

QuasiAlign: Position Sensitive P-Mer Frequency Clustering with Applications to Genomic Classification and Differentiation

Anurag Nagar
Southern Methodist University

Michael Hahsler
Southern Methodist University

Abstract

Recent advances in Metagenomics and the Human Microbiome provide a complex landscape for dealing with a multitude of genomes all at once. One of the many challenges in this field is classification of the genomes present in a sample. Effective metagenomic classification and diversity analysis require complex representations of taxa. With this package we develop a suite of tools, based on novel quasi-alignment techniques to rapidly classify organisms using our new approach on a laptop computer instead of several multi-processor servers. This approach will facilitate the development of fast and inexpensive devices for microbiome-based health screening in the near future.

Keywords: data mining, clustering, Markov chain.

1. Introduction

Metagenomics (Handelsman, Rondon, Brady, Clardy, and Goodman 1998) and the Human Microbiome Turnbaugh, Ley, Hamady, Fraser-Liggett, Knight, and Gordon (2007); Mai, Ukhanova, and Baer (2010) provide a complex landscape for dealing with a multitude of genomes all at once. One of the many challenges in this field is classification of the genomes present in the sample. Effective metagenomic classification and diversity analysis require complex representations of taxa.

A common characteristic of most sequence-based classification techniques (e.g., BALibase (Smith and Waterman 1981), BLAST (Altschul, Gish, Miller, Myers, and Lipman 1990), T-Coffee (Notredame, Higgins, and Heringa 2000), MAFFT (Katoh, Misawa, Kuma, and Miyata 2002), MUSCLE (Edgar 2004b,a), Kalign (Lassmann and Sonnhammer 2006) and ClustalW2 and ClustalX2 (Larkin, Blackshields, Brown, Chenna, McGettigan, McWilliam, Valentin, Wallace, Wilm, Lopez, Thompson, Gibson, and Higgins 2007)) is the use of computationally very expensive sequence alignment. Statistical signatures (Vinga and Almeida 2003) created from base composition frequencies offer an alternative to using classic alignment. These alignment-free methods reduce processing time and look promising for whole genome phylogenetic analysis where previously used methods do not scale well (Thompson, Plewniak, and Poch 1999). However, pure alignment-free methods typically do not provide the desired classification accuracy and do not offer large preprocessed databases which makes the comparison of a sequence with a large set of known sequences impractical.

The position sensitive p -mer frequency clustering techniques developed in this package are

particularly suited to this classification problem, as they require no alignment and scale well for large scale data because it is based on high-throughput data stream clustering techniques resulting in so called quasi-alignments. Also the growth rate of the size of the learned profile models has proven to be sublinear due to the compression achieved by clustering (Kotamarti, Hahsler, Raiford, McGee, and Dunham 2010). Note also that the topology of the model is not predetermined (as for HMMs (Eddy 1998)), but is learned through the associated machine learning algorithms.

2. Using TRACDS for Genomic Applications

Sequence clustering using position sensitive p -mer clustering is based on the idea of computing distances between sequences using p -mer frequency counts instead of computationally expensive alignment between the original sequences. This idea is at the core of so-called alignment-free methods (Vinga and Almeida 2003). However, in contrast to these methods we count p -mer frequencies position specific (i.e., for different segments of the sequence) and then use high-throughput data stream clustering to group similar segments. This approach completely avoids expensive alignment of sequences prior to building the models. Even so, because of the clustering of like sequence segments, a probabilistic local quasi-alignment is automatically achieved.

The occurrences of letters or base compositions $\{A, C, U, G\}$ of a 16S rRNA sequence provide frequency information. The occurrences of all patterns of bases of length p generates a p -mer frequency representation for a sequence. Instead of global frequencies, we count p -mer frequencies locally to retain positional information by first splitting the sequence into segments of a given size L . Within each segment we count the frequencies for all possible p -mers. We call this frequency profile a Numerical Summarization Vector (NSV). For example, suppose we have an input segment containing ACGTGCACG. If counting 2-mers, the NSV count vector would be

$$\langle 0, 2, 0, 0, 1, 0, 2, 0, 0, 1, 0, 1, 0, 0, 1, 0 \rangle$$

representing counts for the subpatterns

AA, AC, AG, AT, CA, CC, CG, CT, GA, GC, GG, GT, TA, TC, TG, and TT.

As we move down the input sequence, in each new segment p -mers are counted. Segment sizes may be varied and may or may not overlap. Also different values for p could be used within the same sequences.

Figure 1 summarizes the model building process. NSVs representing segments are clustered using high-throughput data stream clustering techniques and the sequence information for the NSVs is preserved in a directed graph $G = (N, E)$, where $N = c_1, c_2, \dots, c_N$ is the set of clusters and $E = e_1, e_2, \dots, e_E$ is the set of transitions between clusters. This graph can be interpreted as a Markov Chain, however, unlike a classical Markov Model, each node is not bound to one symbol. In fact, each node represents a cluster consisting of NSVs that are found to be similar during the model building process according to a similarity or dissimilarity metric. Since several NSVs (i.e., segments) can be assigned to the same cluster, the resulting model compresses the original sequence (or sequences if several sequences are clustered into the same model). The directed edges are associated with additional information representing the probabilities of traversal assigned during the model building process.

The similarity between NSVs used for clustering can be calculated using several measures.

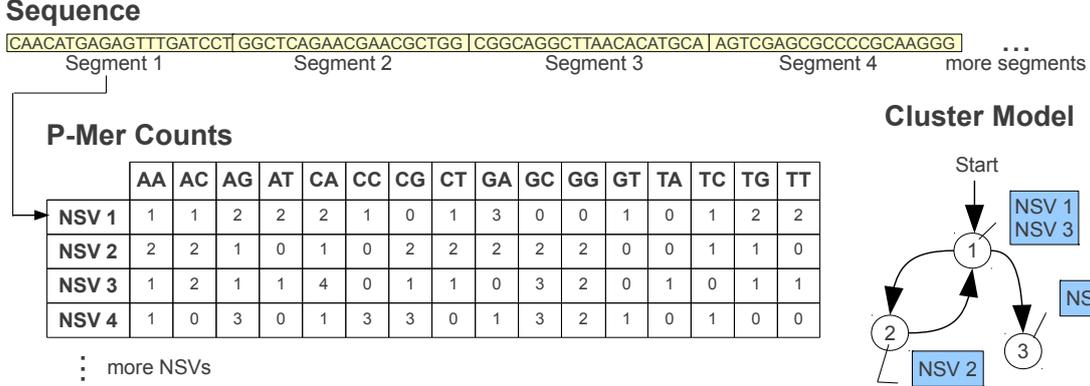


Figure 1: The Model Building Process. The sequence is split into several segments. For each segment a Numerical Summary Vectors (NSV) is calculated by counting the occurrence of p -mers (2-mers in this case). Model building starts with an empty cluster model. As each NSV is processed, it is compared to the existing clusters of the model. If the NSV is not found to be close enough (using a distance measure on the NSVs) a new cluster is created. For example Cluster¹ (circle) is created for NSV¹, Cluster² for NSV² and Cluster³ for NSV³. NSV³ was found close enough to NSV¹ and thus was also assigned to Cluster¹. In addition to the clusters also the transition information between the clusters (arrows) is recorded. When all NSVs are processed, the model building process is finished.

Measures suggested in the literature to compare sequences based on p -mer counts (alignment-free methods) include Euclidean distance, squared Euclidean distance, Kullback-Leibler discrepancy and Mahalanobis distance (Vinga and Almeida 2003). Recently, for Simrank (DeSantis, Keller, Karaoz, Alekseyenko, Singh, Brodie, Pei, Andersen, and Larsen 2011) an even simpler similarity measure, the number of matching p -mers (typically with $p = 7$), was proposed for efficient search of very large database. In the area of approximate string matching Ukkonen proposed the approximate the expensive computation of the edit distance (Levenshtein 1966) between two strings by using q -grams (analog to p -mers in sequences). First q -gram profiles are computed and then the distance between the profiles is calculated using Manhattan distance (Ukkonen 1992). The Manhattan distance between two p -mer NSVs x and y is defined as:

$$d_{\text{Manhattan}}(x, y) = \sum_{i=1}^{4^p} |x_i - y_i|$$

Manhattan distance also has a particularly straightforward interpretation for NSVs. The distance counts the number of p -mers by which two sequences differ which gives the following lower bound on the edit distance between the original sequences s_x, s_y :

$$d_{\text{Manhattan}}(x, y)/(2p) \leq d_{\text{Edit}}(s_x, s_y)$$

This relationship is easy to proof since each insertion/deletion/substitution in a sequences destroys at the most p p -grams and introduces at most p new p -grams. Although, we can

construct two completely different sequences with exactly the same NSVs (see (Ukkonen 1992) for a method for regular strings), we are typically interested in sequences of high similarity in which case $d_{\text{Manhattan}}(x, y)/(2p)$ gets closer to the edit distance. However, not that our approach is not bound to using Manhattan distance, it can be use any distance/similarity measure defined on the frequency counts in NSVs.

A p -mer frequency cluster model can be created for a single sequence and compresses the sequence information first by creating NSVs and then reduces the number of NSVs to the number of clusters needed to represent the whole sequence. Typically, we will create a cluster model for a whole family of sequences by simply adding the NSVs of all sequences to a single model following the procedure in Figure~1. This will lead to even more compression since many sequences within a family will share NSVs stemming from similar sequence segments. We explored this approach for taxonomic classification in (Kotamarti *et al.* 2010).

3. QuasiAlign Package

QuasiAlign builds on the packages **Biostrings** for biological sequence analysis, **RSQLite** and **DBI** for storage and data management functionality, **caTools** for Base64 encoding of models, and **rEMM** for model building using TRACDS (Transitions Among Clusters for Data Streams).

The main components of the package are:

- Data storage an management (GenDB)
- Sequence to NSV conversion
- Model creation
- Visualization
- Classification

3.1. GenDB: Data storage an management

At the heart of the **QuasiAlign** package are genetic databases (GenDB) which are used for efficient storage and retrieval. By default we use s light-weight SQLite databases, but any other compatible database such as MySQL or Oracle can also be used. Figure~2 shows an example of the basic table layout of a GenDB instance with a table containing classification information, a table containing the sequence information and a meta data table. For each sequence we will have an entry in the classification table and an corresponding entry in the sequence table. The tables are connected by a unique sequence ID as the primary key. Classification and sequence are separated because later we will also add tables with NSVs for sequences which will share the classification table.

3.2. Classification

The classification score evaluates how likely it is that a sequence was generated by a given model (Hahsler and Dunham 2012). It is calculated by the length-normalized product or sum

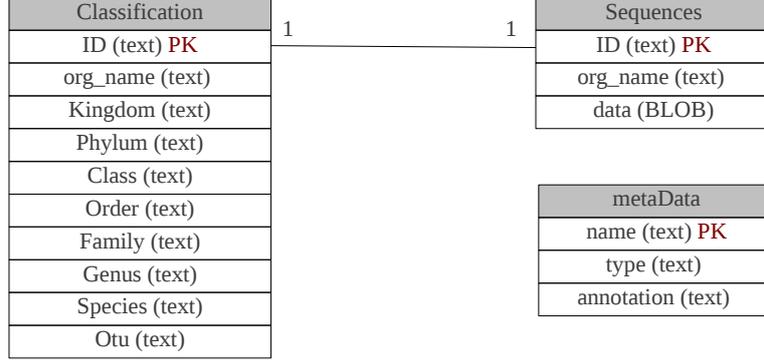


Figure 2: Entity Relationship diagram of genetic database

of probabilities on the path along the new sequence. The scores for a new sequence of length l are defined as:

$$P_{\text{prod}} = \sqrt[l-1]{\prod_{i=1}^{l-1} a_{s(i)s(i+1)}} \quad (1)$$

$$P_{\text{sum}} = \frac{1}{l-1} \sum_{i=1}^{l-1} a_{s(i)s(i+1)} \quad (2)$$

where $s(i)$ is the state the i^{th} NSV in the new sequence is assigned to. NSVs are assigned to the closest cluster. Note that for a sequence of length l we have $l-1$ transitions. If we want to take the initial transition probability also into account we extend the above equations by the additional initial probability $a_{\epsilon,s(1)}$:

$$P_{\text{prod}} = \sqrt[l]{\prod_{i=1}^{l-1} a_{s(i)s(i+1)}} \quad (3)$$

$$P_{\text{sum}} = \frac{1}{l} \left(a_{\epsilon,s(1)} + \sum_{i=1}^{l-1} a_{s(i)s(i+1)} \right) \quad (4)$$

FIXME: State algorithm.

FIXME: Can we compare the score for different models? What if one model is more complicated than the other? It will have automatically a lower score!

FIXME: Models will have a different number of states depending on the data. Can we have different thresholds for different models? Are they still comparable?

FIXME: How do we handle if the closest cluster is very different from the NSV?

FIXME: How do we handle missing states/transitions?

FIXME: How can we incorporate PAM/BLOSUM substitution matrices into distance computation on NSVs (Manhattan on k-gram counts)?

FIXME: How can we compute distance between two models for phylogenetic analysis?

FIXME: How do we know that a score is significant higher than a score created by chance?

3.3. Other components

4. Examples

In the following we will demonstrate the key features of **QuasiAlign** using several examples.

4.1. Setting up a GenDB

First, we load the library into the R environment.

```
R> library(QuasiAlign)
```

To start we need to create an empty GenDB to store and organize sequences.

```
R> db<-createGenDB("example.sqlite")
R> db
```

```
Object of class GenDB with 0 sequences
DB File: example.sqlite
Tables: classification, metaData, sequences
```

The above command creates an empty database with a table structure similar to Figure~2 and stores it in the file `example.sqlite`. If a GenDB already exists, then it can be opened using `openGenDB()`.

The next step is to import sequences into the database by reading FASTA files. This is accomplished by function `addSequences()`. This function automatically extracts the classification information from the FASTA file's description lines. The default is to expect classification in the format used by the Greengenes project, however other meta data readers can be implemented (see manual page for `addSequences`).

The command below uses a FASTA file provided by the package, hence we use `system.file()` instead of just a string with the file name.

```
R> addSequences(db,
+   system.file("examples/phylums/Firmicutes.fasta", package="QuasiAlign"))
```

```
Read 100 entries. Added 100 entries.
```

After inserting the sequences, various querying and limiting functions can be used to check the data and obtain a subset of the sequences. To get a count of the number of sequences in the database, the function `nSequences()` can be used.

```
R> nSequences(db)
```

```
[1] 100
```

The function `getSequences()` returns the sequences as a vector. In the following example we get all sequences in the database and then show the first 50 bases of the first sequence.

```
R> s <- getSequences(db)
R> s
```

```
A DNASTringSet instance of length 100
      width seq                                     names
[1]  1521 TTTGATCCTGGCTCAGG...CGGCTGGATCACCTCCT 1250
[2]  1392 ACGGGTGAGTAACGCGT...TTGGGGTGAAGTCGTAA 13651
[3]  1384 TAGTGGCGGACGGGTGA...TCGAATTTGGGTCAAGT 13652
[4]  1672 GCGCTGCCTAACACATG...TGTAACACGACTTCAT 13654
[5]  1386 ATCTCACCTCTCAATAG...CGAAGGTGGGGTTGGTG 13655
[6]  1438 GCGGACGGGTGAGTAAC...GCTGGATCACCTCCTTA 13657
[7]  1392 ACGGGTGAGTAACGCGT...TTGGGGTGAAGTCGTAA 13658
[8]  1526 AGAGTTTGATCCTGGCT...GCTGGATCACCTCCTTA 13659
[9]  1440 ATCTCACCTCTCAATAG...GCTGGATCACCTCCTTA 13661
...    ...    ...
[92]  1516 GGCTCAGGACGAACGCT...GTAGCCGTTTCGAGAACG 13852
[93]  1506 CGAACGCTGGCGGCGTG...GTAGCCGNTTCGAGAACG 13853
[94]  1505 ATCCTGGCTCAGGACGA...AGTCGTAACAAGGTAGC 13855
[95]  1447 ATGCAAGTCGAACGGGG...GGGGCCGATGATTGGGG 13856
[96]  1446 ATGCAAGTCGAACGGGG...GGGGCCGATGATTGGGG 13857
[97]  1511 ATCCTGGCTCAGGACGA...AGTCGTAACAAGGTAGC 13858
[98]  1544 ATCCTGGCTCAGGACGA...GGTGGATCACCTCCTTC 13860
[99]  1482 GGACGAACGCTGGCGGC...GCCGATGATTGGGGTGA 13861
[100] 1485 GACGAACGCTGGCGGCG...GAAGTCGTAACAAGGTA 13862
```

```
R> length(s)
```

```
[1] 100
```

```
R> s[[1]]
```

```
1521-letter "DNASTring" instance
seq: TTTGATCCTGGCTCAGGACGAACGCTGGCGG...TGTACCGAAGGTGCGGCTGGATCACCTCCT
```

```
R> substr(s[[1]], 1, 50)
```

```
50-letter "DNASTring" instance
seq: TTTGATCCTGGCTCAGGACGAACGCTGGCGGCGTGCCTAATGCATGCAAG
```

Sequences in the database can also be filtered using classification information. For example, we can get all sequences of the genus name “*Desulfosporomusa*” by specifying rank and name.

```
R> s <- getSequences(db, rank="Genus", name="Desulfosporomusa")
R> s
```

```
A DNASTringSet instance of length 7
  width seq                               names
[1] 1498 TNGAGAGTTTGATCCTGG...TGGGGCCGATGATCGGGG 13834
[2] 1481 CTGGCGGCGTGCCTAACA...ATTGGGGTGAAGTCGTAA 13836
[3] 1510 GACGAACGCTGGCGGCGT...AGCCGTATCGGAAGGTGC 13839
[4] 1503 ACGCTGGCGGCGTGCCTA...GGTAGCCGTATCGGAAGG 13844
[5] 1503 ACGCTGGCGGCGTGCCTA...GGTAGCCGTATCGGAAGG 13845
[6] 1429 ACGCTGGCGGCGTGCCTA...GAAGCCGGTGGGGTAACC 13846
[7] 1504 ACGCTGGCGGCGTGCCTA...GGTAGCCGTATCGGAAGG 13847
```

To obtain a single sequence, `getSequences` can be used with `rank` equal to "id" and supplying the sequence's greengenes ID as the name.

```
R> s <- getSequences(db, rank="id", name="1250")
R> s
```

```
A DNASTringSet instance of length 1
  width seq                               names
[1] 1521 TTTGATCCTGGCTCAGGA...GCGGCTGGATCACCTCCT 1250
```

The database also stores a classification hierarchy. We can obtain the classification hierarchy used in the database with `getTaxonomyNames()`.

```
R> getTaxonomyNames(db)

[1] "Kingdom" "Phylum" "Class" "Order" "Family" "Genus"
[7] "Species" "Otu" "Org_name" "Id"
```

To obtain all unique names stored in the database for a given rank we can use `getRank()`.

```
R> getRank(db, rank="Order")
```

```
Order
1 Thermoanaerobacterales
2 Clostridiales
```

The 100~sequences in our example data base contain organisms from 2 different orders. We can obtain the rank name for each sequence individually by using `all=TRUE`. The following code counts how many sequences we have for each genus.

```
R> table(getRank(db, rank="Genus", all=TRUE)[,1])
```

Acidaminococcus	Carboxydothermus	Coprothermobacter
2	2	1
Desulfosporomusa	Desulfotomaculum	Dialister
7	20	3
Mitsuokella	Moorella	Pelotomaculum
1	4	4
Phascolarctobacterium	Selenomonas	Syntrophomonas
2	9	6
Thermacetogenium	Thermaerobacter	Thermoanaerobacter
1	1	10
Thermoanaerobacterium	Thermosinus	Veillonella
8	2	5
unknown		
12		

This information can be easily turned into a barplot showing the abundance of different orders in the data database (see Figure~3).

```
R> oldpar <- par(mar=c(12,5,5,5)) ### make space for labels
R> barplot(sort(
+   table(getRank(db, rank="Genus", all=TRUE)[,1]),
+   decreasing=TRUE), las=2)
R> par(oldpar)
```

Filtering also works for `getRank()`. For example, we can find the genera within the order “Thermoanaerobacterales”.

```
R> getRank(db, rank="Genus",
+   whereRank="Order", whereName="Thermo")
```

	Genus
1	Coprothermobacter
2	Moorella
3	Thermacetogenium
4	Carboxydothermus
5	Thermoanaerobacter

Note that partial matching is performed from “Thermo” to “Thermoanaerobacterales.” Partial matching is available for ranks and names in most operations in **QuasiAlign**.

We can also get the complete classification hierarchy for different ranks down to individual sequences. In the following we get the classification hierarchy for genus *Thermaerobacter*, then all orders matching *Therm* and then for a list of names.

```
R> getHierarchy(db, rank="Genus", name="Thermaerobacter")
```

Kingdom	Phylum	Class
"Bacteria"	"Firmicutes"	"Clostridia"

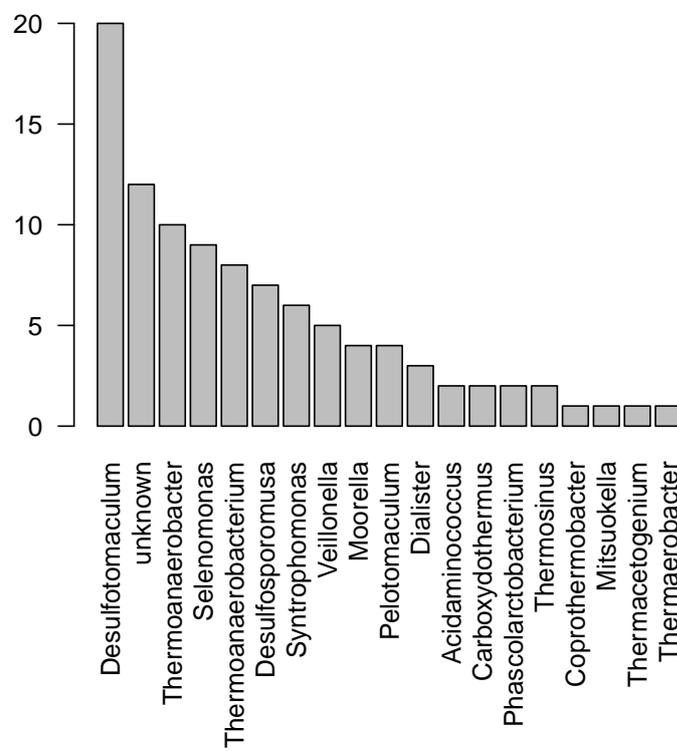


Figure 3: Abundance of different orders in the database.

Order	Family	Genus
"Clostridiales"	"Sulfobacillaceae"	"Thermaerobacter"
Species	Otu	Org_name
NA	NA	NA
Id		
NA		

```
R> getHierarchy(db, rank="Genus", name="Therm")
```

Kingdom	Phylum	Class	Order	
[1,] "Bacteria"	"Firmicutes"	"Clostridia"	"Thermoanaerobacterales"	
[2,] "Bacteria"	"Firmicutes"	"Clostridia"	"Clostridiales"	
[3,] "Bacteria"	"Firmicutes"	"Clostridia"	"Clostridiales"	
[4,] "Bacteria"	"Firmicutes"	"Clostridia"	"Thermoanaerobacterales"	
[5,] "Bacteria"	"Firmicutes"	"Clostridia"	"Clostridiales"	
Family				
[1,] "Thermoanaerobacteraceae"				
[2,] "Sulfobacillaceae"				
[3,] "Thermoanaerobacterales Family III. Incertae Sedis"				
[4,] "Thermoanaerobacteraceae"				
[5,] "Veillonellaceae"				
Genus	Species	Otu	Org_name	Id
[1,] "Thermacetogenium"	NA	NA	NA	NA
[2,] "Thermaerobacter"	NA	NA	NA	NA
[3,] "Thermoanaerobacterium"	NA	NA	NA	NA
[4,] "Thermoanaerobacter"	NA	NA	NA	NA
[5,] "Thermosinus"	NA	NA	NA	NA

```
R> getHierarchy(db, rank="Genus", name=c("Acid", "Thermo"))
```

Kingdom	Phylum	Class	Order	
[1,] "Bacteria"	"Firmicutes"	"Clostridia"	"Clostridiales"	
[2,] "Bacteria"	"Firmicutes"	"Clostridia"	"Clostridiales"	
[3,] "Bacteria"	"Firmicutes"	"Clostridia"	"Thermoanaerobacterales"	
[4,] "Bacteria"	"Firmicutes"	"Clostridia"	"Clostridiales"	
Family				
[1,] "Veillonellaceae"				
[2,] "Thermoanaerobacterales Family III. Incertae Sedis"				
[3,] "Thermoanaerobacteraceae"				
[4,] "Veillonellaceae"				
Genus	Species	Otu	Org_name	Id
[1,] "Acidaminococcus"	NA	NA	NA	NA
[2,] "Thermoanaerobacterium"	NA	NA	NA	NA
[3,] "Thermoanaerobacter"	NA	NA	NA	NA
[4,] "Thermosinus"	NA	NA	NA	NA

To get individual sequences we can use again the unique sequence id.

```
R> getHierarchy(db, rank="id", name="1250")

      Kingdom
      "Bacteria"
      Phylum
      "Firmicutes"
      Class
      "Clostridia"
      Order
      "Thermoanaerobacterales"
      Family
      "Thermodesulfobiaceae"
      Genus
      "Coprothermobacter"
      Species
      "unknown"
      Otu
      "otu_2281"
      Org_name
      "X69335.1Coprothermobacterproteolyticustr.ATCC35245"
      Id
      "1250"
```

4.2. Converting Sequences to NSV

In order to create position sensitive p -mer clustering models, we need to first create Numerical Summarization Vectors (NSVs). The **QuasiAlign** package can easily convert large number of sequences in the database to NSV format and store them in the same database. The following command will convert all the sequences to NSV format and store them in a table called NSV.

```
R> createNSVTable(db, table = "NSV")
```

```
CreateNSVTable: Read 100 entries (ok: 100 / fail: 0 )
```

```
CreateNSVTable: Read 100 entries. Added 100 entries.
```

In the function call above we used the default values for most of the parameters such as word, overlap, and last_window. Custom parameter settings and filter criteria can be easily specified in the following way:

```
R> createNSVTable(db, table = "NSV_genus_Thermosinus",
+   rank = "genus", name = "Thermosinus",
+   window = 100, overlap = 0, word = 3, last_window = FALSE)
```

```
CreateNSVTable: Read 2 entries. Added 2 entries.
```

```
R> db
```

```
Object of class GenDB with 100 sequences
DB File: example.sqlite
Tables: NSV, NSV_genus_Thermosinus, classification, metaData, sequences
```

The above command converts only the sequences that belong to the genus “Thermosinus” and stores them in a separate NSV table called NSV_genus_Thermosinus. The parameters for creating NSVs are also part of the command, such as window size is 100, overlap is 0, word size is 3, and last_window parameter is FALSE indicating that the last (incomplete) window will be ignored.

When a new sequence or NSV table is created, its name and meta information is stored in the metaData table. The meta data can be queried using the `metaGenDB()` function.

```
R> metaGenDB(db)

      name      type
1      sequences sequence
2           NSV     NSV
3 NSV_genus_Thermosinus     NSV

                                annotation
1
2           rank=;name=;window=100;overlap=0;word=3;last_window=FALSE;
3 rank=genus;name=Thermosinus;window=100;overlap=0;word=3;last_window=FALSE;
```

The annotation column contains information about how the NSVs were created.

The sequences in the NSV tables can be queried and filtered using `getSequences()` in the same way as regular sequences, however, the result is an object of class NSVSet.

```
R> NSVs <- getSequences(db, rank="Genus", name="Desulfosporomusa", table="NSV")
R> NSVs
```

```
Object of class NSVSet for 7 sequences (3-mers)
Number of segments (table with counts):
14 15
 3  4
```

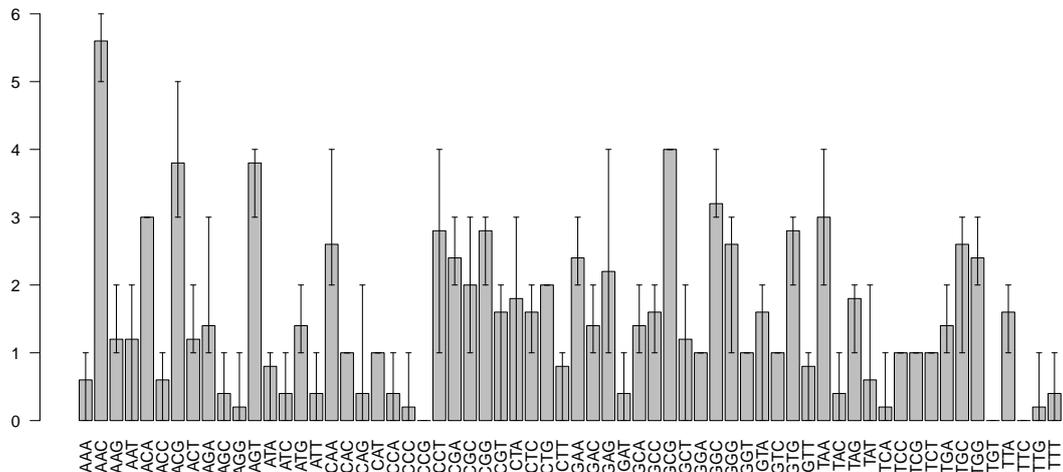
```
R> length(NSVs)
```

```
[1] 7
```

```
R> names(NSVs)
```

```
[1] "13834" "13836" "13839" "13844" "13845" "13846" "13847"
```

The code above selects the NSVs for the genus “Desulfosporomusa” in table NSV. Note sequences of NSVs are not strings like the original sequences but tables of p -mer counts and thus are stored internally in a list. The code below shows the dimensions of the NSV table for the first sequence and then shows the first 2 rows and 16 columns of the table.

Figure 4: p -mer frequency plot.

```
R> dim(NSVs[[1]])
```

```
[1] 14 64
```

```
R> NSVs[[1]][1:5,1:16]
```

	AAA	AAC	AAG	AAT	ACA	ACC	ACG	ACT	AGA	AGC	AGG	AGT	ATA	ATC	ATG	ATT
[1,]	0	5	1	1	3	0	3	1	3	1	1	4	1	1	1	0
[2,]	3	5	3	2	3	3	1	2	3	0	0	2	1	0	2	0
[3,]	1	0	3	0	1	1	1	0	2	3	3	2	0	2	2	1
[4,]	1	1	1	2	2	0	5	3	2	2	2	2	0	1	2	0
[5,]	2	1	4	3	0	1	3	0	1	1	3	2	1	0	2	2

However, regular subsetting is also implemented on NSVSets. For example, we can directly select for the first five sequences the first segment and then create a barplot showing the p -mer frequencies including whiskers for the minimum and maximum appearance in sequences.

```
R> NSVs[1:5,1]
```

```
Object of class NSVSet for 5 sequences (3-mers)
```

```
Number of segments (table with counts):
```

```
1
```

```
5
```

```
R> plot(NSVs[1:5,1])
```

Finally, we can close a GenDB after we are done working with it. The database can later be reopened using `openGenDB()`.

```
R> closeGenDB(db)
```

To permanently remove the database we need to delete the file (for SQLite databases) or remove the database using the administrative tool for the database management system.

```
R> unlink("example.sqlite")
```

FIXME: Is there a purge function in DBI to do this?

Often, we would like to convert sequences from many FASTA files into NSV format in the database in a single step. The convenience function `processSequences()` loads all FASTA files from a directory into the database and then converts them into NSVs.

```
R> db<-createGenDB("example.sqlite")
```

```
R> processSequences(system.file("examples/phylums", package="QuasiAlign"), db)
```

```
Processing file: /tmp/Rtmpyh6x6g/Rinst5c1931a08740/QuasiAlign/examples/phylums/Firmicutes
```

```
Read 100 entries. Added 100 entries.
```

```
Processing file: /tmp/Rtmpyh6x6g/Rinst5c1931a08740/QuasiAlign/examples/phylums/Planctomyc
```

```
Read 100 entries. Added 100 entries.
```

```
Processing file: /tmp/Rtmpyh6x6g/Rinst5c1931a08740/QuasiAlign/examples/phylums/Proteobact
```

```
Read 100 entries. Added 100 entries.
```

```
CreateNSVTable: Read 100 entries (ok: 100 / fail: 0 )
```

```
CreateNSVTable: Read 200 entries (ok: 200 / fail: 0 )
```

```
CreateNSVTable: Read 300 entries (ok: 300 / fail: 0 )
```

```
CreateNSVTable: Read 300 entries. Added 300 entries.
```

Additional parameters (e.g., window or word) will be passed on to creating the NSVs.

4.3. Creating a model

The NSVs created in the previous section can be used for model generation. The models can be created at for all sequences in the database or for any set of sequences selected using filters.

```
R> model <- GenModelDB(db, rank="Genus", name="Desulfosporomusa", table="NSV",
+   measure="Manhattan", threshold =30)
```

```
GenModel: Creating model for Genus: Desulfosporomusa
```

```
GenModel: Processed 7 sequences
```

```
R> model
```

```
Object of class GenModel with 7 sequences
```

```
Genus : Desulfosporomusa
```

```
Model:
```

```
EMM with 42 states/clusters.
```

```
Measure: Manhattan
```

```
Threshold: 30
```

```
Centroid: FALSE
```

```
Lambda: 0
```

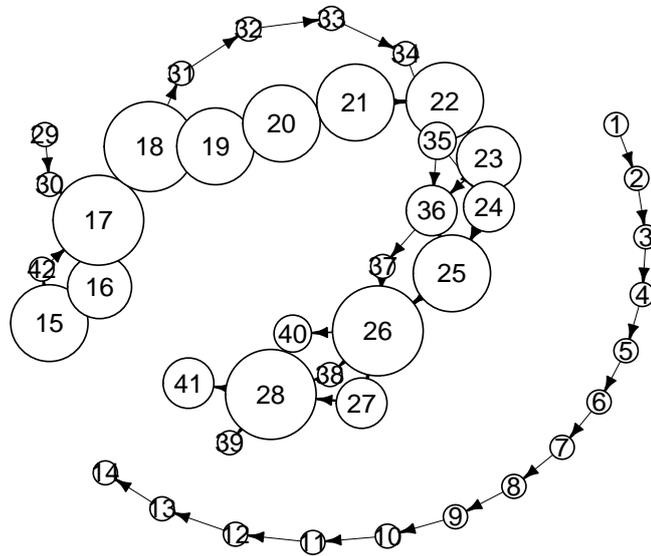


Figure 5: Default plot of a model as a graph using a standard graph-layout algorithm.

The above command builds a model using a subset of the sequences in the NSV table that belong to the genus “Desulfosporomusa”. For creating the model, we use Manhattan distance with a threshold of 30 for clustering NSVs. For more details about model creation, please see the reference manual of the rEMM package ([Hahsler and Dunham 2012](#)). In addition a limit parameter can be used to restrict the maximum number of sequences to be used in model creation.

The model is a compact signature of the sequences and can be easily and efficiently used for analysis. It can be plotted to get a visual display of the various states and transitions using `plot()`.

```
R> plot(model)
```

```
R> plot(model, method="MDS")
```

```
R> plot(model, method="graph")
```

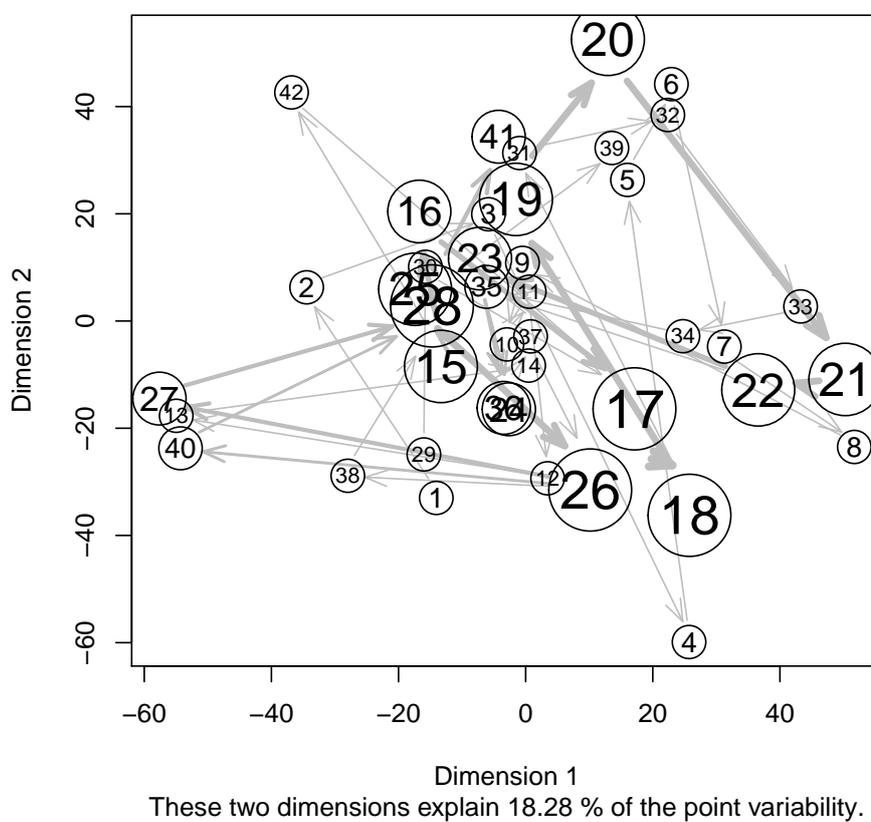


Figure 6: Plot of the model using MDS do display more similar clusters closer together.

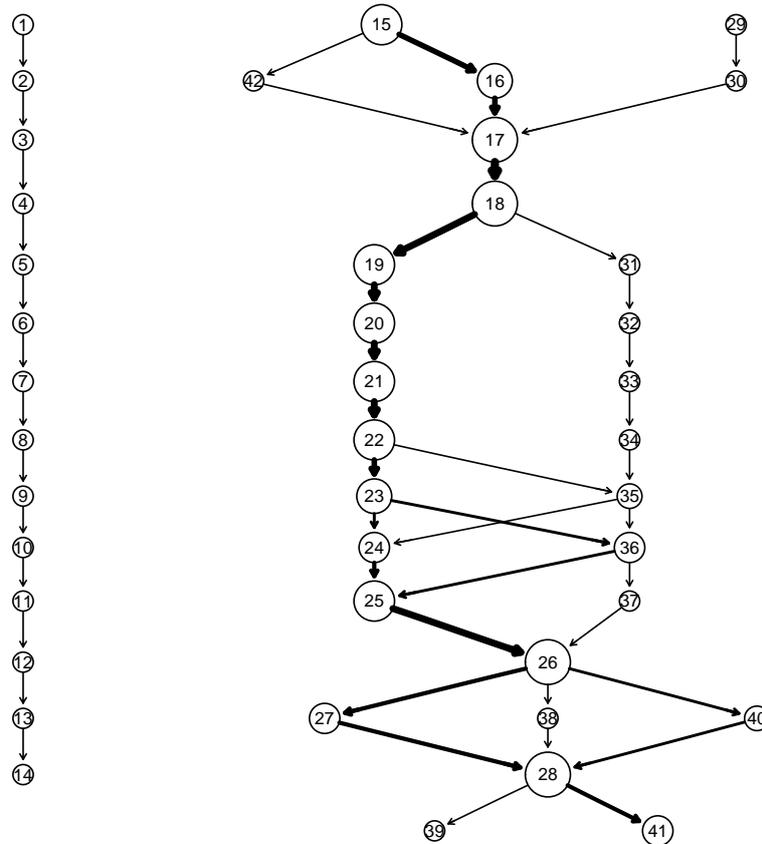


Figure 7: Plot of the model using Graphviz .

The default plot which displays the model as a graph is shown in Figure~5. A plot where the clusters are arranged using multi-dimensional scaling to place similar clusters closer together is shown in Figure~6. Figure~7 shows the model using the Graphviz library.

4.4. Classification

To classify a new sequence at a particular rank level (e.g., at the phylum level), we need to have the set of all models at this level to classify against. For example, we need to create models for each phylum present in the database for predicting the phylum of an unclassified sequence. This can be accomplished using `createModels()`, which creates a set of models and stores them in a directory specified by the `modelDir` parameter.

The following command creates models for all phylums stored in the database and stores them in directory `models` (which is created first) and places them in the subdirectory `phylum`.

```
R> dir.create("models")
R> createModels(modelDir="models", rank="phylum", db)
```

```
GenModel: Creating model for phylum: Firmicutes
GenModel: Processed 100 sequences
GenModel: Creating model for phylum: Planctomycetes
GenModel: Processed 100 sequences
GenModel: Creating model for phylum: Proteobacteria
GenModel: Processed 100 sequences
```

The models are now stored as compressed files.

```
R> list.files("models/phylum")

[1] "Firmicutes.rds"      "Planctomycetes.rds" "Proteobacteria.rds"
```

A model can be loaded using the `readRDS()`.

```
R> model <- readRDS("models/phylum/Firmicutes.rds")
R> model
```

```
Object of class GenModel with 100 sequences
Phylum : Firmicutes
```

```
Model:
EMM with 619 states/clusters.
  Measure: Manhattan
  Threshold: 30
  Centroid: FALSE
  Lambda: 0
```

Once all models have been constructed, they can be used to score and classify new sequences. We can compare the new sequences against just one model or all the models stored in a

directory using `scoreSequence()`. Below, we use the `getSequences()` to get 5 random sequences from the NSV table and then score them against the model for “Firmicutes” using the function `scoreSequence()`.

```
R> random_sequences <- getSequences(db, table="NSV", limit=5, random=TRUE)
R> random_sequences
```

Object of class NSVSet for 5 sequences (3-mers)

Number of segments (table with counts):

14

5

```
R> scoreSequence(model, random_sequences)
```

```
      4439      4451      4529      13816      2777
0.07692 0.15385 0.07692 1.00000 0.07692
```

The default method for scoring a sequence against a model is the *supported_transitions* method. It can be changed by the *method* parameter in the `scoreSequence()`. To find the actual classification, we can use the Greengenes ids of the sequences. The code snippet below illustrated this:

```
R> ids <- names(random_sequences)
R> hierarchy <- getHierarchy(db, rank="id", name=ids)
R> hierarchy[, "Phylum"]
```

```
[1] "Proteobacteria" "Proteobacteria" "Proteobacteria"
[4] "Firmicutes"      "Planctomycetes"
```

We can see that those sequences that belong to the phylum “Firmicutes” have the highest score of 1.0. The above commands also shows how easily the actual classification hierarchy of a sequence can be easily obtained using the `getHierarchy()` for those sequences whose Greengenes id is known.

The function `classify()` can be used to classify sequences in NSV format against all the models stored in a directory. It returns a data.frame containing the score matrix and the actual and predicted ranks.

```
R> unknown <- getSequences(db, table="NSV", rank="Phylum", limit=5, random=TRUE)
R> classification <- classify(modelDir="models", unknown, rank="Phylum")
```

```
classify: Creating score matrix for Firmicutes
classify: Creating score matrix for Planctomycetes
classify: Creating score matrix for Proteobacteria
```

```
R> classification
```

```
$scores
      Firmicutes Planctomycetes Proteobacteria
4476    0.09091         0.0000         1.00000
13845   1.00000         0.0000         0.07143
2780    0.14286         0.8571         0.07143
35108   0.00000         1.0000         0.07692
4480    0.23077         0.2308         0.69231
```

```
$prediction
      id predicted      actual
[1,] "4476"  "Proteobacteria" "Proteobacteria"
[2,] "13845" "Firmicutes"      "Firmicutes"
[3,] "2780"  "Planctomycetes" "Planctomycetes"
[4,] "35108" "Planctomycetes" "Planctomycetes"
[5,] "4480"  "Proteobacteria" "Proteobacteria"
```

```
R> table(classification$prediction[, "actual"],
+        classification$prediction[, "predicted"])
```

	Firmicutes	Planctomycetes	Proteobacteria
Firmicutes	1	0	0
Planctomycetes	0	2	0
Proteobacteria	0	0	2

The default method for `classify()` is `supported_transitions`, but other methods can also be easily used.

4.5. Assessing classification accuracy

For validation we split the data into a training and test set. We use the training sequences for generating the models and then evaluate classification accuracy on the hold out test set. This is implemented in function `validateModels()`. The parameter `pctTest` is used to specify the fraction of sequences to be used for test dataset.

```
R> validation <- validateModels(db, modelDir="models", rank="phylum", pctTest=.1)
```

```
GenModel: Creating model for phylum: Firmicutes
GenModel: Processed 90 sequences
GenModel: Creating model for phylum: Planctomycetes
GenModel: Processed 90 sequences
GenModel: Creating model for phylum: Proteobacteria
GenModel: Processed 90 sequences
classify: Creating score matrix for Firmicutes
classify: Creating score matrix for Planctomycetes
classify: Creating score matrix for Proteobacteria
```

```
R> head(validation$scores)
```

	Firmicutes	Planctomycetes	Proteobacteria
13655	0.25000	0.16667	0.08333
13677	0.30769	0.07692	0.23077
13687	0.23077	0.00000	0.38462
13691	0.08333	0.08333	0.50000
13762	0.07692	0.07692	0.53846
13812	0.15385	0.07692	0.15385

```
R> head(validation$prediction)
```

	id	predicted	actual
[1,]	"13655"	"Firmicutes"	"Firmicutes"
[2,]	"13677"	"Firmicutes"	"Firmicutes"
[3,]	"13687"	"Proteobacteria"	"Firmicutes"
[4,]	"13691"	"Proteobacteria"	"Firmicutes"
[5,]	"13762"	"Proteobacteria"	"Firmicutes"
[6,]	"13812"	"Firmicutes"	"Firmicutes"

```
R> table(validation$prediction[, "actual"],
+       validation$prediction[, "predicted"])
```

	Firmicutes	Planctomycetes	Proteobacteria
Firmicutes	6	1	3
Planctomycetes	1	8	1
Proteobacteria	0	0	10

The function `validateModels()` returns a list of two vectors - the first containing the scores of the test sequences against the training models and the second containing the predicted rank of the sequences based on the highest score. The prediction vector can be used to find the classification accuracy of the models.

4.6. Visualizing Sequences and NSVs

One of the unique advantages of QuasiAlign is that it is able to cluster similar sequences very rapidly and accurately as compared to other methods such as multiple sequence alignment. In this section, we will show how QuasiAlign can be used to plot sequences and visualize similar portions or segments of sequences. Since we already have clusters and associated sequence details available as metadata information in the model, visualization is extremely fast and rapid. Thus it is a very powerful and efficient alternative to multiple sequence alignment. It can provide a visual clue about similar portions of sequences or areas of a sequence which are highly conserved across species.

Each model has associated metadata which gives a list of sequences. Each list contains a vector of states to which the corresponding segments are classified to. Here is an example:

```
R> model <- GenModelDB(db, rank="Gen", name="Syntro")
```

```
GenModel: Creating model for Gen: Syntro
GenModel: Processed 6 sequences
```

```
R> model$clusterInfo
```

```
$`13685`
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

```
$`13687`
```

```
[1] 15 16 17 18 19 20 21 22 23 24 25 26 27 28
```

```
$`13688`
```

```
[1] 29 30 17 31 19 32 21 22 23 33 25 26 27 34
```

```
$`13689`
```

```
[1] 35 36 17 37 19 38 39 22 23 40 25 26 27 41
```

```
$`13690`
```

```
[1] 1 42 3 4 5 6 43 8 9 10 11 12 13 14
```

```
$`13692`
```

```
[1] 44 45 46 17 47 19 48 49 50 23 51 25 52 53 54
```

In certain cases, we are interested in finding details about which sequences and segments are part of a state. This can be easily obtained using the `getModelDetails()`.

```
R> getModelDetails(model, state=17)
```

	sequence	segment
1	13687	3
2	13688	3
3	13689	3
4	13692	4

The above command gives the sequences and the segments within them that are part of state 17. The *state* parameter can be omitted to obtain details of all the states. To see the actual segments that are part of a state, the function `getModelSequences()` can be used. It can provide both the DNA sequences as well as the NSV for a state. The commands below illustrate this.

```
R> sequences <- getModelSequences(db, model, state=17, table="sequences")
R> sequences
```

```
A DNAStrngSet instance of length 4
```

	width	seq	names
[1]	100	CAGTAGCCGGCCTGAGAG...ATTGCGCAATGGGGGAAA	13687
[2]	100	GCAACGATCAGTAGCCGG...TGGGGAATATTGCGCAAT	13688
[3]	100	TCAGTAACCGACCTGAGA...TATTGCTCAATGGGGGAA	13689
[4]	100	GAGAGGTTGGACGGCCAC...GGGAAACCTCGACGCAGC	13692

```
R> nsv <- getModelSequences(db, model, state=17, table="NSV")
R> nsv[[1]]
```

	AAA	AAC	AAG	AAT	ACA	ACC	ACG	ACT	AGA	AGC	AGG	AGT	ATA	ATC	ATG	ATT
[1,]	1	0	0	2	2	0	3	3	3	2	2	2	1	0	1	1
	CAA	CAC	CAG	CAT	CCA	CCC	CCG	CCT	CGA	CGC	CGG	CGT	CTA	CTC	CTG	CTT
[1,]	1	3	4	0	2	1	1	2	0	1	4	0	1	1	3	0
	GAA	GAC	GAG	GAT	GCA	GCC	GCG	GCT	GGA	GGC	GGG	GGT	GTA	GTC	GTG	GTT
[1,]	2	4	4	0	3	4	1	0	5	4	8	1	1	0	2	0
	TAA	TAC	TAG	TAT	TCA	TCC	TCG	TCT	TGA	TGC	TGG	TGT	TTA	TTC	TTG	TTT
[1,]	0	1	1	1	0	1	0	0	2	1	4	0	0	0	1	0

A clearer picture about the clustering will emerge when we visualize the sequences and segments and see how they fit into clusters. In this section, we will introduce some of the plots that are commonly used in the Biological sciences such as the Sequence Logo plot. We will also use the barplot to plot the distribution of NSVs in a cluster of sequences.

At the sequence level we can inspect consensus sequences using the function `consensusString()`.

```
R> consensus <- consensusString(sequences)
R> substring(consensus, 1, 20)
```

```
[1] "BMRDNRVYSRVYNKVVVRVVS"
```

Similarly, consensus matrix can also be created using all the combinations or just the DNA bases.

```
R> consensusMat <- consensusMatrix(sequences, baseOnly=TRUE)
R> consensusMat[,1:10]
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]
A	0	2	2	2	1	1	2	0	0	1
C	1	2	0	0	1	0	1	2	2	0
G	2	0	2	1	1	3	1	0	2	3
T	1	0	0	1	1	0	0	2	0	0
other	0	0	0	0	0	0	0	0	0	0

In certain cases, we might want to visually see which sequences and segments are classified together. For that, the function `modelStatesPlot()` comes handy. It can take one or more sequences and visually show which sequences are part of it.

```
R> modelStatesPlot(model, states=17)
```

We can also visualize more than one states easily.

```
R> modelStatesPlot(model, states=3:6)
```

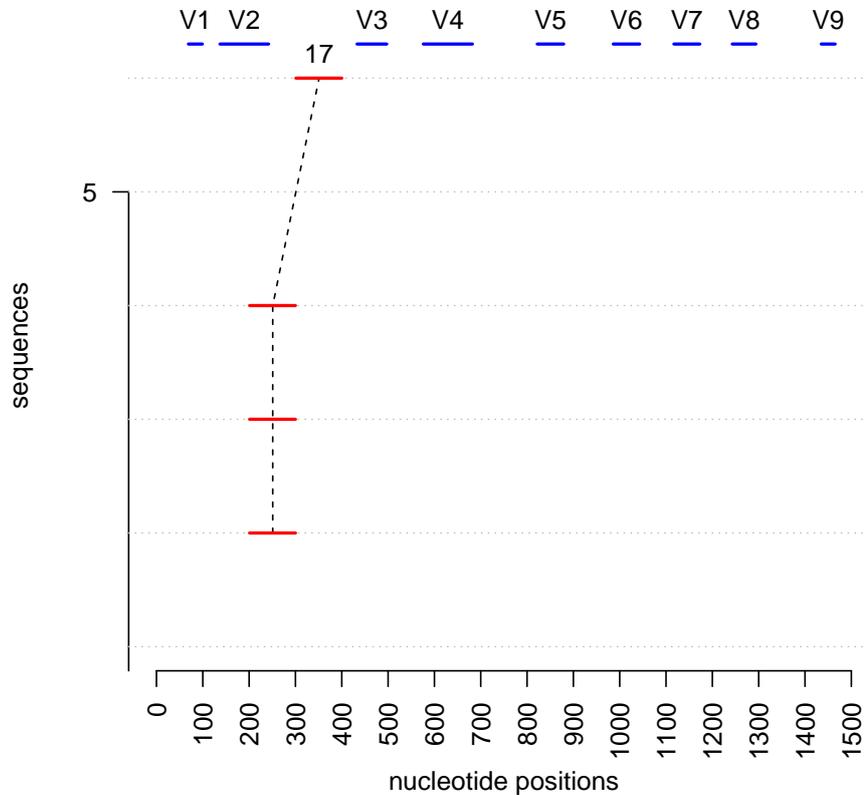


Figure 8: Plot showing the sequences and segments which are part of state 17

One of the more popular approaches in bioinformatics for comparing and analyzing sequences is **Multiple Sequence Alignment**. It is often a computationally expensive process involving comparing each base one at a time between pairs of sequences. QuasiAlign is based on a rapid clustering approach and is thus able to accomplish rapid clustering and classification of sequences. The function `compareSequences()` can compare two or more genetic sequences and find areas of similarity between them. It visually depicts segments (or windows) which are clustered together in a state.

```
R> compareSequences(model, sequences=c(2,3))
```

Figure 10 shows a comparison of sequences 2 and 3. It can be thought of as similar to Sequence Alignment, but we deal with similar windows (or segments of sequences) rather than individual bases. The labels on the windows show the state to which they are classified to. This approach is computationally more efficient than the approach taken by sequence alignment. Similarly, we can easily compare more than two sequences. Figure 11 shows how we can compare more than two sequences by modifying the `sequences` parameter in the `compareSequences()` function. Note that the `sequences` parameter takes the index of the number of the sequences.

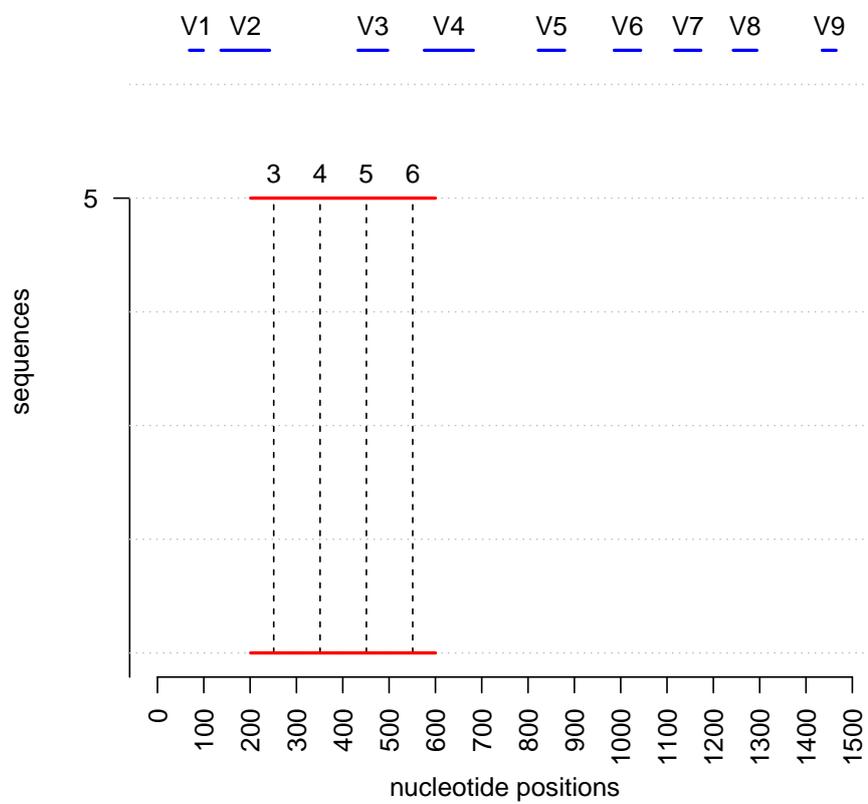


Figure 9: Plot showing the sequences and segments which are part of states 3 through 6

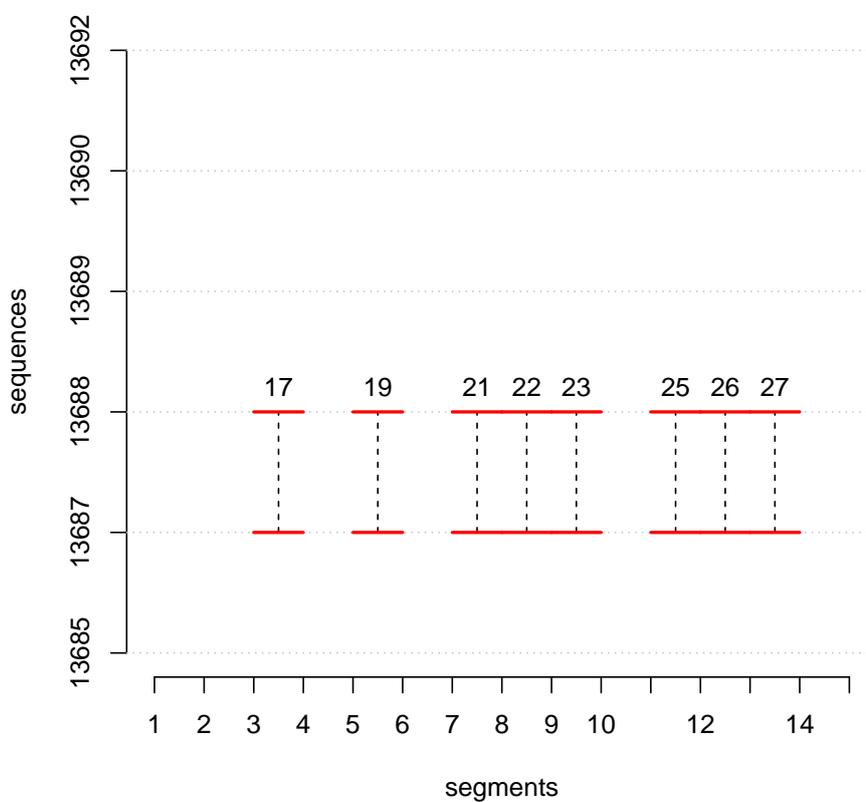


Figure 10: Comparing Sequences of the model to visually analyze similar areas.

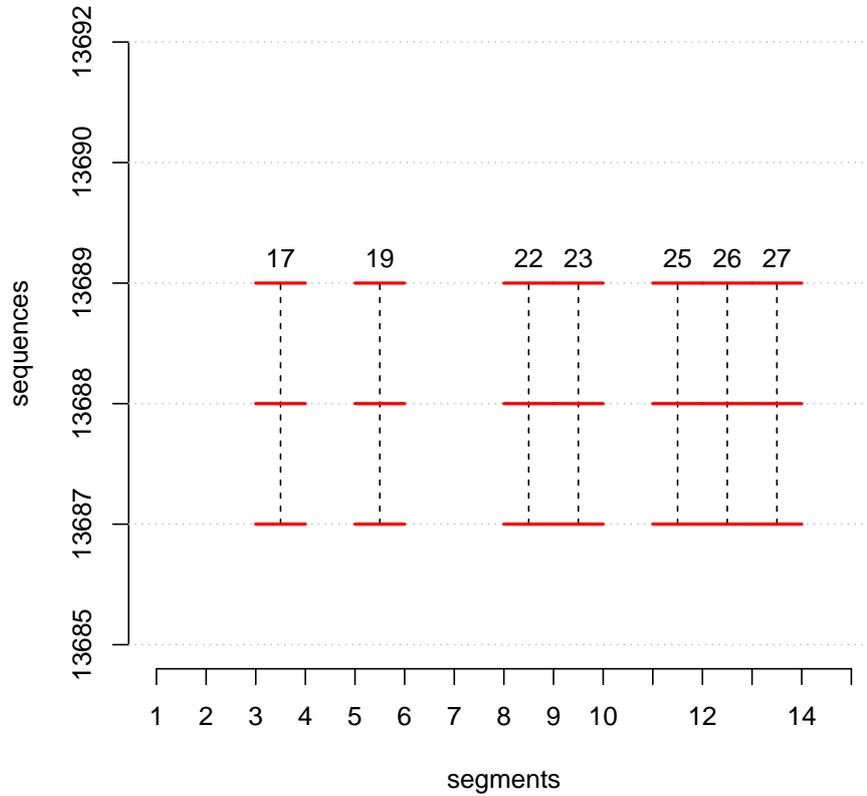


Figure 11: Comparing Multiple Sequences of the model to visually analyze similar areas.

```
R> compareSequences(model, sequences=c(2:4))
```

4.7. Finding conserved segments across sequences

One of the problems in bioinformatics is finding sequences or segments across sequences that are highly similar. This is especially useful in functional and phylogenetic analysis of species and sequences. As a preliminary step, QuasiAlign includes a function `findLargestCommon()` which searches for two or more sequences in a model that have the largest common portions. The common areas are identified by getting a count of common segments and then finding those sequences that share the maximum number of common segments.

```
R> common <- findLargestCommon(model, limit=5)
R> common
```

```
[[1]]
NULL
```

```
[[2]]
[1] 1 5

[[3]]
[1] 2 3 4

[[4]]
[1] 2 3 4 6

[[5]]
integer(0)
```

In the above code, the function `findLargestCommon()` outputs a list containing sequences which share the maximum number of states. For example, the **second** element of the list would give the **two** sequences which share the maximum number of states and so on. It is possible to find the two most similar sequences by the following command:

```
R> compareSequences(model, common[[2]])
```

The output in Figure 12 shows that the sequences 13685 (index=1) and 13690 (index=5) share the most number of states and are likely very similar in functions and origin. In a similar way, we can find out the three most similar sequences. Figure 13 shows the results.

```
R> compareSequences(model, common[[3]])
```

Acknowledgments

This research is supported by research grant no. R21HG005912 from the National Human Genome Research Institute (NHGRI / NIH).

References

- Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990). “Basic local alignment search tool.” *Journal of Molecular Biology*, **215**(3), 403–410. ISSN 0022-2836.
- DeSantis T, Keller K, Karaoz U, Alekseyenko A, Singh N, Brodie E, Pei Z, Andersen G, Larsen N (2011). “Simrank: Rapid and sensitive general-purpose k-mer search tool.” *BMC Ecology*, **11**(1). ISSN 1472-6785. doi:10.1186/1472-6785-11-11.
- Eddy SR (1998). “Profile hidden Markov models.” *Bioinformatics*, **14**(9), 755–763. ISSN 1367-4803.
- Edgar R (2004a). “MUSCLE: a multiple sequence alignment method with reduced time and space complexity.” *BMC Bioinformatics*, **5**(1), 113+. ISSN 1471-2105.

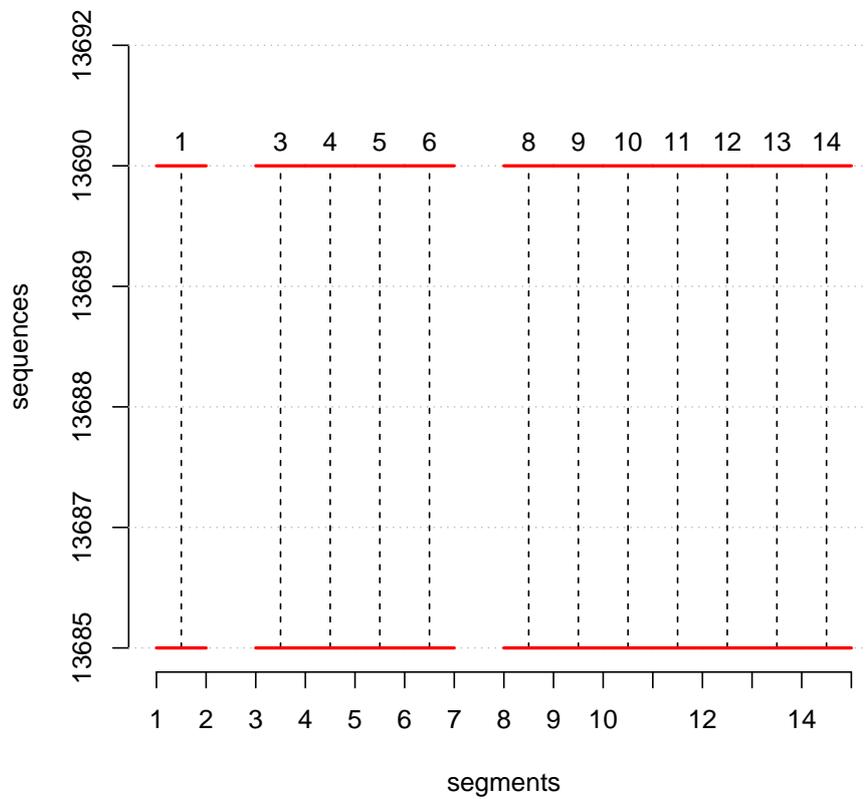


Figure 12: The function `findLargestCommon()` can be used to find the most common sequences. In this figure, we check for the 2 most similar sequences.

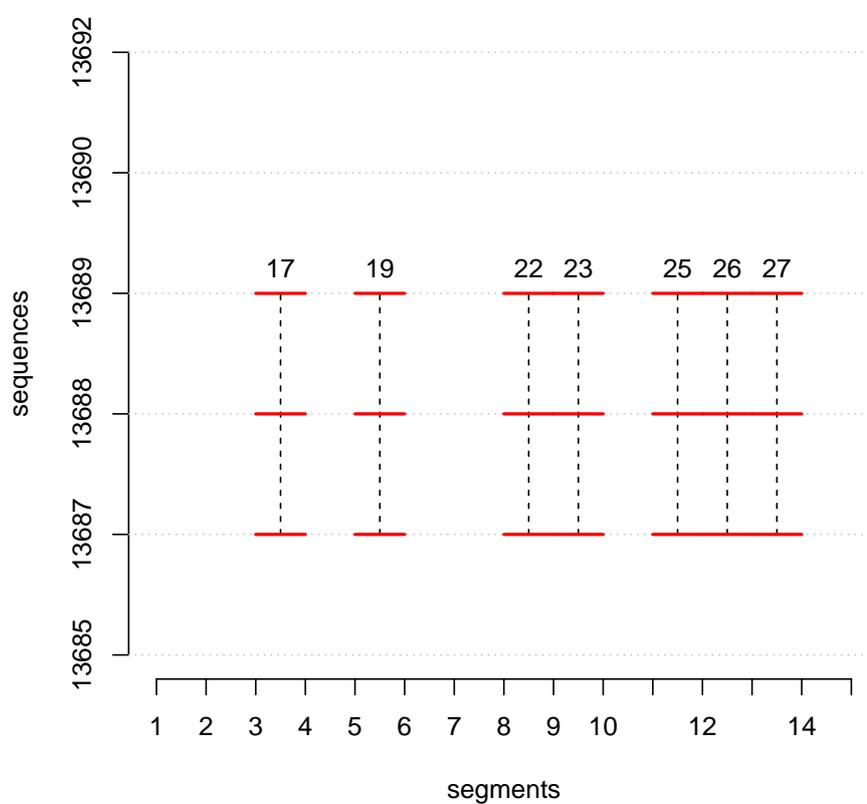


Figure 13: The function `findLargestCommon()` can be used to find the most common sequences. In this figure, we check for the 3 most similar sequences.

- Edgar RC (2004b). “Muscle: multiple sequence alignment with high accuracy and high throughput.” *Nucleic Acids Research*, **32**, 1792–1797.
- Hahsler M, Dunham MH (2012). *rEMM: Extensible Markov Model for Data Stream Clustering in R*. R package version 1.0-3., URL <http://CRAN.R-project.org/>.
- Handelsman J, Rondon MR, Brady SF, Clardy J, Goodman RM (1998). “Molecular biological access to the chemistry of unknown soil microbes: a new frontier for natural products.” *Chemistry and Biology*, **5**(10), 245–249. ISSN 1074-5521.
- Katoh K, Misawa K, Kuma K, Miyata T (2002). “MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform.” *Nucleic Acids Research*, **30**(14), 3059–3066.
- Kotamarti RM, Hahsler M, Raiford D, McGee M, Dunham MH (2010). “Analyzing Taxonomic Classification Using Extensible Markov Models.” *Bioinformatics*, **26**(18), 2235–2241. doi: [10.1093/bioinformatics/btq349](https://doi.org/10.1093/bioinformatics/btq349).
- Larkin M, Blackshields G, Brown N, Chenna R, McGettigan P, McWilliam H, Valentin F, Wallace I, Wilm A, Lopez R, Thompson J, Gibson T, Higgins D (2007). “Clustal W and Clustal X version 2.0.” *Bioinformatics*, **23**, 2947–2948. ISSN 1367-4803.
- Lassmann T, Sonnhammer EL (2006). “Kalign, Kalignvu and Mumsa: web servers for multiple sequence alignment.” *Nucleic Acids Research*, **34**. ISSN 1362-4962.
- Levenshtein V (1966). “Binary Codes Capable of Correcting Deletions, Insertions and Reversals.” *Soviet Physics Doklady*, **10**.
- Mai V, Ukhanova M, Baer DJ (2010). “Understanding the Extent and Sources of Variation in Gut Microbiota Studies; a Prerequisite for Establishing Associations with Disease.” *Diversity*, **2**(9), 1085–1096. ISSN 1424-2818.
- Notredame C, Higgins DG, Heringa J (2000). “T-Coffee: A novel method for fast and accurate multiple sequence alignment.” *Journal of Molecular Biology*, **302**(1), 205–217. ISSN 0022-2836.
- Smith TF, Waterman MS (1981). “Identification of common molecular subsequences.” *Journal of Molecular Biology*, **147**(1), 195–197. ISSN 0022-2836.
- Thompson JD, Plewniak F, Poch O (1999). “BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs.” *Bioinformatics*, **15**(1), 87–88. ISSN 1460-2059.
- Turnbaugh PJ, Ley RE, Hamady M, Fraser-Liggett CM, Knight R, Gordon JI (2007). “The Human Microbiome Project.” *Nature*, **449**, 804–810.
- Ukkonen E (1992). “Approximate String Matching with q-grams and Maximal Matches.” *Theoretical Computer Science*, **92**(1), 191–211.
- Vinga S, Almeida J (2003). “Alignment-free sequence comparison—a review.” *Bioinformatics*, **19**(4), 513–523. ISSN 1367-4803. doi:[10.1093/bioinformatics/btg005](https://doi.org/10.1093/bioinformatics/btg005).

Affiliation:

Anurag Nagar
Computer Science and Engineering
Lyle School of Engineering
Southern Methodist University
P.O. Box 750122
Dallas, TX 75275-0122
E-mail: anagar@smu.edu

Michael Hahsler
Computer Science and Engineering
Lyle School of Engineering
Southern Methodist University
P.O. Box 750122
Dallas, TX 75275-0122
E-mail: mhahsler@lyle.smu.edu
URL: <http://lyle.smu.edu/~mhahsler>