bigSurvSGD: Big Survival Data Analysis via Stochastic Gradient Descent

Aliasghar (Arash) Tarkhan and Noah Simon

2020-08-07

Introduction

We give a short tutorial on using bigSurvSGD package. This package fits Cox Model via stochastic gradient descent (SGD). This implementation avoids computational instability of the standard Cox Model when datasets are large. Furthermore, it scales up with very large datasets that do not fit the memory. It also handles large sparse datasets using the Proximal stochastic gradient descent algorithm.

Installing and loading package

Intallating package

Like many other R packages, the simplest way to obtain `bigSurvSGD` is to install it directly from CRAN:

```
install.packages("bigSurvSGD", repos = "http://cran.us.r-project.org")
```

Users may change the `repos` options depending on their locations and preferences. Alternatively, users can install the package using Github repository:

```
library(devtools)
install_github("atarkhan/bigSurvSGD")
```

Loading package

We load the packages `bigSurvSGD` and `survival` as folowing

```
library(bigSurvSGD)
#> Loading required package: foreach
#> Loading required package: parallel
library(survival)
```

Fitting Cox model

Loading data

We first load dataset `survData` included in package:

```r
data("survData")
```

Now we use `bigSurvSGD` to estimate coefficients as following

```r
fitBigSurvSGD <- bigSurvSGD(formula=Surv(time, status)~.,data=survData,
                            parallel.flag=TRUE, num.cores = 2)
fitBigSurvSGD
#> Call:
#> bigSurvSGD(formula = Surv(time, status) ~ ., data = survData,
#>     parallel.flag = TRUE, num.cores = 2)
#>
#> Coefficients (log hazards ratio)
#>     estimate lower.0.95.CI upper.0.95.CI        z   p-value'
#> X1     0.998        0.9867        1.0093 173.6337 <2e-16 ***
#> X2    1.0466        1.0358        1.0574 190.3402 <2e-16 ***
#> X3    0.9191        0.9096        0.9287 189.0311 <2e-16 ***
#> X4     1.058        1.0452        1.0708 161.9559 <2e-16 ***
#> X5    0.9556        0.9451        0.9661 179.0998 <2e-16 ***
#> X6    1.0337        1.0238        1.0435 205.9408 <2e-16 ***
#> X7    1.0223        1.0104        1.0342 168.6545 <2e-16 ***
#> X8    0.9909        0.9792        1.0027 165.6357 <2e-16 ***
#> X9    0.9254        0.9157        0.9352  185.834 <2e-16 ***
#> X10    0.916        0.9054        0.9266 169.2688 <2e-16 ***
#>
#> Coefficients (hazards ratio)
#>     estimate lower.0.95.CI upper.0.95.CI        z    p-value
#> X1    2.7128        2.6824        2.7435 173.6337 <2e-16 ***
#> X2    2.8478        2.8173        2.8787 190.3402 <2e-16 ***
#> X3    2.5071        2.4833        2.5312 189.0311 <2e-16 ***
#> X4    2.8805        2.8438        2.9177 161.9559 <2e-16 ***
#> X5    2.6003        2.5732        2.6276 179.0998 <2e-16 ***
#> X6    2.8113        2.7838        2.8391 205.9408 <2e-16 ***
#> X7    2.7795        2.7467        2.8128 168.6545 <2e-16 ***
#> X8    2.6937        2.6623        2.7255 165.6357 <2e-16 ***
#> X9    2.5229        2.4984        2.5477  185.834 <2e-16 ***
#> X10   2.4992        2.4728        2.5259 169.2688 <2e-16 ***
```

For comparison, we can use standard Cox Model `coxph` to estimate coefficients as following

```r
fitCox <- coxph(formula=Surv(time, status)~.,data=survData)
fitCox
#> Call:
#> coxph(formula = Surv(time, status) ~ ., data = survData)
#>
#>        coef exp(coef) se(coef)     z      p
#> X1  0.98008   2.66467  0.04374 22.41 <2e-16
#> X2  1.01072   2.74757  0.04430 22.82 <2e-16
#> X3  0.90612   2.47471  0.04297 21.09 <2e-16
#> X4  1.01932   2.77131  0.04460 22.85 <2e-16
#> X5  0.95135   2.58920  0.04401 21.62 <2e-16
#> X6  1.00906   2.74303  0.04490 22.48 <2e-16
#> X7  0.98892   2.68834  0.04421 22.37 <2e-16
#> X8  0.97359   2.64743  0.04449 21.88 <2e-16
#> X9  0.91224   2.48990  0.04505 20.25 <2e-16
#> X10 0.90848   2.48054  0.04359 20.84 <2e-16
```

```
#>
#> Likelihood ratio test=1748  on 10 df, p=< 2.2e-16
#> n= 1000, number of events= 823
```

Users need to use `formula` to specify `time-to-event` and `status` variables if they were named deifferently from `time` and `status`. User may also need to specify a subset of features they need to include in the model. For example, suppose that variable `t` and `s` represent `time-to-event` and `status` variables, and a user wants to only include features $X1$, $X2$, and $X3$:

```
colnames(survData)[c(1,2)] <- c("t", "s")
fitBigSurvSGD <- bigSurvSGD(formula=Surv(time=t, status=s)~X1+X2+X3, data=survData,
                            parallel.flag=TRUE, num.cores=2)
fitBigSurvSGD
#> Call:
#> bigSurvSGD(formula = Surv(time = t, status = s) ~ X1 + X2 + X3,
#>     data = survData, parallel.flag = TRUE, num.cores = 2)
#>
#> Coefficients (log hazards ratio)
#>    estimate lower.0.95.CI upper.0.95.CI        z   p-value'
#> X1   0.3848        0.3783        0.3912 116.4805 <2e-16 ***
#> X2   0.3427         0.337        0.3483 118.9154 <2e-16 ***
#> X3   0.3602        0.3545        0.3659 124.6117 <2e-16 ***
#>
#> Coefficients (hazards ratio)
#>    estimate lower.0.95.CI upper.0.95.CI        z    p-value
#> X1   1.4693        1.4598        1.4788 116.4805 <2e-16 ***
#> X2   1.4087        1.4008        1.4167 118.9154 <2e-16 ***
#> X3   1.4336        1.4255        1.4418 124.6117 <2e-16 ***
```

For comparison, we can estimate coefficents using standard Cox Model `coxph`:

```
fitCox <- coxph(formula=Surv(t, s)~X1+X2+X3, data=survData)
fitCox
#> Call:
#> coxph(formula = Surv(t, s) ~ X1 + X2 + X3, data = survData)
#>
#>       coef exp(coef) se(coef)      z      p
#> X1 0.37111   1.44934  0.03628 10.229 <2e-16
#> X2 0.32047   1.37778  0.03542  9.049 <2e-16
#> X3 0.34446   1.41123  0.03655  9.424 <2e-16
#>
#> Likelihood ratio test=252.2  on 3 df, p=< 2.2e-16
#> n= 1000, number of events= 823
```

Note that the current version only supports numerical variables. Users need to convert non-numerical valriables into numerical variables beforehand.

Fitting large dataset of the hard drive

Package `bigSurvSGD` can handle large datasets that do not fit the memory. For an example, suppose that dataset `bigSurvData` is very big and we saved it with path `/home/arsh/bigSurvData.csv`:

```
data("survData")
write.csv(survData, file = "bigSurvData.csv", row.names = F)
```

Suppose that memory can only handle up to 100 rows of `bigSurdData`. In practice, this number is a maximum number of rows for which R can run without "lack of memory" error. This number would be big if there few features (columns) in dataset and it wold be small if there are many featurs. Note that number of rows per chunk must be at least equal to strata size specified as `strata.size` in this package. For our example, we ask `bigSurvSGD` to use `bigmemory` package (by defining `bigmemory.flag=T`) to read data chunk-by-chunk off the memory with chunk size of 100.

```
fitBigSurvSGD <- bigSurvSGD(formula=Surv(time=time, status=status)~.,
                           data="bigSurvData.csv",
                           bigmemory.flag = T, num.rows.chunk = 100)
#> Warning in read.big.matrix(filename = data, sep = ",", skip = 0, header = T):
#> Because type was not specified, we chose double based on the first line of data.
fitBigSurvSGD
#> Call:
#> bigSurvSGD(formula = Surv(time = time, status = status) ~ .,
#>     data = "bigSurvData.csv", bigmemory.flag = T, num.rows.chunk = 100)
#>
#> Coefficients (log hazards ratio)
#>     estimate lower.0.95.CI upper.0.95.CI        z   p-value'
#> X1      0.997        0.9847        1.0092 159.4618 <2e-16 ***
#> X2     1.0451        1.0329        1.0573 168.2331 <2e-16 ***
#> X3     0.9161        0.9049        0.9273 160.6478 <2e-16 ***
#> X4     1.0592        1.0469        1.0714 169.5921 <2e-16 ***
#> X5     0.9563        0.9459        0.9667 180.7857 <2e-16 ***
#> X6      1.027         1.013         1.041 143.6927 <2e-16 ***
#> X7     1.0194        1.0052        1.0336 140.6698 <2e-16 ***
#> X8     0.9896        0.9764        1.0027  147.505 <2e-16 ***
#> X9     0.9173        0.9036         0.931 131.5349 <2e-16 ***
#> X10    0.9148        0.9037         0.926 161.0244 <2e-16 ***
#>
#> Coefficients (hazards ratio)
#>     estimate lower.0.95.CI upper.0.95.CI        z    p-value
#> X1       2.71         2.677        2.7435 159.4618 <2e-16 ***
#> X2     2.8437        2.8092        2.8785 168.2331 <2e-16 ***
#> X3     2.4996        2.4718        2.5277 160.6478 <2e-16 ***
#> X4     2.8839        2.8488        2.9195 169.5921 <2e-16 ***
#> X5      2.602        2.5752        2.6292 180.7857 <2e-16 ***
#> X6     2.7927        2.7538        2.8321 143.6927 <2e-16 ***
#> X7     2.7716        2.7325        2.8113 140.6698 <2e-16 ***
#> X8       2.69        2.6549        2.7257  147.505 <2e-16 ***
#> X9     2.5025        2.4685         2.537 131.5349 <2e-16 ***
#> X10    2.4963        2.4686        2.5243 161.0244 <2e-16 ***
```

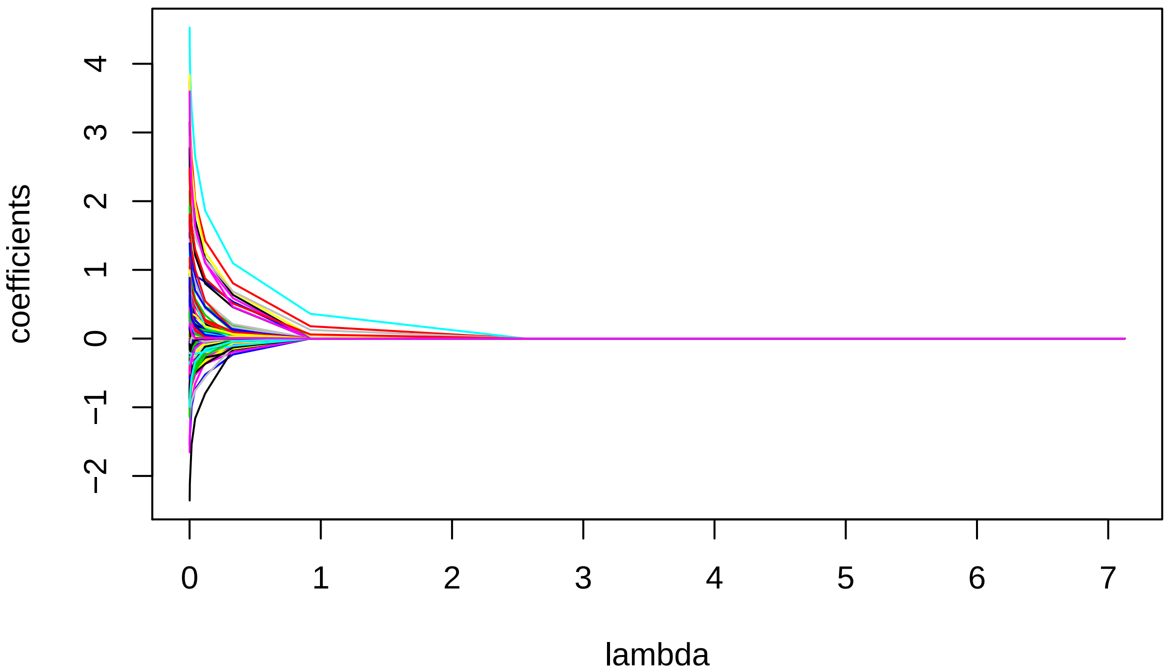
Note that the current package only supports a path to a `.csv` (comma separated values) files.

Fitting Cox Model with sparse data

Now we consider dataset `sparseSurvData` with number of features (columns) larger than number of observations (rows). `bigSurvSGD` package handles sparse datasets. The following fits a regularized Cox Model with the elastic net penalty where the penalty coefficients of l_1 and l_2 norms are $alpha * lambda = 0.09$ and $(1 - alpha) * lambda = 0.01$, respectively.

```
data("sparseSurvData")
fitBigSurvSGD <- bigSurvSGD(formula=Surv(time=time, status=status)~.,data=sparseSurvData,
```

```r
                              alpha = 0.9, lambda = NULL, nlambda = 10)
plot(fitBigSurvSGD)
```




```r
fitBigSurvSGD$coef[1:10,1:10]
#>     lambda=7.1261620161 lambda=2.5610098424 lambda=0.9203792165
#> X1                    0                   0          0.00000000
#> X2                    0                   0          0.17905937
#> X3                    0                   0          0.04639310
#> X4                    0                   0          0.00000000
#> X5                    0                   0          0.35997296
#> X6                    0                   0          0.00000000
#> X7                    0                   0          0.04558453
#> X8                    0                   0          0.12623434
#> X9                    0                   0          0.00000000
#> X10                   0                   0          0.05967845
#>     lambda=0.3307671404 lambda=0.1188715469 lambda=0.0427202189
#> X1            0.6358768           1.1694975           1.7204579
#> X2            0.8044756           1.4223458           2.0100530
#> X3            0.5259262           0.8362096           1.2067481
#> X4            0.5357695           0.8247518           0.9171526
#> X5            1.0957344           1.8603669           2.6458376
#> X6            0.5889533           1.0987510           1.6000542
#> X7            0.6779663           1.2524746           1.9565855
#> X8            0.6846754           1.1486178           1.4791119
#> X9            0.4559595           0.8005427           1.2142223
#> X10           0.5112724           0.8737714           1.2967519
#>     lambda=0.0153528507 lambda=0.0055175284 lambda=0.0019828969
#> X1             2.195030            2.490466            2.795364
#> X2             2.641772            3.050495            3.473942
#> X3             1.560494            1.754157            1.889114
#> X4             1.150724            1.298853            1.466634
#> X5             3.337975            3.772155            4.110122
```

```
#> X6               2.067125          2.271555          2.663205
#> X7               2.469944          2.975209          3.432830
#> X8               2.009799          2.345117          2.655510
#> X9               1.645401          2.005077          2.313680
#> X10              1.630948          2.006114          2.293622
#>      lambda=0.0007126162
#> X1               3.153188
#> X2               3.744907
#> X3               2.142731
#> X4               1.539430
#> X5               4.526998
#> X6               2.881238
#> X7               3.836247
#> X8               2.948863
#> X9               2.770772
#> X10              2.476748
```

Note that if the dataset is very large, users may want to specify maximum number of rows per chunk specified by `max.rows.chunk` to read data chunk-by-chunk off the memory. This avoids lack-of-memory issue, as we discussed in the previous section.