Contents

4.3.1 SeqFeature objects

Chapter 1

Introduction

1.1 What is Biopython?

The Biopython Project is an international association of developers of freely available Python (http://www.python.org) tools for computational molecular biology. Python is an object oriented, interpreted, exible language that is becoming increasingly popular for scienti c computing. Python is easy to learn, has a very clear syntax andtosynte(b)-28()27txtendendtowithdtombulesdtoindtoC,dtooar

1.4 Frequently Asked Questions (FAQ)

1. *How do I cite Biopython in a scientific publication?* Bi.PDBhyo Bi.PDBhyo [997]TJ 0 g 0 G [(,)-337Cec
   rn
   Please cite our application note [1, Cock *et al.*, 2009] as the main Biopython reference. In addition, please cite any publications from the following list i[tions, articular[tions reference fr[tionsspo
   rn Bri [91,[41] [42]TJ 0 g 0 G [(,)-34(DhPr
   rn the f ciarjectrunmean [31,th, 200];.
   wthain Biopytn morenfrmcationen our

5. *Do you have a change-log listing what's new in each release?*
   See the le `NEWS.rst` included with the source code (originally called just `NEWS`), or read the latest NEWS le on GitHub.

6.

9. *What is wrong with my sequence comparisons?*

21.

Chapter 2

Quick Start { What can you do with Biopython?

followed by what you would type in:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT')
>>> print(my_seq)
AGTACACTGGT
>>> my_seq.alphabet
Alphabet()
```

What we have here is a sequence object with a *generic* alphabet - re ecting the fact we have *not* spec-

2.4.2 Simple GenBank parsing example

Now let's load the GenBank le `ls_orchid.gbk`

2.6 What to do next

Now that you've made it this far, you hopefully have a good understanding of the basics of Biopython and are ready to start using it for doing useful work. The best thing to do now is nish reading this tutorial, and then if you want start snooping around in the source code, and looking at the automatically generated documentation.

Once you get a picture of what you want ading(to)28(an)66(w8(ou)-)27 [(ibraries)66(w8)-337(w8)-345(Biop)28(yt(ou)ill

Chapter 3

Sequence objects

Biological sequences are arguably the central object in Bioinformatics, and in this chapter we'll introduce the Biopython mechanism for dealing with sequences, the `Seq` object. Chapter 4 will introduce the related `SeqRecord` object, which combines the sequence information with any annotation, used again in Chapter 5 for Sequence Input/Output.

Sequences are essentially strings of letters like `AGTACACTGGT`, which seems very natural since this is the most common way that sequences are seen in biological le formats.

There are two important di erences between `Seq` objects and standard Python strings. First of all, they have di erent methods. Although the `Seq` object supports many of the same methods as a plain string, its `translate()` method di ers by doing biological translation, and there are also additional biologically relevant methods like `reverse_complement()`. Secondly, the `Seq` object has an important attribute, `alphabet`, which is an object describing what the individual characters making up the sequence string \mean", and how they should be interpreted. For example, is `AGTACACTGGT` a DNA sequence, orharacters mar8Astri306()-1(,)-511306(3(seq.242

```
>>> my_seq = Seq("AGTACACTGGT")
```

The `Seq` object has a `.count()` method, just like a string. Note that this means that like a Python string, this gives a *non-overlapping* count:

```
>>> from Bio.Seq import Seq
```

In all of these operations, the alphabet property is maintained. This is very useful in case you accidentally

In the bacterial genetic code `GTG` is a valid start codon, and while it does *normally*

```
--+---------+---------+---------+---------+--
G | GTT V   | GCT A   | GAT D   | GGT G   | T
G | GTC V   | GCC A   | GAC D   | GGC G   | C
G | GTA V   | GCA A   | GAA E   | GGA G   | A
G | GTG V   | GCG A   | GAG E   | GGG G   | G
--+---------+---------+---------+---------+--
```

For example, you might argue that the two DNA `Seq` objects `Seq("ACGT", IUPAC.unambiguous_dna)` and `Seq("ACGT", IUPAC.ambiguous_dna)`

3.13 UnknownSeq objects

The `UnknownSeq` object is a subclass of the basic

Chapter 4

Sequence annotation objects

Chapter 3 introduced the sequence classes. Immediately \above" the `Seq` class is the Sequence Record or `SeqRecord` class, de ned in the `Bio.SeqRecord`

.annotations { A dictionary of additional information about the sequence. The keys are the name of

As you can see above, the rst word of the FASTA record's title line (after removing the greater than symbol) is used for both the `id` and `name` attributes. The whole title line (after removing the greater than symbol) is used for the record description. This is deliberate, partly for backwards compatibility reasons,

```
>>> record.seq


 from theq-362(LOCUSq-361(line,q-369(while)-362(theq)]T/FF289.9626Tf174.41160Td[idq)]T/F289.9626Tf14.062
>>> recorddescriptionq




>>> recordletter_annotationsq




>>> recorddbxrefsq
```

```
>>> my_location.start
AfterPosition(5)
>>> print(my_location.start)
>>> for feature in record.features:
...     if my_snp in feature:
...         print("%s %s" % (feature.type, feature.qualifiers.get("db_xref")))
...
source ['taxon:229193']
gene ['GeneID:2767712']
CDS ['GI:45478716', 'GeneID:2767712']
```

Note that gene and CDS features from GenBank or EMBL les de ned with joins are the union of the

4.4 Comparison

The `SeqRecord` objects can be very complex, but here's a simple example:

4.6 The format method

The `format()`

For this example we're going to focus in on the pim gene, YP_pPCP05. If you have a look at the GenBank
le directly you'll nd isis loo-30str0(gone,)]TJ/F28 9.9

```
>>> print(sub_record.features[1])
type: CDS
location: [42:480](+)
qualifiers:
    Key: codon_start, Value: ['1']
    Key: db_xref, Value: ['GI:45478716', 'GeneID:2767712']
    Key: gene, Value: ['pim']
    Key: locus_tag, Value: ['YP_pPCP05']
    Key: note, Value: ['similar to many previously sequenced pesticin immunity ...']
```

For the sequence, thi [(F)8Ieshe theSeqheersthethemthe135()1(ohe)8(d.)-412(An)28(yhe)-23featurehi [(F)artheF

Chapter 5

Sequence Input/Output

In this chapter we'll discuss in more detail the `Bio.SeqIO` module, which was brie y introduced in Chapter 2 and also used in Chapter 4

5.1.1 Reading Sequence Files

In general `Bio.SeqIO.parse()` is used to read in sequence les as `SeqRecord` objects, and is typically used with a for loop like this:

```
from Bio import SeqIO
for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
```

```
second_record = next(record_iterator)
print(second_record.id)
print(second_record.description)
```

You can of course still use a for loop with a list ofSeqRecordobjects. Using a list is much more exiblethan an iterator (for example, you ca or:

```
>>> print(first_record.annotations["organism"])
Cypripedium irapeanum
```

In general, `organism' is used for the scienti c name (in Latin, e.g. *Arabidopsis thaliana*), while `source' will often be the common name (e.g. thale cress). In this example, as is often the case, the two elds are identical.

Now let's go through all the records, building up a list of the species each orchid sequence is from:

```
from Bio import SeqIO
all_species = []
for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):
    all_species.append(seq_record.annotations["organism"])
print(all_species)
```

Another way of writing this code is to use a list comprehension:

```
from Bio import SeqIO
all_species = [seq_record.annotations["organism"] for seq_record in \
               SeqIO.parse("ls_orchid.gbk", "genbank")]
print(all_species)
```

In either case, the result is:

```
['Cypripedium irapeanum', 'Cypripedium californicum', ..., 'Paphiopedilum barbatum']
```

Great. That was pretty easy because GenBank les are annotated in a standardised way.

```
...     handle = bz2.open("ls_orchid.gbk.bz2", "rt")  # Python 3
... else:
...     handle = bz2.BZ2File("ls_orchid.gbk.bz2", "r")  # Python 2
...
>>> with handle:
```

The expected output of this example is:

5.4.1.2 Indexing a dictionary using the SEGUID checksum

To give another example of working with dictionaries of `SeqRecord` objects, we'll use the SEGUID checksum function. This is a relatively recent checksum, and collisions should be very rare (i.e. two di erent sequences with the same checksum), an improvement on the CRC64 checksum.

Once again, working with the orchids GenBank le:

```
from Bio import SeqIO
from Bio.SeqUtils.CheckSum import seguid
for record in SeqIO.parse("ls_orchid.gbk", "genbank"):
    print(record.id, seguid(record.seq))
```

This should give:

```
Z78533.1 JUEoWn6DPhgZ9nAyowsgtoD9TTo
Z78532.1 MN/s0q9zDoCVEEc+k/IFwCNF2pY
```

```
>>> seq_record = orchid_dict["Z78475.1"]
>>> print(seq_record.description)
P.supardii 5.8S rRNA gene and ITS1 and ITS2 DNA
>>> seq_record.seq
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGATCACAT...GGT', IUPACAmbiguousDNA())
>>> orchid_dict.close()
```

Note that `Bio.SeqIO.index()` won't take a handle, but only a lename. There are good reasons for this, to preserve the text exactly (e.g. GenBank or EMBL output from Biopython does not yet preserve every last bit of annotation).

Let's suppose you have download the whole of UniProt in the plain text SwissPort le format from their FTP site (ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete/uniprot_sprot.dat.gz

Now, in Python, index these GenBank les as follows:

```
>>> import glob
>>> from Bio import SeqIO
>>> files = glob.glob("gbvrl*.seq")
>>> print("%i files to index" % len(files))
4
>>> gb_vrl = SeqIO.index_db("gbvrl.idx", files, "genbank")
>>> print("%i sequences indexed" % len(gb_vrl))
```

You can use the compressed le in exactly the same way:

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.gbk.bgz", "genbank")
>>> len(orchid_dict)
94
```

5.5 Writing Sequence Files

We've talked about using `Bio.SeqIO.parse()` for sequence input (reading les), and now we'll look at `Bio.SeqIO.write()` which is for sequence output (writing les). This is a function taking three arguments: some `SeqRecord` objects, a handle or lename to write to, and a sequence format.

Here is an example, where we start by creating a few `SeqRecord` objects the hard way (by hand, rather

```
from Bio import SeqIO
count = SeqIO.convert("ls_orchid.gbk", "genbank", "my_example.fasta", "fasta")
print("Converted %i records" % count)
```

The `Bio.SeqIO.convert()` function will take handles *or* lenames. Watch out though { if the output le already exists, it will overwrite it! To nd out more, see the built in help:

```
>>> from Bio import SeqIO
>>> help(SeqIO.convert)
...
```

```
>>> records = (rec.reverse_complement(id="rc_"+rec.id, description = "reverse complement") \
```

6.1.1 Single Alignments

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print("Alignment length %i" % alignment.get_alignment_length())
Alignment length 52
```

Note the website should have an option about showing gaps as periods (dots) or dashes, we've shown dashes

```
Alpha     AAAAAC
Beta      AAACCC
Gamma     AACAAC
Delta     CCCCCA
Epsilon   CCCAAC
...
    5     6
Alpha     AAAACC
Beta      ACCCCC
Gamma     AAAACC
Delta     CCCCAA
Epsilon   CAAACC
```

If you wanted to read this in using `Bio.AlignIO` you could use:

```
from Bio import AlignIO
alignments = AlignIO.parse("resampled.phy", "phylip")
for alignment in alignments:
    print(alignment)
    print("")
```

This would give the following output, again abbreviated for display:

```
SingleLetterAlphabet() alignment with 5 rows anC.ntprint("")
```

As with the function `Bio.SeqIO.parse()`, using `Bio.AlignIO.parse()` returns an iterator. If you want to keep all the alignments in memory at once, which will allow you to access them in any order, then turn the iterator into a list:

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

```
>XXX
ACTACCGCTAGCTCAGAAG
>Alpha
ACTACGACTAGCTCAGG
>YYY
ACTACGGCAAGCACAGG
>Alpha
--ACTACGAC--TAGCTCAGG
>ZZZ
GGACTACGACAATAGCTCAGG
```

In this third example, because of the di ering lengths, this cannot be treated as a single alignment containing

6.2 Writing Alignments

We've talked about using `Bio.AlignIO.read()` and `Bio.AlignIO.parse()` for alignment input (reading les), and now we'll look at `Bio.AlignIO.write()` which is for alignment output (writing les). This is a function taking three arguments: some `MultipleSeqAlignment` objects (or for backwards compatibility the obsolete `Alignment` objects), a handle or lename to write to, and a sequence format.

Here is an example, where we start by creating a few `MultipleSeqAlignment` objects the hard way (by hand, rather than by loading them from a le). Note we create some `SeqRecord` objects to construct the alignment from.

```
from Bio.Alphabet import generic_dna
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Align import MultipleSeqAlignment

align1 = MultipleSeqAlignment([
             SeqRecord(Seq("ACTGCTAGCTAG", generic_dna), id="Alpha"),
             SeqRecord(Seq("ACT-CTAGCTAG", generic_dna), id="Beta"),
             SeqRecord(Seq("ACTGCTAGDTAG", generic_dna), id="Gamma"),
         ])

align2 = MultipleSeqAlignment([
             SeqRecord(Seq("GTCAGC-AG", generic_dna), id="Delta"),
             SeqRecord(Seq("GACAGCTAG", generic_dna), id="Epsilon"),
             SeqRecord(Seq("GTCAGCTAG", generic_dna), id="Zeta"),
         ])

align3 = MultipleSeqAlignment([
```

Its more common to want to load an existing alignment, and save that, perhaps after some simple manipulation like removing certain rows or columns.

Suppose yo(to)-484(w)28(an)2T3neows-4-484ho27(ts-4-484m(an)2Ty-4-484lignmen)272T3so

```
Q9T0Q8_BPIKE/1-52                    RA
COATB_BPI22/32-83                    KA
COATB_BPM13/24-72                    KA
COATB_BPZJ2/1-49                     KA
Q9T0Q9_BPFD/1-49                     KA
COATB_BPIF1/22-73                    RA
```

Alternatively, you could make a PHYLIP format le which we'll name \PF05371_seed.phy":

```
from Bio import AlignIO
AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.phy", "phylip")
```

This time the output looks like this:

```
 7 52
COATB_BPIK AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFKKFSS
Q9T0Q8_BPI AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFVS
COATB_BPI2 DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
COATB_BPM1 AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPZJ AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFAS
Q9T0Q9_BPF AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPIF FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS

           KA
           RA
           KA
           KA
           KA
           KA
           RA
```

One of the big handicaps of the original PHYLIP alignment le format is that the sequence identi ers are strictly truncated at ten characters. In this example, as you can see the resulting names are still unique - but they are not very readable. As a result, a more relaxed variant of the original PHYLIP format is now quite widely used:

```
from Bio import AlignIO
AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.phy", "phylip-relaxed")
```

This time the output looks like this, using a longer indentation to allow all the identifers to be given in full::

```
 7 52
COATB_BPIKE/30-81  AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFKKFSS
Q9T0Q8_BPIKE/1-52  AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFVS
COATB_BPI22/32-83  DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
COATB_BPM13/24-72  AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPZJ2/1-49   AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFAS
Q9T0Q9_BPFD/1-49   AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPIF1/22-73  FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS

                   KA
                   RA
                   KA
                   KA
                   KA
                   KA
                   RA
```

If you have to work with the original strict PHYLIP format, then you may need to compress the identifers somehow { or assign your own names or numbering system. This following bit of code manipulates the record identi ers before saving the output:

```
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
name_mapping = {}
for i, record in enumerate(alignment):
    name_mapping[i] = record.id
    record.id = "seq%i" % i
prinpin30ApfnnV911i
```

```
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
print(alignment.format("clustal"))
```

As described in Section 4.6, the `SeqRecord` object has a similar method using output formats supported

```
>>> print(alignment[:, 6:9])
```

6.3.2 Alignments as arrays

6.4.1 ClustalW

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> help(MuscleCommandline)
...
```

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(input="opuntia.fasta")
>>> stdout, stderr = muscle_cline()
>>> from StringIO import StringIO
>>> from Bio import AlignIO
>>> align = AlignIO.read(StringIO(stdout), "fasta")
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(clwstrict=True)
>>> print(muscle_cline)
muscle -clwstrict
```

Now for the ddly bits using the subprocess module, stdin and stdout:

```
>>> import subprocess
>>> import sys
>>> child = subprocess.Popen(str(cline),
...                          stdin=subprocess.PIPE,
...                          stdout=subprocess.PIPE,
...                          stderr=subprocess.PIPE,
...                          universal_newlines=True,
...                          shell=(sys.platform!="win32"))
```

That should start MUSCLE, but it will be sitting waiting for its FASTA input sequences, which we must supply via its stdin handle:

```
>>> SeqIO.write(records, child.stdin, "fasta")
6
>>> child.stdin.close()
```

After writing the six sequences to the handle, MUSCLE will still be waiting to see if that is all the FASTA sequences or not { so we must signal that this is all the input data by closing the handle. At that point MUSCLE should start to run, and we can ask for the
6aTd [0 -11.955 Td Td [IO.read leLet626Alphabeswte7e 1s3 0 -1aTdcth0 -1aT60 -1aTrow[(>>>)-uld

```
>>> handle = StringIO()
>>> SeqIO.write(records, handle, "fasta")
6
>>> data = handle.getvalue()
```

Why not try running this by hand at the command prompt? You should see it does a pairwise comparison

In this example, we told EMBOSS to write the output to a le, but you *can* tell it to write the output to stdout instead (useful if you don't want a temporary output le to get rid of { use `stdout=True` rather than the `outfile` argument), and also to read *one* of the one of the inputs from stdin (e.g. `asequence="stdin"`, much like in the MUSCLE example in the section above).

This has only scratched the surface of what you can do with `needle` and `water`. One useful trick is that

Score=292.5Local alignments are called similarly with the functionalign.localXX

```
AGAACTC
.|||||.
.GAACT.
<BLANKLINE>
```

6.5.2 The pairwise aligner object

The `PairwiseAligner` object stores all alignment parameters to be used for the pairwise alignmenthhA31(T)83((to)-29seThe

```
query_left_open_gap_score
query_left_extend_gap_score
query_internal_open_gap_score
query_internal_extend_gap_score
query_right_open_gap_score
query_right_extend_gap_score
```

target	query	score
A	-	query left open gap score
C	-	query left extend gap score
C	-	query left extend gap score
G	G	match
G	T	mismatch
G	-	query internal open gap score
A	-	query internal extend gap score
A	-	query internal extend gap score
T	T	match

gap_score	

Note that

```
>> from Bio import Align
>> aligner = Align.PairwiseAligner()
>> alignments = aligner.align("AAA", "AA")
>> for alignment in alignments:
..     print(alignment)
..
AA
|-
A-
BLANKLINE>
AA
-|
-A
BLANKLINE>
AA
||
AA
BLANKLINE>
>> for alignment in alignments:
..     print(alignment)
..
AA
|-
A-
BLANKLINE>
AA
-|
-A
BLANKLINE>
AA
||
AA
BLANKLINE>
```

can also convert the

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> seq1 = "GAACT"
>>> seq2 = "GAT"
>>> alignments = aligner.align(seq1, seq2)
>>> alignment = alignments[0]
```

```
>>> alignment
<Bio.Align.PairwiseAlignment object at 0x10204d250>
```

Each nt stores the t score:

```
>>> alignment.score
3.0
```

as well as pointers the sequences were

```
>>> alignment.target
'GAACT'
>>> alignment.query
'GAT'
```

Print the `PairwiseAlignment` object to show -11gnment explicitly:

```
>>> print(alignment)
GAACT
||--|
GA--T
<BLANKLINE>
```

```
1
>>>a9lignment>a9=>a9lignments[0]1
>>>a9print(6lignment(1)]TJ0-11.955Td[LSPADKTNVKAA1
```

a

Chapter 7

BLAST

The argument `url_base` sets the base URL for running BLAST over the internet. By default it connects to the NCBI, but one can use this to connect to an instance of NCBI BLAST running in the cloud. Please refer to the documentation for the `qblast` function for further details.

The `qblast`

```
>>> from Bio.Blast.Applications import NcbiblastxCommandline
>>> help(NcbiblastxCommandline)
...
>>> blastx_cline = NcbiblastxCommandline(query="opuntia.fasta", db="nr", evalue=0.001,
...                                      outfmt=5, out="opuntia.xml")
>>> blastx_cline
NcbiblastxCommandline(cmd='blastx', out='opuntia.xml', outfmt=5, query='opuntia.fasta',
db='nr', evalue=0.001)
>>> print(blastx_cline)
blastx -out opuntia.xml -outfmt 5 -query opuntia.fasta -db nr -evalue 0.001
>>> stdout, stderr = blastx_cline()
```

Or, you can use a

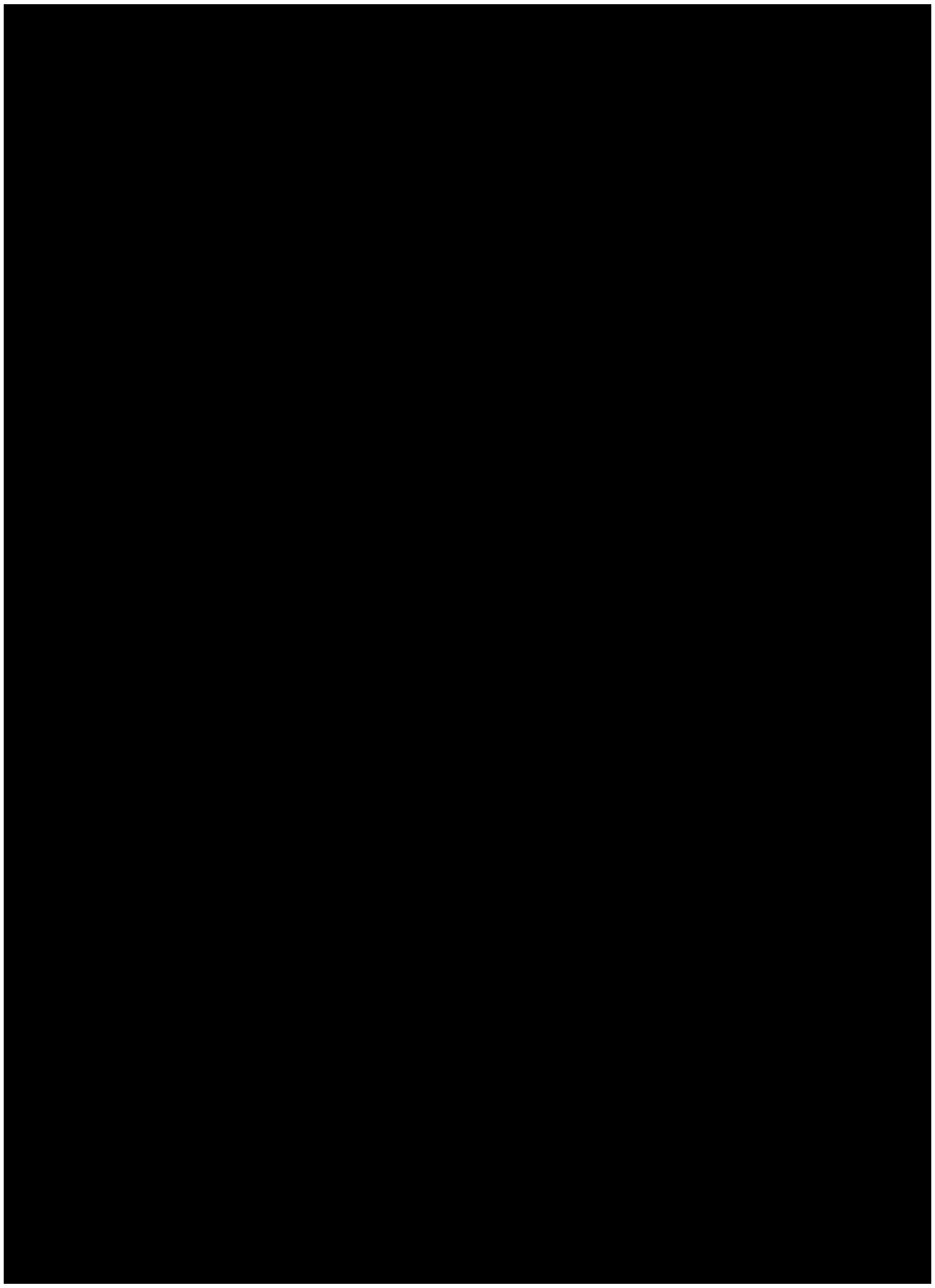

Figure 7.2: Class diagram for the PSIBlast Record class.

```
...              print("****Alignment****")
```

7.5.3 Finding a bad record somewhere in a huge plain-text BLAST le

One really ugly problem that happens to me is that I'll be parsing a huge blast le for a while, and the parser will bomb out with a ValueError. This is a serious problem, since you can't tell if the ValueError is due to a parser problem, or a problem with the BLAST. To make it even worse, you have no idea where the parse failed, so you can't just ignore the error, since this could be ignoring an important data point.

{

Chapter 8

BLAST and other sequence search tools

```
----  -----  ----------------------------------------------------------
   0      1  gi|262205317|ref|NR_030195.1|  Homo sapiens microRNA 52...
   1      1  gi|301171311|ref|NR_035856.1|  Pan troglodytes microRNA...
   2      1  gi|270133242|ref|NR_032573.1|  Macaca mulatta microRNA ...
```

To check how many items (hits) a `QueryResult` has, you can simply invoke Python's `len` method:

```
>>> len(blast_qresult)
100
>>> len(blat_qresult)
1
```

Like Python lists, you can retrieve items (hits) from a `QueryResult` using the slice notation:

```
>>> blast_qresult[0]        # retrieves the top hit
Hit(id='gi|262205317|ref|NR_030195.1|', query_id='42291', 1 hsps)
>>> blast_qresult[-1]       # retrieves the last hit
Hit(id='gi|397513516|ref|XM_003827011.1|', query_id='42291', 1 hsps)
```

To retrieve multiple hits, you can slice `QueryResult`

Here is an example of using `hit_filter` to lter out `Hit` objects that only have one HSP:

```
>>> filter_func = lambda hit: len(hit.hsps) > 1     # the callback function
>>> len(blast_qresult)      # no. of hits before filtering
100
>>> filtered_qresult = blast_qresult.hit_filter(filter_func)
```

```
HSPs: ----  --------  ---------  ------  -------------  -------------
```

What about the span column? The span values is meant to display the complete alignment length, which consists of all residues and any gaps that may be present. The PSL format do not have this information readily available and `Bio.SearchIO` this match is determined by the sequence search tool's algorithms, the

```
>>> blast_hsp.gap_num       # number of gaps
0
>>> blast_hsp.ident_num     # number of identical residues
61
```

```
>>> blast_hsquery61

>>> blast_hshit61
```

ars Tyre

This does not a ect other attributes, though. For example, you can still access the length of the query or hit alignment. Despite not displaying any attributes, the PSL format still have this information so `Bio.SearchIO` can extract them:

```
>>> blat_hsp.query_span     # length of query match
61
>>> blat_hsp.hit_span       # length of hit match
61
```

Other format-speci c attributes are still present as well:

```
>>> blat_hsp.score          # PSL score
61
>>> blat_hsp.mismatch_num   # the mismatch column
0
```

```
31359
>>> blat_hsp2.hit_span_all      # hit span of each fragment
[18, 43]
>>> blat_hsp2.hit_inter_ranges  # start and end coordinates of intervening regions in the hit sequence
[(54233122, 54264420)]
>>> blat_hsp2.hit_inter_spans   # span of intervening regions in the hit sequence
[31298]
```

At this level, the BLAT fragment looks quite similar to the BLAST fragment, save for the query and hit sequences which are not present:

```
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> blat_frag = blat_qresult[0][0][0]    # first hit, first hsp, first fragment
>>> print(blat_frag)
      Query: mystery_seq <unknown description>
        Hit: chr19 <unknown description>
Query range: [0:61] (1)
  Hit range: [54204480:54204541] (1)
  Fragments: 1 (? columns)
```

```
>>> idx = SearchIO.index("tab_2226_tblastn_001.txt", "blast-tab")
>>> sorted(idx.keys())
['gi|11464971:4-101', 'gi|16080617|ref|NP_391444.1|']
>>> idx["gi|16080617|ref|NP_391444.1|"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

Or also with the format-speci c keyword argument:

```
>>> idx = SearchIO.index("tab_2226_tblastn_005.txt", "blast-tab", comments=True)
>>> sorted(idx.keys())
['gi|11464971:4-101', 'gi|16080617|ref|NP_391444.1|', 'random_s00']
>>> idx["gi|16080617|ref|NP_391444.1|"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

Or with the `key_function` argument, as in `Bio.SeqIO`:

```
>>> key_function = lambda id: id.upper()    # capitalizes the keys
>>> idx = SearchIO.index("tab_2226_tblastn_001.txt", "blast-tab", key_function=key_function)
>>> sorted(idx.keys())
['GI|11464971:4-101', 'GI|16080617|REF|NP_391444.1|']
>>> idx["GI|16080617|REF|NP_391444.1|"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

`Bio.SearchIO.index_db` works like as `index`

```
>>> from Bio import SearchIO
>>> SearchIO.convert("mirna.xml", "blast-xml", "results.tab", "blast-tab")
(3, 239, 277, 277)
```

As `convert` uses `write`, it is only limited to format conversions that have all the required attributes. Here, the BLAST XML le provides all the default values a BLAST tabular le requires, so it works just ne. However, other format conversions are less likely to work since you need to manually assign the required attributes rst.

Chapter 9

Accessing NCBI's Entrez databases

Entrez (http://www.ncbi.nlm.nih.gov/Entrez

The Entrez Programming Utilities can also generate output in other formats, such as the Fasta or

In conclusion, be sensible with your usage levelsa1622(Ifh)-392(y)28(oh)-392planr otsr cosiderh

Forhexamplen,ifh yohs tohallh theh(cosiderh)-77(fetcy)28hingheacy cy

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

```
>>> from Bio import Entrez
>>> handle = Entrez.einfo()
>>> record = Entrez.read(handle)
```

Now record is a dictionary with exactly one key:

```
>>> record.keys()
['DbList']
```

The values stored in this key is the list of database names shown in the XML above:

```
>>> record["DbList"]
['pubmed', 'protein', 'nucleotide', 'nuccore', 'nucgss', 'nucest',
 'structure', 'genome', 'books', 'cancerchromosomes', 'cdd', 'gap',
 'domains', 'gene', 'genomeprj', 'gensat', 'geo', 'gds', 'homologene',
 'journals', 'mesh', 'ncbisearch', 'nlmcatalog', 'omia', 'omim', 'pmc',
 'popset', 'probe', 'proteinclusters', 'pcassay', 'pccompound',
 'pcsubstance', 'snp', 'taxonomy', 'toolkit', 'unigene', 'unists']
```

For each of these databases, we can use EInfo again to obtain more information:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> handle = Entrez.einfo(db="pubmed")
>>> record = Entrez.read(handle)
>>> record["DbInfo"]["Description"]
'PubMed bibliographic record'

>>> record["DbInfo"]["B5ibliographie525(Entrez)fo"89604'051
>>> record["DbInfo"].keys()Up33(eibliographie525(Entrez)2008/05/24>>>)-506:45't']Fw
```

```
FILT, Filter, Limits the records
TITL, Title, Words in title of publication
WORD, Text Word, Free text associated with publication
MESH, MeSH Terms, Medical Subject Headings assigned to publication
MAJR, MeSH Major Topic, MeSH terms of major importance to publication
```

```
>>> print("{} computational journals found".format(record["Count"]))
117 computational Journals found
>>> print("The first 20 are\n{}".format(record["IdList"]))
['101660833', '101664671', '101661657', '101659814', '101657941',
 '101653734', '101669877', '101649614', '101647835', '101639023',
 '101627224', '101647801', '101589678', '101585369', '101645372',
 '101586429', '101582229', '101574747', '101564639', '101671907']
```

Again, we could use EFetch to obtain more information for each of these journal IDs.

ESearch has many useful options | see the ESearch help page for more information.

9.4 EPost: Uploading a list of identi ers

EPost uploads a list of UIs for use in subsequent search strategies; see the EPost help page for more information. It is available from Biopython through the `Bio.Entrez.epost()` function.

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

```
                /organism="Selenipedium aequinoctiale"
                /organelle="plastid:chloroplast"
                /mol_type="genomic DNA"
                /specimen_voucher="FLAS:Blanco 2475"
                /db_xref="taxon:256374"
gene            <1..>1302
                /gene="matK"
CDS             <1..>1302
                /gene="matK"
                /codon_start=1
                /transl_table=11
                /product="maturase83.685N3.685-en_voucherNg_id="ACC99456.1"n_start=1
```

'Se.Entrez.eseach(db=1

```
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> handle = Entrez.efetch(db="nucleotide", id="EU490707", retmode="xml")
>>> record = Entrez.read(handle)
>>> handle.close()
>>> record[0]["GBSeq_definition"]
'Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast'
>>> record[0]["GBSeq_source"]
'chloroplast Selenipedium aequinoctiale'
```

S.bTJ/050 at dealt with sequences. For examples of parsing le formats speci c to the other databases (e.g. the MEDLINE format used in PubMed), see Section 9.12.

If you want to perform a search with Bio.Entrez.esearch()

9.9 ESpell: Obtaining spelling suggestions

ESpell retrieves spelling suggestions. In this example, we use `Bio.Entrez.espell()` to obtain the correct spelling of Biopython:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> handle = Entrez.espell(term="biopythooon")
>>> record = Entrez.read(handle)
>>> record["Query"]
'biopythooon'
>>> record["CorrectedQuery"]
'biopython'
```

See the ESpell help page for more information. The main use of this is for GUI tools to provide automatic suggestions for search terms.

9.10 Parsing huge Entrez XML les

The `Entrez.read`

>>8 9.925(record)-525(=)-525(Entreho)-525(you)-525 0 -1oon")

which will generate the following traceback:

```
>>> Entrez.read(handle)
Traceback (most recent call last):
  ...
Bio.Entrez.Parser.CorruptedXMLError: Failed to parse the XML data (no element found: line 16, column 0)
```

Note that the error message tells you at what point in the XML le the error was detected.

The third type of error occurs if the XML le contains tags that do not have a description in the corresponding DTD le. This is an example of such an XML le:

```
<?xml version="1.0"?>
<!DOCTYPE eInfoResult PUBLIC "-//NLM//DTD eInfoResult, 11 May 2002//EN" "http://www.ncbi.nlm.nih.gov/en
<eInfoResult>
        <DbInfo>
        <DbName>pubmed</DbName>
        <MenuName>PubMed</MenuName>
        <Description>PubMed bibliographic record</Description>
        <Count>20161961</Count>

                Þ
                <ocsumLislt>
                        <ocsumt>
                                DsuName>PuDdatd</suName>
                                <sType>4d</sType>>


                                        <ocsumt>

                                                ...

                                                                /<eInfoResult>

                                                                [illegible]
                                                      [illegible]
                                                                           parsor willstoph adhlipdai


                                                                           >>>"eiInf3.?xm"e)
                                                                           >>> recor> Entrez.read(
                                                                           Traceback (most recent
                                                                             ...
                                                                           Bio.Entrez.ParserV(lipda
                                                                           t in the


                                                                           >>>"eiInf3.?xm"e)
```

Of course, the information contained in the XML tags that are not in the DTD are not present in the record returned by `Entrez.read`.

9.12 Specialized parsers

multiple functionality that make it easier to create customised pipelines or
analysis. This review briefly compares the quirks of the underlying languages

```
>>> handle = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="xml")
>>> records = Entrez.read(handle)
>>> for record in records["PubmedArticle"]:
...     print(record["MedlineCitation"]["Article"]["ArticleTitle"])
Biopython: freely available Python tools for computational molecular biology and
 bioinformatics.
```

```
ID          Hs.2
TITLE       N-acetyltransferase 2 (arylamine N-acetyltransferase)
GENE        NAT2
CYTOBAND    8p22
GENE_ID     10
LOCUSLINK   10
HOMOL       YES
EXPRESS      bone| connective tissue| intestine| liver| liver tumor| normal| soft tissue/muscle tissue
RESTR_EXPR   adult
CHROMOSOME  8
STS         ACC=PMC310725P3 UNISTS=272646
STS         ACC=WIAF-2120 UNISTS=44576
STS         ACC=G59899 UNISTS=137181
...
STS         ACC=GDB:187676 UNISTS=155563
PROTSIM     ORG=10090; PROTGI=6754794; PROTID=NP_035004.1; PCT=76.55; ALN=288
PROTSIM     ORG=9796; PROTGI=149742490; PROTID=XP_001487907.1; PCT=79.66; ALN=288
PROTSIM     ORG=9986; PROTGI=126722851; PROTID=NP_001075655.1; PCT=76.90; ALN=288
...
PROTSIM     ORG=9598; PROTGI=114619004; PROTID=XP_519631.2; PCT=98.28; ALN=288

SCOUNT      38
SEQUENCE    ACC=BC0ROT
...
Pn84550130655.1;Pn84550130755.1;UNISTS=44576
...SCOUN[(Pn811629COU955.1;)-5(Pn811629CO6055.1;)-0-TYPE=mRNA5(UNISTS=445788)]TJ0-11.955Td[(...)D-52521
...Pn847113.9855.1;Pn847113.9955.1;UNISTS=44576
```

Specialized objects are returned for the STS, PROTSIM

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> handle = Entrez.esearch(db="pubmed", term="orchid", retmax=463)
>>> record = Entrez.read(handle)
>>> handle.close()
>>> idlist = record["IdList"]
```

This returns a Python list containing all of the PubMed IDs of articles related to orchids:

```
>>> print(idlist)
['18680603', '18665331', '18661158', '18627489', '18627452', '18612381',
'18594007', '18591784', '18589523', '18579475', '18575811', '18575690',
...
```

Hopefully this section gave you an idea of the power and exibility of the Entrez and Medline interfaces

```
AY851612, length 892, with 3 features
AY851611, length 881, with 3 features
AF191661, length 895, with 3 features
AF191665, length 902, with 3 features
AF191664, length 899, with 3 features
AF191663, length 899, with 3 features
AF191660, length 893, with 3 features
AF191659, length 894, with 3 features
AF191658, length 896, with 3 features
```

9.15 Using the history and WebEnv

```
out_handle.write(data)
data = fetch_handle.read()

else:
if 500 <= err.code <= 599:

idtype="acc")
webenv=webenv, query_key=query_key,
retstart=start, retmax=batch_size,
rettype="fasta", retmode="text",

out_handle.close()

fetch_handle = Entrez.efetch(db="nucleotide",
except HTTPError as err:

try:
attempt += 1

while attempt < 3:
print("Going to download record %i to %i" % (start+1, end))
from urllib2 import HTTPError # for Python 2
from urllib.error import HTTPError # for Python 3
```

```
...                                     id=",".join(pmc_ids)))
>>> pubmed_ids = [link["Id"] for link in results2[0]["LinkSetDb"][0]["Link"]]
>>> pubmed_ids
['19698094', '19450287', '19304878', ..., '15985178']
```

This time you can immediately spot the Biopython application note as the third hit (PubMed ID 19304878).

Now, let's do that all again but with the history . . . *TODO*.

And nally, don't forget to include your

Chapter 10

Swiss-Prot and ExPASy

10.1 Parsing Swiss-Prot les

Swiss-Prot (http://www.expasy.org/sprot) is a hand-curated database of protein sequences. Biopython can parse the \plain text" Swiss-Prot le format, which is still used for the UniProt Knowledgebase which

Open a Swiss-Prot le over the internet from the ExPASy database (see section 10.5.1):

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

As of June 2009, the full Swiss-Prot database downloaded from ExPASy contained 468851 Swiss-Prot records. One concise way to build up a list of the record descriptions is with a list comprehension:

```
>>> from Bio import SwissProt
>>> handle = open("uniprot_sprot.dat")
>>> descriptions = [record.description for record in SwissProt.parse(handle)]
>>> len(descriptions)
468851
>>> descriptions[:5]
['RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-2L;']
```

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)
468851
```

Because this is such a large input le, either way takes about eleven minutes on my ne 0 G 0 g is r050dsing the urniprot_sprot.dat"]TJ/F8 9.9626 Tf 9.223 70 -d [(r014le,-333(a))-334(isput)051

```
HI   Ligand: Metal-binding; 2Fe-2S.
CA   Ligand.
//
ID   3D-structure.
AC   KW-0002
DE   Protein, or part of a protein, whose three-dimensional structure has
DE   been resolved exper-dimg;an5nxamplture as
DEof whoseofas
DEan525toreticional
HI 3D-structure.
CA
//
ID   24e-2S.
```

```
[['P11151', 'LIPL_BOVIN'], ['P11153', 'LIPL_CAVPO'], ['P11602', 'LIPL_CHICK'],
['P55031', 'LIPL_FELCA'], ['P06858', 'LIPL_HUMAN'], ['P11152', 'LIPL_MOUSE'],
['O46647', 'LIPL_MUSVI'], ['P49060', 'LIPL_PAPAN'], ['P49923', 'LIPL_PIG'],
['Q06000', 'LIPL_RAT'], ['Q29524', 'LIPL_SHEEP']]
```

The

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

>>> accessions = ["O23729", "O23730", "O23731"]
>>> records = []

>>> for accession in accessions:
...     handle = ExPASy.get_sprot_raw(accession)
...     record = SwissProt.read(handle)
...     records.append(record)
```

If the accession number you provided to `ExPASy.get_sprot_raw` does not exist, then `SwissProt.read(handle)` will raise a `ValueError`. You can catch `ValueExcep0Tdo`

```
>>> for accession in accessions:
...     handle = ExPASy.get_sprot_raw(accession)
...
```

```
handle = ScanProsite.scan(seq=sequence)
```

Ê[{Executing obtain the search results in raw XML format. Instead, use

```
106, 'stop': 111, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00008'}
```

for more information. As an example, passing

The available keys are `name`, `head`, `deposition_date`, `release_date`, `structure_method`, `resolution`, `structure_reference` (which maps to a list of references), `journal_reference`, `author`, and `compound` (which maps to a dictionary with various information about the crystallized compound).

The dictionary can also be created without creating a `Structure` object, ie. directly from the PDB le:

11.2 Structure representation

The overall layout of a `Structure`

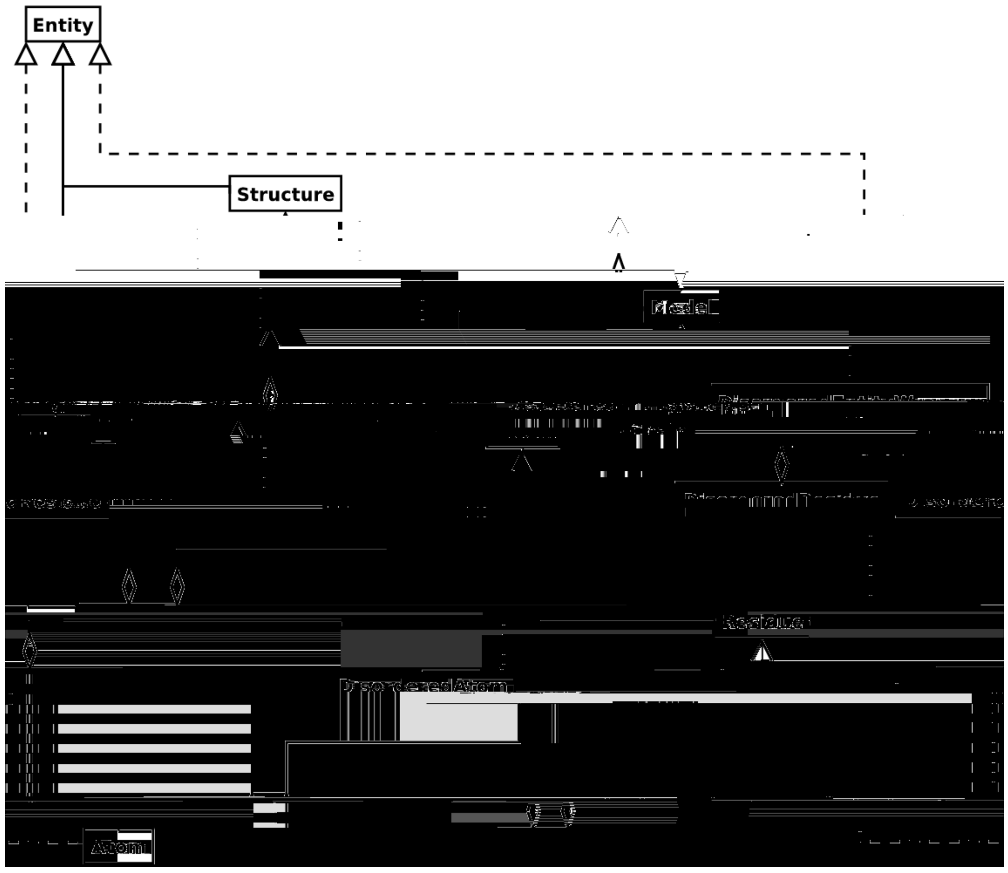

Figure 11.1: UML diagram of SMCRA architecture of the `Structure` class used to represent a macromolec-

The Model with id 0

The Chain with id "A"

The Residue with id (" ", 10, "A").

11.2.4 Residue

A residue id is a tuple with three elements:

The **hetero- eld** (het eld): this is

{ `'W'` in the case of a water molecule;

{ `'H_'` followed by the residue name for other hetero residues (e.g. `'H_GLC'` in the case of a glucose molecule);

{ blank for standard amino and nucleic acids.

This scheme is adopted for reasons described in section 11.4.1.

The **sequence identi er** (resseq), an integer describing the position of the residue in the chain (e.g., 100);

The **insertion code**

11.2.5 Atom

The Atom object stores the data associated with an atom, and has no children. The id of an atom is its atom name (e.g. \OG" for the side chain oxygen of a Ser residue). An Atom id needs to be unique in a Residue. Again, an exception is made for disordered atoms, as described in section 11.3.2.

```
# find rotation matrix that rotates n
# -120 degrees along the ca-c vector
>>> rot = rotaxis(-pi * 120.0/180.0, c)
# apply rotation to ca-n vector
>>> cb_at_origin = n.left_multiply(rot)
# put on top of ca atom
>>> cb = cb_at_origin+ca
```

This example shows that it's possible to do some quite nontrivial vector operations on atomic data,

```
>>> atom.disordered_select("A") # select altloc A atom
>>> print(atom.get_altloc())
"A"
>>> atom.disordered_select("B") # select altloc B atom
>>> print(atom.get_altloc())
"B"
```

11.3.3 Disordered residues

11.4.3 Other hetero residues

or to get all atoms from a chain:

11.6 Analyzing structures

11.6.1 Measuring distances

The minus operator for atoms has been overloaded to return the distance between two atoms.

```
# Get some atoms
>>> ca1 = residue1["CA"]
>>> ca2 = residue2["CA"]
# Simply subtract the atoms to get their distance
>>> distance = ca1-ca2
```

11.6.2 Measuring angles

Use the vector representation of the atomic coordinates, and the `calc`

```
# The moving atoms will be put on the fixed atoms#onon the moving atoms
```

Code	Secondary structure
H	-helix
B	Isolated -bridge residue
E	Strand

11.7.1 Examples

The PDBParser/Structure class was tested on about 800 structures (each belonging to a unique SCOP superfamily). This takes about 20 minutes, or on average 1.5 seconds per structure. Parsing the structure of the large ribosomal subunit (1FKK), which contains about 64000 atoms, takes 10 seconds on a 1000 MHz PC.

Three exceptions were generated in cases where an unambiguous data structure could not be built. In all three cases, the likely cause is an error in the PDB le that should be corrected. Generating an exception in these cases is much better than running the chance of incorrectly describing the structure in a data structure.

11.7.1.1 Duplicate residues

One structure contains two amino acid residues in one chain with the same sequence identi er (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).

11.7.1.2 Duplicate atoms

Structure 1EJG contains a Ser/Pro point mutation in chain A at position 22. In turn, Ser 22 contains some disordered atoms. As expected, all atoms belonging to Ser 22 have a non-blank altloc speci er (B or C). All atoms of Pro 22 have altloc A, except the N atom which has a blank altloc. This generates an exception, because all atoms belonging to two residues at a point mutation should have non-blank altloc. It turns out that this atom is probably shared by Ser and Pro 22, as Ser 22 misses the N atom. Again, this points to a problem in the le: the N atom should be present in both the Ser and the Pro residue, in both cases associated with a suitable altloc identi er.

11.7.2 Automatic correction

Some errors are quite common and can be easilyare-1(oist)-306(for)-30uch intierres-]TJ 0 -11.955 Td [(thtion) cTese aases a

11.7.121 D ,bl-1(ink)-338(altlo)-232c aor a aisordered alomSNormll ach isordered tom of the same atom. This is automaticmll intierresed in the right way

11.7.122 Brok chains

The insertion code (icode).

The het eld string (\W" for waters and \H_" followed by the residue name for other hetero residues)

The residue names of the residues in the case of point mutations (to store the Residue objects in a DisorderedResidue object).

11.8.3 Keeping a local copy of the PDB up to date

This can also be done using the `PDBList` object. One simply creates a `PDBList`

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

So we have two populations, the rst with two individuals, the second with only one. The rst individual of the rst population is called Ind1, allelic information for each of the 3 loci follows. Please note that for any locus, information might be missing (see as an example, Ind2 above).

A few utility functions to manipulate GenePopaisebample,1(o)-r81(1(o,)-33ions)-334(an)4333(exam:51.)0 G 0 g 0 G 0 g

Chapter 13

Phylogenetics with Bio.Phylo

```
                Clade(name='C')
                Clade(name='D')
        Clade()
            Clade(name='E')
            Clade(name='F')
            Clade(name='G')
```

The `Tree` object contains global information about the tree, such as whether it's rooted or unrooted. It has one root clade, and under that, it's nested lists of clades all the way down to the tips.

The function `draw_ascii` creates a simple ASCII-art (plain text) dendrogram. This is a convenient visualization for interactive exploration, in case better graphical tools aren't available.

```
>>> from Bio import Phylo
>>> tree = Phylo.read("simple.dnd", "newick")
>>> Phylo.draw_ascii(tree)
                                                  ________________________ A
                          ________________________|
```

```
>>> tree.clade[0, 1].color = "blue"
```

Finally, show our work (see Fig. 13.2

```
<phy:branch_length>1.0</phy:branch_length>
<phy:clade>
  <phy:branch_length>1.0</phy:branch_length>
  <phy:clade>
    <phy:name>A</phy:name>
    ...
```

```
Clade(branch_length=1.4, name='C')
```

13.3 View and export trees

The simplest way to get an overview of a Tree object is to

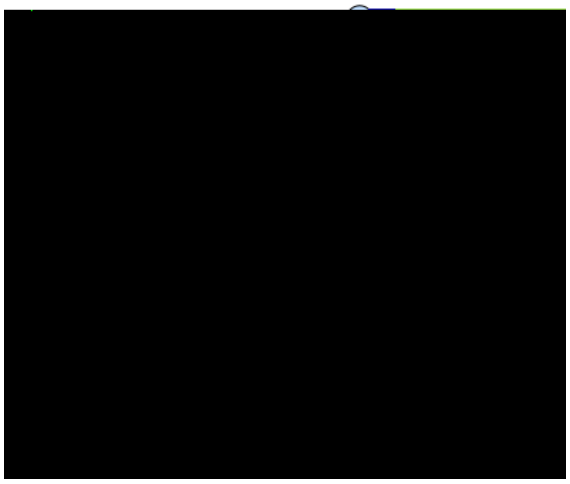

Figure 13.5: A ronnigure drawn with `draw_graphviz`. Since we ronnigurewe can see 0heurennureust

Figure 13.7: A larger tree, using `neato`

Note that branch lengths are not displayed accurately, because Graphviz ignores them when creating

Since oating-point arithmetic can produce some strange behavior, we don't support matching `floats` directly. Instead, use the boolean `True` to match every element with a nonzero value in the speci ed attribute, then lter on that attribute manually with an inequality (or exact number, if you like living dangerously).

If the dictionary contains multiple entries, a matching element must match each of the given attribute values | think \and", not \or".

A **function** taking a single argument (it will be applied to each element in the tree), returning True or False. For convenience, LookupError, AttributeError and ValueError are silenced, so this provides
626vno28(e)28(viy333(thio334(nosear28(h)-361(thf)-346(no15oating-p)-28(7in)28(t)-300(no56(alues)-334(|))-361(thine

13.4.2 Information methods

These methods provide information about the whole tree (or any clade).

`common_ancestor` Find the most recent common ancestor of all the given targets. (This will be a Clade object). If no target is given, returns the root of the current clade (the one this method is called from); if 1 target is given, this returns the target itself. However, if any of the speci ed targets are not found in the current tree (or clade), an exception is raised.

`count_terminals` Counts the number of terminal (leaf) nodes within the tree.

`depths` Create a mapping of tree clades to depths. The result is a dictionary where the keys are all of the Clade instances in the tree, and the values are the distance from the root to each clade (including terminals). By default the distance is the cumulative branch length leading to the clade, but with the `unit_branch_lengths=True` option, only the number of branches (levels in the tree) is counted.

`distance` Calculate the sum of the branch lengths between two targets. If only one target is speci ed, the other is the root of this tree.

`total_`

prune

13.6 PAML integration

Biopython 1.58 brought support for PAML (http://abacus.gene.ucl.ac.uk/software/paml.html), a

Bio.Nexus port Much of this module was written during Google Summer of Code 2009, under the auspices of NESCent, as a project to implement Python support for the phyloXML data format (see 13.4.4).

Chapter 14

Sequence motif analysis using Bio.motifs

then we can create a Motif object as follows:

```
>>> m = motifs.create(instances)
```

The instances are saved in an attribute `m.instances`, which is essentially a Python list with some added

```
>>> m.alphabet
IUPACUnambiguousDNA()
>>> m.alphabet.letters
'GATC'
>>> sorted(m.alphabet.letters)
['A', 'C', 'G', 'T']
>>> m.counts["A",:]
(3, 7, 0, 2, 1)
>>> m.counts[0,:]
(3, 7, 0, 2, 1)
```

14.2 Reading motifs

```
>MA0052.1 MEF2A
```

```
>>> from Bio.motifs.jaspar.db import JASPAR5
>>>
>>> JASPAR_DB_HOST = <hostname>
>>> JASPAR_DB_NAME = <db_name>
>>> JASPAR_DB_USER = <user>
>>> JASPAR_DB_PASS = <passord>
>>>
>>> jdb = JASPAR5(
...     host=JASPAR_DB_HOST,
...     name=JASPAR_DB_NAME,
...     user=JASPAR_DB_USER,
...     password=JASPAR_DB_PASS
```

```
Collection CORE      1       2       3       4       5ORE    4.00  19.00    0.00    0.00    0.00    0.00  16.00
```

```
>>> motif.pseudocounts = motifs.jaspar.calculate_pseudocounts(motif)
```

```
********************************************************************************
MEME - Motif discovery tool
********************************************************************************
MEME version 3.0 (Release date: 2004/08/18 09:07:01)
...
```

Further down, the input set of training sequences is recapitulated:

```
********************************************************************************
TRAINING SET
********************************************************************************
DATAFILE= INO_up800.s
ALPHABET= ACGT
Sequence name            Weight Length  Sequence name            Weight Length
-------------            ------ ------  -------------            ------ ------
CHO1                     1.0000    800  CHO2                     1.0000    800
FAS1                     1.0000    800  FAS2                     1.0000    800
ACC1                     1.0000    800  INO1                     1.0000    800
OPI3                     1.0000    800
********************************************************************************
```

TMtif d1-525(dDescripton)-TJ 0-41.843-11.955 Td [(T------------)
and the exact command line that was us09:07:01)
...

Further down, the input set of trainin traOMMAND-525(LeIN)-525(vSUMMARY]TJ 0 -11.955 Td [(********************

```
>>> record.version
'3.0'
>>> record.datafile
'INO_up800.s'
>>> record.command
'meme -mod oops -dna -revcomp -nmotifs 2 -bfile yeast.nc.6.freq INO_up800.s'
>>> record.alphabet
IUPACUnambiguousDNA()
>>> record.sequences
['CHO1', 'CHO2', 'FAS1', 'FAS2', 'ACC1', 'INO1', 'OPI3']
```

The record is an object of the `Bio.motifs.meme.Record` class. The class inherits from list, and you can think of `record` as a list of Motif objects:

```
>>> len(record)
2
>>> motif = record[0]
>>> print(motif.consensus)
TTCACATGCCGC
>>> print(motif.degenerate_consensus)
TTCACATGSCNC
```

In addition to these generic motif attributes, each motif also stores its speci c information as calculated by MEME. For example,

```
>>> motif.num_occurrences
7
>>> motif.length
12
>>> evalue = motif.evalue
>>> print("%3.1g" % evalue)
0.2
>>> motif.name
'Motif 1'
```

In addition to using an index into the record, as we did above, you can also nd it by its name:

```
>>> motif = record["Motif 1"]
```

Each motif has an attribute `.instances` with the sequence instances in which the motif was found, providing some information on each inwas found,as>>> m1n(record)

```
12
>>> pvalue = motif.instances[0].pvalue
>>> print("%5.3g" % pvalue)
1.85e-08
```

To parse a TRANSFAC le, use

```
>>> with open("transfac.dat") as handle:
...     record = motifs.parse(handle, "TRANSFAC")
...
```

The overall version number, if available, is stored as `record.version`

Table 14.2: Fields used to store references in TRANSFAC les

RN

```
A:   0.40   0.84   0.07   0.29   0.18
C:   0.04   0.04   0.60   0.27   0.71
G:   0.04   0.04   0.04   0.38   0.04
T:   0.51   0.07   0.29   0.07   0.07
<BLANKLINE>
```

```
>>> for pos, seq in r.instances.search(test_seq):
...     print("%i %s" % (pos, seq))
...
6 GCATT
20 GCATT
```

14.6.2 Searching for matches using the PSSM score

It's just as easy to look for positions, giving rise to high log-odds scores against our motif:

```
>>> for position, score in pssm.search(test_seq, threshold=3.0):
...     print("Position %d: score = %5.3f" % (position, score))
...
Position 0: score = 5.622
Position -20: score = 4.601
Position 1tion 1tion 1tion0.031
Position 30: score = 5738:
Position 60: score = 4.601
```

negativ positionerefere to.instancee(the)-453(motif)-46(founde)-453ono(the)-453revstrande the(ates)t)-425(seueancs,)-49(anc
oth for positi ade for egativ values of

```
>>> distribution = pssm.distribution(background=background, precision=10*on(background=background, p1
        0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00
<BLANKLINE>
>>> print(motif.pwm)
        0      1      2      3      4      5
A:   0.20   0.95   0.00   0.00   0.00   0.00
C:   0.80   0.00   1.00   0.00   0.00   0.00
G:   0.00   0.05   0.00   1.00   0.00   1.00
T:   0.00   0.00   0.00   0.00   1.00   0.00
<BLANKLINE>

>>> print(motif.pssm)
        0      1      2      3      4      5
A:  -0.32   1.93   -inf   -inf   -inf   -inf
C:   1.68   -inf   2.00   -inf   -inf   -inf
G:   -inf  -2.32   -inf   2.00   -inf   2.00
T:   -inf   -inf   -inf   -inf   2.00   -inf
<BLANKLINE>
```

The negative in nities appear here because the corresponding entry in the frequency matrix is 0, and we are using zero pseudocounts by default:

```
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.pseudocounts[letter]))
...
A: 0.00
C: 0.00
G: 0.00
T: 0.00
```

If you change the `.pseudocounts` attribute, the position-frequency matrix and the position-speci c scoring matrix are recalculated automatically:

```
>>> motif.pseudocounts = 3.0
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.pseudocounts[letter]))
...
A: 3.00
C: 3.00
G: 3.00
T: 3.00

>>> print(motif.pwm)
        0      1      2      3      4      5
A:   0.22   0.69   0.09   0.09   0.09   0.09
C:   0.59   0.09   0.72   0.09   0.09   0.09
G:   0.09   0.12   0.09   0.72   0.09   0.72
T:   0.09   0.09   0.09   0.09   0.72   0.09
<BLANKLINE>
```

```
>>> print(motif.pssm)
        0      1      2      3      4      5
A:  -0.19   1.46  -1.42  -1.42  -1.42  -1.42
C:   1.25  -1.42   1.52  -1.42  -1.42  -1.42
G:  -1.42  -1.00  -1.42   1.52  -1.42   1.52
T:  -1.42  -1.42  -1.42  -1.42   1.52  -1.42
<BLANKLINE>
```

You can also set the `.pseudocounts` to a dictionary over the four nucleotides if you want to use di erent pseudocounts for them. Setting `motif.pseudocounts` to `None` resets it to its default value of zero.

The position-speci c scoring matrix depends on the background distribution, which is uniform by default:

```
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.background[letter]))
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

Again,k 8(0ify27(k)-er)-38k back33(itd)-260(distrib(k)-er)-38k position-spec8k sringngx(sringi)-333(irecalculated(default:)]T

```
>>> print("%f" % motif.pssm.mean(motif.background))
4.703928
```

as well as its standard deviation:

```
>>> print("%f" % motif.pssm.std(motif.background))
3.290900
```

and its distribution:

```
>>> m_reb1.pseudocounts = {"A":0.6, "C": 0.4, "G": 0.4, "T": 0.6}
>>> m_reb1.background = {"A":0.3,"C":0.2,"G":0.2,"T":0.3}
>>> pssm_reb1 = m_reb1.pssm
>>> print(pssm_reb1)
        0      1      2      3      4      5      6      7      8
A:   0.00  -5.67  -5.67   1.72  -5.67  -5.67  -5.67  -5.67  -0.97
C:  -0.97  -5.67  -5.67  -5.67   2.30   2.30   2.30  -5.67  -0.41
G:   1.30  -5.67  -5.67  -5.67  -5.67  -5.67  -5.67   1.57   1.44
T:  -1.53   1.72   1.72  -5.67  -5.67  -5.67  -5.67   0.41  -0.97
<BLANKLINE>
```

`.version`

`.command`

The motifs returned by the MEME Parser can be treated exactly like regular Motif objects (with instances), they also provide some extra functionality, by adding additional information about the instances.

```
>>> motifsM[0].consensus
Seq('CTCAATCGTA', IUPACUnambiguousDNA())
>>> motifsM[0].instances[0].sequence_name
'SEQ10;'
>>> motifsM[0].instances[0].start
3
>>> motifsM[0].instances[0].strand
'+'

>>> motifsM[0].instances[0].pvalue
8.71e-07
```

14.10 Useful links

Sequence motif in wikipedia

PWM in wikipedia

Consensus sequence in wikipedia

Comparison of di erent motif nding programs

`srand` with the epoch time in seconds, and use the first two random numbers generated by `rand` as seeds for the uniform random number generator in `Bio.Cluster`.

15.1 Distance functions

In order to cluster items into groups based on their similarity, we should first define what exactly we mean by *similar*. `Bio.Cluster` provides eight distance functions, indicated by a single character, to measure similarity, or conversely, distance:

- `'e'`: Euclidean distance;

where

$$
\begin{aligned}
{}^{(0)}_{x} &= \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}; \\
{}^{(0)}_{y} &= \sqrt{\frac{1}{n}\sum_{i=1}^{n} y_i^2}:
\end{aligned}
$$

Weighting

For most of the distance functions available in `Bio.Cluster`, a weight vector can be applied. The weight vector contains weights for the items in the data vector. If the weight for item i is w_i, then that item is treated as if it occurred w_i times in the data. The weight do not have to be integers.

Calculating the distance matrix

The distance matrix is a square matrix with all pairwise distances between the items in `data`, and can be calculated by the function `distancematrix` in the `Bio.Cluster` module:

```
>>> from Bio.Cluster import distancematrix
>>> matrix = distancematrix(data)
```

where the following arguments are de ned:

- `data` (required)
  Array containing the data for the items.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `weight` (default: `None`)
  The weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.
- `transpose` (default: `0`)
  Determines if `data` distances bthweer

`method` (default: a)
describes how the center of a cluster is found:

{ `method=='a'`: arithmetic mean (*k*-means clustering);

{ `method=='m'`: median (*k*-medians clustering).

For other values of `method`, the arithmetic mean is used.

`dist` (default: `'e'`, Euclidean distance)
De nes the distance function to be used (see 15.1

`nclusters` (default: `2`)
The number of clusters *k*.

`npass` (default: `1`)

To apply hierarchical clustering on a precalculated distance matrix, specify the `distancematrix` argument when calling `treecluster`

The rst step to calculate a SOM is to randomly assign a data vector to each cluster in the topology. If rows are being clustered, then the number of elements in each data vector is equa]b91(then)-281(the)-281(n)28(um)28(b)-28(e

`inittau` (default: `0.02`)
The initial value for the parameter

This function returns a tuple `columnmean, coordinates, components, eigenvalues`:

`columnmean`
Array containing the mean over each column in

order
ewe
The

expid
This is a list containing a description of each sample, e.g. experimental condition.

eweight

ordert
ThesamplsnhouledbntorsedIfn,at(a)-02[9n
,rtoert

Calculating the distance between clusters

To calculate the distance between clusters of items stored in the record, use

```
>>> distance = record.clusterdistance()
```

`nxgrid, nygrid` (default: `2, 1`)
The number of cells horizontally and vertically in the rectangular grid on which the Self-Organizing Map is calculated.

`inittau` (default: `0.02`)
The initial value for the parameter   that is used in the SOM algorithm. The default value for `inittau`

15.8 Example calculation

This is an example of a hierarchical clustering calculation, using single linkage clustering for genes and

The logistic regression model gives us appropriate values for the parameters $_0$

```
          [85, -193.94],
          [16, -182.71],
          [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`

```
Iteration: 2 Log-likelihood function: -5.76877209868
Iteration: 3 Log-likelihood function: -5.11362294338
```

`0`, corresponding to class OP and class NOP, respectively. For example, let's consider the gene pairs *yxcE*, *yxcD* and *yxiB*, *yxiA*:

Table 16.2: Adjacent gene pairs of unknown operon status.

Gene pair	Intergene distance x_1	Gene expression score x_2
yxcE | *yxcD*	6	-173.143442352
yxiB |		

```
...
>>> x = [6, -173.143442352]
>>> print("yxcE, yxcD:", kNN.classify(model, x, weight_fn = weight))
yxcE, yxcD: 1
```

By default, all neighbors are given an equal weight.

To nd out how con dent we can be in these predictions, we can call the `calculate` function, which will calculate the total weight assigned to the classes OP and NOP. For the default weighting scheme, this reduces to the number of neighbors in each category. For *yxcE*, *yxcD*, we nd

```
>>> x = [6, -173.143442352]
>>> weight = kNN.calculate(model, x)
>>> print("class OP: weight =", weight[0], "class NOP: weight =", weight[1])
class OP: weight = 0.0 class NOP: weight = 3.0
```

which means that all three neighbors of `x1`, `x2` are in the NOP class. As another example, for *yesK*, *yesL* we nd

```
>>> x = [117, -267.14]
>>> weight = kNN.calculate(model, x)
>>> print("class OP: weight =", weight[0], "class NOP: weight =", weight[1])
class OP: weight = 2.0 class NOP: weight = 1.0
```

which means that two neighbors are op pairs and one neighbor is a non-op pair.

To get some idea of the prediction accuracy of the *k*-nearest neighbors approach, we can apply it to the training data:

```
>>> for i in range(len(ys)):
        print("True:", ys[i], "Predicted:", kNN.classify(model, xs[i]))
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
```

showing that the prediction is correct for all but two of the gene pairs. A more reliable estimate of the prediction accuracy can be found from a leave-one-out analysis, in which the model is recalculated from the training data after removing the gene to be predicted:

```
>>> k = 3
>>> for i in range(len(ys)):
        model = kNN.train(xs[:i]+xs[i+1:], ys[:i]+ys[i+1:], k)
```

```
print("True:", ys[i], "Predicted:", kNN.classify(model, xs[i]))
```

Chapter 17

Graphics including GenomeDiagram

The `Bio.Graphics` module depends on the third party Python library ReportLab. Although focused on producing PDF les, ReportLab can also create encapsulated postscript (EPS) and (SVG) les. In addition

17.1.4 A bottom up example

Now let's produce exactly the same figures, but using the bottom up approach. This means we create the different objects directly (and this can be done in almost any order) and then combine them.

```
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO
record = SeqIO.read("NC_005816.gb", "genbank")
```

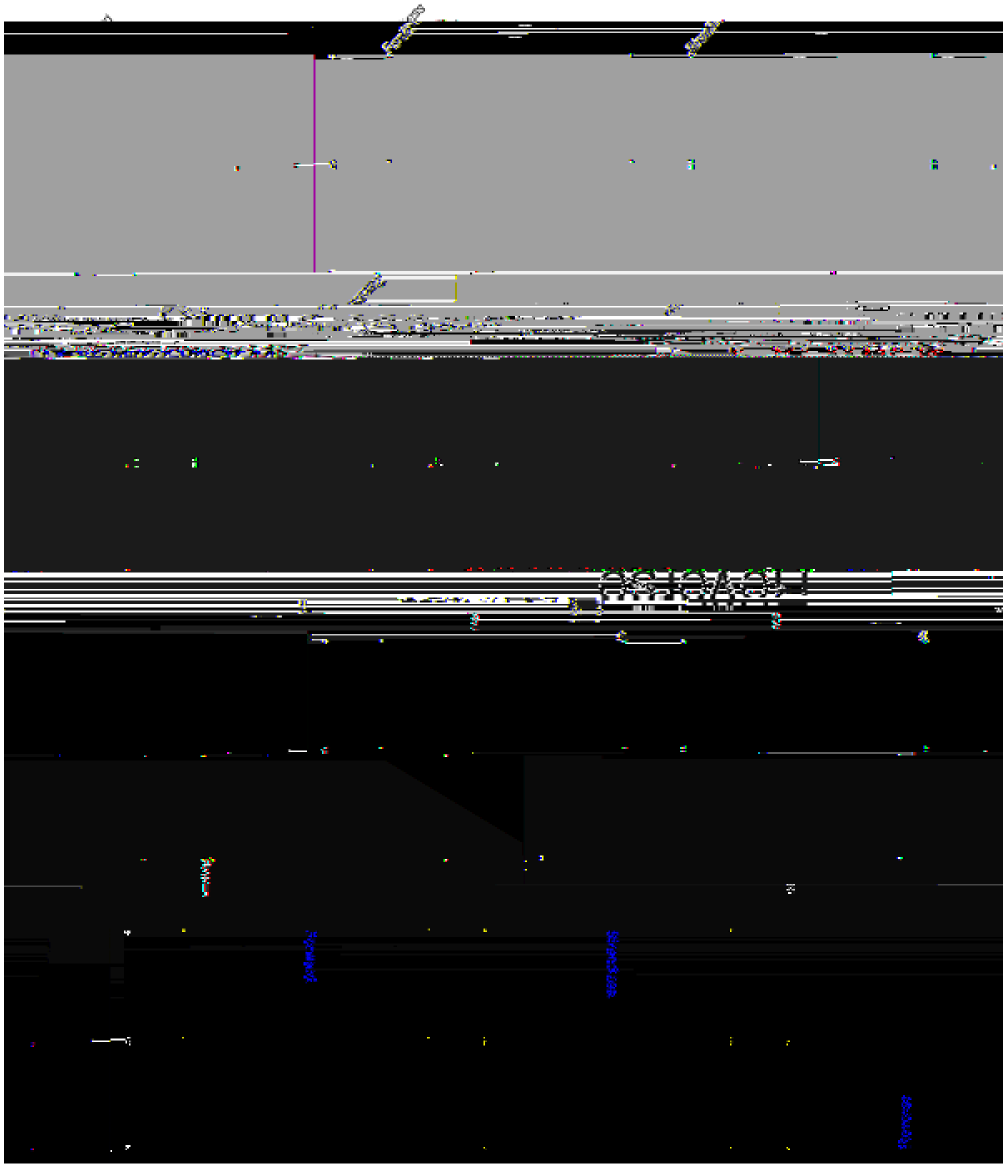
Figure 17.3: Simple GenomeDiagram showing label options. The top plot in pale green shows the default label settings (see Section 17.1.5) while the rest show variations in the label size, position and orientation (see Section 17.1.6).

17.1.7 Feature sigils

Figure 17.4: Simple GenomeDiagram showing di erent sigils (see Section 17.1.7)

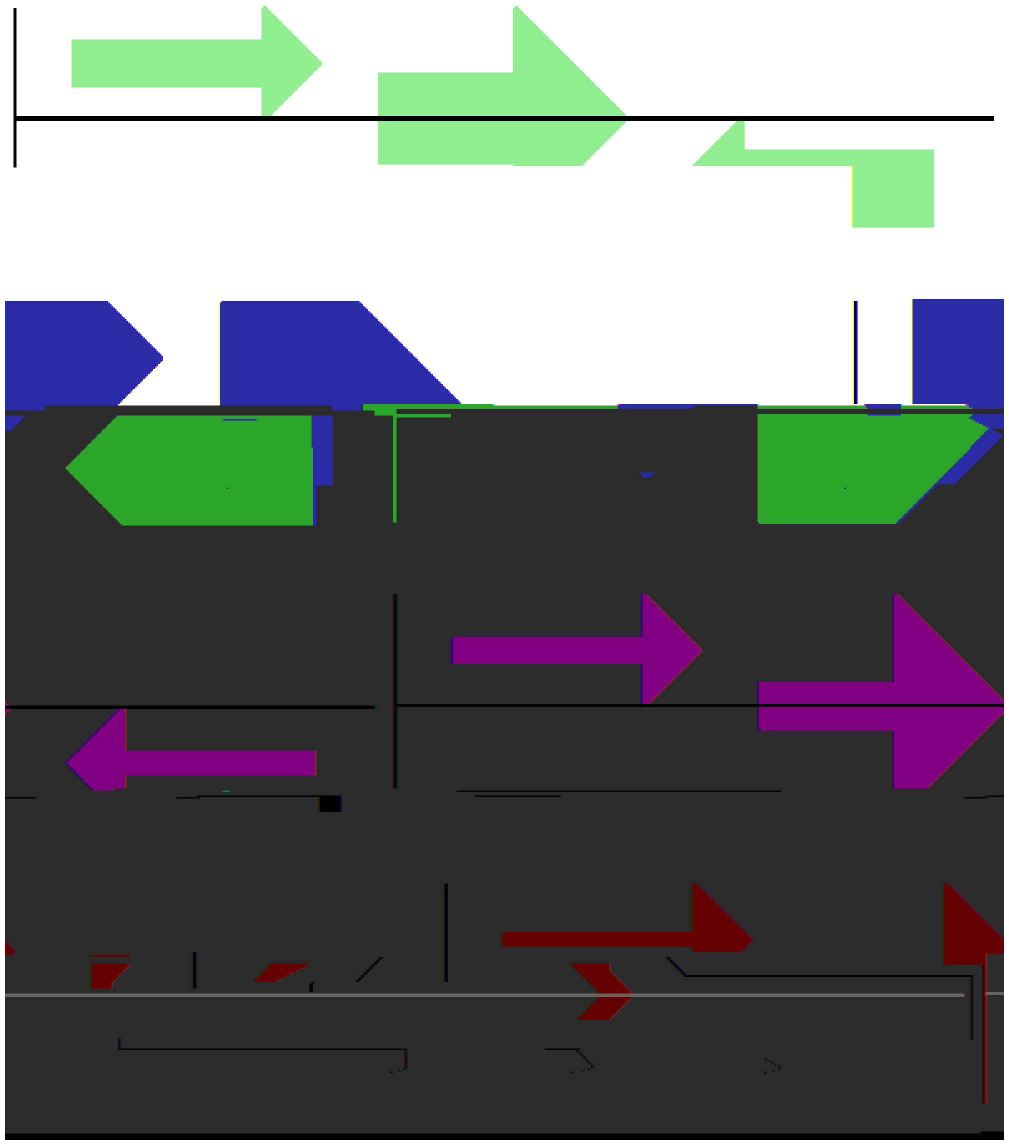

Figure 17.5: Simple GenomeDiagram showing arrow shaft options (see Section 17.1.8)

```
gd_feature_set.add_feature(feature, sigil="BIGARROW")
```

All the shaft and arrow head options shown above for the sARROW.Tf251.4470Td[=.13470Td[(sigir)-300cawne for tp

```
                start=0, end=len(record))
gd_diagram.write("plasmid_linear_nice.pdf", "PDF")
gd_diagram.write("plasmid_linear_nice.eps", "EPS")
gd_diagram.write("plasmid_linear_nice.svg", "SVG")

gd_diagram.draw(format="circular", circular=True, pagesize=(20*cm,20*cm),
                start=0, end=len(record), circle=nr7.7e=nr7.0.5rd))
gd_diagram.write("pl cilinear_nice.pdf", "PDF")
gd_diagram.write("pl cilinear_nice.eps", "EPS")
```

Figure 17.8: Circular diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1 showing selected restriction digest sites (see Section

You can download these using Entrez if you like, see Section

```
        i+=1

gd_diagram.draw(format="linear", pagesize='A4', fragments=1,
                start=0, end=max_len)
gd_diagram.write(name + ".pdf", "PDF")
gd_diagram.write(name + ".eps", "EPS")
gd_diagram.write(name + ".svg", "SVG")
```

The expected output is shown in Figure 17.9. I did wonder why in the original manuscript there were no

Figure 17.9: Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from6-baa)8cua")

Continuing the example from the previous section inspired by Figure 6 from Prouxet al.2002 [5], we

```
    (30, "orf53", "lin2567"),
    (28, "orf54", "lin2566"),
]
```

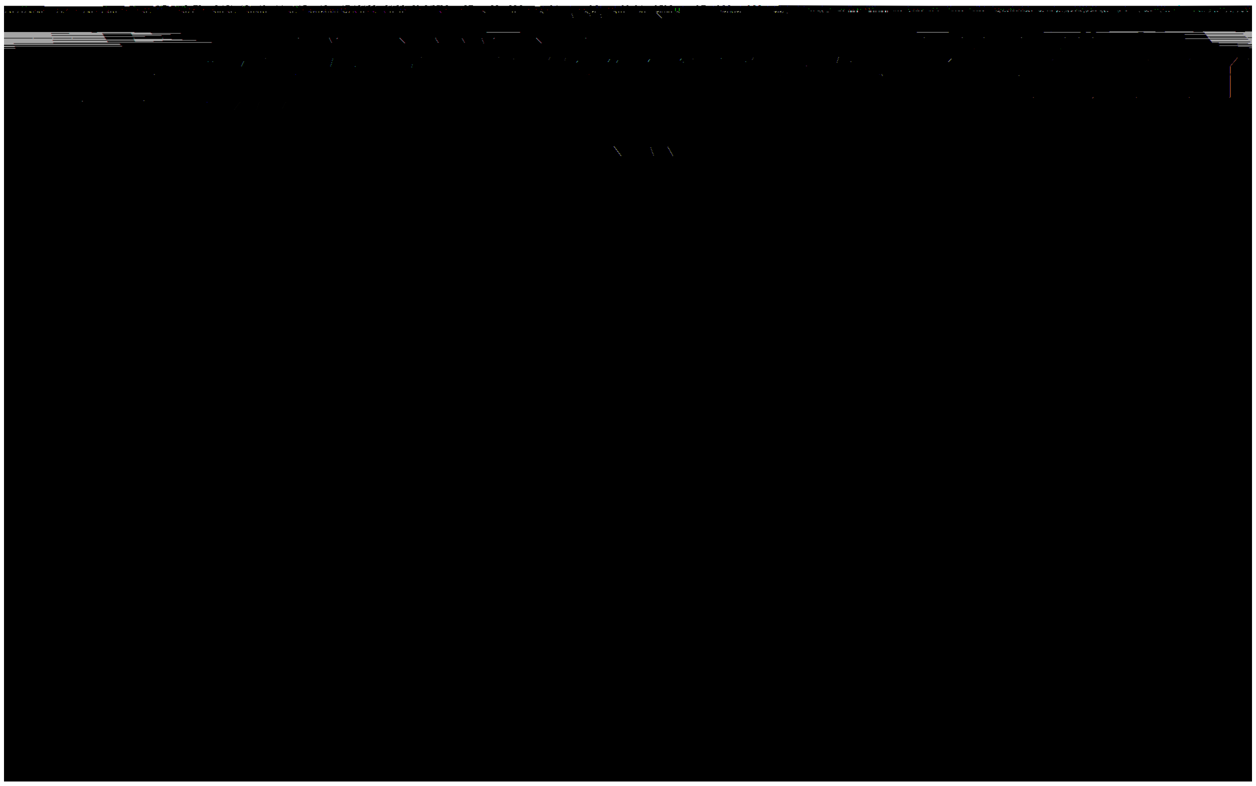

Figure 17.10: Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC_003212) plus basic cross-links shaded by percentage identity (see Section 17.1.11).

is to allocate space for empty tracks. Furthermore, in cases like this where there are no large gene overlaps, we can use the axis-straddling `BIGARROW`

These options are not covered here yet, so for now we refer you to the User Guide (PDF) included with

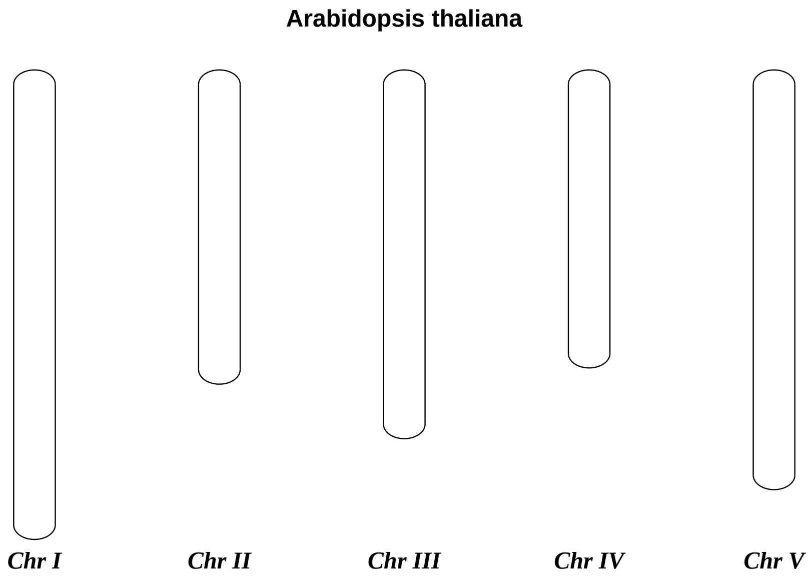

Figure 17.12: Simple chromosome diagram for *Arabidopsis thaliana*.

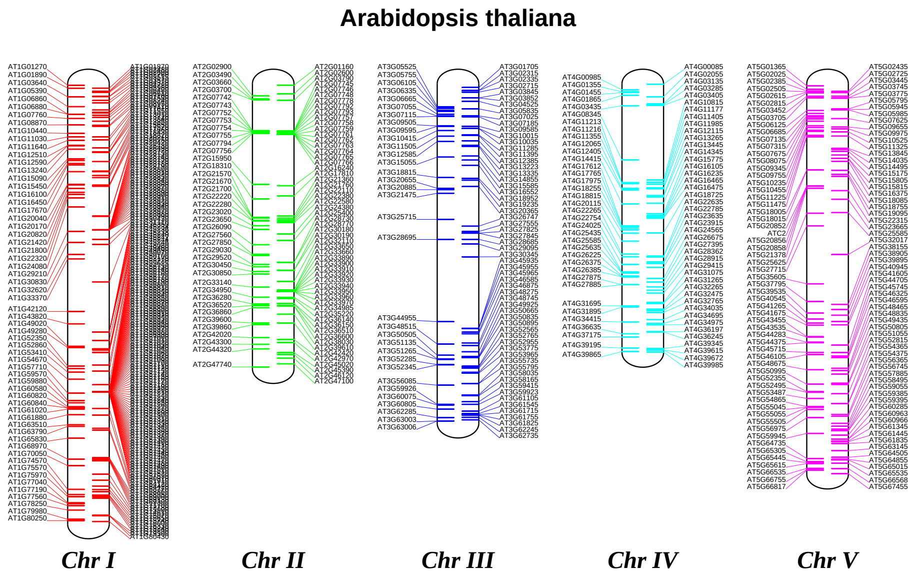

Figure 17.13: Chromosome diagram for *Arabidopsis thaliana* showing tRNA genes.

```
from Bio import SeqIO
entries = [("Chr I", "CHR_I/NC_003070.fna"),
           ("Chr II", "CHR_II/NC_003071.fna"),
           ("Chr III", "CHR_III/NC_003074.fna"),
           ("Chr IV", "CHR_IV/NC_003075.fna"),
           ("Chr V", "CHR_V/NC_003076.fna")]
for (name, filename) in entries:
    record = SeqIO.read(filename,"fasta")
    print(name, len(record))
```

This gave the lengths of the ve chromosomes, which we'll now use in the following short demonstration of the `BasicChromosome` module:

```
from reportlab.lib.units import cm
from Bio.Graphics import BasicChromosome

entries = [("Chr I", 30432563),
```

Chapter 18

```
print("There are %d repair pathways and %d repair genes. The genes are:" % \
      (len(repair_pathways), len(repair_genes)))
print(", ".join(repair_genes))
```

```
>>> corrected = record.subtract_control(control="A01")
>>> record["A01"][63]
336.0
>>> corrected["A01"][63]
0.0
```

19.1.2.4 Parameters extraction

Those wells where metabolic activity is observed show a sigmoid behavior for the colorimetric data. To allow

```
if count < lenuantedn:
```

Now, in order to use `Bio.SeqIO`

20.1.5 Sorting a sequence le

Suppose you wanted to sort a sequence le by length (e.g. a set of contigs from an assembly), and you are working with a le format like FASTA or FASTQ which `Bio.SeqIO` can read, write (and index).

This pulled out only 14580 reads out of the 41892 present. A more sensible thing to do would be to quality

```
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

This takes longer, as this time the output le contains all 41892 reads. Again, we're used a generator

```
original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_adaptors(original_reads, "GATGACGGTGT")
```

Q

20.1.10 Converting FASTA and QUAL les into FASTQ les

FASTQ les hold *both* sequences and their quality strings. FASTA les hold *just* sequences, while QUAL les hold *just*

```
['SRR020192.38240', 'SRR020192.23181', 'SRR020192.40568', 'SRR020192.23186']
>>> fq_dict["SRR020192.23186"].seq
```

20.1.13 Identifying open reading frames

```
table = 11
min_pro_len = 100

def find_orfs_with_trans(seq, trans_table, min_protein_length):
    answer = []
    seq_len = len(seq)
```

before, so you can check this is doing the same thing. Here we have sorted them by location to make it easier to compare to the actual annotation in the GenBank le (as visualised in Section 17.1.9).

If however all you want to nd are the locations of the open reading frames, then it is a waste of time to translate every possible codon, including doing the reverse complement to search the reverse strand too. All you need to do is search for the possible stop codons (and their reverse complements). Using regular

Figure 20.1: Histogram of orchid sequence lengths.

20.2.2 Plot of sequence GC%

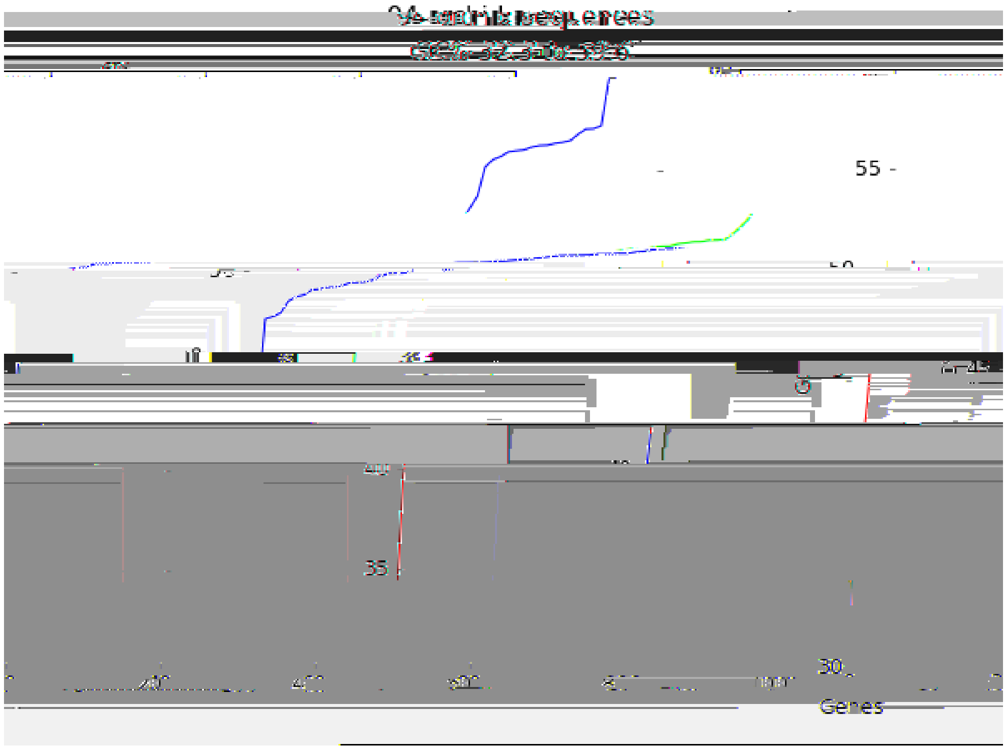

Figure 20.2: Histogram of orchid sequence lengths.

20.2.3 Nucleotide dot plots

A dot plot is a way of visually comparing two nucleotide sequences for similarity to each other. A sliding window is used to compare short sub-sequences to each other, often with a mis-match thr.8lder.Hepare fog 23s.

Dot plot using window size 7
200
400
600

```
dict_two = {}
for (seq, section_dict) in [(str(rec_one.seq).upper(), dict_one),
                            (str(rec_two.seq).upper(), dict_two)]:
    for i in range(len(seq)-window):
        section = seq[i:i+window]
        try:
            section_dict[section].append(i)
        except KeyError:
            section_dict[section] = [i]
#Now find any sub-sequences found in both sequences
#(Python 2.3 would require slightly different code here)
matches = set(dict_one).intersection(dict_two)
print("%i unique matches" % len(matches))
```

In order to use the `pylab.scatter()` we need separate lists for the *x* and *y* co-ordinates:

```
#Create lists of x and y co-ordinates for scatter plot
x = []
y = []
for section in matches:
    for i in dict_one[section]:
        for j in dict_two[section]:
            x.append(i)
            y.append(j)
```

We are now ready to draw the revised dot plot as a scatter plot:

```
import pylab
pylab.cla() #clear any prior graph
pylab.gray()
pylab.scatter(x,y)
pylab.xlim(0, len(rec_one)-window)
pylab.ylim(0, len(rec_two)-window)
pylab.xlabel("%s (length %i bp)" % (rec_one.id, len(rec_one)))
pylab.ylabel("%s (length %i bp)" % (rec_two.id, len(rec_two)))
pylab.title("Dot plot using window size %i\n(allowing no mis-matches)" % window)
pylab.show()
```

That should pop up a new window showing the graphthe.955 Td [(pylab.xlim(0,)-525(len(rec_one)-window))]TJ 0 -1#ur0 T

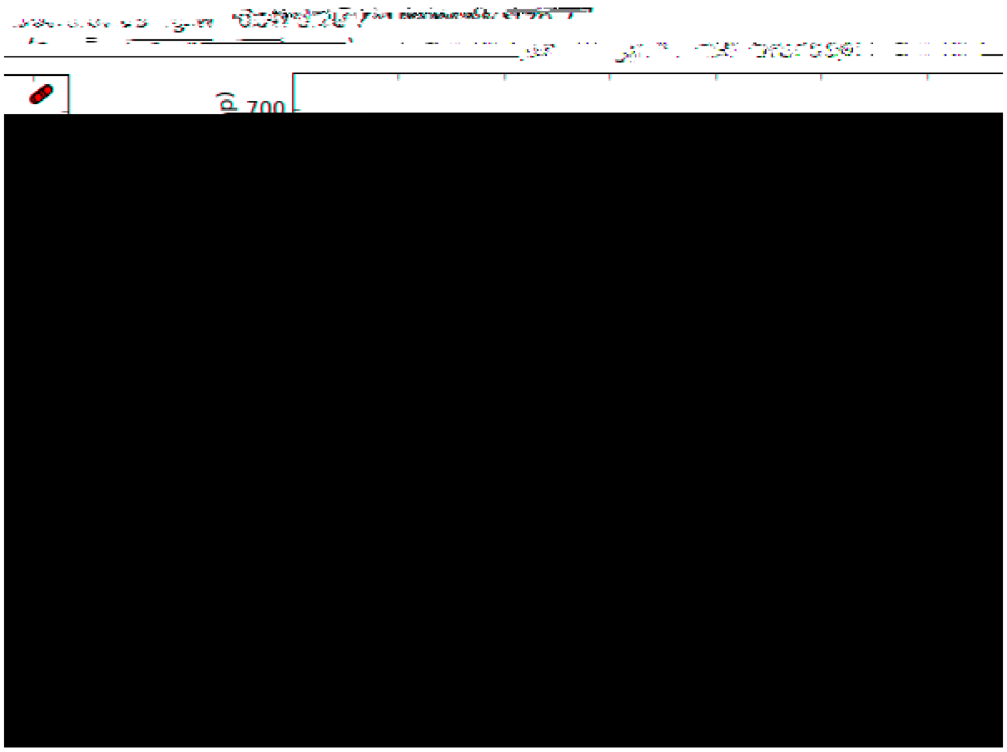

Figure 20.4: Nucleotide dot plot of two orchid sequence lengths (using pylab's scatter function).

```
import pylab
from Bio import SeqIO
for subfigure in [1,2]:
    filename = "SRR001666_%i.fastq" % subfigure
    pylab.subplot(1, 2, subfigure)
    for i,record in enumerate(SeqIO.parse(filename, "fastq")):
        if i >= 50 : break #trick!
        pylab.plot(record.letter_annotations["phred_quality"])
    pylab.ylim(0,45)
    pylab.ylabel("PHRED quality score")
    pylab.xlabel("Position")
pylab.savefig("SRR001666.png")
print("Done")
```

You should note that we are using the `Bio.SeqIO` format name `fastq` here because the NCBI has saved these reads using the standard Sanger FASTQ format with PHRED scores. However, as you might guess

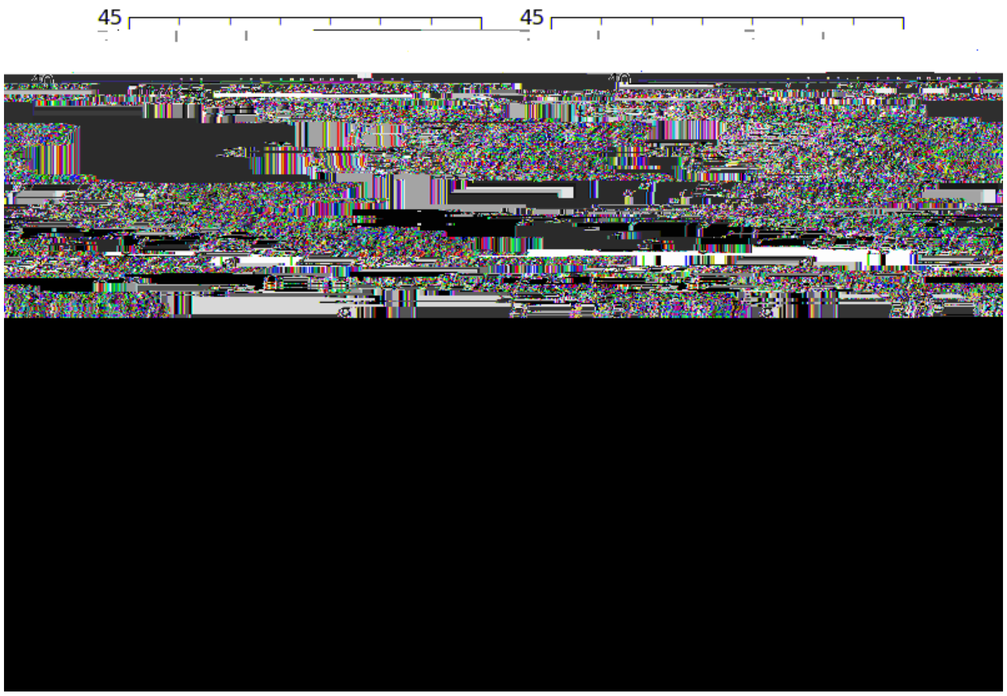

Figure 20.5: Quality plot for some paired end reads.

```
consensus = summary_align.dumb_consensus()
```

As the name suggests, this is a really simple consensus calculator, and will just add up all of the residues at each point in the consensus, and if the most common value is higher than some threshold value will add the common residue to the consensus. If it doesn't reach the threshold, it adds an ambiguity character to

By default nucleotide or amino acid residues with a frequency of 0 in a column are not take into account

Sections 6.4.1 and 20.3.1 contain more information on doing this.

20.5 BioSQL { storing sequences in a relational database

BioSQL is a joint e ort between the OBF projects (BioPerl, BioJava etc) to support a shared database

Chapter 21

The Biopython testing framework

Biopython has a regression testing framework (the le `run_tests.py`) based on

By default, `run_tests.py` runs all tests, including the docstring tests.

1. `test_Biospam.py`

(a) The long way:

online documentaion for unittest. If you are familiar with the unittest system (or something similar like the nose test framework), you shouldn't have any trouble. You may nd looking at the existing example within Biopython helpful too.

Here's a minimal unittest-style test script for Biospam, which you can copy and paste to get started:

```
import unittest
from Bio import Biospam

class BiospamTestAddition(unittest.TestCase):

    def test_addition1(self):
        result = Biospam.addition(2, 3)
        self.assertEqual(result, 5)

    def test_addition2(self):
        result = Biospam.addition(9, -1)
        self.assertEqual(result, 8)

class BiospamTestDivision(unittest.TestCase):

    def test_division1(self):
        result = Biospam.division(3.0, 2.0)
        self.assertAlmostEqual(result, 1.5)

    def test_division2(self):
        result = Biospam.division(10.0, -2.0)
        self.assertAlmostEqual(result, -5.0)


if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity = 2)
    unittest.main(testRunner=runner)
```

In the division tests, we use assertAlmostEqual instead of assertEqual to avoid tests failing due to roundo errors; see the unittestunittestchapter in the Python documentation for detailittest.Ter -254.Ifunc(tatali(-st)2(to)]T1 -34 to execute the tests when the script is run by itself (rather than imported from

```
$ python test_BiospamMyModule.py
An addition test ... ok
A second addition test ... ok
Now let's check division ... ok
A second division test ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```

If your module contains docstring tests (see section

Note that we regard doctests primarily as documentation, so you should stick to typical usage. Generally complicated examples dealing with error conditions and the like would be best left to a dedicated unit test.

Chapter 22

Advanced

22.1 Parser Design

Many of the older Biopython parsers were built around an event-oriented design that includes Scanner and

(a) `__init__(self,data=None,alphabet=None, mat_name='', build_later=0):`

   i. `data`: can be either a dictionary, or another SeqMat instance.
   ii. `alphabet`: a Bio.Alphabet instance. If not provided, construct an alphabet from data.
   iii. `mat_name`: matrix name, such as "BLOSUM62" or "PAM250"
   iv. `build_later`: default false. If true, user may supply only alphabet and empty dictionary, if intending to build the matrix later. this skips the sanity check of alphabet size vs. matrix size.

(b) `entropy(self,obs_freq_mat)`

   i. `obs_freq_mat`: an observed frequency matrix. Returns the matrix's entropy, based on the frequency in `obs_freq_mat`. The matrix instance should be LO or SUBS.

(c)

i. Full matrix size: N*N
ii. Half matrix size: N(N+1)/2

(a) `acc_rep_mat`: user provided accepted replacements matrix

(b) `exp_freq_table`: expected frequencies table. Used if provided, if not, generated from the `acc_rep_mat`.

(c) `logbase`

Chapter 23

Where to go from here { contributing to Biopython

23.1 Bug Reports + Feature Requests

Getting feedback on the Biopython modules is very important to us. Open-source projects like this bene t greatly from feedback, bug-reports (and patches!) from a wide variety of contributors.

23.7 Contributing Code

There are no barriers to joining Biopython code development other than an interest in creating biology-related code in Python. The best place to express an interest is on the Biopython mailing lists { just let us know you are interested in coding and what kind of stu you want to work on. Normally, we try to have

Chapter 24

Appendix: Useful stu   about Python

Bibliography

[1] Peter J. A. Cock, Tiago Antao, Je rey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo

[11] Douglas R. Cavener: \Comparison of the consensus sequence anking translational start sites in

[30] V. Matys, E. Fricke, R. Ge ers, E. Gössling, M. Haubrock, R. Hehl, K. Hornischer, D. Karas, A.E. Kel, O.V. Kel-Margoulis, D.U. Kloos, S. Land, B. Lewicki-Potapov, H. Michael, R. Münch, I. Reuter, S. Rotert, H. Saxel, M. Scheer, S. Thiele, E. Wingender E: \TRANSFAC: transcriptional regulation, from patterns to pro les." Nucleic Acids Research **31** (1): 374{378 (2003). https://doi.org/10.1093/nar/gkg108

[31] Robin Sibson: \SLINK: An optimally e cient algorithm for the single-link cluster method". *The Computer Journal* **16** (1): 30{34 (1973). https://doi.org/10.1093/comjnl/16.1.30

[32]