

# Using dualKS

Eric J. Kort and Yarong Yang

April 23, 2010

## 1 Overview

The Kolmogorov Smirnov rank-sum statistic measures how biased the ranks of a subset of items are among the ranks of the entire set. In other words, do the items tend to occur early or late in the ordered list, or are they randomly dispersed in the list?

This is an intuitive way to think about gene set enrichment. Indeed, it is the basis for Gene Set Enrichment Analysis described by Subramanian et al. (2005). This package takes this approach to the problem of multi-class classification, wherein samples of interest are assigned to one of 2 or more classes based on their gene expression. For example, to which of the several sub-types of renal cell carcinoma does a new kidney tumor sample belong? This methodology is described in detail in our (forthcoming) paper, Kort et al. (2008).

This package is called “dualKS” because it applies the KS algorithm in two ways. First, we use the KS approach to perform discriminant analysis by applying it gene-wise to a training set of known classification to identify those genes whose expression is most biased (upward, downward, or both) in each class of interest. For example, say we take gene 1 and sort the samples by their expression of this gene. Then we ask to what extent each class is biased in that sorted list. If all samples of, say, class 1 occur first, then this gene receives a high score for that class and will be included in the final signature of “upregulated genes” for that class. And so on, for every gene in the dataset. This process is frequently termed “discriminant analysis” because it identifies the most discriminating genes.

The second manner in which the KS algorithm is applied is for classification. Based on the signatures identified in the first step (or via some other mechanism of the user’s choosing), we apply the algorithm sample-wise to ask which of these signatures has the strongest bias in new samples of

interest in order to assign these samples to one of the classes. In other words, when we sort all the genes expression values for a given sample, how early (or late) in that list are the genes for a given signature found? This second step is essentially the same as the algorithm described by Subramanian et al. with equal weights given to each gene. While we have developed this package with the task of *classification* in mind, it can just as readily be used to identify which biologically relevant gene signatures are *enriched* in samples of interest (indeed, classification is simply enrichment analysis with a class-specific gene signature).

The KS approach has several attractive factors. First, the idea of identifying which genes exhibit a class-dependant bias in expression and, conversely, which signatures are most biased in a given sample, is an intuitive way of thinking about discriminant analysis and classification that most biologists can readily grasp. Second, the resulting gene signatures may be quite parsimonious—an attractive feature when one is planning on downstream validation or implementation of a non-array based assay. Third, the algorithm can deal with many classes simultaneously. Finally, the algorithm does not require iteration like random selection algorithms do.

## 2 An Example

The package includes an illustrative dataset that is a subset of GEO data set GDS2621. We have taken a small subset of the genes in the original dataset to make the analysis run quickly, but you will obtain the same results if you use the entire dataset from the GEO website and the analysis will take only a few minutes longer. This data set contains one color affymetrix gene expression data:

```
> library("dualKS")
> data("dks")
> pData(eset)
```

|          | class  |
|----------|--------|
| GSM34379 | normal |
| GSM34383 | normal |
| GSM34385 | normal |
| GSM34388 | normal |
| GSM34391 | normal |
| GSM34393 | osteo  |
| GSM34394 | osteo  |

```

GSM34395      osteo
GSM34396      osteo
GSM34397      osteo
GSM34398      rheumatoid
GSM34399      rheumatoid
GSM34400      rheumatoid
GSM34401      rheumatoid
GSM34402      rheumatoid

```

As you can see, there are five samples each of normal synovial fluid, synovial fluid from patients with Osteoarthritis, and synovial fluid from patients with Rheumatoid Arthritis. We will now build a classifier that can distinguish these diagnoses.

The first step is to rank each gene based on how biased its expression is in each of the classes. You have the option of scoring genes based on how upregulated or downregulated they are, or both. For one color data such as we have in this dataset, genes with low expression exhibit a great deal of noise in the data. Therefore, we will focus only on those genes that are upregulated in each class, as specified by the `type="up"` parameter.

```
> tr <- dksTrain(eset, class = 1, type = "up")
```

The `class = 1` instructs the software to look in column 1 of the `phenoData` object contained within `eset` to determine which class each sample belongs to. Alternatively, you can provide a factor specifying the classes directly.

Now, we will extract a classifier of 100 genes per class from the training data. We can subsequently choose classifiers of different sizes (and, perhaps, select among these classifiers using ROC analysis) by specifying a different value for `n`, without re-running the (more time intensive) `dksTrain` function.

```
> cl <- dksSelectGenes(tr, n = 100)
```

We can then apply this classifier to a test data set. However, in this case we will just run it against the training set to check for internal consistency (if we can't classify the training set from which the classifier is derived, we are in real trouble!)

```
> pr <- dksClassify(eset, cl)
> summary(pr, actual = pData(eset)[, 1])
```

Dual KS Classification Summary:

Predicted class frequencies:

| normal | osteo | rheumatoid |
|--------|-------|------------|
| 11     | 0     | 4          |

Concordance rate (predicted==actual): 60 %

> show(pr)

|    | sample   | predicted class | prediction score |
|----|----------|-----------------|------------------|
| 1  | GSM34379 | normal          | 1024.367         |
| 2  | GSM34383 | normal          | 1073.083         |
| 3  | GSM34385 | normal          | 1116.797         |
| 4  | GSM34388 | normal          | 971.7            |
| 5  | GSM34391 | normal          | 1159.983         |
| 6  | GSM34393 | normal          | 592.5            |
| 7  | GSM34394 | normal          | 671.763          |
| 8  | GSM34395 | normal          | 610.143          |
| 9  | GSM34396 | normal          | 624.89           |
| 10 | GSM34397 | normal          | 604.087          |
| 11 | GSM34398 | normal          | 604.613          |
| 12 | GSM34399 | rheumatoid      | 599.083          |
| 13 | GSM34400 | rheumatoid      | 727.853          |
| 14 | GSM34401 | rheumatoid      | 606.457          |
| 15 | GSM34402 | rheumatoid      | 657.28           |

Note the custom summary and show functions. By specifying the “actual” classes for the samples when calling summary, the percent correspondence rate is calculated and displayed along with the summary.

As you can see, we didn’t do too well with classification (our concordance rate is only 60%), even though we are only trying to classify our training set, which should be self-fulfillingly easy. What is the problem? To gain some insight, let’s look at plots of the data. The prediction object from `dksClassify` has its own `plot` method. The resulting plot allows for easy visualization of how the samples of a given class compare to the other samples in terms of KS scores for each signature. To visualize this, a separate panel is created for each signature. The samples are sorted in decreasing

order according to their score on that signature. Then for each sample, its score for *each* signature is plotted to allow easy comparison between the signature scores for each sample. The signatures are color coded to distinguish one signature score from another. Finally, a color coded bar is plotted below each sample indicating its predicted and (if provided) actual class for ready identification of outliers and misclassified samples.

There is a lot of information compactly summarized in these plots, so let's consider the example:

```
> plot(pr, actual = pData(eset)[, 1])
```



Figure 1: Plot of samples sorted by their score for the indicated class

To begin, examine the first panel of figure 1 corresponding to the signature for “normal” samples. This panel show the samples sorted according to their score for the ‘normal’ signature. Not surprisingly, the normal samples (red bars) are clustered to the left, with the highest scores. As you move to the right, not only does the red line decrease (because the samples are sorted by this score), but the lines for the other signatures (blue line=rheumatoid signature score, green line=osteoarthritis signature score) begin to increase. For each sample (indicated by the bars along the bottom), the predicted class is the signature for which that sample’s score is the maximum. So where the green line is on top, the prediction is “osteoarthritis”, and where the red line is on top, the prediction is “normal”.

The other two panels work in the same way, but now the samples have been sorted by decreasing rheumatoid signature score or osteoarthritis signature score, respectively. This way of plotting the data allows easy visualization of the degree to which each class is upwardly biased in the list of samples sorted by the corresponding signature. It also allows the natural “knee point” or threshold for any given signature to be identified.

This plot demonstrates why our concordance rate is so poor. While the normal gene expression signature is *most* upwardly biased in the normal samples, these genes also have high expression in the other samples (this is not really too surprising—abnormal cells have a lot of the same work to do as normal cells, biologically speaking). So in many cases, the most upwardly biased genes in, say, osteoarthritis are still not as upwardly biased as highly expressed “normal” genes.

There are three methods to correct this problem. The most simplistic method is to solve the problem by brute force. Rather than classifying samples based on maximum raw KS score (as in figure 1), we will classify based on *relative* KS score. We will accomplish this by rescaling the scores for each signature so that they fall between 0 and 1 (by simply subtracting the minimum score for a given score across all samples and then dividing by the maximum). This is achieved by supplying the `rescale=TRUE` option to `dksClassify`:

```
> pr <- dksClassify(eset, cl, rescale = TRUE)
> summary(pr, actual = pData(eset)[, 1])
```

Dual KS Classification Summary:

Predicted class frequencies:

| normal | osteo | rheumatoid |
|--------|-------|------------|
| 5      | 5     | 5          |

Concordance rate (predicted==actual): 100 %

And we can plot the results as before:

```
> plot(pr, actual = pData(eset)[, 1])
```

The resulting plot is shown in figure 2. The classification is now inline with our expectation. A draw back of this approach is that the rescaling

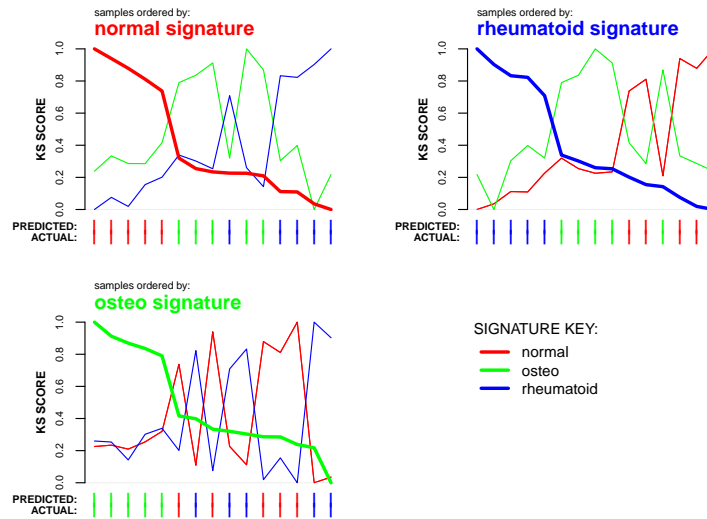


Figure 2: Classification with rescaling.

requires that the dataset being classified contains multiple samples from *each* class. Therefore, a single new sample cannot be classified using rescaling. Conversion to ratio space avoids this problem as discussed in section 4.

Finally, you may want to examine a plot of the running sum of the KS scores for an individual sample. You can call the function `KS` directly to access the running sum for each signature and the plot it. Here is an example of how that might be accomplished for the first sample in our data set using our previously defined classifier object (`cl`):

```
> sc <- KS(exprs(eset)[, 1], cl@genes.up)
> plot(sc$runningSums[, 1], type = "l", ylab = "KS sum",
+      ylim = c(-1200, 1200), col = "red")
> par(new = TRUE)
> plot(sc$runningSums[, 2], type = "l", ylab = "KS sum",
+      ylim = c(-1200, 1200), col = "green")
> par(new = TRUE)
> plot(sc$runningSums[, 3], type = "l", ylab = "KS sum",
+      ylim = c(-1200, 1200), col = "blue")
> legend("topright", col = c("red", "green", "blue"), lwd = 2,
+      legend = colnames(sc$runningSums))
```

As shown in figure 3, it looks like that first sample belongs to the class “normal”. Note that the maximum value achieved by each line corresponds to the KS score (not rescaled) on the corresponding signature for sample 1.

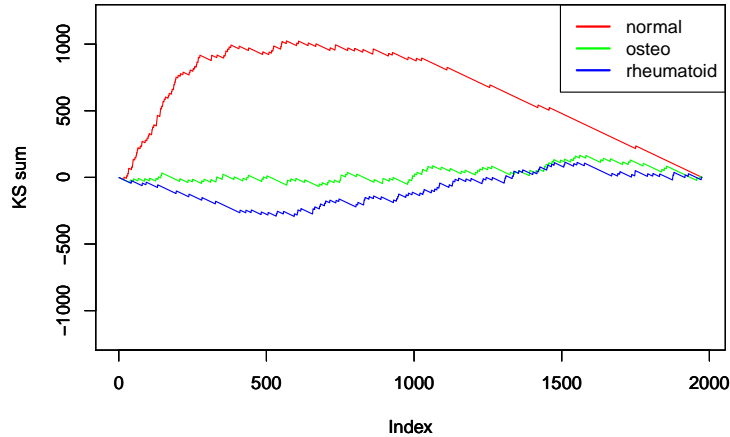


Figure 3: Plot of sample 1’s running sum of KS statistic for each signature.

The other two methods for addressing the problem in our original classification require a bit more effort (mostly on the part of the CPU), but are conceptually more satisfying because they use biologically relevant information encoded in the data. These two methods—weighting by expression level and converting the data to ratios vs. a relevant reference—are covered in the next two sections.

### 3 Discriminant analysis with weights

By default, signatures are defined by this package based on the ranks of members of each class when sorted on each gene. Those genes for which a given class has the highest rank when sorting samples by those genes will be included in the classifier, with no regard to the absolute expression level of those genes. This is the classic KS statistic.

Very discriminant genes identified in this way may or may not be the highest expressed genes. For example, a gene with very low expression but also very low variance may be slightly over-expressed in one subgroup. The result is that signatures identified in this way have arbitrary “baseline” values



as we saw in figure 1. Our first solution was to force the range of values for each signature across samples to be between 0 and 1. An alternative to this brute force approach is to weight the genes based on some biologically interesting statistic.

By adding the `weights = TRUE` option to `dksTrain`, the genes will be weighted according to the  $\log_{10}$  of their mean relative rank in each class. More specifically the weight for gene  $i$  and class  $j$  is:

$$w_{ij} = -\log_{10} \frac{\bar{R}_{ij}}{n}$$

Where  $\bar{R}_{ij}$  is the *average rank* of gene  $i$  across all samples of class  $j$  and  $n$  is the total number of genes. Therefore, genes with highest absolute expression in a given class are more likely to be included in the signature for that class. For example:

```
> tr <- dksTrain(exprs(eset), class = pData(eset)[, 1],
+   type = "up", weights = TRUE)
> cl <- dksSelectGenes(tr, n = 100)
> pr <- dksClassify(exprs(eset), cl)
> plot(pr, actual = pData(eset)[, 1])
```

The results are plotted in figure 4. As you can see, the range of KS scores across all samples and classes are now much more consistent, and the resulting classification is much better.

Alternatively, you may provide your own weight matrix based on the metric of your choosing as the argument to `weights`. This matrix must have one column for each possible value of `class`, and one row for each gene in `eset`. NAs are handled gracefully by discarding any genes for which any column of the corresponding row of `weights` is NA. It is important to note that for `type='down'` or the down component of `type='both'`, the weight matrix will be inverted as `1 - weights`. Therefore, if using one of these methods, the weight matrix should have range 0 - 1.

Calculating the weight matrix is somewhat time consuming. If many calls to `dksTrain` are required in the course of some sort of optimization procedure, the user might want to calculate the weight matrix once. This can be done as follows:

```
> wt <- dksWeights(eset, class = 1)
```

Then the resulting weight matrix can be supplied directly to `dksTrain` to avoid recalculating it on each call:

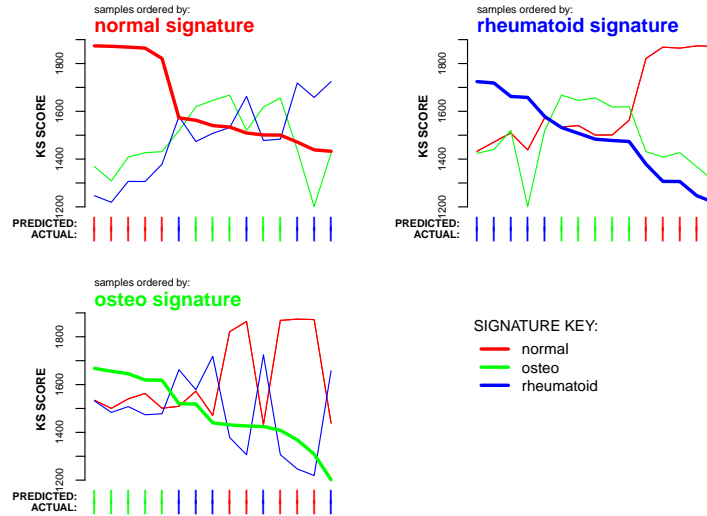


Figure 4: Weighted KS analysis.

```
> tr <- dksTrain(exprs(eset), class = 1, weights = wt)
```

This is a short cut to speed up validation. However, if some type of "leave some out" validation is being performed, the user should carefully consider the implications of calculating the weight matrix on all samples and then training/testing on subsets. Nevertheless, this may be useful for initial validation runs.

The final analytic alternative we will consider is converting the data to ratios based on some rational reference data.

## 4 Discriminant analysis with ratios

As mentioned above, one color microarray data does not lend itself well to including highly *down regulated* genes in the classifier, because genes with very low measured expression levels have very poor signal:noise ratios. (This may be mitigated somewhat by enforcing some reasonable threshold below which the data is discarded). Furthermore, as we saw above, basing classifiers on the raw expression level can lead to problems when genes highly expressed in one class are also highly expressed in other classes (albeit slightly less so).

Bidirectional discriminant analysis (i.e., including both up and down regulated genes) can be performed on ratio data where the "downregulated"

genes are not necessarily of miniscule absolute intensity, but are simply much smaller relative to the reference. Such ratio data may either arise in the course of two color microarray experiments, or may be generated by dividing one color data from samples by the mean expression of some reference samples. By converting to ratios, it is not only easier to include both up and down regulated genes, but we are now examining not raw expression level but *change* in expression level relative to the reference. This is often what is of greatest biological interest. (Note that very small values should still be excluded from the data prior to calculating ratios to avoid artefactually astronomical ratios.)

To illustrate, we will transform our data into ratio data by dividing the osteoarthritis and rheumatoid arthritis samples by the mean expression of each gene in the normal samples. First we calculate the mean values for the normal samples:

```
> ix.n <- which(pData(eset)[, 1] == "normal")
> data <- exprs(eset)
> data.m <- apply(data[, ix.n], 1, mean, na.rm = TRUE)
```

Now we drop the normals from our data set, and calculate the ratios (expression relative to average normal expression) using the `sweep` function.

```
> data <- data[, -ix.n]
> data.r <- sweep(data, 1, data.m, "/")
```

Finally, we convert to log2 space and perform the discriminant analysis and classification of our test cases.

```
> data.r <- log(data.r, 2)
> tr <- dksTrain(data.r, class = pData(eset)[-ix.n, 1],
+   type = "both")
> cl <- dksSelectGenes(tr, n = 100)
> pr <- dksClassify(data.r, cl)
> plot(pr, actual = pData(eset)[-ix.n, 1])
```

And here is the summary information for this classifier:

```
> summary(pr, actual = pData(eset)[-ix.n, 1])
```

Dual KS Classification Summary:



Figure 5: Bidirectional analysis of ratio data.

Predicted class frequencies:

```

oste    rheumatoid
5              5

```

Concordance rate (predicted==actual): 100 %

> show(pr)

|    | sample   | predicted class | prediction score |
|----|----------|-----------------|------------------|
| 1  | GSM34393 | osteo           | 719.427          |
| 2  | GSM34394 | osteo           | 606.983          |
| 3  | GSM34395 | osteo           | 840.297          |
| 4  | GSM34396 | osteo           | 734.7            |
| 5  | GSM34397 | osteo           | 805.537          |
| 6  | GSM34398 | rheumatoid      | 1702.713         |
| 7  | GSM34399 | rheumatoid      | 1668.743         |
| 8  | GSM34400 | rheumatoid      | 2141.69          |
| 9  | GSM34401 | rheumatoid      | 1926.283         |
| 10 | GSM34402 | rheumatoid      | 1997.383         |

## 5 Significance testing

Once you have identified which signature has the maximum score in a given sample, you will likely want to determine if that score is significantly elevated. The distribution of KS scores generated by this package tend to follow a gamma distribution, but the parameters of the distribution vary depending on the size of the signature and the total number of genes. Therefore, we take a bootstrapping approach to generate an estimated distribution and then identify the gamma distribution that best fits the estimate.

To perform the bootstrap, the sample classes are randomly permuted and then signatures are generated for these bootstrap classes. We take this approach because it preserves the relationships between genes (as opposed to generating gene signatures by randomly selecting genes). Each time this is done, we get  $k * c$  bootstrap samples where  $k$  is the number of samples and  $c$  is the number of classes. The entire process is repeated until the requested number of samples is generated.

The function returns a function that is `1-pgamma(x, ...)` where `...` is the optimized gamma distribution parameters identified by the bootstrap and fitting procedures. Then you can simply call this function with one or more KS scores to calculate the p-values. You must provide an `ExpressionSet` (for use in bootstrapping), the class specification for the samples in that set (which will then be permuted), and the number of genes in each signature (`n`).

```
> pvalue.f <- dksPerm(eset, 1, type = "both", samples = 500)
```

That's not nearly enough samples to obtain a reasonable estimate of the gamma distribution (1,000-10,000 is more like it), but it will suffice for this demonstration. (Generating several thousand samples takes a few minutes.) Now let's calculate the estimated p-values for our predicted classes from before:

```
> pvalue.f(pr@predictedScore)
```

| GSM34393      | GSM34394      | GSM34395      | GSM34396      | GSM34397      |
|---------------|---------------|---------------|---------------|---------------|
| 0.07798677658 | 0.13674004715 | 0.04155830261 | 0.07212062357 | 0.04992270575 |
| GSM34398      | GSM34399      | GSM34400      | GSM34401      | GSM34402      |
| 0.00030514036 | 0.00037354061 | 0.00002153325 | 0.00007971971 | 0.00005183275 |

It appears that most of the scores leading to the predicted classes are unlikely to be the result of random variation. (In reality, small bootstrap

samples lead to consistent underestimation of the p-values in this context. A run with much larger **sample** will produce smaller p-values.) When many classes are examined, appropriate controls for multiple comparisons should be considered.

Note that for the resulting probability density function to meaningfully describe the probability of obtaining observed scores, the parameters provided to **dksPerm** must match those used when classifying with **dksClassify**. Different settings will produce different distributions of scores, so you must generate a probability density function for each set of parameters and each training dataset (unless those datasets can be assumed to come from the same underlying distribution).

## 6 Classification using your own gene signatures.

If you have predefined signatures (established empirically or by some other methodology), you can still calculate their enrichment in test samples using **dksClassify**. That function requires an object of type **DKSClassifier**. The package provides a utility function to create this object from your list of gene ids. Rather than create a separate list of genes for each class, you simply provide a single list of gene ids and a factor indicating which class each gene belongs to. Note, however, that you must provide a separate list for upregulated and downregulated genes—although you may provide only one or the other if you wish.

As an example, we will create some arbitrary signatures and perform classification with them. In the example below, **sig.up** would be the meaningful sets of (upregulated) gene ids you have pre-identified, and **cls** would be the class to which each gene in your signature belongs.

```
> cls <- factor(sample(pData(eset)[, 1], 300, replace = TRUE))
> sig.up <- sample(rownames(exprs(eset)), 300)
> classifier <- dksCustomClass(upgenes = sig.up, upclass = cls)
> pr.cust <- dksClassify(eset, classifier)
```

If you were to plot this prediction object with **actual=pData(eset[,1])** you would note that the classification is very poor—which is reassuring since this is a random classifier.

## 7 Accessing slots for downstream analysis

Since the classes defined in this package are not complex, we have not bothered (yet) to write accessors. The reader is referred to the docs for further details. Suffice it to say here that the most useful slots for the user are likely to be those of `DKSPredicted`. For example, we can construct a useful table for downstream analysis:

```
> results <- data.frame(pr@predictedClass, pr@scoreMatrix)
> results
```

|          | pr.predictedClass | osteo                   | rheumatoid |
|----------|-------------------|-------------------------|------------|
| GSM34393 | osteo             | 719.4266666666667333629 | 313.6300   |
| GSM34394 | osteo             | 606.9833333333333484916 | 561.4267   |
| GSM34395 | osteo             | 840.2966666666668515973 | 555.6333   |
| GSM34396 | osteo             | 734.7000000000002728484 | 465.8367   |
| GSM34397 | osteo             | 805.5366666666666333185 | 430.8133   |
| GSM34398 | rheumatoid        | 3.1600000000003771738   | 1702.7133  |
| GSM34399 | rheumatoid        | 3.1600000000003771738   | 1668.7433  |
| GSM34400 | rheumatoid        | 2.1066666666670439412   | 2141.6900  |
| GSM34401 | rheumatoid        | 0.0000000000003774758   | 1926.2833  |
| GSM34402 | rheumatoid        | 7.3733333333337105486   | 1997.3833  |

## References

- E.J. Kort, Y. Yang, Z. Zhang, B.T. Teh, and N. Ebrahimi. Gene selection and classification of microarray data by twins kolmogorov-smirnov analysis. *Technical Report*, 2008.
- A. Subramanian, P. Tamayo, V. K. Mootha, S. Mukherjee, B. L. Ebert, M. A. Gillette, A. Paulovich, S. L. Pomeroy, T. R. Golub, E. S. Lander, and J. P. Mesirov. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proc Natl Acad Sci U S A*, 102(43):15545–50, 2005.