Automating Data Analysis with PERL

Lecture Note for Computational Biology 1 (LSM 5191)

Jiren Wang
http://www.bii.a-star.edu.sg/~jiren

BioInformatics Institute
Singapore
Outline

- Regular Expression and Pattern Matching
- Parsing BLAST Output Using PERL
Three Components in Bioinformatics

- The creation of databases allowing the storage and management of large biological data sets.
- The development of algorithms and statistics to determine relationships among members of large data sets.
- The use of these tools for the analysis and interpretation of various types of biological data.
Regular Expressions

Concepts about regular expression

- A regular expression is a pattern to be matched against a string.
- Matching a regular expression against a string either succeeds or fails.
- Regular expressions are used by many tools, such as grep, sed, awk, ed, vi, and emacs.
- Any regular expression that can be described in one of these tools can also be written in Perl.
Usage of Regular Expression in PERL

- Determine whether a string matches a particular pattern.
  `/abc/`

- Locate patterns within a string, and replace them with something else.
  `s/abc/xyz/`

- Specify not only where something is, but also where it isn’t.

```perl
=line = "/usr/bin/perl:/home/user:/tmp";
@fields = split /:/, $line;
# now @fields is ("/usr/bin/perl", "/home/user", "/tmp");
```
Patterns

Single-Character Patterns

- A single character matches itself.
  /a/
- Dot “.” matches any single character except newline (\n).
  /a./
- A pattern-matching character class is represented by a pair of open and close square brackets and a list of characters between the brackets. It matches any single character that is in the list.
  /[abcde]/
Patterns (cont’d)

- Ranges of characters separated by a dash (-).
  [0-9]
- A negated character class is represented by a leading up-arrow immediately after the left bracket. It matches any single character that is not in the list.
  [^0-9]
# Predefined Character Class Abbreviations

<table>
<thead>
<tr>
<th>Construct</th>
<th>Equivalent</th>
<th>Negated Construct</th>
<th>Equivalent Negated Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d (a digit)</td>
<td>[0-9]</td>
<td>\D</td>
<td>[^0-9]</td>
</tr>
<tr>
<td>\w (word char)</td>
<td>[a-zA-Z0-9_]</td>
<td>\W</td>
<td>[^a-zA-Z0-9_]</td>
</tr>
<tr>
<td>\s (space char)</td>
<td>[ \r\t\n\f]</td>
<td>\S</td>
<td>[^ \r\t\n\f]</td>
</tr>
</tbody>
</table>
Grouping Pattern

◆ Sequence
  ◼ /abc/ - matches an “a” followed by a “b” followed by a “c”.

◆ Quantifiers
  ◼ * - zero or more of the immediately previous character (or character class).
  ◼ + - one or more of the immediately previous character.
  ◼ ? – zero or one of the immediately previous character.
  ◼ General quantifier – a pair of matching curly braces with one or two numbers inside. 
    /x{5, 10}/.
Making Quantifiers March Minimally with “?”

- The quantifiers “*”, “+”, “{}”, and “?” are greedy (or maximal) by default.
- If you want a minimal match that will match as few items as possible, you get that by follow each of quantifiers with a “?”.

```
$alignment =~ /(.*?)(^ Score =.*)/ms;
```
Anchoring Patterns

- The caret (^) anchors the pattern to the beginning of the string.
  
  /^ab/ - matching “absence”, not “cable”.

- The $ anchors the pattern to the end of the string.
  
  /ab$/ - matching “cab”, not “cabbage”.

Pattern Binding Operators

Pattern binding operators =~ and !~

- Pattern binding operator =~ is used for pattern matching, substitution, and translation on strings.
- The argument on the right is the search pattern, substitution, or translation.
- The left argument is what is supposed to be searched, substituted, or translated.
- Pattern binding operator !~ is just like =~ except the return value is negated in the logical sense.
Examples of Operator =~

```perl
$a = "hello world";
$a =~ /^he/;  # true

if ($a =~ /^he/) {  # true
    ...
    # some stuff
}

$string !~ /pattern/
not $string =~ /pattern/
```
**Substitution Operator**

- \$RNA \sim s/T/U/gi;

This operator searches a string for PATTERN, and if found, replaces that match with the REPLACEMENT text and returns the number of substitutes made, which can be more than one with the /g modifier. Otherwise it returns false (0).

- **binding operator**
- **substitute operator**
- **regular expression to be replaced**
- **text to replace the pattern**
- **pattern modifier**
  - g – globally, throughout the string
  - i – case insensitive
  - s – single line (let . match newline)
  - m – multiline (let ^ and $ match before and after embedded newline)

**delimiters to separate the parts of the operator**
Strip Leading Spaces and Trailing Spaces from a String

- Strip leading spaces from a string
  - $string = ~ s/^\s+// ;$

- Strip trailing spaces from a string
  - $string = ~ s/\s+$// ;
The split and join Functions

The split function
- The split function takes a regular expression and a string, and looks for all occurrences of the regular expression within that string. The parts of the string that don’t match the regular expression are returned in sequence as a list of value.

```perl
$line = "/usr/bin/perl:/home/user:/tmp";
@fields = split /:/, $line;
# now @fields is ("/usr/bin/perl", "/home/user", "/tmp");
```

The join function
- The join function takes a list of values and glues them together with a glue string between each list element.

```perl
@fields = ("abc", "def", "xyz");
$outline = join(":", @fields);
# now $outline = "abc:def:xyz";
```
The substr Function

The substr function extracts a substring out of the string.
- Three arguments: a string value, a start position, and a length.

```perl
#!/usr/bin/perl -w
$sequence = 'MVLERILLQTIKFDLQVEHPYQFLKYAKQLK' .
  'GDKNKIQKLVQMAWTFVNDLCTTLSLQWEPE' .
  'IAVAVMYLARLCKFEIQEWTSKP';
$start = 8;
$length = 10;
$subsequence = substr($sequence, $start, $length);
print "The subsequence is: $subsequence\n\n";
# The subsequence is: QTIKFDLQVE
```
The length Function

The length function returns the length in bytes of the scalar value.

```perl
#!/usr/bin/perl -w
$sequence = 'MVLERILLQTIKFDLQVEHPYQFLLKYAKQLK' .
            'GDKNKIQKLVQMAWTFVNDSLCTTLSLQWEPE' .
            'IIAVAVMYLAGRLCKFEIQEWTSDKP';
$length = length($sequence);
print "The length of sequence $sequence is $length.\n";
```
Example of Pattern Matching

#!/usr/bin/perl
# check for non-DNA characters
use strict;
my $string = "CAACCCATGCGGGGagcttgca*caTGCT*agcttt";
if ($string =~ /([^AGCT])/i) {
    print "Warning! Found $1 in the string.\n";
}
# transcribe DNA into RNA
my $DNA = "ACGGGAATGCAGGATCATCCGACTGGATAA";
print "DNA = $DNA\n";
my $RNA = $DNA;
$RNA =~ s/T/U/g;
print "RNA = $RNA\n";
# now $RNA = “ACGGGAAUGCAGGAUAUCUCCGACUGGATAA”
Reversing Complement of Sequence

#!/usr/bin/perl –w

# Calculating the reverse complement of a strand of DNA
$DNA = “ACGGGAGCGGAAAATTACGGCATTAGC”;
$revcom = reverse $DNA;
# $revcom =~ s/A/T/g;
# $revcom =~ s/T/A/g;
# $revcom =~ s/G/C/g;
# $revcom =~ s/C/G/g;
$revcom =~ tr/ACGTacgt/TGCAtgca/;
print “$revcom
”;
Sequence Motif

- **Motifs** are short segments of DNA or protein that are of particular interest and associated with a function.
- Looking for motifs is one of the most common things we do in bioinformatics.
- Motifs may be regulatory elements of DNA or short stretches of protein that are known to conserved across many species.
- Motifs can often be represented as regular expressions.
  
  - zinc finger c2h2
  
  ```
  C-X{2,4}-C-X{3}-[ILV MFYW C]-X{8}-H-X{3,5}-H
  ```
Example of Finding Motif

#!/usr/bin/perl –w
print “Enter the sequence file name:”;
$filename = <STDIN>;
chomp $filename;
open(IN, “$filename”) || die “Cannot open file.”;
@seqs = <IN>;
close IN;
$seq = join(“”,@seqs);
$seq =~ s/\s//g;
print “Enter the motif:”; 
$motif = <STDIN>;
chomp $motif;
if ($seq =~ /$motif/) {
    print “The motif $motif has been found.\n”;
} else {
    print “The motif $motif hasn’t been found.\n”;
}
Some Predefined Read-only Variables

- $&$
  - the part of the string that matched the regular expression.

- $`$
  - the part of the string before the part that matched.

- `$’$
  - the part of the string after the part that matched.
Lookahead Assertion

(?=…) and (?!…)  

- The positive (?=…) and negative (?!…) lookahead assertions are zero-width themselves, but assert that the regular expression represented above by ... would (or would not) match at this point, were we to attempt it.

\(^{(?!>)}.*\n\)+

- the beginning of the line

(?!) – a “>” doesn’t follow.

.* - all non-newline characters.

+ - match all the available lines.
PERL Subroutine

◆ Subroutine definition
- Subroutines are defined by the keyword `sub`, followed by the name of the subroutine, followed by a block enclosed by curly braces `{}` containing the body of the subroutine.

```perl
sub say_hello {
    print "hello, world!\n";
}
```

◆ Invoking a subroutine
- By using the name of the subroutine followed by a parenthesized list of arguments.

```perl
say_hello();
&say_hello();
```
In Perl, the subroutine invocation is followed by a list within parentheses, causing the list to be automatically assigned to a special variable named @_ for the duration of the subroutine.

The subroutine can access this variable to determine the number of arguments and the value of those arguments.
Pass data to subroutine by reference

- Perl passes all arguments into the subroutine as a single array, the special @_ array.
- If you have more complicated arguments, say a mixture of scalars, arrays, and hashes, Perl often cannot distinguish between them.
- Using pass by reference, you can pass a subroutine any collection of scalars, arrays, hashes, and more, and the subroutine can distinguish between them.
In Perl, references behave somewhat as pointers do in other language. When a scalar is holding a reference, it always behaves as a simple scalar. Perl references play a central role in object-oriented programming.
The Backslash Operator

You can create a reference to any named variable by using the backslash operator.

Reference examples

```perl
$scalarref = \$foo;
$arrayref = \@ARGV;
$hashref = \%ENV;
```

Dereference examples

```perl
$$scalarref
@$arrayref
@$hashref
```
Example of Subroutine

```perl
#!/usr/bin/perl –w
use strict;

my $dna = "ACCGACGTCTTGAGGCTGGTGCTGTATC";
print "No. of T in $dna = ", &count_T($dna), "\n";
# output: No. of T in ACCGACGTCTTGAGGCTGGTGCTGTATC = 8
exit;

sub count_T {
    my ($dna) = @_;
    my $count = 0;
    for (my $pos = 0; $pos < length($dna); $pos++) {
        if (substr($dna, $pos, 1) eq 'T') {
            $count++;
        }
    }
    return $count;
}
```
BLAST

Basic Local Alignment Search Tool

- One of the most popular software tools in biological research.
- A collection of programs with versions for query-to-database pairs.
- It tests a query sequence against a database of known sequences in order to find similarity.
Obtaining BLAST

NCBI’s BLAST

Introduction to BLAST Query

Steps

- Choose the program to use and the database to search.
- Input the sequence data.
- Set the program options or choose defaults.
- Set the output formatting options.
- Perform the search.

Sequence for BLAST Query

TCTTTGCTGCTCCTCTCTTACTAAAGCAGACACCCACACCCCTTTCTCCTCTGTTTGTGGTTGCTCTGCCTGCTTCTATCTTA
ACTAAACTGTTTTTTCGTTGCTCCTCCACCTTGGGTGCTCTCTAACAATAAACTTTGTATTTGCTATTCAAGGTTTT
TGCTCTCATGACAAATTCATATTTTCATCGGAGGCAAGGATCCAGCAAGGTTCAGCACTCCCACTGTTGCTGTGTCTGTA
GATGCTAGGATCTCCTGTTTCCATCCAGGCTCCCGCGGAGGACAAGAGTTCTCACTGGGAGCAAAGATCCAGGGAAAAAT
TGGCTTCTAGAGTCTACCCTTTGCTGGCAGATGGCTAGGATTCCTGGTTCTCATCCAGGCTCCCCAGGTTCAAGTCCTGGG
AGAGAATTAAGACCTCGCTTCCTGCTACTGCTGCCTCGTTAAATTCATAACCACGTGAGCTCTGTTCTCTCCTTTTTA
AATGCTCTTTTCAAACCACTGAGGCTTGAGACATTAACAGCAAGCCTACATCTGAGGATTTACATTTTGAGTATATCCCA
AGGACAGCTATGAAGAGTTTCACATTTTATTAATTTTCCACCAACTCCGTGACACTGAAACCCAGGTGAGTTTCGCAGTA
AAGCCAAAGCCTTTTT
BLAST Output

[Header of the page]

BLASTN 2.2.3 [Apr-24-2002]


RID: 1027934170-05931-26815
Query=
(896 letters)

Database: All GenBank+EMBL+DDBJ+PDB sequences (but no EST, STS, GSS, or phase 0, 1 or 2 HTGS sequences)
1,354,612 sequences; 6,180,674,170 total letters

If you have any problems or questions with the results of this search please refer to the BLAST FAQs

Taxonomy reports

Distribution of 56 Blast Hits on the Query Sequence
### BLAST Output (cont’d)

<table>
<thead>
<tr>
<th>Sequence ID</th>
<th>Description</th>
<th>Score</th>
<th>E Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>gi</td>
<td>29</td>
<td>emb</td>
<td>X60495.1</td>
</tr>
<tr>
<td>gi</td>
<td>488602</td>
<td>emb</td>
<td>Z33621.1</td>
</tr>
<tr>
<td>gi</td>
<td>163734</td>
<td>gb</td>
<td>M22244.1</td>
</tr>
<tr>
<td>gi</td>
<td>6138951</td>
<td>gb</td>
<td>AF055981.1</td>
</tr>
<tr>
<td>gi</td>
<td>4335923</td>
<td>gb</td>
<td>AF057133.1</td>
</tr>
<tr>
<td>gi</td>
<td>3599988</td>
<td>gb</td>
<td>AF047026.1</td>
</tr>
<tr>
<td>gi</td>
<td>15138708</td>
<td>gb</td>
<td>AC087180.2</td>
</tr>
<tr>
<td>gi</td>
<td>14572134</td>
<td>gb</td>
<td>AC087160.2</td>
</tr>
<tr>
<td>gi</td>
<td>21903626</td>
<td>gb</td>
<td>AC096629.9</td>
</tr>
<tr>
<td>gi</td>
<td>14423594</td>
<td>gb</td>
<td>AC003059.16</td>
</tr>
<tr>
<td>gi</td>
<td>14423593</td>
<td>gb</td>
<td>AC006542.13</td>
</tr>
<tr>
<td>gi</td>
<td>3047016</td>
<td>gb</td>
<td>AF056973.1</td>
</tr>
<tr>
<td>gi</td>
<td>21903625</td>
<td>gb</td>
<td>AC098686.10</td>
</tr>
<tr>
<td>gi</td>
<td>17386311</td>
<td>gb</td>
<td>AC098687.2</td>
</tr>
<tr>
<td>gi</td>
<td>17298593</td>
<td>gb</td>
<td>AC020926.8</td>
</tr>
</tbody>
</table>

...
Alignments

>gi|29|emb|X60495.1|BT1SVSP  B.taurus exon 1 for bovine seminal vesicle secretory protein
SVSP109 signal peptide
Length = 896

Score = 1776 bits (896), Expect = 0.0
Identities = 896/896 (100%)
Strand = Plus / Plus

Query: 1 tctttgctgcctctcttcactaaagcagacacccacacccttttctcttgttgtgtgc 60
Sbjct: 1 tctttgctgcctctcttcactaaagcagacacccacacccttttctcttgttgtgtgc 60

Query: 61 tctgccttgcttctatcttaactaaactgtttttccgtgtgctctcccacttgtgtgct 120
Sbjct: 61 tctgccttgcttctatcttaactaaactgtttttccgtgtgctctcccacttgtgtgct 120

Query: 841 ccaactccgtgacactgaaacccaggtgagtttcgcagtaaagccaaagccttttt 896
Sbjct: 841 ccaactccgtgacactgaaacccaggtgagtttcgcagtaaagccaaagccttttt 896

>gi|488602|emb|Z33621.1|BTSVSP109  B.taurus SVSP109 gene
Length = 6956

Score = 1731 bits (873), Expect = 0.0
Identities = 894/897 (99%), Gaps = 3/897 (0%)
Strand = Plus / Plus
Query: 156 gtttttgcttcctgacaaa 175
Sbjct: 4543 gtttttgcttcctgacaaa 4562

Database: All GenBank+EMBL+DDBJ+PDB sequences (but no EST, STS, GSS, or phase 0, 1 or 2 HTGS sequences)
Posted date: Jul 29, 2002 12:56 AM
Number of letters in database: 1,885,706,874
Number of sequences in database: 1,354,612

Lambda      K      H
1.37    0.711     1.31

Gapped
Lambda      K      H
1.37    0.711     1.31

Matrix: blastn matrix:1 -3
Gap Penalties: Existence: 5, Extension: 2
Number of Hits to DB: 4,096,583
Number of Sequences: 1354612
Number of extensions: 4096583
Number of successful extensions: 26026
Number of sequences better than 10.0: 49
Length of query: 896
Length of database: 6,180,674,170
Effective HSP length: 21
Effective length of query: 875
Effective length of database: 6,152,227,318
Effective search space: 5383198903250
Effective search space used: 5383198903250
T: 0
A: 30
X1: 6 (11.9 bits)
X2: 15 (29.7 bits)
S1: 12 (24.3 bits)
S2: 20 (40.1 bits)
Three Parts of BLAST Output

- Some header information at the beginning.
- A summary of the alignments and the alignments in the middle.
- Some additional summary parameters and statistics at the end.
Some Concepts in BLAST

String matching
- Algorithms find one string embedded in another.
- Biological string matching looks for similarity as an indication of homology.
- Homology between sequences means the sequences are related evolutionarily.

The raw score
- A measure of similarity and the size of the match.

The E (expect) value
- The chance that string matching (allowing for gaps) occurs in a randomly generated database of the same size and composition.
Parsing BLAST Output Using PERL

Why parse BLAST output?
- Extracting annotation and alignments.
- Parsing BLAST Alignments.
#!/usr/bin/perl
#
# The following program and subroutines are adopted from
# "Beginning Perl for BioInformatics" and some modifications have
# been made.
# Extract annotation and alignments from BLAST output file

use strict;

# declare and initialize variables
my $beginning_annotation = '';
my $ending_annotation = '';
my %alignments = ( );
my $filename = 'blast.txt';

parse_blast($beginning_annotation, $ending_annotation, %alignments, $filename);

foreach my $key (keys %alignments) {
    print "$key
XXXXXXXXXXXX
", $alignments{$key}, "
XXXXXXXXXX
";
}

print $ending_annotation;
exit;
Subroutine get_file_data

sub get_file_data {
    my($filename) = @_; 
    # Initialize variables
    my @filedata = ( );
    unless( open(GET_FILE_DATA, $filename) ) {
        print "Cannot open file \"$filename\"\n\n";
        exit;
    }
    @filedata = <GET_FILE_DATA>;
    close GET_FILE_DATA;
    return @filedata;
}
Subroutine parse_blast

# parse_blast
# parse beginning and ending annotation, and alignments,
# from BLAST output file
sub parse_blast {
    my($beginning_annotation, $ending_annotation, $alignments, $filename) = @_; my $blast_output_file = ''; my $alignment_section = ''; # Get the BLAST program output into an array from a file $blast_output_file = join( '', get_file_data($filename)); # Extract the beginning annotation, alignments, and ending annotation ($$beginning_annotation, $alignment_section, $$ending_annotation) = ($blast_output_file =~ /(.*Alignments\n)(.*)(^Database:.*))/ms); # Populate %alignments hash # key = ID of hit # value = alignment section %$alignments = parse_blast_alignment($alignment_section); }
Subroutine parse_blast_alignment

# parse_blast_alignment
# parse the alignments from a BLAST output file, return hash with
# key = ID and value = text of alignment.
sub parse_blast_alignment {
    my($alignment_section) = @_;  # declare and initialize variables
    my(%alignment_hash) = ();
    # loop through the scalar containing the BLAST alignments,
    # extracting the ID and the alignment and storing in a hash
    # The regular expression matches a line beginning with >,
    # and containing the ID between the first pair of | characters;
    # followed by any number of lines that don't begin with >
    print "$alignment_section\n\n"
    while($alignment_section =~ /^>.*\n(^(?!>).*\n)+/gm) {
        my($value) = $&;
        my($key) = (split(/\|/, $value)) [1];
        $alignment_hash{$key} = $value;
    }
    return %alignment_hash;
}
Persisting BLAST Alignments

#!/usr/bin/perl
# Parse alignments from BLAST output file
use strict;
# declare and initialize variables
my $beginning_annotation = ''; my $ending_annotation = '';
my %alignments = ( ); my $alignment = '';
my $filename = 'blast.txt'; my @HSPs = ( );
my($expect, $query, $query_range, $subject, $subject_range) = ('','','','','');
parse_blast($beginning_annotation, $ending_annotation, %alignments, $filename);
$alignment = $alignments{'163734'};
@HSPs = parse_blast_alignment_HSP($alignment);
($expect, $query, $query_range, $subject, $subject_range) =
extract_HSP_information($HSPs[1]);
# Print the results
print "
-> Expect value:   $expect
";
print "
-> Query string:   $query
";
print "
-> Query range:    $query_range
";
print "
-> Subject String: $subject
";
print "
-> Subject range:  $subject_range
";
exit;
Subroutine parse_blast_alignment_HSP

# parse_blast_alignment_HSP
# parse beginning annotation, and HSPs, from BLAST alignment
# Return an array with first element set to the beginning annotation,
# and each successive element set to an HSP
sub parse_blast_alignment_HSP {
    my($alignment ) = @_;  
    # declare and initialize variables  
    my $beginning_annotation = ''; my $HSP_section = ''; my @HSPs = ();  
    # Extract the beginning annotation and HSPs  
    ($beginning_annotation, $HSP_section ) = ($alignment =~ /(.*?)(^ Score =.*)/ms);  
    # Store the $beginning_annotation as the first entry in @HSPs  
    push(@HSPs, $beginning_annotation);  
    # Parse the HSPs, store each HSP as an element in @HSPs  
    while($HSP_section =~ /(^ Score =.*
)(^(?! Score =).*
)+/gm) {  
        push(@HSPs, $&);  
    }  
    # Return an array with first element = the beginning annotation,  
    # and each successive element = an HSP  
    return(@HSPs);  
}
Subroutine extract_HSP_information

# extract_HSP_information
# - parse a HSP from a BLAST output alignment section
# - return array with elements: Expect value, Query string,
#   Query range, Subject string, Subject range
sub extract_HSP_information {
    my($HSP) = @_; # declare and initialize variables
    my($expect) = ''; my($query) = ''; my($query_range) = ''; my($subject) = ''; my($subject_range) = ''; ($expect) = ($HSP =~ /Expect = (\S+)/); $query = join ( '' , ($HSP =~ /^Query.*\n/gm) ); $subject = join ( '' , ($HSP =~ /^Sbjct(.*)\n/gm) ); $query_range = join('..', ($query =~ /\d+\D\d+/)); $subject_range = join('..', ($subject =~ /\d+\D\d+/)); $query =~ s/[^acgt]///g; $subject =~ s/[^acgt]///g; return ($expect, $query, $query_range, $subject, $subject_range);
}