage 2: cross validation with updated grid
#####################################################
tn <- cv.smtl(y = y,
              X = X,
              study = task,
              commonSupp = FALSE,
              grid = grid,
              nfolds = 5,
              multiTask = FALSE)
```

Fitting Whole Paths

Sometimes we may want to fit entire paths of solution for some fixed s. The algorithm is coded to fit whole paths to efficiently use warm starts to speed up the code substantially and identify better local minima. For that reason, if solutions for multiple values of tuning parameters, $(\lambda_1, \lambda_2, \lambda_z)$ are desired, it is far better to fit them jointly than to call `smtl()` repeatedly. The code is almost identical except that now one or more of the tuning parameter arguments $(\lambda_1, \lambda_2, \lambda_z)$ are a vector not a scalar. Notice that s is a scalar. We cannot put multiple values of s in because the path of solutions is generated independently for each value of s.

```
mod <- smtl(y = y,
            X = X,
            study = task,
            s = 5,
            commonSupp = FALSE,
            lambda_1 = c(0.1, 0.2, 0.3),
            lambda_z = c(0.01, 0.05, 0.1)
            )
```

Now the solutions in `mod$beta` is a $K \times p \times T$ array where T is the number of unique tuning value combinations (3 in the example above) used to fit the path of solutions.

Predictions on Whole Path Fits

Now we many solutions so if we want to make predictions we have to specify which one or else the package will issue a warning and return no predictions.

```
preds <- sMTL::predict(model = mod,
                       X = X,
                       lambda_1 = 0.1,
                       lambda_z = 0.01)
```

References

Gabriel Loewinger, Kayhan Behdin, Kenneth Kishida, Giovanni Parmigiani and Rahul Mazumder. Multi-Task Learning for Sparsity Pattern Heterogeneity: A Discrete Optimization Approach. arXiv: (2022).

Prasad Patil and Giovanni Parmigiani. Training replicable predictors in multiple studies. Proceedings of the National Academy of Sciences (2018).

Yu Zhang and Qiang Yang. A Survey on Multi-Task Learning. IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING (2021).