An Introduction to `multiview`

Daisy Yi Ding   Shuangning Li   Balasubramanian Narasimhan
Robert Tibshirani

March 31, 2023

Contents

Introduction . . . 1
Quick Start . . . 2
Linear Regression: `family = gaussian()` . . . 2
  Commonly-used function arguments . . . 2
  Predicting and plotting . . . 3
  Obtaining model coefficients . . . 4
  Cross-validation . . . 5
  Evaluating the contribution of data views in making predictions . . . 7
  Filtering variables . . . 8
  Checking progress . . . 8
Logistic Regression: `family = binomial()` . . . 9
Poisson Regression: `family = poisson()` . . . 11
References . . . 13

Introduction

`multiview` is a package that fits a supervised learning model called *cooperative learning* for multiple sets of features ("views"), as described in Ding et al. (2022). The method combines the usual squared error loss of predictions, or more generally, deviance loss, with an "agreement" penalty to encourage the predictions from different data views to agree. By varying the weight of the agreement penalty, we get a continuum of solutions that include the well-known early and late fusion approaches. Cooperative learning chooses the degree of agreement (or fusion) in an adaptive manner, using a validation set or cross-validation to estimate test set prediction error. In addition, the method combines the lasso penalty with the agreement penalty, yielding feature sparsity.

This vignette describes the basic usage of multiview in R. `multiview` is written based on the `glmnet` package and maintains the features from `glmnet`. The package includes functions for cross-validation, and making predictions and plots.

For two data views, consider feature matrices $X \in \mathcal{R}^{n\times p_x}$, $Z \in \mathcal{R}^{n\times p_z}$, and our target $y \in \mathcal{R}^n$. We assume that the columns of X and Z have been standardized, and y has mean 0 (hence we can omit the intercept below). For a fixed value of the hyperparameter $\rho \geq 0$, `multiview` finds $\beta_x \in \mathcal{R}^{p_x}$ and $\beta_Z \in \mathcal{R}^{p_z}$ that minimize:

$$\frac{1}{2}||y - X\theta_x - Z\theta_z||^2 + \frac{\rho}{2}||(X\theta_x - Z\theta_z)||^2 + \lambda\Big[(1-\alpha)(||\theta_x||_1 + ||\theta_z||_1) + \alpha(||\theta_x||_2^2/2 + ||\theta_z||_2^2/2)\Big].$$

Here the *agreement* penalty is controlled by ρ. When the weight on the agreement term ρ is set to 0, cooperative learning reduces to a form of early fusion: we simply concatenate the columns of different views and apply lasso or another regularized regression method. Moreover, we show in our paper that when ρ is set to 1, the solutions are the average of the marginal fits for X and Z, which is a simple form of late fusion.

The *elastic net* penalty is controlled by α, bridging the gap between lasso regression ($\alpha = 1$, the default) and ridge regression ($\alpha = 0$), and the tuning parameter λ controls the strength of the penalty. We can compute a regularization path of solutions indexed by λ.

For more than two views, this generalizes easily. Assume that we have `M` data matrices $X_1 \in \mathcal{R}^{n\times p_1}, X_2 \in \mathcal{R}^{n\times p_2}, \ldots, X_M \in \mathcal{R}^{n\times p_M}$, `multiview` solves the problem

$$\frac{1}{2}||y - \sum_{m=1}^{M} X_m \rho_m||^2 + \frac{\rho}{2} \sum_{m<m'} ||(X_m\rho_m - X_{m'}\rho_{m'})||^2 + \sum_{m=1}^{M} \lambda_m \Big[(1-\alpha)\|\rho_m\|_1 + \alpha||\rho_m||_2^2/2\Big].$$

`multiview` fits linear, logistic, and poisson models.

Quick Start

The purpose of this section is to give users a general sense of the package. We will briefly go over the main functions, basic operations and outputs. After this section, users may have a better idea of what functions are available, which ones to use, or at least where to seek help.

First, we load the `multiview` package:

```r
library(multiview)
```

Linear Regression: `family = gaussian()`

The default model used in the package is the Gaussian linear model or "least squares", which we will demonstrate in this section. We load a set of data created beforehand for illustration:

```r
quick_example <- readRDS(system.file("exdata", "example.RDS", package = "multiview"))
x <- quick_example$x
z <- quick_example$z
y <- quick_example$y
```

The command loads an input matrix `x` and a response vector `y` from this saved R data archive.

We fit the model using the most basic call to `multiview`.

```r
fit <- multiview(list(x,z), y, family=gaussian(), rho=0)
```

`fit` is an object of class `multiview` that contains all the relevant information of the fitted model for further use. Note that when `rho` $= 0$, cooperative learning reduces to a simple form of early fusion, where we simply concatenate the columns of X and Z and apply lasso. We do not encourage users to extract the components directly. Instead, various methods are provided for the object such as `plot`, `print`, `coef` and `predict` that enable us to execute those tasks more elegantly.

Commonly-used function arguments

`multiview` provides various arguments for users to customize the fit and maintains most of the arguments from `glmnet`: we introduce some commonly used arguments here. (For more information, type `?multiview` and `?glmnet`.)

- `rho` is for the agreement penalty. $\rho = 0$ is early fusion, and $\rho = 1$ gives a form of late fusion. We recommend trying a few values of `rho` including 0, 0.1, 0.25, 0.5, and 1 first; sometimes `rho` larger than 1 can also be helpful.
- `alpha` is for the elastic net mixing parameter α, with range $\alpha \in [0, 1]$. $\alpha = 1$ is lasso regression (default) and $\alpha = 0$ is ridge regression.
- `weights` is for the observation weights, default is 1 for each observation.

- `nlambda` is the number of λ values in the sequence (default is 100).
- `lambda` can be provided if the user wants to specify the lambda sequence, but typical usage is for the program to construct the lambda sequence on its own. When automatically generated, the λ sequence is determined by `lambda.max` and `lambda.min.ratio`. The latter is the ratio of the smallest value of the generated λ sequence (say `lambda.min`) to `lambda.max`. The program generates `nlambda` values linear on the log scale from `lambda.max` down to `lambda.min`. `lambda.max` is not user-specified but is computed from the input x and y: it is the smallest value for `lambda` such that all the coefficients are zero. For `alpha = 0` (ridge) `lambda.max` would be ∞: in this case we pick a value corresponding to a small value for `alpha` close to zero.)
- `standardize` is a logical flag for `x` variable standardization prior to fitting the model sequence. Default is `standardize = TRUE`. The coefficients are always returned on the original scale. If the variables are in the same units already, you might not want to standardize.

In addition, if there are missing values in the feature matrices: we recommend that you center the columns of each feature matrix, and then fill in the missing values with 0.

For example,

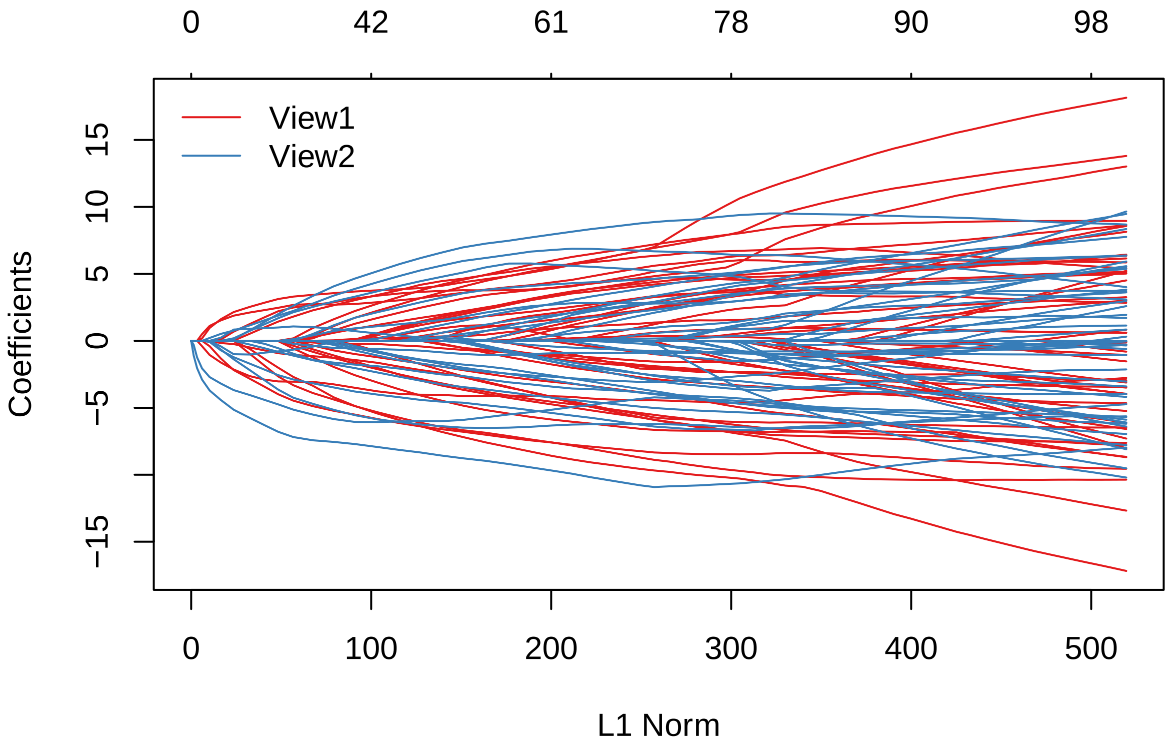
```r
x <- scale(x,TRUE,FALSE)
x[is.na(x)] <- 0
z <- scale(z,TRUE,FALSE)
z[is.na(z)] <- 0
```

Then you can run `multiview` in the usual way. It will exploit the assumed shared latent factors to make efficient use of the available data.

Predicting and plotting

We can visualize the coefficients by executing the `plot` method:

```r
plot(fit)
```



Each curve corresponds to a variable, with the color indicating the data view the variable comes from. It shows the path of its coefficient against the ℓ_1-norm of the whole coefficient vector as λ varies. The axis above indicates the number of nonzero coefficients at the current λ. Users may also wish to annotate the curves: this can be done by setting `label = TRUE` in the plot command.

A summary of the path at each step is displayed if we just enter the object name or use the `print` function:

```r
print(fit)
```

```
##
## Call:  multiview(x_list = list(x, z), y = y, rho = 0, family = gaussian())
##
##      Df  %Dev  Lambda
## 1     0  0.00 12.7200
## 2     2  1.17 11.5900
## 3     2  2.60 10.5600
## 4     5  4.49  9.6210
## 5     9  7.16  8.7670
## 6    13 10.72  7.9880
....
```

It shows from left to right the number of nonzero coefficients, the percent (of null) deviance explained (`%dev`) and the value of λ (`Lambda`).

We can make predictions at specific λ's with new input data:

```r
set.seed(1)
nx <- matrix(rnorm(5 * 50), 5, 50)
nz <- matrix(rnorm(5 * 50), 5, 50)
predict(fit, newx = list(nx, nz), s = c(0.1, 0.05))
```

```
##               s1         s2
## [1,]  47.76359  54.487042
## [2,] -92.00585 -96.943843
## [3,]  22.07899  19.903328
## [4,] -12.10992  -8.626519
## [5,]  36.18702  39.334068
```

Here `s` is for specifying the value(s) of λ at which to make predictions.

The `type` argument allows users to choose the type of prediction returned:

- "link" returns the fitted values.
- "response" gives the same output as "link" for the `gaussian()` family.
- "coefficients" returns the model coefficients.
- "nonzero" returns a list of the indices of the nonzero coefficients for each value of `s`.

Obtaining model coefficients

We can obtain the model coefficients at one or more λ's within the range of the sequence:

```r
coef(fit, s = 0.1)
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
##                        s1
## (Intercept)   2.2284771
## 1             3.0907310
## 2             7.0642659
## 3            -5.5563943
## 4             5.7200716
## 5            12.7545109
## 6             7.0789074
## 7            -2.8869078
....
```

Here `s` is for specifying the value(s) of λ at which to extract coefficients.

We can also obtain a ranked list of standardized model coefficients, where we rank the model coefficients after standardizing each coefficient by the standard deviation of the corresponding feature. Users have to provide the function with a specific value of λ at which to extract and rank coefficients.

```
coef_ordered(fit, s=0.1)
```

```
##      view view_col standardized_coef        coef
## 13  View1       13        16.5629085  16.5629085
## 25  View1       25       -15.4053272 -15.4053272
## 5   View1        5        12.7545109  12.7545109
## 11  View1       11        11.6615186  11.6615186
## 30  View1       30       -11.2198530 -11.2198530
## 36  View1       36       -10.3768851 -10.3768851
## 21  View1       21        -9.2486134  -9.2486134
## 62  View2       12         8.9924546   8.9924546
## 50  View1       50         8.9351513   8.9351513
....
```

The table shows from left to right the data view each coefficient comes from, the column index of the feature in the corresponding data view, the coefficient after being standardized by the standard deviation of the corresponding feature, and the original fitted coefficient. Here the features are ordered by the magnitude of their standardized coefficients, and we have truncated the printout for brevity.

Cross-validation

The function `multiview` returns a sequence of models for the users to choose from. In many cases, users may prefer the software to select one of them. Cross-validation is perhaps the simplest and most widely used method for that task. `cv.multiview` is the main function to do cross-validation here, along with various supporting methods such as plotting and prediction.

In addition to all the `glmnet` parameters, `cv.multiview` has its special parameters including `nfolds` (the number of folds), `foldid` (user-supplied folds), and `type.measure`(the loss used for cross-validation):

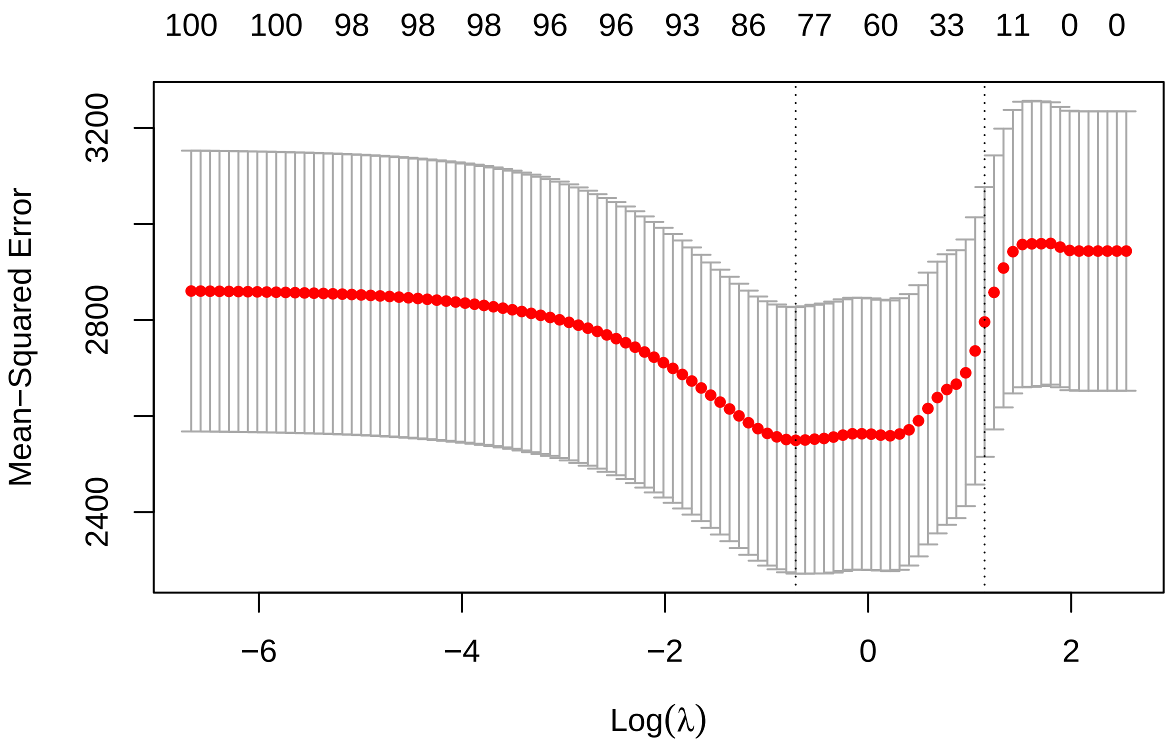
- "deviance" or "mse" for squared loss, and
- "mae" uses mean absolute error.

As an example,

```
cvfit <- cv.multiview(list(x,z), y, rho=0.1, family = gaussian(),
                      type.measure = "mse", nfolds = 20)
```

`cv.multiview` returns a `cv.multiview` object, a list with all the ingredients of the cross-validated fit. As with `multiview`, we do not encourage users to extract the components directly except for viewing the selected values of λ. The package provides well-designed functions for potential tasks. For example, we can plot the object:

```
plot(cvfit)
```



This plots the cross-validation curve (red dotted line) along with upper and lower standard deviation curves along the λ sequence (error bars). Two special values along the λ sequence are indicated by the vertical dotted lines. `lambda.min` is the value of λ that gives minimum mean cross-validated error, while `lambda.1se` is the value of λ that gives the most regularized model such that the cross-validated error is within one standard error of the minimum.

The `coef` and `predict` methods for `cv.multiview` objects are similar to those for a `glmnet` object, except that two special strings are also supported by `s` (the values of λ requested):

- "lambda.min": the λ at which the smallest MSE is achieved.
- "lambda.1se": the largest λ at which the MSE is within one standard error of the smallest MSE (default).

We can use the following code to get the value of `lambda.min` and the model coefficients at that value of λ:

```
cvfit$lambda.min
```

```
## [1] 0.4901186
```

```
coef(cvfit, s = "lambda.min")
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
##                      s1
## (Intercept)  2.228477053
## 1            2.107666801
## 2            1.855941335
## 3            .
## 4            3.235105747
## 5            6.030819004
## 6            0.640540777
## 7            0.085463336
....
```

To get the corresponding values at `lambda.1se`, simply replace `lambda.min` with `lambda.1se` above, or omit the `s` argument, since `lambda.1se` is the default.

Predictions can be made based on the fitted `cv.multiview` object as well. The code below gives predictions for the new input matrix `newx` at `lambda.min`:

```
predict(cvfit, newx = list(x[1:5,],z[1:5,]), s = "lambda.min")
```

```
##      lambda.min
## [1,] -42.167313
## [2,] -62.130628
## [3,] -31.903227
## [4,]  -6.040436
## [5,] -58.914454
```

Evaluating the contribution of data views in making predictions

With multiple data views, we can evaluate the contribution of each data view in making predictions using `view.contribution`. The function has two options. The first option is to set `force` to `NULL`, and then the evaluation of data view contribution will be benchmarked by the null model. Specifically, it evaluates the marginal contribution of each data view in improving predictions as compared to the null model, as well as the model using all data views with cooperative learning as compared to the null model. The function takes in a value of ρ for cooperative learning. It returns a table showing the percentage improvement in reducing error as compared to the null model made by each individual data view and by cooperative learning using all data views.

```
quick_example <- readRDS(system.file("exdata", "example_contribution.RDS",
                                     package = "multiview"))
rho <- 0.3
view.contribution(x_list=list(x=quick_example$x, z=quick_example$z), quick_example$y,
                  rho = rho, family = gaussian(), eval_data = 'train')
```

```
##                  view   metric percentage_improvement
## 1                null 49.14713                0.00000
## 2                   x 32.97143               32.91280
## 3                   z 35.48765               27.79304
## 4 cooperative (all) 28.15596               42.71088
```

The alternative option is to provide `force` with a list of data views as the baseline mode. Then the function evaluates the marginal contribution of each additional data view on top of this benchmarking list of views using cooperative learning. The function returns a table showing the percentage improvement in reducing error as compared to the benchmarking model made by each additional data view.

```
view.contribution(x_list=list(x=quick_example$x, z=quick_example$z, w=quick_example$w),
                  quick_example$y, rho = rho, eval_data = 'train', family = gaussian(),
                  force=list(x=quick_example$x))
```

```
##       view   metric percentage_improvement
## 1 baseline 31.56604                0.00000
## 2        z 27.10145               14.14363
## 3        w 25.87451               18.03054
```

Here by setting `eval_data` to be `"train"`, we are evaluating the contribution of each data view based on the cross-validation error on the training set We can also evaluate the contribution based on the test set performance by setting `eval_data` to be `"test"`. In this case, we need to provide the function with the test data for evaluation.

```
view.contribution(x_list = list(x = quick_example$x, z = quick_example$z),
                  quick_example$y, rho = rho,
                  x_list_test = list(x = quick_example$test_X, z = quick_example$test_Z),
                  test_y = quick_example$test_y, family = gaussian(), eval_data = 'test')
```

```
##                  view   metric percentage_improvement
## 1                null 67.13278                0.00000
## 2                   x 46.17828               31.21352
## 3                   z 55.06684               17.97325
## 4 cooperative (all) 44.33914               33.95307
```

Filtering variables

The `exclude` argument to `multiview` accepts a filtering function, which indicates which variables should be excluded from the fit, i.e. get zeros for their coefficients. The idea is that variables can be filtered based on some of their properties (e.g. too sparse or with small variance) before any models are fit. Of course, one could simply subset these out of x, but sometimes exclude is more useful, since it returns a full vector of coefficients, just with the excluded ones set to zero. When performing cross-validation (CV), this filtering is done separately inside each fold of `cv.multiview`.

To give a motivating example: in genomics, where the X matrix can be very wide, we often filter features based on variance, i.e. filter the genes with no variance or low variance in their expression across samples. Here is a function that efficiently computes the column-wise variance for a wide matrix, followed by a filter function that uses it.

```
uvar <- function(x, means = FALSE) {
  # if means = TRUE, the means and variances are returned,
  # otherwise just the variances
  m <- colMeans(x)
  n <- nrow(x)
  x <- x - outer(rep(1,n),m)
  v <- colSums(x^2) / (n - 1)
  if (means) list(mean = m, var = v) else v
}

uvar_multiple <- function(x_list) lapply(x_list, function(x) uvar(x))

vfilter <- function(x_list, q = 0.3, ...) {
    v <- uvar_multiple(x_list)
    lapply(v, function(x) which(x < quantile(x, q)))
}
```

Here, the filter function `vfilter()` will exclude the fraction q of the variables with the lowest variance. This function gets invoked inside the call to `multiview`, and uses the supplied `x_list` to generate the indices. Note that some of the arguments in the filter function can be omitted, but the `...` must always be there.

```
set.seed(1)
x <- matrix(rnorm(100 * 20), 100, 20)
z <- matrix(rnorm(100 * 20), 100, 20)
y <- rnorm(100)
fit.filter <- multiview(list(x,z), y, exclude = vfilter)
```

The nice thing with this approach is that `cv.multiview` does the right thing: it applies the filter function separately to the feature matrices of each training fold, hence accounting for any bias that may be incurred by the filtering.

```
cvfit.filter <- cv.multiview(list(x,z), y, exclude = vfilter)
```

Checking progress

Ever run a job on a big dataset, and wonder how long it will take? `multiview` and `cv.multiview` come equipped with a progress bar, which can by displayed by passing trace.it = TRUE to these functions.

```r
fit <- multiview(list(x,z), y, trace.it = TRUE)
```

```
##
|===========                                           | 33%
```

This display changes in place as the fit is produced. The progress bar is also very helpful with `cv.multiview` :

```r
fit <- cv.multiview(list(x,z), y, trace.it = TRUE)
```

```
##
|==============================================| 100%

Fold: 1/10

|==============================================| 100%

Fold: 2/10

|==============================================| 100%

Fold: 3/10

|==============================                  |  70%
```

If the user wants `multiview` and `cv.multiview` to always print the progress bar, this can be achieved (for a session) via a call to `multiview.control` with the `itrace` argument:

```r
multiview.control(itrace = 1)
```

To reset it, one makes a similar call and sets `itrace = 0`.

With the tools introduced so far, users are able to fit the entire elastic net family, including lasso and ridge regression, using squared-error loss.

There are also additional features of the `multiview` package that are inherited from the `glmnet` package. The `glmnet` vignette "An Introduction to `glmnet`" provides more detailed explanations of the function parameters and features. This vignette is also written based on the vignette for the `glmnet` package.

Logistic Regression: `family = binomial()`

Logistic regression is a widely-used model when the response is binary. As a generalized linear model, logistic regression has the following objective to be minimized:

$$\ell(X\theta_x + Z\theta_z, y) + \frac{\rho}{2}||(X\theta_x - Z\theta_z)||^2 + \lambda\Big[(1-\alpha)(||\theta_x||_1 + ||\theta_z||_1) + \alpha(||\theta_x||_2^2/2 + ||\theta_z||_2^2/2)\Big],$$

where ℓ is the negative log-likelihood (NLL) of the data.

We make the usual quadratic approximation to the objective, reducing the minimization problem to a weighted least squares (WLS) problem. This leads to an iteratively reweighted least squares (IRLS) algorithm consisting of an outer and inner loop.

For illustration purposes, we load the pre-generated input matrices `X` and `Z` and the response vector `y` from the data file, where y is a binary vector.

```r
example_binomial <- readRDS(system.file("exdata", "example_binomial.RDS",
                                        package = "multiview"))
x <- example_binomial$x
z <- example_binomial$z
y <- example_binomial$by
```

Other optional arguments of `multiview` for binomial regression are almost same as those for Gaussian family. Don't forget to set `family` option to `binomial()`. (For this example, we also need to increase the number of IRLS iterations via `multiview.control()` to ensure convergence, and we reset it after the call.)

```
multiview.control(mxitnr = 100)
fit <- multiview(list(x=x,z=z), y, family = binomial(), rho = 0)
multiview.control(factory = TRUE)
```

As before, we can print and plot the fitted object, extract the coefficients at specific λ's and also make predictions. Prediction is a little different for `family = binomial()`, mainly in the function argument `type`:

- "link" gives the linear predictors.
- "response" gives the fitted probabilities.
- "class" produces the class label corresponding to the maximum probability.

In the following example, we make prediction of the class labels at $\lambda = 0.05, 0.01$.

```
predict(fit, newx = list(x[5:10,],z[5:10,]), type = "class", s = c(0.05, 0.01))
```

```
##      s1  s2 
## [1,] "0" "0"
## [2,] "1" "1"
## [3,] "0" "0"
## [4,] "1" "1"
## [5,] "0" "0"
## [6,] "0" "0"
```

For logistic regression, `cv.multiview` has similar arguments and usage as Gaussian.

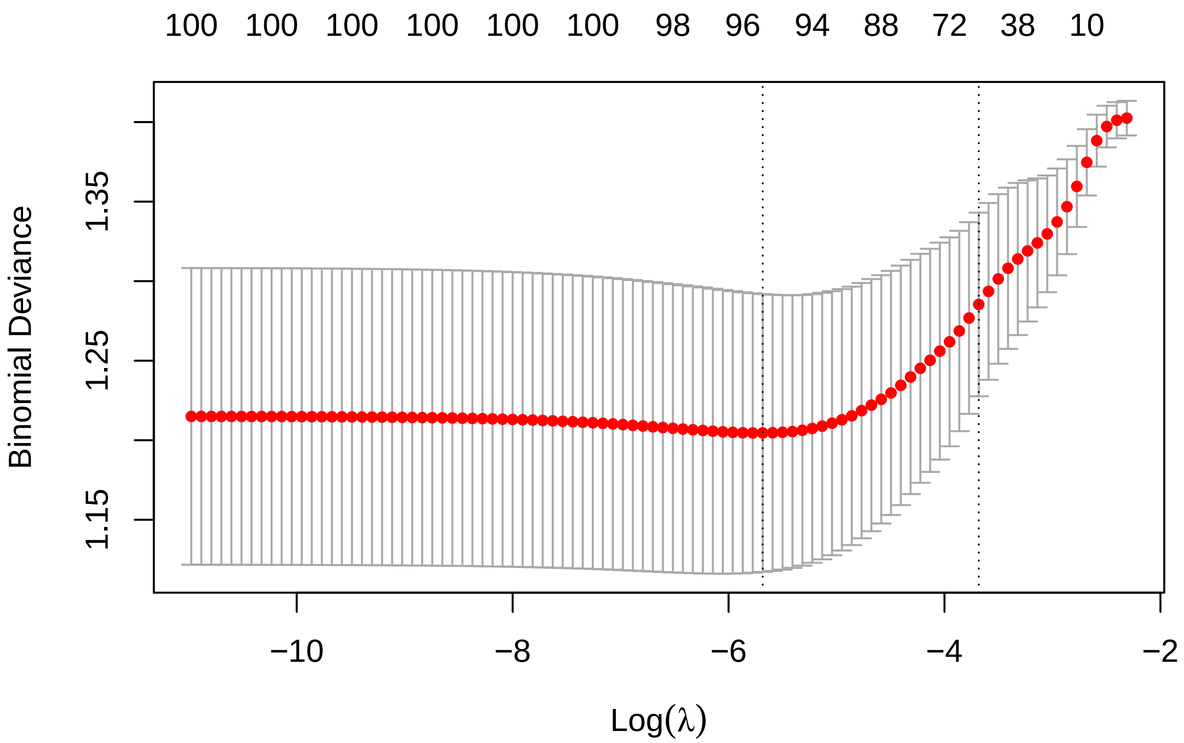
- "mse" uses squared loss.
- "deviance" uses actual deviance.
- "mae" uses mean absolute error.

For example, the code below uses "deviance" as the criterion for 10-fold cross-validation:

```
cvfit <- cv.multiview(list(x = x, z = z), y, family = binomial(),
                      type.measure = "deviance", rho = 0.5)
```

As before, we can plot the object and show the optimal values of λ.

```
plot(cvfit)
```



`coef` and `predict` for the `cv.multiview` object for `family = binomial()` are similar to the Gaussian case and we omit the details.

Poisson Regression: `family = poisson()`

Poisson regression is used to model count data under the assumption of Poisson error, or otherwise non-negative data where the mean and variance are proportional. Like the Gaussian and binomial models, the Poisson distribution is a member of the exponential family of distributions. As before, we minimize the following objective:

$$\ell(X\theta_x + Z\theta_z, y) + \frac{\rho}{2}||(X\theta_x - Z\theta_z)||^2 + \lambda\Big[(1-\alpha)(||\theta_x||_1 + ||\theta_z||_1) + \alpha(||\theta_x||_2^2/2 + ||\theta_z||_2^2/2)\Big],$$

where ℓ is the negative log-likelihood (NLL) of the data.

First, we load a pre-generated set of Poisson data:

```
example_poisson <- readRDS(system.file("exdata", "example_poisson.RDS",
                                       package = "multiview"))
x <- example_poisson$x
z <- example_poisson$z
y <- example_poisson$py
```

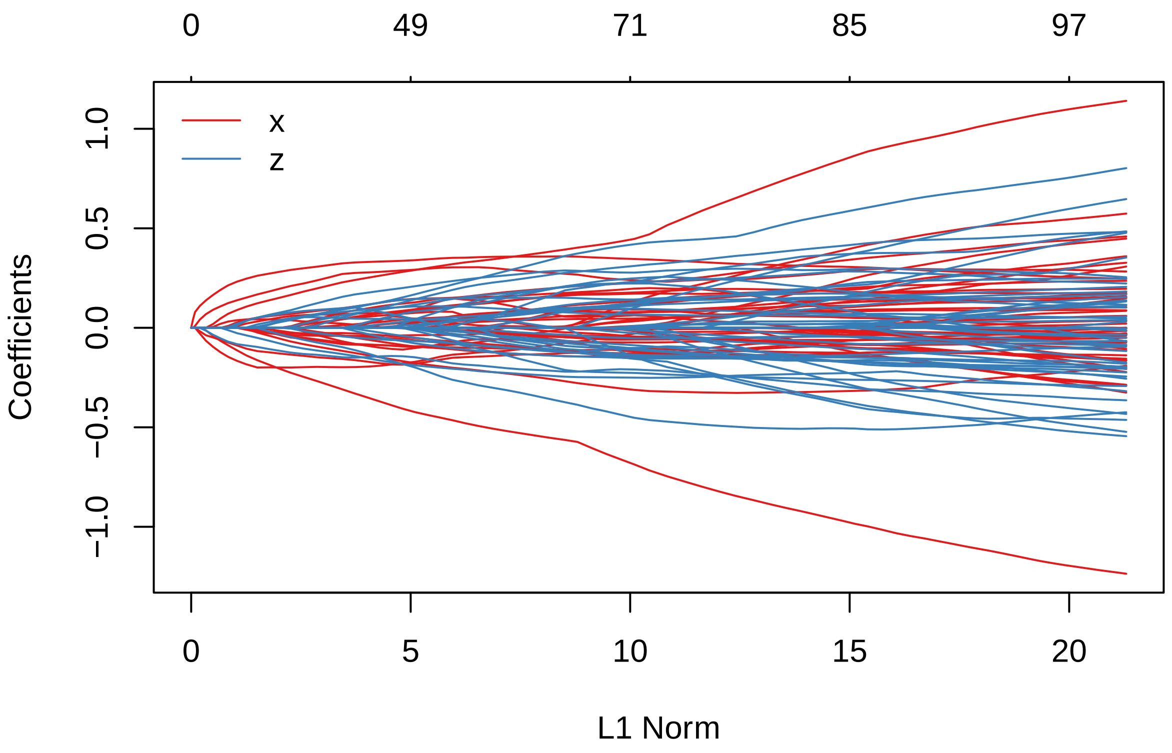
We apply the function `multiview` with `family = poisson()`:

```
fit <- multiview(list(x=x, z=z), y, family = poisson(), rho = 0)
```

The optional input arguments of `multiview` for `"poisson"` family are similar to those for other families.

Again, we plot the coefficients to have a first sense of the result.

```
plot(fit)
```



As before, we can extract the coefficients and make predictions at certain λ's using `coef` and `predict` respectively. The optional input arguments are similar to those for other families. For the `predict` method, the argument `type` has the same meaning as that for `family = binomial()`, except that "response" gives the fitted mean (rather than fitted probabilities in the binomial case). For example, we can do the following:

```
coef(fit, s = 1)
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
##                      s1
## (Intercept)  1.329608205
## 1            .
## 2            .
## 3            .
## 4            0.118781163
....
```

```
predict(fit, newx = list(x[1:5,],z[1:5,]), type = "response", s = c(0.1,1))
```

```
##              s1       s2
## [1,]  3.4548104 4.916919
## [2,]  0.8168098 3.680477
## [3,]  2.5816121 7.247710
## [4,]  2.1124338 4.578768
## [5,] 16.2348442 6.642053
```

We may also use cross-validation to find the optimal λ's and thus make inferences. However, the direct has convergence issues with the default control parameter settings as can be seen below.

```
cvfit <- cv.multiview(list(x,z), y, family = poisson(), rho = 0.1)
```

```
## Error: inner loop 3; cannot correct step size
```

This is because the underlying IRLS algorithm requires more than the default number of iterations to converge. This can be fixed by setting the appropriate `multiview.control()` parameter (see help on `multiview.control`).

To allow more iterations:

```r
multiview.control(mxitnr = 100)
cvfit <- cv.multiview(list(x,z), y, family = poisson(), rho = 0.1)
```

To set the parameters back to the factory default:

```r
multiview.control(factory = TRUE)
```

References

Ding, Daisy Yi, Shuangning Li, Balasubramanian Narasimhan, and Robert Tibshirani. 2022. "Cooperative Learning for Multi-View Analysis." *arXiv Preprint arXiv:2112.12337*.