Software Manual

Institute of Bioinformatics, Johannes Kepler University Linz

KeBABS — An R Package for Kernel Based Analysis of Biological Sequences

Johannes Palme and Ulrich Bodenhofer

Institute of Bioinformatics, Johannes Kepler University Linz
Altenberger Str. 69, 4040 Linz, Austria
kebabs@bioinf.jku.at

Version 1.0.5, February 9, 2015

Scope and Purpose of this Document

This document is the user manual for the R package `kebabs`. It is meant as an introduction to the KeBABS functionality which should make the user acquainted with the basic functions of the package both on a conceptual level as well as from the usage perspective. This means that the document will neither go into full details for the described functions nor can it be considered a complete description of all package functions. As an additional source of information, the help pages including examples are available.

Please also consider that this manual is not an introduction to R nor an introduction to machine learning in general or to support vector machine-based learning. The document provides the information in a form that also allows less experienced users to use the package. However, if you do not have any knowledge of these subjects, please read introductory literature first.

Authorship and Contributions

Except for the header files `khash.h` and `ksort.h` and three small stub files for Bioconductor packages this package was written entirely by Johannes Palme as part of his master thesis in the Bioinformatics master program at the Johannes Kepler University in Linz, Austria, under the supervision of Ulrich Bodenhofer. The author wants to express his sincere gratitude to Ulrich Bodenhofer for numerous lively discussions throughout the development of the package. The two header files mentioned above provide a fast hash table implementation and sorting algorithms implemented in pure C code and are part of the open-source C-library `klib` (AttractiveChaos, 2011). The R code for prediction profile plots partly uses a similar functionality in the package `procoil` as inspiration (Bodenhofer, 2011).

The datasets delivered with the package come from two sources:

- TFBS - transcription factor binding site data: This data set is based on supplementary data provided with the publication "Discriminative prediction of mammalian enhancers from DNA sequence by Lee *et al.* (2011).
- CCoil - protein sequences for coiled coil proteins : This dataset is part of the data collection for package `procoil`[1].

KeBABS relies heavily on functionality provided by other R packages, especially the packages

- `Biostrings` for storing and manipulating biological sequences
- `e1071`, `Liblinear` and `kernlab` for SVM implementations used in binary and multiclass classification and regression scenarios

Special thanks go to the persons who have contributed to these packages, the C library `klib` or any of the data mentioned above.

[1] `http://www.bioinf.jku.at/software/procoil/data.html`

Contents

1 Introduction

The `kebabs` package provides functionality for kernel based analysis of biological sequences (DNA, RNA and amino acid sequences) via Support Vector Machine (SVM) based methods. Sequence kernels define similarity measures between sequences which can be used in various machine learning methods. The package provides four important kernels for sequence analysis in a very flexible and efficient implementation and extends their standard position-independent functionality in a novel way to take the position of patterns in the sequences into account. Following kernels are available in KeBABS:

- Spectrum kernel (Leslie *et al.*, 2002)
- Mismatch kernel (Leslie *et al.*, 2003)
- Gappy pair kernel (Kuksa *et al.*, 2008)
- Motif kernel (Ben-Hur and Brutlag, 2003)

All of the kernels except the mismatch kernel support position-independent and position-dependent similarity measures. Because of the flexibility of the kernel formulation other kernels like the weighted degree kernel (Rätsch and Sonnenburg, 2004) or the shifted weighted degree kernel with constant weighting of positions (Rätsch *et al.*, 2005) are included as special cases. The kernels also exist in an annotation-specific version which uses annotation information placed along the sequence. The package allows for the generation of a kernel matrix or an explicit representation for all available kernels which can be used with methods implemented in other R packages.

KeBABS focuses on SVM-based methods and provides a framework that simplifies the usage of existing SVM implementations in other R packages. In the current version of the package the SVMs in the R packages `kernlab` (Karatzoglou *et al.*, 2004), `e1071` (Chang and Lin, 2011) and `LiblineaR` (Fan *et al.*, 2008) from the "The Comprehensive R Archive Network" (CRAN)[2] are supported. Binary and multi-class classification as well as regression tasks can be used with the SVMs in these packages in a unified way without having to deal with the different functions, parameters and formats of the selected SVM. The package provides cross validation, grid search and model selection functions as support for choosing hyperparameters . Grouped cross validation, a form of cross validation which respects group membership is also available.

Another emphasis of this package is the biological interpretability of the machine learning results. This aspect is reflected in the computation of feature weights for all SVMs and prediction profiles which show the contribution of individual sequence positions to the prediction result and give an indication about the relevance of sequence sections for the learning result.

The flexible and efficient kernels and the supporting SVM related functionality in KeBABS together with the supported SVMs, especially the very fast linear SVMs in package LiblineaR, provide a fast, versatile and easy-to-use sequence analysis framework combined with improved biological interpretation possibilities of learning results through feature weights and prediction profiles.

[2] `http://cran.r-project.org`

2 Installation

2.1 Installation via Bioconductor

The R package kebabs (current version: 1.0.5) is part of the Bioconductor Project[3]. To install the package from this repository, please enter the following commands into your R session:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("kebabs")
```

2.2 Manual installation from source

The above described install method is the standard way of installing the package. If you e.g. want to compile the C/C++ code included in the package with special compile flags, manual installation of the package from source might be necessary. This is done in the following way:

- install packages on which kebabs depends via Bioconductor with

  ```
  source("http://bioconductor.org/biocLiteR")
  biocLite(c("Rcpp", "S4Vectors", "XVector", "Biostrings", "SparseM",
             "kernlab", "e1071","LiblineaR"))
  ```

- open the Bioconductor page of the package[4] in your web browser, download the source package kebabs_1.0.5.tar.gz and save the source package on your hard disk.
- open a shell/terminal/command prompt window and change to the directory where the source package kebabs_1.0.5.tar.gz is stored. Then install the package with:

  ```
  R CMD install kebabs_1.0.5.tar.gz
  ```

3 Getting Started

To load the package, enter the following in your R session:

```
> library(kebabs)
```

When the package is loaded several messages are displayed indicating that some objects are masked. This does not cause any problem for kebabs .Once the R session returns to the R command prompt ">" after loading the package, kebabs is ready for use.

The package includes this user manual which can be viewed in the R session with

```
> vignette("kebabs")
```

[3] http://www.bioconductor.org/

[4] http://www.bioconductor.org/packages/release/bioc/html/kebabs.html

Help pages can be accessed with

```
> help(kebabs)
```

In the following example, we perform SVM-based binary classification for prediction of EP300 bound enhancers from DNA sequence. From a set of sequences derived from ChIP-seq EP300 binding sites and a set of negative sequences sampled from the same genome we train a model which allows us to predict enhancers for new sequences. In this example a small subset of a much larger dataset is used for demonstration purpose.

The analysis in kebabs usually starts with sequence data for DNA, RNA or amino acid sequences. In the following examples, we use DNA sequences which are included in the kebabs package to demonstrate package functionality. In kebabs biological sequences are represented in one of the classes DNAStringSet, RNAStringSet or AAStringSet from package Biostrings. Sequences in this format can be created from character vectors containing the sequence data or through reading sequences from a FASTA file with the functions readDNAStringSet, readRNAStringSet, readAAStringSet from package Biostrings. In the first example, we load sample sequences provided with the kebabs package with

```
> data(TFBS)
```

The dataset contains 500 DNA sequence fragments with lengths between 277 and 2328 bases from the mouse genome mm9 used to study transcription factor binding for the EP300/ CREBBP binding proteins and a label vector which indicates whether the protein binds to a sequence fragment or not. The dataset is a sampled subset of the mouse enhancer forebrain data from Lee *et al.* (2011) based on a ChIP-seq study of Visel *et al.* (2009). Let us look at the sequences stored under the name enhancerFB, the number of sequences in the sequence set and the minimal and maximal length of the contained sequences

```
> enhancerFB
```

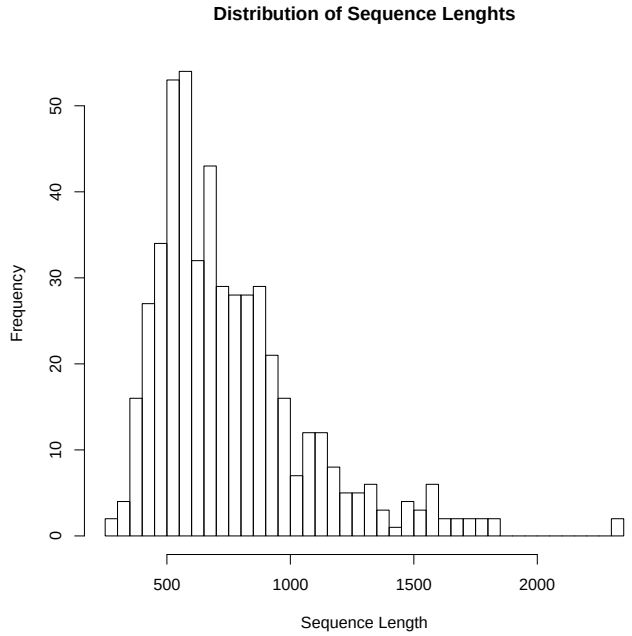
```
  A DNAStringSet instance of length 500
      width seq                                   names
  [1]   827 ACTAAACAACATCAATC...TAGGCAAAATCCTGACA chr19.21240050.21...
  [2]   678 GAATATAGACCCTTGGT...AAGTTATATTAATTTAT chr2.144463827.14...
  [3]   878 CACCCACATGGTGGCTC...TCCCCACTGTATCACTC chr7.38525408.385...
  [4]   927 TGCCGTAGTGTGCCAGC...TGATTCTCCTGGATCAA chr4.90747075.907...
  [5]   953 CTTCATATACCTATTAA...CCTAATTTAAAAAGGGG chr4.9148679.9149631
  ...   ... ...
[496]   478 ACGAGAATGAGGGAAAG...GTCTGCCTTCCGAGTGA chr8.45499713.455...
[497]  1552 AGGGGACTGGAAGAAAG...TCTCTTCTAAGCGAGCA chr8.94657200.946...
[498]   503 GATGAACGTAAAATGCA...TAATGACTCCCTTCTGA chr17.73900306.73...
[499]  1577 ACCCCTCTGAGACCAAG...GTAGGTCTGTATTCCTG chr9.90587075.905...
[500]   778 CACTGTCATAACTTGCT...GTAGAGTAGCTGCTCTC chr17.48680137.48...
```

```
> length(enhancerFB)
```

```
[1] 500
```

The length distribution of the sequences in the dataset can be shown with

```
> hist(width(enhancerFB), breaks=30, xlab="Sequence Length",
+      main="Distribution of Sequence Lengths")
```



To look at individual sequences from this set or parts of them, e.g. for sample number 3, we can use

```
> showAnnotatedSeq(enhancerFB, sel=3)
```

```
Sequence chr7.38525408.38526285 of class DNAStringSet

no annotation available

         0         1         2         3         4
         1         1         1         1         1
  [1]    CACCCACATGGTGGCTCTCAACCATGTATAACTCCAGTTTTCAGGAGATT
 [51]    TGATGCCTTCTTCTGGCCTCTGCAGGCATTGCACACATGTGATGCAGACA
[101]    TACATGCAGACAAAATGCCTATACAAATATAATAATAACAACAACAGCTT
[151]    TTAAATGACAAAGAAAGAAAGCAAATGGCCAAGATGTTAGGGGTGGCCAG
[201]    CCTCCTGTTTCCACCAGTAACCCTTCCTGCCCACGCCTCCCAATCACAGA
[251]    ACAATAGCTTCTATCCCACATTGCACTGACATGACCTCCATCCTGATGAG
```

```
[301]    AGTCTTCCTTCCCCACATGCAAAGTTCAAACAAGTAACTGTGGGGACTCT
[351]    ATGAAACATGGGTGTTTGCTGCACGTCAGTAACAACAGCCATCCCTCCCC
[401]    AGTATGAACAGAGGGATGCTTAACACCACATGAGACAGGATGCAGAAGTC
[451]    ACAGATCAGCCACTCTGATGTTGACCTGTTGCAATACAGAAAGCAGGACT
         0         1         2         3         4
         1         1         1         1         1
[501]    CCCCCAGAGTTCACGCATGCATACGATATTTAAGATGTCAGTCAGCTCTA
[551]    CGTTTTGCAATATCTTGCAATCAAGCAGTGAGCTGGTGGGGAAAAGCCAC
[601]    CCTCTTTTATATACTCTGAGGCTGTCAGTTTCACCCTGGGGCGTGCATCA
[651]    GAAACCTCGGGGTGGGGGCGGTGGCTAGTTCACACACAGGCCAGATCACC
[701]    AGAGGCCCCACGAGTCTGACTGCAGAAGCATGTGAATTTGCATTTCTAAC
[751]    CAGTTGCCATGTGACCTCTGCCCCCCTCTAACCAGGTTACCACTTGGAAA
[801]    ATCATCCCTTGTGTTCTTCTCAGGACCCTGCCACTGTATGGACAGAGACC
[851]    CGCTCCTCTGCTCCCCACTGTATCACTC
```

The label vector is named `yFB` and describes the interaction of the DNA sequence fragment and the binding factor considered in the underlying experiment. It contains a "1" for all positive samples to which the transcription factor binds and a "-1" for all negative samples for which no binding occurs. We look at the first few elements of the label vector with

```
> head(yFB)
```

```
[1]  1 -1 -1  1 -1 -1
```

and see that transcription factor binding occurs for the first and fourth sequence. With

```
> table(yFB)
```

```
yFB
 -1   1
259 241
```

we check the balancedness of the dataset and find that the dataset is balanced, i.e. that it contains approximately the same amount of positive and negative samples. Unbalanced data can have a strong impact on performance for SVM-based learning, therefore the user should always be aware of unbalanced data. Now being familiar with the data in the dataset we can start with sequence analysis.

To train a model we split the data set into a part used to train the model — the training set — and a part to predict the binding behavior from the model — the test set. For the training set we randomly select 70% of the sequences. The remaining sequences become the test set.

```
> numSamples <- length(enhancerFB)
> trainingFraction <- 0.7
> train <- sample(1:numSamples, trainingFraction * numSamples)
> test <- c(1:numSamples)[-train]
```

The vectors `train` and `test` now contain the sample indices for the training set and test set samples. The elements in `train` are unordered because sample just picks elements randomly from the given set.

In the next step, we select a kernel for the analysis. The sequences are character strings which are converted to numerical data representing similarity values with the help of sequence kernels. These similarity values are used to train a model via SVMs. In this example, the spectrum kernel is used. This kernel is described in detail in Section 4.1.1. First we define a kernel object with

```
> specK2 <- spectrumKernel(k=2)
```

which describes the type of kernel and the used setting of the kernel parameters. A spectrum kernel with $k = 2$ considers substrings of length 2 in the sequences for the similarity consideration.

Now everything is available to train a model from the sequences in the training set with the KeBABS function `kbsvm`. In its simplest form, the function expects the training set, the labels for the training samples, the kernel object and the SVM together with SVM parameters. The meaning of most other parameters will be introduced later in this manual. In this example, we perform classification using the C-SVM from package `e1071` with a value 15 for the cost hyperparameter. The `kebabs` SVM name is `C-svc` which stands for "C support vector classification". The hyperparameter name `cost` is the unified `kebabs` name for the cost parameter called `C` in package `kernlab` and `cost` in the packages `e1071` and `LiblineaR`. The function `kbsvm` returns the trained model.

```
> model <- kbsvm(x=enhancerFB[train], y=yFB[train], kernel=specK2,
+                pkg="e1071", svm="C-svc", cost=15)
```

Because of the large number of possible parameters in `kbsvm` the parameter names are included for each parameter in the function.

Now we have trained a linear model that separates the sequences for which transcription factor binding occurs from sequences without binding. The model parameters and all information necessary for prediction is stored in the model object. A description of the KeBABS model object is given in section 6. Once the model is trained, it can be used to predict whether EP300 binding occurs for new sequences in the test set with

```
> pred <- predict(model, enhancerFB[test])
> head(pred)

[1] "1"  "1"  "1"  "1"  "-1" "1"

> head(yFB[test])

[1]  1 -1 -1  1 -1 -1
```

The vector `pred` contains the predictions for the samples in the test set. The true values for the test set are shown in the last output above. With the knowledge of the true labels for the test set we can check the quality of our predictions, i.e. the performance of the model for the sequences in the test set, with

```
> evaluatePrediction(pred, yFB[test], allLabels=unique(yFB))

    1 -1
1  50 20
-1 21 59

Accuracy:              72.667% (109 of 150)
Balanced accuracy:     72.553% (50 of 71 and 59 of 79)
Matthews CC:            0.451

Sensitivity:           70.423% (50 of 71)
Specificity:           74.684% (59 of 79)
Precision:             71.429% (50 of 70)
```

The accuracy shows the percentage of the samples in the test set that were predicted correctly. For balanced datasets, the accuracy is the main performance criterion. For unbalanced datasets, the balanced accuracy or the Matthews Correlation Coefficient (Gorodkin, 2004) might be more appropriate. The confusion matrix in the top part of the output shows the predictions as rows and the correct labels as columns. The values on the diagonal are the correct predictions for the two classes for which prediction and true label matches.

For comparison, we now run the same training with the gappy pair kernel which uses pairs of substrings of length k separated by an arbitrary substring with a length up to m. For details on this kernel see Section 4.1.3.

```
> gappyK1M3  <- gappyPairKernel(k=1, m=3)
> model <- kbsvm(x=enhancerFB[train], y=yFB[train],
+                kernel=gappyK1M3, pkg="e1071", svm="C-svc",
+                cost=15)
> pred <- predict(model, enhancerFB[test])
> evaluatePrediction(pred, yFB[test], allLabels=unique(yFB))

    1 -1
1  51 12
-1 20 67

Accuracy:              78.667% (118 of 150)
Balanced accuracy:     78.321% (51 of 71 and 67 of 79)
Matthews CC:            0.573

Sensitivity:           71.831% (51 of 71)
Specificity:           84.810% (67 of 79)
Precision:             80.952% (51 of 63)
```

If we want to switch to the C-SVM implementation in a different package, e.g. `LiblineaR`, only the package parameter is changed. KeBABS internally converts the data structures and parameters to the package specific format before calling the SVM implementation in the selected package.

```
> gappyK1M3  <- gappyPairKernel(k=1, m=3)
> model <- kbsvm(x=enhancerFB[train], y=yFB[train],
+                kernel=gappyK1M3, pkg="LiblineaR", svm="C-svc",
+                cost=15)
> pred <- predict(model, enhancerFB[test])
> evaluatePrediction(pred, yFB[test], allLabels=unique(yFB))

    1 -1
1  52 13
-1 19 66

Accuracy:              78.667% (118 of 150)
Balanced accuracy:     78.392% (52 of 71 and 66 of 79)
Matthews CC:            0.572

Sensitivity:           73.239% (52 of 71)
Specificity:           83.544% (66 of 79)
Precision:             80.000% (52 of 65)
```

The accuracy is nearly identical, but the confusion matrix has changed. This example shows that completely identical results cannot be expected from the same SVM in different packages. The results between `kernlab` and `e1071` are usually identical because the implementation is very similar.

We use now a different SVM, the L1 regularized L2 loss SVM from `LiblineaR`.

```
> gappyK1M3  <- gappyPairKernel(k=1, m=3)
> model <- kbsvm(x=enhancerFB[train], y=yFB[train],
+                kernel=gappyK1M3, pkg="LiblineaR", svm="l1rl2l-svc",
+                cost=5)
> pred <- predict(model, enhancerFB[test])
> evaluatePrediction(pred, yFB[test], allLabels=unique(yFB))

    1 -1
1  52 15
-1 19 64

Accuracy:              77.333% (116 of 150)
Balanced accuracy:     77.126% (52 of 71 and 64 of 79)
Matthews CC:            0.545

Sensitivity:           73.239% (52 of 71)
Specificity:           81.013% (64 of 79)
Precision:             77.612% (52 of 67)
```

We have changed the value of the hyperparameter `cost` as we switched to a different SVM. Hyperparameter values are dependent on the actual SVM formulation and the best performing

values can be quite different between SVMs. Cross validation and grid search functions described in Section 7 help to select good hyperparameter values. A description of the available SVMs in each package and the hyperparameters of each SVM can be found on the help page for the function `kbsvm` which is shown with

```
> ?kbsvm
```

Please do not get irritated when looking at the large number of parameters in the help page for this function. As we have seen in this example only a small subset of the parameters is needed for simple training. The other parameters are meant to tune the training behavior for specific situations or for more advanced usage scenarios. Most of them will be introduced in later sections in this manual and you will become familiar with them quickly as you start using the package.

The next two sections describe the available sequence kernels in KeBABS and the basic data representations in detail. Starting with Section 6 important aspects of SVM based learning will be presented.

4 Sequence Kernels

The sequence kernels defined in this package are targeting biological sequences and represent different similarity measures between two sequences. All of the kernels in this package split the sequence into patterns. These patterns are also called features in the context of machine learning. The difference between the kernels lies primarily in the way in which this splitting occurs.

Described in a more general form the sequence kernels extract the sequence characteristics related to the features defined by the kernel and compute a similarity value between two sequences from their feature characteristics. In more specific mathematical terminology the kernel maps the sequences into a high dimensional vector space where the individual feature values are the coordinates in this vector space. The kernel value is computed through the dot product of the feature vectors representing the two sequences in high dimensional space.

The sequence kernels in KeBABS come in several flavors:

position-independent which means that the location of a pattern in the sequence is irrelevant for determination of the similarity value, just its occurrence has an influence. This is the“traditional” type of kernel used for sequences.

position-dependent which in addition to the occurrence also take the position into account. This version again is split into two different variants:

- **position-specific** kernels which only consider patterns when they are located at the same position in the two sequences
- **distance-weighted** kernels for which the contribution of a pattern is defined by the weighted distance (measured in positions) of the pattern in the two sequences

Through the introduction of a parameterized distance weighting function a flexible blending between the two extremes of exact position evaluation in the position-specific kernel and complete position independency in the position-independent kernel is possible.

annotation-specific kernels which use annotation information in addition to the patterns in the sequence. The annotation consists of a character string with the same length as the sequence which is aligned to the sequence and delivers for each pattern in the sequence a corresponding pattern in the annotation. For the annotation specific kernel, the feature consists of the concatenation of sequence pattern and annotation pattern. This means that annotation information is used together with the sequence information in the similarity consideration. One typical example is the splitting of gene sequences into intron and exon parts. The annotation specific kernel is available position-independent only.

All KeBABS kernels except the mismatch kernel support all flavors. For the mismatch kernel only the position-independent version without annotation is implemented.

First the position-independent kernel variants are described. The position-dependent variants are presented in Section 4.2. The annotation-specific kernel variant is covered in Section 4.3. The next sections describe the characteristics and the kernel parameters of the different kernels. The generation of a kernel matrix for a specific kernel is described in Section 5.1. A description of the generation of the explicit representation is given in Section 5.2.

4.1 Position-Independent Sequence Kernels

For the position-independent kernel the contribution of a single feature is dependent on the number of occurrences of the feature in the two sequences but independent from their location. The unnormalized kernel for position-independent sequence kernels can generally be described as

$$k(x, y) = \sum_{m \in \mathcal{M}} N(m, x) \cdot N(m, y) \tag{1}$$

where x and y are two sequences, m is one pattern of the feature space $\mathcal{M}$ which contains all possible patterns and $N(m, x)$ is the number of occurrences of pattern m in sequence x. $|\mathcal{M}|$ is the size of this feature space.

A formal definition of $N(m, x)$ which opens up the way to position-dependent kernels is given by

$$N(m, x) = \sum_{p=1}^{length(x)} \mathbf{1}(m, x, p) \tag{2}$$

where $\mathbf{1}(m, x, p)$ is a kernel specific indicator function with a value of 1 if the pattern m matches the sequence x at position p.

Then the feature mapping φ is defined as

$$\varphi(x) = ((N(m, x)))_{m \in \mathcal{M}} \tag{3}$$

i.e., for position-independent kernels, the feature vector contains the counts for the different patterns found in the sequence as vector elements.

Through kernel normalization, feature vectors are scaled to the unit sphere and the similarity value represents the cosine similarity which is just the cosine of the angle between feature vectors in the high-dimensional space. The kernel normalization is performed according to

$$k'(x, y) = \frac{k(x, y)}{\sqrt{k(x, x) \cdot k(y, y)}} \tag{4}$$

4.1.1 Spectrum Kernel

For the spectrum kernel (Leslie *et al.*, 2002) with kernel parameter `k`, the features are all substrings of fixed length k. The feature space is defined through all possible substrings of length k for the given alphabet $\mathcal{A}$. If $|\mathcal{A}|$ is the number of characters in the character set, the feature space dimension is $|\mathcal{M}| = |\mathcal{A}|^k$.

For DNA-sequences with the exact alphabet $\mathcal{A} = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ or RNA-sequences with the exact alphabet $\mathcal{A} = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{U}\}$ the alphabet size $|\mathcal{A}|$ is 4, for amino acid sequences with the exact alphabet it is 21. In addition to the 20 standard amino acids also the 21^{st} amino acid Selenocystein with the abbreviation U or Sec is included in the alphabet, as it occurs throughout all kingdoms and also in 25 human proteins. The feature space grows exponentially with k.

Usually `kebabs` only considers the k-mers that actually occur in sequences to keep performance and memory needs as low as possible, but please be aware that both the runtime and memory usage increase for kernel processing as k becomes larger because of the drastically increasing number of features. This effect becomes more prominent for larger alphabet sizes. For amino acids with $k = 5$ the feature space size is already 4,084,101.

The spectrum kernel walks along the sequence position by position and counts the number of occurrences of each k-mer in the sequence. The kernel first determines the $N(m, x)$ for all k-mers m in the sequences x and y and then computes the kernel value of the unnormalized kernel according to the formula which is the dot product of the two feature vectors for the sequences x and y. Through applying kernel normalization according to equation (4), the normalized kernel value is computed from the unnormalized kernel values .

As already shown in Section 3, a kernel object for the spectrum kernel is created with

```
> specK2 <- spectrumKernel(k=2)
```

This represents a normalized kernel for the spectrum kernel with k-mer length 2 because the `normalized` parameter is set by default to `TRUE`. The kernel object for the unnormalized kernel is created with

```
> specK2 <- spectrumKernel(k=2, normalized=FALSE)
```

When working with sequences of different lengths the normalized kernel usually leads to better results. Kernel objects are used for training as already shown in Section 3 , invisibly for the user, also in prediction and for explicit creation of a kernel matrix or an explicit representation.

With the kernel parameter `exact` set to `TRUE` the exact character set without ambiguous characters is used for the analysis. This is the default setting. Any character found in the sequence which is not in the character set is interpreted as invalid and leads to drop of open patterns and a restart of the pattern search with the next sequence position. When setting this parameter to `FALSE` the IUPAC character set with exact matching on each character of the character set is used. The IUPAC character set defines characters with ambiguous meaning in addition to A, C, G and T.

The kernel parameter `ignoreLower` with default setting `TRUE` defines that lower case characters in the sequence should be ignored and treated like invalid characters described above. When the parameter is set to `FALSE` the kernel becomes case insensitive and treats lower case characters like upper-case characters. The classes `DNAStringSet, RNAStringSet` and `AAStringSet` derived from `XStringSet` do not store lowercase characters but convert them to uppercase automatically on creation. Lowercase characters can be stored in the `BioVector` derived classes `DNAVector, RNAVector` and `AAVector`. These classes are structured similar to the `XStringSet` derived classes and they are treated identically in KeBABS. The `BioVector` derived classes should be used only when lowercase characters are needed, e.g. for representing repeat regions, or in context with string kernels from package `kernlab`. In all other situations sequence data should be represented through `XStringSet` derived classes.

With the kernel parameter `presence` set to `TRUE`, only the presence of a pattern in the sequence is relevant for the similarity consideration not the actual number of occurrences.

Reverse complements of patterns are considered identical to the forward version of patterns when the kernel parameter `revComplement` is set to `TRUE`. For example, in the case of a spectrum kernel with `k=3`, the k-mer ACG and its reverse complement CGT are treated as one feature. The parameter can be helpful when analyzing interaction with double stranded DNA.

The parameter `r` allows exponentiation of the kernel value. Only values larger than 0 are allowed. This parameter can only be different from 1 when computing kernel matrices explicitly but not as part of a regular SVM training or when generating an explicit representation with this kernel. Please be aware that only integer values ensure positive definiteness of the resulting kernel matrix.

When similarity values of multiple kernels are combined to define a new similarity measure we speak of a mixture kernel. In the mixed spectrum kernel k-mers with multiple k-mer lengths are taken into account for the similarity value. Within KeBABS similarity values are only combined through weighted sums. The kernel parameter `mixCoef` is used to define the mixture coefficients of a mixture kernel of several spectrum kernels. The parameter must be a numeric vector with k elements. For instance, for $k = 3$ the spectrum kernel values for $k = 1$, $k = 2$ and $k = 3$ are computed and mixed with the three elements in `mixCoef` as mixture coefficients. If individual kernels components in the mixture kernel should not be included in the mixture the corresponding element in `mixCoef` is set to zero.

4.1.2 Mismatch Kernel

The mismatch kernel (Leslie *et al.*, 2003) is similar to the spectrum kernel. The kernel parameter `k` has the same meaning in the mismatch kernel as in the spectrum kernel. The mismatch kernel also extracts substrings of fixed length k. The kernel parameter `m` defines the maximum number of mismatches that are allowed for each k-mer. For example with $k = 2$ and $m = 1$ the occurrence of substring TA found in the sequence matches to the features AA, CA, GA, TA, TC, TG, TT with not more than one mismatch and therefore increases the feature count for these seven features. The feature space and the feature space dimension with $|\mathcal{M}| = |\mathcal{A}|^k$ is identical to the spectum kernel. The parameter `m` has no impact on feature space size.

A kernel object for the normalized mismatch kernel with $k = 3$ and $m = 1$ is created in the following way:

```
> mismK3M1 <- mismatchKernel(k=3, m=1)
```

Apart from the new kernel parameter `m`, the kernel parameters `k`, `r`, `normalized`, `exact`, `ignoreLower` and `presence` of the mismatch match kernel have the same functionality as described in Section 4.1.1 for the spectrum kernel. The position-dependent kernel variants, the annotation-specific variant and the mixture kernel are not supported for the mismatch kernel.

4.1.3 Gappy Pair Kernel

The gappy pair kernel (Kuksa *et al.*, 2008; Mahrenholz *et al.*, 2011) looks for pairs of k-mers separated by several wildcard positions in between for which matching occurs for every alphabet character. For this kernel the kernel parameter `m` has a different meaning than in the mismatch kernel. It defines the maximum number of wildcard positions. A single feature of the gappy pair kernel is structured in the following way

$$\underbrace{kmer1}_{k}\underbrace{.\ .\ .\ .\ .\ .\ .}_{0\leq i\leq m}\underbrace{kmer2}_{k}$$

An irrelevant position i can contain any character from the character set. In the feature names the irrelevant positions are marked as dots. For example, for the sequence CAGAT the gappy pair kernel with $k = 1$ and $m = 2$ finds pairs of monomers with zero to two irrelevant positions in between. i.e. it finds the features CA, C.G, C..A, AG, A.A, A..T, GA, G.T and AT. The gappy pair kernel object for $k = 1$ and $m = 2$ with normalization is created with:

```
> gappyK1M2 <- gappyPairKernel(k=1, m=2)
```

The feature space for the gappy pair kernel is $|\mathcal{M}| = (m + 1)|\mathcal{A}|^{2k}$. It is considerably larger than that of the spectrum or mismatch kernel with same values for `k`. Through the factor 2 in the exponent the dimensionality grows much faster with increasing k. For amino acids with $k = 2$ and $m = 10$ the feature space size is already larger than 2 million features. The feature space size grows exponentially with `k` and linearly with `m`, which means that through the parameter `m` the complexity of the kernel can be adjusted better than for other sequence kernels.

The other kernel parameters are identical to the spectrum kernel as described in section 4.1.1.

4.1.4 Motif Kernel

The motif kernel (Ben-Hur and Brutlag, 2003) is created from a pre-defined motif list. With this kernel the sequence analysis can be restricted to a user-defined set of features. The kernel also gives greater flexibility concerning the definition of individual features.

A single motif is described similar to a regular expression through a sequence of following elements:

- single character from the character set, e.g. C for DNA sequences, which matches only this character
- the wildcard character '.' that matches any character of the character set
- the definition of a substitution group, which is a list of single characters from the character set enclosed in square brackets - e.g. [CT] which matches any of the listed characters, in this example C or T. With a leading caret character in the list the meaning is inverted, i.e. with [ˆCT] only the not listed characters A and G would be matched for the exact DNA character set.

The motif collection must be defined as character vector which is passed to the constructor of the motif kernel object as in the following example for a normalized motif kernel:

```
> motifCollection1 <- c("A[CG]T","C.G","C..G.T","G[^A][AT]",
+                       "GT.A[CA].[CT]G")
> motif1 <- motifKernel(motifCollection1)
```

The feature space size of the motif kernel corresponds to the number of motifs in the motif collection. Here a motif kernel with a small collection of motifs is defined just to show some of the possibilities for pattern definition. In realistic scenarios the motif collection is usually much larger and could contain up to several hundred thousand motifs. Often motif collections are created from a subset of entries in a motif database.

Apart from kernel parameters `revComplement` and `mixCoef` which are not available for the motif kernel, the other kernel parameters are identical to the spectrum kernel as described in Section 4.1.1. The motif kernel can also be used to run sequence analysis with a subset of the features of a spectrum or gappy pair kernel.

Hint: For a normalized motif kernel with a feature subset of a normalized spectrum / gappy pair kernel the explicit representation will not be identical to the subset of an explicit representation for the normalized spectrum / gappy pair kernel because the motif kernel is not aware of the other k-mers which are used in the spectrum / gappy pair kernel additionally for normalization.

4.2 Position-Dependent Kernels

Position-dependent kernels take the position into account where a pattern occurs in two sequences. To ensure that positions in both sequences are comparable some kind of alignment is required for

all kinds of position-dependent kernels as prerequisite. In the simplest form, the kernel only considers patterns which are found at the same position for the similarity measure. This kernel is named as position-specific kernel. Patterns occur in two sequences at exactly the same position only if the sequences are evolutionary or functionally related and when no position-changing mutation occurred. Once a single insert or deletion happens, the pattern sequence with the same patterns at identical positions is disturbed. A more flexible approach is to define a neighborhood of a pattern in which to look for the same pattern on the other sequence. This neighborhood is defined in KeBABS through introduction of a parameterized distance weighting function. The following two subsections describe both kernel variants. More detailed information on the modeling of position dependency can be found in Bodenhofer *et al.* (2009).

4.2.1 Position-Specific Kernel

For the position-specific kernel, only patterns which occur at the same position in both sequences are relevant. The feature mapping $\varphi(x)$ for a sequence x of length L for this kernel is defined as

$$\varphi(x) = ((\mathbf{1}(m, x, 1), ..., \mathbf{1}(m, x, L)))_{m \in \mathcal{M}} \tag{5}$$

The feature vector is a binary vector containing for the part associated with one position a one for all patterns that occurs at this position position and zero otherwise. For the spectrum kernel and the mismatch kernel a single pattern occurs at each position, for the gappy pair kernel and the motif kernel multiple patterns could be relevant at a single position. The feature space of the position-specific version of a kernel is $|\mathcal{M}_{pos}| = L_m \cdot |\mathcal{M}_{nopos}|$ where L_m is the maximum sequence length in the sequence set and $|\mathcal{M}_{nopos}|$ is the feature space size of the corresponding position-independent kernel. Because of the high dimensionality of the feature space of position-specific kernels, an explicit representation is not supported for this kernel variant.

The kernel can be expressed with the kernel specific indicator function $\mathbf{1}(m, x, p)$ as

$$k(x, y) = \sum_{m \in \mathcal{M}} \sum_{p=1}^{L} \mathbf{1}(m, x, p) \cdot \mathbf{1}(m, y, p) \tag{6}$$

with L being the minimum length of the sequences x and y. The unnormalized kernel version just counts the occurrences where the same pattern at a position is found in both sequences.

The spectrum kernel, the gappy pair kernel, and the motif kernel can be used in the position-specific variant. For creating position-specific kernels, the parameter `distWeight` is set to 1 as in the following example of a position-specific, unnormalized gappy pair kernel:

```
> gappyK1M2ps <- gappyPairKernel(k=1, m=2, distWeight=1,
+                                normalized=FALSE)
```

The reason for using the parameter `distWeight` with position-specific kernels will become obvious in a few moments when the distance-weighted kernel is explained.

As mentioned above, the sequences analyzed with a position-specific kernel must be aligned in some way. Flexible alignment without changing the sequence set is possible through assignment of an offset vector as metadata to the sequence set. This vector gives the offset from the sequence start to position 1 for each sequence. The offset metadata is element metadata because it contains metadata for each element of the sequence set. For example for a set of amino acid sequences given as AAStringSet the offset vector is assigned with

```
> seq1 <- AAStringSet(c("GACGAGGACCGA","AGTAGCGAGGT","ACGAGGTCTTT",
+                       "GGACCGAGTCGAGG"))
> positionMetadata(seq1) <- c(3, 5, 2, 10)
```

The offset vector must contain positive or negative integer values and must have the same length as the sequence set. If position metadata is not assigned for a sequence set the position-specific kernel assumes that each sequence starts as position 1. In this case, the sequences must be stored aligned in the sequence set.

It should be mentioned that the weighted degree kernel is the mixture kernel for the position-specific spectrum kernel. In the following example, we use a weighted degree kernel in unnormalized form and create the kernel matrix with this kernel for the small sequence set given above

```
> wdK3 <- spectrumKernel(k=3, distWeight=1, mixCoef=c(0.5,0.33,0.17),
+                        normalized=FALSE)
> km <- getKernelMatrix(wdK3, seq1)
> km

An object of class "KernelMatrix"
      [,1]  [,2]  [,3]  [,4]
[1,] 11.33  1.00  5.83  4.83
[2,]  1.00 10.33  1.00  0.50
[3,]  5.83  1.00 10.33  4.33
[4,]  4.83  0.50  4.33 13.33
```

We delete the position metadata through assignment of NULL and recreate the kernel matrix without alignment of the sequences via an offset.

```
> positionMetadata(seq1) <- NULL
> km <- getKernelMatrix(wdK3, seq1)
> km

An object of class "KernelMatrix"
      [,1]  [,2]  [,3]  [,4]
[1,] 11.33  1.33  0.50  3.33
[2,]  1.33 10.33  2.33  0.50
[3,]  0.50  2.33 10.33  1.00
[4,]  3.33  0.50  1.00 13.33
```

4.2.2 Distance-Weighted Kernel

The distance-weighted kernel is the second variant of position-dependent kernels. This kernel also evaluates position information and alignment of the sequences is important here as well. However, instead of just considering patterns which are located at the same position in the two sequences, this kernel searches for pattern occurrences in the proximity. If the same pattern is found in the neighborhood on the second sequence the contribution of this pattern depends on the distance of the pattern on the two sequences. For flexible definition of the neighborhood characteristics a parameterized distance weighting function is introduced. The contribution of the occurrence of a pattern to the similarity measure is computed as weighted distance between the pattern positions.

The distance-weighted kernel is defined as

$$k(x,y) = \sum_{m \in \mathcal{M}} \sum_{p=1}^{L_x} \sum_{q=1}^{L_y} \mathbf{1}(m,x,p) \cdot E(p,q) \cdot \mathbf{1}(m,y,q) \tag{7}$$

with L_x and L_y being the length of the sequences x and y and $E(p,q)$ the distance weighting function for positions p and q. The position-specific kernel described in the previous subsection is a special case of this kernel with the following distance weighting function:

$$E_{\mathrm{ps}}(p,q) = \begin{cases} 1 & \text{if } p = q \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

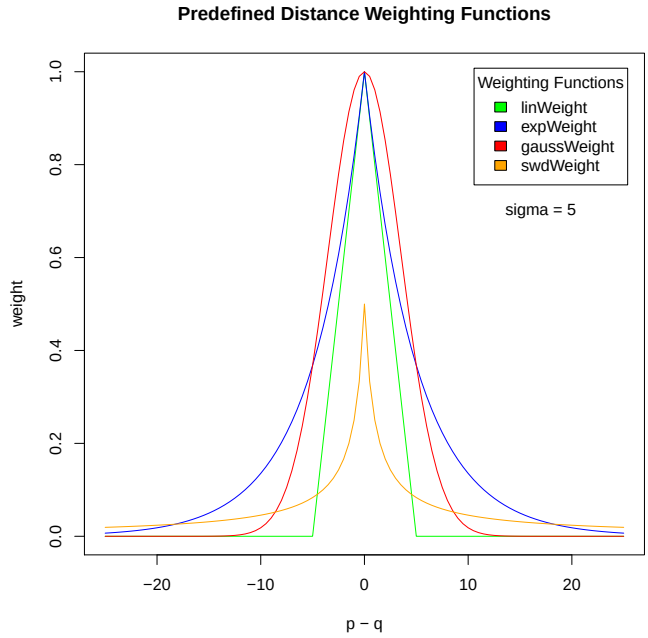
The position-independent kernel also is a special case of this kernel with the trivial distance weighting function $E_{\mathrm{pi}}(p,q) = 1$ for all $p \in [1,....,L_x]$, $q \in [1,....,L_y]$.

With a parameterized distance function, the neighborhood can be adjusted as needed. The package contains three pre-defined distance weighting functions, here defined as functions of the distance $d = |p - q|$

- linear distance weigthing:
  $E_{\mathrm{lin},\sigma}(d) = \max(1 - \frac{d}{\sigma}, 0)$
- exponential distance weigthing:
  $E_{\mathrm{exp},\sigma}(d) = \exp(-\frac{d}{\sigma})$
- gaussian distance weigthing:
  $E_{\mathrm{exp},\sigma}(d) = \exp(-\frac{d^2}{\sigma^2})$

In all three cases the parameter σ controls the proximity behavior with a strict cut off at sigma for linear distance weighting. For small sigma close to 0, the distance-weighted kernel becomes the position-specific kernel, for very large values of sigma the kernel becomes the position-independent kernel. The parameter sigma allows flexible blending between both extremes of strict position evaluation and complete position independency for all three distance weighting functions. Positive definiteness of the resulting kernel matrix is ensured for these functions for any parameter values. The following plot shows the three parameterized functions with parameter value 5 for sigma.

```
> curve(linWeight(x, sigma=5), from=-25, to=25, xlab="p - q", ylab="weight",
+       main="Predefined Distance Weighting Functions", col="green")
> curve(expWeight(x, sigma=5), from=-25, to=25, col="blue", add=TRUE)
> curve(gaussWeight(x, sigma=5), from=-25, to=25, col="red", add=TRUE)
> curve(swdWeight(x), from=-25, to=25, col="orange", add=TRUE)
> legend('topright', inset=0.03, title="Weighting Functions", c("linWeight",
+        "expWeight", "gaussWeight", "swdWeight"), fill=c("green", "blue",
+        "red", "orange"))
> text(17, 0.70, "sigma = 5")
```



The shifted weighted degree kernel (SWD) (Rätsch *et al.*, 2005) with equal position weighting is a special case of the distance-weighted mixture spectrum kernel with a distance weighting function of

$$E_{SWD}(d) = \frac{1}{2 * (|d| + 1)} \tag{9}$$

This weighting function is similar to exponential distance weighting for $\sigma = 2$ but has a lower peak and larger tails. For brevity input parameter checks are omitted.

```
> swdWeight <- function(d)
+ {
+     if (missing(d))
+         return(function(d) swdWeight(d))
```

```
+
+     1 / (2 * (abs(d) + 1))
+ }
> swdK3 <- spectrumKernel(k=3, distWeight=swdWeight(),
+                         mixCoef=c(0.5,0.33,0.17))
```

We load sequence data provided with the package, assign shorter sample names for display purpose and compute the kernel matrix for this kernel with user-defined distance weighting for the first few sequences to save runtime for vignette generation. Position-dependent kernels are considerably slower than their position-independent counterparts.

```
> data(TFBS)
> names(enhancerFB) <- paste("Sample", 1:length(enhancerFB), sep="_")
> enhancerFB
```

```
  A DNAStringSet instance of length 500
      width seq                                    names
  [1]   827 ACTAAACAACATCAATC...TAGGCAAAATCCTGACA Sample_1
  [2]   678 GAATATAGACCCTTGGT...AAGTTATATTAATTTAT Sample_2
  [3]   878 CACCCACATGGTGGCTC...TCCCCACTGTATCACTC Sample_3
  [4]   927 TGCCGTAGTGTGCCAGC...TGATTCTCCTGGATCAA Sample_4
  [5]   953 CTTCATATACCTATTAA...CCTAATTTAAAAAGGGG Sample_5
  ...   ... ...
[496]   478 ACGAGAATGAGGGAAAG...GTCTGCCTTCCGAGTGA Sample_496
[497]  1552 AGGGGACTGGAAGAAAG...TCTCTTCTAAGCGAGCA Sample_497
[498]   503 GATGAACGTAAAATGCA...TAATGACTCCCTTCTGA Sample_498
[499]  1577 ACCCCTCTGAGACCAAG...GTAGGTCTGTATTCCTG Sample_499
[500]   778 CACTGTCATAACTTGCT...GTAGAGTAGCTGCTCTC Sample_500
```

```
> kmSWD <- getKernelMatrix(swdK3, x=enhancerFB, selx=1:5)
> kmSWD[1:5,1:5]
```

```
An object of class "KernelMatrix"
          Sample_1  Sample_2  Sample_3  Sample_4  Sample_5
Sample_1 1.0000000 0.4907802 0.5557066 0.5391745 0.5444010
Sample_2 0.4907802 1.0000000 0.4844622 0.4563475 0.4740511
Sample_3 0.5557066 0.4844622 1.0000000 0.5794030 0.5548927
Sample_4 0.5391745 0.4563475 0.5794030 1.0000000 0.5452410
Sample_5 0.5444010 0.4740511 0.5548927 0.5452410 1.0000000
```

The function `swdWeight` was a simple example of a user-defined weighting function without parametrization. In fact it is also contained as predefined weighting function in the package, but was used here to show how a user defined function can be defined. Please note that the weighting function returns itself as function if the distance parameter `d` is missing. If parameters are used in a user defined weighting function it must be defined as closure for which all arguments except the

distance argument are fixed once the function is called to do the actual distance weighting. In this way, the function can be passed when creating a kernel just fixing the parameters without actually passing the distance data which is not available at this time. As an example we use the following weighting function

$$E_{SWD}(d, b) = b^{-|d|} \tag{10}$$

with the base `b` as parameter. This example is meant to show the definition of parameterized user-defined weighting functions and of course does not lead to anything new as it is just a scaled version of exponential weighting.

The new weighting function is defined as

```
> udWeight <- function(d, base=2)
+ {
+     if (missing(d))
+         return(function(d) udWeight(d, base=base))
+
+     return(base^(-d))
+ }
> specudK3 <- spectrumKernel(k=3, distWeight=udWeight(base=4),
+                                 mixCoef=c(0,0.3,0.7))
> kmud <- getKernelMatrix(specudK3, x=enhancerFB, selx=1:5)
```

User-defined distance functions must consider three aspects:

Definition as closure: When all parameters are passed to the function or set by default, but the distance argument is missing, the function must return a new unnamed function with the distance argument as single argument and all parameters set to their fixed or predefined value.

Vectorized processing: The function must be able to process numeric vectors containing multiple distances and return a numeric vector of weighted distances of identical length.

Positive definiteness: This means that the kernel matrix generated with this distance weighting function must be positive definite. One necessary, but not sufficient, precondition of positive definiteness is that the distance weighting function must be symmetric, i.e. $E(p, q) = E(q, p)$ must be fulfilled. For the example given above, the positive definiteness is ensured because it is only a scaled version of the package provided exponential weighting.

For all learning methods that require positive definite kernel matrices, especially the SVMs supported in this package the third aspect is very important. Please also note that explicit representations are not available for the position-dependent kernel variants.

4.3 Annotation Specific Kernel

The annotation specific kernel evaluates annotation information together with the patterns in the sequence. For example the similarity considerations for gene sequences could be split according to exonic and intronic parts of the sequences. This is achieved by aligning an annotation string which contains one annotation character for each sequence position to the sequence. Each pattern in the sequence concatenated with the corresponding pattern of the annotation characters is used as feature. In the abovementioned example of exon/intron annotation the feature space contains separate features for identical patterns in exon and intron regions.

The following small toy example shows how to assign exon/intron annotation to a DNA sequence set. A sequence set with just two sequences for very similar genes HBA1 and HBA2 from the human genome hg19 is generated. For this example the packages `BSgenome.Hsapiens.UCSC.hg19`, `TxDb.Hsapiens.UCSC.hg19.knownGene` and `BSgenome` must be installed. Each gene contains three exons. Again checks on input parameters are omitted.

```
> getGenesWithExonIntronAnnotation <- function(geneList, genomelib,
+                                              txlib)
+ {
+     library(BSgenome)
+     library(genomelib, character.only=TRUE)
+     library(txlib, character.only=TRUE)
+     genome <- getBSgenome(genomelib)
+     txdb <- eval(parse(text=txlib))
+     exonsByGene <- exonsBy(txdb, by ="gene")
+
+     ## generate exon/intron annotation
+     annot <- rep("", length(geneList))
+     geneRanges <- GRanges()
+     exonsSelGenes <- exonsByGene[geneList]
+
+     if (length(exonsSelGenes) != length(geneList))
+         stop("some genes are not found")
+
+     for (i in 1:length(geneList))
+     {
+         exons <- unlist(exonsSelGenes[i])
+         exonRanges <- ranges(exons)
+         chr <-  as.character(seqnames(exons)[1])
+         strand <- as.character(strand(exons)[1])
+         numExons <- length(width(exonRanges))
+
+         for (j in 1:numExons)
+         {
+             annot[i] <-
+                 paste(annot[i],
+                       paste(rep("e", width(exonRanges)[j]),
```

```
+                             collapse=""), sep="")
+
+             if (j < numExons)
+             {
+                 annot[i] <-
+                     paste(annot[i],
+                           paste(rep("i", start(exonRanges)[j+1] -
+                                 end(exonRanges)[j] - 1),
+                           collapse=""), sep="")
+             }
+         }
+
+         geneRanges <-
+             c(geneRanges,
+               GRanges(seqnames=Rle(chr),
+                       strand=Rle(strand(strand)),
+                       ranges=IRanges(start=start(exonRanges)[1],
+                                      end=end(exonRanges)[numExons])))
+     }
+
+     ## retrieve gene sequences
+     seqs <- getSeq(genome, geneRanges)
+     names(seqs) <- geneList
+     ## assign annotation
+     annotationMetadata(seqs, annCharset="ei") <- annot
+     seqs
+ }
> ## get gene sequences for HBA1 and HBA2 with exon/intron annotation
> ## 3039 and 3040 are the geneID values for HBA1 and HBA2
> hba <- getGenesWithExonIntronAnnotation(c("3039", "3040"),
+             "BSgenome.Hsapiens.UCSC.hg19",
+             "TxDb.Hsapiens.UCSC.hg19.knownGene")
```

For exon/intron annotation, we defined the annotation character set with character "e" denoting an exon position and "i" for an intron position. Up to 32 characters can be defined by the user in the annotation character set. The character "-" is used to mask sequence parts that should not be evaluated in the sequence kernel. It must not be used as regular annotation character defined in the annotation character set because with masking a different function is assigned already. The annotation strings are built up from the length of exons and introns for the two genes in the gene set. Annotation strings must have identical number of characters as the sequences. With the function `annotationMetadata` the annotation character set and the annotation strings are assigned to the sequence set. Individual sequences or sequence parts can be shown together with annotation information in the following way

```
> annotationCharset(hba)
> showAnnotatedSeq(hba, sel=1, start=1, end=400)
```

Annotation metadata can be deleted for the sequence set through assignment of NULL. Now we generate an explicit representation with and without annotation. First a kernel object for the standard and the annotation-specific version of the spectrum kernel is created. The annotation metadata is only evaluated in the kernel when the kernel parameter annSpec is set to TRUE, i.e. for the second kernel object which can be used for sequence analysis in the same way as the standard spectrum kernel object.

```
> specK2 <- spectrumKernel(k=2)
> specK2a <- spectrumKernel(k=2, annSpec=TRUE)
> erK2 <- getExRep(hba, specK2, sparse=FALSE)
> erK2[,1:6]
> erK2a <- getExRep(hba, specK2a, sparse=FALSE)
> erK2a[,1:6]
```

We create a kernel matrix from the explicit representations via the function linearKernel.

```
> km <- linearKernel(erK2)
> km
> kma <- linearKernel(erK2a)
> kma
```

For the normalized sequence kernels, the similarity values have a range from 0 to 1. For 1 the sequences are considered identical for the given similarity measure defined through the kernel. The similarity between HBA1 and HBA2 is close to 1 and the sequences are very similar related to dimer content. For the annotation specific kernel, the similarity between the sequences HBA1 and HBA2 decreases because the dimer content of exonic and intronic regions is considered separately.

The second example for annotation specific kernels predicts the oligomerization state of coiled coil proteins from sequence using a heptad annotation for the amino acid sequences. The sequence data available on the web page for the R package procoil contains amino acid sequences for which the formation of dimers or trimers is known. We first predict the oligomerization state from sequence alone. In a second step a heptad pattern placed along the sequence is used in addition as annotation information. The heptad characters describe position-specificas of the amino acids within the coiled coil structure.

The analysis is done with the gappy pair kernel, first without annotation and then using the annotation specific version of this kernel. The coiled coil data are 477 amino acid sequences with lengths between 8 and 123 amino acids. Roughly 80% of the sequences form dimers and only 20% trimers, which means that the data is very unbalanced. We will see how we can better deal with this unbalancedness. First the data is loaded, the sequences ccseq are stored in an AAStringSet, the annotation ccannot in a character vector and the labels yCC in a factor. The annotation is already assigned to the sequence set but to show the assignment it is deleted in the first step.

```
> data(CCoil)
> ccseq
```

```
  A AAStringSet instance of length 477
      width seq                                      names
  [1]    57 LRELQEALEEEVLTRQS...ARNRDLEAHVRQLQERM PDB1
  [2]    50 PLDISQNLAAVNKSLSD...DKIEEILSKIYHIENEI PDB2
  [3]    50 VMESFAVQNQNIDAQGE...QLILEALQGINKRLDNL PDB3
  [4]    47 SLVQAQTNARAIAAMKN...EGTQRLAIAVQAIQDHI PDB4
  [5]    46 LAGIVQQQQQLLDVVKR...KNLQTRVTAIEKYLKDQ PDB5
  ...   ... ...
[473]    11 AMKLVEKIREE                            PDB473
[474]    11 NLLILYDAIGT                            PDB474
[475]    11 EAACSAFATLE                            PDB475
[476]    10 LAVALHELAS                             PDB476
[477]   123 KKAAEDRSKQLEDELVS...QKDEEKMEIQEIQLKEA PDB477

> ccannot[1:3]

[1] "abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga"
[2] "abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga"
[3] "defgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd"

> head(yCC)

[1] TRIMER TRIMER TRIMER TRIMER TRIMER TRIMER
Levels: DIMER TRIMER

> yCC <- as.numeric(yCC)
> ## delete annotation metadata
> annotationMetadata(ccseq) <- NULL
> annotationMetadata(ccseq)

NULL

> gappy <- gappyPairKernel(k=1, m=10)
> train <- sample(1:length(ccseq), 0.8*length(ccseq))
> test <- c(1:length(ccseq))[-train]
> model <- kbsvm(ccseq[train], y=yCC[train], kernel=gappy,
+                pkg="LiblineaR", svm="C-svc", cost=100)
> pred <- predict(model, ccseq[test])
> evaluatePrediction(pred, yCC[test], allLabels=unique(yCC))

   2  1
2  4  5
1 18 69

Accuracy:              76.042% (73 of 96)
```

```
Balanced accuracy:     55.713% (4 of 22 and 69 of 74)
Matthews CC:            0.165

Sensitivity:           18.182% (4 of 22)
Specificity:           93.243% (69 of 74)
Precision:             44.444% (4 of 9)
```

The high accuracy is misleading because the larger class is distorting this value. Because of the unbalancedness of the data the balanced accuracy is the relevant performance measure here. Now annotation metadata is assigned to the sequence set and the same test is performed with the annotation specific gappy pair kernel. The annotation sequences are added as metadata to the sequence set together with the annotation character set specified as string containing all characters that appear in the character set. The "-" character must not be used in this character set because it is used for masking sequence parts with the annotation strings.

```
> ## assign annotation metadata
> annotationMetadata(ccseq, annCharset="abcdefg") <- ccannot
> annotationMetadata(ccseq)[1:5]

[1] "abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga"
[2] "abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga"
[3] "defgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd"
[4] "defgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga"
[5] "abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd"

> annotationCharset(ccseq)

[1] "abcdefg"

> showAnnotatedSeq(ccseq, sel=2)

Sequence PDB2 of class AAStringSet

Annotation character set: abcdefg

        0         1         2         3         4
        1         1         1         1         1
 [1]    PLDISQNLAAVNKSLSDALQHLAQSDTYLSAIEDKIEEILSKIYHIENEI
        abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga

> gappya <- gappyPairKernel(k=1, m=10, annSpec=TRUE)
> model <- kbsvm(ccseq[train], y=yCC[train], kernel=gappya,
+                pkg="LiblineaR", svm="C-svc", cost=100)
> pred <- predict(model, ccseq[test])
> evaluatePrediction(pred, yCC[test], allLabels=unique(yCC))
```

```
   2  1
2  3  2
1 19 72

Accuracy:              78.125% (75 of 96)
Balanced accuracy:     55.467% (3 of 22 and 72 of 74)
Matthews CC:            0.207

Sensitivity:           13.636% (3 of 22)
Specificity:           97.297% (72 of 74)
Precision:             60.000% (3 of 5)
```

We see that the balanced accuracy decreases which could indicate that the annotation specific kernel with annotation performs worse than the kernel without annotation, but we have silently assumed that the same cost value should be used for the annotated gappy pair kernel. The comparison of different kernels with the same hyperparameter values is usually not adequate because the best performing hyperparameter value might be quite different for the two kernels. Hyperparameter selection must be performed individually for each kernel. Please also note that comparing kernels should be done based on multiple training runs on independent data or cross validation and not from a single training run. In the following example grid search (see section 7.2) is performed with both kernels at the same time with a kernel list and a vector of values for the `cost` hyperparameter using 5-fold cross validation for each grid point. With the parameter `perfObjective` the balanced accuracy is set as objective for the parameter search. Multiple performance parameters (accuracy, balanced accuracy and Matthews correlation coefficient) are recorded for each grid point when the parameter `perfParameters` is set to `"ALL"`.

```
> ## grid search with two kernels and 6 hyperparameter values
> ## using the balanced accuracy as performance objective
> model <- kbsvm(ccseq[train], y=yCC[train],
+                kernel=c(gappy, gappya), pkg="LiblineaR", svm="C-svc",
+                cost=c(1,10,50,100,200,500), explicit="yes", cross=5,
+                perfParameters="ALL", perfObjective="BACC",
+                showProgress=TRUE)
> result <- modelSelResult(model)
> result

Grid search result object of class  "ModelSelectionResult"

cross                 : 5
noCross               : 1
Best bal. accuracy    : 0.68472821

Grid Rows:
Kernel_1    GappyPairKernel: k=1, m=10
Kernel_2    GappyPairKernel: k=1, m=10, annSpec=TRUE
```

```
Grid Columns:
          cost
GridCol_1    1
GridCol_2   10
GridCol_3   50
GridCol_4  100
GridCol_5  200
GridCol_6  500

Grid Balanced Accuracies:

         GridCol_1 GridCol_2 GridCol_3 GridCol_4 GridCol_5 GridCol_6
Kernel_1 0.5643137 0.6555972 0.6670449 0.6613445 0.6833658 0.6843915
Kernel_2 0.6113568 0.6682231 0.6718697 0.6847282 0.6529720 0.6692549

Selected Grid Row:
GappyPairKernel: k=1, m=10, annSpec=TRUE

Selected Grid Column:
          cost
GridCol_4  100
```

In grid search with balanced accuracy as performance objective, the annotated gappy pair kernel shows better performance. The detailed performance information for each grid point can be retrieved in the following way:

```
> perfData <- performance(result)
> perfData

$ACC
         GridCol_1 GridCol_2 GridCol_3 GridCol_4 GridCol_5 GridCol_6
Kernel_1 0.8397471 0.8450444 0.8556391 0.8607656 0.8687286 0.8556049
Kernel_2 0.8056391 0.8371839 0.8107314 0.8216336 0.8107656 0.8134997

$BACC
         GridCol_1 GridCol_2 GridCol_3 GridCol_4 GridCol_5 GridCol_6
Kernel_1 0.5643137 0.6555972 0.6670449 0.6613445 0.6833658 0.6843915
Kernel_2 0.6113568 0.6682231 0.6718697 0.6847282 0.6529720 0.6692549

$MCC
         GridCol_1 GridCol_2 GridCol_3 GridCol_4 GridCol_5 GridCol_6
Kernel_1 0.2926806 0.3950426 0.4290534 0.4547895 0.4874316 0.4550451
Kernel_2 0.3694618 0.4406164 0.4424624 0.4767670 0.4438935 0.4233070

> which(perfData$BACC[1,] == max(perfData$BACC[1,]))
```

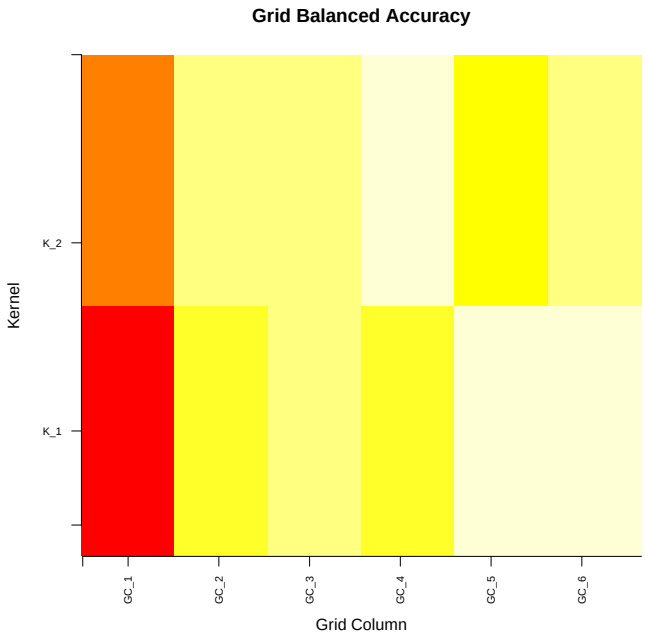
```
GridCol_6
        6
```

```
> which(perfData$BACC[2,] == max(perfData$BACC[2,]))
```

```
GridCol_4
        4
```

We see from the balanced accuracy of both kernels that the balanced accuracy of the annotated gappy pair kernel is often higher than that of the kernel without annotation. The best performing hyper parameter value is different for both kernels. This example also shows clearly that the accuracy as performance measure does not help at all. Also the Matthews correlation coefficient can be selected as performance objective for unbalanced datasets and would have delivered similar results. The best value for MCC is observed for the annotated gappy pair kernel for the same cost value of 30. Finally we can plot the grid performance data for one performance measure. This is helpful especially in situations where multiple kernels and a larger number of hyperparameter values should be compared. Then kernels and regions of good or worse performance can be identified more easily.

```
> plot(result, sel="BACC")
```



The color range in this plot is similar to heatmaps. The annotated kernel, which is `K_2` in the plot, shows higher balanced accuracy in the grid search than the gappy pair kernel without annotation for several values of the hyperparameter.

Through user-definable annotation the sequence kernels provide great flexibility for including additional sequence related information in kernel processing. As shown in this example the prediction of coiled-coil interactions in proteins as proposed in Mahrenholz *et al.* (2011) and described in the vignette of the R package `procoil`[5] can be performed with KeBABS in a faster and more flexible way. The heptad information is assigned as sequence annotation to the amino acid sequences. The annotation specific gappy pair kernel can be used instead of the `PrOCoil` kernel with regular and irregular heptad patterns.

4.4 List of Spectrum Kernel Variants

In this section the different variants are recapitulated in an exemplary manner for the spectrum kernel. Here only the normalized version is listed. All kernels can be used unnormalized as well.

```
> ## position-independent spectrum kernel normalized
> specK2 <- spectrumKernel(k=3)
> #
> ## annotation specific spectrum normalized
> specK2a <- spectrumKernel(k=3, annSpec=TRUE)
> #
> ## spectrum kernel with presence normalized
> specK2p <- spectrumKernel(k=3, presence=TRUE)
> #
> ## mixed spectrum normalized
> specK2m <- spectrumKernel(k=3, mixCoef=c(0.5, 0.33, 0.17))
> #
> ## position-specific spectrum normalized
> specK2ps <- spectrumKernel(k=3, distWeight=1)
> #
> ## mixed position-specific spectrum kernel normalized
> ## also called weighted degree kernel normalized
> specK2wd <- spectrumKernel(k=3, dist=1,
+                            mixCoef=c(0.5, 0.33, 0.17))
> #
> ## distance-weighted spectrum normalized
> specK2lin <- spectrumKernel(k=3, distWeight=linWeight(sigma=10))
> specK2exp <- spectrumKernel(k=3, distWeight=expWeight(sigma=10))
> specK2gs <- spectrumKernel(k=3, distWeight=gaussWeight(sigma=10))
> #
> ## shifted weighted degree with equal position weighting normalized
> specK2swd <- spectrumKernel(k=3, distWeight=swdWeight(),
+                             mixCoef=c(0.5,0.33,0.17))
> #
> ## distance-weighted spectrum kernel with user defined distance
> ## weighting
```

[5] http://www.bioconductor.org/packages/release/bioc/html/procoil.html

```
> udWeight <- function(d, base=2)
+ {
+     if (!(is.numeric(base) && length(base==1)))
+         stop("parameter 'base' must be a single numeric value\n")
+
+     if (missing(d))
+         return(function(d) udWeight(d, base=base))
+
+     if (!is.numeric(d))
+         stop("'d' must be a numeric vector\n")
+
+     return(base^(-d))
+ }
> specK2ud <- spectrumKernel(k=3, distWeight=udWeight(b=2))
```

4.5 Kernel Lists

For grid search (see section 7.2) and model selection (see section 7.3) often multiple kernels are relevant. In such cases a list of kernel objects can be passed to `kbsvm` instead of a single kernel object. Kernel lists are either created implicitly through passing a vector for a single kernel parameter to the kernel constructor or explicitly through the user by putting multiple kernel objects into a list. In the following example we show the first variant in a grid search with hyperparameter `cost`. The output shows the cross validation error for 5-fold cross validation for each combination of kernel and hyperparameter cost.

```
> specK25 <- spectrumKernel(k=2:5)
> specK25

[[1]]
Spectrum Kernel: k=2

[[2]]
Spectrum Kernel: k=3

[[3]]
Spectrum Kernel: k=4

[[4]]
Spectrum Kernel: k=5

> train <- 1:100
> model <- kbsvm(x=enhancerFB[train], y=yFB[train],
+                kernel=specK25, pkg="LiblineaR", svm="C-svc",
+                cost=c(1,5,10,20,50,100), cross=5, explicit="yes",
+                showProgress=TRUE)
> modelSelResult(model)
```

```
Grid search result object of class  "ModelSelectionResult"

cross                 : 5
noCross               : 1
Smallest CV error     : 0.22

Grid Rows:
Kernel_1    SpectrumKernel: k=2
Kernel_2    SpectrumKernel: k=3
Kernel_3    SpectrumKernel: k=4
Kernel_4    SpectrumKernel: k=5

Grid Columns:
          cost
GridCol_1    1
GridCol_2    5
GridCol_3   10
GridCol_4   20
GridCol_5   50
GridCol_6  100

Grid Errors:

         GridCol_1 GridCol_2 GridCol_3 GridCol_4 GridCol_5 GridCol_6
Kernel_1      0.45      0.36      0.32      0.38      0.36      0.37
Kernel_2      0.35      0.29      0.26      0.31      0.32      0.34
Kernel_3      0.29      0.29      0.30      0.25      0.24      0.22
Kernel_4      0.32      0.27      0.26      0.23      0.30      0.31

Selected Grid Row:
SpectrumKernel: k=4

Selected Grid Column:
          cost
GridCol_6  100
```

A value range is supported in the spectrum kernel for the kernel parameter `k` and in the gappy pair kernel for the parameters `k` and `m`. The range of `k` for the gappy pair kernel is limited because of the exponential decrease of the feature space with an additional factor 2. Kernel lists can also be created by putting multiple kernel objects into a list, for example

```
> kernelList1 <- list(spectrumKernel(k=3), mismatchKernel(k=3,m=1),
+                     gappyPairKernel(k=2,m=4))
```

or by concatenating existing kernel lists

```
> kernelList2 <- c(spectrumKernel(k=2:4), gappyPairKernel(k=1, m=2:5))
```

Apart from grid search and model selection kernel lists cannot be used instead of single kernel objects.

5 Data Representations

For SVM based learning, the training and prediction data can either be presented in the form of a numeric kernel matrix or numeric data matrix in which the rows represent the data samples with the features as columns. KeBABS kernels support the generation of both data structures. For position-dependent sequence kernels, the explicit representation is not available. For the supported packages `kernlab`, `e1071` and `LiblineaR` learning via explicit representation is possible. The learning via kernel matrix is currently only available with package `kernlab`.

The generation of these data structures is usually performed internally by KeBABS as part of the `kbsvm` function invisible for the user. For very large datasets, the data structures can be precomputed once and saved to file to avoid repeated computation. For precomputing or usage in other machine learning methods or R packages not supported by KeBABS the following sections give a short overview about both data structures and describes how they can be created.

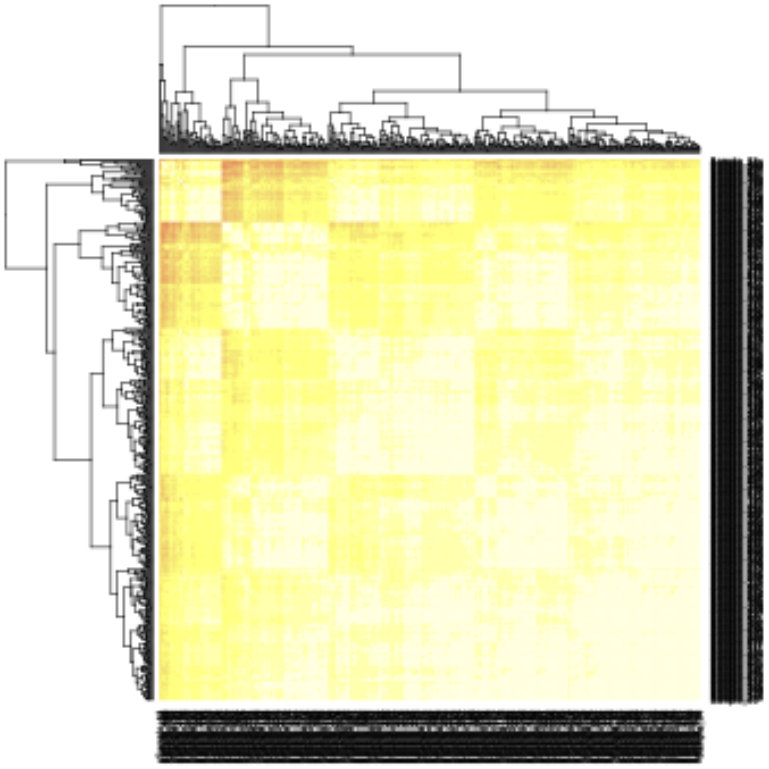
5.1 Kernel Matrix

The kernel matrix contains similarity values between pairs of sequences computed with the kernel. One matrix element describes the similarity between the samples represented by the corresponding row name and column name. Within KeBABS kernel matrices are stored in the class `KernelMatrix` which is essentially a lightweight wrapper around the standard dense matrix class `matrix`. KeBABS supports only dense kernel matrices. For usage in other R packages, the kernel matrix must be converted to the format needed in the respective package. A kernel matrix can be created in two ways. We have already seen the first of them in previous sections.

```
> specK2 <- spectrumKernel(k=2)
> km <- getKernelMatrix(specK2, x=enhancerFB)
> class(km)

[1] "KernelMatrix"
attr(,"package")
[1] "kebabs"

> dim(km)

[1] 500 500

> km[1:3, 1:3]

An object of class "KernelMatrix"
          Sample_1  Sample_2  Sample_3
Sample_1 1.0000000 0.8797649 0.9450536
Sample_2 0.8797649 1.0000000 0.9176770
Sample_3 0.9450536 0.9176770 1.0000000
```

If the argument y is not specified in the call of getKernelMatrix, a quadratic and symmetrical kernel matrix with all samples against all samples is created. For normalized sequence kernels, the range of kernel values in the matrix is between 0 and 1, for unnormalized kernels the values are larger than or equal to 0. A heatmap of the kernel matrix showing a few clusters is plotted with

```
> heatmap(km, symm=TRUE)
```



An alternative way to generate the kernel matrix through direct invocation of the kernel matrix function via the kernel object is shown below.

```
> specK2 <- spectrumKernel(k=2)
> km <- specK2(x=enhancerFB)
> km[1:3, 1:3]

An object of class "KernelMatrix"
          Sample_1  Sample_2  Sample_3
Sample_1 1.0000000 0.8797649 0.9450536
Sample_2 0.8797649 1.0000000 0.9176770
Sample_3 0.9450536 0.9176770 1.0000000
```

The kernel matrix can also be generated for a subset of the sequences without subsetting of the sequence set through using the parameter selx. In the following example we use only 5 sequences

```
> km <- getKernelMatrix(specK2, x=enhancerFB, selx=c(1,4,25,137,300))
> km
```

```
An object of class "KernelMatrix"
            Sample_1  Sample_4 Sample_25 Sample_137 Sample_300
Sample_1   1.0000000 0.9598054 0.9664655  0.9761818  0.9263253
Sample_4   0.9598054 1.0000000 0.9446113  0.9322925  0.9128547
Sample_25  0.9664655 0.9446113 1.0000000  0.9700230  0.9244838
Sample_137 0.9761818 0.9322925 0.9700230  1.0000000  0.9448881
Sample_300 0.9263253 0.9128547 0.9244838  0.9448881  1.0000000
```

In the following example, we create a rectangular kernel matrix with similarity values between two sets of sequences. First we create two sets of sequences from the package provided sequence set and then the kernel matrix

```
> seqs1 <- enhancerFB[1:200]
> seqs2 <- enhancerFB[201:500]
> km <- getKernelMatrix(specK2, x=seqs1, y=seqs2)
> dim(km)
```

```
[1] 200 300
```

```
> km[1:4,1:5]
```

```
An object of class "KernelMatrix"
         Sample_201 Sample_202 Sample_203 Sample_204 Sample_205
Sample_1  0.9204162  0.9641722  0.9577941  0.9505347  0.9202454
Sample_2  0.8945181  0.8964983  0.9646127  0.9334171  0.8897708
Sample_3  0.9087309  0.9304040  0.9660947  0.9661688  0.9223802
Sample_4  0.9097372  0.9522583  0.9887571  0.9641908  0.9108740
```

or the same rectangular kernel matrix via parameters `selx` and `sely`

```
> km <- getKernelMatrix(specK2, x=enhancerFB, selx=1:200,
+                       y=enhancerFB, sely=201:500)
> dim(km)
```

```
[1] 200 300
```

As mentioned above for normal training and prediction in KeBABS you do not need to compute the kernel matrices explicitly because this is done automatically.

5.2 Explicit Representation

The explicit representation contains the feature counts for all features occurring in the samples. By default, any feature from the feature space which is not present in the sample set is not listed in the explicit representation. From the sequence set an explicit representations can be generated in sparse or dense matrix for all position-independent kernels.

5.2.1 Linear Explicit Representation

The explicit representation is stored in an object of class `ExplicitRepresentationSparse` or `ExplicitRepresentationDense`. Apart from the feature counts, it also contains the kernel object used for generating the explicit representation and a flag that indicates whether it is a linear or quadratic explicit representation. The feature count matrix can be extracted with a standard coercion to type `matrix` if needed. In the following example, we generate an explicit representation for the first five package-provided sequences for the unnormalized spectrum kernel with $k = 2$ in dense format.

```
> specK2 <- spectrumKernel(k=2, normalized=FALSE)
> erd <- getExRep(enhancerFB, selx=1:5, kernel=specK2, sparse=FALSE)
> erd

Dense explicit representation of class  "ExplicitRepresentationDense"

Quadratic             : FALSE
Used kernel           :
   Spectrum Kernel: k=2, normalized=FALSE

          AA AC AG AT CA CC CG CT GA GC GG GT TA TC TG TT
Sample_1 106 37 77 48 68 54  9 66 49 47 24 38 45 59 48 51
Sample_2  26 23 33 37 40 57 13 79 22 46 46 36 31 63 57 68
Sample_3  59 65 59 57 98 75 10 63 49 50 42 41 34 56 71 48
Sample_4  69 47 70 46 80 73 17 82 56 52 58 40 28 80 61 67
Sample_5 105 47 81 57 69 49  3 75 64 37 62 39 52 62 57 93
```

Because of the short k-mer length all features are present in all sequences. If `k` is increased to 6, the picture changes.

```
> specK6 <- spectrumKernel(k=6, normalized=FALSE)
> erd <- getExRep(enhancerFB, selx=1:5, kernel=specK6,
+                 sparse=FALSE)
> dim(erd)

[1]    5 2243

> erd[,1:6]
```

```
Dense explicit representation of class  "ExplicitRepresentationDense"

Quadratic             : FALSE
Used kernel           :
   Spectrum Kernel: k=6, normalized=FALSE

         AAAAAA AAAAAC AAAAAG AAAAAT AAAACA AAAACC
Sample_1      1      0      1      1      2      0
Sample_2      0      0      0      0      0      0
Sample_3      0      0      0      0      0      0
Sample_4      0      0      0      0      1      0
Sample_5      0      1      1      0      0      1
```

Here we see that most features occur in only a few sequences. To save storage space, features that are not present in any sample are removed from the dense explicit representation. In spite of this storage improvement a sparse storage format is much more efficient in this situation. A sparse explicit representation is created through changing parameter `sparse` to TRUE. Now we create the explicit representation for all sequences both in sparse and dense format to see the space saving of the sparse format.

```
> specK6 <- spectrumKernel(k=6, normalized=FALSE)
> erd <- getExRep(enhancerFB, kernel=specK6, sparse=FALSE)
> dim(erd)

[1]  500 4095

> object.size(erd)

16553012 bytes

> ers <- getExRep(enhancerFB, kernel=specK6, sparse=TRUE)
> dim(ers)

[1]  500 4095

> object.size(ers)

3903936 bytes

> ers[1:5, 1:6]

Sparse explicit representation of class  "ExplicitRepresentationSparse"

Quadratic             : FALSE
```

```
Used kernel           :
   Spectrum Kernel: k=6, normalized=FALSE

5 x 6 sparse Matrix of class "dgRMatrix"
         AAAAAA AAAAAC AAAAAG AAAAAT AAAACA AAAACC
Sample_1      1      .      1      1      2      .
Sample_2      .      .      .      .      .      .
Sample_3      .      .      .      .      .      .
Sample_4      .      .      .      .      1      .
Sample_5      .      1      1      .      .      1
```

The dense explicit representation is with around 16MB four times as large as the sparse explicit representation with 4MB. The factor of memory efficiency becomes more prominent when the feature space dimension and the number of sequences increase. The sparse format can also lead to considerable speedups in the learning. The SVMs in package `kernlab` do not support the sparse format for explicit representations, the SVMs in the other packages support both variants. KeBABS internally selects the appropriate format for the used SVM during training and prediction. With the parameter `explicitType` in `kbsvm` dense or sparse processing can be enforced.

When setting the kernel flag `zeroFeatures` zero feature columns remain in the explicit representation and the number of columns increases to the full feature space dimension. For sparse explicit representations, just the names of the features which do not occur are added when the flag is set. This could be necessary when using the explicit representation together with other software.

Please consider that for all kernels except the motif kernel the feature space dimension increases drastically with increasing kernel parameter `k` with an especially large increase for the gappy pair kernel. For the feature space size, the parameter `m` is not relevant in the mismatch kernel and the increase is not as drastical for `m` as for `k` for the gappyPairKernel allowing also larger values of m.

An explicit representation can be generated for a subset of the features in the feature space. This subset is specified as character vector in the `feature` argument of the function getExplicitRepresentation. For normalization all features of the feature space are used. Dependent on the setting of the flag `zeroFeatures`, features from this subset that do not occur in the sequences are included in the explicit representation or not. It should be mentioned that an explicit representation generated from a feature subset in a normalized spectrum kernel or normalized gappy pair kernel is not identical to an explicit representation generated from the same features in a normalized motif kernel because because the feature space and therefore the normalization is different in the two cases.

Finally a small example with clustering shows how a kernel matrix or an explicit representation generated with a KeBABS kernel can be used in other ML methods. We compute a kernel matrix for the 500 samples with the gappy pair kernel and use `APCluster` (Bodenhofer *et al.*, 2011). The package implements affinity propagation clustering to find cluster structures in the data from the similarity matrix. The package `apcluster`[6] must be installed for this example.

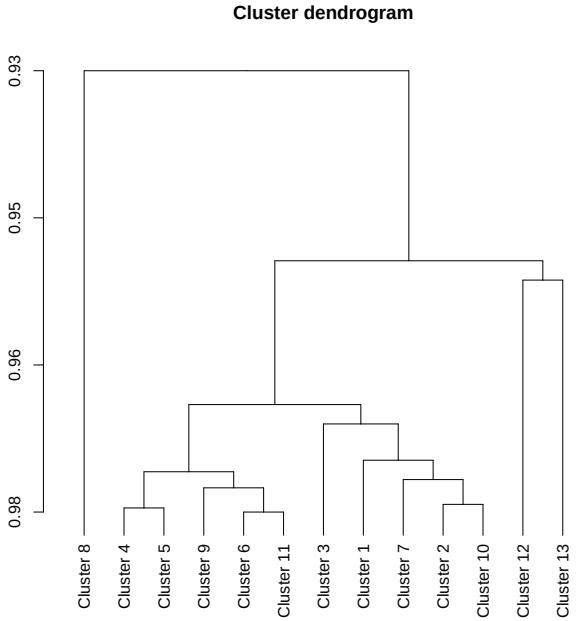
```
> library(apcluster)
> gappyK1M4 <- gappyPairKernel(k=1, m=4)
```

[6] `http://cran.r-project.org/package=apcluster`

```
> km <- getKernelMatrix(gappyK1M4, enhancerFB)
> apres <- apcluster(s=km, p=0.8)
> length(apres)

[1] 13
```

With the given setting of the preference parameter `p=0.8` thirteen clusters were found. The clustering result is stored in the result object `apres`. We perform agglomerative clustering starting from the previous clustering result to generate a cluster dendrogram.

```
> aggres <- aggExCluster(km, apres)
> plot(aggres)
```



The cluster dendrogram shows that either a two or a four cluster solution are quite stable with respect to the preference parameter. Also the explicit representation could have been used for clustering together with one of the similarity measures defined in `apcluster`. The simplest case of a linear kernel as similarity measure is identical to the clustering with the kernel matrix and with a changed preference value we get four clusters.

```
> exrep <- getExRep(enhancerFB, gappyK1M4, sparse=FALSE)
> apres1 <- apcluster(s=linearKernel, x=exrep, p=0.1)
> length(apres1)

[1] 4
```

5.2.2 Quadratic Explicit Representation

Learning with a quadratic kernel, i.e. a polynomial kernel of degree 2 without offset, is based on looking at pairs of features instead of single features. An analysis with a quadratic kernel is performed when the parameter `featureType` in `kbsvm` is set to `"quadratic"`. The SVMs in package `LiblineaR` are linear SVMs. These SVMs do not support non-linear kernels. To allow the usage of the SVMs in `LiblineaR` for quadratic analysis a quadratic explicit representation can be computed from the linear explicit representation. The features in the quadratic explicit representation are computed from pairs of features of the linear explicit representation. In the feature names of the quadratic explicit representation, the individual features of the pair are separated by the underscore character.

Please note that symmetric feature pairs are combined into a single value to reduce the dimensionality of the quadratic explicit representation. The quadratic explicit representation can be used in training and prediction on `LiblineaR` in the same way as a linear explicit representation. For the SVMs in other packages the quadratic explicit representation is not needed because the linear explicit representation is combined with a polynomial kernel of degree 2 for quadratic processing.

All of the necessary conversions from linear to quadratic explicit representation are handled within KeBABS automatically. If the quadratic explicit representation is needed for usage in an external software package it can be generated from any linear explicit representation in the following way - in this example again for only five sequences

```
> exrep <- getExRep(x=enhancerFB, selx=1:5, gappyK1M4, sparse=FALSE)
> dim(exrep)

[1]  5 80

> erquad <- getExRepQuadratic(exrep)
> dim(erquad)

[1]    5 3240

> erquad[1:5,1:5]

Dense explicit representation of class  "ExplicitRepresentationDense"

Quadratic             : TRUE
Used kernel           :
   gappy pair kernel: k=1, m=4

               AA_AA      AA_A.A     AA_A..A    AA_A...A   AA_A....A
Sample_1 0.047622277 0.064806633 0.059723760 0.063535915 0.064171274
Sample_2 0.004195657 0.005933555 0.004792486 0.006161768 0.006389982
Sample_3 0.013718389 0.026963728 0.024990773 0.025319599 0.023346643
Sample_4 0.017229898 0.020835342 0.022601049 0.021541625 0.025426180
Sample_5 0.036195010 0.044849977 0.042899978 0.043387478 0.043874978
```

The number of features increases drastically in the quadratic explicit representation compared to the linear explicit representation.

6 Training and Prediction for Binary Classification

As already seen in the Section 3, the KeBABS training method `kbsvm` follows the usual SVM training logic. In addition to sequences, the label vector and a sequence kernel the function allows for selection of one SVM from the supported packages. SVM parameters passed to the function are dependent on the selected SVM. They are specified in a unified format to avoid changes in parameter names or formats when switching SVM implementations. KeBABS translates the parameter names and formats to the specific SVM which is used in the training run. Details on the supported packages, the available SVMs and the unified SVM parameters for each SVM can be found on the help page for `kbsvm`. Please be aware that identically named parameters e.g. `cost` do not have the same values in different SVM formulations. This means that the value of such SVM parameters need to be adapted when changing to an SVM with a different SVM formulation.

Please note that in binary classification the value used for the positive label must be selected in a specific way. For numeric labels in binary classification the positive class must have the larger value, for factor or character based labels the positive label must be at the first position when sorting the labels in descendent order according to the C locale.

Instead of sequences, also a precomputed kernel matrix or an explicit representation can be passed as training data to the method `kbsvm`. Precomputed kernel matrices can be interesting for large data sets or kernels with higher runtime when multiple training runs should be performed without the overhead of recreating the kernel matrix on each run. A sequence kernel should be passed to `kbsvm` also in this situation as it might be needed for the computation of feature weights. The following example shows how to train and predict with a precomputed explicit representation. In this example the C-SVM from package `kernlab` is used and a dense explicit representation is generated because the SVMs in `kernlab` currently only support dense explicit representations. For the packages `e1071` and `LiblineaR` sparse explicit representations can also be used which are more efficient both statically and dynamically and also reduce the training time, especially for larger feature spaces.

```
> gappyK1M4 <- gappyPairKernel(k=1, m=4)
> exrep <- getExRep(enhancerFB, gappyK1M4, sparse=FALSE)
> numSamples <- length(enhancerFB)
> trainingFraction <- 0.8
> train <- sample(1:numSamples, trainingFraction * numSamples)
> test <- c(1:numSamples)[-train]
> model <- kbsvm(x=exrep[train, ], y=yFB[train], kernel=gappyK1M4,
+                pkg="kernlab", svm="C-svc", cost=15)
> pred <- predict(model, exrep[test, ])
> evaluatePrediction(pred, yFB[test], allLabels=unique(yFB))

    1 -1
1  44  8
-1  6 42

Accuracy:              86.000% (86 of 100)
Balanced accuracy:     86.000% (44 of 50 and 42 of 50)
```

```
Matthews CC:            0.721

Sensitivity:           88.000% (44 of 50)
Specificity:           84.000% (42 of 50)
Precision:             84.615% (44 of 52)
```

When the training is performed with a precomputed kernel matrix also in prediction a kernel matrix must be passed to the `predict()` method. In this case the kernel matrix is rectangular, the rows of the kernel matrix correspond to the test samples and the columns to the support vectors determined in the training as shown below. Please note that the support vector indices read from the model with the accessor `SVindex()` refer to the training set order. Currently training and prediction via kernel matrix is only supported by the SVMs implemented in package `kernlab`. With a precomputed kernel matrix the repeated effort for kernel matrix computation in multiple training runs can be avoided. However, for training with a precomputed kernel matrix the computation of feature weights is not possible.

```
> ## compute quadratic kernel matrix for training samples
> kmtrain <- getKernelMatrix(gappyK1M4, x=enhancerFB, selx=train)
> model1 <- kbsvm(x=kmtrain, y=yFB[train], kernel=gappyK1M4,
+                 pkg="kernlab", svm="C-svc", cost=15)
> ## compute rectangular kernel matrix of test samples against support vectors
> kmtest <- getKernelMatrix(gappyK1M4, x=enhancerFB, y=enhancerFB, selx=test,
+                           sely=train[SVindex(model1)])
> pred1 <- predict(model1, kmtest)
> evaluatePrediction(pred1, yFB[test], allLabels=unique(yFB))

    1 -1
1  44  8
-1  6 42

Accuracy:              86.000% (86 of 100)
Balanced accuracy:     86.000% (44 of 50 and 42 of 50)
Matthews CC:            0.721

Sensitivity:           88.000% (44 of 50)
Specificity:           84.000% (42 of 50)
Precision:             84.615% (44 of 52)
```

The method `kbsvm()` returns a KeBABS model of class `KBModel` which is needed for prediction, calculation of feature weights and generation of prediction profiles. It stores all data needed for these functions in an SVM independent format. It also contains the native model returned by the selected SVM. This model can be retrieved from the KeBABS model with the accessor `svmModel`. Apart from this data, the KeBABS model also can contain results returned from cross validation, grid search and model selection (see Section 7).

The method `predict()` is used to predict the class for new sequences from the model. The methods returns the class labels by default. Controlled through the parameter `predictionType`,

the `predict()` method can also return the decision values or class membership probability values instead of the class label. For class membership probability values, a probability model is necessary which is computed during training when the parameter `probModel` is set to `TRUE`. When feature weights (see Section 9) are available in the model, prediction is performed entirely within KeBABS. Native prediction in the selected SVM can be enforced by setting the flag `native` to `TRUE` in the `predict()` method.

Prediction profiles (see Section 10) are computed during prediction for the predicted sequences when the parameter `predProfiles` is set to `TRUE` in `predict`. Prediction profile generation requires the presence of feature weights (see Section 9) in the KeBABS model.

For classification the `evaluatePrediction()` method can be used to compute the performance parameters accuracy (ACC), balanced accuracy (BACC), Matthews Correlation Coefficient (MCC), sensitivity (SENS), specificity (SPEC) and precision (PREC) from the predicted values and the true labels as shown in the previous examples. Performance values can stored in a data frame for user-specific processing. The columns of the data frame are explained on the help page of the function `evaluatePrediction()`.

```
> perf <- evaluatePrediction(pred, yFB[test], allLabels=unique(yFB), print=FALSE)
> perf

  NUM PBAL TP FP FN TN ACC BAL_ACC SENS SPEC     PREC    MAT_CC
1 100  0.5 44  8  6 42  86      86   88   84 84.61538 0.7205767
```

7 Selection of Kernel Parameters, Hyperparameters and Models

For selection of kernel, kernel parameters, and SVM hyperparameters KeBABS provides cross validation and grid search. Both functionalities are implemented entirely in the `kebabs` package. They are available for all supported SVMs in a package-independent way for binary and multi-class classification as well as regression. Also model selection is provided by KeBABS for all supported SVMs. All of these functions require repeated training and prediction and, therefore, KeBABS was designed with strong focus on good performance of individual training and prediction runs allowing reasonable number of repetitions.

Under special circumstances, it might make sense to first narrow down the parameter range with the fast SVM implementations in package `LiblineaR` and then refine the reduced range with the slower but likely slightly more accurate SVMs from package `e1071` or `kernlab`.

7.1 Cross Validation

Cross validation (CV) is provided in KeBABS in three different variants:

k-fold cross validation: For k-fold cross validation the data set is split into k roughly equally sized parts called folds. One of these folds is used as test set and the other $k-1$ folds as training set. In k training runs the test fold is changed cyclically giving k different performance values on different subsets of the data. This variant is selected through setting the parameter `cross`, which defines the number of folds, to values larger than 1 in `kbsvm`.

Leave-One-Out cross validation (LOOCV): The LOOCV is k-fold CV taken to the extreme with a fold size of one sample. This variant is dynamically quite demanding and leads to the same number of training runs as there are samples in the data set. It is selected through setting the parameter `cross` to -1 in `kbsvm`.

Grouped cross validation (GCV): For this variant samples must be assigned to groups before running cross validation. Grouping can be performed in a way that learning can focus on the relevant features and is not distracted by simple features which do not really give adequate information about the classes. GCV is a specialized version of k-fold CV which respects group membership when splitting the data into folds. This means that one group is never distributed over more than one fold. GCV is performed when passing the group assignment of samples as integer vector or factor with the parameter `groupBy` to `kbsvm` together with a value of parameter `cross` that is smaller or equal to the number of groups.

For all types of CV, the result can be retrieved from the model with the accessor function `cvResult`. For classification the cross validation error is the mean classification error over all CV runs. For regression, the average of the mean squared errors of the runs is used.

For classification tasks, the collection of additional performance parameters during cross validation runs can be requested with the parameter `perfParameters` in `kbsvm`. This could be helpful especially when working with unbalanced datasets. For multiple repetitions of CV, the parameter `noCross` is used.

In the following example, grid search with grouped CV for the coiled coil sequences is shown as 3-fold grouped cross validation with two repetitions. In this example the groups were determined through single linkage clustering of sequence similarities derived from ungapped heptad-specific pairwise alignment of the sequences. The variable `ccgroup` contains the pre-calculated group assignments for the individual sequences. The grouping is a measure against over-representation of specific sequences in the PDB and ensures, that no pair of sequences from two different clusters match to a degree of 60% or higher of match positions related to the length of the shorter sequence (see also data Web page of package `procoil`[7]).

```
> data(CCoil)
> ccseq

  A AAStringSet instance of length 477
      width seq                                        names
  [1]    57 LRELQEALEEEVLTRQS...ARNRDLEAHVRQLQERM PDB1
  [2]    50 PLDISQNLAAVNKSLSD...DKIEEILSKIYHIENEI PDB2
  [3]    50 VMESFAVQNQNIDAQGE...QLILEALQGINKRLDNL PDB3
  [4]    47 SLVQAQTNARAIAAMKN...EGTQRLAIAVQAIQDHI PDB4
  [5]    46 LAGIVQQQQQLLDVVKR...KNLQTRVTAIEKYLKDQ PDB5
  ...   ... ...
[473]    11 AMKLVEKIREE                              PDB473
[474]    11 NLLILYDAIGT                             PDB474
[475]    11 EAACSAFATLE                             PDB475
```

[7] http://www.bioinf.jku.at/software/procoil/data.html

```
[476]    10 LAVALHELAS                              PDB476
[477]   123 KKAAEDRSKQLEDELVS...QKDEEKMEIQEIQLKEA PDB477

> head(yCC)

[1] TRIMER TRIMER TRIMER TRIMER TRIMER TRIMER
Levels: DIMER TRIMER

> head(ccgroups)

PDB1 PDB2 PDB3 PDB4 PDB5 PDB6
  45    2   44   43   38   42

> gappyK1M6 <- gappyPairKernel(k=1, m=6)
> model <- kbsvm(x=ccseq, y=as.numeric(yCC), kernel=gappyK1M6,
+                pkg="LiblineaR", svm="C-svc", cost=30, cross=3,
+                noCross=2, groupBy=ccgroups, perfObjective="BACC",
+                perfParameters=c("ACC", "BACC"))
> cvResult(model)

Cross validation result object of class  "CrossValidationResult"

cross                 : 3
noCross               : 2

Mean CV error         : 0.23246314
Mean accuracy         : 0.76753686
Mean balanced accuracy: 0.54850886

Run CV errors         :
0.24281431, 0.22211198

Run accuracies        :
0.75718569, 0.77788802

Run bal. accuracies   :
0.54557438, 0.55144335

groupBy:
45,2,44 ... 154,47,121

Fold CV errors:
0.20994475, 0.22302158, 0.27564103, 0.24736842, 0.24285714, 0.19594595

Fold accuracies:
```

```
0.79005525, 0.77697842, 0.72435897, 0.75263158, 0.75714286, 0.80405405

Fold balanced accuracies:
0.56762974, 0.56660232, 0.54230769, 0.48341014, 0.52678571, 0.60431759
```

7.2 Grid Search

The grid search functionality gives an overview of the performance behavior for different parameter settings and supports the user in the selection of the sequence kernel and optimal values for the kernel parameters and SVM hyperparameters. The grid in this functionality is defined as rectangular data structure, where the rows correspond to different sequence kernels including their kernel parameters. Each row corresponds to a single sequence kernel object with a specific setting of the kernel parameters. The columns correspond to the SVM parameters. KeBABS automatically performs cross validations for each of these grid points and returns the CV error as performance measure by default. Also grouped CV can be used for each grid point as shown above. The accessor `modelSelResult` to the grid search results in the model is identical to the accessor for model selection results.

```
> specK24 <- spectrumKernel(k=2:4)
> gappyK1M24 <- gappyPairKernel(k=1, m=2:4)
> gridKernels <- c(specK24, gappyK1M24)
> cost <- c(1,10, 100, 1000, 10000)
> model <- kbsvm(x=enhancerFB, y=yFB, kernel=gridKernels,
+                pkg="LiblineaR", svm="C-svc", cost=cost, cross=3,
+                explicit="yes", showProgress=TRUE)
> modelSelResult(model)

Grid search result object of class  "ModelSelectionResult"

cross                : 3
noCross              : 1
Smallest CV error    : 0.17794531

Grid Rows:
Kernel_1    SpectrumKernel: k=2
Kernel_2    SpectrumKernel: k=3
Kernel_3    SpectrumKernel: k=4
Kernel_4    GappyPairKernel: k=1, m=2
Kernel_5    GappyPairKernel: k=1, m=3
Kernel_6    GappyPairKernel: k=1, m=4

Grid Columns:
           cost
GridCol_1     1
GridCol_2    10
GridCol_3   100
```

```
GridCol_4  1000
GridCol_5 10000

Grid Errors:

         GridCol_1 GridCol_2 GridCol_3 GridCol_4 GridCol_5
Kernel_1 0.3559988 0.2859462 0.2680302 0.3140466 0.4340235
Kernel_2 0.2539620 0.2179617 0.2259697 0.2880143 0.2639540
Kernel_3 0.2019335 0.2180098 0.2379217 0.2359137 0.2640743
Kernel_4 0.3559748 0.2081019 0.2059976 0.2918380 0.3302071
Kernel_5 0.3580189 0.2160017 0.2021259 0.2480341 0.2479138
Kernel_6 0.3701272 0.2361542 0.1779453 0.3200467 0.2943511

Selected Grid Row:
GappyPairKernel: k=1, m=4

Selected Grid Column:
          cost
GridCol_3  100
```

In this example, only 3-fold CV was used to save time for vignette generation. When using 10-fold CV the differences between the kernels become more prominent because the effect of random splits of the data average out better. When the parameter `showProgress` is set to `TRUE` in `kbsvm` , a progress indication is displayed.

The default performance parameter in grid search is the CV error, as in cross validation. For non-symmetrical datasets other performance parameters set with parameter `perfParameters` might be more expressive. For grid search, the performance objective can be defined explicitly with the parameter `perfObjective` as criterion to select the best setting of the grid parameters. The accuracy (ACC), balanced accuracy (BACC) and Matthews Correlation Coefficient (MCC) can be chosen as performance objective.

Grid search can also be done over multiple SVMs in the same or in different packages as in the next example. In this case vectors of identical length must be passed for the parameters `pkg`, `svm` and the respective SVM parameters. The corresponding entries in these vectors refer to one SVM with its hyperparameter(s) represent one grid column. With the parameter `showCVTimes` set to `TRUE` in `kbsvm` the cross validation runtimes are recorded for each individual grid point and are displayed at the end of the grid search.

```
> specK34 <- spectrumKernel(k=3:4)
> gappyK1M34 <- gappyPairKernel(k=1, m=3:4)
> gridKernels <- c(specK34, gappyK1M34)
> pkgs <- c("e1071", "LiblineaR", "LiblineaR")
> svms <- c("C-svc","C-svc","l1rl2l-svc")
> cost <- c(50, 50, 12)
> model <- kbsvm(x=enhancerFB, y=yFB, kernel=gridKernels,
+                pkg=pkgs, svm=svms, cost=cost, cross=10,
```

```
+                explicit="yes", showProgress=TRUE,
+                showCVTimes=TRUE)


Grid Search Times:

         GridCol_1 GridCol_2 GridCol_3
Kernel_1      1.21      2.53      3.01
Kernel_2      4.43      4.67      3.94
Kernel_3      1.12      2.54      2.84
Kernel_4      1.47      2.51      3.46


> modelSelResult(model)


Grid search result object of class  "ModelSelectionResult"

cross                 : 10
noCross               : 1
Smallest CV error     : 0.184

Grid Rows:
Kernel_1    SpectrumKernel: k=3
Kernel_2    SpectrumKernel: k=4
Kernel_3    GappyPairKernel: k=1, m=3
Kernel_4    GappyPairKernel: k=1, m=4

Grid Columns:
                pkg        svm cost
GridCol_1     e1071      C-svc   50
GridCol_2 LiblineaR      C-svc   50
GridCol_3 LiblineaR l1rl2l-svc   12

Grid Errors:

         GridCol_1 GridCol_2 GridCol_3
Kernel_1     0.218     0.230     0.228
Kernel_2     0.246     0.232     0.268
Kernel_3     0.188     0.184     0.210
Kernel_4     0.198     0.192     0.202

Selected Grid Row:
GappyPairKernel: k=1, m=3

Selected Grid Column:
                pkg   svm cost
GridCol_2 LiblineaR C-svc   50
```

7.3 Model Selection

In KeBABS model selection is performed very similar to grid search. It is based on nested cross validation with an inner cross validation loop that determines the best setting of the hyper parameters and an outer cross validation loop to test the performance of the model with the best parameter settings on independent test sets. The result of model selection is the performance of the complete model selection process, not that of a single best model.

Model selection is triggered when the parameter `nestedCross` in `kbsvm` is set to a value larger than 1. This parameter defines the number of folds in the outer cross validation. The number of folds in the inner cross validation is defined through the parameter `cross` as usual. Multiple repetitions of model selection can be requested with the parameter `noNestedCross`.

In the following example, model selection is performed with 4-fold outer cross validation and 10-fold inner cross validation. The parameters of the best models selected in the inner cross validation are shown in the model selection result object and the results of the outer cross validation in the CV result object extracted from the model.

```
> specK34 <- spectrumKernel(k=3:4)
> gappyK1M34 <- gappyPairKernel(k=1, m=3:4)
> gridKernels <- c(specK34, gappyK1M34)
> cost <- c(10, 50, 100)
> model <- kbsvm(x=enhancerFB, y=yFB, kernel=gridKernels,
+                pkg="LiblineaR", svm="C-svc", cost=cost, cross=10,
+                explicit="yes", nestedCross=4)
> modelSelResult(model)

Model selection result object of class  "ModelSelectionResult"

cross                : 10
noCross              : 1
nestedCross          : 4
noNestedCross        : 1

Grid Rows:
Kernel_1    SpectrumKernel: k=3
Kernel_2    SpectrumKernel: k=4
Kernel_3    GappyPairKernel: k=1, m=3
Kernel_4    GappyPairKernel: k=1, m=4

Grid Columns:
          cost
GridCol_1   10
GridCol_2   50
GridCol_3  100

Selected Grid Row:
```

```
1  GappyPairKernel: k=1, m=3
2  GappyPairKernel: k=1, m=3
3  SpectrumKernel: k=3
4  GappyPairKernel: k=1, m=3

Selected Grid Column:
  selGridCol cost
1  GridCol_3  100
2  GridCol_3  100
3  GridCol_3  100
4  GridCol_3  100
```

```
> cvResult(model)

Outer cross validation result object of class  "CrossValidationResult"

nestedCross           : 4
noNestedCross         : 1
CV error:             : 0.206

Fold CV errors:
0.224, 0.168, 0.272, 0.160
```

The accuracy of the model selection procedure is the average of the results from the outer CV runs. The performance for the best model is pretty similar and the gappy pair kernel together with a cost parameter value of 50 is selected as best model in most cases. This result shows that the best model with the gappy pair kernel is pretty stable for different splits of the data. Please also consider the small amount of data that is used for model selection in this example. Usually a larger number of samples should be used. Even under these unfavorable conditions, the performance is very reasonable.

8 Regression

The classification examples shown up to now were based on real world data for transcription factor binding. For regression, no real-world data exists for the sequences provided in the package. We use a toy dataset with an artificial set of real values as target values for the regression example through linear combination of feature values for the features A.A, CC, C.G, GG, G.G, TG and T.G plus added noise. The vector with the target values is called `yReg`. We perform grid search with the gappy pair kernel on the nu support vector regression in e1071 for hyperparameter `nu`.

```
> head(yReg)

                              [,1]
chr19.21240050.21240876  1.017952
```

```
chr2.144463827.144464504 1.341635
chr7.38525408.38526285   1.268669
chr4.90747075.90748001   1.426697
chr4.9148679.9149631     1.116384
chr5.113189184.113189986 1.459810

> gappyK1M2 <- gappyPairKernel(k=1, m=2)
> model <- kbsvm(x=enhancerFB, y=yReg, kernel=gappyK1M2,
+                pkg="e1071", svm="nu-svr", nu=c(0.5,0.6,0.7,0.8),
+                cross=10, showProgress=TRUE)
> modelSelResult(model)

Grid search result object of class  "ModelSelectionResult"

cross                : 10
noCross              : 1
Smallest CV error    : 0.0030053677

Grid Rows:
Kernel_1    GappyPairKernel: k=1, m=2

Grid Columns:
           nu
GridCol_1 0.5
GridCol_2 0.6
GridCol_3 0.7
GridCol_4 0.8

Grid Errors:

           GridCol_1   GridCol_2   GridCol_3   GridCol_4
Kernel_1 0.003037051 0.003005368 0.003072086 0.003125145

Selected Grid Column:
           nu
GridCol_2 0.6
```

Now a model is trained on a training set with the best performing hyperparameters `nu=0.7` and the target value for the remaining sequences is predicted. Usually the hyperparameters should be selected on independent data which is not used for verification of the performance of the selected hyperparameter set. Because of the small size of the dataset we could not perform hyperparameter selection on independent data. In the regression scenario the mean squared error is used as performance measure. Finally we check the feature weights (see section 9) which were computed by default.

```
> numSamples <- length(enhancerFB)
> trainingFraction <- 0.7
```

```
> train <- sample(1:numSamples, trainingFraction * numSamples)
> test <- c(1:numSamples)[-train]
> model <- kbsvm(x=enhancerFB[train], y=yReg[train],
+                kernel=gappyK1M2, pkg="e1071", svm="nu-svr",
+                nu=0.7)
> pred <- predict(model, enhancerFB[test])
> mse <- sum((yReg[test] - pred)^2)/length(test)
> mse

[1] 0.003035779

> featWeights <- featureWeights(model)
> colnames(featWeights)[which(featWeights > 0.4)]

[1] "CC"   "G.C"  "GG"   "G.G"  "G..G" "TG"   "T.G"
```

We see that the strongest features in the model were extracted correctly even in the presence of 30% noise on the target. The feature A.A had the smallest contribution to the target value. As the target values were created from the features of a gappy pair kernel a spectrum kernel is not able to extract the same amount of high weight features but those present in the feature space of the spectrum kernel are found.

```
> model <- kbsvm(x=enhancerFB[train], y=yReg[train],
+                kernel=spectrumKernel(k=2), pkg="e1071", svm="nu-svr",
+                nu=0.7)
> pred <- predict(model, enhancerFB[test])
> featWeights <- featureWeights(model)
> colnames(featWeights)[which(featWeights > 0.4)]

[1] "CC" "GG" "TG"
```

Comparing the performance with model selection for parameter nu the gappy pair kernel shows the better performance as expected. Please note the different parameter ranges for the hyperparameter nu between spectrum kernel and gappy pair kernel.

```
> model <- kbsvm(x=enhancerFB[train], y=yReg[train],
+                kernel=spectrumKernel(k=2), pkg="e1071", svm="nu-svr",
+                nu=c(0.5,0.55,0.6), cross=10, nestedCross=5)
> modelSelResult(model)

Model selection result object of class  "ModelSelectionResult"

cross                : 10
noCross              : 1
nestedCross          : 5
```

```
noNestedCross         : 1

Grid Rows:
Kernel_1    SpectrumKernel: k=2

Grid Columns:
            nu
GridCol_1 0.50
GridCol_2 0.55
GridCol_3 0.60

Selected Grid Column:
  selGridCol   nu
1  GridCol_3 0.60
2  GridCol_2 0.55
3  GridCol_3 0.60
4  GridCol_1 0.50
5  GridCol_2 0.55

> cvResult(model)

Outer cross validation result object of class  "CrossValidationResult"

nestedCross           : 5
noNestedCross         : 1
CV error:             : 0.0045182942

Fold CV errors:
0.0042509905, 0.0043292578, 0.0040706061, 0.0050293959, 0.0049112209

> model <- kbsvm(x=enhancerFB[train], y=yReg[train],
+                kernel=gappyPairKernel(k=1,m=2), pkg="e1071",
+                svm="nu-svr", nu=c(0.6,0.65,0.7), cross=10,
+                nestedCross=5)
> modelSelResult(model)

Model selection result object of class  "ModelSelectionResult"

cross                 : 10
noCross               : 1
nestedCross           : 5
noNestedCross         : 1

Grid Rows:
Kernel_1    GappyPairKernel: k=1, m=2
```

```
Grid Columns:
            nu
GridCol_1 0.60
GridCol_2 0.65
GridCol_3 0.70

Selected Grid Column:
  selGridCol   nu
1  GridCol_1 0.60
2  GridCol_3 0.70
3  GridCol_1 0.60
4  GridCol_2 0.65
5  GridCol_2 0.65

> cvResult(model)

Outer cross validation result object of class  "CrossValidationResult"

nestedCross           : 5
noNestedCross         : 1
CV error:             : 0.0035745177

Fold CV errors:
0.0041253353, 0.0024458940, 0.0037646516, 0.0040094624, 0.0035272451
```

9 Feature Weights

For the position-independent kernel the kernel was expressed as

$$k(x,y) = \sum_{m \in \mathcal{M}} N(m,x) \cdot N(m,y) \tag{11}$$

The discrimination function of a support vector machine with the Langrange multipliers $\alpha_1, ..., \alpha_l$ is given as

$$\begin{aligned} g(x) &= b + \sum_{i}^{l} \alpha_i \cdot y_i \cdot k(x, x_i) \\ &= b + \sum_{i}^{l} \alpha_i \cdot y_i \cdot \sum_{m \in \mathcal{M}} N(m,x) \cdot N(m, x_i) \\ &= b + \sum_{m \in \mathcal{M}} N(m,x) \cdot \underbrace{\sum_{i}^{l} \alpha_i \cdot y_i \cdot N(m, x_i)}_{=w(m)} \end{aligned} \tag{12}$$

Through rearrangement of the sums, terms that are not dependent on the current sample can be split off and can be computed in advance from the Lagrange multipliers and the explicit representation of the support vectors. These precomputed terms are called feature weights because they give the weight for each feature with which the feature counts are weighted.

The feature weight values give the contribution of each occurrence of a pattern to the discrimination value. As mentioned before, this gives some rough hint on the relevance of a pattern but does not describe the importance of a pattern exactly because of overlapping features and substring features. We will see in the Section 10 how to get a clearer picture about the relevance of specific positions in the sequence.

Feature weights are computed by default if possible during training. With the parameter `featureWeights` set to `FALSE` in `kbsvm` the generation of feature weights can be suppressed. When a pre-computed kernel matrix for a sequence kernel is passed to KeBABS training feature weights cannot be computed. Feature weights are used for prediction and they are a prerequisite for the generation of feature profiles.

For the position-dependent kernel, a similar representation of the feature weights can be derived. The position-dependent kernel in its most general form can be described as

$$k(x,y) = \sum_{m\in\mathcal{M}} \sum_{p=1}^{L_x} \sum_{q=1}^{L_y} \mathbf{1}(m,x,p) \cdot E(p,q) \cdot \mathbf{1}(m,y,q) \tag{13}$$

The discrimination function of a support vector machine with the Lagrange multipliers $\alpha_1, ..., \alpha_l$ is given for the position-dependent kernel as

$$\begin{aligned} g(x) &= b + \sum_{i}^{l} \alpha_i \cdot y_i \cdot k(x,x_i) \\ &= b + \sum_{i=1}^{l} \alpha_i \cdot y_i \cdot \sum_{m\in\mathcal{M}} \sum_{p=1}^{L_x} \sum_{q=1}^{L_{xi}} \mathbf{1}(m,x,p) \cdot E(p,q) \cdot \mathbf{1}(m,x_i,q) \\ &= b + \sum_{m\in\mathcal{M}} \sum_{p=1}^{L_x} \mathbf{1}(m,x,p) \cdot \underbrace{\sum_{i=1}^{l} \sum_{q=1}^{L_{xi}} \mathbf{1}(m,x_i,q) \cdot \alpha_i \cdot y_i \cdot E(p,q)}_{=\tilde{w}(m,p)} \end{aligned} \tag{14}$$

In a similar way as in the position-independent case a part that is independent of the current sample can be split off through exchange of sums and rearrangement. In the case of position-specific kernels the feature weights are dependent both on patterns and positions. For more details see Bodenhofer *et al.* (2009).

As feature weights for the position-dependent kernel are dependent on the positions, they are not computed fully during the training. Only a first step of feature weight computation is performed during training allowing faster computation of full feature weights during prediction or computation of feature profiles when the actual sequences are available.

Feature weights can be extracted from the model with the accessor `featureWeights`. For pairwise multiclass the feature weights of multiple pairwise SVMs are stored as list in the model. When feature weights are present in the model the prediction is performed via feature weights in KeBABS except for multiclass classification where always native prediction is used.

10 Prediction Profiles

Prediction profiles are computed from feature weights to show the contribution of each sequence position to the decision value. If stretches of the DNA sequence show a clear positive or negative contribution they are likely relevant for the biological function underlying the learning task.

Prediction profiles can be computed for all sequence kernels provided in KeBABS. Feature weights are a prerequisite for computation of feature profiles and must either be computed during training (see parameter `featureWeights` in method `kbsvm`) or added later to the model with the function `getFeatureWeights()`. Prediction profiles can either be generated during prediction (see parameter `predProf` in `predict()`) or specifically for a given set of sequences from a given model with `getPredictionProfile`. Finally the predicition profiles for a single sequence or a pair of sequences can be plotted. Usually only plotting to a PDF file makes sense to get a reasonable resolution for longer sequences.

The example for prediction profiles is based on the sequence data for coiled coiled proteins that we used earlier already. A more detailed description is presented in (Mahrenholz *et al.*, 2011). We train a model on the full dataset and compute prediction profiles for the wild-type yeast protein GCN4. Finally the prediction profile for this sequence is generated.

```
> data(CCoil)
> gappya <- gappyPairKernel(k=1,m=11, annSpec=TRUE)
> model <- kbsvm(x=ccseq, y=as.numeric(yCC), kernel=gappya,
+                pkg="e1071", svm="C-svc", cost=15)
> featureWeights(model)[,1:5]

    A......Aa......a                AAab   A.......Aa.......b
          0.11453726           0.11933186            0.04043841
              A.Aa.c A........Aa........c
         -0.08263644           0.05626802

> GCN4 <- AAStringSet(c("MKQLEDKVEELLSKNYHLENEVARLKKLV",
+                       "MKQLEDKVEELLSKYYHTENEVARLKKLV"))
> names(GCN4) <- c("GCN4wt", "GCN_N16Y,L19T")
> annCharset <- annotationCharset(ccseq)
> annot <- c("abcdefgabcdefgabcdefgabcdefga",
+            "abcdefgabcdefgabcdefgabcdefga")
> annotationMetadata(GCN4, annCharset=annCharset) <- annot
> predProf <- getPredictionProfile(GCN4, gappya,
+                                  featureWeights(model),
+                                  modelOffset(model))
> predProf
```

```
An object of class  "PredictionProfile"

Sequences:

  A AAStringSet instance of length 2
    width seq                                          names
[1]    29 MKQLEDKVEELLSKNYHLENEVARLKKLV           GCN4wt
[2]    29 MKQLEDKVEELLSKYYHTENEVARLKKLV           GCN_N16Y,L19T

gappy pair kernel: k=1, m=11, annSpec=TRUE

Baselines:  0.02365245 0.02365245

Profiles:
                     Pos 1        Pos 2            Pos 28       Pos 29
       GCN4wt  0.111889252 -0.140135744  ...  0.063537667  0.130766121
GCN_N16Y,L19T  0.115429679 -0.144569953  ...  0.059671695  0.122502550
```

The prediction profiles are contained in a data structure together with the sequences and the sequence kernel used for model training and profile generation. Each row in a profile is associated with one sample, each entry corresponds to the contribution of the specific position in the sample to the decision value. As all patterns relevant at a position are included in the profile values the relevance of positions for prediction results become visible. The prediction profile of the wild-type GCN4 protein which builds dimers is plotted with

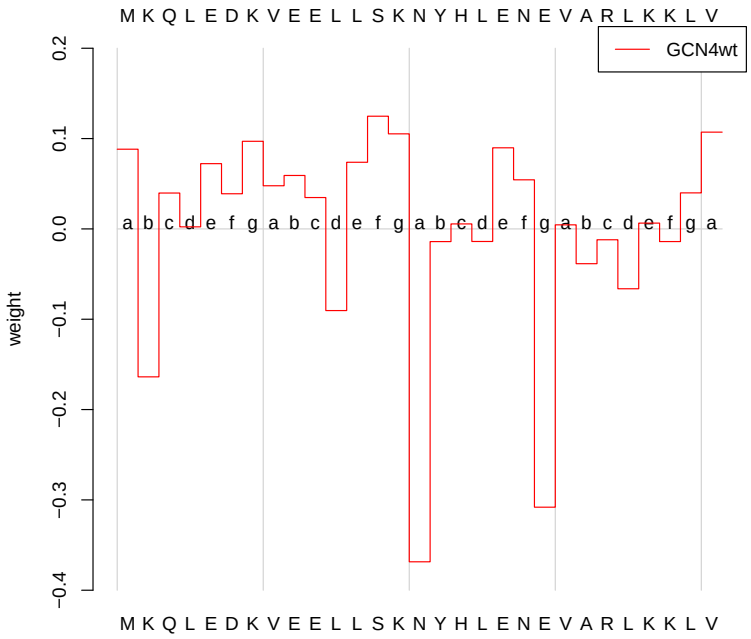
```
> plot(predProf, sel=1, ylim=c(-0.4, 0.2), heptads=TRUE, annotate=TRUE)
```

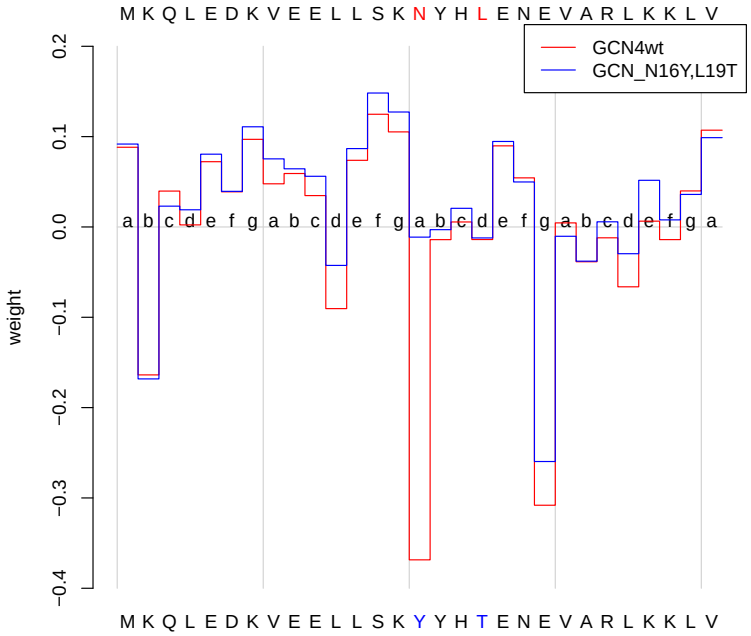
We see that especially position 16 has a strong negative contribution, i.e. a contribution for prediction as dimer. In the trimeric second protein $GCN4_{N16Y,L19T}$ with a mutation at this location the prediction profile shows the difference. The next plot shows both prediction profiles overlayed. The strong contribution of position 16 to dimeric oligomerization prediction is lost.

```
> plot(predProf, sel=c(1,2), ylim=c(-0.4, 0.2), heptads=TRUE, annotate=TRUE)
```

In this simple example with short sequences the plot can easily be displayed. When using longer sequences plotting to PDF file is necessary. The help page for `plot()` in KeBABS shows an example. The plotting function also provides the possibility for smoothing the profile with a sliding window average. This could be interesting in situations where regions of interest should be identified.

In contrast to feature weights which show the individual contribution of a single feature occurrence but are not as clearly indicative because of overlaps with other features and substring features in prediction profiles the contribution of all features present at a single position is summed up showing the full contribution of each position to the decision value. In this way prediction profiles provide much better biological interpretability of the learning results because individual positions as in this example or stretches of the sequence containing the driving patterns become visible.


M K Q L E D K V E E L L S K N Y H L E N E V A R L K K L V
GCN4wt
weight
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
a b c d e f g a b c d e f g a b c d e f g a b c d e f g a
M K Q L E D K V E E L L S K N Y H L E N E V A R L K K L V


M K Q L E D K V E E L L S K N Y H L E N E V A R L K K L V
GCN4wt
GCN_N16Y,L19T
weight
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
a b c d e f g a b c d e f g a b c d e f g a b c d e f g a
M K Q L E D K V E E L L S K Y Y H T E N E V A R L K K L V

The sequence patterns driving the microbiological mechanism which is studied might be more complex than the patterns available in a specific kernel. Therefore the high contributing feature weights / sequence positions might be part of a larger pattern that is not fully captured by the used kernel and additional analysis steps might be necessary once the relevant sections of the sequences are identified. Considering these aspects the results of such an analysis need to be taken with a grain of salt. If some biological understanding of the underlying pattern(s) is available already especially the motif kernel provides high flexibility to define the features according to this knowledge.

11 Multiclass Classification

As in the regression case, no real-wold multi-class labels are available for the sequences provided with the package. We create synthetic labels by splitting the value range of the regression target into three similar sized intervals. The multiclass label vector provided with the package is called `yMC`. Different types of multiclass are implemented for the non-native multiclass variants in the supported packages. In `LiblineaR` all non-native multiclass implementations are based on One-Against-The-Rest which is used in the following example of multiclass training and prediction.

```
> table(yMC)

yMC
  1   2   3
137 193 170

> gappyK1M2 <- gappyPairKernel(k=1, m=2)
> model <- kbsvm(x=enhancerFB[train], y=yMC[train],
+                kernel=gappyK1M2, pkg="LiblineaR",
+                svm="C-svc", cost=300)
> pred <- predict(model, enhancerFB[test])
> evaluatePrediction(pred, yMC[test], allLabels=unique(yMC))

   1  2  3
1 32  9  0
2  3 39 12
3  0  6 49

Accuracy:              80.000% (120 of 150)
Balanced accuracy:     81.326%
Matthews CC:            0.698

Class 1:
Sensitivity:           91.429% (32 of 35)
Specificity:           76.522% (88 of 115)
Precision:             78.049% (32 of 41)
```

```
Class 2:
Sensitivity:           72.222% (39 of 54)
Specificity:           84.375% (81 of 96)
Precision:             72.222% (39 of 54)

Class 3:
Sensitivity:           80.328% (49 of 61)
Specificity:           79.775% (71 of 89)
Precision:             89.091% (49 of 55)
```

In the One-Against-The-Rest approach with k classes multi-class training consists of k binary classification training runs using in each run one class and the union of all other classes as second class. When feature weights are computed, the model also contains the feature weights for these k binary classifiers structured as list.

```
> featWeights <- featureWeights(model)
> length(featWeights)

[1] 3

> featWeights[[1]][1:5]

[1]  0.6479387 -1.2134351 -0.6851692 -3.9860212 -3.1079928

> featWeights[[2]][1:5]

[1]  3.465998 17.818891  8.071615  4.436099 11.572472

> featWeights[[3]][1:5]

[1]   2.540421 -13.636588  -2.976245  -2.182230   3.705525
```

Multiclass prediction is always performed natively in the package used for training. Prediction profiles can be generated in the multiclass case for the individual binary SVMs with the function `getPredictionProfiles` by passing the respective element from feature weights and from the model offset vector as in the following example.

```
> predProf <- getPredictionProfile(enhancerFB, gappyK1M2,
+                                  featureWeights(model)[[2]],
+                                  modelOffset(model)[2])
> predProf
```

```
An object of class  "PredictionProfile"

Sequences:

  A DNAStringSet instance of length 500
      width seq                                       names
  [1]   827 ACTAAACAACATCAATC...TAGGCAAAATCCTGACA Sample_1
  [2]   678 GAATATAGACCCTTGGT...AAGTTATATTAATTTAT Sample_2
  [3]   878 CACCCACATGGTGGCTC...TCCCCACTGTATCACTC Sample_3
  [4]   927 TGCCGTAGTGTGCCAGC...TGATTCTCCTGGATCAA Sample_4
  [5]   953 CTTCATATACCTATTAA...CCTAATTTAAAAAGGGG Sample_5
  ...   ... ...
[496]   478 ACGAGAATGAGGGAAAG...GTCTGCCTTCCGAGTGA Sample_496
[497]  1552 AGGGGACTGGAAGAAAG...TCTCTTCTAAGCGAGCA Sample_497
[498]   503 GATGAACGTAAAATGCA...TAATGACTCCCTTCTGA Sample_498
[499]  1577 ACCCCTCTGAGACCAAG...GTAGGTCTGTATTCCTG Sample_499
[500]   778 CACTGTCATAACTTGCT...GTAGAGTAGCTGCTCTC Sample_500

gappy pair kernel: k=1, m=2

Baselines:  0.04542863 0.05541221   ...   0.02382338 0.04828982

Profiles:
                 Pos 1        Pos 2          Pos 2327     Pos 2328
  Sample_1  0.015555941  0.015500810  ...  0.000000000  0.000000000
  Sample_2  0.022805180  0.036948987  ...  0.000000000  0.000000000
  Sample_3  0.016479432  0.026201173  ...  0.000000000  0.000000000
  Sample_4  0.036142286  0.019158091  ...  0.000000000  0.000000000
  Sample_5  0.026309827  0.030322645  ...  0.000000000  0.000000000
  ........       ......       ......           ......       ......
Sample_496  0.022685640  0.042002925  ...  0.000000000  0.000000000
Sample_497  0.014142052  0.004848784  ...  0.000000000  0.000000000
Sample_498  0.043612814  0.048000646  ...  0.000000000  0.000000000
Sample_499  0.011629614  0.006878189  ...  0.000000000  0.000000000
Sample_500  0.010950992  0.037774399  ...  0.000000000  0.000000000
```

The non-native multiclass implementation in packages `kernlab` and `e1071` are based on pairwise binary classifications. For k classes $k * (k-1)/2$ binary classifications are performed and the feature weights and model offsets of these models are stored in the KeBABS model. For larger numbers of classes the number of binary training runs is considerably higher than for the One-Against-The-Rest approach which was chosen in `LiblineaR` because this package is especially aiming at large feature spaces and large number of samples. If the performance is not an issue, usually pairwise multiclass is considered as the preferred method. The native multiclass approach according to Crammer and Singer is available in packages `kernlab` and `LiblineaR` by selecting the value `mc-natC` for the parameter `svm`. The Weston/Watkins native multiclass approach is only available in `kernlab` and is selected with the parameter value `mc-natW`. For Weston/Watkins

multiclass no feature weights and prediction profiles are available. As KeBABS supports all of the variants via the supported packages and as switching between implementations is simple the performance of all methods can be compared easily if the data size allows it.

12 Future Extensions

The current version of the package is single threaded and does not make use of multiprocessing infrastructures. The performance of this framework can further be improved by parallelization of various functions in the package for multicore machines.

The AUC should be included as additional performance measure for cross validation, grid search and model selection.

To restrict the effort the current implementation only considers SVMs in the packages `kernlab`, `e1071` and `LiblineaR`. In the future additional SVMs should be included in the framework, e.g.

- Potential Support Vector Machine (PSVM) (Hochreiter and Obermayer, 2006)
- Relevance Vector Machine (RVM) (Tipping, 2001)
- Least Squares Support Vector Machine (LS-SVM) (Suykens and Vandewalle, 1999)
- . . .

Currently only `kernlab` supports training via the kernel matrix. For all other SVMs training can only be performed via the explicit representation. The dense version of `libsvm` should be included to provide an alternative training possibility via kernel matrix.

13 Change Log

Version 1.0.5:

- new accessors selGridRow, selGridCol and fullModel for class ModelSelectionResult
- change of naming of feature weights because of change in LiblineaR 1.94-2
- GCC warnings in Linux removed

Version 1.0.4:

- change in LiblineaR - upgrade to LIBLINEAR 1.94, in function LiblineaR the parameter labels was renamed to target
- correction in model selection for performance parameters
- minor changes in help pages
- minor changes in vignette

Version 1.0.3:

- extension of function linearKernel to optionally return a sparse kernel matrix

- new accessor SVindex for class KBModel
- error correction in subsetting of sparse explicit representation for head / tail on sparse explicit representation
- error correction of vector length overflow in sparse explicit representation for very large number of sequences in spectrum, gappy pair and motif kernel
- error correction for training with position specific kernel and computation of feature weights
- error correction in coercion of kernel to character for distance weighting
- error correction in subsetting of prediction profile
- error correction in spectrum, gappy pair and motif kernel for kernel matrix - last feature was missing in kernel value in rare situations
- error correction and minor code changes for mismatch kernel
- check uniqueness of motifs in motif kernel
- build warnings on Windows removed
- minor changes in help pages
- change name of vignette Rnw to lowercase
- minor changes in vignette

Version 1.0.2:

- correction of MCC
- correction of computation of feature weights for LiblineaR with more than 3 classes

Version 1.0.1:

- correction for cross validation with factor label
- correction for storing prob model in kebabs model for kernlab
- removal of clang warnings for unused functions

Version 1.0.0: first official release as part of Bioconductor 3.0, released September 25, 2014

14 How to cite this package

If you use this package for research that is published later, you are kindly asked to cite it as follows:

> J. Palme and U.Bodenhofer (2014).
> KeBABS - An R Package for Kernel Based Analysis of Biological Sequences.
> Unpublished. http://www.bioconductor.org/packages/release/html/kebabs.html.

To obtain BibTEX entries of the reference, you can enter the following into your R session:

```
> toBibtex(citation("kebabs"))
```

References

AttractiveChaos (2011). klib – a standalone and lightweight c library. `https://github.com/attractivechaos/klib`.

Ben-Hur, A. and Brutlag, D. L. (2003). Remote homology detection: a motif based approach. *Bioinformatics*, **19**, 26–33.

Bodenhofer, U. (2011). Procoil — a web service and an r package for predicting the oligomerization of coiled coil proteins. `http://www.bioconductor.org/packages/release/html/procoil.html`.

Bodenhofer, U., Schwarzbauer, K., Ionescu, M., and Hochreiter, S. (2009). Modelling position specificity in sequence kernels by fuzzy equivalence relations. In J. P. Carvalho, D. Dubois, U. Kaymak, and J. M. C. Sousa, editors, *Proc. Joint 13th IFSA World Congress and 6th EUSFLAT Conference*, pages 1376–1381, Lisbon.

Bodenhofer, U., Kothmeier, A., and Hochreiter, S. (2011). APCluster: an R package for affinity propagation clustering. *Bioinformatics*, **27**, 2463–2464.

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, **2**, 27:1–27:27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. (2008). LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, **9**, 1871–1874.

Gorodkin, J. (2004). Comparing two k-category assignments by a k-category correlation coefficient. *Computational Biology and Chemistry*, **28**(5-6), 367–374.

Hochreiter, S. and Obermayer, K. (2006). Support vector machines for dyadic data. *Neural Comput*, **18**(6), 1472–1510.

Karatzoglou, A., Smola, A., Hornik, K., and Zeileis, A. (2004). kernlab – an S4 package for kernel methods in R. *Journal of Statistical Software*, **11**(9), 1–20.

Kuksa, P., Huang, P.-H., and Pavlovic, V. (2008). A fast, large-scale learning method for protein sequence classification. In *8th Int. Workshop on Data Mining in Bioinformatics*, pages 29–37, Las Vegas, NV.

Lee, D., Karchin, R., and Beer., M. A. (2011). Discriminative prediction of mammalian enhancers from DNA sequence. *Genome Research*, **21**(12), 2167–2180.

Leslie, C. S., Eskin, E., and Noble, W. S. (2002). The spectrum kernel: A string kernel for SVM protein classification. In R. B. Altman, A. K. Dunker, L. Hunter, K. Lauderdale, and T. E. D. Klein, editors, *Pacific Symposium on Biocomputing*, pages 566–575. World Scientific.

Leslie, C. S., Eskin, E., Cohen, A., Weston, J., and Noble, W. S. (2003). Mismatch string kernels for discriminative protein classification. *Bioinformatics*, **1**(1), 1–10.

Mahrenholz, C. C., Abfalter, I. G., Bodenhofer, U., Volkmer, R., and Hochreiter, S. (2011). Complex networks govern coiled-coil oligomerizations - predicting and profiling by means of a machine learning approach. *Mol Cell Proteomics*, **10**(5), M110.004994.

Rätsch, G. and Sonnenburg, S. (2004). *Kernel Methods in Computational Biology*, chapter Accurate Splice Site Prediction for Caenorhabditis elegans, pages 277–298. MIT, Cambridge, MA.

Rätsch, G., Sonnenburg, S., and Schölkopf, B. (2005). RASE: Recognition of alternatively spliced exons in C. elegans. *Bioinformatics*, **21**(suppl1), i369–i377.

Suykens, J. A. K. and Vandewalle, J. (1999). Least squares support vector machine classifiers. *Neural Processing Letters*, **9**(3), 293–300.

Tipping, M. E. (2001). Sparse bayesian learning and the Relevance Vector Machine. *Journal of Machine Learning Research*, **1**(211–244).

Visel, A., Blow, M. J., Li, Z., Zhang, T., Akiyama, J. A., Holt, A., Plajzer-Frick, I., Shoukry, M., Wright, C., Chen, F., Afzal, V., Ren, B., Rubin, E. M., and Pennacchio, L. A. (2009). ChIP-seq accurately predicts tissue-specific activity of enhancers. *Nature*, **457**(7231), 854–858.