Design Microarray Probes

Erik S. Wright

October 24, 2023

Contents

1 Introduction

This document describes how to design and validate sequence-specific probes for synthesis onto a DNA microarray. As a case study, this tutorial focuses on the development of a microarray to identify taxonomic groups based on previously obtained 16S ribosomal RNA sequences. The same approach could be applied to differentiate sequences representing any number of groups based on any shared region of DNA. The objective of microarray probe design is straightforward: to determine a set of probes that will bind to one group of sequences (the target consensus sequence) but no others (the non-targets). Beginning with a set of aligned DNA sequences, the program chooses the best set of probes for targeting each consensus sequence. More importantly, the design algorithm is able to predict when potential cross-hybridization of the probes may occur to non-target sequence(s). An integrated design approach enables characterizing the probe set before fabrication, and then assists with analysis of the experimental results.

2 Getting Started

2.1 Startup

To get started we need to load the DECIPHER package, which automatically loads several other required packages.

```
> library(DECIPHER)
```

Help for the `DesignArray` function can be accessed through:

```
> ? DesignArray
```

If DECIPHER is installed on your system, the code in each example can be obtained via:

```
> browseVignettes("DECIPHER")
```

2.2 Creating a Sequence Database

We begin with a set of aligned sequences belonging to the 16S rRNA of several samples obtained from drinking water distribution systems. Be sure to change the path names to those on your system by replacing all of the text inside quotes labeled "<<path to ...>>" with the actual path on your system.

```
> # specify the path to your sequence file:
> fas <- "<<path to FASTA file>>"
> # OR find the example sequence file used in this tutorial:
> fas <- system.file("extdata", "Bacteria_175seqs.fas", package="DECIPHER")
```

Next, there are two options for importing the sequences into a database: either save a database file or maintain the database in memory. Here we will build the database in memory because it is a small set of sequences and we do not intend to use the database later:

```
> # specify a path for where to write the sequence database
> dbConn <- "<<path to write sequence database>>"
> # OR create the sequence database in memory
> dbConn <- dbConnect(SQLite(), ":memory:")
> Seqs2DB(fas, "FASTA", dbConn, "uncultured bacterium")
Reading FASTA file chunk 1

175 total sequences in table Seqs.
Time difference of 0.04 secs
```

2.3 Defining Groups

At this point we need to define groups of related sequences in the database we just created. In this case we wish to cluster the sequences into groups of at most 3% distance between sequences.

```
> dna <- SearchDB(dbConn)
Search Expression:
select row_names, sequence from _Seqs where row_names in (select row_names
from Seqs)

DNAStringSet of length: 175
Time difference of 0.01 secs
> dMatrix <- DistanceMatrix(dna, verbose=FALSE)
> clusters <- TreeLine(myDistMatrix=dMatrix, type="clusters", cutoff=0.03, method="compl
> Add2DB(clusters, dbConn, verbose=FALSE)
```

Now that we have identified 100 operational taxonomic units (OTUs), we must form a set of consensus sequences that represent each OTU.

```
> conSeqs <- IdConsensus(dbConn, colName="cluster", verbose=FALSE)
> dbDisconnect(dbConn)
> # name the sequences by their cluster number
> ns <- lapply(strsplit(names(conSeqs), "_", fixed=TRUE), `[`, 1)
> names(conSeqs) <- gsub("cluster", "", unlist(ns), fixed=TRUE)
> # order the sequences by their cluster number
> o <- order(as.numeric(names(conSeqs)))
> conSeqs <- conSeqs[o]
```

3 Array Design and Validation Steps

3.1 Designing the Probe Set

Next we will design the optimal set of 20 probes for targeting each OTU. Since there are 100 OTUs, this process will result in a set of 2,000 probes. By default, probes are designed to have the optimal length for hybridization at 46°C and 10% (vol/vol) formamide. We wish to allow up to 2 permutations for each probe, which will potentially require more space on the microarray. Since not all of the sequences span the alignment, we will design probes between alignment positions 120 and 1,450, which is the region encompassed by most of the sequences.

```
> probes <- DesignArray(conSeqs, maxPermutations=2, numProbes=20,
          start=120, end=1450, verbose=FALSE)
> dim(probes)
[1] 2000   12
> names(probes)
 [1] "name"          "start"         "length"        "start_aligned"
 [5] "end_aligned"   "permutations"  "score"         "formamide"
 [9] "hyb_eff"       "target_site"   "probes"        "mismatches"
```

We can see the probe sequence, target site positioning, melt point (formamide), and predicted cross-hybridization efficiency to non-targets (mismatches). The first probe targeting the first consensus sequence (OTU #1) is predicted to have 84% hybridization efficiency at the formamide concentration used in the experiment (10%). This probe is also predicted to cross-hybridize with OTU #5 with 59% hybridization efficiency.

```
> probes[1,]
  name start length start_aligned end_aligned permutations        score
1    1     1     22           120         143            1 22.75502....
     formamide      hyb_eff            target_site
1 16.06832.... 84.26072.... GCATCGGAACGTGTCCTAAAGT
                                       probes mismatches
1 ACTTTAGGACACGTTCCGATGCTTTTTTTTTTTTTTTTTT  5 (59.4%)
```

If we wished to have the probe set synthesized onto a microarray, all we would need is the unique set of probes. Note that the predictive model was calibrated using NimbleGen microarrays. Although predictions are likely similar for other microarray platforms, hybridization conditions should always be experimentally optimized.

```
> u <- unique(unlist(strsplit(probes$probes, ",", fixed=TRUE)))
> length(u)
[1] 1899
> head(u)
```

```
[1] "ACTTTAGGACACGTTCCGATGCTTTTTTTTTTTTTTTTTTT"
[2] "GTATTAGCGCATCTTTCGATGCTTTTTTTTTTTTTTTTTTT"
[3] "GTCTTTCGATCCCCTACTTTCCTCTTTTTTTTTTTTTTTTTTT"
[4] "GGCCGCTCCAAAAGCATAAGGTTTTTTTTTTTTTTTTTTT"
[5] "ATGGCAATTAATGACAAGGGTTGCTTTTTTTTTTTTTTTTTTT"
[6] "CAGTGTGGTTGGCCATCCTCTTTTTTTTTTTTTTTTTTT"
```

3.2 Validating the Probe Set

Before fabrication onto a DNA microarray it may be useful to predict whether the probe set will adequately discriminate between the OTUs. This can be accomplished by simulating the hybridization process multiple times while incorporating error. We begin by converting the predicted cross-hybridization efficiencies into a sparse matrix that mathematically represents the microarray (**A**). Here the rows of the matrix represent each probe, and the columns of the matrix represent each OTU. The entries of the matrix therefore give the hybridization efficiency of probe *i* with OTU *j*. We can neglect all hybridization efficiencies less than 5% because these will likely not hybridize or have insufficient brightness.

```
> A <- Array2Matrix(probes, verbose=FALSE)
> w <- which(A$x < 0.05)
> if (length(w) > 0) {
          A$i <- A$i[-w]
          A$j <- A$j[-w]
          A$x <- A$x[-w]
 }
```

We then multiply the matrix **A** by the amount (x) of each OTU present to determine the corresponding brightness (b) values of each probe. We can add a heteroskedastic error to the brightness values to result in a more accurate simulation (b = **A**x + error). Furthermore, we can introduce a 5% rate of probes that hybridize randomly.

```
> # simulate the case where 10% of the OTUs are present in random amounts
> present <- sample(length(conSeqs), floor(0.1*length(conSeqs)))
> x <- numeric(length(conSeqs))
> x[present] <- abs(rnorm(length(present), sd=2))
> # determine the predicted probe brightnesses based on the present OTUS
> background <- 0.2
> b <- matrix(tapply(A$x[A$j]*x[A$j], A$i, sum), ncol=1) + background
> b <- b + rnorm(length(b), sd=0.2*b) # add 20% error
> b <- b - background # background subtracted brightnesses
> # add in a 5% false hybridization rate
> bad_hybs <- sample(length(b), floor(0.05*length(b)))
> b[bad_hybs] <- abs(rnorm(length(bad_hybs), sd=max(b)/3))
```

Finally, we can solve for the amount of each OTU present on the microarray by solving **A**x = b for x using non-negative ($x \geq 0$) least squares. Plotting the expected amount versus the predicted amount shows that this probe set may result in a small number of false positives and false negatives (Fig. 1). False negatives are the expected observations below the dashed threshold line, which represents the minimum amount required to be considered present. If this threshold is lowered then false negatives will appear where no amount was expected.

```
> # solve for the predicted amount of each OTU present on the array
> x_out <- NNLS(A, b, verbose=FALSE)
```

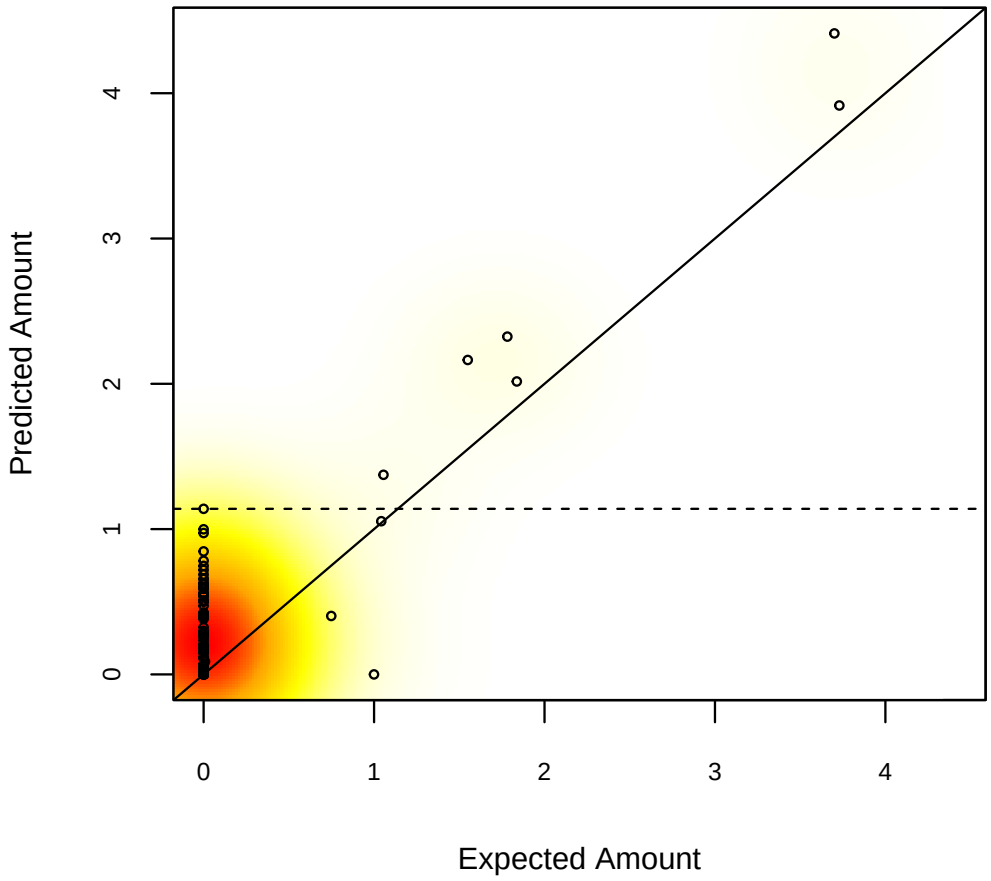

Figure 1: Characterization of predicted specificity for the designed probe set.

3.3 Further Improving the Result

Least squares regression is particularly sensitive outlier observations and heteroskedastic noise. For this reason we will decrease the effects of outlier observations by using weighted regression. With each iteration the weights will be refined using the residuals from the prior solution to **A**x = b.

```
> # initialize weights to one:
> weights <- matrix(1, nrow=nrow(b), ncol=ncol(b))
> # iteratively unweight observations with high residuals:
> for (i in 1:10) { # 10 iterations
          weights <- weights*exp(-0.1*abs(x_out$residuals))
          A_weighted <- A
          A_weighted$x <- A$x*weights[A$i]
          b_weighted <- b*weights
          x_out <- NNLS(A_weighted, b_weighted, verbose=FALSE)
 }
```

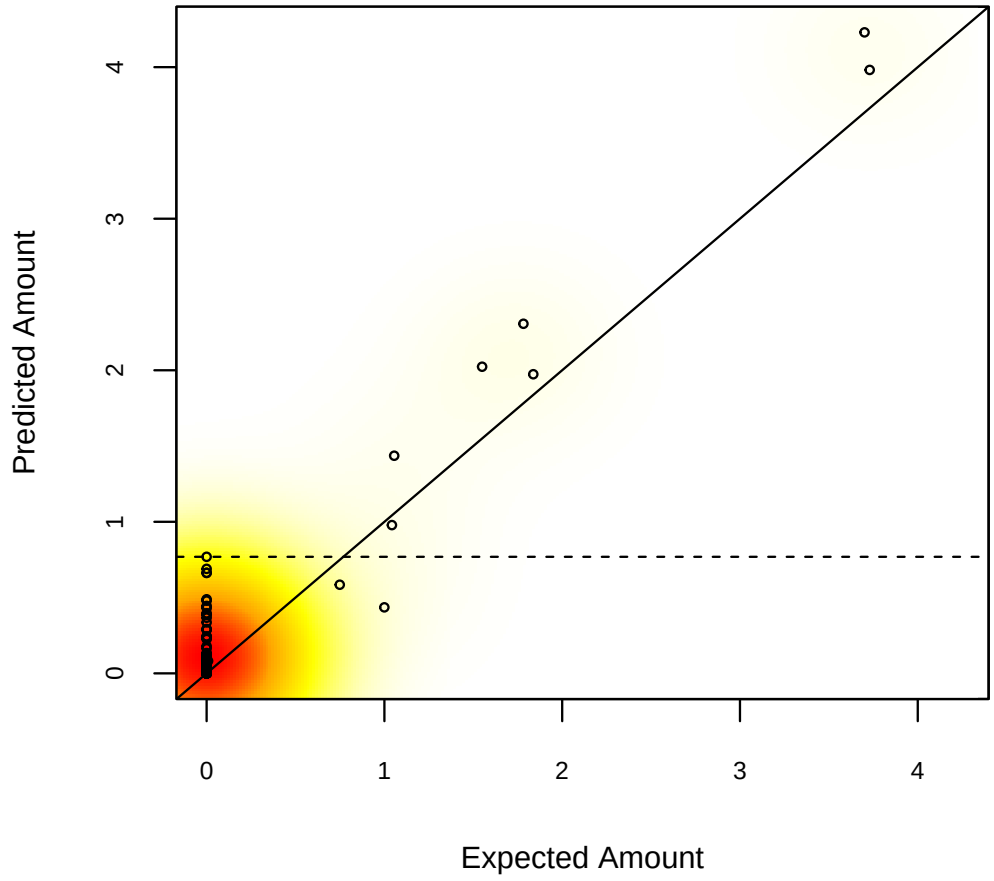

Figure 2: Improved specificity obtained by down-weighting the outliers.

Weighted regression lowered the threshold for detection so that more OTUs would be detectable (Fig. 2). However, false negatives still remain based on this simulation when a very small amount is expected. If the threshold is lowered to capture all of the expected OTUs then we can determine the false positive(s) that would result. These false positive sequences are substantially different from the nearest sequence that is present.

```
> w <- which(x_out$x >= min(x_out$x[present]))
> w <- w[-match(present, w)] # false positives
> dMatrix <- DistanceMatrix(conSeqs, verbose=FALSE)
> # print distances of false positives to the nearest present OTU
> for (i in w)
         print(min(dMatrix[i, present]))
[1] 0.2164179
[1] 0.006102212
[1] 0.0624524
[1] 0.1066879
[1] 0.07205067
[1] 0.1194969
[1] 0.2252788
[1] 0.1495253
[1] 0.1168048
[1] 0.1265823
[1] 0.07238395
[1] 0.1078869
[1] 0.1336982
[1] 0.1992424
[1] 0.0229794
[1] 0.2022556
[1] 0.1093394
[1] 0.03770739
[1] 0.1531599
[1] 0.05242868
[1] 0.09939759
[1] 0.06175772
[1] 0.1
[1] 0.09124629
[1] 0.08314607
[1] 0.1171384
[1] 0.1970037
[1] 0.1157654
[1] 0.1360595
[1] 0.1381173
[1] 0.1531176
[1] 0.1557312
[1] 0.1441308
[1] 0.1628615
[1] 0.1238938
[1] 0.1338403
[1] 0.2076176
[1] 0.2048012
```

3.4 Extending the Simulation

The above simulation can be repeated multiple times and with different initial conditions to better approximate the expected number of false positives and false negatives (Fig. 3). In the same manner the design parameters can be iteratively optimized to further improve the predicted specificity of the probe set based on the simulation results. After

fabrication, validation experiments using known samples should be used in replace of the simulated brightness values.

```
> # simulate multiple cases where 10% of the OTUs are present in random amounts
> iterations <- 100
> b <- matrix(0, nrow=dim(b)[1], ncol=iterations)
> x <- matrix(0, nrow=length(conSeqs), ncol=iterations)
> for (i in 1:iterations) {
          present <- sample(length(conSeqs), floor(0.1*length(conSeqs)))
          x[present, i] <- abs(rnorm(length(present), sd=2))

          # determine the predicted probe brightnesses based on the present OTUS
          b[, i] <- tapply(A$x[A$j]*x[A$j, i], A$i, sum) + background
          b[, i] <- b[, i] + rnorm(dim(b)[1], sd=0.2*b[, i]) # add 20% error
          b[, i] <- b[, i] - background # background subtracted brightnesses

          # add in a 5% false hybridization rate
          bad_hybs <- sample(dim(b)[1], floor(0.05*length(b[, i])))
          b[bad_hybs, i] <- abs(rnorm(length(bad_hybs), sd=max(b[, i])/3))
 }
> x_out <- NNLS(A, b, verbose=FALSE)
```

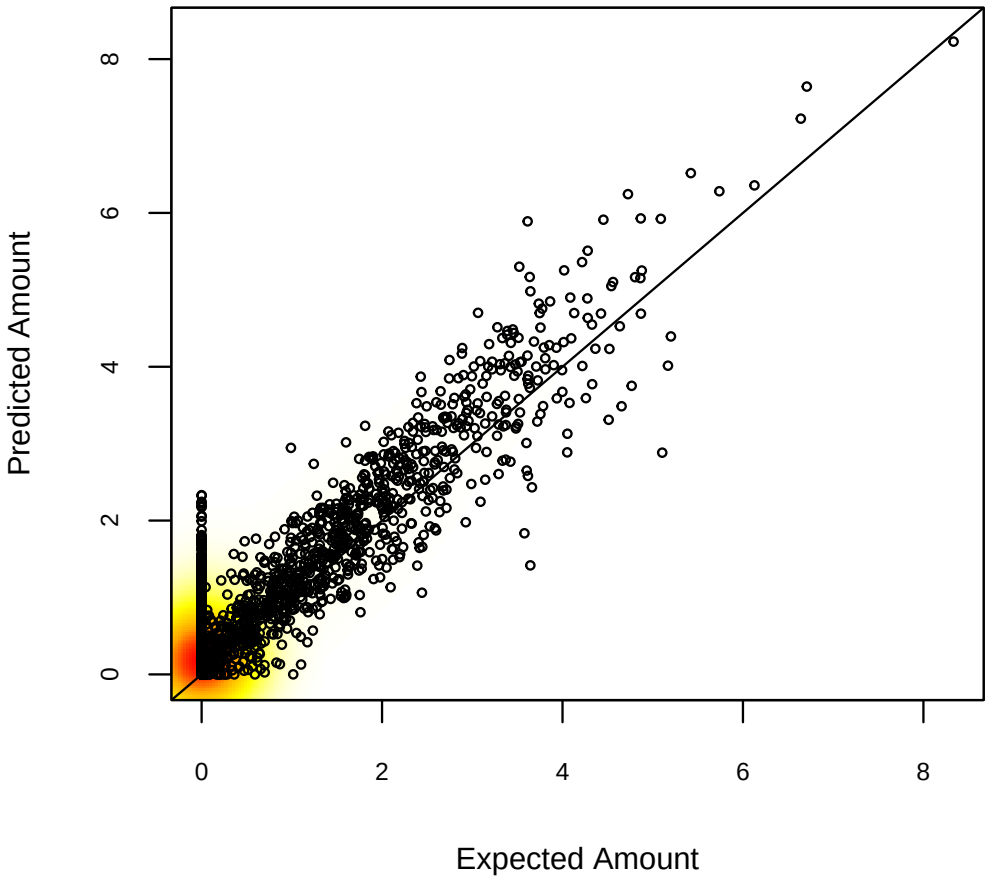

Figure 3: The combined results of multiple simulations.

4 Session Information

All of the output in this vignette was produced under the following conditions:

- R version 4.3.1 Patched (2023-06-17 r84564), `x86_64-apple-darwin20`
- Running under: `macOS Monterey 12.6.5`
- Matrix products: default
- BLAS: `/Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRblas.0.dylib`
- LAPACK: `/Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRlapack.dylib` ; LAPACK version3.11.0
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
- Other packages: BiocGenerics 0.48.0, Biostrings 2.70.0, DECIPHER 2.30.0, GenomeInfoDb 1.38.0, IRanges 2.36.0, RSQLite 2.3.1, S4Vectors 0.40.0, XVector 0.42.0
- Loaded via a namespace (and not attached): DBI 1.1.3, GenomeInfoDbData 1.2.11, KernSmooth 2.23-22, RCurl 1.98-1.12, bit 4.0.5, bit64 4.0.5, bitops 1.0-7, blob 1.2.4, cachem 1.0.8, cli 3.6.1, compiler 4.3.1, crayon 1.5.2, fastmap 1.1.1, memoise 2.0.1, pkgconfig 2.0.3, rlang 1.1.1, tools 4.3.1, vctrs 0.6.4, zlibbioc 1.48.0