Contents

11.3 Naïve Bayes

Chapter 1

  - Standalone Blast from NCBI
  - Clustalw alignment program.

- A standard sequence class that deals with sequences, ids on sequences, and sequence features.

4.

Chapter 2

followed by what you would type in:

2.4 Parsing sequence file formats

A large part of much bioinformatics work involves dealing with the many types of file formats designed to hold biological data. These files are loaded with interesting biological data, and a special challenge is parsing these files into a format so that you can manipulate them with some kind ofsom-thatg95fsom--that9(Ho-280w1(g)-367()1(g)-r)

2.4.2 Simple GenBank parsing example

Now let's load the GenBank file ls_orchid.gbk instead - notice that the code to do this is almost identical to the snippet used above for the FASTA file - the only difference is we change the filename and the format string:

```
from Bio import SeqIO
handle = open("ls_orchid.gbk")
for seq_record in SeqIO.parse(handle, "genbank") :
    print seq_record.id
    print repr(seq_record.seq)
    print len(seq_record)
handle.close()
```

This should give:

```
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguousDNA())
740
...
Z78439.1
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', IUPACAmbiguousDNA())
592
```

This time `Bio.SeqIO`

The code in these modules basically makes it easy to write python code that interact with the CGI scripts on these pages, so that you can get results in an easy to deal with format. In some cases, the results can be tightly integrated with the Biopython parsers to make it even easier to extract information.

2.6 What to do next

```
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT', Alphabet())
>>> my_seq.alphabet
Alphabet()
```

Note that using the `Bio.SeqUtils.GC()`

You can also use the Seq object directly with a %s placeholder when using the python string formatting or interpolation operator (%):

3.6 Nucleotide sequences and (reverse) complements

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> template_dna = coding_dna.reverse_complement()
>>> template_dna
Seq('CTATCGGGCACCCTTTCAGCGGCCCATTACAATGGCCAT', IUPACUnambiguousDNA())
```

You can also translate directly from the coding strand DNA sequence:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> coding_dna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_name["Standard"]
>>> mito_table = CodonTable.unambiguous_dna_by_name["Vertebrate Mitochondrial"]
```

Alternatively, these tables are labeled with ID numbers 1 and 2, respectively:

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_id[1]
>>> mito_table = CodonTable.unambiguous_dna_by_id[2]
```

You compare the actual tables visually by printing them:

```
>>> print standard_table
Table 1 Standard, SGC0

  |  T      |  C      |  A      |  G      |
--+---------+---------+---------+---------+--
T | TTT F   | TCT S   | TAT Y   | TGT C   | T
T | TTC F   | TCC S   | TAC Y   | TGC C   | C
T | TTA L   | TCA S   | TAA Stop| TGA Stop| A
T | TTG L(s)| TCG S   | TAG Stop| TGG W   | G
--+---------+---------+---------+---------+--
C | CTT L   | CCT P   | CAT H   | CGT R   | T
```

```
print second_record.id
print second_record.description

handle.close()
```

Note that if you try and use `.next()` and there are no more results, you'll either get back the special Python object `None` or a `StopIteration` exception.

You can of course still use a for loop with a list of SeqRecord objects. Using a list is much more flexible than an iterator (for example, you can determine the number of records from the length of the list), but does need more memory because it will hold all the records in memory at once.

4.1.4 Extracting data

The SeqRecord object and its annotation structures are described more fully in Section 13.1. For now, as an example of how annotations are stored, we'll look at the output from parsing the first record in the GenBank file ls_ d

```
print first_r
```

You can check by hand, but for every record the species name is in the description line as the second word. This means if we break up each record's `.description`

This should give:

```
Z78533.1 JUEoWn6DPhgZ9nAyowsgtoD9TTo
Z78532.1 MN/s0q9zDoCVEEc+k/IFwCNF2pY
...
Z78439.1 H+JfaShya/4yyAj7IbMqgNkxdxQ
```

Now, recall the `Bio.SeqIO.to_dict()` function's `key_function`

```
                   +"TGEGLEWGVLFGFGPGLTVETVVLHSVAT", generic_protein),
                 id="gi|13925890|gb|AAK49457.1|",
                 description="chalcone synthase [Nicotiana tabacum]")

my_records = [rec1, rec2, rec3]
```

```
records = (make_rc_record(rec) for rec in SeqIO.parse(in_handle, "fasta") if len(rec)<700)
```

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_protein

record = SeqRecord(Seq("MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD" \
                       +"GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPGLISK" \
                       +"NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRATREVLSEYGNM" \
                       +"SSAC", generic_protein),
                   id="gi|14150838|gb|AAK54648.1|AF376133_1",
                   description="chalcone synthase [Cucumis sativus]")

print record.format("fasta")
```

which should give:

```
>gi|14150838|gb|AAK54648.1|AF376133_1 chalcone synthase [Cucumis sativus]
MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD
GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPGLISK
NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRATREVLSEYGNM
SSAC
```

This `format` method takes a single mandatory argument, a lower case string which is supported by `Bio.SeqIO` as an output format. However, some of the file formats `Bio.SeqIO` can write to

Chapter 5

Sequence Alignment Input/Output

In this chapter we'll discuss the `Bio.AlignIO` module, which is very similar to the `Bio.SeqIO` module from the previous chapter, but deals with

```
print "Alignment length %i" % alignment.get_alignment_length()
for record in alignment :
    print "%s - %s" % (record.seq, record.id)
```

Epsilon CCCAAC

```
from Bio import AlignIO
alignments = list(AlignIO.parse(open("resampled.phy"), "phylip"))
last_align = alignments[-1]
first_align = alignments[0]
```

5.1.3 Ambiguous Alignments

Many alignment file formats can explicitly store more than one alignment, and the division between each alignment is clear. However, when a general sequence file format has been used there is no such block structure. The most common such situation is when alignments have been saved in the FASTA file format. For example consider the following:

```
>Alpha
ACTACGACTAGCTCAG--G
>Beta
ACTACCGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Beta
ACTACCGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
```

This could be a single alignment containing six sequences (with repeated identifiers). Or, judging from the

Chapter 6

BLAST

Hey, everybody loves BLAST right? I mean, geez, how can get it get any easier to do comparisons between one of your sequences and every other sequence in the known world? But, of course, this section isn't about how cool BLAST is, since we already know that. It is about the problem with BLAST – it can be really difficult to deal with the volume of data generated by large runs, and to automate BLAST runs in general.

Fortunately, the Biopython folks know this only too well, so they've developed lots of tools for dealing with BLAST and making things much easier. This section details how to use these tools and do useful things with them.

Dealing with BLAST can be split up into two steps, both of which can be done from within Biopython.

```
>>> my_blast_db = "/home/mdehoon/Data/Genomes/Databases/bsubtilis"
# I used formatdb to create a BLAST database named bsubtilis
# (for Bacillus subtilis) consisting of the following three files:
# /home/mdehoon/Data/Genomes/Databases/bsubtilis.nhr
# /home/mdehoon/Data/Genomes/Databases/bsubtilis.nin
# /home/mdehoon/Data/Genomes/Databases/bsubtilis.nsq
```

6.3 Saving BLAST output

Before parsing the results, it is often useful to save them into a file so that you can use them later without having to go back and re-blasting everything. I find this especially useful when debugging my code that extracts info from the BLAST files, but it could also be useful just for making backups of things you've done.

```
>>> for blast_record in blast_records:
...     # Do something with blast_record
```

Basically, you can do anything you want to with the info in the BLAST report once you have parsed it.

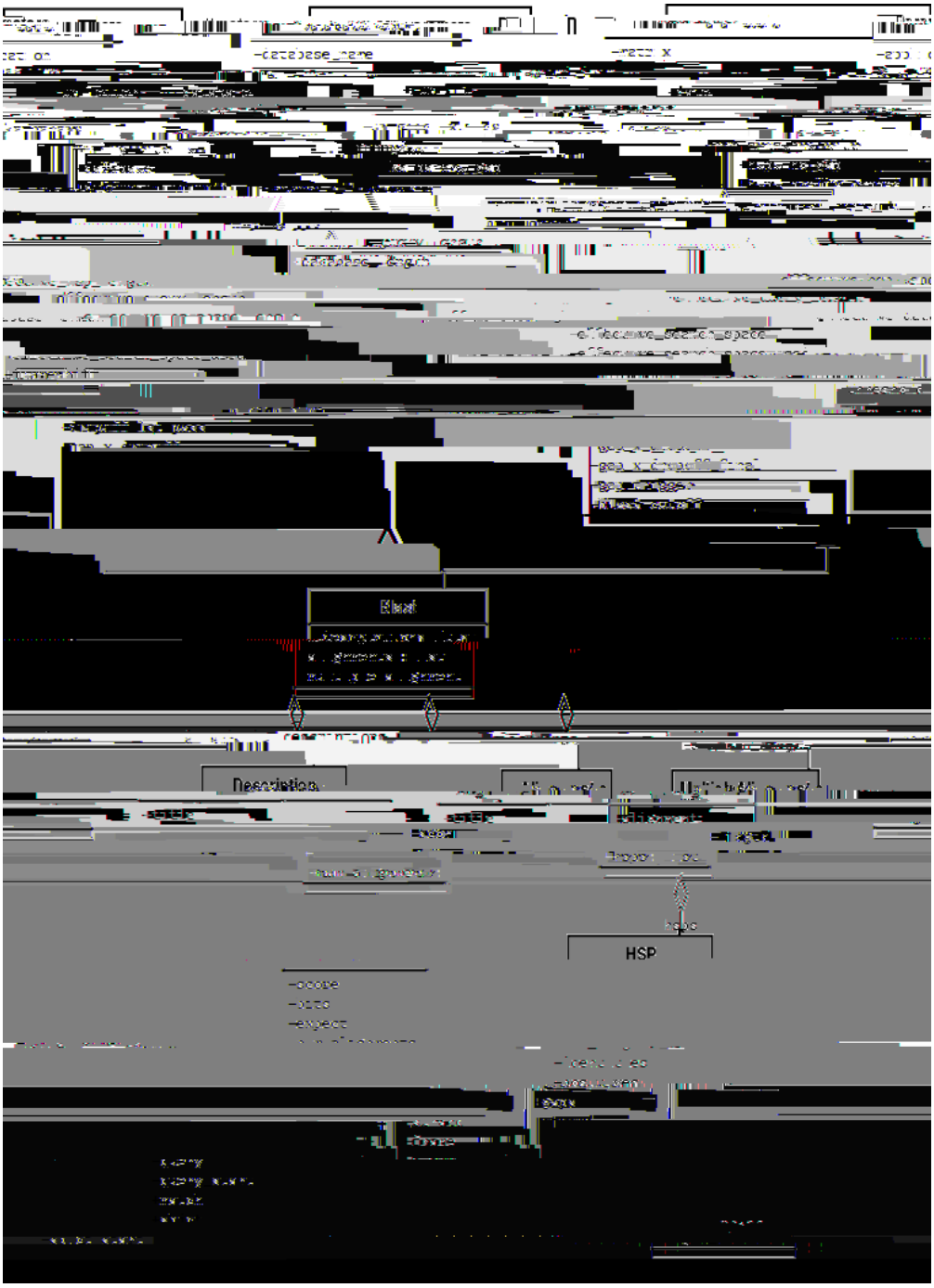

Figure 6.1: Class diagram fol6346346346389.1047-6-55-45346as55-4t047-6Re55-4cord047-6c334(d)1(iarepr346e55-4se55-4

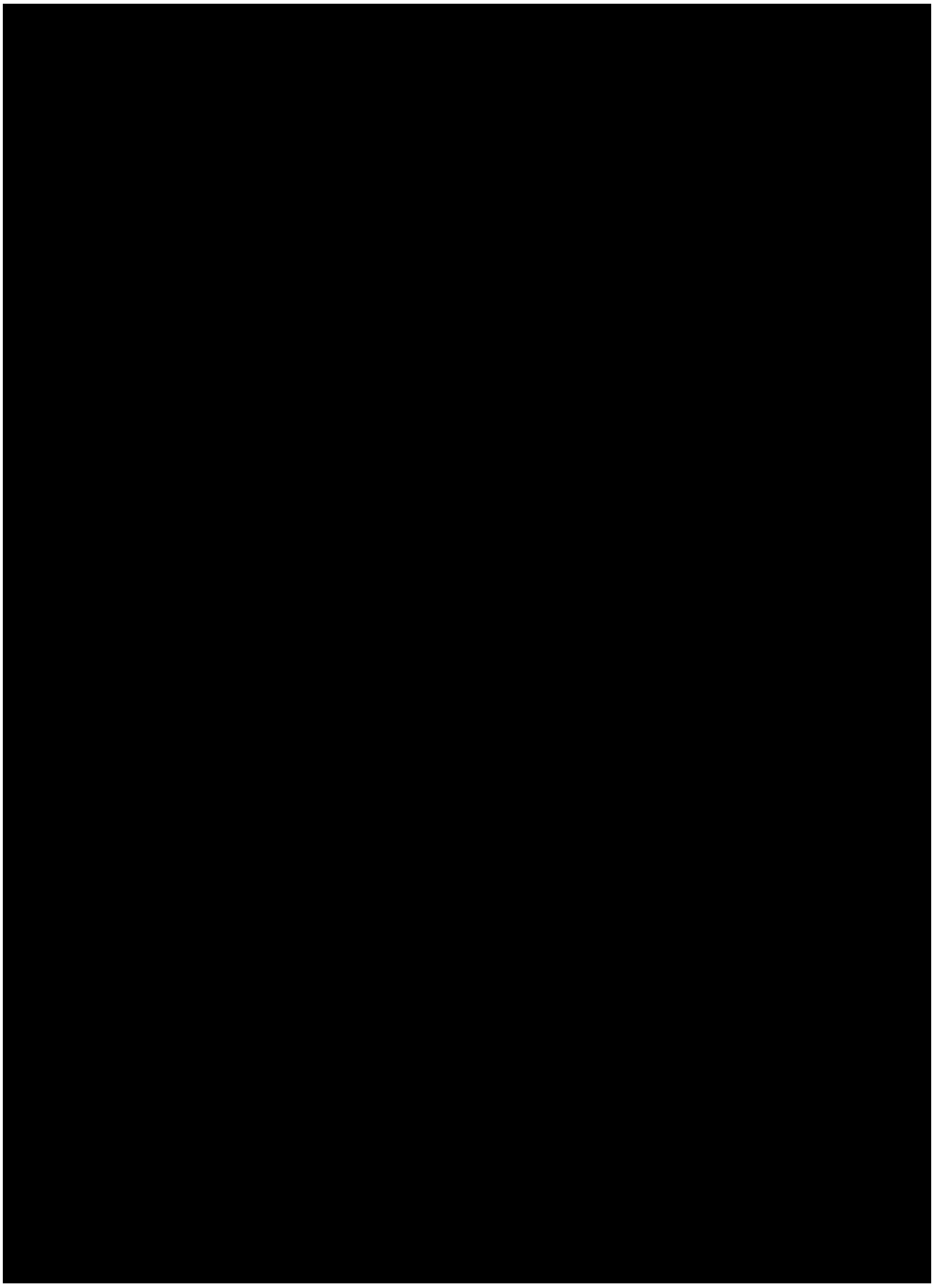

Figure 6.2: Class diagram for the PSIBlast Record class.

```
...             print hsp.query[0:75] + '...'
...             print hsp.match[0:75] + '...'
...             print hsp.sbjct[0:75] + '...'
```

If you also read the section 6.4 on parsing BLAST XML output, you'll notice that the above code is identical to what is found in that section. Once you parse something into a record class you can deal with it independent of the format of the original BLAST info you were parsing. Pretty snazzy!

Sure, parsing one record is great, but I've got a BLAST file with tons of records – how can I parse them all? Well, fear not, the answer lies in the very next section.

6.6.3 Finding a bad record somewhere in a huge file

– `item[1]` – The id of the input record that caused the error. This is really useful if you want to record all of the records that are causing problems.

As mentioned, with each error generated, the BlastErrorParser will write the o□ending record to the specified `error_handle`

```
        <DbName>gap</DbName>
        <DbName>domains</DbName>
        <DbName>gene</DbName>
        <DbName>genomeprj</DbName>
        <DbName>gensat</DbName>
        <DbName>geo</DbName>
        <DbName>gds</DbName>
        <DbName>homologene</DbName>
        <DbName>journals</DbName>
        <DbName>mesh</DbName>
        <DbName>ncbisearch</DbName>
        <DbName>nlmcatalog</DbName>
        <DbName>omia</DbName>
        <DbName>omim</DbName>
        <DbName>pmc</DbName>
        <DbName>popset</DbName>
        <DbName>probe</DbName>
        <DbName>proteinclusters</DbName>
        <DbName>pcassay</DbName>
        <DbName>pccompound</DbName>
        <DbName>pcsubstance</DbName>
        <DbName>snp</DbName>
        <DbName>taxonomy</DbName>
        <DbName>toolkit</DbName>
        <DbName>unigene</DbName>
        <DbName>unists</DbName>
</DbList>
</eInfoResult>
```

Since this is a fairly simple XML file, we could extract the information it contains simply by string searching. Using `Bio.Entrez`'s parser instead, we can directly parse this XML file into a Python object:

```
>>> record["DbInfo"]["Description"]
'PubMed bibliographic record'
>>> record["DbInfo"]["Count"]
'17989604'
>>> record["DbInfo"]["LastUpdate"]
'2008/05/24 06:45'
```

Try `record["DbInfo"].keys()`

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"     # Always tell NCBI who you are
```

```
>>> from Bio import Entrez
```

```
>>> record["PMID"]
'12230038'
>>> record["AB"]
'Bioinformatics research is often difficult to do with commercial software.
The Open Source BioPerl, BioPython and Biojava projects provide toolkits with
multiple functionality that make it easier to create customised pipelines or
analysis. This review briefly compares the quirks of the underlying languages
and the functionality, documentation, utility and relative advantages of the
Bio counterparts, particularly from the point of view of the beginning
biologist programmer.'
```

```
>>> print records[0]["GBSeq_definition"]
Cypripedium calceolus voucher Davis 03-03 A maturase (matR) gene, partial cds;
mitochondrial

>>> print records[0]["GBSeq_organism"]
Cypripedium calceolus
```

You could use this to quickly set up searches – but for heavy usage, see Section 7.12.

7.11.3 Searching, downloading, and parsing GenBank records

The GenBank record format is a very popular method of holding information about sequences, sequence features, and other associated sequence information. The format is a good way to get information from the NCBI databases at

```
>>> record["IdList"][0]
'158330'
```

Now, we use efetch to download this entry in the Taxonomy database, and then parse it:

```
>>> handle = Entrez.efetch(db="Taxonomy", id="158330", retmode="xml")
>>> records = Entrez.read(handle)
```

Again, this record stores lots of information:

```
>>> records[0].keys()
[u'Lineage', u'Division', u'ParentTaxId', u'PubDate', u'LineageEx',
 u'CreateDate', u'TaxId', u'Rank', u'GeneticCode', u'ScientificName',
 u'MitoGeneticCode', u'UpdateDate']
```

We can get the lineage directly from this record:

However, you also get given two additional pieces of information, the `WebEnv` session cookie, and the `QueryKey`:

```
>>> webenv = search_results["WebEnv"]
>>> query_key = search_results["QueryKey"]
```

Having stored these values in variables

```
    out_handle.write(data)
out_handle.close()
```

Chapter 8

Swiss-Prot, Prosite, Prodoc, and ExPASy

8.1 Bio.SwissProt: Parsing Swiss-Prot files

Swiss-Prot (http://www.expasy.org/sprot

Or, using a for loop over the record iterator:

```
>>> from Bio import SwissProt
>>> descriptions = []
>>> handle = open("uniprot_sprot.dat")
>>> for record in SwissProt.parse(handle) :
...     descriptions.append(record.description)
...
>>> len(descriptions)
290484
```

Because this is such a large input file, either way takes about seven minutes on my new desktop computer (using the uncompressed `uniprot_sprot.dat`unipre,nasng

The entries in this file can be parsed by the `parse` function in the `Bio.SwissProt.KeyWList` module. Each entry is then stored as a

```
>>> record.accession
'PS00005'
>>> record.name
'PKC_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00005'
```

)60(ifrec60(m)-332(d)-1(c61(f)you're interested in how many Prosite records there are (434TE)]TJ0-11.95(ute)nd)r).S0000528914.34629.962

sprot_search_ful To search for a Swiss-Prot record

sprot_search_de To search for a Swiss-Prot record

To access this web server from a Python script, we use the `Bio.ExPASy` module.

```
>>> from Bio import ExPASy
```

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prodoc_entry('PDOC00001')
>>> html = handle.read()
>>> output = open("myprodocrecord.html", "w")
>>> output.write(html)
>>> output.close()
```

For these functions, an invalid accession number returns an error message in HTML format.

Chapter 9

Going 3D: The PDB module

Biopython also allows you to explore the extensive realm of macromolecular structure. Biopython comes with a PDBParser class that produces a Structure object. The Structure object can be used to access the atomic data in the file in a convenient manner.

9.1 Structure representation

A macromolecular structure is represented using a structure, model chain, residue, atom (or SMCRA)

```
full_id=residue.get_full_id()

print full_id
```

residue name GLC) with sequence identifier 10 would have residue id (''H_GLC'', 10, '' '')

9.2 Disorder

9.2.1 General approach

would have id “SER” in the DisorderedResidue object, while residue Cys 60 would have id “CYS”. The user can select the active Residue object in a DisorderedResidue object via this id.

9.5.1.1 Duplicate residues

One structure contains two amino acid residues in one chain with the same sequence identifier (resseq 3) and icode. Upon inspection it was found that this chain contains the residues Thr A3, ..., Gly A202, Leu A3, Glu A204. Clearly, Leu A3 should be Leu A203. A couple of similar situations exist for structure 1FFK (which e.g. contains Gly B64, Met B65, Glu B65, Thr B67, i.e. residue Glu B65 should be Glu B66).

9.5.1.2 Duplicate atoms

Chapter 10

```
        ('Other1', [(1, 1),  (4, 3), (200, 200)],
    ]
]
```

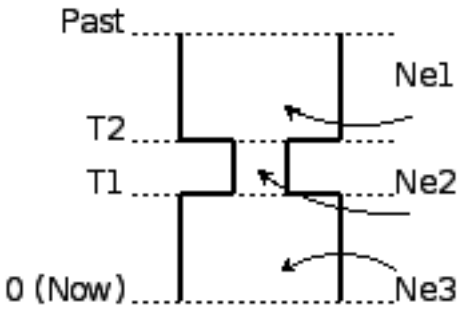

Figure 10.1: A bottleneck

Chapter 11

Supervised learning methods

11.1 The Logistic Regression Model

```
          [15, -180.41],
          [-26, -181.73],
          [58, -259.87],
          [126, -414.53],
          [191, -249.57],
          [113, -265.28],
          [145, -312.99],
          [154, -213.83],
          [147, -380.85],
          [93, -291.13]]
>>> ys = [1,
          1,
          1,
          1,
          1,
          1,
          1,
          1,
          1,
          1,
          0,
          0,
          0,
          0,
          0,
          0,
          0]
>>> model = LogisticRegression.train(xs, ys)
```

Here, `xs` and `ys` are the training data: `xs` contains the predictor variables for each gene pair, and `ys`

11.2.2 Initializing a *k*

By default, all neighbors are given an equal weight.

Chapter 12

Cookbook – Cool things to do with it

12.1 Sequence parsing plus simple plots

This section shows some more examples of sequence parsing, using the `Bio.SeqIO` module described in Chapter 4, plus the py8-111.errary1.e1('sthe)]TJ/F349.962616-42572720TpylabqIO

```
pylab.ylabel("Count")
pylab.show()
```

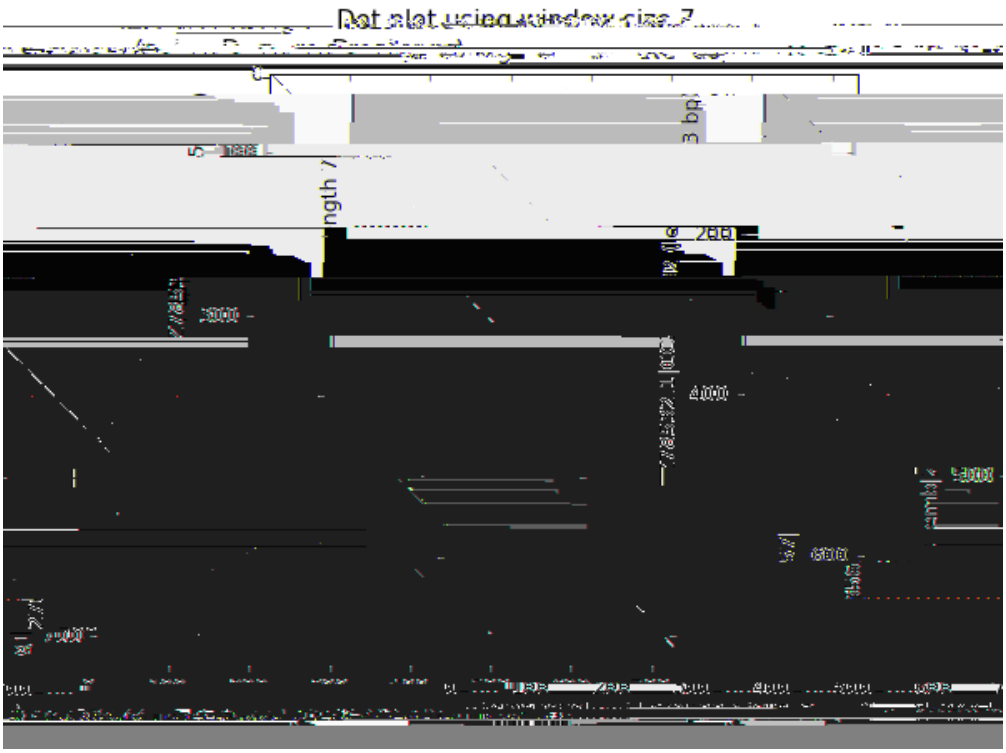

Figure 12.3: Nucleotide dot plot of two orchid sequence lengths (using pylab's imshow function).

Figure 12.4: Nucleotide dot plot of two orchid sequence lengths (using pylab's scatter function).

```
from Bio.Clustalw import MultipleAlignCL

cline = MultipleAlignCL(os.path.join(os.curdir, "opuntia.fasta"))
cline.set_output("test.aln")
```

First we import the `MultipleAlignCL` object, which models running a multiple alignment from clustalw. We then initialize the command line, with a single argument of the fasta file that we are going to be using

This makes it easy to write youvalignitbac-333k334(itin-333--333(toa333(tofile)28(esw)-31(h28(ouvall28(ouvofs)-1(y)h3(v

12.2.4 Position Specific Score Matrices

Position specific score matrices (PSSMs) summarize the alignment information in a different way than a consensus, and may be useful for different tasks. Basically, a PSSM is a count matrix. For each column in the alignment, the number of each alphabet letters is counted and totaled. The totals are displayed relative to some representative sequence along the left axis. This sequence may be the consesus sequence, but can also be any sequence in the alignment. For instance for the alignment,

```
GTATC
AT--C
CTGTC
```

the PSSM is:

```
    G A T C
  G 1 1 0 1
  T 0 0 3 0
  A 1 1 0 0
  T 0 0 2 0
  C 0 0 0 3
```

Let's assume we've got an alignment object called `c_align`. To get a PSSM with the consensus sequence along the side we first get a summary object and calculate the consensus sequence:

```
summary_align = AlignInfo.SummaryInfo(c_align)
consensus = summary_align.dumb_consensus()
```

```
C  0.0 7.0 0.0 0.0
A  7.0 0.0 0.0 0.0
T  0.0 0.0 0.0 7.0
T  1.0 0.0 0.0 6.0
...
```

You can access any element of the PSSM by subscripting like `your_pssm[sequence_number][residue_count_name]`. For instance, to get the counts for the 'A' residue in the second element of the above PSSM you would do:

```
>>> print my_pssm[1]["A"]
7.0
```

```
expect_freq = {
```

- exp_freq_table

- Function
-

```
>>> from Bio.Seq import Seq
>>> simple_seq = Seq("GATC")
>>> from Bio.SeqRecord import SeqRecord
>>> simple_seq_r = SeqRecord(simple_seq)
```

Additionally, you can also pass the id, name and description to the initialization function, but if not they will be set as strings indicating they are unknown, and can be modified subsequently:

ref_db – This works along with `ref` to provide a cross sequence reference. If there is a reference, `ref_db` will be set as None if the reference is in the same database, and will be set to the name of the database otherwise.

strand

I just mention this because sometimes I get confused between the two.

(e)

Chapter 14

Where to go from here – contributing

Macintosh – We would love to find someone who wants to maintain a Macintosh distribution, and make it available in a Macintosh friendly format like bin-hex. This would basically include finding a way to compile everything on the Mac, making sure all of the code written by us UNIX-based developers works well on the Mac, and providing any Mac-friendly hints for us.