The SVA package for removing batch effects and other unwanted variation in high-throughput experiments

Jeffrey Leek[1]*, W. Evan Johnson[2], Andrew Jaffe[1], Hilary Parker[1], John Storey[3]
[1]Johns Hopkins Bloomberg School of Public Health
[2]Boston University
[3]Princeton University
email: jleek@jhsph.edu

Modified: October 24, 2011 Compiled: October 17, 2016

Contents

1 Overview

The *sva* package contains functions for removing batch effects and other unwanted variation in high-throughput experiments. Specifically, the *sva* package contains functions for identifying and building surrogate variables for high-dimensional data sets. Surrogate variables are covariates constructed directly from high-dimensional data (like gene expression/RNA sequencing/methylation/brain imaging data) that can be used in subsequent analyses to adjust for unknown, unmodeled, or latent sources of noise.

The *sva* package can be used to remove artifacts in two ways: (1) identifying and estimating surrogate variables for unknown sources of variation in high-throughput experiments and (2) directly removing known batch effects using ComBat [**?**].

Leek et. al (2010) define batch effects as follows:

> Batch effects are sub-groups of measurements that have qualitatively different behaviour across conditions and are unrelated to the biological or scientific variables in a study. For example, batch effects may occur if a subset of experiments was run on Monday and another set on Tuesday, if two technicians were responsible for different subsets of the experiments, or if two different lots of reagents, chips or instruments were used.

The *sva* package includes the popular ComBat [**?**] function for directly modeling batch effects when they are known. There are also potentially a large number of environmental and biological variables that are unmeasured and may have a large impact on measurements from high-throughput biological experiments. For these cases the `sva` function may be more appropriate for removing these artifacts. It

is also possible to use the `sva` function with the `ComBat` function to remove both known batch effects and other potential latent sources of variation. Removing batch effects and using surrogate variables in differential expression analysis have been shown to reduce dependence, stabilize error rate estimates, and improve reproducibility (see [**?**, **?**, **?**] for more detailed information).

This document provides a tutorial for using the *sva* package. The tutorial includes information on (1) how to estimate the number of latent sources of variation, (2) how to apply the*sva* package to estimate latent variables such as batch effects, (3) how to directly remove known batch effects using the `ComBat` function, (4) how to perform differential expression analysis using surrogate variables either directly or with the*limma* package, and (4) how to apply "frozen" `sva` to improve prediction and clustering.

As with any R package, detailed information on functions, along with their arguments and values, can be obtained in the help files. For instance, to view the help file for the function `sva` within R, type `?sva`. The analyses performed in this experiment are based on gene expression measurements from a bladder cancer study [**?**]. The data can be loaded from the *bladderbatch* data package. The relevant packages for the Vignette can be loaded with the code:

```
> library(sva)
> library(bladderbatch)
> data(bladderdata)
> library(pamr)
> library(limma)
```

2 Setting up the data

The first step in using the *sva* package is to properly format the data and create appropriate model matrices. The data should be a matrix with features (genes, transcripts, voxels) in the rows and samples in the columns. This is the typical genes by samples matrix found in gene expression analyses. The *sva* package assumes there are two types of variables that are being considered: (1) adjustment variables and (2) variables of interest. For example, in a gene expression study the variable of interest might an indicator of cancer versus control. The adjustment variables could be the age of the patients, the sex of the patients, and a variable like the date the arrays were processed.

Two model matrices must be made: the "full model" and the "null model". The null model is a model matrix that includes terms for all of the adjustment variables but not the variables of interest. The full model includes terms for both the adjustment variables and the variables of interest. The assumption is that you will be trying to analyze the association between the variables of interest and gene expression, adjusting for the adjustment variables. The model matrices can be created using the `model.matrix`.

3 Setting up the data from an ExpressionSet

For the bladder cancer study, the variable of interest is cancer status. To begin we will assume no adjustment variables. The bladder data are stored in an expression set - a Bioconductor object used for

storing gene expression data. The variables are stored in the phenotype data slot and can be obtained as follows:

```
> pheno = pData(bladderEset)
```

The expression data can be obtained from the expression slot of the expression set.

```
> edata = exprs(bladderEset)
```

Next we create the full model matrix - including both the adjustment variables and the variable of interest (cancer status). In this case we only have the variable of interest. Since cancer status has multiple levels, we treat it as a factor variable.

```
> mod = model.matrix(~as.factor(cancer), data=pheno)
```

The null model contains only the adjustment variables. Since we are not adjusting for any other variables in this analysis, only an intercept is included in the model.

```
> mod0 = model.matrix(~1,data=pheno)
```

Now that the model matrices have been created, we can apply the `sva` function to estimate batch and other artifacts.

4 Applying the `sva` function to estimate batch and other artifacts

The `sva` function performs two different steps. First it identifies the number of latent factors that need to be estimated. If the `sva` function is called without the `n.sv` argument specified, the number of factors will be estimated for you. The number of factors can also be estimated using the `num.sv`.

```
> n.sv = num.sv(edata,mod,method="leek")
> n.sv
```

```
[1] 2
```

Next we apply the `sva` function to estimate the surrogate variables:

```
> svobj = sva(edata,mod,mod0,n.sv=n.sv)
```

```
Number of significant surrogate variables is:  2
Iteration (out of 5 ):1  2  3  4  5
```

The `sva` function returns a list with four components, `sv`, `pprob.gam`, `pprob.b`, `n.sv`. `sv` is a matrix whose columns correspond to the estimated surrogate variables. They can be used in downstream analyses as described below. `pprob.gam` is the posterior probability that each gene is associated with one or more latent variables [**?**]. `pprob.b` is the posterior probability that each gene is associated with the variables of interest [**?**]. `n.sv` is the number of surrogate variables estimated by the `sva`.

5 Adjusting for surrogate variables using the `f.pvalue` function

The `f.pvalue` function can be used to calculate parametric F-test p-values for each row of a data matrix. In the case of the bladder study, this would correspond to calculating a parametric F-test p-value for each of the 22,283 rows of the matrix. The F-test compares the models `mod` and `mod0`. They must be nested models, so all of the variables in `mod0` must appear in `mod`. First we can calculate the F-test p-values for differential expression with respect to cancer status, without adjusting for surrogate variables, adjust them for multiple testing, and calculate the number that are significant with a Q-value less than 0.05.

```
> pValues = f.pvalue(edata,mod,mod0)
> qValues = p.adjust(pValues,method="BH")
```

Note that nearly 70% of the genes are strongly differentially expressed at an FDR of less than 5% between groups. This number seems artificially high, even for a strong phenotype like cancer. Now we can perform the same analysis, but adjusting for surrogate variables. The first step is to include the surrogate variables in both the null and full models. The reason is that we want to adjust for the surrogate variables, so we treat them as adjustment variables that must be included in both models. Then P-values and Q-values can be computed as before.

```
> modSv = cbind(mod,svobj$sv)
> mod0Sv = cbind(mod0,svobj$sv)
> pValuesSv = f.pvalue(edata,modSv,mod0Sv)
> qValuesSv = p.adjust(pValuesSv,method="BH")
```

Now these are the adjusted P-values and Q-values accounting for surrogate variables.

6 Adjusting for surrogate variables using the limma package

The *limma* package is one of the most commonly used packages for differential expression analysis. The *sva* package can easily be used in conjunction with the *limma* package to perform adjusted differential expression analysis. The first step in this process is to fit the linear model with the surrogate variables included.

```
> fit = lmFit(edata,modSv)
```

From here, you can use the *limma* functions to perform the usual analyses. As an example, suppose we wanted to calculate differential expression with respect to cancer. To do that we first compute the contrasts between the pairs of cancer/normal terms. We do not include the surrogate variables in the contrasts, since they are only being used to adjust the analysis.

```
> contrast.matrix <- cbind("C1"=c(-1,1,0,rep(0,svobj$n.sv)),"C2"=c(0,-1,1,rep(0,svobj$n.s
> fitContrasts = contrasts.fit(fit,contrast.matrix)
```

The next step is to calculate the test statistics using the eBayes function:

```
> eb = eBayes(fitContrasts)
> topTableF(eb, adjust="BH")
```

	C1	C2	C3	AveExpr	F	P.Value	adj.P.Val
207783_x_at	-13.45607	0.26592268	-13.19015	12.938786	8622.529	1.207531e-69	1.419929e-65
201492_s_at	-13.27594	0.15357702	-13.12236	13.336090	8605.649	1.274450e-69	1.419929e-65
208834_x_at	-12.76411	0.06134018	-12.70277	13.160201	6939.501	4.749368e-67	3.527673e-63
212869_x_at	-13.77957	0.26008165	-13.51948	13.452076	6593.346	1.939773e-66	1.080599e-62
212284_x_at	-13.59977	0.29135767	-13.30841	13.070844	5495.716	2.893287e-64	1.289423e-60
208825_x_at	-12.70979	0.08250821	-12.62728	13.108072	5414.741	4.350100e-64	1.615555e-60
211445_x_at	-10.15890	-0.06633356	-10.22523	9.853817	5256.114	9.845076e-64	3.133969e-60
213084_x_at	-12.59345	0.03015520	-12.56329	13.046529	4790.107	1.260201e-62	3.510132e-59
201429_s_at	-13.33686	0.28358293	-13.05328	12.941208	4464.995	8.675221e-62	2.147888e-58
214327_x_at	-12.60146	0.20934783	-12.39211	11.832607	4312.087	2.257025e-61	5.029329e-58

7 Applying the ComBat function to adjust for known batches

The ComBat function adjusts for known batches using an empirical Bayesian framework [**?**]. In order to use the function, you must have a known batch variable in your dataset.

```
> batch = pheno$batch
```

Just as with sva, we then need to create a model matrix for the adjustment variables, including the variable of interest. Note that you do not include batch in creating this model matrix - it will be included later in the ComBat function. In this case there are no other adjustment variables so we simply fit an intercept term.

```
> modcombat = model.matrix(~1, data=pheno)
```

Note that adjustment variables will be treated as given to the ComBat function. This means if you are trying to adjust for a categorical variable with p different levels, you will need to give ComBat p-1 indicator variables for this covariate. We recommend using the model.matrix function to set these up. For continuous adjustment variables, just give a vector in the containing the covariate values in a single column of the model matrix.

We now apply the ComBat function to the data, using parametric empirical Bayesian adjustments.

```
> combat_edata = ComBat(dat=edata, batch=batch, mod=modcombat, par.prior=TRUE, prior.plot
```

```
Found 5 batches
Adjusting for 0 covariate(s) or covariate level(s)
Standardizing Data across genes
Fitting L/S model and finding priors
Finding parametric adjustments
Adjusting the Data
```

This returns an expression matrix, with the same dimensions as your original dataset. This new expression matrix has been adjusted for batch. Significance analysis can then be performed directly on the adjusted data using the model matrix and null model matrix as described before:

```
> pValuesComBat = f.pvalue(combat_edata,mod,mod0)
> qValuesComBat = p.adjust(pValuesComBat,method="BH")
```

These P-values and Q-values now account for the known batch effects included in the batch variable.

There are two additional options for the `ComBat` function. By default, it performs parametric empirical Bayesian adjustments. If you would like to use nonparametric empirical Bayesian adjustments, use the `par.prior=FALSE` option (this will take longer). Additionally, use the `prior.plots=TRUE` option to give prior plots with black as a kernel estimate of the empirical batch effect density and red as the parametric estimate. For example, you might chose to use the parametric Bayesian adjustments for your data, but then can check the plots to ensure that the estimates were reasonable.

Finally, we have now added the `mean.only=TRUE` option, that only adjusts the mean of the batch effects across batches (default adjusts the mean and variance). This option is recommended for cases where milder batch effects are expected (so no need to adjust the variance), or in cases where the variances are expected to be different across batches due to the biology. For example, suppose a reseracher wanted to project a knock-down genomic signature to be projected into the TCGA data. In this case, the knockdowns samples may be very similar to each other (low variance) whereas the signature will be at varying levels in the TCGA patient data. Thus the variances may be very different between the two batches (signature perturbation samples vs TCGA), so only adjusting the mean of the batch effect across the samples might be desired in this case.

8 Removing known batch effects with a linear model

Direct adjustment for batch effects can also be performed using the `f.pvalue` function. In the bladder cancer example, one of the known variables is a batch variable. This variable can be included as an adjustment variable in both `mod` and `mod0`. Then the `f.pvalue` function can be used to detect differential expression. This approach is a simplified version of `ComBat`.

```
> modBatch = model.matrix(~as.factor(cancer) + as.factor(batch),data=pheno)
> mod0Batch = model.matrix(~as.factor(batch),data=pheno)
> pValuesBatch = f.pvalue(edata,modBatch,mod0Batch)
> qValuesBatch = p.adjust(pValuesBatch,method="BH")
```

9 Surrogate variables versus direct adjustment

The goal of the `sva` is to remove all unwanted sources of variation while protecting the contrasts due to the primary variables included in `mod`. This leads to the identification of features that are consistently different between groups, removing all common sources of latent variation.

In some cases, the latent variables may be important sources of biological variability. If the goal of the analysis is to identify heterogeneity in one or more subgroups, the `sva` function may not be appropriate. For example, suppose that it is expected that cancer samples represent two distinct, but unknown subgroups. If these subgroups have a large impact on expression, then one or more of the estimated surrogate variables may be very highly correlated with subgroup.

In contrast, direct adjustment only removes the effect of known batch variables. All sources of latent biological variation will remain in the data using this approach. In other words, if the samples were obtained in different environments, this effect will remain in the data. If important sources of heterogeneity (from different environments, lab effects, etc.) are not accounted for, this may lead to increased false positives.

10 Variance filtering to speed computations when the number of features is large ($m > 100,000$)

When the number of features is very large ($m > 100,000$) both the `num.sv` and `sva` functions may be slow, since multiple singular value decompositions of the entire data matrix must be computed. Both functions include a variance filtering term, `vfilter`, which may be used to speed up the calculation. `vfilter` must be an integer between 100 and the total number of features m. The features are ranked from most variable to least variable by standard deviation. Computations will only be performed on the `vfilter` most variable features. This can improve computational time, but caution should be exercised, since the surrogate variables will only be estimated on a subset of the matrix. Running the functions with fewer than 1,000 features is not recommended.

```
> n.sv = num.sv(edata,mod,vfilter=2000,method="leek")
> svobj = sva(edata,mod,mod0,n.sv=n.sv,vfilter=2000)
```

```
Number of significant surrogate variables is:  2
Iteration (out of 5 ):1  2  3  4  5
```

11 Applying the `fsva` function to remove batch effects for prediction

The surrogate variable analysis functions have been developed for population-level analyses such as differential expression analysis in microarrays. In some cases, the goal of an analysis is prediction. In this case, data sets are generally composed a training set and a test set. For each sample in the training set, the outcome/class is known, but latent sources of variability are unknown. For the samples in the test set, neither the outcome/class or the latent sources of variability are known.

"Frozen" surrogate variable analysis can be used to remove latent variation in the test data set. To illustrate these functions, the bladder data can be separated into a training and test set.

```
> set.seed(12354)
> trainIndicator = sample(1:57,size=30,replace=F)
> testIndicator = (1:57)[-trainIndicator]
> trainData = edata[,trainIndicator]
> testData = edata[,testIndicator]
> trainPheno = pheno[trainIndicator,]
> testPheno = pheno[testIndicator,]
```

Using these data sets, the *pamr* package can be used to train a predictive model on the training data, as well as test that prediction on a test data set.

```
> mydata = list(x=trainData,y=trainPheno$cancer)
> mytrain = pamr.train(mydata)
```

```
1234567891011121314151617181920212223242526272829 30
```

```
> table(pamr.predict(mytrain,testData,threshold=2),testPheno$cancer)
```

```
        Biopsy Cancer Normal
  Biopsy      3      1      4
  Cancer      0     16      1
  Normal      0      2      0
```

Next, the sva function can be used to calculate surrogate variables for the training set.

```
> trainMod = model.matrix(~cancer,data=trainPheno)
> trainMod0 = model.matrix(~1,data=trainPheno)
> trainSv = sva(trainData,trainMod,trainMod0)
```

```
Number of significant surrogate variables is:  6
Iteration (out of 5 ):1  2  3  4  5
```

The fsva function can be used to adjust both the training data and the test data. The training data is adjusted using the calculated surrogate variables. The testing data is adjusted using the "frozen" surrogate variable algorithm. The output of the fsva function is an adjusted training set and an adjusted test set. These can be used to train and test a second, more accurate, prediction function.

```
> fsvaobj = fsva(trainData,trainMod,trainSv,testData)
> mydataSv = list(x=fsvaobj$db,y=trainPheno$cancer)
> mytrainSv = pamr.train(mydataSv)
```

```
1234567891011121314151617181920212223242526272829 30
```

```
> table(pamr.predict(mytrainSv,fsvaobj$new,threshold=1),testPheno$cancer)
```

```
        Biopsy Cancer Normal
  Biopsy      3      0      1
  Cancer      0     19      0
  Normal      0      0      4
```

12 sva for sequencing (svaseq)

In our original work we used the identify function for data measured on an approximately symmetric and continuous scale. For sequencing data, which are often represented as counts, a more suitable model may involve the use of a moderated log function [**?**, **?**]. For example in Step 1 of the algorithm we may first transform the gene expression measurements by applying the function $log(g_{ij} + c)$ for a small positive constant. In the analyses that follow we will set $c = 1$.

First we set up the data by filtering low count genes and identify potential control genes. The group variable in this case consists of two different treatments.

```
> library(zebrafishRNASeq)
> library(genefilter)
> data(zfGenes)
> filter = apply(zfGenes, 1, function(x) length(x[x>5])>=2)
> filtered = zfGenes[filter,]
> genes = rownames(filtered)[grep("^ENS", rownames(filtered))]
> controls = grepl("^ERCC", rownames(filtered))
> group = as.factor(rep(c("Ctl", "Trt"), each=3))
> dat0 = as.matrix(filtered)
```

Now we can apply svaseq to estimate the latent factor. In this case, we set $n.sv = 1$ because the number of samples is small ($n = 6$) but in general svaseq can be used to estimate the number of latent factors.

```
> ## Set null and alternative models (ignore batch)
> mod1 = model.matrix(~group)
> mod0 = cbind(mod1[,1])
> svseq = svaseq(dat0,mod1,mod0,n.sv=1)$sv
```

```
Number of significant surrogate variables is:  1
Iteration (out of 5 ):1  2  3  4  5
```

```
> plot(svseq,pch=19,col="blue")
```

13 Supervised sva

In our original work we introduced an algorithm for estimating the genes affected only by unknown artifacts empirically [**?**, **?**]. Subsequently, Gagnon-Bartsch and colleagues [**?**] used our surrogate variable model but made the important point that for some technologies or experiments control probes can be used to identify the set of genes only affected by artifacts. Supervised sva uses known control probes to estimate the surrogate variables. You can use supervised sva with the standard sva function. Here we show an example of how to perform supervised sva with the svaseq function.

```
> sup_svseq = svaseq(dat0,mod1,mod0,controls=controls,n.sv=1)$sv
```

```
sva warning: controls provided so supervised sva is being performed.
Number of significant surrogate variables is:  1
```

```
> plot(sup_svseq, svseq,pch=19,col="blue")
```

Here we passed the controls argument, which is a vector of values between 0 and 1, representing the probability that a gene is affected by batch but not affected by the group variable. Since we have known negative control genes in this example, we simply set controls[i] = TRUE for all control genes and controls[i] = FALSE for all non-controls.

14 What to cite

The sva package includes multiple different methods created by different faculty and students. It would really help them out if you would cite their work when you use this software.

To cite the overall sva package cite:

- Leek JT, Johnson WE, Parker HS, Jaffe AE, and Storey JD. (2012) The sva package for removing batch effects and other unwanted variation in high-throughput experiments. Bioinformatics DOI:10.1093/bioinformatics/bts034

For sva please cite:

- Leek JT and Storey JD. (2008) A general framework for multiple testing dependence. Proceedings of the National Academy of Sciences , 105: 18718-18723.
- Leek JT and Storey JD. (2007) Capturing heterogeneity in gene expression studies by 'Surrogate Variable Analysis'. PLoS Genetics, 3: e161.

For combat please cite:

- Johnson WE, Li C, Rabinovic A (2007) Adjusting batch effects in microarray expression data using empirical Bayes methods. Biostatistics, 8 (1), 118-127

For svaseq please cite:

- Leek JT (2014) svaseq: removing batch and other artifacts from count-based sequencing data. bioRxiv doi: TBD

For supervised sva please cite:

- Leek JT (2014) svaseq: removing batch and other artifacts from count-based sequencing data. bioRxiv doi: TBD
- Gagnon-Bartsch JA, Speed TP (2012) Using control genes to correct for unwanted variation in microarray data. Biostatistics 13:539-52.

For fsva please cite:

- Parker HS, Bravo HC, Leek JT (2013) Removing batch effects for prediction problems with frozen surrogate variable analysis arXiv:1301.3947

For psva please cite:

- Parker HS, Leek JT, Favorov AV, Considine M, Xia X, Chavan S, Chung CH, Fertig EJ (2014) Preserving biological heterogeneity with a permuted surrogate variable analysis for genomics batch correction Bioinformatics doi: 10.1093/bioinformatics/btu375