iParameterEstimation User Guide

Bob Zimmermann

Washington University in St. Louis
Box 1045
One Brookings Drive
St. Louis, MO 63130

Preface

Welcome to iParameterEstimation! I hope this program is useful to you and your lab. It will take some getting used to, and I hope I can guide you through the toughest points here.

At writing time, the software is just barely released. Hopefully there aren't any discrepancies between the current version and the one used at writing. In general, you can feel free to contact me at rpz@cse.wustl.edu if you are having problems with the program. I'll try and respond as soon as I can, but I am no longer in the country or working on this, so there might be an issue.

iParameterEstimation is a maximum likelihood estimator for generalized hidden Markov Models. Expectation maximization is not implemented, and will probably never be implemented.[1]

You may find many things odd and idiosyncratic about iParameterEstimation. The goal of the project when we started was not only to tackle estimating parameters for N-SCAN or Twinscan, but to create the basis a reliable tool for analysis that could ostensibly be extended for any system at all using a generalized hidden Markov model. This may or may not be practical at this point, but it certainly is a possibility if anyone became interested.

You are probably reading this for the sake of retraining N-SCAN to run on your favorite organism. Why is there not a straight, simple way to do this? There are a few reasons, and I'm hoping that in the end learning from this guide will be somewhat rewarding, anyway.

1. **Parameter estimation isn't that easy anyway.** There is a reason this code base is behemouth, and it is because making it easy isn't easy. Brent lab has written several programs over the years to estimate parameters and have had to rewrite every time the model has changed, essentially. I'm trying to put an end to this.

2. **Parameter estimation has to be done well.** A lot of care was taken into making the counting correct and complete for all input training ex-

[1] This is because the maximization step is the only step iParameterEstimation would have to offer as a component to Twinscan or N-SCAN, and this step is really trivial with respect to the rest of the package, and the details of implementing the forward-backward algorithm. I would recommend that you implement it in the predictor itself, as that will be faster and involve less bookkeeping. iParameterEstimation would probably serve as a good starting point for seeding your parameters. For a good reference on EM in HMMs see [11], [10], and [4].

amples. This also leads to extending the code without having to actually reimplement parts of it that would be otherwise painful and error-prone. Through experience in Brent lab, several mistakes have been made in the past, and I've corrected a lot of them.

3. **Genscan/Twinscan/N-SCAN doesn't use a "pure" hidden Markov model.** You'll find several "hacks" if you dig deep enough into usage. There are several differences between the "pure" generalized hidden Markov model and the one that we use for gene prediction. In explicitly requesting all these differences, we have a transparent understanding of how the parameters are estimated as well as an avenue to change and experiment with them.

4. **Parameter estimation should be flexible.** I've tried to include as much extensibility and flexibility on all levels as possible. This isn't really easy or fully realizable. If I wanted it to be fast, I would have written it in C. If I wanted it to have short, simple inputs, I would have used my own, non-XML parser. The emphasis is on flexibility and extensibility, while maintaining a high level of accuracy, reliability and user-friendliness. This motivated a lot of the design choices.

Hopefully you aren't already disillusioned.

The writing, if you haven't already detected, is highly informal, and will not contain too many mathematical formuale or any other formalisms. The idea is to get you going with parameter estimation, hopefully give you a notion of what to expect, and also give ME time to do other things.

The contents may be incomplete, and of course, if you see a problem, come up with a correction and send it to me! Or just complain, and I may listen.

I hope this is helpful to you. Good luck!

BZ

Contents

Chapter 1

Introduction

Why is this book so long, and why are you reading it? Good question. Parameter estimation tools are commonly simple and terse, and designed for specific purposes. Our lab has a long-term commitment to gene prediction, so I have built a parameter estimation tool that will adapt to present and future uses. iParameterEstimation was designed for estimating parameters for anything involving sequences with annotations, on any model which vaguely resembels a hidden Markov model, although it can be used for anything at all. Here's an introduction to the key ideas of this project.

1.1 What is Parameter Estimation?

Broadly stated, parameter estimation is guessing the most likely numbers that model a phenomenon. For example, if you were trying to find out the probability of rolling a 1 with a single die, you might guess that the probability is one in six. The actual probability of this event occurring might depend on the die itself, how it is rolled, and where it is rolled. So you might decide to gather information about rolls under certain conditions, for example rolled by your friend Bob on your coffee table from about ten inches above the table. Then you could count the total number of times he rolls a one, and get an estimation of the true frequency of rolling ones.

Even then, there might be some other factors biasing the rolls. In essence, we do not have complete information about how the rolls are being generated, and thus don't know *a priori* the true parameterization of this model. But we *do* have a good idea of how this model looks. We know for certain that the die will probably roll onto the table in some way in which the roller cannot control. And we thus have a good idea that there is very little relationship between consecutive rolls.

This resembles the problem of understanding how to model biological phenomena, now doesn't it? We don't have complete information, but we make guesses about it and generalizations about the mainstream events, and form a

model on which to estimate parameters.

The model for our die-rolling event is simple, however estimating parameters for large data sets on complex models such as these is unweildy, tedious and largely uninteresting. This contributes to a large amount of mistakes in the forms little bugs in uncommented Perl scripts and C code. This has produced a lot of agony and distaste for parameter estimation, and for many, the entire process is a complete mystery.

1.2 What is iParameterEstimation?

iParameterEstimation is an attempt at deconvoluting parameter estimation. The parameter estimation scripts in Brent lab, to date, have been largely inaccessible. iParameterEstimation tries to fix this by not only making the process clean and correct, but also modifyable and extensible.

The art of estimating parameters not only implies counting, but also model design. No amount of additional estimation with new sequence from the same species can make a notable improvement to your model. Models must change and improve. Writing scripts to do this correctly is boring.

iParameterEstimation can do this gracefully and simply. The beauty of the program is that it has **no prior knowledge** of the model that you are attempting to estimate parameters for. It takes in a complete description of the model and produces exactly what you would expect.

iParameterEstimation is even extensible at the code level. It assumes nothing about the annotation format or the sequence file format that you input. You may write new, small modules that will read in other annotation or sequence formats, and plop them into the mainstream of the program for counting. It will even let you add new kinds of sequence models, new kinds of smoothing methods, and new ways of matching piecewise distributions.

1.3 What Can iParameterEstimation Do For You?

iParameterEstimation has loads of uses (and potential uses, given more coding on my part :), and here are a few:

- **Re-Estimate parameters for your favorite organism.** While most higher Eukariotes have similar sequence features, slight performance gains can be seen in re-estimating parameters for a specific species.
- **Perform experiments with different informant species.** You can see if another species works better for gene prediction just by changing the input files and rerunning the same model you had before.
- **Perform sequence analysis.** You can get all the sequence features in FASTA-formatted files returned to you from iParameterEstimation. Use these to look at trends in different phenomena.

- **Speed up Twinscan/N-SCAN's performance.** You can eliminate costly parts of predicting by tweaking the models such that fewer things need to be predicted, and get a performance boost.

- **Do parameter estimation for your own project.** iParameterEstimation outputs XML files which are parsible by any number of readily available libraries and can be used as inputs to your research project. The code is completely adaptable to any type of HMM or other model, so this might be a good starting point for you.

While we have primarily used iParameterEstimation for Twinscan and N-SCAN, there are any number of uses you can put it to. If you have any ideas, feel free to contact me at rpz@cse.wustl.edu.

Chapter 2

Preliminaries

This chapter is designed to introduce you to some of the technical things that go into iParameterEstimation that you may or may not be familiar with, as well as a broad idea of the goal of parameter estimation. A lot of documentation exists on the web for this stuff, but I'm attempting to produce a quick introduction to what you need to know.

2.1 XML

XML is short for Extensible Markup Language, which is thus ideal for an extensible system such as iParameterEstimation. XML allows you to specify a data heirarchy, its keywords and the data itself. Thus it allows documents to be created whose data format is completely specified by the user (in this case, me).

XML is a sparse language which really only specifies very little in terms of rules, and thus the tokens are largely defined by the user. I've done all of the defining for you, so you won't have to worry too much about making up your own language to input into iParameterEstimation.

XML looks a lot like HTML, so if you're familiar with that, this should be a piece of cake. Let's take a look at a simple XML file:

```
<?xml version="1.0"?>
<!DOCTYPE notes SYSTEM "notes.dtd">

<notes>
    <note to="Bob" from="happy_user">
        Thanks for writing parameter estimation!
    </note>
    <note to="Bob" from="sad_user">
        I hate XML!
    </note>
    <!-- I hate this note -->
```

```
</notes>
```

At the top is the obligatory cryptic nonsense `<?xml version="1.0"?>` which specifies the version of XML, 1.0, which will probably be the only verison of XML. Thanks, guys. Next is location of the Ducoment Type Definition. This is where the document format is specified. We'll get to that later.

The guts is a heirarchical data structure rooted at "notes". The root **element**, or unit of data in XML, is denoted by a pair of **tags**, caret-bracketed representations of elements (as in `<notes>`), one beginning the element and one ending the element. All of the element's "children" are enclosed between these two tags.

The name of the element is also referred to as the tag, and this gets a bit confusing. In the case of the first child, the tag is "note". The element has two **attributes**, who the note is from and who the note is to. In general, you have to follow the `name="value"` syntax for all attributes. Attributes generally define characteristics of the data element, and the stuff between the tags is either more elements or simply character data (in our example, the note itself). Sometimes an element will have no data between the start and end. This can be abbreviated with a start/end tag:

```
<empty_note to="Bob" from="quiet_user" />
```

You can include notes about the XML code you're writing in the form of comments. The syntax of a comment is shown in the above code, beginning with a `<!--` and ending with a `-->`. You may also put comment delimiters around XML code. This practice, casually referred to as "commenting out", is generally good practice. It makes sure that if you change your mind and want to reinsert the code, you can do so without re-typing it.

XML has many tight specifications about how to properly form elements. This generally isn't a big problem if you make your XML look like the above code.

There are several XML editors out there. Since XML files get long and overwhelming, you might want to be able to collapse the longer elements that you're not concerned with, and also have checked syntax. I, personally, haven't gotten any of these to work and mostly edit them by hand or with a handful of scripts (some of them are provided).

2.2 DTD

You can probably skip this section on a first pass through the guide if you don't plan on modifying models.

DTD stands for Document Type Definition. It describes what a well-formed document looks like for a certain type you've specified. In our case, this includes what a well formed generalized hidden Markov model file looks like. It also serves as a rough documentation for the format of the XML file. If you ever have a problem with the parsing of the XML file, it could be for one of two reasons:

1. Your XML file syntax not look like the syntax in the section above, or
2. It does not match the DTD specification.

So what does a DTD file look like? Worse than an XML file. Here's an example:

```
<!ELEMENT notes (thread*, note+, empty_note*, memo*, thread*)>
    <!ELEMENT note (#PCDATA)>
    <!ELEMENT empty_note EMPTY>
    <!ELEMENT thread (note|memo)+>
    <!ELEMENT memo ANY>

    <!ATTLIST note
        from    CDATA   ""
        to      CDATA   #REQUIRED >
    <!ATTLIST empty_note
        from    CDATA   ""
        to      CDATA   #REQUIRED >
    <!ATTLIST thread >
    <!ATTLIST memo >
```

2.2.1 Element definitions

This is a description for the example document we gave in the last section. Each element type is described with respect to its contents and attributes. The root node, `notes`, is first. Within the parens we indicate that it may contain `note`s and `empty_note`s. The funny symbols after the element names are **quantifiers** which specify how many of these elements can be occur as a child element. The '+' indicates that at least 1 of this kind of element must be present, the '*' indicates 0 or more may be present and the '?' indicates up to 1 element of this type may be present.

Another important thing to note about this content specifier is that it defines a *specific sequence* of elements that are expected to follow. If these are out of order in the XML file, a parse error will occur and everything stops.

Moving on to the other elements, the note has a funny symbol in its contents, `#PCDATA`. This stands for Parsed Character DATA, which is a reserved word in DTD indicating any sort of 'free text' which is meaningful to the program receiving the data. In our case, it is simply a text note, which our XML parser presumably reads and stores somewhere.

We see another reserved word, `EMPTY`, as the contents of the `empty_note`. This explicitly indicates that nothing may be contained within a `empty_note` element. The `thread` element contains 1 or more note OR anonymous note element. The pipe ("`|`") symbol represents an "or". Additionally, `memo`s, which are of a more general nature (and are broadcast so they have no `to` or `from` attributes), contain anything, indicated by `ANY`. That means it can have parsed character data, or any of the elements described in the DTD.

2.2.2 Attribute lists

Moving on to the `ATTLIST`s, we see definitions of the attributes of each element we defined above. All attributes in an XML file will be defined in an `ATTLIST`. For each attlist, we first indicate which element we're defining attributes for (in the first case, a `note`), then define each of the attributes. The `from` attribute is `CDATA` (character data) with default value `""`. This is the value that the `from` attribute is given if no `from` attribute is listed in a `note` tag[1].

Unlike the `from` field, the third token of the `to` attribute definition is another reserved DTD word, `#REQUIRED`. This indicates that the attribute must be specified for every instantiation of the element `note`. If it is not, parsing will stop.

The remaining `ATTLIST`s are fairly similar, except the `thread ATTLIST`, which has no attributes at all. It is not necessary to include this, however as a convention I tend to, to express no attributes are associated with this element type. The root node has no `ATTLIST`, and as a rule it is a good idea to make it simply a container for the document.

All whitespace in DTDs and XML files is completely flexible, tabs are only for readability.

2.2.3 Including DTDs

DTDs allow you to identify what kind of an XML file we are working with. We can put a special tag at the beginning of an XML file to indicate this, as in our previous example:

```
<!DOCTYPE notes SYSTEM "notes.dtd">
```

This indicates that the machine should look for the `notes.dtd` file in order to figure out how to check the contents of the XML file. Note that no path is given. There are several possibilities for how to resolve the correct path, but the one in UNIX libxml is to check the environment variable `SGML_SEARCH_PATH`.

2.3 GTF

Parameter estimation is sometimes referred to as "learning". The idea is that you give the machine a set of examples which represent true positives of the phenomenon you want to "learn", and then the system is ready to try and guess "on its own" what examples are true positives and which are not.

In our case, we're predicting genes. Our examples will be gene transcript annotations and our predictions will be the output of our gene predictor.

iParameterEstimation is designed such that it could be extended to take in any type of annotation format, with a little elbow grease. Currently, however,

[1] The funny thing about default values is that they are completely ignored by libxml. They do indicate what the value should be if no attribute of this type is given, however it is up to the programmer to insure that the default values are heeded.

it only accepts GTF. We use this format because we have developed several scripts at Brent lab to check to see if these are clean annotations (many of the RefSeq genes on the UCSC genome browser are totally bogus).

Below I've included the GTF 2.2 annotation format specification from the Brent Lab website, `http://mblab.wustl.edu`.

2.3.1 Introduction

GTF stands for Gene transfer format. It borrows from GFF, but has additional structure that warrants a separate definition and format name.

Structure is as GFF, so the fields are: `<seqname> <source> <feature> <start> <end> <score> <strand> <frame> [attributes] [comments]`

Here is a simple example with 3 translated exons. Order of rows is not important.

```
381 iscan CDS          380 401 . + 0 gene_id "1"; transcript_id "1.1";
381 iscan CDS          501 650 . + 2 gene_id "1"; transcript_id "1.1";
381 iscan CDS          700 707 . + 2 gene_id "1"; transcript_id "1.1";
381 iscan start_codon  380 382 . + 0 gene_id "1"; transcript_id "1.1";
381 iscan stop_codon   708 710 . + 0 gene_id "1"; transcript_id "1.1";
```

The whitespace in this example is provided only for readability. In GTF, fields must be separated by a single TAB and no white space.

2.3.2 GTF Field Definitions

`<seqname>`

The name of the sequence. Commonly, this is the chromosome ID or contig ID. Note that the coordinates used must be unique within each sequence name in all GTFs for an annotation set.

`<source>`

The source column should be a unique label indicating where the annotations came from — typically the name of either a prediction program or a public database.

`<feature>`

The following feature types are required: "`CDS`", "`start_codon`", "`stop_codon`". The features "`5UTR`", "`3UTR`", "`inter`", "`inter_CNS`", "`intron_CNS`" and "`exon`" are optional. All other features will be ignored. The types must have the correct capitalization shown here.

`CDS` represents the coding sequence starting with the first translated codon and proceeding to the last translated codon. Unlike Genbank annotation, the stop codon is not included in the `CDS` for the terminal exon. The optional feature "`5UTR`" represents regions from the transcription start site or beginning of the

known 5' UTR to the base before the start codon of the transcript. If this region is interrupted by introns then each exon or partial exon is annotated as a separate 5UTR feature. Similarly, "3UTR" represents regions after the stop codon and before the polyadenylation site or end of the known 3' untranslated region. Note that the UTR features can only be used to annotate portions of mRNA genes, not non-coding RNA genes.

The feature "exon" more generically describes any transcribed exon. Therefore, exon boundaries will be the transcription start site, splice donor, splice acceptor and poly-adenylation site. The start or stop codon will not necessarily lie on an exon boundary.

The "start_codon" feature is up to 3bp long in total and is included in the coordinates for the "CDS" features. The "stop_codon" feature similarly is up to 3bp long and is included in the coordinates for the "3UTR" features, if used.

The "start_codon" and "stop_codon" features are not required to be atomic; they may be interrupted by valid splice sites. A split start or stop codon appears as two distinct features. All "start_codon" and "stop_codon" features must have a 0,1,2 in the <frame> field indicating which part of the codon is represented by this feature. Contiguous start and stop codons will always have frame 0.

The "inter" feature describes an intergenic region, one which is by almost all accounts not transcribed. The "inter_CNS" feature describes an intergenic conserved noncoding sequence region. All of these should have an empty transcript_id attribute, since they are not transcribed and do not belong to any transcript. The "intron_CNS" feature describes a conserved noncoding sequence region within an intron of a transcript, and should have a transcript_id associated with it.

<start> <end>

Integer start and end coordinates of the feature relative to the beginning of the sequence named in <seqname>. <start> must be less than or equal to <end>. Sequence numbering starts at 1. Values of <start> and <end> that extend outside the reference sequence are technically acceptable, but they are discouraged.

<score>

The score field indicates a degree of confidence in the feature's existence and coordinates. The value of this field has no global scale but may have relative significance when the <source> field indicates the prediction program used to create this annotation. It may be a floating point number or integer, and not necessary and may be replaced with a dot.

<frame>

0 indicates that the feature begins with a whole codon at the 5' most base. 1 means that there is one extra base (the third base of a codon) before the first

whole codon and 2 means that there are two extra bases (the second and third bases of the codon) before the first codon. Note that for reverse strand features, the 5' most base is the `<end>` coordinate.

Here are the details excised from the GFF spec. Important: Note comment on reverse strand.

'0' indicates that the specified region is in frame, i.e. that its first base corresponds to the first base of a codon. '1' indicates that there is one extra base, i.e. that the second base of the region corresponds to the first base of a codon, and '2' means that the third base of the region is the first base of a codon. If the strand is '-', then the first base of the region is value of `<end>`, because the corresponding coding region will run from `<end>` to `<start>` on the reverse strand.

Frame is calculated as (3 - ((length-frame) mod 3)) mod 3.

1. (length-frame) is the length of the previous feature starting at the first whole codon (and thus the frame subtracted out).

2. (length-frame) mod 3 is the number of bases on the 3' end beyond the last whole codon of the previous feature.

3. 3-((length-frame) mod 3) is the number of bases left in the codon after removing those that are represented at the 3' end of the feature.

4. (3-((length-frame) mod 3)) mod 3 changes a 3 to a 0, since three bases makes a whole codon, and 1 and 2 are left unchanged.

`[attributes]`

All nine features have the same two mandatory attributes at the end of the record:

- `gene_id value;` A globally unique identifier for the genomic locus of the transcript. If empty, no gene is associated with this feature.

- `transcript_id value;` A globally unique identifier for the predicted transcript. If empty, no transcript is associated with this feature.

These attributes are designed for handling multiple transcripts from the same genomic region. Any other attributes or comments must appear after these two and will be ignored.

Attributes must end in a semicolon which must then be separated from the start of any subsequent attribute by exactly one space character (NOT a tab character).

Textual attributes should be surrounded by doublequotes.

These attributes are required even for non-mRNA transcribed regions such as "`inter`" and "`inter_CNS`" features.

`[comments]`

Comments begin with a hash ('#') and continue to the end of the line. Nothing beyond a hash will be parsed. These may occur anywhere in the file, including at the end of a feature line.

2.3.3 Examples

Here is an example of a gene on the negative strand including UTR regions. Larger coordinates are 5' of smaller coordinates. Thus, the start codon is 3 bp with largest coordinates among all those bp that fall within the CDS regions. Similarly, the stop codon is the 3 bp with coordinates just less than the smallest coordinates within the CDS regions and the largest coordenates among all those within the 3UTR regions.

```
140 iscan inter        5141   8522  . - . gene_id ""; transcript_id "";
140 iscan inter_CNS    8523   9711  . - . gene_id ""; transcript_id "";
140 iscan inter        9712   13182 . - . gene_id ""; transcript_id "";
140 iscan 3UTR         65149  65487 . - . gene_id "0"; transcript_id "0.1";
140 iscan 3UTR         66823  66995 . - . gene_id "0"; transcript_id "0.1";
140 iscan stop_codon   66993  66995 . - 0 gene_id "0"; transcript_id "0.1";
140 iscan CDS          66996  66999 . - 1 gene_id "0"; transcript_id "0.1";
140 iscan intron_CNS   70103  70151 . - . gene_id "0"; transcript_id "0.1";
140 iscan CDS          70207  70294 . - 2 gene_id "0"; transcript_id "0.1";
140 iscan CDS          71696  71807 . - 0 gene_id "0"; transcript_id "0.1";
140 iscan start_codon  71805  71806 . - 0 gene_id "0"; transcript_id "0.1";
140 iscan start_codon  73222  73222 . - 2 gene_id "0"; transcript_id "0.1";
140 iscan CDS          73222  73222 . - 0 gene_id "0"; transcript_id "0.1";
140 iscan 5UTR         73223  73504 . - . gene_id "0"; transcript_id "0.1";
```

Note the frames of the coding exons. For example:

1. The first CDS (from 71807 to 71696) always has frame zero.

2. Frame of the 1st CDS =0, length =112. (3-((length - frame) mod 3)) mod 3 = 2, the frame of the 2nd CDS.

3. Frame of the 2nd CDS=2, length=88. (3-((length - frame) mod 3)) mod 3 = 1, the frame of the terminal CDS.

4. Alternatively, the frame of terminal CDS can be calculated without the rest of the gene. Length of the terminal CDS=4. length mod 3 =1, the frame of the terminal CDS.

Note the split start codon. The second start codon region has a frame of 2, since it is the second base, and has an accompanying CDS feature, since CDS always includes the start codon.

Here is an example in which the "exon" feature is used. It is a 5 exon gene with 3 translated exons.

```
381 iscan  exon         150  200 . + . gene_id "381"; transcript_id "381.1";
381 iscan  exon         300  401 . + . gene_id "381"; transcript_id "381.1";
381 iscan  CDS          380  401 . + 0 gene_id "381"; transcript_id "381.1";
381 iscan  exon         501  650 . + . gene_id "381"; transcript_id "381.1";
381 iscan  CDS          501  650 . + 2 gene_id "381"; transcript_id "381.1";
381 iscan  exon         700  800 . + . gene_id "381"; transcript_id "381.1";
381 iscan  CDS          700  707 . + 2 gene_id "381"; transcript_id "381.1";
381 iscan  exon         900 1000 . + . gene_id "381"; transcript_id "381.1";
381 iscan  start_codon  380  382 . + 0 gene_id "381"; transcript_id "381.1";
381 iscan  stop_codon   708  710 . + 0 gene_id "381"; transcript_id "381.1";
```

Several Perl scripts have been written for checking, parsing, correcting, and comparing GTF-formatted annotations. Most of the important ones are included the Eval package, which comes equipped with a GTF parsing Perl package GTF.pm.

The Eval documentation contains a complete code-level documentation of GTF.pm, suitable for able Perl programmers to create and parse GTF files.

The script `validate_gtf.pl` included in the Eval package is particularly useful for checking that your GTF annotation is consistent and well-formed. This can also be done over the web.

You can retrieve all of these at http://mblab.wustl.edu.

2.4 Terms

In this section I'm listing some terms that you might want to familiarize yourself with in order to follow along. These are not intended as authoritative definitions, but rather to indicate which definition will be used in this text.[2]

2.4.1 Probabilistic Terms

- **parameter** - refers to a number which defines the likelihood of an event. This can refer to either the actual number, or its meaning to the model.
- **model** - in our case, a probabilistic model, and specifically, a parameterized probabalistic model. A collection of parameters which in some way define the character of a phenomenon.
- **distribution** - a function (sometimes probability density function) which differentiates the probability of an event between different examples.
- **Gaussian distribution** - (sometimes "normal distribution") a distribution function whose greatest density is focused at its mean and lowest density focused on the extremites of the mean. (PDF: $Pr(X) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$).
- **exponential distribution** - a distribution with a constant average rate of change whose mean is focused near 0, and whose tail limits at 0 as the function approaches infinity. (PDF: $Pr(X) = \lambda e^{-\lambda x}$).

[2] Rule of thumb: Wikipedia is your friend.

- **geometric distribution** - the discrete case of the exponential distribution (PDF: $Pr(n) = (1-p)^n p$).
- **function smoothing** - one of many techniques applied to an emperical distribution to eliminate radical differentials between two nearby examples.
- **conditional independence** - an assumption that states that given background information x, the probability of event y is unchanged. That is, $Pr(y|x) = Pr(y)$.
- **pseudocount** - A method of making up for unobserved data by adding one or more counts to the total counts used in determining the probability of an event.
- **Markov chain** - a discrete-time stochastic[3] process[4] which approximates the likelihood of an event happening given all previous events. It has the Markov property of being conditional on only the current state.
- **higher order Markov chain** - a Markov chain which is dependent on more than on previous event to generate the current event. For example, if it has rained for 5 days, how likely is it to rain tomorrow? We can model this (pretty naïvely) with a higher-order Markov chain. We can assume that the weather before 5 days ago has no influence on the weather tomorrow (probably incorrectly), and create a "5th order" Markov chain, modelling the occurence of rain on the 6th day.
- **hidden Markov model** - a probabilistic state machine with the Markov property. The idea is that some variable is "hidden". An example (taken from [6]) is the case of Casino which occasionally uses a loaded die. If you are given the rolls, the hidden variable is the kind of die being used. This is often called a probabilistic state machine, one which moves from one state to the next with probability given by the current state (a 1st order Markov chain). For more detailed descriptions, see [6].
- **generalized hidden Markov model** - a hidden Markov model where the stay in a particular state modeled by any arbitrary length distribution. This type of hidden Markov model is ideal for gene prediction. Features such as coding exons don't follow an exponential length distribution, as is imposed on ordinary hidden Markov models. The use of a generalized hidden Markov model allows for an arbitrary length distribution, as well as modeling features in an exon, such as the acceptor and donor site. For more information, see [10], [6] and [4].
- **fitting** - A model is said to be fit to the data if there is sufficient training data to obtain reliable parameters. There are two possible problems that

[3] read: probabilistic
[4] read: function

can occur when attempting to fit a model. If there is too little data, the model is said to be overfit, because the sparse examples make the parameters the result of few, possibly inaccurate examples. Another problem is underfitting, where the model has too few parameters to reliably discriminate between true positive examples and negative ones.

2.4.2 Genomic Terms

- **G+C Isochore** - commonly, one of 4 or less ranges of DNA guanine plus cytosine percent content in genomic DNA sequences.
- **transcript** - a collection of boundaries in a DNA sequence which is posited to be transcribed. These boundaries represent exons.
- **gene** - a collection of one or more transcripts which are believed to be originating from the same transcription process, due to similarity in splice pattern.
- **acceptor site** - a $\sim$40 base pair window upstream of an internal or terminal exon (or internal UTR exon) which contains the pyrimidine tract, the branch point, and the AG consensus site.
- **donor site** - a $\sim$9 base pair window upstream and downstream of an internal or initial exon (or internal UTR exon) which contains the GT donor consensus site.

2.4.3 Bioinformatics Terms

- **feature** - any section of a genomic sequence that has a certain characteristic. For example, an initial exon feature would be a coding exon beginning with an ATG and ending with a donor site.
- **alignment** - a matching of two sequences which have clear indication of similarity. For a more detailed treatment, see [6].
- **BLAST** - Basic Local Alignment and Search Tool. A program for finding good local alignments of two sequences. Two major flavors exist, NCBI-BLAST and our weopon of choice, WU-BLAST [7].
- **BLASTZ** - variant of BLAST designed to be more sensitive in aligning (i.e. generate more alignments). See [12].
- **MULTIZ** - multiple sequence aligner. Takes several outputs of BLASTZ and merges them into a multiple sequence alignment. See [1].
- **Twinscan** - the external name the research project in [9]. This is essentially Genscan [5] plus the conservation sequence.
- **target sequence** - the genomic sequence on which genes are being predicted.

- **informant** - species aiding prediction by providing conservation information.
- **conservation sequence** - a sequence representing the result of a two-way alignment with WU-BLAST. The conservation alphabet includes three characters, "`|`" for match, "`:`" for mismatch or gap, and "`.`" for unaligned. Internally, this is represented as the numbers 0, 1, and 2 for mismatch/gap, match, and unaligned, respectively.
- **N-SCAN** - the external name for the research project in [8]. This is essentially Genscan plus a Baysian network representing the differing evolutionary rates in a multiple alignment of one or more informant species.

2.4.4 Brent Lab Terms

- **iscan** - the standalone program which predicts genes using a conservation sequence or multiple alignment. Essentially, it is Twinscan, N-SCAN, Twinscan-EST, N-SCAN_EST and our implementation of Genscan.
- **zoe** - Ian Korf's daughter, whose name permeates the `iscan` code. iscan is often referred to as the 'zoe codebase,' and files input and output from iscan in native format are said to be in 'zoe' format. This includes parameter files, which are suffixed with `.zhmm`.

Chapter 3

Installation

iParameterEstimation should work on any UNIX system with Perl 5 installed. Certain prerequisites are required, but these can easily be obtained.

There are two main ways to install iParameterEstimation: with the pre-built `rpm` (Redhat Package Manager) package, or by hand. I personally recommend using an rpm client like `yum` or `yast`, since it will do a lot of things automatically and smoothly for you, as well as prevent you from reinstalling the same thing twice in different spots if you are updating.

3.1 Installing via RPM

The rpm (Redhat Package Manager) installation offers a potentially cleaner, less interactive install. We recommend that you use the rpm wrapper that comes with your linux distribution (for example, `yum` with RedHat, `yast` with Suse). This will automatically resolve, fetch and install dependencies for you.

Whether you rpm by hand or do it with a package manager, you will need to install the eval rpm, which is available on the site http://mblab.wustl.edu.

Here we provide a guide to the manual installation, if you are doing one.

1. **Check your dependencies.** You will basically need the eval rpm, which you can download from the website. Install it with `rpm -Uvh eval.*.rpm`.

   You will also need the perl-libxml-enno, perl-libxml-perl and perl-XML-LibXML packages, which can be downloaded from any number of standard sites with rpm packages for an OS distribution. This may follow you further down more chains of dependencies, so keep following until all dependencies are resolved. With yum or yast you can usually just request the package by name.

2. **Install the rpm.** Simply use the rpm update command to do this:

   ```
   rpm -Uvh ipe-1.0-1.i386.rpm
   ```

This will place all the install files in `/usr/bin`, `/usr/lib` or `/usr/lib64` (depending on your archetecture), `/usr/share/man` and `/usr/share/iPE`, where you'll find the documentation and configuration files.

It will also create a file which sets the system-wide `SGML_SEARCH_PATH` variable to `/usr/share/sgml`, where the DTD files are stored.

3.2 Installing via deb

The debian package manager is orthoganal to RPM, however it is equally supported in iPE. You can retrieve the `.deb` file from `http://mblab.wustl.edu`, and install it with `dpkg -i ipe.*.deb`. You will also need to install the `eval` debian package prior to installing iPE.

3.3 Installing Manually

1. **Get root access.** Go beat up your administrator, unless he's nice. Otherwise, just hand this over to him.

2. **Make sure Perl 5 is installed.** Type `perl -v` on a command prompt, and you should get something like

   ```
   This is perl, v5.8.8 built for darwin-2level

   Copyright 1987-2006, Larry Wall
   ...
   ```

   with more information following.

3. **Make sure you have other prerequisites installed.** The modules `XML::Parser::Checker` and `XML::LibXML` are required. These can be installed using the `CPAN` module, for example,

   ```
   perl -MCPAN -e 'install XML::LibXML'
   ```

   This can run into a number of snags, with getting dependencies or tests failing. It's often advisible to try to find a package, RPM or otherwise, that automatically this, or you can also bypass the tests by building them in the `.cpan` build directory.

   The `eval` module is also required. This can be obtained from the Brent Lab website at `http://mblab.wustl.edu`.

4. **Select an install directory.** For typical users this will be `/usr/bin` and `/usr/lib`, however if you intend to run on NFS, you may want to select the NFS mounted directories as an install point.

iParameterEstimation will install itself into the directories `share`, where the manpages and DTD files will be kept, `lib` where the module files will be kept, and `bin` where the run script will be kept.

5. **Run the install script.** If you haven't already, unpack the tarball you found on `http://mblab.wustl.edu` (`gzip -cd iPE.tar.gz | tar xvf -`). You will find the script `install_ipe.sh` in the top level directory of iPE. Run it as root (or `sudo ./install_ipe.sh`).

   First you will be prompted for a prefix for installation:

   ```
   Please enter a prefix (or just press enter for the default)
   where your bin and lib directories are
   Install prefix: [/usr]
   ```

   If you want to install the `lib`, `bin` and `share` under a different directory besides `/usr`, enter a full path here (with a leading "/"). Hit enter to take the default (in brackets).

   Now select where the configuration files will go:

   ```
   A standard set of configuration files used for parameter estimation
   come bundled in this package.  They should be place somewhere where
   everyone can use them.

   Where to put configuration files: [/usr/share/iPE/conf]
   ```

   Here, you are asked where to put the configuration files that come with iParameterEstimation that help you run the program. These include various models, standard command options, and feature maps (we'll get into the detail of these later). You'll want this in a place where all users who want to use iParameterEstimation can see and read the files (not necessarily write). Hit enter to take the default (in brackets).

   Next select where the DTD files will be stored:

   ```
   DTD files are description files which describe the format of XML files,
   which are the input to iParameterEstimation.  They need to be system-wide
   accessible in order to run iParamterEstimation.

   Where to put DTD files: [/usr/share/sgml]
   ```

   As described in a previous chapter, DTD files define what XML files look like, and XML files are the main input to iParameterEstimation. These files are small and harmless, so installing them here is fine, unless for some reason you want to group them with other DTD files. Hit enter to take the default (in brackets).

```
iParameterEstimation comes with user-level documentation in PDF form.
You may install them anywhere on the system.

Where to put documentation: [/usr/share/iPE/doc]
```

This manual, as well as example files will be installed to this directory. Again, pick a spot where everyone can read but not necessarily write.

After this, iParameterEstimation should install itself.

6. **Set the environment variable** `SGML_SEARCH_PATH`**.** This points to the location of your DTD files so iParameterEstimation can find them when run. Find a system-wide login script or create one that executes the line (in `bash`):

```
export SGML_SEARCH_PATH=/path/to/dtd/files
```

where the path is the path where you specified for your DTD files. In many systems, you can make a new bash script in `/etc/profile.d` to do this.

3.4 Testing your installation

The easy way to do this is just run the executable, `ipestimate` from any location. You should get a usage statement if everything is OK:

```
$ ipestimate
iParameterEstimation v1.0.0  06 June 2007
Copyright (C) 2005-2007 Washington University in Saint Louis
by Bob Zimmermann and Brent Lab
http://mblab.wustl.edu

Usage: ipestimate instance-file

iParameterEstimation gets all command line-type options from an XML file
called the instance file.  For more details, see the user guide that came
with this package, or http://mblab.wustl.edu/software/iparameterestimation.
```

We'll get into some test cases later in the Getting Started chapter.

If you have problems, make sure your `PERL5LIB` environment variable is set to the locations of where iParameterEstimation was installed. (If you forgot where, just rerun the script, and you can see the verbose output of the exact locations.) Generally, `/usr/lib` is included in the `PERL5LIB` list.

You also have to make sure your `SGML_SEARCH_PATH` variable is set to the location of the DTD files (often in `/usr/share/sgml`).

Chapter 4

Getting Started

As mentioned in the Preface, iParameterEstimation was intended as both a parameter estimator for Twinscan/N-SCAN and a general purpose parameter estimation tool. You will notice that iParameterEstimation inputs and outputs many items that are relevant specifically to Twinscan/N-SCAN, but many of the files input and output are generic in nature.

If you haven't already, download `iscan` from http://genes.cs.wustl.edu, and read about it in [5], [9] and [8]. `iscan` should come packaged with some parameter files with the extension `.zhmm` already. Take a look at them. Pretty opaque, huh? Don't worry about understanding them. If I've done my job, you will never have to look at one of those.

4.1 Input Files

You might have noticed that when you type the `ipestimate` command, the usage only asks for one file, an "instance" file, and no command line options. There are many options passed to iParameterEstimation, however they are all in files.

As shown in Figure 4.1, instance files point to all the files that will be used by iParameterEstimation. It points to an annotation set (the training examples), a gHMM file (the parameter definitions), and one or more feature map files. In addition, instance files contain the "missing" commandline options for iParameterEstimation. Included in the iParameterEstimation tarball is an example set of input files. This should be installed where this documentation file is. Let's take a look at them one by one.

You can find some example input files in the documentation directory, which was installed with iParameterEstimation (typically in `/usr/share/iPE/doc`). You will find two sequence files which have been `bzip2`ed, they can be decompressed with `bzip2 -d`.[1]

[1] If you don't already have bzip2, get it. It is far superior to gzip for compressing sequence files, especially conseq files.

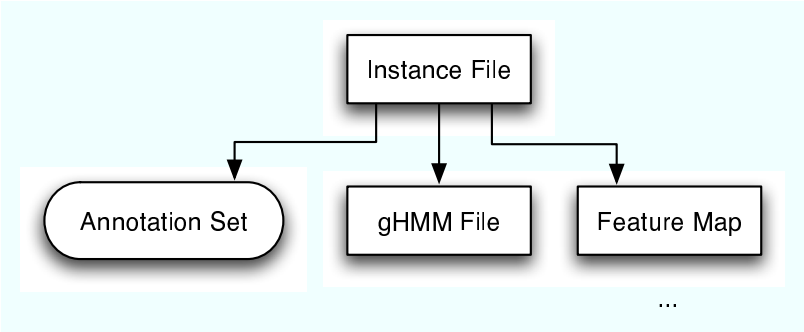


Figure 4.1: The high-level shematic of an instance file

4.1.1 The Instance File

An instance file is designed to represent a single run of iParameterEstimation, thereby freezing the training and keeping a record of how it was done[2]. The instance file can be reused, of course, and if the run fails, it doesn't hurt to just revise it until it works.

Like most input files to iParameterEstimation, the instance file is an XML file. Let's take a look at the example file:

```
<?xml version="1.0"?>
<!DOCTYPE iPE_instance SYSTEM "iPE_instance.dtd">
```

Like all XML files, we start out by defining what kind of file is coming along. This time we're asserting that this file is defined in the DTD `iPE_instance.dtd`. Hopefully this file is within the paths contained in the environment variable `SGML_SEARCH_PATH`. If not, check your installation.

```
<iPE_instance>
    <author>Bob Zimmermann</author>
    <date>5/8/06</date>
```

The root element is `iPE_instance`, and all elements in this document are contained within it. Its author and creation date are manually identified by the user.

Next are the locations of the `gHMM` file and `feature_map` files:

```
    <gHMM_file>
        gHMM.xml
    </gHMM_file>
    <feature_map_files>
        gtf_map.xml
    </feature_map_files>
```

[2]From experience, leaving a long paper trail to your experiments is wise, as results are sometimes difficult to reproduce.

The gHMM file, broadly, defines our global gene prediction model. It defines all the parameters that will be estimated and some that won't. Our feature map file explains how to convert the features in our annotation sets into features that pertain to the gHMM. We'll discuss these more later.

Now we get to our annotation set:

```
<annotation_files>
    chr1.eval.short.gtf
</annotation_files>
<seq_files type="dna">
    chr20.masked.short.fa
</seq_files>
<seq_files type="cons">
    chr20.conseq.short.fa
</seq_files>
```

Each file in the annotation set corresponds to all the sequence files in the order they're defined. If there were additional annotations and sequences, they would all correspond to each other. There must be equal numbers of files in each of these.

Sequence files are defined by their identifying sequence type, such as "dna", and "cons", shown above. Several different types of sequences can be input to iParameterEstimation. Here are the important ones:

- `dna` - These are files of DNA sequences, containing all possible bases and ambiguity codes.
- `conseq` - These are conservaetion sequence files, used in Twinscan. These are sequences of 0's, 1's and 2's, representing the result of an alignment by WU-BLAST.
- `estseq` - These are EST sequence files, used in Twinscan-EST and N-SCAN-EST [13].
- `malign` - These are alignment files, adapted from either MULTIZ or BLASTZ runs. These are used with NSCAN.

Note that iParameterEstimation will input any number of these types of sequences along with the annotation, as long as the properly correspond.

iParameterEstimation is sensitive to filename extensions. The following extensions are currently supported:

- `.fa` - FASTA-formatted file. The format begins with a header, starting with a `>`, and continuing with a sequence definition. The following lines are assumed to contain sequence.
- `.conseq` - RAW-formatted file. This is a file containing no header and no whitespace. It simply contains the sequence.

- `.align` - Alignment file. The first line is a FASTA header. The definition in the header is expected to contain comma-separated names of the target sequence followed by each of the informant sequencs. The second line is the target sequence, containing no gaps or whitespace. The following lines are each of the informant sequences, in the order indicated by the header.
- `.gtf` - Gene Transfer Format file. This file is an annotation file, described in the first chapter.

At the bottom of the file, you'll see another element called `<options>` with several subelements. These are used in place of commandline options. This way we can preserve all inputs to iParameterEstimation without any extra work except to save the file. Each option is an element with no attributes and a value between the element's start and end tags. For example:

```
<verbose>true</verbose>
```

In this example we have an option named `verbose` and have set it to true. This is also an example of a boolean option. All other options have some sort of value associated with it.

There are many options, and they are all detailed in the appendix, but here are some of the important ones:

- `verbose` - This option tells iParameterEstimation whether or not to print progress messages to `stderr`. If the `messageOutputFile` option is set, then the output is redirected to that file.
- `outputBaseDir` - This is the base directory of all output files.
- `zoeOutputFile` - This is the name (not path) of the output parameter file to be used with `iscan`.
- `performCount`, `performNormalize`, `performScore` - These are boolean options (must be set to `true` or `false`) that indicate how far to take the estimation. You can set all of these to true, and still extract the counts, porbabilities and parameters from the output files listed in `countOutputFile`, `probOutputFile` and `xmlOutputFile`.

4.1.2 gHMM files

The `gHMM` file is the complete definition of every parameter that is to be estimated for iParameterEstimation. I won't go into too much detail here, since you may never have to look at these files.

These files are used to estimate parameters for the different flavors of `iscan`: Twinscan, N-SCAN and their respective EST versions. All of these `gHMM` files are provided with the software distribution in the `conf` directory. Sometimes the `gHMM` file is modified for special runs on different organisms, for example ones with less bias in different isochores, or shorter genomes. The included `gHMM` files are optimized for the higher eukariotes, unless otherwise noted.

4.1.3 Feature Map Files

`feature_map` files map an annotation format's features into features that can be understood by the `gHMM` provided. For example, GTF only uses the CDS feature to represent coding sequence, however, Twinscan's `gHMM` uses initial, internal, terminal and single exon types to represent coding exons. This is where the feature map comes in.

In practice, only two feature maps are used, `gtf_map.xml` and `gtf_map_utr.xml`. The former is used in Twinscan without UTR predictions, and the latter is used in Twinscan with UTR predictions, N-SCAN and their respective EST variants.

These will be discussed in more detail in the appendix.

All of these things come together to give you a complete picture of how parameters were estimated for a particular run if iParameterEstimation. I highly recommend you hold onto these after every successful run.

4.2 Running iParameterEstimation

Once you've done the work of preparing your annotation set and preparing the configuration files correctly, you are pretty much on your way to estimating parameters.

We'll be running through the example provided, so first get a copy of the example files in the installed documentation directory, then launch `ipestimate`. The sequence files are bzip2 compressed, so you must decompress them before running.

Run the following commands (if your example is in a different directory, adjust for this):

```
login32 ~ $ mkdir iPE_test
login32 ~ $ cd iPE_test/
login32 iPE_test $ cp /usr/share/iPE/doc/example/* .
login32 iPE_test $ bzip2 -d *.bz2
login32 iPE_test $ ipestimate instance.xml
```

Here's a step-by-step guide to what's happening as you run iParameterEstimation.

1. **Perform sanity check.**

   If everything is correct, `ipestimate` will begin working immediately. Sometimes some of the files are not present, or the output directory is missing, or other things are mismatched. `ipestimate` begins by doing a "sanity check" on most of these things so it doesn't end up running for hours just to find out it cannot write out the output file.

   If, for example, I had not decompressed the files, I would have seen the following error message:

   ```
   iPE::Instance: No such file chr1.masked.short.fa.
   ```

2. **Read in configuration files.**

   The instance file contains all our options and those are parsed first. We should then see a message showing us that our gHMM and feature maps are being parsed.

   ```
   iParameterEstimation v1.0.0  06 June 2007
   Copyright (C) 2005-2007 Washington University in Saint Louis
   by Bob Zimmermann and Brent Lab
   http://mblab.wustl.edu

   [ Reading gHMM.xml ... done ]
   [ Reading feature maps ... done ]
   Initialization successful.
   ```

   While parsing these files, `ipestimate` will check for any problems with the syntax or content of the files.

   After initialization is done, all the data structures required to count the examples have been created.

3. **Count examples.**

   `ipestimate` will now run in a loop over all the annotations and their corresponding sequences that were defined in our instance file. In this case there is only one set of annotations and sequences.

   ```
   [ Reading chr20.masked.short.fa ... done ]
   [ Reading chr20.conseq.short.fa ... done ]
   [ Getting G+C% levels for DNA ... done ]
   [ Converting chr1.eval.short.gtf ... done ]
   [ Outputting fasta files ... done ]
   [ Counting durations and transitions ... done ]
   [ Counting sequence models ... done ]
   ```

   First it reads in the sequence files, then gets the genomic G+C% percentage. It then converts the annotation formats into partial paths through the given gHMM. Often, you will see several error messages about it kicking out transcripts which it deems unworthy or truncating transcripts if they step out of the bounds of the sequence. This is nothing to be concerned about, however is informative in case you seem to see that most of your annotations are invalid.

   In the example instance file, we specified that we wanted to have all the features and models output to FASTA files. In general, you probably won't want to do this, as it takes up extra disk space, but this can serve as a good tool for analysis of sequences[3].

[3] You can even bypass parameter estimation and just have it output these sequences for you, by turning off the `performCount`, `performNormalize`, and `performScore` options.

Finally, the examples are counted. More warning messages may be generated here, especially if inframe stop codons are found in a transcript. This can indicate a bad annotation or a mismatched annotation/sequence set. You may want to double check your annotations if this happens.

4. **Wrap up and output files.** Once the counting is complete, `ipestimate` will smooth, normalize, and convert probabilities to scores, and finally output one or more files:

```
Counting complete.

[ Outputting counts.xml ... done ]
[ Smoothing counts ... done ]
[ Outputting smoothed_counts.xml ... done ]
[ Maximizing parameters ... done ]
[ Outputting prob.xml ... done ]
[ Outputting parameters.xml ... done ]
[ Outputting parameters.zhmm ... done ]
```

If all went well, you can use the file parameters.zhmm with `iscan` to predict genes.

4.3 A note on directory structure

It's a fairly good idea to keep things organized in general, and since iParameterEstimation outputs a lot of files in different places, it's a good idea to keep these things in line.

When I run iPE, I usually keep a top level directory for the experiment called `experimentid_estimation`, where `experimentid` is some sort of rational name for the experiment. Below that directory I usually keep several subdirectories:

- `conf` - This is where I keep my instance files, and if I make any custom gHMMs, I put them here too. I generally make a brand new instance file if there is a major change to the way I'm estimating things, so the old experiment is kept documented.
- `out` - This is the output directory where all the first-draft parameter files go, as well as any other output files. If there are any mess-ups that aren't used, I leave them here.
- `final` - This is where I keep copies of parameter files from `out` that might go on the record. This keeps the directory clean, and points you to the important files in a mess of unimportant files in `out`.
- `annotations`, `seq`, etc. - If I've made custom annotations for this estimation which are not on our system-wide database, I keep them here.

The advantage of this is that when you're done and ready to archive, you can make a tarball of the entire directory tree, and chuck it. It's helpful to put the location of the archive as a comment in the final parameter file that comes out of the experiment.

4.4 Output Files

Our example run spit out a great deal of files. The only one of current practical use is the `parameter.zhmm` file, however the others can be used for parameter analysis and sequence analysis. Note that the names and paths of these files were all specified in your instance file (under the `options` element), so there is nothing special about the names of the files below.

- `counts.xml` - This is an XML file containing all the counts in the models from the examples passed into iParameterEstimation. The file will look a lot like the gHMM file, but will contain some additional elements such as transitions and comments about how the file was generated.

- `smoothed_counts.xml` - This XML file contains the counts after they have been processed through smoothing for sparse data. The actual values don't all represent something meaningful with respect to the original counts, but do with respect to the counted distribution itself.

- `prob.xml` - This XML file contains the normalized probabilities for all the models. This should contain models which all sum to 1[4]

- `parameters.zhmm and parameters.xml` - These files are your parameter files. The `.zhmm` ("zoe" HMM) is of use to `iscan`. These are all either probabilities, log scores, or log-odds ratios of probabilities, depending on what is expected by `iscan`, and indicated by the gHMM file.

- `features` and `models` directories - These contain the output sequences that correspond to the examples (annotations) passed to iParameterEstimation.

Congratulations! You've made a successful run of iParameterEstimation. At this point, you should know enough to learn and understand basic operation iParameterEstimation with the aid of the Appendices. The rest of the book will be dedicated to explaining concepts for tuning iParameterEstimation for your specific needs. Most of this is not necessary to know for a high level understanding of the program.

[4] There are some exceptions here–but most of them are specified in your gHMM file. More on this later.

4.5 A Word About Warnings

iPE outputs warnings associated with the instance run. These are often wordy, and hopefully self-explanatory for the most part.

In our example, the warnings were redirected to a file in `warnings.txt`. This should contain the most common and disconcerting warning that iPE users encounter:

```
$ cat warnings.txt
0 samples in duration distribution Intron
for isochore 100.0.  There is probably no
sequence in the 100.0 isochore group
0 samples in duration distribution Intron
for isochore 43.0.  There is probably no
sequence in the 43.0 isochore group
0 samples in duration distribution Intron
for isochore 57.0.  There is probably no
sequence in the 57.0 isochore group
WARNING: scoring GEOMETRIC duration Intron with mean 0 because no samples were found.
WARNING: scoring GEOMETRIC duration Intron with mean 0 because no samples were found.
WARNING: scoring GEOMETRIC duration Intron with mean 0 because no samples were found.
```

Here we see 3 warnings about 3 models. This may or may not make sense right now, but let me tell you what they mean: you don't have enough genomic sequence from your organism in some of the isochore levels you have chosen. In this case, it's human, and really there is a lot of sequnce of all isochores, but we've only used chromosome 20, so there is less sequence.

When this happens in "real life", you should eliminate isochores from your model. Doing this should be explained in the appendix somehwere, but if I never get around to it, just look for instances of the word "isochore" and "ISO" use only the number "100" instead of more than one number. Sorry, bro, but that's how it's gotta be.

Chapter 5

HMMs to Hacks to Gene Prediction

This chapter is designed to give you a broad, pragmatic overview of how HMM gene prediction works to ease you into the major concepts of parameter estimation. If you do have a theoretical background in HMMs and probability, a lot of this will be a breeze, however some of the sections explaining implementation details may be pertinent to you. First we're going to look at what the gene finding problem is, then look at HMMs and how they relate to gene finding and finally look at some of the practical transforms applied to the model in order to make it computationally feasible.

5.1 What is the parsing problem?

Gene prediction is a parsing problem. Given a sequence of DNA letters, we want our computer program to parse them into transcripts made up of exons and introns. This parse will take the form of a GTF file which annotates the input sequence at the locations that our predictor "guessed" where the exons were.

How does the predictor guess these locations? Certainly any set of coordinates could be output, but we want some sort of way of telling what a good guess is and what a bad guess is. The way we do this is through a "machine" that can take in an input and assign a value the likelihood of a region being an exon or not.

A common approach (and our approach) is the use of a hidden Markov model.

5.2 What is a hidden Markov model?

An HMM is a probabilistic state machine in which some variable is unobservable or "hidden". The state the machine is in indicates the unobservable value.

An example of something we might model with a hidden Markov model is a nefarious friend who occasionally uses a tails-heads coin (fair) and a heads-heads coin (unfair). We can only observe the coin flips, but we cannot observe our friend switching coins. (This is much like our gene finding problem: we can observe the DNA, but we can't observe the exons.) Since the results of the coin flips are going to be related to which coin is being flipped, we can tie these observables to the unobservables through a probability distribution. We call this the "emission" probabilities, the probabilities that the hidden Markov model emits the observables.

5.2.1 What does an HMM look like?

The nice thing about HMMs is that you can draw them. Our example is shown in Figure 5.1.

Concretely, an HMM is defined as:

- A set of states S (the bubbles in our diagram)
- Prior probabilities of being in the state π_S
- Transition probabilities between the states $\tau_{S \times S}$ (the arrows in our diagram)
- The emission probabilities within each state $e_S(\Sigma)$

In our example, the set of states is the fair coin and the unfair coin modes of our nefarious friend, the priors are average amount of time he uses a fair coin and unfair coin, the transition probabilities are the frequencies of him switching from one coin to the next, and the emission probabilities are 50/50 (heads/tails) for the fair coin, and 100/0 for the unfair coin.

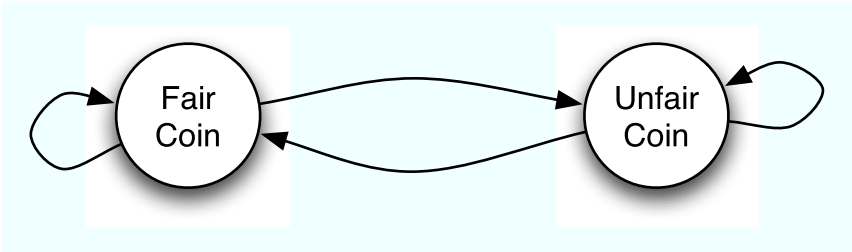


Figure 5.1: An example of a hidden Markov model.

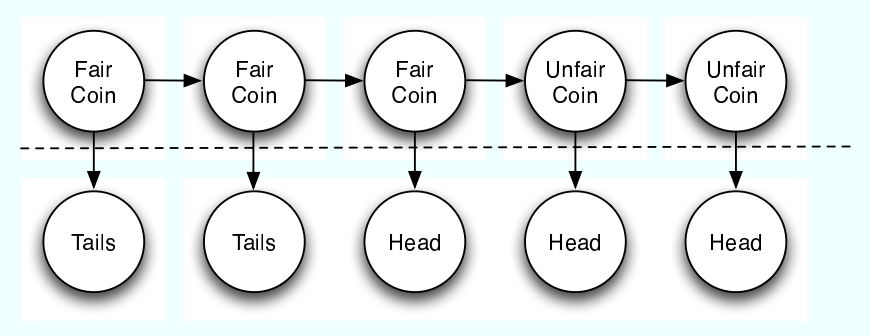


Figure 5.2: A graphical model representation of an HMM. The states above the dotted line are hidden, the flip results below are observable. The hidden Markov model in Figure 5.1 could have generated the observables shown here.

An HMM is a *generative* model, that is, the model is hypothesized to generate the observables.[1] A way to visualize this is shown in Figure 5.2. The key property in our speedup is how this model generates the observables. The observable only depends on the current state of the machine, and the current state of the machine only depends on the state immediately previous to it.

This is known as the Markov property. Formally, we define it as the independence assumption that asserts that all future states of a stochastic process are dependent only on the present state of the process, or

$$\Pr(X_i|X_{i-1}, X_{i-2}, ...X_0) = \Pr(X_i|X_{i-1})$$

This enables us to compute the most likely outcome more quickly.

The important thing to take away from this section is that a full *parameterization* of an HMM is the initial probabilities (π_S), the transition probabilities ($\tau_{S\times S}$), and the emission probabilities ($e_S(\Sigma)$). These emission probabilities are tied to the states of the machine, and characterize the states by providing the bridge between the model and the observables.

5.2.2 What is a generalized HMM?

In our previous example, the transition probability from the Fair state to another Fair state was a constant probability. Without proving it, this implies the period of time that we stay in any state in an ordinary HMM is destributed by the geometric probability density function.

What if this doesn't resemble the true distribution? What if our friend decides to stick with the Fair coin on average for 20 flips, but rarely switches

[1] It is interesting to stop and think here for a moment. Since we're modeling transcriptional machinery, why are we using something like an HMM which in no way resembles this? The reason is that it *works*, and it's one of the best ideas anyone has come up with for modeling gene structure.

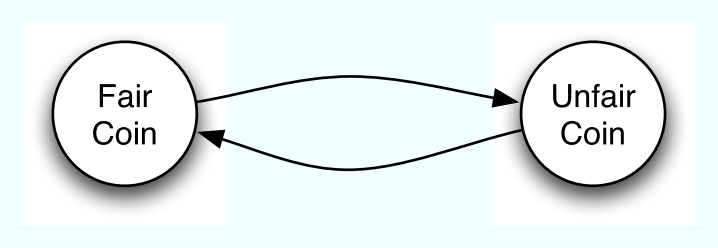


Figure 5.3: The generalized hidden Markov model version of our coin-flipping problem.

after 1 flip or 40 flips? The way to model this is to use a generalized hidden Markov model (gHMM).

A generalized hidden Markov model is essentially the same, except that self transitions are not modeled with the Markov property. Instead they are modeled arbitrarily, and any function can be attributed to the probability of staying in a certain state. We introduce a new parameter set to our original hidden markov model:

- Duration probabilities within each state $l_S(t)$

The transitions *between* the states still carry the Markov property, but the transitions *within* the states don't. We would redraw our fair/unfair HMM as shown in Figure 5.3.

Without going into detail, our computer the oracle now has to compute optimal parses differently under this model. Since the length of the hidden feature is no longer only dependent on the previous state, it is going to have to consider all possible lengths of a feature. This costs us in time, but also gives us the benefit of being able to model more complex things.

Let's say we know something about the beginnings and ends of each feature, say, an exon which begins with an acceptor site and ends with a donor site. As long as we're going ahead and taking the time to consider the entire feature, we might as well model all these things, the donor, the acceptor, and the DNA content while we're at it.

This is a very high level discussion of HMMs, and for more detail you might look at [6] or [10].

5.2.3 How does this relate to parameter estimation?

When we say we are estimating parameters for a gHMM, we are estimating the probabilities described above. The parameter file that is output by iParameter-Estimation program completely describes the gHMM that will be used for gene prediction, including the states, initial probabilities, transition probabilities, length probabilities and emission probabilities.

iParameterEstimation takes the input annotations and converts these annotations (via the input feature maps and the gHMM file) to valid state sequences and uses these to estimate parameters. The result is a long file with a lot of numbers in it that characterize the features (exons, introns, acceptors, donors, etc.), and the gene predictor uses this to predict genes.

5.3 Biological HMMs

Twinscan and N-SCAN use a gHMM that largely resembles the one Genscan uses, shown in Figure 5.4. There are two sets of duplicated states, one for genes on the forward (+) strand, and ones for genes on the reverse (-) strand.

Each of the states shown is "tied" to one or more models, specific to the feature that the state implies. These models are simply a set of numbers which score the input sequence (based on probability). A higher score for a segment of input sequence will imply that the sequence is more likely to belong to that model as opposed to any other.

As mentioned earlier, a state can have several models tied to it. The states "E_{init}" and "E_{term}" are initial and terminal exons. Initial exons are modeled with a start codon model and a donor site model. Terminal exons are modeled with an acceptor site model and a stop codon model. The "E_{sngl}" state represents a single-exon state, including a start and stop codon. The states labeled "E" are internal coding exons, and they include an acceptor site model and a donor site model.

There are three "E" states, one for each reading frame. The states are used to track the total length of the coding portion of a transcript, in order to make sure its length is divisible by three.[2] This provides a somewhat informal constraint on the lengths of stays in these states: a terminal exon must round off the coding portion of the transcript length to be divisible by three, and the internal and initial exon states transition to intron states only if they have a compatible reading frame.

Also as mentioned earlier, a generalized hidden Markov model has some function $l_S(t)$ which defines how likely it is to stay in the state for a duration of t. This can be any probability density function, including the geometric PDF. States with the geometric PDF are faster to compute, so for the states which are roughly geometrically distributed, a geometric PDF is used for $l_S(t)$. All diamond-shaped states in Figure 5.4 are modeled using the geometric distribution as $l_S(t)$.

To get a picture of what is going on, imagine that every sequence that you put into a gene predictor has some "true" path through this state diagram, beginning at intergenic, eventually transitioning to a promotor and through some transcript sequence, then finally back to intergenic, several times. There

[2]Note this is needed because of the Markov property: each state is only aware of the previous state, so in order to track reading frame, we need to keep track of where the previous exon left off.

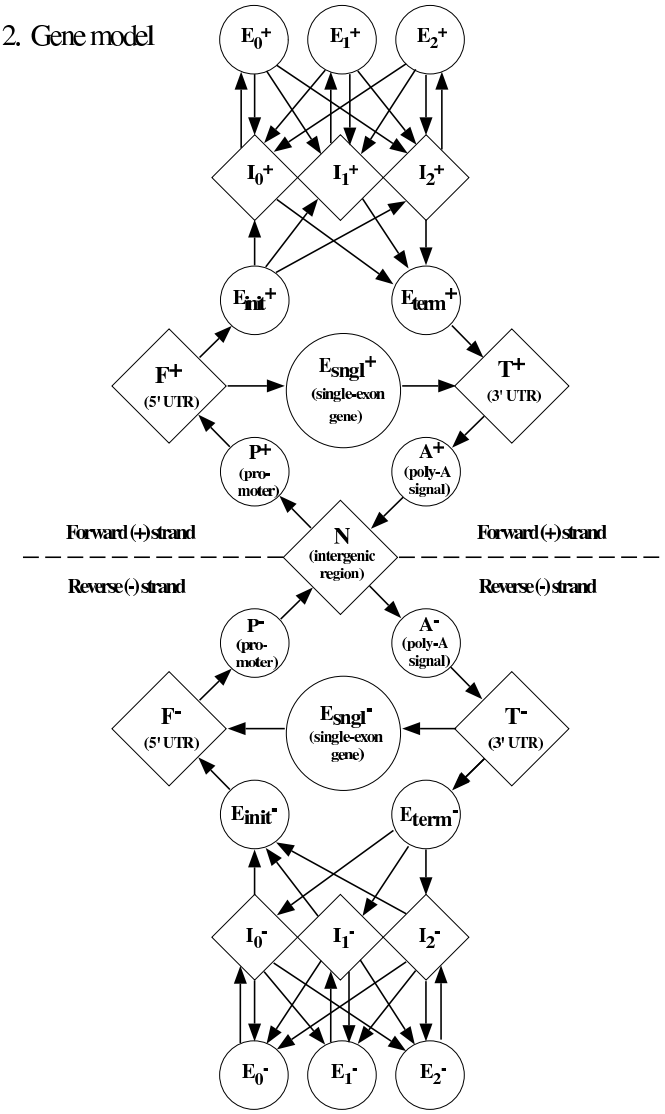


Figure 5.4: The Genscan generalized hidden Markov model for gene prediction [4].

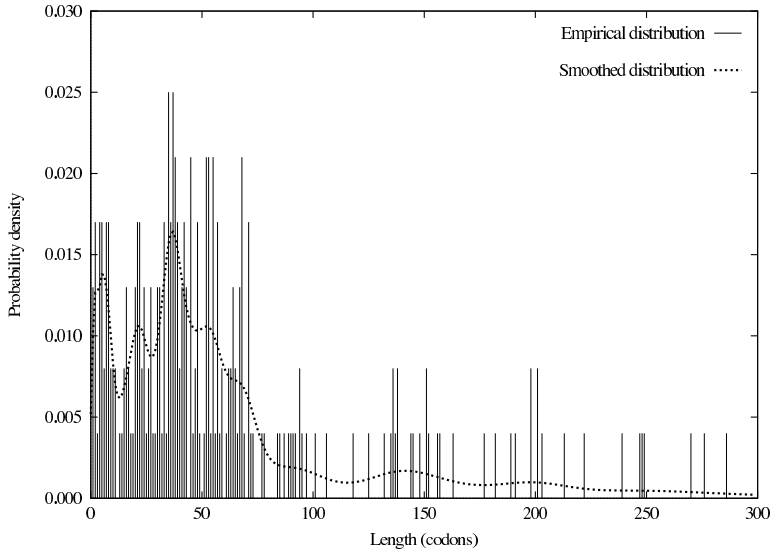


Figure 5.5: An explicit length distribution for exons, taken from [4].

are several possible paths through this with every sequence, and the gene finder finds the most probable path, given the model.

5.4 The models that make up a gene predicting gHMM

Several different models have bene established, mostly in [4], as being best for predicting genes. This is a summary of the types of models used.

5.4.1 Length distributions

Exons are modeled with an explicit length distribution, also known as a histogram. It assigns a different probability to each base, based on the frequency of each observed length in the training set. An example of this is shown in Figure 5.5. Typically all internal exons are modeled under the same length distribution, and intitial, terminal and single exons are modeled under a different distribution.

Introns roughly follow a geometric length distribution in human genomic DNA, but in some genomes a heterogeneous length distribution, one with an explicit section for the first 1000 or so lengths (depending on the genome), and then the end is a "geometric tail". (Thus all possible intron lengths are modeled, but the first 1000 are modeled more carefully.)

An important thing to note here is that a limit is placed on every explicit length distribution. We define a model within a certain range, and outside of

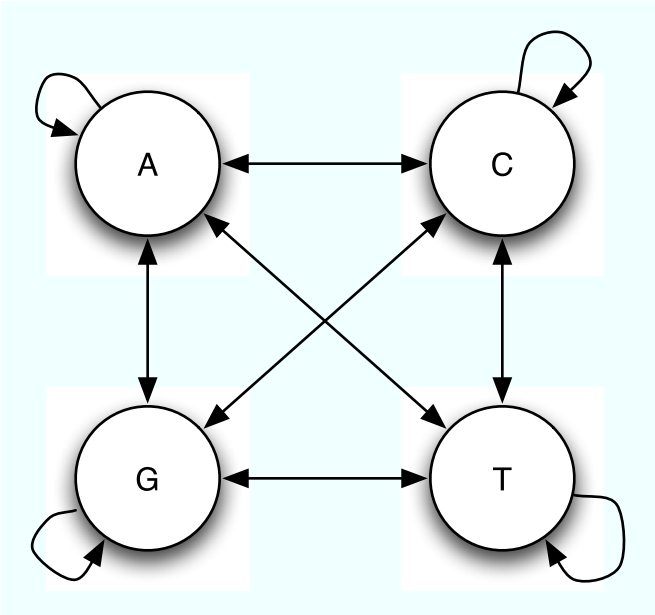


Figure 5.6: A first-order Markov chain for DNA. The states are the current DNA letter, and the transitions are the probabilities of the next letter given the current letter (represented by arrows).

that range the lengths are disallowed. This means that the gene predictor only has to look at the possible lengths *in that range*, and the rest of the lengths are not considered. Shortening the range speeds up the predictions, since fewer lengths are checked for optimality.

5.4.2 Content models

In our HMM examples above, we saw a single emission probability assigned to each observation. Given the fact that DNA only contains four letters, it would be hard to distinguish coding regions from noncoding regions with only the likelihood of seeing an A, C, G or T in an exon.

Higher-order Markov chains

Markov chains are probabilistic models like HMMs, except that there is no hidden variable. The current state is apparent from the observations, and a probability is assigned to each next state, given the current state. This can prove to be more informative about the sequence's function, since more information is present. An example of a Markov chain for DNA is shown in Figure 5.6.

For the most part, however, the current base of context is not enough information to predict the next base. To remedy this we use something called

	1	2	3	4	5	6	7	8	9	10	11	12
A	0.18	0.14	0.17	0.49	0.16	0.15	1.00	0.00	0.00	0.11	0.26	0.12
C	0.35	0.38	0.45	0.06	0.51	0.60	0.00	0.00	0.00	0.21	0.39	0.30
G	0.33	0.34	0.34	0.42	0.30	0.22	0.00	0.00	1.00	0.50	0.18	0.40
T	0.14	0.14	0.04	0.03	0.04	0.03	0.00	1.00	0.00	0.17	0.17	0.19

Table 5.1: An example of a weight matrix model for a start codon. The 6 bases upstream and 3 bases downstream of the consensus are modeled.

a higher-order Markov chain. An ordinary Markov chain would be referred to as a 1st order Markov chain, because it relies on one state for context. A 2^{nd} or 3^{rd} order Markov chain uses 2 or 3 previous states, respectively. Typically, the DNA content of introns and intergenic regions are modeled with 5^{th} order Markov chains.

Periodic higher-order Markov chains

Coding region DNA sequence has been shown to have content bias in each of the three codon positions. In order to take advantage of this, gene predictors often use a 3-periodic 5^{th} order Markov chain. This was introduced in GeneMark [2], a program for predicting genes in prokaryotic organisms.

A periodic Markov chain is one which cycles through different state transition probabilities. That is every n^{th} state is predicted with a the same Markov chain in an n-periodic Markov chain. For coding region, this is 3 different Markov chains, one for each codon position.

Position-specific scoring matrices

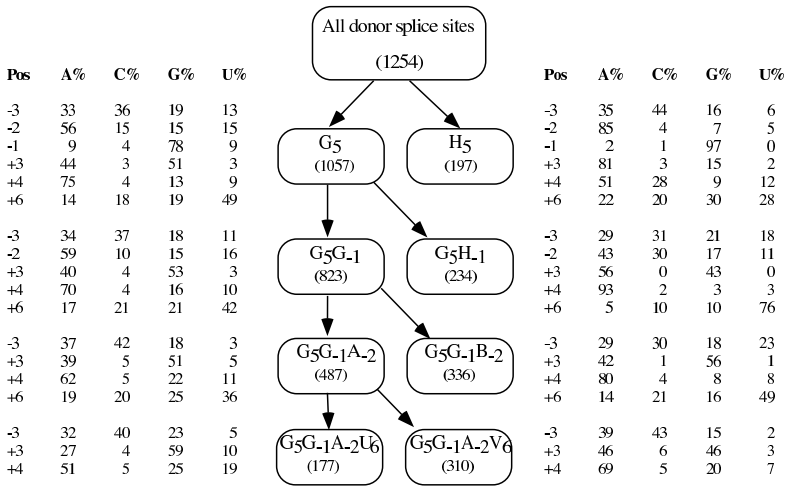
In the cases of start codons, stop codons, acceptors and donor sites, the bias in composition is position dependent. For example, a start codon is required to have an "ATG" sequence at the predicted site. To take advantage of this, gene predictors use a model called a position-specific scoring matrix (PSSM). This is also referred to as a weight matrix model (WMM).

The idea is that the area being predicted with a specific consensus site is given different probabilities for each base at each position. The bases surrounding the consensus are often modeled as well, if a bias is detected. An example of this is shown in Table 5.1.

Higher-order position-specific scoring matrices

Often more information can be gained by including more bases of context in each position of a PSSM can yeild a better predictor. The basic idea is to use a Markov chain at each position in the PSSM. The result is called an n^{th} order PSSM or weight array model (WAM). The acceptor site is modeled with a 2^{nd} order WAM.

When data was sparse (i.e. when Genscan was released), the data was underfit to a higher-order PSSM. Burge introduced the windowed weight array model



Pos	A%	C%	G%	U%
-3	33	36	19	13
-2	56	15	15	15
-1	9	4	78	9
+3	44	3	51	3
+4	75	4	13	9
+6	14	18	19	49
-3	34	37	18	11
-2	59	10	15	16
+3	40	4	53	3
+4	70	4	16	10
+6	17	21	21	42
-3	37	42	18	3
+3	39	5	51	5
+4	62	5	22	11
+6	19	20	25	36
-3	32	40	23	5
+3	27	4	59	10
+4	51	5	25	19

Pos	A%	C%	G%	U%
-3	35	44	16	6
-2	85	4	7	5
-1	2	1	97	0
+3	81	3	15	2
+4	51	28	9	12
+6	22	20	30	28
-3	29	31	21	18
-2	43	30	17	11
+3	56	0	43	0
+4	93	2	3	3
+6	5	10	10	76
-3	29	30	18	23
+3	42	1	56	1
+4	80	4	8	8
+6	14	21	16	49
-3	39	43	15	2
+3	46	6	46	3
+4	69	5	20	7

Figure 5.7: The maximal dependence decomposition estimated for Genscan.

(WWAM), a model which averages the examples in surrounding positions in order to give a closer fit to the data. Now that data is typically sufficient to estimate reliable parameters, this is no longer used, but remains in the iParameterEstimation as an artifact.

Sequence decision trees

Burge found relationships in donor sites between distal bases in the typical 9 base long WMM model. In order to compensate for this, he used something he called the maximal dependence decomposition, and that we call the sequence decision tree.

The idea is that χ^2 statistics are applied to all pairs of positions to determine statistically significant relationships between distal positions, and then the pair with the greatest difference are "decomposed" into two different WMMs, and the cycle is repeated until all statistically significant pairs of positions. Burge's results are shown in Figure 5.7.

We rely on Burge's estimation and do not re-estimate the decomposition. Parameters are reestimated for each of the leaf nodes in Figure 5.7. In addition to the weight matrix models, a transition probability is applied to coming from the root node to each of the leaf nodes.

5.4.3 Isochore-divided models

Several biases were found in [4] that show that the distributions of higher-order Markov chains, duration distributions, priors and transition functions look different depending on the G+C% content, i.e. the percentage of G's and

C's in the target sequence.[3] Some of the models are thus divided into isochores, and used depending on the G+C% of the input sequence to Twinscan/N-SCAN.

The current method for estimating parameters is to consider 1,000,000-base non-overlapping windows for their G+C% content. All the models within each window are scored dependent on the G+C% content of that window.

When defining isochores, a sequence is said to be in a certain isochore if it has less or equal the content of the isochore number, and more than the smaller isochore number. For example, if the isochores were 43%, 51%, 57% and 100%, anything more than 57% would go in the 100% isochore bucket, anything more than 51% and less or equal to 57% would go in the 57% bucket and so on. All isochore models must have a 100% bucket defined.

5.4.4 Conservation models

The primary contribution of Twinscan/N-SCAN to the gene prediction world is the use of conservation models. With Twinscan, a sequence called the conservation sequence (conseq) is fed in along with the target sequence to the gene predictor to enhance prediction. Twinscan uses a separate set of models for the conservation sequence, mirroring the target sequence models. The scores for both are "multiplied", i.e.

$$\Pr(\mathit{feature}|\theta) = \Pr(\mathit{DNA}|\theta)\Pr(\mathit{Conseq}|\theta)$$

under the assumption that the conservation and target sequences are independent. The models used for conservation sequence are higher-order Markov chains and weight array models.

5.4.5 Bayesian network tree models

N-SCAN implenets Bayesian network tree (BNTREE) models. These are detailed in [8]. Unlike the conservation sequence, these model multiple alignment phylogenies. The probabilities are factored such that the original Genscan $\Pr(\mathit{DNA})$ is factored out so the probabilities again look like

$$\Pr(\mathit{feature}|\theta) = \Pr(\mathit{DNA}|\theta)\Pr(\mathit{malign}|\theta)$$

Sam Gross, author of N-SCAN, developed an expectation-maximization algorithm to estimate the branch lengths on the phylogenetic tree.

5.5 Hacks

There are several implementation details which are slightly different from the theoretical perspective on computing the optimal path through a gHMM. Here are a few:

[3] The percentage of G's and C's is the same in the forward strand and the reverse strand since G and C are complements.

5.5.1 Initial probabilities

In HMMs, initial (prior) probabilities for each state are correctly estimated by the frequency per position of each state. In our coin example, if our friend used an unfair coin a quarter of the time and a fair coin the rest of the time, the priors for the Fair and Unfair states would be .75 and .25, respectively.

In real training examples, however, the true mass of intergenic region is unknown, since genomes are not fully annotated. Thus the initial probabilities are specified in the gHMM file by the user as estimates for the frequency of intergenic region and introns in the target genome.

Exons are falsely given zero initial probability, because predicting a gene that starts in the middle of an exon is impossible, as reading frame cannot be known with out a start codon at the beginning (or a stop codon on the reverse strand).

5.5.2 Transition probabilities

There is a tendency for our gene predicting gHMM to predict fewer exons per gene than the actual average exons per gene. In order to preven this from happening the transition probabilities to terminal exons (and initial exons on the reverse strand) are fixed at a low probability. Similarly, transitions *into* single-exon gene states are assigned low probability.

5.5.3 Log-probability space

When computing the optimal path through a gHMM, probabilities of entering, transitioning, emitting and leaving are all multiplied together, one for each letter in the input sequence. Since all probabilities are less than one, for any substiantially long sequence, the probability is going to be very low.

Computers have are only finitely precise, that is they can only compute numbers out to a limited number of decimal places. This presents a problem.

Part of the solution is to convert these probabilities into logs of probabilities and add them together instead of multiply them. This is mathematically legal, and the maximum path will still be maximum even though the numbers have been transformed. That is for any sequence S, and HMM parameters θ, the maximum parse F holds the property

$$\arg\max_i \log(\Pr(F_i|S,\theta)) = \arg\max_i \Pr(F_i|S,\theta)$$

In practical terms, this is done for all duration and emission probabilities, and stored as such in the parameter file.

5.5.4 The null model

Although the numbers will no longer become too small when converted to log probabilities, the numbers will all be negative. For any substantially large se-

quence, adding the probabilities together will go beyond the lower limits of the computer's capacity, and will again present another problem.

Enter one of the most confusing aspects of parameter estimation: the null model.

The general idea is that we can pick any model from the HMM and use it as a global null model, or negative model. Here the term "model" is used a little loosely: it refers to a particular phenomenon, and not a probabilistic model. Generally, the null model is comprised of non-coding regions, intronic and intergenic.

iParameterEstimation is less ambiguous in its terminology for null models. It asks the user to input null "regions" relative to the intergenic, and for each model that has a corresponding null model, that is considered an instance of a null "model".

How does this help? If we take all the emission probabilites and divide them (subtract in log space) by the probabilities under the null model, the log probabilities will no longer all be negative. Specifically, this is

$$S(o|q) = \log \frac{\Pr(o|q)}{\Pr(o|NULL)}$$

where q is the positive model and o is the observation. We refer to these new transfored probabilities as "log-odds ratios" or "scores". In `iscan` the scores are multiplied by ten and rounded to the nearest whole number. A log base 2 is used.

Observe also, that all states which are scored under the null model (intron and intergenic states) will now have score 0:

$$S(o|NULL) = \log \frac{\Pr(o|NULL)}{\Pr(o|NULL)} = \log 1 = 0$$

This means that the score of any non-null feature will be a likelihood estimate of how much it resembles the non-null phenomenon, e.g. an exon region with a high score will be more likely an exon region rather than a non-coding region on the magnitude of its score. Anything which scores negatively will be more likely a non-coding region.

This, in and of itself, is probabilistically correct, and still yields the same optimal path that the original gHMM did. The proof of this is beyond the scope of this book.

Hacks to the null model

Although the Twinscan/N-SCAN null model asserts that the null model is tied to intergenic and intron (non-coding) content, the actual null model is estimated differently. For DNA content, parameter estimation scripts use the content of the first 1000 bases of the first intron after the initial exon as examples for null regions. The conservation models in both Twinscan and N-SCAN correctly use both intergenic and intron sequences for examples of non-coding sequence.

All the WAMs and WMMs have their own null model, determined by "pseudo-sites". iParameterEstimation searches the null examples for the consensus site of each of the signal models (start, stop, donor, acceptor) and counts those toward the null model. Unlike the content models, a 5^{th} order model is not used, instead iParameterEstimation uses an analagous model. All of these things are explicitly specified in the gHMM file, so none of this is hidden from the user.

In addition, at consensus sites such as "GT", "AG", "ATG", and so on, the null model is a bit different. Although the null model is estimated at pseudo-sites to help distinguish between true positives and false positives, the null scores at consensus is actually estimated from the overall base composition bias of each of the consensus characters. For example, if a genome were compeltely equally biased, ($\Pr(A) = \frac{1}{4}$, $\Pr(C) = \frac{1}{4}$, etc.) the score at the consensus site would be

$$10 \times \log_2 \frac{\Pr(l|q)}{\Pr(l|NULL)} = 10 \times \log_2 \frac{1}{\frac{1}{4}} = 20$$

5.5.5 Inframe stop-codon tracking shadow states

An inframe stop codon can occur across multiple exons, if part of the codon begins at the end of an internal exon. In order to prevent this from happening in the gHMM context, special "shadow" states are added to the gHMM model to track if the beginning of a potnential split stop codon has occurred in the previous exon. These states include the `Exon1T`, `Exon2TA` and `Exon2TG`. These are not modelled as separate states, and receive the same emission probabilities and transition probabilities as their parent states.

iParameterEstimation does not internally acknowledge these, and they are explicitly denoted shadow states in the gHMM file. (More on this later.)

5.5.6 Codon-level explicit length distributions

The evolutionary model for mutation in exons is proposed to be the insertion or deletion of a whole codon instead of a base. Thus the length distributions are modeled as a function of codon-level length probability. Internally, these are still represented as single-base probabilities. To compensate for this, the original histogram is computed as a codon-lev distribution, then output as a base-level distribution, with the same probability assigned lengths 1-3, 4-6, and so on. The total sum of this distribution ends up being 3, as a result, however this is compensated for in the `iscan` code.[4]

[4] At the time of writing this, this is not actually compesnated for in the code, however, this is a bug that should be fixed, and the standard will remain this way.

Chapter 6

Getting Deeper: `feature_maps` and `gHMMs`

This chapter covers many of the main concepts you need in order to understand how to revise the models in iParamterEstimation. The two files involved in tuning the models are the feature map and gHMM files.

The feature map file is a relatively short file that provides a description of which annotation features can be mapped to states in the gHMM.

The gHMM file is a long XML file which completely specifies the gHMM to be used for parameter estimation and gene prediction. Any generalized (or ordinary) hidden Markov model can be specified in this file, but the general expectation is that it relates to gene prediction.

Several example gHMM and feature map files are provided for you with the iPE package, usually under `/usr/share/iPE/conf`. You may never need to edit these files, but being able to do so affords you complete control over the experiment you are running.

You will notice in reading this chapter that some of the elements and attributes are specific to `iscan`. These are all necessary for the correct functioning of `iscan`. I have covered the details in the appendix, however as a rule of thumb, if the particular feature doesn't make sense to you, don't change it. Just copy it from something else. Your life will be easier.

To begin, I'll talk a little bit about the process that iParameterEstimation uses to convert annotations into models (along the way we'll learn how feature maps are constructed), then give an overview of the gHMM file and finally talk a little bit about the available models themselves.

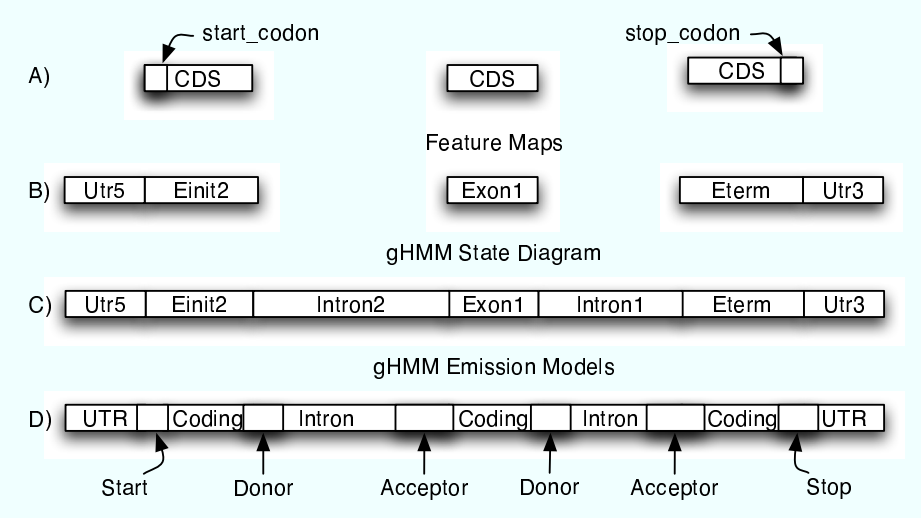


Figure 6.1: An example conversion process for a 3-exon transcript. The GTF feature map converts the GTF annotation in A into a collection of Twinscan features in B. iParameterEstimation then uses the state diagram in order to determine how to fill in the gaps to create the complete state sequence of this transcript in C. Finally, these features are subdivided into particular models in D by the models defined in the gHMM file.

6.1 How iParameterEstimation Converts Annotations

iParameterEstimation can be seen as a mediator between the gene prediction program and the transcription data. It takes in annotations, counts the examples and converts these into model parameters which describe the sequence data input. In order to do this, it has to view the annotations and sequences the same way the gene predictor does. In essence, it converts every annotation into a state sequence that corresponds to the annotation's correct prediction.

6.1.1 Overview

In order to get there, we need to define a number of things:

1. How the annotations correspond to state sequences (`feature_map`s)
2. How the states connect to each other (`states`)
3. How the states are divided up into models (`string_model`s)

This is illustrated in Figure 6.1. As we see in our figure, the first step is to make an intitial pass at converting the annotated features into states. Our format is GTF, so the only annotated features are CDS, and the introns are missing.

In order to fill in these extra features, iParameterEstimation takes a look at the state diagram and tries to find paths through the state diagram which start at, say, `Einit2` and end at `Exon1`. If you take a look at Figure 5.4, you'll notice that there is only state that could go in between these two flanking states.[1]

While the state sequences will help us with the estimation of state durations and transitions, we need to know the boundaries of the specific content models `iscan` applies when scoring potential features. This is done in the model definitions. Each sequence model identifies which state(s) it belongs to as well as where they are along the state sequence. For example, a start codon is defined as beginning 6 bases before the consensus site to 3 bases after the end of the start codon.

6.1.2 Boundary, ordinality and quantity definition: `L`, `N` and `+`

In order for the user to input these features, states, models, and submodels abstractly, iParameterEstimation provides two constructs for defining things according to the boundaries of some object: `L` and `N`. Put simply, `L` is the length

[1] The Genscan HMM differs in the way it tracks reading frame from the Twinscan/N-SCAN hmm. The right overhang is considered, and there are three initial exon states. As a result the roles of internal exons and introns are reversed: an internal intron can transition to any exon, but an exon can only transition to one internal intron. More on this in section 6.2.

of some feature and N is the total number of some group of feature or other object.

Let's see how this applies. Feature maps map coordinates of annotations to coordinates of state sequences (as shown in Figure 6.1). This is done as a collection of feature_mappings which define a particular conversion of a certain kind of feature to a single state. One feature may map to several different states. (Keep reading, you'll see.)

In the case of feature maps for GTF, we see that there is only one feature of interest (for the most part), CDS. Unfortunately, this doesn't differentiate initial exons from internal exons from terminal exons from terminal exons. So we'll want to define our initial exons as being the first CDS feature (on the plus strand) and terminal exons as being the last CDS feature in a multi-exon gene. This is done as follows (taken from the gtf_map.xml file):

```
 <feature_mapping state_name="Einit0" feature="CDS"
    first_feature="0" last_feature="0"
    number_in_transcript="2+"
    state_region_start="0" state_region_end="L-1" />
<!-- ... -->
<feature_mapping state_name="Eterm" feature="CDS"
    first_feature="N-1" last_feature="N-1"
    number_in_transcript="2+"
    state_region_start="0" state_region_end="L+2" />
```

Here we see how we can get an Einit0 from a CDS feature. First we define which feature of this type it is in the transcript with the attributes first_feature and last_feature. These values mean "the first feature in the transcript that is eligible for conversion to this state is the 0th feature (the first feature in computer speak), and the last feature eligible for conversion to this state is the 0th feature". So in essence, Einit0 must be the first CDS feature.

For the plus strand terminal exon Eterm, we define it as the last feature in the transcript, or the N-1th feature. This means given N features in a transcript, define Eterm as (potentially) being the feature whose ordinal is N-1. Again, ordinals are 0-based, so they are numbered from 0 to N-1.

Next we define the number of features in the transcript that contains it. By saying there must be 2+ (two or more) features, we are asserting that an Einit0 or Eterm only occurs in a multi-exon gene.

After iParameterEstimation determines that the feature given matches the feature mapping qualifications, and the strand and frame match, the coordinates of the feature must be mapped to state sequence coordinates. Again, we use ordinals beginning at 0. For the initial exon, Einit0 spans the entire feature, from 0 to L-1. For the terminal exon, Eterm, an additional 3 bases are taken at the end of the exon. This is because GTF does not include the stop codon in the terminal exon, but iscan does.

> **Sidenote: The 0-based index**
> Computers are quite computer-centric. This is especially the case with indices. When counting objects, they like to start with the number 0 rather than 1. There is a good reason for this. Think of a set of objects, say your fingers. Say you knew where your thumb was, and you knew each of the following fingers was a set distance from your thumb, and you wanted to denote everything as being relative to your thumb. The first item in this set would be your thumb, and would be at index thumb:0, since your thumb is 0 distance from your thumb. Your index finger would be at thumb:1, and so on. This is similar to how computers index memory. The arithmatic behind addressing objects as being some distance from another one is simpler, and thus the first one starts at 0.

A nice byproduct of this abstract definition is that we can use this to map unannotated features by the boundaries of annotations. In Figure 6.1, we see that UTR states are defined, even though no GTF feature corresponds to it. Here is how it's done:

```
<feature_mapping state_name="Utr5" feature="CDS"
    number_in_transcript="1+"
    first_feature="0" last_feature="0"
    state_region_start="-150" state_region_end="-1" />
<!-- ... -->
<feature_mapping state_name="Utr3" feature="CDS"
    number_in_transcript="1+" first_feature="N-1" last_feature="N-1"
    state_region_start="L+3" state_region_end="L+152" />
```

The `Utr5` feature is mapped to being 150 bases upstream of any transcript (a transcript with one or more CDS features, off the beginning of teh first feature), and the `Utr3` feature is mapped as the 150 bases off the end of the transcript (plus three bases for the stop codon, again).

6.1.3 Models and submodels use `L`, too

The last step in Figure 6.1 is to take the state sequence and subdivide them by models. These models are used for estimating parameters for the content of sequences. Different model regions are defined for different sequence types. In the example shown in Figure 6.1, the common DNA models for Genscan and Twinscan are shown.

The conversion is guided by the `string_model` (and their meta-model variants[2]) elements of the gHMM file. A simple example of this the `Start` model (taken from `gHMM.xml`):

[2] more later

```
 <string_model name="Start" source="dna" model="SDT"
        states="Einit0 Einit1 Einit2 Einit- Esngl Esngl-"
        begin="-6" end="5" null_model="1"
        length="3" focus="0" symbols="4" >
    <string_submodel name="NNNNNNATGNNN" zoe_name="ATG"
        model="WMM" begin="0" end="11" ordinal="0"
        length="12" focus="6" symbols="4"  />
    <fixed_string_submodel name="NNN" model="WMM"
        ordinal="1" length="1" focus="0" symbols="4" >
        . . . .
    </fixed_string_submodel>
</string_model>
```

That in itself is a lot to take in, but I'll highlight the important points here. This string model is defined to apply to the states in the `states` attribute. It begins 6 bases upstream of the beginning of the state, and ends 5bases downstream of the beginnning of the state (`begin` and `end`). This means that every time any of these states is found, iParameterEstimation will tag these beginning ranges for counting the `Start` model.

This particular model has two submodels. Without going into details, `iscan` uses these models to insure that only start codons with the ATG consensus get a score greater than negative infinity. The submodel `NNNNNNATGNNN` is defined for coordinates 0 through 11. Unlike the parent model `Start`, the submodel is defined as relative to its parent model's coordinates.

The `Stop` model is defined relative to the 3' end of the feature:

```
<string_model name="Stop" model="SDT" source="dna"
        states="Eterm Eterm0- Eterm1- Eterm2- Esngl Esngl-"
        begin="L-3" end="L+2" null_model="1"
        length="3" focus="0" symbols="4" >
    <string_submodel name="TAANNN" zoe_name="TAA"
        model="WMM" begin="0" end="5" ordinal="0"
        length="6" focus="0" symbols="4"  />
    <!--- Omitted other submodels here -->
</string_model>
```

Here it begins 3 bases before the end of terminal and single exons, and ends 3 bases downstream (including `L`, `L+1` and `L+2`; it takes some getting used to).

Note that we were able to define these models for both plus and minus strand features, even though, for example, the actual coordinates of a stop codon in a minus strand terminal exon would be at the lower boundary coordinate. This is because iParameterEstimation takes care of the reverse-complemented coordinates for you internally. You can think of models as going from the 5' to 3' end, regardless of strand.

6.1.4 What about the rest of the feature real estate?

Some models don't have a specific start and end region. These models are often referred to as the "content" models. iParameterEstimation calls them "default" models (another meta-model[3]), or the models for a feature for which nothing else.

For example:

```
<default_string_model name="Coding" source="dna" model="ISO"
        states="Einit0 Einit1 Einit2 Einit- Esngl Esngl-
            Exon0 Exon1 Exon2 Exon0- Exon1- Exon2-
            Eterm Eterm0- Eterm1- Eterm2-"
        null_model="1" length="1" focus="0" symbols="4"
        data="43.0 100.0">
    <string_submodel name="Coding43.0" model="CDS"
        ordinal="0" length="3" focus="2" symbols="4"
        data="5" />
    <string_submodel name="Coding100.0" model="CDS"
        ordinal="1" length="3" focus="2" symbols="4"
        data="5" />
</default_string_model>
```

The coding model is a `default_string_model`, rather than an ordinary one. Notice that there are no `begin` or `end` attributes; the model is assumed to take up all the space not covered by other models.

6.2 Exploring the gHMM file

In this section, I'm giving a guided tour of the gHMM file, the core configuration file for iParameterEstimation. I'll go over the major sections in the file here, so as to make it easier to browse and understand what role each component plays. In the appendix, there are detailed descriptions of the definition of each attribute and tag in the gHMM file.

The following two sections will go into more detail on the more complecated elements of the gHMM file: the duration and sequence model elements. I'll touch on them briefly here.

A look at the top line of the gHMM DTD file (usually located in `/usr/share/sgml`) gives us an outline of what a gHMM xml file should contain:

```
<!ELEMENT gHMM (author, date, states, zoe_gtf_conversion,
init_model?, trans_model, pseudo_transitions?, state_durations,
null_region_definitions?, sequence_models?)>
```

6.2.1 `author` and `date`

These elements identify the user who wrote these files and the date of creation.

[3] again, more later.

6.2.2 states

The subelements of this element define the states of the gHMM and their outgoing transitions, e.g.:

```
<state name="Intron0" strand="+" frame_name="0"
    type="Internal" init_model="Intron" seq_model="Intron"
    cons_model="INTRONCONS" dur_model="Intron"
    transitions="Exon0 Exon1 Exon2 Eterm" />
```

This is a subelement of the states element in `gHMM.xml`, which comes with your installation of iPE. There are several different attributes here:

- `name` - The name of the state, `Intron0`.
- `strand` - The strand it is on, in this case the forward `+` strand. The reverse strand is denoted `-`.
- `type` - This is a special indicator for the zoe codebase to tell it what kind of duration model will be used. `Internal` indicates that this has a geometric length distribution. The rest are detailed inthe appendix.
- `init_model` - This points to which initial model the state belongs to (detailed later in the file).
- `seq_model`, `cons_model` and `dur_model` - These point to which models (detailed later in the file) are associated with this state. `seq_model` is the DNA sequence model, `cons_model` is the conservation sequence model, and `dur_model` is the duration model.
- `transitions` - This is a list of all the outgoing transitions (arrows in our diagrams in the HMM chapter), to the current state.

Pseudostates are also defined in this section. These states are "shadow" states which don't really act as distinct states from their parent states in the HMM, but are placeholders for states which carry special characteristics. `iscan` uses these states to track in-frame stop codons. Here is an example:

```
<pseudostate name="Einit2TA" frame_name="2TA"
    actual_state="Einit2"/>
```

This state "shadows" the `Einit2` state and takes on the exact same initial and transition probabilities that the `Einit2` state has.[4] They indicate that part of a potential split stop codon was seen at the end of the feature, as indicated by the `frame_ name` attribute.

[4] This will mean that the total initial and transition probability will add up to more than one, but since these states simply indicate that a T, TA or TG was seen at the end of the feature, they do not represent anything different in terms of the gene prediction.

Sidenote: Reading frame
Reading frame is a frustratingly tedious concept. The idea of reading frame is introduced to help guide tools dealing with coding sequence as to where the codon positions are with respect to the beginnings and ends of exons. Internal exons will sometimes have one or two bases at the beginning and/or end which are a part of a codon that has been split at the splice sites.
iParameterEstimation handles this by considering each feature, regardless of strand, by the bases on the left and right sides of the feature which are not a part of any whole codon. This is pertinent when defining exon states, for example:

```
<state name="Exon0" strand="+"
    start_frame="N" end_frame="0"
    frame_name="0" type="External"
    seq_model="Exon" cons_model="CDSCONS"
    dur_model="Exon"
    transitions="Intron0" />
```

The `start_frame` and `end_frame` attributes define the expected right and left overhang of a feature. Valid values are "0", "1", "2" and "N". "N" indicates that any overhang is valid.
In `iscan`, the convention is to define a state's frame by its right overhang. Since it scans from the beginning to end of the sequence, this makes it easier to track frame. You'll notice that there are 3 `Einit` states, one for each right overhang, and three `Eterm-` states, and the `Eterm` and `Einit-` states are singular. This is because every predicted terminal exon on the forward strand and initial exon on the reverse strand are required to round the transcript length off to a multiple of three, and thus there may be no right overhang on either of these features.
Notice, also, that every coding exon which transitions to an intron state only transitions to one intron frame type. When `iscan` is checking for a candidate exon, it makes sure that the left overhang of the current exon is compatible with the right overhang of the previous exon (they add up to 3).

6.2.3 zoe_gtf_conversion

This is an `iscan`-specific feature which helps `iscan` decide how to output the features it predicts. This is a code-level feature, and changing it or adding to it will not do anything without changes to the `iscan` code. There are two general types of conversion: ones for 5' UTR-predicting `iscan` and ones without UTR features. The following is for non-UTR predicting `iscan`:

```
Einit => start_codon
Exon  => CDS
Eterm => stop_codon
Esngl => zEsngl
```

The UTR-predicting version adds the following lines:

```
Epa => 5UTR
Ep => 5UTR
Ea => 5UTR
Enc => 5UTR
```

6.2.4 init_model

iParameterEstimation refuses to estimate this, since no annotation can be reliable enough to truly represent the prior probability of each state in the system. (This is especially a the case with intergenic regions.)

Instead, iParameterEstimation provides you with a framework to easily take known statistics about the overall composition of different features in a genome, and apply them to your states. An example of this is the following:

```
<init_model isochores="43.0 51.0 57.0 100.0">
    <init_prob name="Intron" probs="0.0944 0.1128 0.3378 0.3900" />
    <init_prob name="Inter"  />
</init_model>
```

In the `init_model` tag, the attribute `isochores` defines the isochore subdivisions of the initial model. Since noncoding regions are estimated to have different biases at different levels of G+C% content, we define the priors differently at each level. The total mass of introns at each G+C% level is defined in the `probs` attribute of the `init_prob` element. Note that nothing is defined for the intergenic region probabilities. This is because the intergenic region is bound, and that the remaining probability mass will be assigned to the `Inter` model after all others are added up.

iParameterEstimation will evenly distribute the probability mass for the `Intron` model to each of the states defined above that claim to be a part of the `Intron` probability model, with the exception of the pseudostates, which, as described above, simply shadow their parent states.

6.2.5 trans_model

The transition model serves two purposes: to define the isochores for transitions and to mark some of the transitions to have a set probability which might be different from the actual observed probability. This is done to "tweak" some of the predictions, if a transition is overrepresented with respect to the true biological frequency. Here is an example:

```
<trans_model isochores="43.0 51.0 57.0 100.0" pseudocounts="1" >
    <fixed_transition from="Utr5" to="Esngl"
        values="0.001 0.001 0.001 0.001" />
```

We see that 4 different isochores are used for transition probabilities, the transition probabilities are given 1 pseudocount.[5] Here we are fixing the transition to the single exon state to be low to prevent the common over prediction of single exon genes in our gene finder.

6.2.6 pseudo_transitions

This section defines all transitions which are not explicitly modeled in our parameter estimation. This is used to assign probabilities to transitions between in-frame stop codon tracking states. All transitions here are assigned the same probability as the "parent" transitions defined. For example:

```
<pseudo_transition source="Einit1" dest="Intron1"
    pseudo_source="Einit1T" pseudo_dest="Intron1T" />
```

Here the "parent" transition is `Einit1`→`Intron1`, and the pseudo-transition is `Einit1T`→`Intron1T`. `Einit1T`→`Intron1T` will receive the same transition probabilities as `Einit1`→`Intron1`.

There are a lot of these, but fortunately you probably will never have to understand or use this feature.

6.2.7 state_durations

This will be covered in the next major section.

This gets into some of the meat of the gHMM file. Duration models are very important to the way a gene predictor runs. In general, the coding exons are modeled with explicit durations, i.e. a histogram model constructed from the examples fed into the training software, and the other states are modeled with geometric length distributions. It is sometimes beneficial to use an explicit duration model for introns of a certain length. A lot of tuning can go into this. Details of duration models covered in the next section.

[5] The pseudocount here is not used for underrepresentation, but instead for the fact that some transitions are never observed in the annotations, and thus some count needs to be assigned to them.

6.2.8 null_region_definitions

This deceptively short section is very important to the estimation process. This defines what sections of the annotation examples will be used for sampling the null models of each of the positive models. Somewhat confusingly, they are defined only for the plus strand, even though they are used with states on both strands.

```
<null_region_definitions>
    <null_region_definition seqtype="dna"
        states="Intron0 Intron1 Intron2"
        start="0" end="999"
        first_feature="0" last_feature="0"/>
    <null_region_definition seqtype="dna"
        states="Intron0- Intron1- Intron2-"
        start="L-1000" end="L-1"
        first_feature="N-1" last_feature="N-1"/>
    <null_region_definition seqtype="cons"
        states="Intron0 Intron1 Intron2 Intron0- Intron1- Intron2-"
        start="0" end="L-41"
        first_feature="0" last_feature="N-1"/>
 </null_region_definitions>
```

Since the null model often differs between different sequence types, one or more null region is defined for each of the sequence type. iParameterEstimation applies these region definitions to the annotations and earmarks them for counting in the null models. The `states` attribute defines which states these definitions will be applied to in the annotations. The `first_feature` and `last_feature` attributes define which features, relative to the transcript, to consider for the The `start` and `end` attributes define where in the annotation of this state to define a null region. For example, if it found an `Intron0` at the coordinates 5,000 to 7,000 after an initial coding exon, the coordinates 5,000 through 5,999 would be earmarked for null-model counting.

6.2.9 sequence_models, etc.

The final section of the file includes all models that apply to the actual content of the input sequences, including the DNA, conservation sequence, alignment, and/or EST sequence, depending on which flavor of `iscan` you intend on training. All of these models comprise the emission probabilities in our gHMM. They are independently tied to different states, depending on the model. For example, the acceptor model is tied to all internal and terminal exon states, and the coding model is tied to all coding exon states.

6.3 Duration Models

Duration models are tied to the states which claim them in the `states` section with the `dur_model` attribute. No states are explicitly mentioned in this section. Here is an example of an explicit duration model for initial coding exons:

```
<duration_model region="Einit" isochores="100">
    <duration_submodel isochore="100" distributions="2">
        <duration_distribution model="DEFINED" min="1" max="6000"
            length_unit="3" pseudocounts="0." smoothing="gaussian"/>
        <fixed_duration_distribution model="CONSTANT" min="6001">
            -300
        </fixed_duration_distribution>
    </duration_submodel>
<duration_model>
```

Notice there are several layers here. First there is the `duration_model` which defines the name and the different isochores used for this model. In this case, only one is used, so the generic all-isochores "100" is used.

Next is the `duration_submodel` which defines the entire distribution for a particular isochore level. In this case there is only one.

All `duration_submodel`s must define some probability for every value from 1 to infinity. This is accomplished with multiple `duration_distribution`s and/or `fixed_duration_distribution`s.

A duration distribution whose `model` attribute is `DEFINED` has an explicit duration model. The `min` and `max` attributes define the range for an explicit duration model. (No `max` means that the distribution goes to infinity.)

A `fixed_duration_distribution` is one which is defined by the user before the estimation is run and remains the same in the output parameter files. These are commonly placeholders to define an upper bound for a duration distribution, as in this case, or to define a distribution which cannot be estimated from the examples, such as intergenic region.

Here is an example of a `GEOMETRIC` intron distribution with four isochores:

```
<duration_model region="Intron" isochores="43.0 51.0 57.0 100.0">
    <duration_submodel isochore="43.0" distributions="1">
        <duration_distribution model="GEOMETRIC" min="1" />
    </duration_submodel>
    <duration_submodel isochore="51.0" distributions="1">
        <duration_distribution model="GEOMETRIC" min="1" />
    </duration_submodel>
    <duration_submodel isochore="57.0" distributions="1">
        <duration_distribution model="GEOMETRIC" min="1" />
    </duration_submodel>
    <duration_submodel isochore="100.0" distributions="1">
        <duration_distribution model="GEOMETRIC" min="1" />
    </duration_submodel>
```

```
</duration_model>
```

6.3.1 Smoothing

Often, an explicit length distribution will be smoothed in order to prevent large differentials between two close lengths, and to give an approximation of the "real" length distribution. (For a visual example of this, see the dotted line in Figure 5.5.)

Two methods are available for this: kernel-Gaussian smoothing and plain Gaussian neighbor smoothing. There are different benefits to each, but the general effect is the same. For a more detailed treatment of this, see the appendix.

6.4 Sequence models

Sequence models are the core models for a gene predicting gHMM. The other models do have an influence on the way things are predicted, however DNA content and conservation levels truly guide the predictions.

I introduced you to sequence models in Section 6.1. There you saw how they play a role in subdividing the annotation beyond the state sequences. This is a characteristic common to all sequence models. In this section we will discuss the parameter aspect of sequence models, their structure and use.

All gHMM sequence models, regardless of what type of sequence they model, are placed between the `sequence_models` tags.

iParameterEstimation was built to implement any type of model which could be estimated from a sequence. That is to say, in the future, more models may exist, or you may find it possible to implement your own model with the source code.

Here is an example of a model:

```
<default_string_model name="UTR" source="dna" model="LUT"
    null_model="1" null_params="1"
    states="Utr5 Utr5- Utr3 Utr3-"
    length="6" focus="5" symbols="4"  data="order=5"/>
```

In section 6.1, we discussed the roles of the `begin` and `end` attributes, as well as the role of the `default_string_model` tag. In this section we will be focusing in on the `model` and `data` attributes.

The model attribute defines what class of models this belongs to. A class might be something like a Markov chain, a weight array matrix or a sequence decision tree. In our above example, the class is `LUT`, or a Markov chain. The `data` attribute is a special attribute that changes in meaning depending on the model class. Internally, iParameterEstimation passes the value of the `data` attribute onto the model class, and the model class attempts to parse it.

Data attribute takes the format of `setting=value`, where `setting` indicates what the parameter is describing (in the above example, the order of the Markov chain) and `value` is the value of that parameter. There are no spaces in any

values, and as of now all values are numbers. A model can have any number of settings that are meaningful to it.

What follows is a description of each model class. If you haven't already, refer to Section 5.4 for a (slightly) more detailed treatment of these models.

6.4.1 A note on `length`, `focus`, and so on

A long time ago, it was my ulterior mission to obliterate the cryptic `.zhmm` format. The polymorphic fields of "`length`", "`focus`" and friends (listed in more detail in the appendix, I hope), really will never be documented, because they are just too stupid and confusing. Had we transitioned the zoe code base to interpret xml files, indeed, we would not have this problem, and specifying an HMM would be a feasible user task. As it stands, it is really only an lab-internal task. Had I had time, had someone else wanted to, had gene prediction not petered off, whatever, this might be different.

6.4.2 The model classes

`LUT`

This model is simply a Markov chain model. It has no fancy bells or whistles, it simply estimates the conditional probability of a sequence character given the context, depending on the order of the chain. The only data setting is `order`, which defines the order of the Markov chain.

`CDS`

This model is the 3-periodic Markov chain model. This is almost identical to the `LUT` class, except that it estimates 3 Markov chains, one for each codon position.

The output is somewhat peculiar as compared to other models. In `zoe` format, iParameterEstimation automatically outputs 3 `LUT` submodels, one for each codon position. In `xml` format, iParameterEstimation simply outputs 3 markov chains in order.

The only data setting is `order`, which defines the order of each Markov chain.

`WMM`

This model is a weight matrix model. It is simply a position-specific weight matrix. It assigns a probability to each character at each position for the entire length of the model. It has no data settings.

`WAM`**, or** `WWAM`

A `WAM` model is a weight array matrix. It is like a `WMM`, except that at each position, a Markov chain is defined. In essence, the `WMM` is actually a 0th order `WAM` (0th order meaning no context).

The WWAM is a special kind of WAM designed for Genscan. In order to fit the parameters for some models, such as the acceptor, the count data needed to be augmented. To do this, Chris Burge added in the counts in the neighboring 2 positions on either side of the target position. This bulked up the counts. This was ideal for the situation where the information was not consistently positioned at a particular place, but may be in the neighborhood of 5 positions, and when the model was overfit.[6]

A strange fallout of the zoe implementation is that only the keyword WWAM is recognized. It means the same thing as WWAM for scoring purposes. So in all cases of WAMs, a zoe_model attribute should be set to WWAM.

The only data setting for a WAM is the order which is the order of each Markov chain. WWAM has an additional windowRadius parameter which indicates the number of neighboring positions to count for each position.

MARG_WAM

This model is a hack. In reality, it is simply a WAM. It allows you to output a lower-order WAM in a higher-order format. You will see how this is useful in the SPLIT section.

The data settings are inherited from the WAM model class, and has the additional (required) printedOrder setting. This indicates the (higher) order to print the WAM out in.

SPLIT

This model is also a hack. It allows you to combine two different models into a single model and output it as though it were a single model.

This is useful in particular for the Acceptor model proposed by Burge. In his model, the first 20 bases are modeled with a 2nd order WWAM and the final 23 bases are modeled with a 1st order WAM. For zoe the Acceptor model is hardcoded and expected to arrive in the parameter file as a single model. The SPLIT mechanism provides a way to do this, putting the two different models under a single guise. Since the models have different context levels, using a MARG_WAM will fix this.

Typically, the zoe_model attribute is used to set the name of the model as it appears to zoe, which is typically different (in the Acceptor model, WWAM).

The zoeHeaderEnd data setting is used to put an additional ending on the zoe header of the model. Additionally, a printSubmodelsToZhmm value of true or false will indicate whether to actually divide the output .zhmm text into submodels or to mimic a single model as the concatenation of two or more submodels (default behavior).

[6] This isn't often necessary anymore, since we have sufficient sequence data to fit most models.

`ISO`

This model allows you to subdivide the parameters for a particular model in isochore-dependent buckets. For example, biases in coding content are found in human between all regions with 43% or less G+C content, and more than 43% G+C content. The CDS model is thus subdivided as such.

The required `ISOs` data setting is a comma-separated list of isochore levels to subdivide the models into.

`SDT`

An `SDT` is a sequence decision tree. See section 5.4 for more information on the model itself. The model generally contains submodels which are defined by some consensus sequence. All submodels with this consensus sequence get counts in each particular bucket.

Sometimes this model is used to simply filter out models which do not have an expected consensus, for example the start codon model. All sequences which have the ATG consensus at the expected position are given counts, and the ones which do not are thrown away.

Currently, no data settings are used.

6.4.3 The five meta-models

You may have noticed in the previous section that the example elements describing the models had differing tags. These tags denote what meta-model this is. The elements to describe the actual models are all defined in the same way, with different classes of `string_model`s.

There are five different meta-models. They serve to describe how the model is to be converted from the annotation they originate from.

- `default_string_model` - This is the vanilla string meta-model which applies to all parts of the states it is tied to. For example, coding exons have a 3-periodic Markov chain model in all sections not covered by the acceptor, donor or start or stop codon models. Thus, the `CDS` model generally falls under the `default_strong_model` meta-model category.

- `string_model` - This is a fixed-length meta-model (i.e. has a defined beginning and end relative to the feature coordinates). This is commonly used for position specific models.

- `fixed_string_model` - This is a model which has no estimated parameters. The parameters are contained within the text betweent the start and end tags, and are assumed to have some meaning to the program (usually `iscan`) parsing the parameter file. A common example is the signal peptide model (`SIG_PEP`) which is not commonly annotated, so a fixed model is passed through iParameterEstimation to the paramter file.

- `string_submodel` - This is a model which is a submodel to some parent model, contained within a `default_string_model` or `string_model`. Note that only certain models, in particular `ISO` and `SDT` in the current implementation, will accept submodels.
- `fixed_string_submodel` - This is a submodel for which no parameters need be estimated. This is commonly used for the sake of discarding sequences which do not fit a consensus. In the start codon model, for example, all sequences without the ATG consensus get a negative infinity score.

Hopefully now you have a basic vocabulary with which to parse a gHMM file for your own purposes. To edit the file, you need to understand the particular attributes and their meanings. You can use the appendix to do this.

Chapter 7

Common Tasks with gHMM Files

7.1 Removing/Changing Isochores

It is often the case that some of the isochores (aka G+C%) simply aren't found in the DNA sequence of the species you are estimating parameters from. In this case, you will want to eliminate or change the isochores that you estimate parameters from.

A few "isochore hotspots" that I'll list below will help you find where the isochores of interest are in gHMM files. Currently, these are the only places where parameters have been divided by isochore.

- The initial probability model
- The transition probability model
- The intronic duration model
- The intergenic duration model
- The coding model

Generally speaking, human parameters use the isochores 43%, 51%, 57% and 100% for the initial model, transition model, intron length model and intergenic length model. The coding models are divided into the isochore levels of 43% and 100%. However in *D.melanogaster*, only 43% and 100% are used across the board. In plants, no isochores are used. In *C.elegans*, 32%, 35%, 39% and 100% are used for all hotspots except the coding model, which uses 32% and 100%.[1]

The problem of dealing with isochores usually arises when too few or no examples are found in one or ore isochore bucket. You might receive a warnings such as

[1]The provided `C.elegans` gHMM file was used for a competition where too few exaples were given to cover the isochores, so no isochores are present.

```
0 samples in duration distribution Intron
for isochore 100.0.  There is probably no
sequence in the 100.0 isochore group
0 samples in duration distribution Intron
for isochore 43.0.  There is probably no
sequence in the 43.0 isochore group
0 samples in duration distribution Intron
for isochore 57.0.  There is probably no
sequence in the 57.0 isochore group
WARNING: scoring GEOMETRIC duration Intron with mean 0 because no samples were found.
WARNING: scoring GEOMETRIC duration Intron with mean 0 because no samples were found.
WARNING: scoring GEOMETRIC duration Intron with mean 0 because no samples were found.
```

If you receive warnings that 2 or 3 isochores have no examples, then you should probably change all isochores to only 100% in all these places. Otherwise you might try consolidating two of the isochore buckets. You can do this by changing, for example, this:

```
<init_model isochores="43.0 51.0 57.0 100.0">
```

to this:

```
<init_model isochores="100.0">
```

7.2 Changing the Intron Duration Model

It is often desirable, especially for species with shorter introns on average, to make the intron duration model into an explicit model, (i.e. change the model to `DEFINED`). This is accomplished fairly easily: simply add something like the example shown in figure 7.1. (See section 7.4 for details on the geometric tail.) The caveat is that you must change the `state type` attribute to "`Explicit`" for all intron states, or `iscan` will reject the parameter file.

7.3 Smoothing Methods on Durations

As mentioned in section 6.3, it is possible to smooth a length distribution by 3 methods: the kernel-gaussian smoother and the plain gaussian smoother. The kernel-gaussian smoother is slightly better in that it applies a global function to the distribution instead of the local function applied by the gaussian smoother. You can find information in the book [3], and it is used in Augustus.

In order to apply any of these smoothing methods to simply set the `smoothing` attribute to either `kernel` or `gaussian`.

```
<duration_model region="Intron" isochores="43.0 100.0">
  <duration_submodel isochore="43.0" distributions="2">
    <duration_distribution model="DEFINED" min="1" max="500"
       smoothing="kernel"/>
    <duration_distribution model="GEOMETRIC" min="501"
       normalizing="match_with_density"/>
  </duration_submodel>
  <duration_submodel isochore="100.0" distributions="2">
      <duration_distribution model="DEFINED" min="1" max="500"
         smoothing="kernel"/>
      <duration_distribution model="GEOMETRIC" min="501"
         normalizing="match_with_density"/>
  </duration_submodel>
</duration_model>
```

Figure 7.1: Code to use explicit introns with a geometric tail.

7.4 Fitting a geometric tail to an explicit length duration

It is common practice to use a piecewise model for introns, modeling the first 500 or 1000 lengths explicitly with a histogram, and the rest with a geometric function. The resulting function is tricky to properly estimate, because the probability at the end of the explicit length section must match the probability at the beginning of the geometric length distribution. iParameterEstimation provides an automated way of doing this. Here is an example:

In this example, the first 500 lengths of an intron are modeled with a histogram, and all lengths from 501 to infinity are modeled with a geometric distribution. The attribute-value pair `normalizing="match_with_density"` tells iParameterEstimation to match the previous model at its beginning with the appropriate overall density of each respective model. This means that, if half the samples are of length 1 to 500, then both the histogram and the geometric tail will have equal density. The details of the math behind this are covered in the appendix.

In theory, other methods could be used, but this is the only method currently implemented.

7.5 (Un)tweaking Transition Probabilities

Genscan (and likewise, Twinscan and N-SCAN), with properly estimated parameters had a tendency to predict genes that were shorter than the average on the human genome. For this reason, all these packages adjust the transition probabilities for transcript-ending states to particularly low values. In iPE, this is done in the `trans_model` section:

```
<trans_model isochores="43.0 51.0 57.0 100.0" pseudocounts="1" >
  <fixed_transition from="Utr5" to="Esngl"
    values="0.001 0.001 0.001 0.001" />
  <fixed_transition from="Intron0" to="Eterm"
    values="0.0005 0.0005 0.0005 0.0005" />
  <fixed_transition from="Intron1" to="Eterm"
    values="0.0005 0.0005 0.0005 0.0005" />
  <fixed_transition from="Intron2" to="Eterm"
    values="0.0005 0.0005 0.0005 0.0005" />
  <fixed_transition from="Intron0-" to="Einit-"
    values="0.0005 0.0005 0.0005 0.0005" />
  <fixed_transition from="Intron1-" to="Einit-"
    values="0.0005 0.0005 0.0005 0.0005" />
  <fixed_transition from="Intron2-" to="Einit-"
    values="0.0005 0.0005 0.0005 0.0005" />
  <fixed_transition from="Utr3-" to="Esngl-"
    values="0.001 0.001 0.001 0.001" />
</trans_model>
```

Those states which are final states (reading from left to right) in transcripts get low incoming transition probability. Most species besides humans do not need this tweak, and some actually are handicapped by this tweak. If this appears to be a problem with your gene predictions, consider commenting out all the `fixed_transition` elements.

7.6 Notes on Changing Content Models

Unfortunately, making a major change to a content model is not as simple as changing the iPE parameters. Many of the models are hardcoded into `iscan`, which requires certain models to be implemented in order to function properly. The good news is if you want to implement your own gene predictor, this program can estimate the parameters for you however you want them.

Appendix A

Instance Reference Guide

The instance file, as mentioned earlier, is the main entry point for iPE. It describes every file that will be used for the estimation process as well as some global options to apply to iPE. In this chapter we'll go through all the options and elements present at time of writing.

A.1 Identification section

A.1.1 `author` element

The text should be the name of the author of the instance file.

A.1.2 `date` element

The text should be the date of creation of the file.

A.2 The input files section

All of these elements, except the `gHMM_file` element, have a `basedir` attribute. This refers to the directory in which to search for the files. In essence, it is used to shorten the file names given between the element tags to its base name without the absolute path. Since they all have the same meaning, no definition is given below for the `basedir` element.

In general, it is advisable to use absolute paths (either with the `basedir` attribute or without) to refer to files, since many files have the same name.

A.2.1 `gHMM_file` element

Indicate the gHMM description XML file to estimate parameters on here.

A.2.2 `feature_map_files` element

Indicate the feature map that should be used to translate features from the annotation to features from the states of the gHMM. Note this is dependent both on the gHMM and the annotation format. For example, the included feature map `gtf_map_utr.xml` converts 5' UTR features into 5' UTR states in the gHMM. For gHMMs that have no 5' UTR states, this feature map will not work and an error will be emitted.

A.2.3 `annotation_files` element

Indicate the annotation files to be used for estimation. Any format, in theory, can be used here, provided it has the right extension. (E.g. all GTF files must end in `.gtf`.) At time of writing, however, there is only one annotation format supported: GTF.

A.2.4 `seq_files` element

NB: In previous versions of iPE, there were several different elements that were used in place of this new, umbrella `seq_files` element. They included `dna_files`, `conseq_files` and `estseq_files`, among others. These elements are equivalent to indicating the sequence type with the `type` attribute. I mention this because there are probably a number of instance files that use this old format floating around.

Indicate the files that correspond to the `annotation_files` above in sequence form. This means that the first annotation file will correspond to the first sequence file here, the second to the second and so on. To belabor the point, there should be the same number of sequence files in each `seq_files` element as there are in the `annotation_files` element.

Any number of types of `seq_files` can be present. The gHMM will indicate which ones are modeled and which are not.

The more sequence files that are used, the more memory iPE will need. Memory becomes a problem in cases where whole-chromosome sequence files are present. If you are running out of memory because of the length of the sequences, you may choose to set the `loadSequences` option to `false`, described below. (You may also split the files up into smaller files, but that is not that fun to do.)

Attributes

- `basedir` - same as always
- `type` - one of the sequence types in iPE

A.3 Options in the instance file

After all the files are listed in the top section of the instance, an `options` element indicates the beginning of the options section. Each option appears as a subelement to the `options` element. The text between the elements' tags indicate the setting of that particular option.

Below is a list of options available at time of writing. The name is listed first, then the type of input expected, and finally the default value of the option (all options are optional). All options with **boolean** type of input should have either `true` or `false` as the option's value.

A.3.1 Runtime options

- `verbose` - **boolean** default `false`

  Whether to emit progress messages. This does not include warnings or debug messages, but clean, clear progress messages. When set to true, output to the message stream (see `messageOutputFile` below.

- `suppressWarnings` - **boolean** default `false`

  Whether not to print warnings about the parameter estimation process. It is not advisable to set this to true unless you are sure everything is working. If you would rather not have the warnings clutter `stderr`, then you can set `warningOutputFile`, described below.

- `debugOutput` - **boolean** default `false`

  Print a massive amount of messages that only make sense to me, the developer, Bob. This should invariably be set to false unless you are making a run that crashes iPE and want to send me a debug file. If this is on, it is best to also set `debugOutputFile` shown below.

- `randomSeed` - **number** default return value of `time()`

  Setting a random seed to a particular value guarantees the same path of execution every time iPE is run. Random numbers are used when sampling models that are given the `sampling_rate` parameter. Setting the random seed is useful for testing purposes, or if you want to see the differences in two runs of iPE, but don't want the differences from sampling to clutter up your diff output.

- `performCount` - **boolean** default `true`

  Have iPE count the examples.

- `performNormalize` - **boolean** default `true`

  Have iPE normalize the counts to probabilities.

- `performScore` - **boolean** default `true`

Sidenote: about the `perform` settings

These features were added at a time when we thought that there would be more uses for iPE. If you really only want to see the counts of a set of examples, then you can disable the other two (`Normalize` and `Score`). You might potentially want to see multi-fasta files of the features themselves. This can be accomplished by disabling all three `perform` settings and setting the `feature` and `model` base directories.
Note that if any of the downstream settings are set to true, the upstream settings are ignored. For example, if you set `performCount` and `performNormalize` to `false`, but set `performScore` to true, counting and normalizing will still be performed, since these are necessary to obtain scores.

Have iPE convert the probabilities to log scores and scale them according to the `scaleFactor` setting, below. All sequence model parameters which receive no counts will be set to the value of the `sequenceNegInf` setting and all duration model parameters which receive no counts will be set to the value of `durationNegInf`.

A.3.2 Output options

- `outputBaseDir` - **path** default “”

  The base directory to output all files, except feature and model files, described below.

- `featureBaseDir` - **path** default “”

  The base directory to output multifasta files of the features. A “feature” is a sequence under which an annotation of one of the states of the gHMMs is present. After converting the annotations to state sequences, iPE will output the sequences underlying the state sequences, including all types of sequences input. If left blank, no features are output.

- `modelBaseDir` - **path** default “”

  The base directory to output multifasta files of the models. A “model” is a sequence under which an annotation of one of the models (e.g. Acceptor, Start, Coding) of the gHMMs is present. If left blank, no models are output.

- `countOutputFile` - **filename** default “”

  The name of the file to output the xml-formatted counts. This file will look very similar to your input gHMM file, except that the elements will

contain text indicating the counts of each parameter. If left blank, no file is output. See the sidebar on intermediate files.

- `smoothedCountOutputFile` - **filename** default ""

  The name of the file to output the xml-formatted smoothed counts. After counting the examples, iPE smooths all the models according to their `smoothing` setting. If left blank, no file is output. See the sidebar on intermediate files.

- `probOutputFile` - **filename** default ""

  The name of the file to output the xml-formated probabilities. This will contain normalized counts in every model which represent probabilities of each event modeled. If left blank, no file is output. See the sidebar on intermediate files.

- `xmlOutputFile` - **filename** default ""

  The name of the file to output the xml-formatted parameters. This will contained the scaled log-probability and log-odds scores. If left blank, no file is output.

- `zoeOutputFile` - **filename** default ""

  The name of the file to output the `zoe`-formatted parameters. This file, unlike any of the above files, can be used with `iscan`, and is probably the only file you are interested in having output. If left blank, no file is output.

- `blacklistOutputFile` - **filename** default ""

  The name of the file to output the list of unused genes. If a gene is discarded because of an error, a warning is emitted, and it is added to the blacklist. This blacklist can be viewed using this feature. If left blank, no file is output.

- `messageOutputFile` - **filename** default "`/dev/stderr`"

  Where to output the progress messages. If left blank, the messages are displayed to standard error.

- `warningOutputFile` - **filename** default "`/dev/stderr`"

  Where to output the warning messages. If left blank, the warning messages are displayed to standard error.

- `debugOutputFile` - **filename** default "`/dev/stderr`"

  Where to output the debug messages. If left blank, the debug messages are displayed to standard error (not advised).

- `zoeCommentsAtEnd` - **boolean** default `false`

  By default, iPE adds a large amount of comments to the beginning of the `zoe` output file indicating all the files used to generate that parameter file. This can get unwieldy, and so it might be desirable to leave these at the bottom so the `zoe` file can be viewed more readily.

A.3.3 Annotation options

All of these features pertain to how the Annotation object and the FeatureMap object filter input annotations. If the annotation is not deemed worthy, it is sent to the blacklist. Here you can do a little tweaking on what stays and what gets the boot.

- `allowPartialTranscripts` - **boolean** default `false`

  If a transcript has no start or stop codon, it is considered partial. If this value is set to true, then these transcripts are allowed to be used for parameter estimation.

- `allowGappedTranscripts` - **boolean** default `true`

  Often the conversion of a transcript from an annotation to a state sequence will leave a gap in the state sequence. This is not usually a bad thing, it just means that the annotation cannot complete the picture of the state sequence. If this is set to false, these transcripts are not allowed.

- `allowBadTransitions` - **boolean** default `false`

  Sometimes two state features from an annotation will butt up against each other, but the two states have no valid transition in the gHMM graph. If this setting is set to true, these transcripts are used.

- `annotationSanityCheck` - **boolean** default `false`

  This setting tells the AnnotationPlugin (usually GTF) to check for obviously bogus transcripts. In general, you should do this before running parameter estimation, and if you are confident of your transcripts, then you may leave this false. If you are doing a rush job, this may be a good idea to set to true.

A.3.4 Model options

- `isochoreWindow` - **number** default `1000000`

  During initialization and loading of a sequences, the sequence is divided up into isochores according to G+C% content. This is done in a non-overlapping window fashion, where each window is assigned a value, and

Sidenote: intermediate files and the null model
Null model counts, smoothed counts and probabilities can be viewed with the intermeidate files. (Parameter files contain no null model scores because they are incorporated into the log-odds ratio.) The way this is displayed is for each parameter that has a corresponding null parameter, the positive model parameter is displayed to the left of a pipe symbol ("—") and the null model parameter is displayed to the right. Here is an example in the start codon model:

```
<string_model name="Start" source="dna" model="SDT"
  null_model="1" states="Einit0 Einit1 Einit2 Einit-
  Esngl Esngl-" begin="-6" end="5" length="3"
  focus="0" symbols="4">
    <string_submodel name="NNNNNNATGNNN"
    zoe_name="ATG" model="WMM" begin="0" end="11"
    ordinal="0" length="12" focus="6" symbols="4">
17.0|169.5  22.0|141.3  42.7|198.3  11.2|176.9
16.2|188.1  31.0|147.9  25.9|175.6  19.8|176.8
22.3|189.0  40.8|124.9  18.5|213.1  11.2|162.8
38.3|165.3  10.0|136.4  37.9|225.9   6.6|163.2
21.8|228.6  38.0|116.4  21.3|204.8  11.7|141.9
10.0|226.9  49.6|172.0  26.2|197.3   7.0| 96.5
92.9|692.8   0.0|  0.0   0.0|  0.0   0.0|  0.0
 0.0|0.0     0.0|  0.0   0.0|  0.0  92.9|692.8
 0.0|0.0     0.0|  0.0  92.9|692.8   0.0|  0.0
23.8|167.8  14.0|139.7  49.0|221.1   6.0|163.8
24.0|162.9  32.5|151.7  26.0|205.1  10.3|172.7
16.6|163.5  20.5|137.2  40.0|186.1  15.7|199.1
   </string_submodel>
    <fixed_string_submodel name="NNN" model="WMM"
    ordinal="1" length="1" focus="0" symbols="4">
. . . .
    </fixed_string_submodel>
</string_model>
```

In this example (which is sparse, by the way, because it is taken from the example packaged with iPE which only uses chromosome 20), the first event, A in the first position of the model, has 17 positive counts and 169.5 null counts.

all models falling inside that window use that window's isochore level. This setting is fairly arbitrary, but we have found that a window size of 1 megabase (default) works well.

- `weightCounts` - **boolean** default `true`

  When converting the input annotations, iPE takes overlapping transcripts into account and segments the annotation at points where the state sequences change. This allows iPE to act as though these overlapping features are partially one type of feature and partially another. This means that if you have an intron overlapping a coding exon, this overlapping region is given half counts as an exon and half as being an intron.

  This feature was envisioned as a way of making the parameter estimation more consistent, and not to give single bases count values of more than one because of annotation bias. In practice, it failed to show any improvement. Leaving this to `true`, however, is an advisable "better safe than sorry" practice.

- `sequenceNegInf` - **number** default `-100`

  The score to assign events for which no examples were found in sequence models. For example, in the coding models, all in-frame stop codons receive a score of -100, because there should be none in the annotation.[1] We have always used -100 for this setting, however this is arbitrary. Leaving this feature "outside" the code avoided difficult-to-find hard-coding in the software.

- `durationNegInf` - **number** default `-300`

  The score to assign events for which no examples were found in duration models. Note that this only really applies (at time of writing) to `DEFINED` models, since all other models have fewer parameters.

- `scaleFactor` - **number** default `10`

  The factor to multiply all log-scores by. This is arbitrary, and has always been 10. Other values might work.

A.3.5 N-SCAN options

- `nscanTopology` - **string** default ""

  The binary tree that N-SCAN uses to model its multiple informants. By "binary" tree, we mean that for every node in the tree, there are two children. Every internal node is represented as [species:number,species:number]. Each of these will imply 3 nodes: an ancestor node, and the two children described between the brackets. These can be built upon each other, for example, [[galGal2:4,[mm5:2,rn3:3]],none]. The innermost nodes, mouse and rat are at the bottom of the tree, an ancestor joins mouse's and rat's

[1] If some are found, a warning is emitted. The count is used nonetheless.

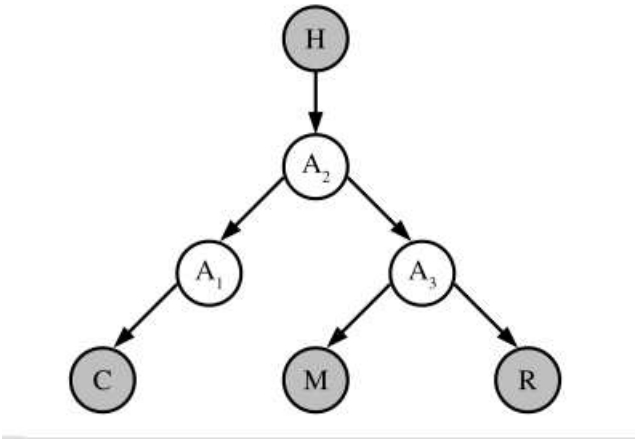


Figure A.1: An example of an N-SCAN multiple-informant tree.

ancestor to chicken, and then one more ancestor joins chicken with a null node.

Figure A.1 shows this example, except that the excess ancestor, A_1, is not present in the tree.[2] Note, also, that the root node is not described in this format. The root ancestor has the target species (in the example, human) as an ancestor node. For more information see [8].

The null node, described earlier, is in theory just a place holder for when there are an odd number of species. However, due to a possible bug in the N-SCAN code, there are reports of crashing when no null ("`none`") node is present. So it is advisable to use a null node even if it is not needed.

- `nscanModel` - **string** default "R1"

  This model is one of R (reversible), G (generalized) and TT (transition-transversion) followed by the order of the model which is one of 0, 1 or 2. The details of these models are discussed briefly in [8], however the only one that has ever been used in production is R1, and so it is not advised to change this without a deep understanding of how N-SCAN works.

- `keepSSFiles` - **boolean** default `false`

  In order to generate N-SCAN parameters, intermediate files called SS (significant statistics) files are output first, and Sam's N-SCAN code takes over from there to maximize the parameters.[3] These intermediate files are the only place counts are emitted, and it might be desirable to keep them

[2]The reason for this is apparent in the paper: the original tree has an ancestor joining human and mouse and rat's ancestor, but this is no longer needed when human becomes root.

[3]This is a very ugly hack, but messing with Sam's code would probably have been worse than rewriting it.

to see how many counts each event received (useful for debugging). The files will be left in the `outputBaseDir`.

A.3.6 Sequence options

- `loadSequences` - **boolean** default `true`

  Whether or not to load the sequences into memory, if possible. Generally speaking, it is much faster to load the sequences into memory because using the disk substantially slows down any operation. However it is at times not desirable to load the sequences when too much memory is required for an instance. Set this to false to read the sequences directly off the disk during estimation.[4]

- `expandAmbigSequences` - **boolean** default `false`

  If set to true, all IUPAC ambiguity codes are expanded into all their possible encodings and each possible one recieves the appropriate fraction of the count. For example, if an R is seen, that base is given half a count for an A and another half a count for a G.

[4] Admittedly, a better medium could have been achieved, but there's only so much time on this planet.

Appendix B

gHMM Reference Guide

This chapter is a fairly dry definition of all the things that can potentially be put in the gHMM file. Hopefully it will clarify most of the details I left out in the main section. If any of these definitions are not clear, feel free to email me and I might respond :)

For an exact definition of the formatting requirements of the gHMM file, you may refer to the `gHMM.dtd` file in the `sgml` path.

B.1 Terms, types and symbols

B.1.1 Terms

- **feature** - In this chapter, "feature" refers to the state features that have been converted from an annotation. A feature is defined by a type (for example, `Einit`) and region (for example, beginning at 1053 and ending at 1225).
- **0-based** - Used to describe any number that begins at 0, instead of 1.

B.1.2 Types

All of the attributes have a type of data that they will be expecting. I will use these shorthand terms to simplify the definitions of the attributes.

- **relative coordinate** - A coordinate defined relative to the beginning or ending boundary of some other parent region.
- **ordinal** - a number which corresponds to some place in the order of elements in which this falls. For example, you might have a group of submodels which have to be in a particular order for them to make sense.[1]

[1] XML does not specify that elements will be parsed in any particular order, so the order sometimes needs to be explicitly laid out in the XML element.

- **number** - a positive number or 0.
- **text** - some meaningful text string.

B.1.3 Symbols

Some of the attributes described within will be used by iParameterEstimation only, will be passed directly onto the zoe parameter file, or will be used both by iParameterEstimation and the zoe code. The ones passed directly on to the parameter file have no meaning to iParameterEstimation, but do have significant meaning to the zoe code. I'm attempting to define some of these here, but zoe being largely undocumented, it is a little tricky and I might miss a thing or two.

In order to denote this for each attribute, I'm putting a symbol before the definition of each attribute to indicate which category the attribute falls under:

- $\odot$ - Used by iParameterEstimation only
- $\ominus$ - Used by zoe only
- $\oplus$ - Used by both iParameterEstimation and zoe.

"$\star$ " denotes a required attribute.

B.2 The author section

The author section contains the author element.

B.2.1 `author` element

Attributes

None.

Text

Should include the name of the author of the gHMM.

Submelements

None.

B.3 The date section

The date section contains the date element.

B.3.1 `date` element

Attributes

None.

Text

Should include the date of creation of the file.

Subelements

None.

B.4 The states section

This section defines all the states in the gHMM, as well as many characteristics of the states.

B.4.1 `states` element

Attributes

None.

Text

None.

Subelements

`state`
`altsplice_state`
`pseudostate`

B.4.2 `state` element

Attributes

- `name` - $\oplus$ **text** $\star$

  The name of the state. May be any text string, but not the same name as another state.

- `strand` - $\oplus$ **text** $\star$

  The strand of the state. Must be either '+' or '-', even if the state doesn't imply a strand. By convention, intergenic states and other non-coding states are given the '+' strand.

- `start_frame` - ⊙ **number** or "N" default "N"

  The number of bases before the first full codon for this state relative to the lowest coordinate. "N" indicates that this number is not defined for this state.

  This, along with `end_frame` is used to determine if the frame of the candidate feature in the feature mapping phase is qualified to be converted to this state type.

- `end_frame` - ⊙ **number** or "N" default "N"

  The number of bases before the first full codon for this state relative to the lowest coordinate. "N" indicates that this number is not defined for this state.

- `frame_name` - ⊖ **text** default "0"

  This is the name passed on to the zoe parameter file which indicates the frame of the state. The zoe convention is to use the right overhang as the frame. If no frame is necessary for the feature, "0" is used.

- `type` - ⊖ **text** ⋆

  This is one of the following special keywords which clue the zoe codebase into what kind of state this is:

  - `Internal` - A state with a geometric length distribution.
  - `GInternal` - This is a bit of a hack. The state is required to have a duration model no matter what, but when a state is of type `GInternal`, it uses the length distribution of the intergenic state.
  - `Explicit` - A state with an explicit length distribution.
  - `External` - A state which has some special meaning to the zoe codebase. All coding exons, promoters and poly-A states are `External`.

- `init_model` - ⊕ **text** default ""

  This is the name of some initial probability model which this state belongs to. This must correspond to the name some model defined in the `init_model` section.

- `seq_model` - ⊖ **text** ⋆

  This is either the name of one of the models in the `sequence_models` section, or, if the state is an `External` state, one of the special names for the state types. This includes `Einit` (initial coding exon), `Exon` (internal coding exon), `Eterm` (terminal coding exon), `Esngl` (single coding exon gene), `Prom` (promoter region) or `PolyA` (poly-adenylation site). Note that in order for these models to function, the following `sequence_models` must be defined: `Start`, `Stop`, `Donor`, `Acceptor`, `PolyA` and `Prom`. These names are hardcoded into the zoe codebase.

- `cons_model` - ⊙ **text** default “”

  The conservation sequence model associated with this state. This is not really used anywhere, but it helps clarify things a bit.

- `phylo_model` - ⊙ **text** default “”

  This is the phylogenetic model associated with this state. This is not really used anywhere, but it helps clarify things a bit.

- `dur_model` - ⊕ **text** ⋆

  This is the duration model associated with this state. The zoe codebase uses this to determine which duration model to look to when scoring the state duration, and iParameterEstimation uses this to associate states with duration models. Note this must correspond to a `region` attribute in one of the `duration_model` elements, defined later in the file.

- `transitions` - ⊕ **text** ⋆

  This is a space-separated list of state names which this state can legally transition into. Note that the names must correspond to `name` attributes of another `state` in the model. No self-transitions are allowed.

Text

None.

Subelements

None.

B.4.3 `pseudostate` element

Pseudostates are “shadow” states, meaning they take on the properties of some other “parent” state. These states are generally used to track pontential inframe stop codons in the zoe code. The key characteristic is that they take on the same initial probability, transition probability, duration and content parameters as their “parent” state, without contributing any mass to the distribution. The initial probability and transition probability distributions will add up to more than one, but since these states don't really exist in the system (i.e. they only indicate whether there was part of a potential inframe stop codon was detected), they aren't counted toward the overall probability mass.

The line in the zoe parameter file will be identical to the one for the pseudostate's “parent” state, excepting the `name` and `frame_name` fields.

This could have some other use besides inframe stop codon tracking, however, at present, there is none.

Attributes

- `name` ⊖ **text** ⋆

  The name of the pseudostate. Must be unique.

- `frame_name` ⊖ **text** ⋆

  The name of the frame. This usually corresponds to the right overhang of the parent state, plus the overhanging bases, i.e. `1T`, `2TA` or `2TG`.

- `actual_state` ⊙ **text** ⋆

  The name of the parent state. Must correspond to a `name` attribute in on of the `state` elements.

Text

None.

Subelements

None.

B.5 The Zoe GTF conversion section

This section is a part of the `.zhmm` file which describes how to convert state-features into GTF featres. This is completely inextensible, and if you wanted to make any meaningful change to this, you would have to actually change the zoe code.

B.5.1 `zoe_gtf_conversion` element

Attributes

None.

Text

As mentioned before, this is really not something you need to change. For reference, there are basically two different values for the text here:

Non-UTR:

```
Einit => start_codon
Exon  => CDS
Eterm => stop_codon
Esngl => zEsngl
```

This is used in any gHMM without any of the 5' UTR "Ea", "Ep", "Epa" and "Inc" states. This includes Twinscan and Twinscan_EST gHMMs.

UTR:

```
Einit => start_codon
Exon  => CDS
Eterm => stop_codon
Esngl => zEsngl
Epa => 5UTR
Ep => 5UTR
Ea => 5UTR
Enc => 5UTR
```

Surprise, surprise, this contains the 5UTR states and features. While there is no use in changing this section, it is necessary for the proper functioning of the zoe code base to include the proper text here.

B.6 The initial model section

Here the model for the initial probabilities are defined. In general, different classes of states, such as Intron, UTR and Intergenic states, are considered together for initial probability. As mentioned earlier, there is no estimation of initial probabilities. iPE simply does some of the bookkeeping for you. I'll stick an example at the end here.

B.6.1 `init_model` element

This encapsulates the entire model. We consider the isochores as the global parameter, since we define a separate distribution for each isochore.

Attributes

- `isochores` - $\oplus$ **space-separated number list** $\star$

  Defines the isochore levels that are present in the initial probabilities model.

Text

None.

Subelements

`init_prob`

B.6.2 `init_prob` element

This is the main player. It defines a state class, and the actual probability of landing in this state class, given any base in genomic sequence.

Attributes

- `name` - ⊙ **text** ⋆

  The name of the class. This should correspond to some of the states' `init_model` attributes in the `states` section.

- `probs` - ⊕ **space-separated number list**

  The actual probability of landing in this state class. There must be as many numbers in this attribute as there are global isochores for the initial model. Note that only one of the state classes can have no `probs` attribute. The one without the probs attribute is considered the "bound" parameter, and is computed by iPE.

Text

None.

Subelements

None.

B.6.3 An example

```
<init_model isochores="43.0 51.0 57.0 100.0">
    <init_prob name="Intron" probs="0.0944 0.1128 0.3378 0.3900" />
    <init_prob name="Utr" probs="0.0049 0.0110 0.090 0.072" />
    <init_prob name="Inter"  />
</init_model>
```

In this example, used in Human, there are four isochores. The two Intron and UTR free parameters have four initial probabilities defined for them. The probability mass for each will be divided evenly between each of the states in the probability class. The Inter class receives the density remaining after all is calculated.

The above example implies that we should see, for instance, one or more states that have `init_model="Intron"` as an attribute. This will result in something like

```
Intron0    Internal  + 0   Intron Intron    0.0157333333   0.0188000000   0.0563000000
```

where this the initial probabilities on the right are some even slice of the pie to the right.

OK, I think I've sufficiently beaten that one to death.

B.7 The transition model section

Here we define the transition models. Instead of actually listing the transitions here, rendering a big mess of XML, the actual topology is defined within the `states` section, where outgoing transitions are defined for each state. Here we define the isochores for the model, and we can also tweak[2] the transition probabilities here.

B.7.1 `trans_model` element

This defines the globals of the transition model: isochore levels and pseudocounts.

Attributes

- `isochores` - ⊕ **space-separated number list** ⋆

  Defines the isochore levels for the transition model.

- `pseudocounts` - ⊕ **number** ⋆

  Defines the number of counts to assign to each of the parameters to account for uncounted or underrepresented transitions. NB: It is adviseable that this be non-zero, so that transitions which never happen from gtf-annotated features (for example Inter to CNC) are assigned a non-zero probability. In short, put "1".

Text

None.

Subelements

`fixed_transition`

B.7.2 `fixed_transition` element

This defines a tweaked transition value. This is commonly done in Human to force longer transcripts by manually lowering the transition probability from an intron to a terminal exon, and so on.

Attributes

- `from` - ⊙ **state name** ⋆ The transition source state name. Must correspond to a name of a state in the `states` section.

- `to` - ⊙ **state name** ⋆ The transition sink state name. Must correspond to a name of a state in the `states` section.

[2] hack

- `values` - ⊖ **space-separated number list** ⋆ The actual probabilities to assign to this tweaked transition. The number of probabilities must equal the number of isochores in the transition model.

Text

None.

Subelements

None.

B.8 The pseudo-transitions section

This very unfortunately long and ugly section is part of the fallout of making iPE entirely tabula rasa, or perhaps a hacky way of compensating for a feature that I didn't feel like completely implementing. You decide.

Early on, I discovered that most of the transition probability distributions added up to something more than one. The reason for this is that several of the transitions are assigned the same probability because they are really not distinct events. This applies exclusively to the inframe stop codon tracking states, such as `Einit1T` and friends. Since there's no good way of estimating parameters for this, nor any point, the transition for, zum beispiel, `Einit1T` to `Intron1T` gets the same probability as `Einit1` to `Intron1`. Make sense?

Point being, you probably will never touch this,[3] especially since the coding exon state topology is pretty well established and unlikely to change.[4]

B.8.1 `pseudo_transitions` element

This is simply a container for all the `pseudo_transition` elements.

Attributes

None.

Text

None.

Subelements

`pseudo_transition`

[3] As a side note, really this should be implemented as a "framed state" class, which handles all such things, such as defining all separate phase preference states, as well as inframe stop-codon tracking features. This is far too much of a pain for me to actually do. Meddlers anywhere?

[4] At one point we thought we might, adding alternative splicing states for optional coding regions, however this idea was very bad and did not work at all.

B.8.2 `pseudo_transition` element

The sum of the meaning of each of these elements is:

$$Pr(\texttt{dest}|\texttt{source}) = Pr(\texttt{pseudo\_dest}|\texttt{pseudo\_source})$$

Attributes

- `source` - ⊙ **state name** ⋆
  The source of the original transition.
- `dest` - ⊙ **state name** ⋆
  The destination of the original transition.
- `pseudo_source` - ⊙ **state name** ⋆
  The source of the pseudo-transition.
- `pseudo_dest` - ⊙ **state name** ⋆
  The destination of the pseudo-transition.

Text

None.

Subelements

None.

B.9 The duration model section

The duration model defines the probability distribution function that is used to calculate the probability of a length of stay in any given state. In brief, it is a model of the lengths of introns, exons, etc. Certain compromises are taken here for the sake of speed. For example, most gHMMs use an exponential distribution for introns because it would take much longer to compute the best path in a gHMM with a free, histogram-style distribution.

Each state should have some distribution function defined, and for the ones that cannot be estimated through training examples (for example, intergenic[5]), there should be a distribution with fixed parameters that represent an estimate of their true values. Each state has a `dur_model` attribute which points to one of the durations here, with the name given in the `region` attribute of the `duration_model` element.

Another description is provided in section 6.3.

[5] One might contend that you can estimate intergenic length distributions from the annotations, however, I really think this is a bad idea. If all intergenic regions were truly intergenic regions as annotated, then there would be no need to actually be predicting genes on the genome, would there? And since all annotation falls down the "annotation bias" slippery slope, it's really pointless to try and get anything out of the annotations.

B.9.1 `duration_model` element

The top-level element of a duration distribution. It houses one or more duration submodels which define the models for each isochore.

Attributes

- `region` - ⊕ **duration name** ⋆

  Defines the name of the duration model. This should correspond to one or more states' `dur_model` attributes in the `states` section.

- `isochores` - ⊕ **space-separated number list** ⋆

  Defines the isochores used in estimating this model.

Text

None.

Subelements

`duration_submodel`

B.9.2 `duration_submodel` element

This defines the model used for a specific isochore level for a duration model. This contains a number of submodels which define a distribution on a range of lengths. All possibile lengths should be covered.

Attributes

- `isochore` - ⊕ **number** default "100"

  The isochore of this submodel.

- `distributions` - ⊙ **number** default ""

  This is not required, and not really used. It has the number of distributions underneath it in many provided gHMM files.[6]

Text

None.

Subelements

`duration_distribution`
`fixed_duration_distribution`

[6] I would have gotten rid of this altogether had I the time to eliminate them from all gHMMs and debug the results. Oh, well.

B.9.3 `duration_distribution` element

Defines an actual probability distribution function on a range of lengths for a duration model.

Attributes

- `model` - ⊕ **duration model name** ⋆

  This is one of the supplied duration distribution types from iPE. These are:

  - `DEFINED`

    This is a histogram-style distribution of which models features that have an arbitrary length distribution (for example, coding exons, which do not have a very nice piece-wise distribution function). States that use this distribution take the longest to compute in gene prediction, because all lengths must be considered to find the optimum length of stay in the state. For an example of this, see figure 5.5.

    This contains a number of parameters equal to the range of the distribution. Each parameter defines the probability of getting a length of stay equal to that length.

  - `GEOMETRIC`

    This defines an exponential distribution of the form

    $$Pr(X) = \lambda e^{-\lambda x}$$

    λ corresponds to the inverse of the mean length of the distribution. These are quicker to compute.

    One or two parameters are present in these distributions, the first being the mean length of the features for this distribution, and the second (optional) is a scaling factor for distributions which are piecewise geometric or have a geometric tail.

  - `CONSTANT`

    This is a simple constant probability assigned to features in a range of lengths, usually a low probability. This is to make certain lengths unlikely. This is also quick to compute in gene prediction.

- `min` - ⊕ **number** default "0"

  Defines the minimum length of a feature considered by this specific model.

- `max` - ⊕ **number** default "-1"

  Defines the maximum length of a feature considered by this specific model. A value of "-1" (default when attribute is not provided) indicates all lengths greater than or equal to the minimum.

- `length_unit` - ⊙ **number** default "1"

  A `length_unit` of 1 indicates that the all length values are considered, a `length_unit` of 3 indicates that every feature is rounded down to the nearest length that is a multiple of three. The latter is typically used for coding exons, since the assumption (as detailed in [4]) is that evolution of exons occurs at the codon-level. In `zoe` format, the each parameter is repeated 3 times.

- `pseudocounts` - ⊙ **number** default "0"

  The number of counts to assign each considered length automatically. After the examples are counted, each length in the range (the maximum length seen if the `max` is "-1") of the model is given this many more counts before normalizing.

- `smothing` - ⊙ **number** default "0"

  The type of smoother to be used. These are dynamically discovered, so any implemented smoother can be used. Current types are `kernel` and `gaussian`. This is described in section 7.3.

- `smothing_data` - ⊙ **number** default "0"

  The data to pass to the selected smoother. This is described in section 7.3.

- `normalizing` - ⊙ **method** default "normalize"

  The mothod to use for normalization of the distribution. This is described in section 7.4.

Text

None.

Subelements

None.

B.9.4 `fixed_duration_distribution` element

This is a duration distribution whose parameters are not estimated from the examples. The parameters are provided by the user in the text of the element. The parameters are assumed to have meaning in the `zoe` format.

Attributes

- `min` - ⊕ **number** default "0"

  Defines the minimum length of a feature considered by this specific model.

- `max` - ⊕ **number** default "-1"

  Defines the maximum length of a feature considered by this specific model. A value of "-1" (default when attribute is not provided) indicates all lengths greater than or equal to the minimum.

Text

The parameters of the model, assumed to have some meaning to the program (`iscan`) that interprets the parameter file.

Subelements

None.

B.10 The null region definitions section

This defines what regions of input annotations to consider for the "null" model which is subtracted from the positive model where indicated. More discussion of this is in section 6.2.

B.10.1 `null_region_definitions` element

Contains the `null_region_definition` elements.

Attributes

None.

Text

None.

Subelements

`null_region_definition`

This is an element which tells iPE where to add null regions to the annotation. This is commonly introns or intergenic regions, but you can specify exact subregions of each state. (This is provided because the DNA model considers only the first kilobase of the first coding intron as the null region.) Note that unlike most things in iPE, strand is not taken into account, and so the coordinates are absolute. In this example:

```
<null_region_definition seqtype="dna" states="Intron0 Intron1 Intron2"
    start="0" end="999" first_feature="0" last_feature="0"/>
<null_region_definition seqtype="dna" states="Intron0- Intron1- Intron2-"
    start="L-1000" end="L-1" first_feature="N-1" last_feature="N-1"/>
```

the minus strand states are separated because of the absolute coordinates.

Attributes

- `states` - ⊙ **space-separated state names** ⋆

  The states that pass through this null region.

- `start` - ⊙ **relative coordinate** ⋆

  The start of the null region within the state annotation.

- `end` - ⊙ **relative coordinate** ⋆

  The end of the null region within the state annotation.

- `first_feature` - ⊙ **relative coordinate** ⋆

  The first feature in the set of all the eligible features (those whose states are listed in the `states` attribute) per transcipt that is considered as being in the null region.

- `last_feature` - ⊙ **relative coordinate** ⋆

  The last feature in the set of all the eligible features (those whose states are listed in the `states` attribute) per transcipt that is considered as being in the null region.

- `seqtype` - ⊙ **sequence name** ⋆

  The type of sequence that this null region applies to.

- `sampling_rate` - ⊙ **number** default "1"

  The rate at which to sample all the features found eligible for this null region. This is done in case there is an abundance of sequence for this null region definition, to avoid slowing down parameter estimation by counting too many examples.

Text

None.

Subelements

None.

B.11 The sequence models section

This is the core of the parameter estimation. This is where the content is modeled for each state (usually many states to one model). See section 6.4 for more discussion.

B.11.1 `sequence_models` element

This is the umbrella element that contains all `string_models`.

Attributes

None.

Text

None.

Subelements

`string_model` `default_string_model` `fixed_string_model`

B.11.2 `string_model` element

This describes a model that has a specific beginning and end within one or more feature(s). This allows for both position-specific models and position-independent models. See the subsection about the Five Meta-models in section 6.4 for more information.

Attributes

- `name` - ⊕ **text** ⋆

  The name of the model. This should be globally unique among all string models.

- `zoe_name` - ⊖ **text** default “”

  The name of the model to be printed in the `.zhmm` output, if different from the main name.

- `source` - ⊕ **sequence type** ⋆

  The source is the type of sequence being modeled. Sequence classes are dynamically discovered. Implemented sequences include (but are not limited to) `dna`, `cons` (Conseq), `malign` (N-SCAN multiple alignment) and `est` (ESTSeq).

- `model` - ⊕ **emission model name** ⋆

  The type of model. See section 6.4 for more information.

- `zoe_model` ⊕ **zoe emission model name** “”

  The type of model that is visible in the `.zhmm` file. This is necessary, for example, with `WAM` models since `zoe` only interprets the keyword `WWAM`, thus all `WAM` models must have `WWAM` as the `zoe_model`.

- `states` ⊙ **space-separated state name list** ⋆

  The list of states where the parameters for this model will be estimated in. For every state in the examples that is also listed here, iPE will generate a region within that state sequence that corresponds to the `begin` and `end`

attributes of this model.
Note that this does not necessarily coincide with the model given in the top `states` section. For example, you may assume that the conservation patterns in UTRs are the same as they are in conserved non-coding regions, and thus use UTR exons to estimate parameters for that model. (This is in fact done in the N-SCAN model.)

- `begin` ⊙ **0-based boundary coordinate** ⋆

  This defines the beginning of subregion of a state to use for this particular model. Use "`L`" to define coordinates relative to the end of the feature. For example, the `Start SDT` model begins at `-6` and ends at `5`. This means it starts 6 bases upstream of the beginning of the state sequences listed and ends 5 bases downstream, for a total of 12 bases. The `Donor SDT` begins at `L-3`, **two** bases upstream of the end of the feature (the feature ends at the feature's length minus one since it is 0-based), and ends at `L+5`, **six** bases downstream of the end of the feature, for a total length of nine.

- `end` ⊙ **0-based boundary coordinate** ⋆

  This defines the end of the model's subregion. See the discussion on the `begin` attribute for details.

- `wildcard` ⊙ **literal, lexical or penalty** default "literal"

  This defines how wildcard characters (in DNA, they are "`N`", in all other implemented sequences these are not used outside of iPE). If it is LITERAL, the wildcard is to be scored as another character in the alphabet, e.g., "`N`" gets its own counts. If it is LEXICAL, it is to be treated as all possible letters in the alphabet (besides itself), e.g., "`N`" indicates that the letters "`A`", "`C`", "`G`" and "`T`" should get `.25` counts. If it is PENALTY, then the wildcard is to be penalized wherever it is seen. For example, this might be used with "`N`"-containing 5-character DNA models, for which the presence of an "`N`" indicates that the sequence is repeat-masked, and is likely not actual coding sequence. The value for this penalty is given with the `wildcardPenalty` option in the instance file.
  In general, this setting is code-level and not interesting to a pedestrian user. If you are confused about this, the best thing to do is leave it alone.

- `sampling_rate` ⊙ **number** default "1"

  This defines how often the model is counted out of all the examples seen on a scale of 0 to 1. This is generally used for overrepresented features, such as introns, which would considerably slow down parameter estimation if all examples were counted.

- `pseudocounts` ⊙ **number** default "0"

  How many prior counts to give each parameter in the model. This is used for underrepresented parameters in the model which otherwise would receive a score of negative infinity.

In practice, it is mostly the case that DNA models do not get pseudocounts, and all other models get a pseudocount of 1.

- `smothing` ⊙ **smoothing type** default "none"

  What smothing method to use after counts have been totalled. This has never yet been used for sequence models.

- `smoothing_data` ⊙ **string** default ""

  String to pass to the smoother specified in `smoothing`.

- `null_model` ⊙ **0 or 1** ⋆

  If 1, include an analagous null model to the positive model for the output parameters. That is, for each `null_region` defined above discovered, tell the model to count the null model and incorporate it into the final scores as a log-odds ratio. For more information, see section 5.5.

- `length` ⊖ **number** ⋆

  This number is completely ignored by parameter estimation and passed along to the `.zhmm` file. It indicates a number of different things to `zoe` and the documentation of this is out of the scope of this manual.

- `focus` ⊖ **number** ⋆

  This number is completely ignored by parameter estimation and passed along to the `.zhmm` file. The documentation of this is out of the scope of this manual.

- `symbols` ⊕ **number** ⋆

  This number indicates the number of symbols in the alphabet. In general, this should be equal to the number of characters in the alphabet of the `source` indicated above. The exception is the DNA sequence type, which can have 4 or 5 characters (including the "N" or not). If "5" is chosen, parameters for all possible sequences containing A,C,G,T or N are output.

- `data` ⊙ **setting** ⋆

  This uses the `setting=value` format, which is specific to each model. The settings at time of writing are enumerated in section 6.4.

Text

None.

Subelements

`string_submodel`
`fixed_string_submodel`

B.11.3 `default_string_model` element

The `default_string_model` defines a model with no beginning and end. Its boundaries are defined by the remaining sequence after all `sequence_models` have been defined. For example, if you had both a donor and acceptor model for an exon, that exon's `default_string_model` will cover all the space in between. See the subsection about the Five Meta-models in section 6.4 for more information.

Here we list attributes inherited from `string_model` which have identical function, and their meaning can be referred to in the previous section.

Inherited Attributes

- `name` from `string_model`
- `zoe_name` from `string_model`
- `source` from `string_model`
- `model` from `string_model`
- `zoe_model` from `string_model`
- `states` from `string_model`
- `wildcard` from `string_model`
- `sampling_rate` from `string_model`
- `smoothing` from `string_model`
- `smoothing_data` from `string_model`
- `null_model` from `string_model`
- `length` from `string_model`
- `focus` from `string_model`
- `symbols` from `string_model`
- `data` from `string_model`

Attributes

- `null_params` $\oplus$ **0 or 1** default "0"

  Instead of outputting actual estimated parameters, the model is assumed to be part of the null model, and all its log-odds scores become zero. For a discussion of why this is, see section 5.5.

Text

None.

Subelements

`string_submodel`
`fixed_string_submodel`

B.11.4 `string_submodel` element

This model is any model which is subordinate to some top-level model. In the case of `string_model`s, they represent a subunit of the region for which the `string_model` is defined. For example, one might model the branch of the Acceptor with a 2nd order `WWAM` and the signal and pyrimidine rich region with a 1st order `WAM`.[7] One would enter coordinates for begin and end relative to the region defined in the top level model, i.e., the first base is "0" of the submodel.

Inherited Attributes

- `name` from `string_model`
- `zoe_name` from `string_model`
- `model` from `string_model`
- `zoe_model` from `string_model`
- `wildcard` from `string_model`
- `pseudocounts` from `string_model`
- `smoothing` from `string_model`
- `smoothing_data` from `string_model`
- `length` from `string_model`
- `focus` from `string_model`
- `symbols` from `string_model`
- `data` from `string_model`

[7] This is how it is done in Genscan.

Attributes

- `ordinal` ⊙ **number** default “”

  Submodels are often sensitive to order. By placing a number here, you define which sumbodels this model is before and/or after, indicated by its numerical value.

- `begin` ⊙ **0-based boundary-relative coordinate** default “”

  This is similar in concept to the `begin` of `string_model`s, however it defines a beginning relative to the parent model's region. So the boundary “0” is no longer the beginning of the feature, but the beginning of the parent model, and “L”, not feature end, but parent end.

- `end` ⊙ **0-based boundary-relative coordinate** default “”

  See `begin` above.

Text

None.

Subelements

`string_submodel`
`fixed_string_submodel`

An Example

Here is a rather complex example of submodels:

```
<string_model name="Acceptor" source="dna" model="SDT"
  states="Exon0 Exon0- Exon1 Exon1- Exon2 Exon2-
          Eterm Eterm0- Eterm1- Eterm2-" null_model="1"
  begin="-40" end="2" length="2" focus="1" symbols="4">
    <string_submodel
      name="NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNAGNNN"
      zoe_name="AG" model="SPLIT" zoe_model="WWAM"
      begin="0" end="42" length="43" focus="42"
      symbols="4" ordinal="0" data="zoeHeaderEnd=3">
      <string_submodel name="branch" model="MARG_WAM"
        zoe_model="WWAM" begin="0" end="19"
        length="20" focus="42" symbols="4"
        data="order=2 printedOrder=3"/>
      <string_submodel name="pyrimidine-rich/acceptor"
        model="MARG_WAM" zoe_model="WWAM"
        begin="20" end="42" data="order=1 printedOrder=3"
        length="23" focus="42" symbols="4"/>
    </string_submodel>
```

```
  <fixed_string_submodel name="NN" model="WMM" null_model="1"
  ordinal="1" length="1" focus="0" symbols="4"  >
  . . . .
  </fixed_string_submodel>
</string_model>
```

The top-level model is an `SDT`, sequene decision tree. Dependent on what the sequence of the current example is, one of the submodels is chosen. The first string submodel is the node of the tree where the correct `AG` signal is in place for the decision tree, with 40 bases upstream of the `AG`. This submodel is a `SPLIT` model, which does piecewise modeling of a region.[8] The first submodel, the `branch` model uses a 2nd order `WAM` for the weak signal (in order not to overfit).[9] The next part is modeled as a 1st order `WAM`.

Note the ordinal values. Since XML provides no guarantee that the parser will order the subelements in any order, we use ordinals to guarantee the order of the models. This is especially important with the `SDT` model: it checks the first pattern first for a match, and if none is found, goes to the second pattern. If the final pattern is the wildcard-only pattern (match anything), and it were interpreted first, then all regions would undesirably go to the first, wildcard-only pattern.

B.11.5 `fixed_string_model` element

It is often the case that we cannot estimate parameters very easily from annotation data, and so we have to retrieve parameters from public data[10] or estimate them offline of parameter estimation. In other cases we may not have data with which to estimate parameters. For example, we might not have UTR annotations so we need UTR parameters from another species.

In both cases, we use the `fixed_string_model` element. This allows us to "hard-code" parameters into the gHMM file. These parameters will be output directly to the `.zhmm` file as well as all intermediate XML files.

Inherited Attributes

- `name` from `string_model`
- `source` from `string_model`
- `model` from `string_model`
- `null_model` from `string_model`
- `begin` from `string_model`

[8] Another feature of this model is that it doesn't show up as 2 models to `zoe` unless you tell it to. See section 6.4 for more information.

[9] The `MARG_WAM` model is there to make the model appear as a 3rd order model to `zoe`.

[10] For Twinscan, a lot of the parameters were inherited from Genscan, including the PolyA and Promoter parameters.

- `end` from `string_model`
- `length` from `string_model`
- `focus` from `string_model`
- `symbols` from `string_model`
- `data` from `string_model`

Attributes

There are no new attributes unique to this element. All are inherited from `string_model`.

Text

This is some string meaningful to the program (i.e. `iscan`, `zoe`) which is interpreting it. In practice, a set of parameters. For example,

```
<fixed_string_model name="PB_CAP" source="dna" model="WMM"
        null_model="1" states="Prom Prom-" begin="L-7" end="L"
        focus="3" length="8" symbols="4" >
    -6  -7  -1  9
    -100 20  -100 -100
    19  -100 -100 -23
    -15 0 6 0
    0 3 -100 8
    -2  6 -7  0
    -8  2 0  4
    -6  3 -4  4
</fixed_string_model>
```

Describes a `WMM` for the poly-A binding cap of the promoter. Since we don't have promoters annotated in our input files, we need to use the same parameters all the time. Its length is 8, and thus 8 rows of parameters for each of the 4 DNA characters are shown.

Subelements

None.

B.11.6 `fixed_string_submodel` element

This element is the same as a `fixed_string_model` except that it is a submodel element instead of a top-level model element.

Inherited Attributes

- `name` from `string_model`
- `model` from `string_model`
- `ordinal` from `string_submodel`
- `null_model` from `string_model`
- `length` from `string_model`
- `focus` from `string_model`
- `symbols` from `string_model`
- `data` from `string_model`

Attributes

No new attributes, all are inherited.

Text

This is some string meaningful to the program (i.e. `iscan`, `zoe`) which is interpreting it. See the `fixed_string_model` description above.

Subelements

None.

Appendix C

Feature Map Reference Guide

If you're reading this first, you might want to look at the terms section in Appendix B.

In this chapter, we use the word "feature" to refer to a feature in the annotation, rather than a feature in the converted state sequence.

C.1 The author section

The author section contains the author element.

C.1.1 `author` element

Attributes

None.

Text

Should include the name of the author of the feature map.

Submelements

None.

C.2 The date section

The date section contains the date element.

C.2.1 `date` element

Attributes

None.

Text

Should include the date of creation of the file.

Subelements

None.

C.3 The title section

Should include the title of the feature map file.

C.3.1 `title` element

Attributes

None.

Text

Should be a title, giving a general description of what the feature map is for, for example, if it is specifically for a UTR-predicting gene predictor or non-UTR predicting gene predictor.

C.4 The file type description section

Contains a text description of the file format.

C.4.1 `file_type_description` element

Attributes

None.

Text

Should contain the English full description of the file type.

Subelements

None.

C.5 The filename extension section

Contains the filename extension of the format to be converted.

C.5.1 `filename_extension` element

Attributes

None

Text

Should contain the exact filename extension (without the leading dot) of the filetype that this feature map converts.

Subelements

None.

C.6 The feature mappings section

This contains the meat of the feature map. A feature map is comprised of a set of feature mappings, each of which map a feature in the annotation format, along with certain context qualities, to a single state in the gHMM. Note that the state must be present in the corresponding gHMM in order for the feature map to be accepted.

C.6.1 `feature_mappings` element

Attributes

None.

Text

None.

Subelements

- `feature_mapping`

C.6.2 `feature_mapping` element

This element defines criteria for a feature in the annotation to be mapped to a specific state.

Attributes

- `state_name` - The name of the state in the corresponding gHMM that these criteria map to.
- `feature` - The name of the feature, meaningful to the annotation format, which this feature mapping corresponds to.
- `number_in_transcript` - The number of features with name given in the `feature` attribute.
- `first_feature` - The 0-based ordinal of the first such feature of name given in the `feature` element that may be mapped to the state given in the `state_name` attribute.
- `last_feature` - The 0-based ordinal of the last such feature of name given in the `feature` element that may be mapped to the state given in the `state_name` attribute.
- `state_region_start` - The 0-based relative coordinate of the beginning of the region that will be converted to the state given in the `state_name` attribute.
- `state_region_end` - The 0-based relative coordinate of the ending of the region that will be converted to the state given in the `state_name` attribute.

Text

None.

Subelements

None.

Appendix D

Sequence Types Included in iPE

iPE provides a very flexible and easily extensible framework for working with and creating many different sequence types. Each sequence type is a class unto itself, inheriting from the `iPE::Sequence` class. One can get the basic documentation for any sequence type by typing

```
man iPE::Sequence::Dna
```

for instance. Here I provide a brief description of each sequence type included in iPE.

Each sequence described has several fields. The first is the class name. You can use that to type in for the man page. The second is the iPE name, meaning the name used to refer to the sequence in the gHMM file and the instance file. The third is the zoe name, which is the name output to `.zhmm` files. The alphabet is the set of all letters in the sequence type. The ambiguity codes are all the codes which refer to more than one character in the alphabet. The wildcard is the character that refers to all letters in the alphabet.

D.1 Dna

Classname: iPE::Sequence::Dna
iPE Name: dna
zoe Name: DNA
Alphabet: A,C,G,T
Wildcard: N
Ambiguity codes:

```
R =  A,G
Y =  C,T
W =  A,T
```

```
S =  C,G
M =  A,C
K =  G,T
B =  C,G,T
D =  A,G,T
H =  A,C,T
V =  A,C,G
N =  A,C,G,T
```

D.2 Cons

Classname: iPE::Sequence::Cons
iPE Name: cons
zoe Name: CONSEQ
Alphabet: 0,1,2
Wildcard: w
Ambiguity codes:

```
w = 0,1,2
```

The conservation sequence alphabet was the invention of Twinscan, to coarsely indicate the level of onservation of each base by asserting what resulted from a genome-to-genome alignment of an informant species. The paper and the actual implementation are different:

```
Paper iscan   Meaning
:     0       mismatch/gap
|     1       match
.     2       unaligned
```

These are generated with `twinscan_driver.pl` or `conseq.pl`, included in the zoe package.

D.3 Est

Classname: iPE::Sequence::Est
iPE Name: est
zoe Name: ESTSEQ
Alphabet: 0,1,2
Wildcard: w
Ambiguity codes:

```
w = 0,1,2
```

This is another sequence type which indicates the alignment of ESTs to model the likelihood of transcription. Their meanings are as follows:

```
0   unknown (not an exon or intron or conflict between exon and intron)
1   exon in EST alignment
2   intron in an EST alignment
```

These are generated by running `blat` on all ESTs to the genome and then running `estseq_psl.pl` on them. (This should be included in the zoe package.

D.4 Malign

Classname: iPE::Sequence::Malign
iPE Name: malign
zoe Name: (none)
Alphabet: A,C,G,T,_,.
Wildcard: w
Ambiguity codes:

```
R =  A,G
Y =  C,T
W =  A,T
S =  C,G
M =  A,C
K =  G,T
B =  C,G,T
D =  A,G,T
H =  A,C,T
V =  A,C,G
N =  A,C,G,T
w =  A,C,G,T,\_,.
```

The Malign class is not strictly a sequence, per se, since it actually contains multiple sequences. It can behave as a sequence if we consider the fact that all alignments are projected onto the target sequence so that the total length is always equal to the target sequence. The two non-letter characters, _ and ., refer to a gap and an unaligned region, respectively.

D.5 Array

Classname: iPE::Sequence::Array
iPE Name: array
zoe Name: ARRAYSEQ
Alphabet: 0,1,2
Wildcard: w
Ambiguity codes:

```
w = 0,1,2
```

This is for the current experimental approach to gene prediction using Affymetrix tiling array data. The letters mean the following (I think):

```
0   region not queried by tiling array
1   region queried but not transcribed (not in a transfrag)
2   region queried and transcribed (in a transfrag)
```

D.6 Other Sequence Types

Several other sequences (i.e. Asest, Cola, Repeat and Tile) are included in the iPE package. They are not described here because they were the children of failed experiments that will not be explored any further.

Appendix E

How Exponential Tails Are Fit

The exponential PDF comes in the form

$$\Pr(x) = \frac{1}{\lambda} e^{-\frac{x}{\lambda}} \tag{E.1}$$

where λ is the expected value of the PDF and s is used in our case to scale the area under the curve.

To fit a geometric tail from A to B with density N_i (i representing the current piece of the distribution) and start probability $\Pr_{i-1}(A)$, we can view this as a system of equations.

Our constraints are that

$$\Pr_{i-1}(A) = \Pr_i(A) \tag{E.2}$$

$$\int_A^B \Pr_i(x)dx = N_i \tag{E.3}$$

Each, respectively, imply

$$\Pr_{i-1}(A) = \frac{s_i}{\lambda_i} e^{-\frac{A}{\lambda_i}} \tag{E.4}$$

$$\int_A^B \frac{s_i}{\lambda_i} e^{\frac{-x}{\lambda_i}} dx = N_i \tag{E.5}$$

The variable s_i is a constant factor different from N_i which scales the exponential such that the area under the curve between A and B is N_i, and not the area of the whole curve from 0 to infinity.

We know $\Pr_{i-1}(A)$ and N_i, and we hope to find solutions for λ_i and s_i in terms of these, so we can each piece one at a time without any fancy global inference.

Integrating (E.5), we get

$$s_i(e^{-\frac{A}{\lambda_i}} - e^{-\frac{B}{\lambda_i}}) = N_i \tag{E.6}$$

From (E.4), we can get s_i alone.

$$s_i = \lambda_i \mathrm{Pr}_{i-1}(A) e^{\frac{A}{\lambda_i}} \tag{E.7}$$

Now we can substitute this back into (E.6).

$$\lambda_i \mathrm{Pr}_{i-1}(A)(1 - e^{\frac{A-B}{\lambda_i}}) = N_i \tag{E.8}$$

This must be heuristically solved, as it is a transcendental equation. We put each instance of λ_i on either side of the equation:

$$\mathrm{Pr}_{i-1}(A)(1 - e^{\frac{A-B}{\lambda_i}}) = \frac{N_i}{\lambda_i} \tag{E.9}$$

Then set λ_i for a number of reasonable (say, 1 to 1,000,000) values on each side of the equation, and pick the one pair that is closest. We can easily find s_i once λ_i is found with equation (E.7).

In the simple case, where there is only one tail, the parameters can be found in closed-form:

Your constraints are as follows:

$$N_n = \int_A^\infty f_n(x)dx \tag{E.10}$$

$$f_{n-1}(A) = f_n(A) \tag{E.11}$$

Integrating (E.10) gives us

$$N_n = s_n e^{-\frac{A}{\lambda_n}} \tag{E.12}$$

Expanding (E.11) gives us

$$f_{n-1}(A) = \frac{s_n}{\lambda_n} e^{-\frac{A}{\lambda_n}} \tag{E.13}$$

Dividing equation (E.13) by equation(E.12) gives us

$$\frac{N_n}{f_{n-1}(A)} = \lambda_n \tag{E.14}$$

which we can use to solve for s_n in the equation (E.13).

Acknowlegements

Thanks to Randy Brown, Sam Gross, Chauchun Wei, Manimozhian Arumugam, and Paul Flicek for aid in recreating the many different subtleties of our sordid parameter estimation past. Thanks to Evan Keibler for many explainations and discussions of both `iscan` and `GTF`. Thanks to Brian Koebbe for help putting together the RPMs and in general for keeping our computers running. Thanks, of course, to my advisor, Michael Brent.

Bibliography

[1] Matthieu Blanchette, W. James Kent, Cathy Riemer, Laura Elnitski, Arian Smit, Roskin Krishna, Robert Baertch, Kate Rosenbloom, Hiram Clawson, Eric Gree, David Haussler, and Webb Miller. Aligning multipe genomic sequences with the threaded blockset aligner. *Genome Research*, 14(7):708–715, 2004.

[2] M. Borodovsky and J. McInich. Genemark: parallel gene recognition for both dna strands. *Computers & Chemistry*, 17(19):123–133, 1993.

[3] Adrian Bowman and Adelchi Azzalini. *Applied Smoothing Techniques for Data Analysis*. Oxford Science Publications, Oxford, UK, 1997.

[4] Christopher Burge. *Identification of genes in human genomic DNA*. PhD thesis, Stanford University, May 1997.

[5] Christopher Burge and Samuel Karlin. Prediction of complete gene structures in human genomic dna. *Journal of Molecular Biology*, 268(1), 1997.

[6] Richard Durbin, Sean Eddy, and Graeme Krogh, Anders Mitchinson. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, UK, 1998.

[7] Warren Gish. Wu-blast. http://blast.wustl.edu.

[8] Samuel Gross and Michael Brent. Using multiple alignments to improve gene prediction. *Journal of Computational Biology*, 13(2), 2006.

[9] Ian Korf, Paul Flicek, Daniel Duan, and Michael Brent. Integrating genomic homology into gene structure prediction. *Bioinformatics*, 17(Suppl 1), 2001.

[10] Lawrence R. Rabiner. An tutorial on hidden markov models and selected applications in speech recognition. volume 77, pages 4–16, 1989.

[11] Lawrence R. Rabiner and B. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, pages 4–16, January 1986.

[12] Scott Schwarts, W. James Kent, Arian Smit, Zheng Zhang, Robert Baertsch, Ross Hardison, David Haussler, and Webb Miller. *Genome Research*, 13(1):103–107, 2003.

[13] Chaochun Wei. *Using Expressed Sequence Tags to Improve Gene Structure Prediction*. PhD thesis, Washington University in St. Louis, May 2006.