`prim`: Designing and Testing PCR Primers

Bernhard Haubold

11 Jan 2024, v0.2-7

Contents

Chapter 1

Introduction

The package `prim` consists of four programs for designing and scoring diagnostic PCR primers. Table 1.1 lists the programs, of which the first two, `fa2prim` and `prim2tab` support the design of primers with `primer3` [1], while the second two, `scop` and `cops`, are used calculate the sensitivity and specificity of primers using *in silico* PCR. The program `scop` is based on the NCBI taxonomy, while `cops` corrects the output of `scop` using evolutionary distances.

Table 1.1: The four programs of the package `prim`.

Program	Description
`fa2prim`	Convert template sequence to `primer3` input
`prim2tab`	Convert `primer3` output to table
`scop`	Score primers by sensitivity and specificity
`cops`	Correct primer scores

Chapter 2

`fa2prim`: Convert Template Sequences to `primer3` Input

Introduction

The program `fa2prim` automates the conversion of FASTA templates to `primer3` input. Figure 2.1 shows an abridged example of output by `fa2prim`.

Implementation

The outline of `fa2prim` has hooks for imports, types, functions, and the logic of the main function.

Prog. 1 (`fa2prim`)

3 ⟨*fa2prim.go* 3⟩≡

```
package main

import (
        ⟨Imports, Pr. 1 4b⟩
)

⟨Types, Pr. 1 6b⟩
⟨Functions, Pr. 1 6e⟩
func main() {
        ⟨Main function, Pr. 1 4a⟩
}
```

```
PRIMER_TASK=generic
PRIMER_PICK_LEFT_PRIMER=1
PRIMER_PICK_RIGHT_PRIMER=1
PRIMER_PICK_INTERNAL_OLIGO=1
PRIMER_MIN_SIZE=15
PRIMER_MAX_SIZE=25
PRIMER_PRODUCT_SIZE_RANGE=70-150
PRIMER_MIN_TM=54.0
PRIMER_OPT_TM=56.0
PRIMER_MAX_TM=58.0
PRIMER_INTERNAL_MIN_TM=43.0
PRIMER_INTERNAL_OPT_TM=45.0
PRIMER_INTERNAL_MAX_TM=47.0
SEQUENCE_TEMPLATE=CGGGAGAGTAAGGAAGGCCGTGGGTGCA...
=
```

Figure 2.1: Output by `fa2prim`, which then serves as input to `primer3`.

In the main function we set the name of `fa2prim`, set the usage, declare and parse the options, and parse the input.

4a ⟨*Main function, Pr. 1* 4a⟩≡ (3)
```
util.SetName("fa2prim")
⟨Set usage, Pr. 1 4c⟩
⟨Declare options, Pr. 1 4e⟩
⟨Parse options, Pr. 1 6a⟩
⟨Parse input, Pr. 1 6d⟩
```

We import `util`.

4b ⟨*Imports, Pr. 1* 4b⟩≡ (3) 4d▷
```
"github.com/evolbioinf/prim/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `fa2prim`, and an example command.

4c ⟨*Set usage, Pr. 1* 4c⟩≡ (4a)
```
u := "fa2prim [option]... [template.fasta]..."
p := "Convert FASTA sequences to primer3 input."
e := "fa2prim foo.fasta | primer3_core"
clio.Usage(u, p, e)
```

We import `clio`.

4d ⟨*Imports, Pr. 1* 4b⟩+≡ (3) ◁4b 4f▷
```
"github.com/evolbioinf/clio"
```

We declare the obligatory version option and options to drive `primer3`.

4e ⟨*Declare options, Pr. 1* 4e⟩≡ (4a)
```
optV := flag.Bool("v", false, "version")
⟨Options for primer3, Pr. 1 5a⟩
```

We import `flag`.

4f ⟨*Imports, Pr. 1* 4b⟩+≡ (3) ◁4d 7a▷
```
"flag"
```

Table 2.1: Options for driving `primer3` and their default values.

Option	Meaning	Default
`primMinSize`	minimum primer size	15
`primMaxSize`	maximum primer size	25
`prodMinSize`	minimum product size	70
`prodMaxSize`	maximum product size	150
`primMinTm`	minimum primer melting temperature	54
`primOptTm`	optimal primer melting temperature	56
`primMaxTm`	maximal primer melting temperature	58
`inMinTm`	internal oligo minimum melting temperature	43
`inOptTm`	internal oligo optimal melting temperature	45
`inMaxTm`	internal oligo maximum melting temperature	47

We declare the 10 options for `primer3` listed in Table 2.1 together with their default values. We begin with the four options to specify the sizes of the primers and the internal oligo.

5a ⟨*Options for* `primer3`*, Pr. 1* 5a⟩≡ (4e) 5b▷

```
primMinSize := flag.Int("primMinSize", 15,
          "minimum primer size")
primMaxSize := flag.Int("primMaxSize", 25,
          "maximum primer size")
prodMinSize := flag.Int("prodMinSize", 70,
          "minimum product size")
prodMaxSize := flag.Int("prodMaxSize", 150,
          "maximum product size")
```

We continue with the six options to specify the melting temperature of the primer and the internal oligo.

5b ⟨*Options for* `primer3`*, Pr. 1* 5a⟩+≡ (4e) ◁5a

```
primMinTm := flag.Float64("primMinTm", 54,
          "minimum primer T_m")
primOptTm := flag.Float64("primOptTm", 56,
          "optimal primer T_m")
primMaxTm := flag.Float64("primMaxTm", 58,
          "maximum primer T_m")
inMinTm := flag.Float64("inMinTm", 43,
          "minimum internal oligo T_m")
inOptTm := flag.Float64("inOptTm", 45,
          "optimal internal oligo T_m")
inMaxTm := flag.Float64("inMaxTm", 47,
          "maximum internal oligo T_m")
```

We parse the options and respond to a version request, as this stops `fa2prim`. We also store the options for the `primer3` run.

6a ⟨*Parse options, Pr. 1* 6a⟩≡ (4a)
```
flag.Parse()
if *optV {
        util.Version()
}
```
⟨*Store options for* `primer3`*, Pr. 1* 6c⟩

To store the parameters for `primer3`, we declare the `struct` `Parameters`.

6b ⟨*Types, Pr. 1* 6b⟩≡ (3)
```
type Parameters struct {
        primMinSize, primMaxSize,
        prodMinSize, prodMaxSize int
        primMinTm, primOptTm, primMaxTm,
        inMinTm, inOptTm, inMaxTm float64
}
```

We store the options.

6c ⟨*Store options for* `primer3`*, Pr. 1* 6c⟩≡ (6a)
```
pa := new(Parameters)
pa.primMinSize = *primMinSize
pa.primMaxSize = *primMaxSize
pa.prodMinSize = *prodMinSize
pa.prodMaxSize = *prodMaxSize
pa.primMinTm = *primMinTm
pa.primOptTm = *primOptTm
pa.primMaxTm = *primMaxTm
pa.inMinTm = *inMinTm
pa.inOptTm = *inOptTm
pa.inMaxTm = *inMaxTm
```

The remaining tokens in the command line are taken as file names. We parse these files using `ParseFiles`, which applies the function `parse` to each one. Parse, in turn, takes the parameters for `primer3` as parameter.

6d ⟨*Parse input, Pr. 1* 6d⟩≡ (4a)
```
files := flag.Args()
clio.ParseFiles(files, parse, pa)
```

Inside `parse`, we retrieve the parameter container. Then we iterate over the sequences in the current input file. For each file we print a set of `primer3` instructions.

6e ⟨*Functions, Pr. 1* 6e⟩≡ (3)
```
func parse(r io.Reader, args ...interface{}) {
        p := args[0].(*Parameters)
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                s := string(sc.Sequence().Data())
                ⟨Print primer3 input, Pr. 1 7b⟩
        }
}
```

We import `io` and `fasta`.

7a ⟨*Imports, Pr. 1* 4b⟩+≡ (3) ◁4f 7c▷
```
"io"
"github.com/evolbioinf/fasta"
```

For a given template, the input to `primer3` consists of a constant and a variable part and is terminated by `=`.

7b ⟨*Print* `primer3` *input, Pr. 1* 7b⟩≡ (6e)
```
⟨Print constant primer3 input, Pr. 1 7d⟩
⟨Print variable primer3 input, Pr. 1 7e⟩
fmt.Println("=")
```

We import `fmt`.

7c ⟨*Imports, Pr. 1* 4b⟩+≡ (3) ◁7a
```
"fmt"
```

In the constant part of the instruction block we ask for pairs of primers, each augmented by an internal oligo.

7d ⟨*Print constant* `primer3` *input, Pr. 1* 7d⟩≡ (7b)
```
fmt.Println("PRIMER_TASK=generic")
fmt.Println("PRIMER_PICK_LEFT_PRIMER=1")
fmt.Println("PRIMER_PICK_RIGHT_PRIMER=1")
fmt.Println("PRIMER_PICK_INTERNAL_OLIGO=1")
```

In the variable part of the instruction block we set the lengths of the primer and the product, the melting temperatures of the primers and the internal oligo, and the template sequence.

7e ⟨*Print variable* `primer3` *input, Pr. 1* 7e⟩≡ (7b)
```
fmt.Printf("PRIMER_MIN_SIZE=%d\n", p.primMinSize)
fmt.Printf("PRIMER_MAX_SIZE=%d\n", p.primMaxSize)
fmt.Printf("PRIMER_PRODUCT_SIZE_RANGE=%d-%d\n",
          p.prodMinSize, p.prodMaxSize)
fmt.Printf("PRIMER_MIN_TM=%.1f\n", p.primMinTm)
fmt.Printf("PRIMER_OPT_TM=%.1f\n", p.primOptTm)
fmt.Printf("PRIMER_MAX_TM=%.1f\n", p.primMaxTm)
fmt.Printf("PRIMER_INTERNAL_MIN_TM=%.1f\n", p.inMinTm)
fmt.Printf("PRIMER_INTERNAL_OPT_TM=%.1f\n", p.inOptTm)
fmt.Printf("PRIMER_INTERNAL_MAX_TM=%.1f\n", p.inMaxTm)
fmt.Printf("SEQUENCE_TEMPLATE=%s\n", s)
```

We've finished `fa2prim`, time to test it.

Testing

Our program for testing `fa2prim` has hooks for imports and the testing logic.

8a ⟨*fa2prim_test.go* 8a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Pr. 1 8c⟩
)

func TestFa2prim(t *testing.T) {
        ⟨Testing, Pr. 1 8b⟩
}
```

We construct a set of tests and run them.

8b ⟨*Testing, Pr. 1* 8b⟩≡ (8a)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 1 8d⟩
for i, test := range tests {
        ⟨Run test, Pr. 1 9d⟩
}
```

We import `exec`.

8c ⟨*Testing imports, Pr. 1* 8c⟩≡ (8a) 9e▷
```
"os/exec"
```

All our tests take as input the random DNA sequence `test.fasta`. Our first test analyzes this using only default options.

8d ⟨*Construct tests, Pr. 1* 8d⟩≡ (8b) 8e▷
```
f := "./test.fasta"
test := exec.Command("./fa2prim", f)
tests = append(tests, test)
```

In the next three tests we vary the melting temperature of the internal oligo.

8e ⟨*Construct tests, Pr. 1* 8d⟩+≡ (8b) ◁8d 9a▷
```
test = exec.Command("./fa2prim", "-inMaxTm", "48", f)
tests = append(tests, test)
test = exec.Command("./fa2prim", "-inMinTm", "44", f)
tests = append(tests, test)
test = exec.Command("./fa2prim", "-inOptTm", "46", f)
tests = append(tests, test)
```

In the next three tests we vary the melting temperature of the primers.

9a ⟨*Construct tests, Pr. 1* 8d⟩+≡ (8b) ◁8e 9b▷
```
test = exec.Command("./fa2prim", "-primMaxTm", "59", f)
tests = append(tests, test)
test = exec.Command("./fa2prim", "-primMinTm", "55", f)
tests = append(tests, test)
test = exec.Command("./fa2prim", "-primOptTm", "57", f)
tests = append(tests, test)
```

In the next two tests we vary the primer length.

9b ⟨*Construct tests, Pr. 1* 8d⟩+≡ (8b) ◁9a 9c▷
```
test = exec.Command("./fa2prim", "-primMaxSize", "26", f)
tests = append(tests, test)
test = exec.Command("./fa2prim", "-primMinSize", "16", f)
tests = append(tests, test)
```

In our last two tests we vary the product size.

9c ⟨*Construct tests, Pr. 1* 8d⟩+≡ (8b) ◁9b
```
test = exec.Command("./fa2prim", "-prodMaxSize", "151", f)
tests = append(tests, test)
test = exec.Command("./fa2prim", "-prodMinSize", "71", f)
tests = append(tests, test)
```

We run a test and compare the result we get with the result we want, which is contained in files `r1.txt`, `r2.txt`, and so on.

9d ⟨*Run test, Pr. 1* 9d⟩≡ (8b)
```
get, err := test.Output()
if err != nil {
          t.Error(err)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := os.ReadFile(f)
if err != nil {
          t.Error(err)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `os`, and `bytes`.

9e ⟨*Testing imports, Pr. 1* 8c⟩+≡ (8a) ◁8c
```
"strconv"
"os"
"bytes"
```

Chapter 3

`prim2tab`: Convert `primer3` Output to Table

```
PRIMER_TASK=generic
PRIMER_PICK_LEFT_PRIMER=1
PRIMER_PICK_RIGHT_PRIMER=1
PRIMER_PICK_INTERNAL_OLIGO=1
...
PRIMER_PAIR_0_PENALTY=0.991909
...
PRIMER_LEFT_0_SEQUENCE=AGGCGATATCTTCAACGGTA
PRIMER_RIGHT_0_SEQUENCE=AAAACTGTTTGCCGAGGAAG
PRIMER_INTERNAL_0_SEQUENCE=GGGAAGGGTAGATTCATG
...
=
```

Figure 3.1: Sample output from the primer design program `primer3` [1].

Introduction

The primers generated by `primer3` [1] often need to be ordered to allow easy extraction of those with the lowest penalty. However, the output of `primer3` has the key/value structure shown in Figure 3.1, which makes direct sorting difficult.

The program `prim2tab` takes the output of `primer3` as input and prints a table of primers where each row consists of the following four columns:

1. penalty of primer pair
2. sequence of left primer
3. sequence of right primer
4. sequence of internal primer

As shown in Figure 3.1, this is also the order in which these items are reported by `primer3`.

Here is some sample output from `prim2tab`, which can now be ordered by score by piping it through `sort -n`.

```
# Penalty  Forward                Reverse                Internal
0.991909   AGGCGATATCTTCAACGGTA   AAAACTGTTTGCCGAGGAAG   GGGAAGGGTAGATTCATG
0.991909   TAGGCGATATCTTCAACGGT   AAAACTGTTTGCCGAGGAAG   GGGAAGGGTAGATTCATG
0.998004   GTAGGCGATATCTTCAACGG   AAAACTGTTTGCCGAGGAAG   GGGAAGGGTAGATTCATG
0.998004   GGTAGGCGATATCTTCAACG   AAAACTGTTTGCCGAGGAAG   GGGAAGGGTAGATTCATG
```

Implementation

The program `prim2tab` has hooks for imports, variables, functions, and the logic of the main function.

Prog. 2 (`prim2tab`)

11 ⟨*prim2tab.go* 11⟩≡

```
package main

import (
```

```
        ⟨Imports, P. 2 12b⟩
)
⟨Variables, P. 2 14a⟩
⟨Functions, P. 2 13c⟩
func main() {
        ⟨Main function, P. 2 12a⟩
}
```

In the main function we first set the name of `prim2tab`. Then we set the usage, declare and parse the options, and parse the input files.

12a ⟨*Main function, P. 2* 12a⟩≡ (11)

```
util.SetName("prim2tab")
⟨Set usage, P. 2 12c⟩
⟨Declare options, P. 2 12e⟩
⟨Parse options, P. 2 12g⟩
⟨Parse input files, P. 2 13a⟩
```

We import `util`.

12b ⟨*Imports, P. 2* 12b⟩≡ (11) 12d▷

```
"github.com/evolbioinf/prim/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `prim2tab`, and an example command.

12c ⟨*Set usage, P. 2* 12c⟩≡ (12a)

```
u := "prim2tab [option]... [file]..."
p := "Convert output of primer3 to table."
e := "primt2tab primer3.out | sort -n"
clio.Usage(u, p, e)
```

We import `clio`.

12d ⟨*Imports, P. 2* 12b⟩+≡ (11) ◁12b 12f▷

```
"github.com/evolbioinf/clio"
```

There is only one option, the version.

12e ⟨*Declare options, P. 2* 12e⟩≡ (12a)

```
optV := flag.Bool("v", false, "version")
```

We import `flag`.

12f ⟨*Imports, P. 2* 12b⟩+≡ (11) ◁12d 13b▷

```
"flag"
```

We parse the option and if the user requested the version, we print it, which also stops `prim2tab`.

12g ⟨*Parse options, P. 2* 12g⟩≡ (12a)

```
flag.Parse()
if *optV {
        util.Version()
}
```

The remaining tokens on the command line are taken as input files. We iterate over them and print the primers in a single table. We format this table using a `tabwriter`, which we construct and initialize with column headers. Parsing of the input files is then delegated to the function `ParseFiles`, which applies the function `scan` to each file and in turn takes the `tabwriter` as an argument. Having parsed the input, we flush the `tabwriter`.

13a ⟨*Parse input files, P. 2* 13a⟩≡ (12a)
```
files := flag.Args()
w := tabwriter.NewWriter(os.Stdout, 0, 1, 2, ' ', 0)
fmt.Fprintf(w, "# Penalty\tForward\tReverse\tInternal\n")
clio.ParseFiles(files, scan, w)
w.Flush()
```

We import `tabwriter`, `os`, and `fmt`.

13b ⟨*Imports, P. 2* 12b⟩+≡ (11) ◁12f 13d▷
```
"text/tabwriter"
"os"
"fmt"
```

Inside `scan` we retrieve the `tabwriter` and iterate over the lines in the current input file. As already mentioned, lines of `primer3` output have a key/value structure separated by an equal sign, `k=v` (Figure 3.1). The only exception to this rule is the last line, which is a single equal sign. So we split each line at the equal sign, and if this results in two fields, we go on to extract the desired data.

13c ⟨*Functions, P. 2* 13c⟩≡ (11)
```
func scan(r io.Reader, args ...interface{}) {
        w := args[0].(*tabwriter.Writer)
        sc := bufio.NewScanner(r)
        for sc.Scan() {
                fields := strings.Split(sc.Text(), "=")
                if len(fields) == 2 {
                        ⟨Extract data, P. 2 13e⟩
                }
        }
}
```

We import `io`, `tabwriter`, `bufio`, and `strings`.

13d ⟨*Imports, P. 2* 12b⟩+≡ (11) ◁13b 14b▷
```
"io"
"text/tabwriter"
"bufio"
"strings"
```

We extract the four columns we are looking for, penalty, forward, reverse, and internal.

13e ⟨*Extract data, P. 2* 13e⟩≡ (13c)
```
⟨Extract penalty, P. 2 14c⟩
⟨Extract forward primer, P. 2 14e⟩
⟨Extract reverse primer, P. 2 14g⟩
⟨Extract internal primer, P. 2 15b⟩
```

The penalty for primer pairs is reported as, for example,

```
PRIMER_PAIR_0_PENALTY=1.320714
```

Since we need to cover all primer pairs, not just pair zero, we define a regular expression for matching.

14a ⟨*Variables, P. 2* 14a⟩≡ (11) 14d▷
```
var penaltyRE = regexp.MustCompile(
          `PRIMER_PAIR_[0-9]+_PENALTY`)
```

We import `regexp`.

14b ⟨*Imports, P. 2* 12b⟩+≡ (11) ◁13d
```
"regexp"
```

Now we can check for a match to pair penalty and if successful print the value.

14c ⟨*Extract penalty, P. 2* 14c⟩≡ (13e)
```
if penaltyRE.MatchString(fields[0]) {
          fmt.Fprintf(w, "%s", fields[1])
}
```

A forward primer is reported as, for example,

```
PRIMER_LEFT_0_SEQUENCE=CGGCAATATCATAGACATCGT
```

so we define the corresponding regular expression.

14d ⟨*Variables, P. 2* 14a⟩+≡ (11) ◁14a 14f▷
```
var forwardRE = regexp.MustCompile(
          `PRIMER_LEFT_[0-9]+_SEQUENCE`)
```

We extract the forward primer.

14e ⟨*Extract forward primer, P. 2* 14e⟩≡ (13e)
```
if forwardRE.MatchString(fields[0]) {
          fmt.Fprintf(w, "\t%s", fields[1])
}
```

A reverse primer is reported as, for example,

```
PRIMER_RIGHT_0_SEQUENCE=CGGCAATATCATAGACATCGT
```

and we define the corresponding regular expression.

14f ⟨*Variables, P. 2* 14a⟩+≡ (11) ◁14d 15a▷
```
var reverseRE = regexp.MustCompile(
          `PRIMER_RIGHT_[0-9]+_SEQUENCE`)
```

We extract the reverse primer.

14g ⟨*Extract reverse primer, P. 2* 14g⟩≡ (13e)
```
if reverseRE.MatchString(fields[0]) {
          fmt.Fprintf(w, "\t%s", fields[1])
}
```

An internal primer is reported as, for example,

```
PRIMER_INTERNAL_0_SEQUENCE=TCACGACGATAATTATCTTT
```

and we define the required regular expression.

15a ⟨*Variables, P. 2* 14a⟩+≡ (11) ◁14f
```
var internalRE = regexp.MustCompile(
          `PRIMER_INTERNAL_[0-9]+_SEQUENCE`)
```

We extract the internal primer and end the line with a linebreak.

15b ⟨*Extract internal primer, P. 2* 15b⟩≡ (13e)
```
if internalRE.MatchString(fields[0]) {
          fmt.Fprintf(w, "\t%s\n", fields[1])
}
```

We have finished `prim2tab`, let's test it.

Testing

Our framework for testing `prim2tab` has hooks for imports and the testing logic.

15c ⟨*prim2tab_test.go* 15c⟩≡
```
package main

import (
          "testing"
          ⟨Testing imports, P. 2 15f⟩
)
func TestPrim2tab(t *testing.T) {
          ⟨Testing, P. 2 15d⟩
}
```

We construct one test and run it.

15d ⟨*Testing, P. 2* 15d⟩≡ (15c)
```
⟨Construct test, P. 2 15e⟩
⟨Run test, P. 2 16a⟩
```

For testing, we apply `prim2tab` to a file of `primer3` output, `prim.out`.

15e ⟨*Construct test, P. 2* 15e⟩≡ (15d)
```
test := exec.Command("./prim2tab", "prim.out")
```

We import `exec`.

15f ⟨*Testing imports, P. 2* 15f⟩≡ (15c) 16b▷
```
"os/exec"
```

We run the test and compare the result we get to the result we want, which is stored in the file `r.txt`.

16a ⟨*Run test, P. 2* 16a⟩≡ (15d)

```
get, err := test.Output()
if err != nil {
        t.Error(err)
}
want, err := os.ReadFile("r.txt")
if err != nil {
        t.Error(err)
}
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `os` and `bytes`.

16b ⟨*Testing imports, P. 2* 15f⟩+≡ (15c) ◁15f

```
"os"
"bytes"
```

Chapter 4

scop: Score Primers

Introduction

Before using PCR primers *in vitro*, it is wise to estimate their sensitivity and specificity through digital PCR against a large database. The program `scop` scores primers by calculating their *in silico* sensitivity and specificity. It takes as input one or more primers intended for one reaction mix. As additional input it takes a set of target taxon IDs and a Blast database linked to the NCBI taxonomy, for example `nt`. It then returns the sensitivity of the primer set,

$$s_{\mathrm{n}} = \frac{t_{\mathrm{p}}}{t_{\mathrm{p}} + f_{\mathrm{n}}}. \tag{4.1}$$

where t_{p} is the number of true positives and f_{n} the number of false negatives.

It also calculates the specificity as the fraction of true hits,

$$s_{\mathrm{p}} = \frac{t_{\mathrm{p}}}{t_{\mathrm{p}} + f_{\mathrm{p}}}, \tag{4.2}$$

where f_{p} is the number of false positives.

In addition, `scop` prints the true positives, false positives, and the false negatives, if any, for further checking with the program `cops`, which corrects the primer scores obtained by `scop`.

To construct an example run, we download a sample database and unpack it.

```
$ wget guanine.evolbio.mpg.de/prim/sample.tgz
$ tar -xvzf sample.tgz
```

Then we analyze the sample primers in the file `prim.fasta`, which should amplify all database entries of the taxa listed by taxon IDs in `tarTax.txt`. These target taxon IDs might have been generated, for example, using the program `neighbors` from the Neighbors package[1].

```
$ ./scop -d sample -t tarTax.txt prim.fasta
```

In this setup, the sensitivity of the tested primers is maximal, so there are no false negatives, but there appear to be a large number of false positives, leading to a specificity score of only 0.47.

```
PrimerSet:      prim.fasta
Sensitivity:    1
Specificity:    0.47
TruePositives:  AP018488.1 BA000007.3...
FalsePositives: AE005174.2 AP026080.1...
```

Implementation

The outline of `scop` contains hooks for imports, types, methods, functions, and the logic of the main function.

[1] `github.com/evolbioinf/neighbors`

Prog. 3 (scop)

19a ⟨*scop.go* 19a⟩≡
```
package main
import (
        ⟨Imports, Pr. 3 19c⟩
)
⟨Types, Pr. 3 25a⟩
⟨Methods, Pr. 3 27a⟩
func main() {
        ⟨Main function, Pr. 3 19b⟩
}
```

In the main function we first set the name of `scop`. Then we set its usage, declare the options, parse them, and parse the input.

19b ⟨*Main function, Pr. 3* 19b⟩≡ (19a)
```
util.SetName("scop")
⟨Set usage, Pr. 3 19d⟩
⟨Declare options, Pr. 3 20a⟩
⟨Parse options, Pr. 3 20c⟩
⟨Parse input, Pr. 3 22e⟩
```

We import `util`.

19c ⟨*Imports, Pr. 3* 19c⟩≡ (19a) 19e ▷
```
"github.com/evolbioinf/prim/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `scop`, and an example command.

19d ⟨*Set usage, Pr. 3* 19d⟩≡ (19b)
```
u := "scop -d <db> -t <taxids.txt> [option]... [foo.fa]..."
p := "Score primers by comparing them to a Blast database."
e := "scop -d nt -t targets.txt prim.fa"
clio.Usage(u, p, e)
```

We import `clio`.

19e ⟨*Imports, Pr. 3* 19c⟩+≡ (19a) ◁ 19c 20b ▷
```
"github.com/evolbioinf/clio"
```

We declare seven options, the first two of which are necessary for the program to run, so we shall make them mandatory,

1. `-d`: Blast database
2. `-t`: file of target taxon IDs
3. `-i`: maximum number of mismatches
4. `-l`: maximum length of amplicon
5. `-e`: E-value
6. `-T`: number of threads, which we initailize to the number of CPUs

7. `-m`: let e be the expected number of target accessions, then the maximum number of target sequences reported by Blast is set to $e \times$ `-m`.

8. `-v`: version

The `-m` option requires some explaining. It is important that we get all Blast hits, not just the subset limited by the Blast parameter `-max_target_seqs`, 500 by default. We might be tempted to set this parameter to the number of expected accessions, but that would preclude finding any hits beyond this set. So we give ourselves a three-fold safety margin, but also allow the user to increase or decrease this value. If you are surprised by low sensitivity, try increasing `-m` and you might observe a reduction in false negatives, and hence an increase in the sensitivity.

20a ⟨*Declare options, Pr. 3* 20a⟩≡ (19b)
```
optD := flag.String("d", "", "Blast database")
optT := flag.String("t", "", "file of target taxon IDs")
optI := flag.Int("i", 5, "maximum number of mismatches")
optL := flag.Int("l", 4000, "maximum length of amplicon")
optE := flag.Float64("e", 10.0, "E-value")
nt := runtime.NumCPU()
optTT := flag.Int("T", nt, "number of threads (default CPUs)")
optM := flag.Float64("m", 3.0, "set the maximum number " +
          "of target sequences in Blast to the expected " +
          "number of accessions times -m")
optV := flag.Bool("v", false, "version")
```

We import `flag`.

20b ⟨*Imports, Pr. 3* 19c⟩+≡ (19a) ◁19e 21a▷
```
"flag"
```

We parse the options and first respond to `-v`, as a request for the version stops `scop`. Then we check whether the mandatory options—Blast database (`-d`) and target taxon IDs (`-t`)—have been set. If so, we look up the expected target accessions and respond to the number of threads, (`-T`).

20c ⟨*Parse options, Pr. 3* 20c⟩≡ (19b)
```
flag.Parse()
⟨Respond to -v, Pr. 3 20d⟩
⟨Has -d been set? Pr. 3 20e⟩
⟨Has -t been set? Pr. 3 21b⟩
⟨Look up expected target accessions, Pr. 3 21c⟩
⟨Respond to -T, Pr, 3 22c⟩
```

If the user requested the version, we print it.

20d ⟨*Respond to -v, Pr. 3* 20d⟩≡ (20c)
```
if *optV {
          util.Version()
}
```

If the user didn't supply a Blast database, we bail with a friendly message.

20e ⟨*Has -d been set? Pr. 3* 20e⟩≡ (20c)
```
if *optD == "" {
          log.Fatal("please supply a Blast datbase")
}
```

We import `log`.

21a ⟨*Imports, Pr. 3* 19c⟩+≡ (19a) ◁20b 21e▷
```
"log"
```

Similarly, if the user didn't supply a file with target taxon IDs, we bail with a friendly message.

21b ⟨*Has* `-t` *been set? Pr. 3* 21b⟩≡ (20c)
```
if *optT == "" {
        log.Fatal("please supply a file of target taxon IDs")
}
```

We store the expected target accessions in a string map and obtain them by querying the Blast database and analyzing the query results.

21c ⟨*Look up expected target accessions, Pr. 3* 21c⟩≡ (20c)
```
etacc := make(map[string]bool)
⟨Query Blast database, Pr. 3 21d⟩
⟨Analyze query result, Pr. 3 22a⟩
```

We query the Blast database by calling the program `blastdbcmd` such that it returns the accessions and title lines for entries that belong to the target taxa. Here is an example command to achieve this,

```
blastdbcmd -db nt -taxidlist tarTax.txt -outfmt "%a %t"
```

Note the output format, where `%a` is the accession and `%t` the title line. We construct this command and run it. Notice that we construct the argument array by splitting the command string into its constituent fields. However, since the output format takes as value a composite string, we append that to the argument slice separately. Then we run the command and check the error variable it returns

21d ⟨*Query Blast database, Pr. 3* 21d⟩≡ (21c)
```
tmpl := "blastdbcmd -db %s -taxidlist %s -outfmt "
cs := fmt.Sprintf(tmpl, *optD, *optT)
args := strings.Fields(cs)
args = append(args, "%a %t")
cmd := exec.Command("blastdbcmd")
cmd.Args = args
out, err := cmd.Output()
util.Check(err)
```

We import `fmt`, `strings`, and `exec`.

21e ⟨*Imports, Pr. 3* 19c⟩+≡ (19a) ◁21a 22b▷
```
"fmt"
"strings"
"os/exec"
```

We split the query output at the line breaks into entries of the Blast database. We iterate over these entries and save the accessions of “complete genomes” that are not “plasmids”.

22a ⟨*Analyze query result, Pr. 3* 22a⟩≡ (21c)
```
cg := []byte("complete genome")
pl := []byte("plasmid")
entries := bytes.Split(out, []byte("\n"))
for _, entry := range entries {
        if bytes.Contains(entry, cg)  &&
                !bytes.Contains(entry, pl) {
                acc := string(bytes.Fields(entry)[0])
                etacc[acc] = true
        }
}
```

We import `bytes`.

22b ⟨*Imports, Pr. 3* 19c⟩+≡ (19a) ◁21e 22d▷
```
"bytes"
```

If the user didn’t set the number of threads, we set it to the number of CPUs.

22c ⟨*Respond to -T, Pr, 3* 22c⟩≡ (20c)
```
if *optTT == 0 {
        (*optTT) = runtime.NumCPU()
}
```

We import `runtime`.

22d ⟨*Imports, Pr. 3* 19c⟩+≡ (19a) ◁22b 23b▷
```
"runtime"
```

The remaining tokens on the command line are taken as the names of input files containing sets of primers. If there are none, we expect that the primer set is supplied via the standard input and copy it from there. Then we iterate over the files and analyze each one.

22e ⟨*Parse input, Pr. 3* 22e⟩≡ (19b)
```
primerSets := flag.Args()
if len(primerSets) == 0 {
        ⟨Copy primer set from standard input, Pr. 3 23a⟩
}
for _, primerSet := range primerSets {
        ⟨Analyze primer set, Pr. 3 23c⟩
}
```

We create a temporary file, write the primer set that we read from the standard input stream to it, and store its name.

23a ⟨*Copy primer set from standard input, Pr. 3* 23a⟩≡ (22e)

```
ps, err := os.CreateTemp("", "prim*.fasta")
util.Check(err)
defer ps.Close()
defer os.Remove(ps.Name())
sc := fasta.NewScanner(os.Stdin)
for sc.ScanSequence() {
        seq := sc.Sequence()
        fmt.Fprintf(ps, "%s\n", seq)
}
primerSets = append(primerSets, ps.Name())
```

We import `os` and `fasta`.

23b ⟨*Imports, Pr. 3* 19c⟩+≡ (19a) ◁22d 25d▷

```
"os"
"github.com/evolbioinf/fasta"
```

To analyze a primer set, we run Blast, get the observed target accessions from the Blast output, and compare them to the expected target accessions. From this comparison we get the true positives, false positives, and false negatives. This allows us to calculate the sensitivity and specificity of our primer set according to equations (4.1) and (4.2), which we report.

23c ⟨*Analyze primer set, Pr. 3* 23c⟩≡ (22e)

⟨*Run Blast, Pr. 3* 24a⟩
⟨*Get observed target accessions, Pr. 3* 24c⟩
⟨*Compare observed and expected target accessions, Pr. 3* 27d⟩
⟨*Calculate sensitivity, Pr. 3* 28d⟩
⟨*Calculate specificity, Pr. 3* 28e⟩
⟨*Report sensitivity and specificity, Pr. 3* 29a⟩

Table 4.1: The six columns of output in our run of `blastn` for finding amplicons.

Col.	Name	Meaning
1	`length`	alignment length
2	`qlen`	query length
3	`mismatch`	number of mismatches
4	`saccver`	subject accession with version
5	`sstart`	start in subject
6	`send`	end in subject

We construct the Blast command for short queries like we constructed the `blastdbcmd` command. However, this time the command is called `blastn` and its task is called `blastn-short`.

We are looking for hits where the alignment length is equal to the primer length, that is, the query length, with no more than a maximum number of mismatches. Any pairs of such hits are checked to see whether they might form an amplicon by investigating their subject accession and position. So as our output we request a table consisting of alignment length, query length, the number of mismatches, the subject accession, the subject start, and the subject end (Table 4.1). Once constructed, we run the Blast command, store its output, and check the error it returns.

24a ⟨*Run Blast, Pr. 3* 24a⟩≡ (23c)

```
tmpl = "blastn -task blastn-short -query %s -db %s -evalue " +
          "%g -num_threads %d -max_target_seqs %d -outfmt "
mts := int(*optM * float64(len(etacc))) + 1
//fmt.Printf("mts: %d\n", mts)
cs = fmt.Sprintf(tmpl, primerSet, *optD, *optE, *optTT, mts)
args = strings.Fields(cs)
args = append(args, "6 length qlen mismatch " +
          "saccver sstart send")
cmd = exec.Command("blastn")
cmd.Args = args
out, err = cmd.CombinedOutput()
⟨Check Blast error, Pr. 3 24b⟩
```

If Blast returned an error, we print the output and quit.

24b ⟨*Check Blast error, Pr. 3* 24b⟩≡ (24a)

```
if err != nil {
          log.Fatal(string(out))
}
```

We construct a map for storing the observed target accessions and a slice of Blast results. Then we store the Blast hits, before we filter them and look for amplicons.

24c ⟨*Get observed target accessions, Pr. 3* 24c⟩≡ (23c)

```
otacc := make(map[string]bool)
hits := make([]*Hit, 0)
⟨Store Blast hits, Pr. 3 25b⟩
⟨Filter Blast hits, Pr. 3 26a⟩
⟨Find amplicons, Pr. 3 26b⟩
```

We declare a Blast hit to consist of the six fields listed in Table 4.1.

25a ⟨*Types, Pr. 3* 25a⟩≡ (19a) 26d▷
```
type Hit struct {
        length, qlen, mismatch int
        saccver string
        sstart, send int
}
```

We iterate over the primer sets of the Blast output and from every line that consists of six fields, we extract the hit.

25b ⟨*Store Blast hits, Pr. 3* 25b⟩≡ (24c)
```
lines := bytes.Split(out, []byte("\n"))
for _, line := range lines {
        fields := bytes.Fields(line)
        hit := new(Hit)
        if len(fields) == 6 {
                ⟨Extract hit, Pr. 3 25c⟩
        }
        hits = append(hits, hit)
}
```

We convert the byte slices of a hit either to string or to integer. If we convert to integer, we also check the error returned.

25c ⟨*Extract hit, Pr. 3* 25c⟩≡ (25b)
```
hit.length, err = strconv.Atoi(string(fields[0]))
util.Check(err)
hit.qlen, err = strconv.Atoi(string(fields[1]))
util.Check(err)
hit.mismatch, err = strconv.Atoi(string(fields[2]))
util.Check(err)
hit.saccver = string(fields[3])
hit.sstart, err = strconv.Atoi(string(fields[4]))
util.Check(err)
hit.send, err = strconv.Atoi(string(fields[5]))
util.Check(err)
```

We import `strconv`.

25d ⟨*Imports, Pr. 3* 19c⟩+≡ (19a) ◁23b 26c▷
```
"strconv"
```

p_f 5’—TGACCGCCAATATTGCCAGT—3’
p_r 5’—TTCTTACGGGGAGACGCAAC—3’

---

p_f	5’→		TGACCGCCAATATTGCCAGT					→3’
t_f	5’→	...	TGACCGCCAATATTGCCAGT	...	GTTGCGTCTCCCCGTAAGAA	...		→3’
t_r	3’←	...	ACTGGCGGTTATAACGGTCA	...	CAACGCAGAGGGGCATTCTT	...		←5’
p_r	3’←				CAACGCAGAGGGGCATTCTT			←5’

Figure 4.1: Forward and reverse PCR primers, p_f and p_r (top panel), along the forward or reverse strands of a template, t_f and t_r (bottom panel).

We retain hits with query length equal to the alignment length and with no more than the maximum number of mismatches.

26a ⟨*Filter Blast hits, Pr. 3* 26a⟩≡ (24c)
```
i := 0
for _, hit := range hits {
        if hit.qlen == hit.length &&
                hit.mismatch <= *optI {
                hits[i] = hit
                i++
        }
}
hits = hits[:i]
```

Amplicons are hits on the same subject where the 5’-hit is on the forward strand and the 3’-hit on the reverse strand (Figure 4.1). So we begin our search for amplicons by ordering the hits.

In Blast, strandedness is encoded in the relationship between the start and the end position of a hit. If the start is less than the end, the hit is on the forward strand, if the start is greater than the end, the hit is on the reverse strand. So we iterate over the ordered hits and for each potential forward primer that hasn’t yet produced an amplicon look for a reverse primer.

26b ⟨*Find amplicons, Pr. 3* 26b⟩≡ (24c)
```
sort.Sort(HitSlice(hits))
for i, hit := range hits {
        if !otacc[hit.saccver] && hit.sstart < hit.send {
                fp := hit
                ⟨Look for reverse primer, Pr. 3 27c⟩
        }
}
```

We import `sort`.

26c ⟨*Imports, Pr. 3* 19c⟩+≡ (19a) ◁25d
```
"sort"
```

We declare the sortable type `HitSlice`.

26d ⟨*Types, Pr. 3* 25a⟩+≡ (19a) ◁25a
```
type HitSlice []*Hit
```

To allow sorting, we specify the three Methods required by the `Sort` interface, `Len`, `Swap`, and `Less`. We begin with `Len` and `Swap`.

27a ⟨*Methods, Pr. 3* 27a⟩≡ (19a) 27b▷

```
func (h HitSlice) Len() int {
        return len(h)
}
func (h HitSlice) Swap(i, j int) {
        h[i], h[j] = h[j], h[i]
}
```

In `Less` we sort by subject accession and within identical subjects by start position.

27b ⟨*Methods, Pr. 3* 27a⟩+≡ (19a) ◁27a

```
func (h HitSlice) Less(i, j int) bool {
        if h[i].saccver == h[j].saccver {
                return h[i].sstart < h[j].sstart
        }
        return h[i].saccver < h[j].saccver
}
```

Reverse primers are located on the reverse strand of the same subject within the range of permissible amplicon lengths.

27c ⟨*Look for reverse primer, Pr. 3* 27c⟩≡ (26b)

```
for j := i+1; j < len(hits); j++ {
        if hits[j].sstart > hits[j].send &&
                fp.saccver == hits[j].saccver {
                rp := hits[j]
                if rp.send - fp.sstart + 1 <= *optL {
                        otacc[fp.saccver] = true
                        break
                }
        }
}
```

We now have the expected and the observed target accessions in hand and compare them to calculate the number of true positives, t_p, false positives, f_p, and false negatives, f_n.

27d ⟨*Compare observed and expected target accessions, Pr. 3* 27d⟩≡ (23c)

⟨*Calculate* t_p*, Pr. 3* 28a⟩
⟨*Calculate* f_p*, Pr. 3* 28b⟩
⟨*Calculate* f_n*, Pr. 3* 28c⟩

The true positives are the observed accessions that are also expected. We count and save them.

28a ⟨*Calculate* t_p*, Pr. 3* 28a⟩≡ (27d)

```
tp := 0
truePositives := make([]string, 0)
for o, _ := range otacc {
        if etacc[o] {
                truePositives = append(truePositives, o)
                tp++
        }
}
```

The false positives are the observed accessions that are not expected. We also save these accessions.

28b ⟨*Calculate* f_p*, Pr. 3* 28b⟩≡ (27d)

```
fp := 0
falsePositives := make([]string, 0)
for o, _ := range otacc {
        if !etacc[o] {
                fp++
                falsePositives = append(falsePositives, o)
        }
}
```

The false negatives are the expected accessions that weren't observed; we save them, too.

28c ⟨*Calculate* f_n*, Pr. 3* 28c⟩≡ (27d)

```
fn := 0
falseNegatives := make([]string, 0)
for e, _ := range etacc {
        if !otacc[e] {
                fn++
                falseNegatives = append(falseNegatives, e)
        }
}
```

We calculate the sensitivity of our primer sample according to equation (4.1).

28d ⟨*Calculate sensitivity, Pr. 3* 28d⟩≡ (23c)

```
sn := float64(tp) / (float64(tp) + float64(fn))
```

We calculate the specificity of our primer sample according to equation (4.2).

28e ⟨*Calculate specificity, Pr. 3* 28e⟩≡ (23c)

```
sp := float64(tp) / (float64(tp) + float64(fp))
```

We report the name of our primer sample, its sensitivity, and its specificity. In addition, we list the true positives, false positives, and false negatives.

29a ⟨*Report sensitivity and specificity, Pr. 3* 29a⟩≡ (23c)
```
fmt.Printf("PrimerSet:\t%s\n", primerSet)
fmt.Printf("Sensitivity:\t%.3g\n", sn)
fmt.Printf("Specificity:\t%.3g\n", sp)
⟨Print true positives, Pr. 3 29b⟩
⟨Print false positives, Pr. 3 29c⟩
⟨Print false negatives, Pr. 3 29d⟩
```

We sort the true positives to make their order reproducible, and list them as a blank-delimited row.

29b ⟨*Print true positives, Pr. 3* 29b⟩≡ (29a)
```
if len(truePositives) > 0 {
        sort.Strings(truePositives)
        fmt.Printf("TruePositives:\t%s", truePositives[0])
        for i := 1; i < tp; i++ {
                fmt.Printf(" %s", truePositives[i])
        }
        fmt.Printf("\n")
}
```

We also print the sorted false positives as a blank-delimited row.

29c ⟨*Print false positives, Pr. 3* 29c⟩≡ (29a)
```
if len(falsePositives) > 0 {
        sort.Strings(falsePositives)
        fmt.Printf("FalsePositives:\t%s", falsePositives[0])
        for i := 1; i < fp; i++ {
                fmt.Printf(" %s", falsePositives[i])
        }
        fmt.Printf("\n")
}
```

We finally list the sorted false negatives as a blank-delimited row.

29d ⟨*Print false negatives, Pr. 3* 29d⟩≡ (29a)
```
if len(falseNegatives) > 0 {
        sort.Strings(falseNegatives)
        fmt.Printf("FalseNegatives:\t%s", falseNegatives[0])
        for i := 1; i < fn; i++ {
                fmt.Printf(" %s", falseNegatives[i])
        }
        fmt.Printf("\n")
}
```

We are finished with `scop`, time to test it.

Testing

Our test for `scop` contains hooks for imports and the testing logic.

30a ⟨*scop_test.go* 30a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Pr. 3 30d⟩
)
func TestScop(t *testing.T) {
        ⟨Testing, Pr. 3 30b⟩
}
```

We construct one test and run it.

30b ⟨*Testing, Pr. 3* 30b⟩≡ (30a)
```
⟨Construct test, Pr. 3 30c⟩
⟨Run test, Pr. 3 30e⟩
```

We use `scop` to score the primers in `prim.fasta` using the sample database, `sample`, and the target taxa in `tarTax.txt`.

30c ⟨*Construct test, Pr. 3* 30c⟩≡ (30b)
```
test := exec.Command("./scop", "-d", "sample",
        "-t", "tarTax.txt", "prim.fasta")
```

We import `exec`.

30d ⟨*Testing imports, Pr. 3* 30d⟩≡ (30a) 30f▷
```
"os/exec"
```

We run the test and compare the result we get to the result we want, which is contained in the file `r.txt`.

30e ⟨*Run test, Pr. 3* 30e⟩≡ (30b)
```
get, err := test.Output()
if err != nil {
        t.Error(err)
}
want, err := os.ReadFile("r.txt")
if err != nil {
        t.Error(err)
}
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `os` and `bytes`.

30f ⟨*Testing imports, Pr. 3* 30d⟩+≡ (30a) ◁30d
```
"os"
"bytes"
```

Chapter 5

`cops`: Correct Primer Scores

Table 5.1: Reclassification of a taxon i based on the relationship between its distance to the reference strain, d_i, and the threshold distance, d_t.

Original Classification	Expectation	Reclassification
true positive	$d_i \leq d_\mathrm{t}$	false positive
false positive	$d_i > d_\mathrm{t}$	true positive
false negative	$d_i \leq d_\mathrm{t}$	true negative

```
PrimerSet:      prim.fasta
Sensitivity:    1
Specificity:    0.47
TruePositives:  AP018488.1 BA000007.3...
FalsePositives: AE005174.2 AP026080.1...
```

Figure 5.1: Example output from `scop`; since no false negatives were found, the sensitivity is 1 and no false negatives are listed.

Introduction

The primer scores generated by `scop` consist of sensitivity and specificity values calculated on the basis of the given taxonomy. This taxonomy might not be correct, so we double check it. The purpose of the program `cops` is to correct the primer scores returned by `scop`.

The program `scop` returns the accessions of the strains it classified as true positive, true negative, and false negative. These accessions are read by the program `cops`, together with the accession of a reference stain, the name of the Blast database used by `scop`, and a threshold distance, d_t. For every accession, i, read as input, `cops` calculates the distance to the reference stains, d_i, and compares d_i to d_t. A true positive should have $d_i \leq d_\mathrm{t}$, otherwise it is reclassified as a false positive. Similarly, a false positive should have $d_i > d_\mathrm{t}$, otherwise it is reclassified as a true positive. And a false negative should have $d_i \leq d_\mathrm{t}$, otherwise it is reclassified as a true negative. These classifications are summarized in Table 5.1.

Figure 5.1 shows example output from `scop` taken from an analysis of *E. coli* O157:H7. Two things are remarkable about this output. First, accession AE005174, which is listed as a false positive, happens to correspond to the type strain on O157:H7, EDL933. So it certainly isn't a false positive. The reason for its misclassification is that the header of AE005174 only contains the term "genome", but not "complete genome", which is required by `scop` for inclusion in the set of target accessions. The second thing to note is the low specificity of 0.47. The program `cops` leaves the sensitivity unchanged but corrects the specificity to a more respectable 0.97 (Figure 5.2).

The user can also request that the accessions are annotated with distances to the reference strain. Figure 5.3 the distance to the reference, zero, and the distance to the true positive that is alphabetically next, 7×10^{-4}. In contrast, the two distance to the false positives shown are greater than 1%.

In this example analysis `cops` returns distances for 334 true positives and 11 false positives. Figure 5.4 shows the distribution of all log-distances, except for the zero

```
PrimerSet:      prim.fasta
Sensitivity:    1
Specificity:    0.97
TruePositives:  AE005174.2 AP018488.1...
FalsePositives: CP027437.1 CP043542.1...
```

Figure 5.2: Corrected primer scores using the data in Figure 5.1 as input.

```
PrimerSet:      prim.fasta
Sensitivity:    1
Specificity:    0.97
TruePositives:  AE005174.2 0 AP018488.1 0.000699...
FalsePositives: CP027437.1 0.0218 CP043542.1 0.0144...
```

Figure 5.3: Corrected primer scores with distances to the reference using the data in Figure 5.1 as input.

distance. Note the major mode centered on -3.5 for the true positives and the minor mode between -2 and -1.5 for the 11 false positives. Their existence is probably due to horizontal transfer of the marker.

Implementation

Our outline of `cops` contains hooks for imports and functions, and for the logic of the main function.

Prog. 4 (`cops`)

33a ⟨*cops.go* 33a⟩≡

```
package main

import (
        ⟨Imports, Pr. 4 34a⟩
)
⟨Functions, Pr. 4 36d⟩
func main() {
        ⟨Main function, Pr. 4 33b⟩
}
```

In the main function we first set the name of `prim`. Then we set the usage, declare the options, parse the options, and parse the input.

33b ⟨*Main function, Pr. 4* 33b⟩≡ (33a)

```
util.SetName("cops")
⟨Set usage, Pr. 4 34b⟩
⟨Declare options, Pr. 4 35a⟩
⟨Parse options, Pr. 4 35c⟩
⟨Parse input, Pr. 4 36c⟩
```

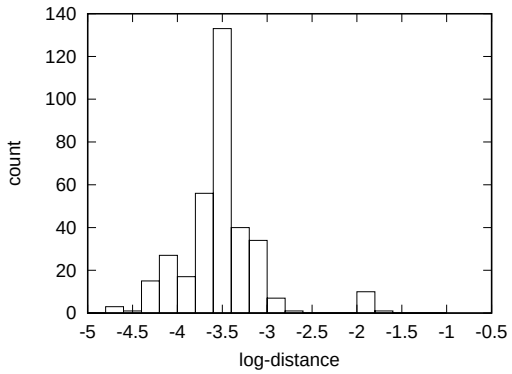

Figure 5.4: Distribution of 345 log-distances between genomes *in silico* amplified by `scop` and the type strain of *E. coli* O157:H7, EDL933; note the false positives between -2 and -1.5.

We import `util`.

34a ⟨*Imports, Pr. 4* 34a⟩≡ (33a) 34c ▷

```
"github.com/evolbioinf/prim/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `cops`, and an example command.

34b ⟨*Set usage, Pr. 4* 34b⟩≡ (33b)

```
u := "cops -d <db> -r <ref> -t <d> [option]... [foo.txt]..."
p := "Correct primer scores calculated with scop."
e := "cops -d nt -r AE005174 -t 2e-3 scop.out"
clio.Usage(u, p, e)
```

We import `clio`.

34c ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁ 34a 35b ▷

```
"github.com/evolbioinf/clio"
```

We declare the obligatory version option, `-v`, and three mandatory options, the database, `-d`, the accession of the reference strain, and the threshold distance to the reference. In addition, the user can run `phylonium` with a different number of threads than the number of CPUs, print the distances as part of the output, and include the true positives in the checking. You might wander why anyone would want to also double-check the true positives. However, it is possible to find the marker in the wrong genome that has been misclassified as target. This is highly unlikely, but not impossible, which is why we don't routinely include the true positives in the checking, but at the same time don't deny the user the possibility to do so.

35a ⟨*Declare options, Pr. 4* 35a⟩≡ (33b)

```
optV := flag.Bool("v", false, "version")
optD := flag.String("d", "", "Blast datbase")
optR := flag.String("r", "",
          "accession of reference target strain")
optT := flag.Float64("t", 0, "threshold distance to reference")
numThreads := runtime.NumCPU()
optTT := flag.Int("T", numThreads, "number of threads")
optDD := flag.Bool("D", false, "include distances in output")
optP := flag.Bool("p", false, "also check true positives " +
          "(default only check false positives and false negatives)")
```

We import `flag` and `runtime`.

35b ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁34c 36b▷

```
"flag"
"runtime"
```

We parse the options and respond to the version option, `-v`, as this stops `cops`. Then we ensure the mandatory options have been set.

35c ⟨*Parse options, Pr. 4* 35c⟩≡ (33b)

```
flag.Parse()
⟨Respond to -v, Pr. 4 35d⟩
⟨Ensure mandatory options, Pr. 4 36a⟩
```

If the user requested the version, we print it.

35d ⟨*Respond to* `-v`*, Pr. 4* 35d⟩≡ (35c)

```
if *optV {
          util.Version()
}
```

We make sure the user supplied a database, a reference strain, and a threshold distance. We quit with a friendly message if that's not the case.

36a ⟨*Ensure mandatory options, Pr. 4* 36a⟩≡ (35c)
```
if *optD == "" {
        log.Fatal("please supply a Blast database")
}
if *optR == "" {
        log.Fatal("please supply a reference strain")
}
if *optT == 0 {
        log.Fatal("please supply the threshold " +
                "distance to the reference")
}
```

We import `log`.

36b ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁35b 36e▷
```
"log"
```

The remaining tokens on the command line are interpreted as file names. We parse these files using the function `ParseFiles`, which applies the function `parse` to each one of them. The function `parse` in turn takes as arguments the name of the Blast database, the reference strain, the threshold distance, the number of threads, whether or not to include the distances in the output, and whether or not to include the true positives.

36c ⟨*Parse input, Pr. 4* 36c⟩≡ (33b)
```
files := flag.Args()
clio.ParseFiles(files, parse, *optD, *optR,
        (*optT), *optTT, *optDD, *optP)
```

Inside `parse`, we retrieve the arguments we just passed. Then we split the file into reports generated by `scop` (Figure 5.1). We analyze each report, correct it, and print a corrected version.

36d ⟨*Functions, Pr. 4* 36d⟩≡ (33a) 44d▷
```
func parse(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Pr. 4 37a⟩
        d, e := io.ReadAll(r)
        util.Check(e)
        reports := bytes.Split(d, []byte("PrimerSet"))
        reports = reports[1:]
        for _, report := range reports {
                ⟨Analyze report, Pr. 4 37b⟩
                ⟨Correct report, Pr. 4 38a⟩
                ⟨Print corrected report, Pr. 4 43a⟩
        }
}
```

We import `io` and `bytes`.

36e ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁36b 39b▷
```
"io"
"bytes"
```

We retrieve the database, the reference, the threshold, the number of threads, the distance switch, and the true positives switch.

37a ⟨*Retrieve arguments, Pr. 4* 37a⟩≡ (36d)
```
db := args[0].(string)
re := args[1].(string)
dt := args[2].(float64)
nt := args[3].(int)
pd := args[4].(bool)
truePos := args[5].(bool)
```

We split the report into its constituent lines and check whether a terminal carriage return created an empty line, which we cut. Then we check we have at least three lines and no more than six. Note at this point that the terminal If not, we bail with a friendly message. If we're still going, we store the name of the primer set. Then we skip the lines containing the sensitivity and specificity and store the accessions of true positives, false positives, and false negatives in a map.

37b ⟨*Analyze report, Pr. 4* 37b⟩≡ (36d)
```
lines := bytes.Split(report, []byte("\n"))
if len(lines[len(lines)-1]) == 0 {
        lines = lines[:len(lines)-1]
}
l := len(lines)
if l < 3 || l > 6 {
        log.Fatalf("mal-formed report:\n%s\n", string(report))
}
primerSet := string(bytes.Fields(lines[0])[1])
lines = lines[3:]
accessions := make(map[string][]string)
⟨Store accessions, Pr. 4 37c⟩
```

We iterate over the remaining lines and split them into their constituent fields. The first field minus the trailing colon is the accession type, the remaining fields are the actual accessions.

37c ⟨*Store accessions, Pr. 4* 37c⟩≡ (37b)
```
for _, line := range lines {
        fields := bytes.Fields(line)
        if len(fields) > 0 {
                k := string(fields[0][:len(fields[0])-1])
                for i := 1; i < len(fields); i++ {
                        v := string(fields[i])
                        accessions[k] = append(accessions[k],
                                v)
                }
        }
}
```

Correcting the report essentially means, running `phyonium` on all input accessions. We do this by first constructing a directory for holing the sequence files on which we run `phylonium`. Then we place the sequences into that directory, run `phylonium`, and read the distance matrix it produces. These distances allow us to correct the false

positives, the false negatives, and, if desired, the true positives. True positives might turn out to be new false positives, for which we reserve a slice.

38a ⟨*Correct report, Pr. 4* 38a⟩≡ (36d)
```
⟨Create directory for sequence files, Pr. 4 38b⟩
⟨Get sequences, Pr. 4 38c⟩
⟨Run phylonium, Pr. 4 40c⟩
⟨Read distance matrix, Pr. 4 41a⟩
⟨Correct false positives, Pr. 4 41c⟩
⟨Correct false negatives, Pr. 4 42a⟩
nfp := make([]string, 0)
if truePos {
        ⟨Correct true positives, Pr. 4 42c⟩
}
```

We create a temporary directory, which is deleted when `parse` returns.

38b ⟨*Create directory for sequence files, Pr. 4* 38b⟩≡ (38a)
```
td, err := os.MkdirTemp("", "temp*")
util.Check(err)
defer os.RemoveAll(td)
```

To get the sequences, we write the accessions to a, query the blast database with `blastdbcmd`, and write the sequences returned by `blastdbcmd` to separate files.

38c ⟨*Get sequences, Pr. 4* 38c⟩≡ (38a)
```
⟨Write accessions to file, Pr. 4 38d⟩
⟨Run blastdbcmd, Pr. 4 39e⟩
⟨Write sequences to files, Pr. 4 40a⟩
```

We open a file for the accessions and write the reference accession, followed by the query accessions.

38d ⟨*Write accessions to file, Pr. 4* 38d⟩≡ (38c)
```
⟨Open accessions file, Pr. 4 38e⟩
⟨Write reference accession, Pr. 4 39a⟩
⟨Write query accessions, Pr. 4 39c⟩
```

The accessions file is inside the temporary directory, which means it is also removed at the end of the run.

38e ⟨*Open accessions file, Pr. 4* 38e⟩≡ (38d)
```
f, err := os.CreateTemp(td, "acc*.txt")
util.Check(err)
defer f.Close()
```

When dealing with the reference accession, we first need its exact version, that is, including the suffix for the sequence version. We get this by a call to `blastdbcmd`. The resulting accession is terminated by a newline, which we remove, before we print it.

39a ⟨*Write reference accession, Pr. 4* 39a⟩≡ (38d)
```
cmd := exec.Command("blastdbcmd", "-db", db, "-entry", re,
          "-outfmt", "%a")
out, err := cmd.CombinedOutput()
if err != nil {
          log.Fatalf("%s\n", out)
}
re = string(out[:len(out)-1])
fmt.Fprintf(f, "%s\n", re)
```

We import `exec` and `fmt`.

39b ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁36e 40b▷
```
"os/exec"
"fmt"
```

We write the accessions of the false positives, the false negatives, and, if desired, the true positives.

39c ⟨*Write query accessions, Pr. 4* 39c⟩≡ (38d)
```
keys := []string{"FalsePositives", "FalseNegatives"}
if truePos {
          keys = append(keys, "TruePositives")
}
for _, key := range keys {
          accs := accessions[key]
          if accs != nil {
                    ⟨Write accessions, Pr. 4 39d⟩
          }
}
```

We iterate over the current set of accessions and print each one.

39d ⟨*Write accessions, Pr. 4* 39d⟩≡ (39c)
```
for _, acc := range accs {
          fmt.Fprintf(f, "%s\n", acc)
}
```

We run `blastdbccmd` in batch mode to retrieve the sequences corresponding to the accessions we just wrote to file.

39e ⟨*Run `blastdbcmd`, Pr. 4* 39e⟩≡ (38c)
```
cmd = exec.Command("blastdbcmd", "-db", db, "-entry_batch", f.Name())
out, err = cmd.CombinedOutput()
if err != nil {
          log.Fatalf("%s\n", out)
}
```

We iterate over the sequences returned by `blastdbcmd` and write each one in a separate file inside the temporary directory. The file names are the sequence accessions with suffix `.fasta`.

40a ⟨*Write sequences to files, Pr. 4* 40a⟩≡ (38c)
```
sc := fasta.NewScanner(bytes.NewReader(out))
for sc.ScanSequence() {
          seq := sc.Sequence()
          fn := strings.Fields(seq.Header())[0]
          f, err := os.Create(td + "/" + fn + ".fasta")
          util.Check(err)
          fmt.Fprintf(f, "%s\n", seq)
          f.Close()
}
```

We import `fasta`, `strings`, `bytes`, and `os`.

40b ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁39b 40d▷
```
"github.com/evolbioinf/fasta"
"strings"
"bytes"
"os"
```

We run `phylonium` with the given number of threads and store its output. Note that we construct the paths of the input files using the `Glob` function of `filepath`, because direct submission of an expandable name doesn't work—we are in Go, not in Unix. Also, in the actual run of `phylonium` we ignore the error returned as this may be non-zero if the homology is low, but a distance was still returned. This is a bit risky, but still better than obscurely failing whenever a mildly divergent sequence is included in the analysis.

40c ⟨*Run phylonium, Pr. 4* 40c⟩≡ (38a)
```
cmd = exec.Command("phylonium")
nts := strconv.Itoa(nt)
args := []string{"phylonium", "-t", nts, "-r",
          td + "/"+ re + ".fasta"}
p, err := filepath.Glob(td + "/*.fasta")
util.Check(err)
args = append(args, p...)
cmd.Args = args
out, _ = cmd.Output()
```

We import `strconv` and `filepath`.

40d ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁40b 41b▷
```
"strconv"
"path/filepath"
```

We read the distance matrix or, failing that, give up.

41a ⟨*Read distance matrix, Pr. 4* 41a⟩≡ (38a)
```
var mat *dist.DistMat
r := bytes.NewReader(out)
scanner := dist.NewScanner(r)
if scanner.Scan() {
        mat = scanner.DistanceMatrix()
} else {
        log.Fatal("couldn't read distance matrix")
}
```

We import `dist`.

41b ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁40d 41e ▷
```
"github.com/evolbioinf/dist"
```

In order to correct the given classification, we first need a map between accessions and positions in the distance matrix. Using this map, we look up the index of the reference sequence, `ri`. Then we iterate over the false positive accessions and analyze each one. This analysis might uncover new true positives, which we store.

41c ⟨*Correct false positives, Pr. 4* 41c⟩≡ (38a)
```
accMap := make(map[string]int)
for i, name := range mat.Names {
        accMap[name] = i
}
ri := accMap[re]
ntp := make([]string, 0)
n := 0
accs := accessions["FalsePositives"]
for _, acc := range accs {
        ⟨Analyze false positive, Pr. 4 41d⟩
}
accessions["FalsePositives"] = accs[:n]
```

We look up the distance between the reference and the current accession, d_i. Comparison of d_i to the threshold distance, d_t, allows us to determine whether we are dealing with a "true" false positive, or a new true positive. Note that we classify a distance that is "not a number" as a distance that is always greater than the threshold.

41d ⟨*Analyze false positive, Pr. 4* 41d⟩≡ (41c)
```
j := accMap[acc]
di := mat.Matrix[ri][j]
if di > dt || math.IsNaN(di) {
        accs[n] = acc
        n++
} else {
        ntp = append(ntp, acc)
}
```

We import `math`.

41e ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁41b 43c ▷
```
"math"
```

Having correcte the false positives, we now analyze the false negatives, which may in fact be true negatives. But since we do not count true negatives, we don't store the reclassified accessions either.

42a ⟨*Correct false negatives, Pr. 4* 42a⟩≡ (38a)
```
n = 0
accs = accessions["FalseNegatives"]
for _, acc := range accs {
        ⟨Analyze false negative, Pr. 4 42b⟩
}
accessions["FalseNegatives"] = accs[:n]
```

The distance between the reference and the current accession allows us to distinguish between "true" false negatives and true negatives.

42b ⟨*Analyze false negative, Pr. 4* 42b⟩≡ (42a)
```
i := accMap[re]
j := accMap[acc]
di := mat.Matrix[i][j]
if di <= dt {
        accs[n] = acc
        n++
}
```

In our final correction step we analyze the accessions of the true positives, which may in fact be false positives, which we store. We also store and count the old true positives and after the loop reslice their storage accordingly.

42c ⟨*Correct true positives, Pr. 4* 42c⟩≡ (38a)
```
n = 0
accs = accessions["TruePositives"]
for _, acc := range accs {
        ⟨Analyze true positive, Pr. 4 42d⟩
}
accessions["TruePositives"] = accs[:n]
```

Depending on the distance to the reference, we classify the current accession as either a true positive or a new false positive.

42d ⟨*Analyze true positive, Pr. 4* 42d⟩≡ (42c)
```
j := accMap[acc]
di := mat.Matrix[ri][j]
if di <= dt {
        accs[n] = acc
        n++
} else {
        nfp = append(nfp, acc)
}
```

We construct slices of true positives, false positives, and false negative accessions. From the lengths of these slices we calculate the sensitivity and the specificity of our primer set. That suffices for printing the first three lines of our report, as shown in Figure 5.2. Then we print the corrected accessions.

43a ⟨*Print corrected report, Pr. 4* 43a⟩≡ (36d)
```
⟨Construct slices of accessions, Pr. 4 43b⟩
⟨Calculate sensitivity, Pr. 4 43d⟩
⟨Calculate specificity, Pr. 4 43e⟩
⟨Print first three lines of report, Pr. 4 43f⟩
⟨Print accessions, Pr. 4 44b⟩
```

We merge the approved true positives and the new positives into single slices. We also sort the accessions.

43b ⟨*Construct slices of accessions, Pr. 4* 43b⟩≡ (43a)
```
tps := accessions["TruePositives"]
tps = append(tps, ntp...)
sort.Strings(tps)
fps := accessions["FalsePositives"]
fps = append(fps, nfp...)
sort.Strings(fps)
fns := accessions["FalseNegatives"]
sort.Strings(fns)
```

We import `sort`.

43c ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁41e 44a▷
```
"sort"
```

Let t_p be the number of true positives and f_n the number of false negatives, then the sensitivity is defined as the fraction of taxa that should have been identified,

$$s_\mathrm{n} = \frac{t_\mathrm{p}}{t_\mathrm{p} + f_\mathrm{n}}.$$

43d ⟨*Calculate sensitivity, Pr. 4* 43d⟩≡ (43a)
```
tp := float64(len(tps))
fn := float64(len(fns))
sn := tp / (tp + fn)
```

Let also f_p be the number of false positives, then the specificity is defined as the fraction of true hits,

$$s_\mathrm{p} = \frac{t_\mathrm{p}}{t_\mathrm{p} + f_\mathrm{p}}.$$

43e ⟨*Calculate specificity, Pr. 4* 43e⟩≡ (43a)
```
fp := float64(len(fps))
sp := tp / (tp + fp)
```

The first three lines of the report consist of the primer set, the sensitivity, and the specificity.

43f ⟨*Print first three lines of report, Pr. 4* 43f⟩≡ (43a)
```
fmt.Printf("PrimerSet:\t%s\n", primerSet)
fmt.Printf("Sensitivity:\t%.3g\n", sn)
fmt.Printf("Specificity:\t%.3g\n", sp)
```

We import `fmt`.

44a ⟨*Imports, Pr. 4* 34a⟩+≡ (33a) ◁43c
```
"fmt"
```

We print the accessions of the true positives, the false positives, and the false negatives, in that order.

44b ⟨*Print accessions, Pr. 4* 44b⟩≡ (43a)
```
if len(tps) > 0 {
        ⟨Print true positives, Pr. 4 44c⟩
}
if len(fps) > 0 {
        ⟨Print false positives, Pr. 4 45c⟩
}
if len(fns) > 0 {
        ⟨Print false negatives, Pr. 4 45d⟩
}
```

We print the true positives either with distances or without. Both cases are reused for the false positives and the false negatives, so we delegate them to the functions `printAcc` and `printAccDist`, which we still need to write.

44c ⟨*Print true positives, Pr. 4* 44c⟩≡ (44b)
```
fmt.Print("TruePositives:\t")
if pd && truePos {
        printAccDist(tps, re, mat, accMap)
} else {
        printAcc(tps)
}
```

Inside `printAccDist` we take some time to prepare the printing, then print the first accession, followed by the rest.

44d ⟨*Functions, Pr. 4* 36d⟩+≡ (33a) ◁36d 45b▷
```
func printAccDist(accs []string, re string,
        mat *dist.DistMat,
        accMap map[string]int) {
        ⟨Prepare printing, Pr. 4 44e⟩
        ⟨Print first accession, Pr. 4 44f⟩
        ⟨Print remaining accessions, Pr. 4 45a⟩
}
```

By way of preparation, we isolate the distance matrix and the index of the reference strain in that matrix.

44e ⟨*Prepare printing, Pr. 4* 44e⟩≡ (44d)
```
dists := mat.Matrix
i := accMap[re]
```

We print the first accession and distance delimited by a blank.

44f ⟨*Print first accession, Pr. 4* 44f⟩≡ (44d)
```
acc := accs[0]
j := accMap[acc]
d := dists[i][j]
fmt.Printf("%s %.3g", acc, d)
```

We iterate over the remaining accessions and print them together with the distances, keeping to the blank-delimited format. We end the line with a carriage return.

45a ⟨*Print remaining accessions, Pr. 4* 45a⟩≡ (44d)
```
accs = accs[1:]
for _, acc := range accs {
          j := accMap[acc]
          d := dists[i][j]
          fmt.Printf(" %s %.3g", acc, d)
}
fmt.Printf("\n")
```

Printing just accessions is simpler, so inside `printAcc` we print the first accession without a leading blank and then the rest with.

45b ⟨*Functions, Pr. 4* 36d⟩+≡ (33a) ◁44d
```
func printAcc(accs []string) {
          fmt.Printf("%s", accs[0])
          accs = accs[1:]
          for _, acc := range accs {
                  fmt.Printf(" %s", acc)
          }
          fmt.Printf("\n")
}
```

Using the functions `printAcc` and `printAccDist` we also print the false positives.

45c ⟨*Print false positives, Pr. 4* 45c⟩≡ (44b)
```
fmt.Print("FalsePositives:\t")
if pd {
          printAccDist(fps, re, mat, accMap)
} else {
          printAcc(fps)
}
```

Finally, we print the false negatives.

45d ⟨*Print false negatives, Pr. 4* 45d⟩≡ (44b)
```
fmt.Print("FalseNegatives:\t")
if pd {
          printAccDist(fns, re, mat, accMap)
} else {
          printAcc(fns)
}
```

We have finished `cops`, let's test it.

Testing

Our code for testing `cops` contains hooks for imports and the testing logic.

46a ⟨*cops_test.go* 46a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Pr. 4 46c⟩
)

func TestCops(t *testing.T) {
        ⟨Testing, Pr. 4 46b⟩
}
```

We construct a set of tests and iterate over them.

46b ⟨*Testing, Pr. 4* 46b⟩≡ (46a)
```
var tests []*exec.Cmd
⟨Construct tests, Pr. 4 46d⟩
for i, test := range tests {
        ⟨Run test, Pr. 4 47a⟩
}
```

We import `exec`.

46c ⟨*Testing imports, Pr. 4* 46c⟩≡ (46a) 47b ▷
```
"os/exec"
```

Our tests run on the sample database, with reference strain EDL933, which has accession AE005174, threshold distance 0.002, and input `scop.out`. The first test takes only default parameters. The second test also returns the distances. The third and last test returns distances and includes the true positives.

46d ⟨*Construct tests, Pr. 4* 46d⟩≡ (46b)
```
d := "../data/sample"
r := "AE005174"
x := "2e-3"
i := "scop.out"
test := exec.Command("./cops", "-d", d, "-r", r, "-t", x, i)
tests = append(tests, test)
test = exec.Command("./cops", "-d", d, "-r", r, "-t", x,
        "-D", i)
tests = append(tests, test)
test = exec.Command("./cops", "-d", d, "-r", r, "-t", x,
        "-D", "-p", i)
tests = append(tests, test)
```

When running a test, we compare the result we get with the result we want, which is contained in files `r1.txt`, `r2.txt`, and `r3.txt`.

47a ⟨*Run test, Pr. 4* 47a⟩≡ (46b)

```
get, err := test.Output()
if err != nil {
          t.Error(err)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := os.ReadFile(f)
if err != nil {
          t.Error(err)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `os`, and `bytes`.

47b ⟨*Testing imports, Pr. 4* 46c⟩+≡ (46a) ◁46c

```
"strconv"
"os"
"bytes"
```

Chapter 6

Package `util`

6.1 util

The package util collects utility functions. Its outline provides hooks for imports, variables, and functions.

Package 1 (util)

49a ⟨*util.go* 49a⟩≡

```
// Package util provides utility functions for the programs
// indexNeighbors and neighbors.
package util

import (
        ⟨Imports, Pa. 1 49d⟩
)
⟨Variables, Pa. 1 49c⟩
⟨Functions, Pa. 1 49b⟩
```

6.1.1 PrintInfo

PrintInfo prints program information and exits.

49b ⟨*Functions, Pa. 1* 49b⟩≡ (49a) 50a▷

```
func PrintInfo(program string) {
        author := "Bernhard Haubold"
        email := "haubold@evolbio.mpg.de"
        license := "Gnu General Public License, " +
                "https://www.gnu.org/licenses/gpl.html"
        clio.PrintInfo(program, version, date,
                author, email, license)
        os.Exit(0)
}
```

We declare the variables version and date, which ought to be injected at compile time.

49c ⟨*Variables, Pa. 1* 49c⟩≡ (49a) 50f▷

```
var version, date string
```

We import clio and os.

49d ⟨*Imports, Pa. 1* 49d⟩≡ (49a) 50b▷

```
"github.com/evolbioinf/clio"
"os"
```

6.1.2 Open

Open opens a file with error checking.

50a ⟨*Functions, Pa. 1* 49b⟩+≡ (49a) ◁49b 50c▷

```
func Open(file string) *os.File {
        f, err := os.Open(file)
        if err != nil {
                fmt.Fprintf(os.Stderr, "couldn't open %s\n", file)
                os.Exit(1)
        }
        return f
}
```

We import fmt and os.

50b ⟨*Imports, Pa. 1* 49d⟩+≡ (49a) ◁49d 50d▷

```
"fmt"
"os"
```

Check

Check checks an error and aborts if it isn't nil.

50c ⟨*Functions, Pa. 1* 49b⟩+≡ (49a) ◁50a 50e▷

```
func Check(err error) {
        if err != nil {
                log.Fatal(err)
        }
}
```

We import log.

50d ⟨*Imports, Pa. 1* 49d⟩+≡ (49a) ◁50b

```
"log"
```

6.1.3 SetName

The function SetName sets the name of the program. It stores the name in a global variable and prepares the log package to print that name in the event of an error message.

50e ⟨*Functions, Pa. 1* 49b⟩+≡ (49a) ◁50c 51a▷

```
func SetName(n string) {
        name = n
        s := fmt.Sprintf("%s: ", n)
        log.SetPrefix(s)
        log.SetFlags(0)
}
```

We declare the global string variable name.

50f ⟨*Variables, Pa. 1* 49c⟩+≡ (49a) ◁49c

```
var name string
```

6.1.4 `Version`

The function `Version` prints the version and other information about the program and exits. `Version` simply wraps a call to `PrintInfo`.

51a ⟨*Functions, Pa. 1* 49b⟩+≡ (49a) ◁50e
```
func Version() {
        PrintInfo(name)
}
```

We are done with the `util` package, time to test it.

6.1.5 Testing

Our testing code for `util` contains hooks for imports and the logic of the testing function.

51b ⟨*util_test.go* 51b⟩≡
```
package util

import (
        "testing"
        ⟨Testing imports, Pa. 1 51d⟩
)

func TestUtil(t *testing.T) {
        ⟨Testing, Pa. 1 51c⟩
}
```

There is only one function we can sensibly test, `Open`. So we open a test file and read the string "success" from it.

51c ⟨*Testing, Pa. 1* 51c⟩≡ (51b)
```
f := Open("r.txt")
defer f.Close()
sc := bufio.NewScanner(f)
if !sc.Scan() {
        t.Error("scan failed")
}
get := sc.Text()
want := "success"
if get != want {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `bufio`.

51d ⟨*Testing imports, Pa. 1* 51d⟩≡ (51b)
```
"bufio"
```

Chapter 7

Tutorial

Find Primers

The file `tutorial/template.fasta` contains two short template sequences deemed diagnostic for *Escherichia coli* serovar O157:H7. We convert the sequences to input for `primer3`.

53a ⟨*tut.sh* 53a⟩≡ 53b ▷
```
fa2prim template.fasta > prim.in
```

We construct primers using `primer3`.

53b ⟨*tut.sh* 53a⟩+≡ ◁53a 53c ▷
```
primer3_core prim.in > prim.out
```

We extract the primers into a table.

53c ⟨*tut.sh* 53a⟩+≡ ◁53b 53d ▷
```
prim2tab prim.out
```

```
# Penalty  Forward                Reverse               Internal
1.320714   CGGCAATATCATAGACATCGT   AACGATTATGTGTGGGCAAA  TCACGACGATAATTATCTTT
1.592652   TCGGCAATATCATAGACATCG   AACGATTATGTGTGGGCAAA  TCACGACGATAATTATCTTT
1.826015   TCGGCAATATCATAGACATCG   ACGATTATGTGTGGGCAAAT  TCACGACGATAATTATCTTT
2.081523   ATCGGCAATATCATAGACATCG  AACGATTATGTGTGGGCAAA  TCACGACGATAATTATCTTT
2.194407   CGGCAATATCATAGACATCGT   TAACGATTATGTGTGGGCAA  TCACGACGATAATTATCTTT
1.181309   GTAGTATCAGAAGAGAACGCG   AGTATTGGTTGTCAGGAGCT  CTAGTCCATAAGCAAGAAAA
1.682847   GTAGTATCAGAAGAGAACGCG   AAGTATTGGTTGTCAGGAGC  CTAGTCCATAAGCAAGAAAA
2.164691   TAGTATCAGAAGAGAACGCG    AAGTATTGGTTGTCAGGAGC  CTAGTCCATAAGCAAGAAAA
2.213344   TGTAGTATCAGAAGAGAACGC   GTATTGGTTGTCAGGAGCTG  CTAGTCCATAAGCAAGAAAA
2.222044   TGTAGTATCAGAAGAGAACGC   AGTATTGGTTGTCAGGAGCT  CTAGTCCATAAGCAAGAAAA
```

We sort the table by the penalty and pick the forward and reverse primer with the lowest penalty. (We ignore the internal oligo in this Tutorial.)

53d ⟨*tut.sh* 53a⟩+≡ ◁53c 53e ▷
```
prim2tab prim.out |
    tail -n +2 |
    sort -n |
    head -n 1 |
    awk '{printf ">f\n%s\n>r\n%s\n", $2, $3}'
```

```
>f
GTAGTATCAGAAGAGAACGCG
>r
AGTATTGGTTGTCAGGAGCT
```

We save the primer pair to a file.

53e ⟨*tut.sh* 53a⟩+≡ ◁53d 54a ▷
```
prim2tab prim.out |
    tail -n +2 |
    sort -n |
    head -n 1 |
    awk '{printf ">f\n%s\n>r\n%s\n", $2, $3}' > prim.fasta
```

We are done designing the primers. Of course, we could have carried out the above steps all in one short pipeline.

54a ⟨*tut.sh* 53a⟩+≡ ◁53e 54b▷

```
fa2prim template.fasta |
    primer3_core |
    prim2tab |
    tail -n +2 |
    sort -n |
    head -n 1 |
    awk '{printf ">f\n%s\n>r\n%s\n", $2, $3}' > prim.fasta
```

Test Primers

In a production setting, we would test a set of primers by running them against a large DNA sequence database, for example, the non-redundant collection of nucleotide sequences provided by the NCBI. However, for the purposes of this Tutorial, we work with an abridged database, which we download into the `data` folder and unpack.

54b ⟨*tut.sh* 53a⟩+≡ ◁54a 54c▷

```
wget guanine.evolbio.mpg.de/prim/sample.tgz
mv sample.tgz ../data
cd ../data
tar -xvzf sample.tgz
cd ../tutorial
```

We score our primers using the program `scop`. It calculates the sensitivity and specificity of our primers based on the accuracy with which they amplify the target taxa listed by their taxon IDs in the file `tarTax.txt`. These taxa all belong to *E. coli* O157:H7 and were obtained using `neighbors`[1]. However, they could also be obtained using the webbrowser of the NCBI taxonomy[2]. We find no false negatives, so our sensitivity is maximal. However, there appear to be more false positives than true positives, hence our specificity is a rather low, 0.479.

54c ⟨*tut.sh* 53a⟩+≡ ◁54b 54d▷

```
scop -d ../data/sample -t tarTax.txt prim.fasta
```

```
PrimerSet:      prim.fasta
Sensitivity:    1
Specificity:    0.479
TruePositives:  AP018488.1 BA000007.3 CP001164.1...
FalsePositives: AE005174.2 AP026080.1 AP026082.1...
```

We save the output of `scop` to a file.

54d ⟨*tut.sh* 53a⟩+≡ ◁54c 55a▷

```
scop -d ../data/sample -t tarTax.txt prim.fasta > scop.out
```

[1] github.com/evolbioinf/neighbors
[2] https://www.ncbi.nlm.nih.gov/taxonomy

We check the false positives returned by calculating their distances to the type strain of O157:H7, EDL933, which has accession CP008957. This distance between CP008957 and any false positive—or false negative, if there were any—is compared to a threshold, which we need to pick. Here we use twice the branch length from AE005174 to the parent of the target clade, 0.0018. This is calculated with another program from the `neighbors` package, `climt`. Again, users are free to pick any distance they see fit. If the distance to a false positive is less than the threshold, it is reclassified as a true positive. Similarly, if the distance to a (hypothetical) false negative is greater than the threshold, it is reclassified as a true negative. Based on these distances, there are now only three false positives leading to the much better corrected specificity of 0.991.

55a ⟨*tut.sh* 53a⟩+≡ ◁54d 55b▷

```
cops -d ../data/sample -r CP008957 -t 1.8e-3 scop.out
```

```
PrimerSet:      prim.fasta
Sensitivity:    1
Specificity:    0.991
TruePositives:  AE005174.2 AP018488.1 AP026080.1...
FalsePositives: CP057173.1 CP057250.1 CP084534.1
```

The program `cops` can also return the distances to the reference, which are greater 1.5% for the three false positives. In other words, these distances are far removed from the threshold.

55b ⟨*tut.sh* 53a⟩+≡ ◁55a 55c▷

```
cops -d ../data/sample -r CP008957 -t 1.8e-3 -D scop.out
```

```
PrimerSet:      prim.fasta
Sensitivity:    1
Specificity:    0.991
TruePositives:  AE005174.2 AP018488.1 AP026080.1...
FalsePositives: CP057173.1 0.0157 CP057250.1 0.0158 CP084534.1 0.0158
```

If we are also interested in the distances among the true positives, we can rerun `cops` with the `-p` switch for also checking the true positives. This slows down the run, but shows that the true positives are also safely removed from the threshold. So we are not dealing with classifications that would change had we picked a slightly different threshold.

55c ⟨*tut.sh* 53a⟩+≡ ◁55b

```
cops -d ../data/sample -r CP008957 -t 1.8e-3 -D -p scop.out
```

```
PrimerSet:      prim.fasta
Sensitivity:    1
Specificity:    0.991
TruePositives:  AE005174.2 2.18e-05 AP018488.1 0.000654 AP026080.1 0.000307...
FalsePositives: CP057173.1 0.0157 CP057250.1 0.0158 CP084534.1 0.0158
```

Bibliography

[1] A. Untergasser, I. Cutcutache, T. Koressaar, J. Ye, B. C. Faircloth, M. Remm, and S. G. Rozen. Primer3—new capabilities and interfaces. *Nucleic Acids Research*, 40:e115, 2012.