Masters Project Report

Eval: A Gene Set Comparison System

Evan Keibler
evan@cse.wustl.edu

Table of Contents

Table of Contents ... - 2 -
Chapter 1: Introduction ... - 5 -
1.1 Gene Structure ... - 5 -
1.2 Gene Predictors ... - 7 -
1.3 Eval ... - 8 -
Chapter 2: User Level Documentation ... - 10 -
2.1 GTF Validator ... - 10 -
2.2 Eval Overview ... - 12 -
2.2.1 Statistics ... - 12 -
Gene Level ... - 14 -
Transcript Level ... - 14 -
Exon Level ... - 16 -
Nuc Level ... - 17 -
Signal Level: ... - 17 -
2.2.2 Evaluate ... - 18 -
2.2.3 General Statistics ... - 18 -
2.2.4 Filter ... - 18 -
2.2.5 Graph ... - 19 -
2.2.6 Overlap ... - 20 -
2.2.7 Distribution ... - 21 -
2.3 Eval GUI ... - 22 -
2.3.1 Overview ... - 22 -
2.3.2 Loading the GUI ... - 22 -
2.3.3 Menus ... - 23 -
2.3.4 Eval Screen ... - 24 -
2.3.5 GenStats Screen ... - 24 -
2.3.6 Filter Screen ... - 24 -
2.3.7 Graph Screen ... - 25 -
2.3.8 Overlap Screen ... - 26 -
2.3.9 Dist Screen ... - 26 -
2.4 Eval Command Line Interfaces ... - 27 -
2.4.1 evaluate_gtf.pl ... - 27 -
2.4.2 get_general_stats.pl ... - 27 -
2.4.3 filter_gtfs.pl ... - 28 -
2.4.4 graph_gtfs.pl ... - 28 -
2.4.5 get_overlap_stats.pl ... - 29 -
2.4.6 get_distribution.pl ... - 30 -
Chapter 3: Code Level Documentation ... - 32 -
3.1 Overview ... - 32 -
3.1.1 Data Types ... - 32 -
3.1.2 Naming Schemes ... - 33 -
3.2 GTF.pm ... - 33 -

3.2.1 Overview ... - 33 -
3.2.2 GTF Object ... - 34 -
Global Variables ... - 34 -
Constructor ... - 35 -
Accessor Functions ... - 36 -
Modifier Functions ... - 37 -
Internal Functions ... - 37 -
3.2.3 Gene Object ... - 38 -
Global Variables ... - 38 -
Constructor ... - 38 -
Accessor Functions ... - 38 -
Modifier Functions ... - 40 -
3.2.4 Transcript Object ... - 40 -
Global Variables ... - 40 -
Constructor ... - 42 -
Accessor Functions ... - 42 -
Modifier Functions ... - 44 -
Internal Functions ... - 45 -
3.2.5 Feature Object ... - 46 -
Global Variables ... - 46 -
Constructor ... - 47 -
Accessor Functions ... - 48 -
Modifier Functions ... - 50 -
3.3 Eval.pm ... - 51 -
3.3.1 Definition of Statistics ... - 51 -
Top-level Statistics Functions ... - 52 -
Gene Level Statistics Functions ... - 52 -
Transcript Level Statistics Functions ... - 53 -
Exon Level Statistics Functions ... - 54 -
Nuc Level Statistics Functions ... - 54 -
Signal Level Statistics Functions ... - 55 -
3.3.2 Evaluate Functions ... - 56 -
List Comparison Functions ... - 56 -
Object Comparison Functions ... - 58 -
Initialization and Clean up Functions ... - 59 -
Data Collection Functions ... - 60 -
Statistic Calculation Functions ... - 61 -
3.3.3 General Statistics Functions ... - 61 -
3.3.4 Filter Funcitons ... - 62 -
3.3.5 Graph Functions ... - 64 -
3.3.6 Overlap Functions ... - 67 -
Specific Overlap Type Functions ... - 67 -
Cluster Building Functions ... - 68 -
Overlap Test Functions ... - 70 -
3.3.7 Distribution functions ... - 70 -
3.3.8 General Functions and Variables ... - 72 -

Global Variables ... - 72 -
Functions ... - 72 -
3.4 eval.pl ... - 72 -
3.4.1 Overview ... - 72 -
Data Types ... - 73 -
3.4.2 Constants ... - 73 -
3.4.3 Global Variables ... - 74 -
3.4.4 Functions ... - 75 -
Initialization Functions ... - 75 -
General Functions ... - 75 -
Menu Functions ... - 76 -
Options Functions ... - 77 -
Eval Frame Functions ... - 77 -
GenStats Frame Functions ... - 79 -
Filter Frame Functions ... - 79 -
Graph Frame Functions ... - 80 -
Overlap Statistics Frame Functions ... - 81 -
Dist Frame Functions ... - 82 -
3.5 Eval Scripts ... - 83 -
3.5.1 evaluate_gtf.pl ... - 83 -
3.5.2 get_general_stats.pl ... - 83 -
3.5.3 filter_gtfs.pl ... - 83 -
3.5.4 graph_gtfs.pl ... - 84 -
3.5.5 get_overlap_stats.pl ... - 84 -
3.5.6 get_distribution.pl ... - 84 -
Chapter 4: Future Work ... - 85 -
Appendix A: GTF File Format Specification ... - 86 -
Appendix B: Fasta File Format ... - 89 -
Appendix C: Conservation File Format ... - 90 -
Appendix D: Example Eval Report ... - 91 -
Index ... - 101 -
References ... - 104 -

Chapter 1: Introduction

1.1 Gene Structure

As large amounts of high quality genomic sequence became available for many organisms the problem of gene-finding changed from the analysis of small segments of the genome, typically less than 150,000 bases, to find a single protein coding gene, to the analysis of large amounts of genomic sequence, up to billions of bases, to identify all protein coding genes [8, 11, 14]. In the past, analysis typically consisted of a single expert manually looking at all available evidence and trying to annotate the gene structure by hand. This is possible when annotating small amounts of sequence for a small number of genes, but is very expensive and tedious. Attempting to annotate billions of bases of sequence by hand is not feasible due to the enormous number of man-hours it would require. Therefore automated gene prediction systems are required to process this large amount of data [5, 6, 8, 11, 14]. Before further discussion of current automated gene prediction systems a brief introduction to gene structure, the output of these systems, is needed.

The central dogma of molecular biology states that DNA is transcribed into RNA which is in turn translated into protein. A gene can be defined as the region of DNA which codes for a particular protein and all adjacent regions which regulate its expression. Gene expression is the processes by which the information encoded in a gene is decoded into a protein. Genes are processed differently in two types of organisms: prokaryotes, which are organisms whose cells have no nucleus, and eukaryotes, which are organisms whose cells do have a nucleus. The translation process is identical for these two types of organisms but the transcription process differs [13].

In prokaryotic organisms the region of the gene which is transcribed into RNA is a continuous stretch of DNA, all of which is then translated into a protein. In eukaryotic organisms the translated region of the gene, the region from which the protein will be built, is normally not continuous. Instead, the transcribed region is comprised of alternating stretches of exons and introns, where only the exon regions will be translated. The transcription process takes place in the cell nucleus and transcribes both exons and introns into a primary RNA transcript in the same order as they appear in the genomic sequence. A process called splicing removes the intron regions and combines the exons regions to create the mature messenger RNA (mRNA). The mRNA is then exported from the nucleus and translated into a protein [13] (Figure 1).

In both prokaryotes and eukaryotes RNA is transcribed into protein in three base pair increments called codons. A protein is a string of amino acids, and each codon signals that a specific amino acid should be added to the end of the protein. Every mRNA should have length evenly divisible by three since it must contain only whole codons [13].

In certain situations the same primary transcript can be spliced in more than one way to yield different proteins. This is called alternative splicing. Primary RNA transcripts are not spliced differently in the same cell at the same time but instead in different cells or at different times [13].

This paper deals primarily with eukaryotic gene predictions. Though the software described can be used for prokaryotic gene predictions also, many of the tools described have significantly less utility when no genes contain introns.

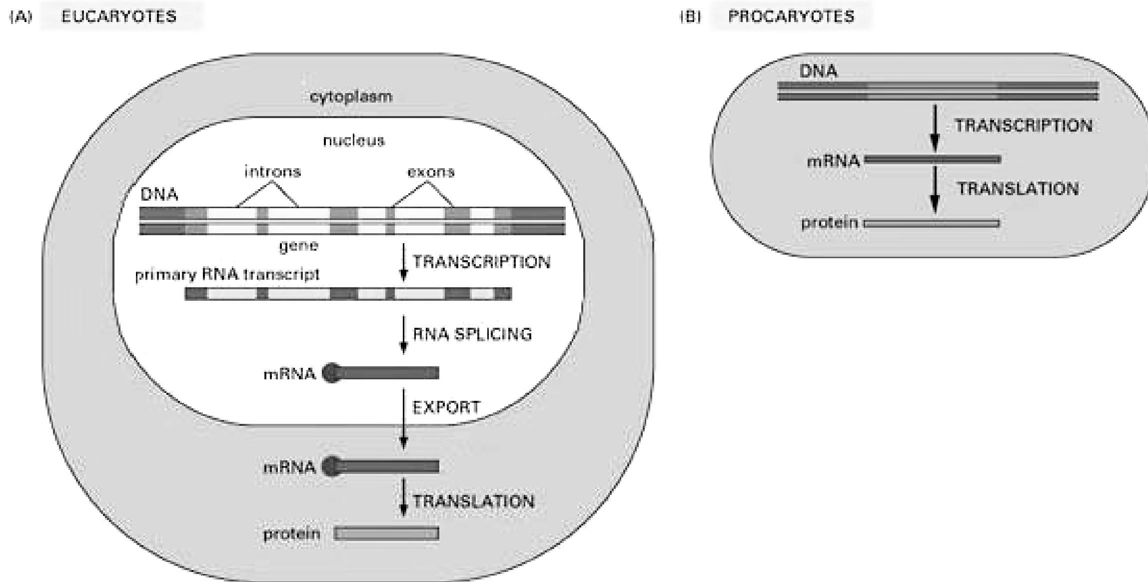

Figure 1. Gene transcription and translation in eukaryotes (A) and prokaryotes (B).

In its natural state, DNA is double stranded helix made up of pairs of four different nucleotides: adenine, guanine, cytosine, and thymine which are referred to by the symbols A, C, G, and T respectively. A always pairs with T and C always pairs with G and these pairings are said to be complementary. Genes may reside on either strand but are always processed from the 5' end to the 3' end. The 5' end of one strand is the 3' end of the other so each strand is the reverse complement of the other [13]. This means that if one were to write the string of nucleotides which make up each strand of a DNA sequence from the 5' end to the 3' end separately, each strand's string would be made of only four characters (A, C, G, and T) and each would be the same as the other except that the nucleotides would occur in the reverse order and all nucleotides would be replaced with their complementary nucleotide. Typically the strands are called the plus strand and the minus strand, and the 5' end of the plus strand is considered position 1.

A multi-exon eukaryotic gene has the following structure. It begins with a promoter region, which regulates gene expression, and is followed by a transcribed but non-

protein-coding region called the 5' untranslated region (UTR). Next comes the initial exon, which contains the start codon, and an alternating series of introns and internal exons, followed by the terminal exon, which contains the stop codon. The terminal exon is followed by another non-coding region called the 3' UTR. The start and stop codons are specific three base pair long sequences which signal the beginning and the end of the transcript to the protein creation machinery. The exon-intron boundaries, also known as splice sites, are signaled by specific two base pair sequences located at the edges of the intron. The 5' end of an intron is called the donor splice site, and the 3' end of an intron is called the acceptor splice site [13].

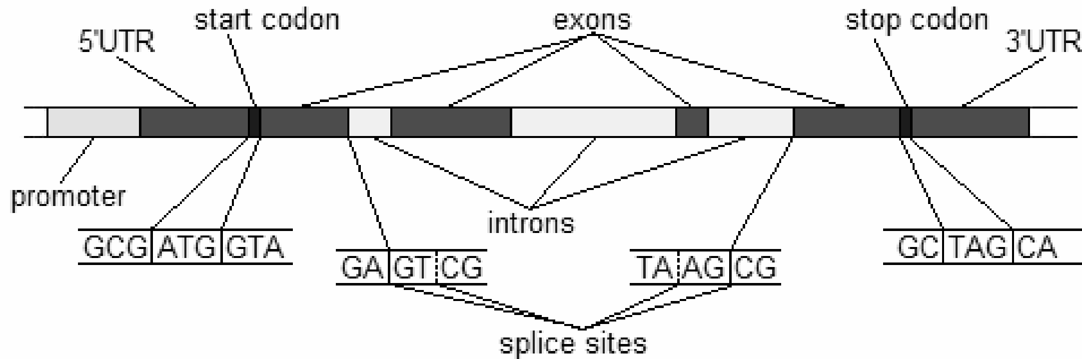

Figure 2. A typical multi-exon eukaryotic gene structure. The start and stop codons as well as the examples of both donor and acceptor splice sites are shown in detail.

It is often very difficult to locate all regulatory regions of a gene and knowing the protein product of a gene is more useful than knowing the location of all regulatory regions so most automated gene predictors focus on finding only the region of DNA which is transcribed to RNA [5, 6].

1.2 Gene Predictors

Most modern automated gene prediction systems fall into one of two categories: transcript alignment based prediction systems and *ab initio*, *de novo* or genome-only prediction systems. The transcript alignment based systems attempt to map known transcripts from the species being annotated or from other species to the genome being annotated. If successful, it is highly likely that the region the known transcript was mapped to is also a gene. The *de novo* gene predictors use only genomic data to predict genes. They are generally based on some type of complex probability model, derived from the expected structure of genes. These computational gene prediction systems generally have many input parameters, in addition to the sequence to be annotated, which determine the types and number of genes they predict. Their output is a set of gene structures, which identify the location and structure of the genes in the input sequence (i.e. [5, 6, 9, 15]).

One standard file format for storing gene structures is GTF. GTF stands for Gene Transfer Format. This format facilitates the storage of four types of “features”: exons,

coding exons, start codons, and stop codons. The difference between an exon and a coding exon is that a coding exon is both translated and transcribed while a non-coding exon is only transcribed. In general exons can be located in the UTR region, which defines them non-coding exons as the UTR is not translated into protein. This distinction is important both because it is required to determine the protein product of a gene and because coding exons must obey more rules (i.e. no in-frame stop codons) than non-coding exons. Each feature takes up one line of the GTF file and stores the information needed to uniquely identify that feature. Features are grouped into transcripts and transcripts are grouped into genes. A transcript corresponds to a single RNA transcript as described above and a gene corresponds to several transcripts which are alternative splices of each other. A complete specification of the GTF file format can be found in Appendix A.

1.3 Eval

As stated above, most automated gene prediction systems are typically based on large, complex probability models with many parameters. Changing these parameters can change the gene predictor's performance as measured by the accuracy with which it predicts the exons and gene structures in a standard annotation. While traditional measures of accuracy convey the performance of gene predictors [7, 10], these measures are often not enough to yield insight into why a gene predictor is performing well or poorly. A deep analysis requires considering many features of a prediction set and its relation to the standard set, such as the distribution of number of exons per gene, the distribution of predicted exon length, and accuracy as a function of GC percentage. Such statistics can reveal which parameters or parameter sets are working well and which need tuning.

When gene predictors are run on whole mammalian chromosomes they will be processing ~50-200 million bases of sequence and predicting thousands of genes. When gene predictors are run on whole mammalian genomes they will be processing up to billions of bases of sequence and predicting up to tens of thousands of genes (i.e. [9, 15]). An analysis system is needed to process this large amount of data and present it in a compact enough form for a human to view.

Because of the size and complexity of automated gene predictors and the volume of data generated by running them on the ever-growing amount of genomic sequence, we developed the Eval system. Eval is a software tool for analyzing and comparing gene sets. These summaries provide a human with a comprehensive overview of the large quantities of data produced by high-throughput automated gene predictors. It can compare a standard annotation set to a prediction set and generate a wide range of statistics showing how and to what extent the sets are similar. It can also compute statistics on a single set of gene annotations. It includes functionality to produce graphs of computed statistics versus characteristics of the genes and also graphs of distributions of any computed statistic across the gene set. It can do multi-way comparison between gene sets to determine the similarities and differences among multiple gene sets. It can

also build new gene sets from subsets of the genes which meet a specified set of criteria. Thus, Eval provides a powerful set of tools for analyzing the differences and similarities between gene prediction systems and adjusting their behavior.

Eval was primarily written for TWINSCAN [12], a *de novo* gene predictor. The TWINSCAN system compares the input genomic sequence to that of another organism, finds the similarities, and creates a new sequence, called the conservation sequence, which is given as input to the gene prediction software. Although Eval has some features which are primarily useful for viewing TWINSCAN gene predictions, such as conservation sequence graphs, the vast majority of its functionality is very useful for viewing and comparing gene sets from any source.

Although the GTF file format is a fairly simple and well defined format, data is often claimed to be in GTF format when it does not comply completely with the specification. Most data is generated in some proprietary format specific to the particular program or lab which produced it. These proprietary formats often differ in small subtle ways, such as the sequence being indexed starting at position 0 or 1, or the start/stop codon being inside or outside of the initial/terminal exon. If the data is to be effectively shared with others it must be in a standard, well defined format. Though many labs do convert their data to GTF format, the files they generate rarely comply completely with the specification. For this reason the GTF validator was created. The validator allows the user to verify that the data is in correct GTF format before sharing with others. This makes communication more efficient because the receiver does not have to locate and fix the subtle differences between the many file formats.

Chapter 2: User Level Documentation

This chapter provides user level documentation for the Eval package. Each program's input and output is described is described in detail and examples of use-cases for each program are presented.

All programs and libraries are written in Perl for use on Linux based systems. The programs have been tested extensively on Red Hat Linux 6 or greater and Perl version 5.6 or greater. The Perl Tk module version 8.0 or greater is required to load the graphical user interface (GUI) to the Eval package and gnuplot [4] version 3.0 or greater is required to display graphs when using the GUI. If gnuplot is not in the user's path the *EVAL_GNUPLOT* environment variable should be set to the full path and filename of gnuplot (i.e. /usr/bin/gnuplot).

Although the GTF specification does not state that all genes in a gtf file must be from the same sequence or in the same coordinate system, this is a requirement for using the Eval software. Any GTF file used by any of the programs or libraries described below must contain annotation of a single sequence with all genes in the same coordinate system (that of the sequence they annotate).

2.1 GTF Validator

The GTF validator, validate_gtf.pl, has two main functions: verifying correct GTF file format and verifying that the genes specified by the GTF file do not violate the rules of gene structure. The validator takes a GTF file and optionally a fasta file (see Appendix B for the fasta file format specification) containing the genomic sequence which the GTF file annotates. When run without the corresponding genomic sequence, the validator checks the file for format errors and that no genes violate the rules of gene structure (i.e. no coding exons after the stop codon, all transcripts contain coding region, etc). When run with the corresponding sequence the validator also checks that the gene structures could have come from this sequence (i.e. start and stop codons and splice sites have the correct sequence, the genes contain no in-frame stop codons prior to any annotated stop codon, etc.). Checking the GTF file with the sequence can also help to identify indexing problems in the file (i.e. off by 1 error) but increases the running time drastically.

All GTF fields as well as the <gene_id> and <transcript_id> attributes will be listed in angled brackets (<field name>) to designate them as GTF field names.

Arguments

These and all other arguments described in this document are listed in the same order they must be given to the program.

GTF File	The GTF file to validate.
Fasta File	The sequence that *GTF File* annotates, in fasta format. This argument is optional.

Options

These and all other options described in this document can be given to the program in any order but must come before all of the arguments to the function.

-t <*file*>	Writes each spliced transcript's sequence, including the start and stop codons, to *file*. This option can only be used if the optional *Fasta File* argument is given.
-f	Creates a new GTF file with the same name as *GTF File* but ending in ".fixed.gtf". The new file is identical to the original file but has no GTF format errors. If the input file is very badly formatted it may not be possible to automatically fix it. "Fixed" files should always be checked for correctness either by hand or by rerunning validate_gtf.pl.
-s	Outputs the <transcript_id> of each transcript containing an in-frame stop codon prior to either the annotated stop codon or the end of the gene if no stop codon is annotated.
-c	Suppress warnings about missing start/stop codons.
-p	Suppress warnings about non-standard splice sites.

Output

The validator's output is written to standard out. The first five occurrences of any error or warning are displayed in detail, but details about any additional occurrences are suppressed. This is done to make the output more readable. Since this program is used to check for correct file format it often finds systematic errors that occur thousands of times in a GTF file, and seeing details about a specific format error five times is just as informative as seeing it a thousand times. Following these detailed descriptions, the total number of each error and warning is listed. The last data reported are general statistics about the number of genes, transcripts, and coding exons in the file.

The validator is useful for checking a GTF file for errors before sharing the file or using it as input to another program. Whenever the GTF file may contain problems (the program that produced it has changed, the file was imported from somewhere else, etc.) it should be checked for errors before being shared or used. The validator is also useful for identifying common gene predictor errors during gene predictor development. Problems such as in-frame stop codons are identified allowing the developers to find and correct the error.

Often the source of gene annotation and genomic sequence are different, because different labs work to sequence organisms and annotate sequence. Also, many version of a particular sequence are often available, since the sequence, especially if it is a large-scale sequence like a whole chromosome, is constantly being updated and with more

reads and better assemblies until a final, complete version of the sequence is made available. The validator can be used to ensure that the annotation is for some particular version of the sequence and not some other.

2.2 Eval Overview

The Eval package is used to compare and analyze sets of gene predictions. It has six main functions which perform different types of gene set comparisons and statistical calculations. Each of the main functions is described below. This section gives an overview of what the Eval package can do, not instructions on how to do it. That information is given in the descriptions of the user interfaces to the Eval package in sections 2.3 and 2.4.

The inputs to all main Eval functions are sets of GTF files. For a description of the GTF file format see Appendix A. A set of GTF files is just an ordered list of GTF files each of which resides in its own coordinate system. When an Eval function compares GTF sets, the first GTF file from each list is compared to each other, then the second GTF files are compared, and so on. So, when comparing GTF sets the user must be certain that the GTF files in the first position of all GTF sets are in the same coordinates as each other, as are the files in the second position, and so on. In the case of whole genome comparisons, GTF sets would be loaded which contain a GTF file for each chromosome. Eval would compare the chromosome 1 GTF from one list to the chromosome 1 GTF from the other lists, then compare the chromosome 2 GTF files, and so on through the list. For using Eval to analyze single GTF files, GTF sets can contain only a single GTF file.

Eval is primarily used for gathering data on the coding region of genes. As such, it ignores any exon type features, as they are used to designate non-coding exons. exon type features are used to calculate the start and end of transcripts, but are never directly used in any comparison. Any Eval function which reports statistics on exons is, in fact, reporting statistics on CDS features (coding exons).

Some of the descriptions below use the term “transcript region”. The transcript region is defined as the area from the 5’ end of the 5’ most feature of the transcript to the 3’ end of the 3’ most feature of the transcript. In other words, the entire genomic region which is transcribed into RNA.

2.2.1 Statistics

Each main function of the Eval package uses the same set of statistics. Comparisons between sets of gene data require that one set be designated the annotation set and one set the prediction set. The statistics reported on this comparison show how similar the prediction set is to the annotation set. Although most statistics do not change when swapping the annotation and prediction sets (other than the prediction statistics becoming the annotation statistics and vice versa) some do and the distinction is important.

The statistics are organized into a hierarchy of three levels: Level, Type, Stat. Stat is the most specific of the three and each Stat contains a single statistic about the data. Level is the most general of the three and organizes all Stats into groups which contain data about similar kinds of objects (i.e. exons, genes, etc). Types are used to further partition all Stats at a given Level into groups containing data about similar kinds of objects (i.e. specific types of exons). Objects are designated as being of a certain Level and Type, and the Stats at a given Level and Type contain data only on the objects which are of that Level and Type. Each level of the hierarchy is described in detail below and an example of all Stats at every Level and Type can be seen in Appendix D.

Statistics are split into five Levels: *Gene*, *Transcript*, *Exon*, *Nuc*, and *Signal*. The *Gene* Level contains statistics which deal with genes, the *Transcript* Level contains statistics which deal with transcripts, and so on.

Each Level is further split into Types. Each Type is a subset of the statistics at a given Level which contain data on a specific subset of the objects at that Level. Whether or not an object at a given Level is of a certain Type must be able to be determined from that object alone, without making any comparisons to other objects. Examples of *Exon* Level Types are *Initial* and *Terminal*, which contain statistics only on exons which are initial or terminal exons, respectively. Note that determining that an exon is an initial exon or a terminal exon does not require any comparison to any other gene.

Each Level contains a set of Stats which are calculated for each Type of this Level. Stats contain the actual data reported by Eval, and are made up of two non-overlapping sets called General Stats and Comparison Stats. General Stats are those which can be calculated using a single object (no comparisons are needed). Examples of General Stats are *Average Length* and *Average Score*, since the score and length of an object do not depend on anything but the object itself. Comparison Stats are those whose calculations do require comparison to other objects. Comparison Stats are made up of groups of Substats which are organized into Comparison Stat Types. Comparison Stat Types are subsets of the objects at a given Level and Type, where membership in the subset requires comparison to some other object. Examples of Comparison Stat Types are *Overlap* and *Correct*, since the object must overlap or be correct as compared to some other object. For each Comparison Stat Type the same four Substats are calculated. The Substats are: *Count*, *Matched*, *Sensitivity* and *Specificity*. So for the *Overlap* Comparison Stat Type, four Stats are calculated for each Type at the current Level: *Overlap Count*, *Overlap Matched*, *Overlap Sensitivity*, and *Overlap Specificity*. *Count* is just the number of objects of the current Level and Type which are found to be part of this Comparison Stat Type subset. *Matched* is the number of annotation objects with which some prediction object matches to be included in the Comparison Stat Type subset. *Sensitivity* is defined as true positives divided by the sum of true positives and false negatives, but is calculated as *Matched* divided by the total number of annotation objects of this Level and Type. A positive indicates that the object is in the subset and a negative indicates that it is not. *Specificity* is defined as true positives divided by the sum of true positives and false positives and is calculated as *Count* divided by the total number of prediction objects of this Level and Type. By calculating *Sensitivity* and *Specificity* in this way they

are guaranteed never to exceed 100%. This is not true if they were calculated according to their definition since two or more prediction objects could match a single annotation object or a single prediction object could match two or more annotation objects, allowing the true positive count to be greater than either the total number of annotation objects or the total number of prediction objects at a given Level and Type.

Appendix D contains an example Eval report which contains all Stats at each Level and Type.

Below is a definition of each Type and Stat for each Level. Any object mentioned in a statistics description below can be assumed to be from the set the statistic is being calculated for unless it is explicitly stated to be an object from the annotation set being compared against.

Gene Level

Types

All	All genes.

General Stats

Count	The number of genes.
Ann Count	The number of genes in the set being compared against.
Total Transcripts	The total number of transcripts in all genes.
Transcripts Per	The average number of transcripts per gene.

Comparison Stat Types

Correct	A gene is in any of the these Comparison Stat Types if and only if one or more of its transcripts is in the *Transcript* Level Comparison Stat Type of the same name.
Exact	
Overlap	
Nuc Overlap	
All Introns	
All Exons	
Exact Intron	
Exact Exon	
Start Codon	
Stop Codon	
Start Stop	

Transcript Level

Types

All	All transcripts.
Complete	Transcripts which have both a start and a stop codon.
Incomplete	Transcripts which do not have both a start and a stop codon.

General Stats

Count	The total number of transcripts.
Ann Count	The total number of transcripts in the set being compared against.
Average Length	The average transcript region length.
Total Length	The sum of the length of each transcript region.
Average Coding Length	The average coding length of all transcripts, where the coding length is defined to be the sum of the length of all coding exons in a transcript.
Total Coding Length	The sum of the coding length of each transcript.
Average Score	The average of the sum of the scores of all GTF features of all transcripts.
Total Score	The sum of the score of each transcript.
Exons Per	The average number of coding exons per transcript.
Total Exons	The total number of coding exons in all transcripts.

Comparison Stat Types

Correct	A predicted transcript which exactly matches all features in some annotated transcript. The predicted transcript may contain features beyond either or both ends of the annotated transcript as long as the annotated transcript is not "closed" with a start or stop codon on that end.
Exact	A predicted transcript which is identical to some annotated transcript over their entire length of both transcripts.
Overlap	A predicted transcript whose region overlaps some annotated transcript's region and is on the same strand.
Nuc Overlap	A predicted transcript which has at least one coding exon which overlaps a coding exon from some annotated transcript by at least one base pair. The two transcripts must be on the same strand.
All Introns	Each intron in the annotated transcript is also in the predicted transcript and the predicted transcript contains no additional introns which overlap the annotated transcript. Similar to the *Correct* measure the predicted transcript can contain introns beyond the end of the annotated transcript if the annotated transcript is not "closed" with a start or stop codon on that end.
All Exons	Similar to the *All Introns* measure except that all exons instead of introns must be identical. This is the same as the *Correct* measure expect that the start and stop codon are not checked.
Exact Intron	A predicted transcript which matches at least one intron exactly to some intron in an annotated transcript.
Exact Exon	A predicted transcript which matches at least one exon exactly to some exon in an annotated transcript.

Start Codon	A predicted transcript which has exactly the same start codon as some annotated transcript.
Stop Codon	A predicted transcript which has exactly the same stop codon as some annotated transcript.
Start Stop	A predicted transcript which has exactly the same start and stop codons as some annotated transcript.

Exon Level

When two of the same exons exist in a data set they are treated as a single exon when calculating the *Exon* Level Stats. Two of the same exons can exist in the same data set when transcripts which are alternative splices of each other have some exon in common. Exons are considered to be the same exon if they have the same <start>, <end>, and <strand> values. In this way, the *Exon* Level Stats are a set of statistics on all unique exons in the data set.

Types

All	All coding exons.
Initial	The 5' most coding exon in any multi-exon transcript which has a start codon.
Internal	All coding exons which are not Initial, Terminal or Single.
Terminal	The 3' most coding exon in any multi-exon transcript which has a stop codon.
Single	The only coding exon in any transcript containing only one coding exon.
Intron	All introns.

General Stats

Count	The total number of objects.
Ann Count	The total number of objects in the set being compared against.
Average Length	The average length of all objects.
Total Length	The sum of the lengths of all objects.
Average Score	The average of the scores of all objects.
Total Score	The sum of the scores of all objects.

Comparison Stat Types

Correct	A predicted object which is identical (same <start>, <end>, and <strand> values) to some annotated object.
Overlap	A predicted object which overlaps some annotated object on the same strand.
Overlap 80p	A predicted object which overlaps some annotated object on the same strand by at least 80% of the length of the longer object.
Splice 5	A predicted object which has the same 5' boundary as some annotated object on the same strand.

Splice 3	A predicted object which has the same 3' boundary as some annotated object on the same strand.

Nuc Level

Types

All	Any nucleotide covered by any exon.
Initial	Any nucleotide covered by an Initial exon.
Internal	Any nucleotide covered by an Internal exon.
Terminal	Any nucleotide covered by a Terminal exon.
Single	Any nucleotide covered by a Single exon.
Intron	Any nucleotide covered by an intron.

General Stats

Count	The total number of nucleotides.
Ann Count	The total number of annotated nucleotides.

Comparison Stat Types

Correct	If the Type is not Intron, then a nucleotide is correct anytime any annotated coding exon overlaps it. If the Type is Intron then the nucleotide is correct any time any annotated intron overlaps it.

Signal Level:

Types

Splice Donor	The 3' end of an Initial or Internal exon.
Splice Acceptor	The 5' end of a Terminal or Internal exon.
Start Codon	The start codon of any transcript.
Stop Codon	The stop codon of any transcript.

General Stats

Count	The total number of signals of this Type in the prediction.
Ann Count	The total number of signals of this Type in the annotation.

Comparison Stat Types

Correct	The signal is correct if is appears exactly in both the prediction and the annotation. For splice sites this means they have the same position and are on the same strand. For start and stop codons this means that all three bases of the codon are in the same location and on the same strand.

2.2.2 Evaluate

The Evaluate function is the main function of the Eval package. It is used to compare a set of prediction GTF files to a set of annotation GTF files and is useful for finding the degree of similarity between many aspects of the two sets. Comparisons between the two sets of GTF files are reported as a set of statistics, which is described above. The output from the Evaluate function is called an Eval report. The Evaluate function is the most used Eval function and many of the other functions are nothing but alternative ways of viewing its results.

This function is primarily used for comparing the output from gene predictors to some standard annotation. It reports to what extent the gene predictions and the annotation are similar or different. This is useful for judging the performance of a gene predictor. It can handle comparisons of everything from single genes, to whole chromosomes, to whole genomes. The function can also be used to compare to sets of predictions to each other to see how similar they are.

2.2.3 General Statistics

This function is used to get general statistics about a single GTF set. It reports a subset of the statistics reported by the Evaluate function, containing all Levels, all Types, but only General Stats. The output of the General Statistics function is also referred to as an Eval report as it is in the same format as the output from the Evaluate function but with some values (those for Comparison Stats) left out.

This function is useful for getting a general overview of a single GTF set. When first dealing with a new genome it is good to know what, on average, genes of this genome look like. For example, how many exons per transcript does it have? What is the average exon length? What is the average gene-density? This is useful information which varies from genome to genome. This function is also useful for tuning parameters of a gene predictor to output genes with some specific characteristics. Suppose a gene predictor is currently outputting transcripts which, on average, contain 7 exons and have a length of 30,000 base pairs, but the organism which it is being run on has genes that, on average, contain 9 exons and have a length of 40,000 base pairs. This change can probably be achieved by altering the input parameters to the gene predictor, but gene predictors are made of complex mathematical models and it is rarely clear how changing an input parameter will change the output. This function allows the user to check that the desired changes in the output did occur and that no additional, undesired changes in others statistics occurred.

2.2.4 Filter

This function is used to select a subset of GTF style objects (Genes, Transcripts, and Exons) from a GTF set and create a new GTF set from them. The selected subset may be any subset calculated by the Evaluate function. This includes all Types and Comparison

Stat Types listed in section 2.2.1 above. The union, intersection, and compliment of any valid subsets may also be selected.

The Filter function is useful for tacking down bugs or improving performance in gene predictors. All predicted transcripts which overlap but do not exactly match an annotated transcript can be selected and checked closer manually to find why they are not exactly matching it. The Filter function is also good for finding examples. Perhaps a gene which gene predictor 1 predicts correctly but gene predictor 2 predicts incorrectly is wanted. This is easy to find using the Filter function but would be tedious to find by hand.

2.2.5 Graph

This function is used to graph a certain statistic as a function of some other computable value on the objects. The objects first are split into consecutive, non-overlapping bins according to some X-Split type at some Level (see below). The statistic being graphed is then computed for each bin. Bins are graphed on the x-axis and the value of the statistic being computed is graphed on the y-axis.

Level

Gene	Separate objects into bins of genes.
Transcript	Separate objects into bins of transcripts.
Exon	Separate objects into bins of exons.

X-Splits

GC%	Separate objects into bins according to each object's GC percentage.
Match%	Separate objects into bins according to the percentage of matched bases in each object's conservation sequence.
Mismatch%	Separate objects into bins according to the percentage of mismatched bases in each object's conservation sequence.
Unaligned%	Separate objects into bins according to the percentage of unaligned bases in each object's conservation sequence.
Length	Separate objects into bins according to the length of each object.

Bins are consecutive, non-overlapping ranges of values of the X-Split type. The number of bins and size of each bin is determined by parameters to the function. Values for the y-axis can be any statistic computed in the Evaluate function. If the split is made at the Exon Level the statistic cannot be from the Transcript of Gene Level because a single transcript or gene may have had its exons split into more than one bin. Similarly if the split was done at the Transcript Level, the y-axis statistic cannot be from the Gene Level.

The Graph function is useful for seeing how a statistic is changing as compared to another property of the objects from which the statistic was computed. Graphing *Transcript Sensitivity* against *Transcript Length* can show if a gene predictor is having problems predicting short or long transcripts relative to the other. Often times, genes with different GC percentages have different characteristics [13]. The Graph function can be used to see if a certain gene predictor is having trouble with genes in a particular

GC range or just to see how genes in a particular GC range look in general. The *Match%*, *Mismatch%*, and *Unaligned%* X-Splits are useful primarily with TWINSCAN, but can also be used with any other gene predictor which uses some secondary sequence.

2.2.6 Overlap

The Overlap function is used to build clusters of GTF style objects which share some property, called the overlap property. A cluster is defined as a group of objects, each of which shares the overlap property with at least one other object in the cluster and no objects outside the cluster. Given one or more GTF sets, this function builds clusters of objects and outputs statistics describing how the objects were clustered together. Possible overlap properties are:

Transcript Exact Exon Overlap	Transcripts which have one or more exons in common.
Transcript Exact Intron Overlap	Transcripts which have one or more introns in common.
Transcript Coding Overlap	Transcripts whose coding regions overlap by at least one base pair.
Transcript Region Overlap	Transcripts whose regions overlap by at least one base pair.
Transcript 80p Both Region Overlap	Transcripts whose regions overlap by at least 80% of the length of the longer region.
Transcript 80p Region Overlap	Transcripts whose regions overlap by at least 80% of the length of the shorter region.
Transcript Exact Overlap	Transcripts which are identical (start and stop codons and all coding exons are the same).
Exon One Base Overlap	Exons which overlap each other by at least 1 base pair.
Exon 80p Both Overlap	Exons which overlap each other by at least 80% of the longer exon.
Exon 80p Overlap	Exons which overlap each other by at least 80% of the shorter exon.
Exon Exact Overlap	Exons which are identical (<start> and <end> values are the same).

*All overlap properties require that the two objects are on the same strand.

Once the clusters are built they are separated into cluster types. A cluster type corresponds to a subset of the GTF sets which were given as input to this function and contain clusters with objects from all and only the GTF sets in the subset. For example if clusters were built from three sets of GTF files, label them A, B, and C, clusters which contain objects from A and B and none from C would be one cluster type, and clusters which contain only objects from B would be another. A cluster type exists for each non-empty subset of the input GTF sets (this example has seven cluster types: A, B, C, AB,

AC, BC, and ABC). For each cluster type, the number of clusters in that type, as well as the number of objects in clusters of this type which came from each input GTF set is reported. In the example above, if only two clusters of type AB were created and one contained one object from A and one object from B and the other contained two objects from A and one from B, the results reported by this function would include a description of the AB cluster type which would state that it contained two clusters, three objects from the A input set, and two objects from the B input set.

The Overlap function allows the user to see how multiple sets of gene predictions are similar to one another. All other functions of Eval do only pair wise comparison but overlap analysis can find three-way or greater similarities between GTF sets. Though it is useful for looking a one or two sets, it is most useful for seeing how three or more sets compare to each other, as other Eval functions can give more detailed analysis of two set comparisons.

Building clusters of identical genes from a standard annotation set and two prediction sets from two gene predictors can show how similar the prediction sets are as compared to the annotation set. It could show that the two gene predictors are predicting the same or completely separate sets of correct and incorrect genes. If the two gene predictors correct gene sets have a small intersection and their incorrect gene sets have a large intersection, then the two gene predictors could be combined to create a system which has both a higher sensitivity and specificity than either one its own. This would signal that either gene predictor could benefit from incorporating features of the other.

2.2.7 Distribution

The Distribution function is used to see how the density of objects changes across values of some property. For example you could use this to view a distribution of *Exons Per Transcript*, which would report the number of transcripts with *n* exons, for all values of *n*. Instead of reporting all values of *n*, the function can report bins of values of *n* to make the results easier to read and to allow for continuous values or small sample spaces.

Distributions which can be calculated are:

Transcripts Per Gene	The number of transcripts in a gene.
Exons Per Transcript	The number of exons in a transcript.
Transcript Length	The length of a transcript's region.
Transcript Coding Length	The sum of the length of all coding exons of a transcript.
Exon Length	The length of an exon.
Exon Score	The score of an exon.

Distributions are useful for seeing how some property is distributed across the data. The Evaluate function gives average values but averages can be misleading when they are coming from certain types of distributions, such as a bi-modal distribution. The average

value can give no indication of the distribution of values in the data and may even be a value that is never or rarely seen in the data. The distribution of gene lengths could be plotted for an annotation set and a prediction set and this could show that the prediction set is over- or under-predicting short and long genes relative to moderate length genes, which is something that an average value alone cannot show.

2.3 Eval GUI

2.3.1 Overview

eval.pl is the graphical user interface for the Eval package. It is the easiest way to use the Eval package and is more efficient when multiple analyses are being run. The user can load one of more GTF sets into memory and analyze them using the Eval package. This allows multiple analyses which use the same data set to be run without having to reload the data to memory. Since whole chromosome or genome data sets are very large, loading them often takes a significant fraction of the total time of the analysis, so keeping the data in memory provides a considerable decrease in the total time required for multiple analyses.

The Eval GUI contains a help system consisting of several postscript files. These files should be located in a directory called *help* located off of the directory containing eval.pl. The files can be moved to some other directory but the *EVAL_HELP* environment variable must then be set to that directory in order to view the files (i.e. /usr/local/eval/help). The GUI attempts to display the files using ghostview (gv) [2]. If the ghostview program is not in the user's path, the *EVAL_GV* environment variable should be set to the full path and filename of ghostview or some other postscript viewer program (i.e. /usr/X11R6/bin/gv).

2.3.2 Loading the GUI

Running eval.pl starts the GUI. It has no required arguments but can optionally take one or more list files to load into memory. A list file is used to load GTF sets into memory. Each line of a list file contains the information for loading a single GTF file and optionally nucleotide and conservation sequence files. Each line contains three tab separated fields and has the following format:

<GTF filename> [nucleotide sequence filename] [conservation sequence filename]

The nucleotide sequence should be in fasta format. Both sequence files are optional, and either can be included with or without the other. If the conservation sequence is included and the nucleotide sequence is not, two tabs should be places between the GTF filename and the conservation filename. All files loaded from a list file make up a single GTF set.

The program has the following options:

-c	Specify a file, other than .evalrc in the users home directory (see section 2.3.3 below), from which to load the users options.
-g	Load command line arguments as single GTF files unless they end in .list.
-l	Load command line arguments as list files unless they end in .gtf or .gff. This is the default.
-n	Do not load nucleotide sequence or conservation sequence files.
-v	Turn verbose mode on. This will send status reports to standard error.
-V	Turn really verbose mode on. This is normal verbose mode plus reports of all errors and warnings generated while loading individual GTF files.
-h	Display the usage statement and exit.

The GUI is organized around and provides access to the six main functions of the Eval package. At the top of the screen is a menu bar which is described in the section 2.3.3 below. Under that is a horizontal bar of buttons, each of which corresponds to one of the main Eval functions. Clicking one of these buttons displays the screen from which its function can be run. Each individual function's screen is described in its own section below.

2.3.3 Menus

A standard menu bar appears across the top of the window. It contains menus entitled *File*, *Edit*, and *Help*. Each of these menus are described in detail below.

The *File* menu contains four commands: *Open*, *Save*, *Remove* and *Exit*. *Open* opens a Open File dialog box which allows the user to open new list or GTF files. *Save* allows the user to save any GTF set currently in memory. This is used to save new GTF sets generated by the Filter function. GTF sets containing only one GTF file save only that file and allow the user to specify the filename to which it is saved. GTF sets containing more than one GTF file are saved as a new list file, under a filename selected by the user, and all GTF files in the set are saved in the same directory as the list file under filenames of the form *filename.#.gtf*, where *filename* is the name the list file was written to and # is the position of each GTF file in the list. The *Remove* command allows the user to unload any of the GTF sets from memory. The final command of the File menu is *Exit*, which unloads all GTF sets from memory and closes the GUI.

The *Edit* menu contains a single command, *Options*, which brings up the *Edit Options* screen. This allows the user to edit his .evalrc file which contains the preferences to be loaded at startup. The *Edit Options* screen has two panes. The first allows the user to select, for each Level, which Stats and Types are included in any Eval report. If a box is unchecked, it means that that stat or type will be left out of all reports generated. The second pane allows the user to select the graph resolutions for each type of graph X-Split. Two options are available for graph resolutions. The first is the Uniform resolution which allows the user to specify a *Minimum X Value*, *Bin Size*, and *Number of Bins*, and generates *Number of Bin*s consecutive bins, each having size *Bin Width* and the first

starting at *Minimum X Value*. The second graph resolution type is User Defined. This allows the user to create bins of any size in any location. The only restrictions are that bins cannot overlap and there can be no gaps between bins. So, if a bin is defined from 0-100 and a second is defined from 150-300, a new bin from 100-150 is automatically added. The bottom of the screen has buttons to save the options and close the window. Closing the window discards any unsaved changes to the options.

The Help menu contains two commands: *About*, which displays a dialog with general information about the eval.pl program, and *Help*, which opens the Eval package documentation.

2.3.4 Eval Screen

This screen provides access to the Evaluate function of the Eval package. A single annotation GTF set is selected from the upper listbox and one or more prediction GTF sets are selected from the lower listbox. The *Run Eval* button starts the comparison. Once the calculations are complete the results are displayed in a new window. The report contains three sections: Summary Statistics, General Statistics, and Detailed Statistics. The Summary Statistics section reports the *Correct Sensitivity* and *Correct Specificity* for *All Genes*, *All Transcript*, *All Exons* and *All Nucleotides*. This gives a good overview of how similar the GTF sets are. The General Statistics section reports all General Stats, organized into Levels and Types. The Detailed Statistics section reports all Stats, sorted by Level and Type. Each prediction GTF set has its own column of values in each section of the report. All sections include Stats on all prediction sets that were selected, but the General Statistics section also displays the General Stats for the annotation set. Buttons to close the output window and to save the output are located at the bottom of the window. The *Save* button opens a Save File dialog box which allows the user to chose a file to which the output is saved. Files are saved in text format with each line having the statistic name on the left followed by the value for each prediction set. Each prediction has its own column of values and columns are tab separated to make the file easy to load into a spreadsheet. Each label and value is also padded with spaces to make the report readable in a standard text editor using a fixed width font. The *Close* button closes the window displaying the results.

2.3.5 GenStats Screen

This screen provides access to the General Statistics function of the Eval package. GTF sets are selected from the listbox and the *Get Stats* button is used to start the calculations. The results are displayed and saved in identically the same was as they are on the Eval screen, except that only the General Statistics section is reported.

2.3.6 Filter Screen

This screen allows the user to filter one or more GTF sets, based on comparison to some annotation GTF set, using the Filter function of the Eval package. The initial screen is

for the selection of a single annotation set and one or more prediction sets, exactly as in the Eval function. The *Select Filters* button displays the next screen which is used to select the filter to apply. The top two listboxes allow the user to choose single filters to use. Selecting a Level in the left listbox displays a set of filters for that Level in the right listbox. The *Add Filter* button at the bottom of the screen adds the currently selected filters to the *Filter Key* listbox. The *Remove Filter* button removes all highlighted filters in the *Filter Key* listbox. All filters in the *Filter Key* listbox are assigned one character alphabetic labels, which are used to represent that filter in the Filter String. The Filter String tells the program how to apply the filters you have chosen. Each single filter from the *Filter Key* listbox specifies a subset of the objects in the GTF sets which are being filtered and can be combined in several ways. Single filters can be joined with set intersection by separating them with "&&" (i.e. "A&&B") or nothing (i.e. "AB"), or with set union by separating them with "||" (i.e. "A||B"). Filters can also be group with parenthesis (i.e. "A||(BC)") to specify the order which in filters should be applied. Filters may be negated with a set complement by "!" (i.e. "!A" or "!(A||B)") which selects all objects which are not selected by the negated filter. The *Run* button will filter the selected predictions according to the Filter String, and add the new, filtered GTF sets to the list of possible GTF sets to use in each Eval function. The filtered GTF sets will be added under the same name as the GTF set they were created from except that the string in the bottom textbox is inserted into the name prior to ".gtf" or ".list". If the bottom textbox is empty "filtered" is inserted into the name. To save the filtered GTF sets for future use, use the *Save* command under the *File* menu.

2.3.7 Graph Screen

This screen allows users to make Eval style graphs from one or more prediction sets string the Graph function of the Eval package. The first screen is used to select a single annotation set and one or more prediction sets from which to make graphs. Graphs are made only from the prediction sets, not from the annotation set. The *Select Graphs* button moves to the next screen where the X-Split and Level are chosen. The top listbox selects the Level at which the data will be split, and the middle listbox selects the property by which the data are split. The *Selected Graphs* listbox shows all the graphs which will be calculated. The *Add* button adds the currently selected Level/X-Split combination to the *Selected Graphs* listbox. The *Remove* button removes any currently selected graphs in the *Selected Graphs* listbox. The *Create Graphs* button will calculate all graphs specified in the *Selected Graphs* listbox and move to the next screen. All possible y-values are calculated for each Level/X-Split combination. All graphs are calculated using the user's graph resolution options, accessible from the *Options* command under the *Edit* menu. The final screen selects which graphs to display or save. The top listbox selects one or more prediction sets to include in the graph, the middle listbox selects the Level/X-Split combination, and the bottom listboxes select the value to graph on the y-axis. All available y-values for the current Level/X-Split are listed. The *Choose Graphs* button will return to the previous screen so that new graphs can be created. The *View* button displays the currently selected graph using gnuplot. The *Save* button will display a Save File dialog box which saves the graph as a tab delimited text file. The first line contains the y-value and Level/X-split of this graph the second line

contains a tab delimited list of the prediction set names. All following lines contain the bin followed by a tab delimited list of the value of this bin for each prediction set, in the same order as the names are listed on the second line. Each bin has the format "# - #" where the first "#" is the lower-bound for the bin and the second "#" is the upper-bound for the bin.

2.3.8 Overlap Screen

This screen allows the user to build overlap clusters from the GTF sets using the Overlap function of the Eval package. The upper listbox is used to select the GTF sets from which to build the clusters. The lower listbox is used to select the overlap type, which specifies how to build the clusters. The *Get Overlap* button will calculate overlap clusters from the selected GTF sets using the selected overlap type and display the results in a new window. The top of the new window shows the *Label Key* which maps labels to GTF set names. Below the *Label Key* is a list of all possible cluster types and the number of clusters of that type, and for each GTF set the percentage of objects from that set which are in clusters of this type, and finally the percentage of total clusters which are of this type. At the bottom of the screen is a *Save* button, which opens a Save File dialog box which allows the results to be saved to a tab delimited text file in the same format as the display window with all data separated by tabs, and a *Close* button which closes the display window.

2.3.9 Dist Screen

This screen allows the user to build Eval distributions from one or more GTF sets. The upper listbox is used to select one or more GTF sets from which the distributions will be made. The lower listbox is used to select the type of distributions to generate. The *Get Distribution* button generates one distribution for each selected GTF set for each selected distribution type, and displays the next screen. On this screen the listbox is used to select the GTF set distribution to view or save. Two textboxes allow the user to enter values which set the upper-bound of the distribution and a bin size to use when reporting the data. The lower-bound of any distribution is always zero. Any data point occurring above the upper bound will be place in an "extra" bin just past the upper bound. A checkbox allows the user to change the distribution to a cumulative distribution, where the reported value for each bin is the size of the bin plus the sizes of all bins which are located below the current bin. The *Back* button allows the user to return to the previous screen to select new GTF sets and predictions to generate. This will cause any previously calculated distributions to be discarded. The *View* button will display the distribution in a new window using gnuplot and the *Save* button will display a Save File dialog box which allows the user to save the distribution to a text file. If saved as a text file the first line will be the distribution name followed by lines containing single tab separated bin-size pairs. The bin has format "#-#", where the "#" symbols are two numbers specifying the lower- and upper-bound of the bin, and the size is a single number representing how many objects fall into that bin. On the final line the bin has the form "#+" where "#" is the upper bound of the maximum valued bin and its size represents how many objects fall above the maximum bin.

2.4 Eval Command Line Interfaces

The Eval package includes, for each main Eval function, a Perl script to run that function from the command line. The command line interfaces to the Eval functions provide quick, efficient access to each Eval function. If only a single analysis is being run, they can save time by avoiding the overhead associated with loading and displaying the GUI. Also, if many separate analyses need to be run, the command line interfaces allow them to be run on a compute cluster. The command line interfaces do not require the Tk Perl module.

2.4.1 evaluate_gtf.pl

This is the command line interface to the Evaluate function of the Eval package. It takes a set of annotation GTF files and one or more sets of prediction GTF files, runs the Evaluate function to compare all predictions to the annotation and outputs all statistics calculated. It takes the following arguments:

Annotation	A list file containing the annotation GTF set.
Prediction 1	A list file containing the first prediction GTF set.
Prediction 2	A list file containing second next prediction GTF set.

Prediction 2 is optional and can be followed by any number of additional prediction GTF sets to be compared to the annotation set.

The program has the following options:

-g	The arguments are single GTF files rather than list files.
-v	Turns verbose mode on.
-h	Displays the usage statement and exits.

The output is sent to standard out and is in the same format as when saved to a text file from the GUI.

2.4.2 get_general_stats.pl

This is the command line interface to the General Statistics function of the Eval package. It takes one or more sets of GTF files and reports Eval General Stats about them. It takes the following arguments:

GTF Set 1	A list file containing the first GTF set.
GTF Set 2	A list file containing the second GTF set.

GTF Set 2 is optional and can be followed by any number of additional GTF sets for which to calculate general statistics.

The program has the following options:

-g	The arguments are single GTF files rather than list files.
-v	Turns verbose mode on.
-h	Displays the usage statement and exits.

The output is sent to standard out and is in the same format as when saved to a text file from the GUI.

2.4.3 filter_gtfs.pl

This program is the command line interface to the Filter function of the Eval package. It takes a filter file, a set of annotation GTF files, and one or more sets of prediction GTF files, filters the predictions according to the filter file, and saves the results to files with the same name as the input prediction files but ending in .filtered.list or .filtered.gtf. It takes the following arguments:

Filter File	A file specifying the filters to be used. See below for format.
Annotation	A list file containing the annotation GTF set.
Prediction 1	A list file containing the first GTF set to be filtered.
Prediction 2	A list file containing the second GTF set to be filtered.

Prediction 2 is optional and can be followed by any number of additional GTF sets to be filtered.

The program has the following options:

-f	Prints all valid single filters to standard out and exits.
-g	The arguments are single GTF files rather than list files.
-h	Displays the usage statement and exits.

Each line of the *Filter File* should have a single character label, followed by a dash, followed by a single filter, all separated by any amount of whitespace. As many single filters as needed can be listed but all must have a different one character label. After all filters have been listed, there should be an empty line (no empty lines are allowed prior to this one), and then a line containing the Filter String. The Filter String is as described in the section 2.3.6 above.

2.4.4 graph_gtfs.pl

This program is the command line interface to the Graph function of the Eval package. It takes a graph file, an annotation set, and one or more prediction sets, creates the graphs

specified in the graph file, and outputs the data to standard out. It takes the following arguments:

Graph File	A text file specifying the graphs to be made. See below for format.
Annotation	A list file containing the annotation GTF set.
Prediction 1	A list file containing the first prediction GTF set to be graphed.
Prediction 2	A list file containing the second prediction GTF set to be graphed.

Prediction 2 is optional and can be followed by any number of additional GTF sets to be graphed. The *Graph File* is a text file with each line describing a graph to be created. Each line has the format: "Y-Level::Y-Type::Y-Stat vs X-Split::X-Level". Where the Y-terms specify the statistic to be graphed and the X-terms specify how and at what level to split the objects into bins.

The program has the following options:

-G	Prints all possible values for the X-Split, Level, and y-axis to standard out and exits.
-g	The arguments are single GTF files rather than list files.
-r <*Res File*>	A text file specifying the location and size of the bins on the x-axis. See below for format.
-h	Displays the usage statement and exits.

If no *Res File* is specified with the –r option, the users .evalrc file is used. If the user has no .evalrc file then a default resolution is used.

The *Res File* specifies the bins for each split type. Each line has the X-Split type followed by either "User" or "Uniform". "User" is used to specify the size and location of each bin, and should be followed by a series of increasing numbers, which are the bounds of the bins. The first bin runs from the first number to the second, the second bin runs from the second number to the third, and so on. "Uniform" is used to specify a list of identically sized bins, and should be followed by three numbers. The first is the lower bound of the lowest bin, the second is the size of each bin, and the third is the number of bins. All fields in the *Res File* are separated by a single tab.

2.4.5 get_overlap_stats.pl

This is the command line interface to the Overlap function of the Eval package. It takes one or more sets of GTF files, builds overlap clusters from them using the specified Overlap Mode, and reports statistics on the number and composition of each cluster type. It takes the following arguments:

GTF Set 1	A list file containing the first GTF set.
GTF Set 2	A list file containing the second GTF set.

GTF Set 2 is optional and can be followed by any number of additional GTF sets from which to build clusters.

The program has the following options:

-m <*mode*>	Sets the overlap mode to *mode*, which must be a positive integer. Valid options are listed in the usage statement and below. Default is 1.
-g	The arguments are single GTF files rather than list files.
-v	Turns verbose mode on.
-h	Displays the usage statement and exits.

Overlap Modes:

1	*Transcript Exact Overlap*
2	*Transcript Exact Exon Overlap*
3	*Transcript 80p Region Overlap*
4	*Exon One Base Overlap*
5	*Exon 80p Overlap*
6	*Transcript Region Overlap*
7	*Exon Exact Overlap*
8	*Transcript Exact Intron Overlap*
9	*Transcript Coding Overlap*
10	*Exon 80p Both Overlap*
11	*Transcript 80p Both Region Overlap*

Overlap Modes are described in the section 2.3.8 above.

The output is sent to standard out and is in the same format as described in section 2.3.8 above.

2.4.6 get_distribution.pl

This is the command line interface to the Distribution function of the Eval package. It takes a distribution type, the upper bound of the values to be displayed, the size of the bins which the data should be split into, and one or more sets of GTF files and outputs the distribution of the given distribution type over the input GTF files. It takes the following arguments:

Max Value	The highest value to list in the distribution. All value above this will be placed in a special bin, which goes from this value to infinity.
Bin Size	The size of the bins in which the data will be places.
GTF Set 1	A list file containing the first GTF set.

GTF Set 2	A list file containing the second GTF set.

GTF Set 2 is optional and can be followed by any number of additional GTF sets for which to calculate general statistics.

The program has the following options:

-d	This displays a list of all valid distribution types
-m <*mode*>	Set the distribution type to *mode*, which must be a positive integer. Valid options are listed in the usage statement and below. Default is 1.
-g	The arguments are single GTF files rather than list files.
-h	Displays the usage statement and exits.

Distribution Types:

1	*Transcripts Per Gene*
2	*Transcript Length*
3	*Transcript Coding Length*
4	*Exons Per Transcript*
5	*Exon Length*
6	*Exon Score*

The output is sent to standard out and is in the same format as when saved to a text file from the GUI.

Chapter 3: Code Level Documentation

3.1 Overview

This chapter describes the structure and inner workings of all source code for all programs and libraries in the Eval package. A familiarity with the use of the programs and libraries described in Chapter 2 is assumed.

All code was written in Perl for several reasons. First, Perl makes text processing very easy and dealing with the GTF files requires a significant amount of text processing. Second, many bioinformatics scripts and programs are written in Perl so by making the libraries Perl libraries bioinformaticists can use the libraries' data structures and functions in their code. Also, the code is not exceptionally computationally intensive, so the speed loss from using Perl instead of a faster, compiled language like C is not detrimental. Three-way comparison of full human genome annotations, comprising over 80,000 genes and 600,000 exons, takes less than 30 minutes on a 900MHz Pentium running Red Hat Linux 7.1. Comparisons between two GTF sets containing thousands of genes each take less than 30 seconds on the same machine. As such, the benefits of Perl outweighed those of faster languages, and Perl was chosen as the implementation language.

3.1.1 Data Types

Below is a list of data types that variables can take. Throughout this document all data types are listed in the same `font`. Although Perl does not have explicit data types, the documentation provides them for all variables because the code requires that variables are of a certain type and knowing the expected type of each variable makes the documentation and code easier to understand. The most general, commonly used data types are listed below. Other, more complex data types are introduced in the sections which they are used.

`array`	Perl array
`aref`	Reference to an `array`
`hash`	Perl hash
`href`	Reference to a `hash`
`int`	Integer value
`float`	Floating point value
`Boolean`	Boolean value*
`fh`	Filehandle
`fref`	Reference to a function
`pvar`	Any Perl data type

* A `Boolean` is simply an `int` which takes the value of 0 for false and is otherwise true.

3.1.2 Naming Schemes

Variable names and function names are also displayed in distinctive fonts in order to identify them and make the documentation easier to read.

Throughout the code a naming scheme is used to identify different types of variables and functions. Constant values are listed in all capitals (*CONSTANT*), global variables are listed with the first letter of each word capitalized (*Global_Variable*), local variables and public functions are listed in all lowercase (*local_var*, public_function), and private functions are preceded by an underscore and listed in all lowercase (_private_function).

3.2 GTF.pm

3.2.1 Overview

The GTF.pm library contains data structures used to store and provide easy access to a GTF file. It defines four objects: `GTF`, `GTF::Gene`, `GTF::Transcript`, and `GTF::Feature`. Since all code in the Eval package imports the entire GTF.pm library the `GTF::` is not needed so `GTF::Gene`, `GTF::Transcript` and `GTF::Feature` objects will be referred to as `Gene`, `Transcript` and `Feature` objects. The `GTF` object stores all data for a single GTF file, the `Gene` object stores all data for a single gene, the `Transcript` object stores all data for a single transcript, and the `Feature` object stores all data for a single GTF feature. A feature is a single line of the GTF file, a transcript contains all features with the same <transcript_id> and a gene contains all transcripts with the same <gene_id>.

The `Gene` object stores the <seqname>, <source>, <strand>, and <gene_id> GTF fields as well as a list of all transcripts of this gene, stored as `Transcript` objects. The `Transcript` objects stores the <transcript_id> as well as a list, for each GTF feature type (exon, CDS, start_codon, and stop_codon) , of all `Feature` objects of that type in this transcript. The `Feature` object stores the remaining GTF fields (<start>, <end>, <score>, and <frame>) each of which is specific to a single feature.

The standard use of the GTF.pm library is to store in memory and provide easy access to the data in GTF files. A GTF file is loaded into a `GTF` object by calling the constructor with the appropriate arguments (see below). Lists of all genes, transcripts, or coding exons from the file can then be retrieved as arrays of GTF.pm style objects (`Gene`, `Transcript`, and `Feature`). Each of these GTF.pm style objects contains functions to retrieve all data associated with them.

All objects defined in the GTF.pm library are Perl style objects. Perl style objects are essentially just `hash` objects with fields for each of the object's global variables and functions. This is abstracted away when using the objects, but the objects' code does show these details. In the code of a Perl object, each function has a required first

argument which is the object itself. So, in the code of GTF.pm, all non-constructor functions take an addition argument, before all other arguments, which is the object the function is being called on. This argument should never be given when using the function as it is automatically added by Perl, and, as such, is left out of all function descriptions below.

3.2.2 GTF Object

The basic function of the `GTF` object is to parse a GTF file, store all data associated with it, and provide access to that data.

Global Variables

`aref`	*Genes*	A reference to an `array` of `Gene` objects representing all genes in this GTF file.
`aref`	*Transcripts*	A reference to an `array` of `Transcript` objects representing all transcripts in this GTF file.
`aref`	*CDS*	A reference to an `array` of `Feature` objects representing all coding exons in this GTF file.
`string`	*Filename*	The name of the file from which this GTF was loaded.
`string`	*Sequence*	The name of the file containing the genomic sequence which this GTF annotates.
`string`	*Conseq*	The name of the conservation sequence file corresponding to genomic sequence this GTF annotates.
`hash`	*Total_Conseq*	A `hash` containing the number of match, mismatch, and unaligned bases in the conservation sequence for this GTF file. If no conservation sequence is supplied to the constructor all counts are set to -1.
`hash`	*Total_Seq*	A `hash` containing the number of A, C, G, T, and N bases in the genomic sequence of this GTF file. If no genomic sequence is supplied to the constructor all counts are set to -1.
`aref`	*Warning_Skips*	A reference to an `array` of `int` values which are error indices into the list returned by get_error_messages. Each error indexed by a value in this array will not be reported when parsing the GTF file.
`fh`	*Tx*	A filehandle to which to write spliced transcripts when parsing the GTF file.

Boolean	*Fix_GTF*	A Boolean specifying whether or not to attempt to fix non-format errors in the GTF file, such as unannotated start or stop codons. Format errors are automatically fixed when possible.
Boolean	*Inframe_Stops*	A Boolean specifying whether or not to output the <transcript_id> of any transcript which contains in-frame stop codons prior to the annotated stop codon.
Boolean	*Modified*	A Boolean specifying if the lists of genes, transcripts, and coding exons need to be re-sorted.

Constructor

GTF new(href *info*);

The *info* hash contains the following fields all of which are optional:

gtf_filename	The filename for the GTF file to be loaded. If this field is not given an empty GTF object is returned and all other fields in *info* will have no effect.
seq_filename	A file containing the genomic sequence which this GTF file annotates. If given, each Feature object in the GTF will know the number of A, C, G, T, and N bases in its sequence. Also the GTF file is checked to ensure that the start and stop codons and splice sites have correct sequence. This field has no effect unless the *gtf_filename* field is given.
conseq_filename	A file containing the conservation sequence corresponding to the genomic sequence which this GTF file annotates. If given, each Feature object created will know the number of match, mismatch, and unaligned bases in its conservation sequence. This field has no effect unless the *gtf_filename* field is given.
tx_out_fh	A filehandle to which to write all spliced transcripts. This field has no effect unless the *gtf_filename* and *seq_filename* fields are given.
warning_fh	A filehandle to which all errors and warnings generated while parsing the GTF file are written. If *seq_filename* is given, errors and warnings found while checking the GTF file against the sequence are also written to *warning_fh*. This field has no effect unless the *gtf_filename* field is given.

The constructor parses the data in the *info* hash into global variable. If the *gtf_filename* field is given it loads the specified file with the _parse_gtf function, otherwise it returns an empty GTF object.

Accessor Functions

`aref` genes()
This function returns a reference to an `array` of `Gene` objects (*Genes*), corresponding to all genes in this GTF file, sorted in increasing order of their start value.

`aref` transcripts()
This function returns a reference to an `array` of `Transcript` objects (*Transcripts*), corresponding to all transcripts of all genes in this GTF file, sorted in increasing order of their start value.

`aref` cds()
This function returns a reference to an `array` of CDS type `Feature` objects (*CDS*), corresponding to all coding exons in all transcripts in all genes in this `GTF` object, sorted in increasing order of their start value.

`string` filename()
Returns the name of the GTF file that was loaded into this `GTF` object (*Filename*). If this `GTF` was not loaded from a file it will return an empty `string`. The value returned can be changed with the set_filename function.

`href` conservation_count()
Returns a reference to a `hash` containing fields "0", "1", and "2" (*Total_Seq*). Each field contains the number of times that symbol appeared in the conservation sequence of the genomic sequence which this `GTF` annotates. If the conservation sequence was not loaded when the object was created all counts are returned as -1.

`href` sequence_count()
Returns a reference to a `hash` containing fields "A", "C", "G", "T", and "N" (*Total_Conseq*). Each field contains the number of times that symbol appeared in the genomic sequence which this `GTF` annotates. If the sequence was not loaded when the object was created all counts are returned as -1.

`void` output_gtf_file(`fh` *filehandle*)
Writes the data stored in this `GTF` object to *filehandle* in GTF format. If *filehandle* is not given the data is written to standard out.

`void` output_gff_file(`fh` *filehandle*)
Writes the data stored in this `GTF` object to *filehandle* in GFF format. If *filehandle* is not given the data is written to standard out.

`aref` get_error_messages()
Returns a list of error messages used by the _parse_gtf function.

Modifier Functions

void add_gene(Gene *gene*)
This function inserts *gene* into the list of Gene objects stored by this GTF object (*Genes*).

void set_genes(aref *genes*)

genes	A reference to an array of Gene objects

This sets the list of Gene objects stored by this GTF object (*Genes*) to *genes*. All genes that were previously stored will be forgotten.

int remove_gene(string *gene_id*)
This function removes any gene whose <gene_id> is *gene_id* from the list of genes stored by this GTF (*Genes*). It returns the number of genes it removed, which should always be 0 or 1.

void set_filename(string *filename*)
This function sets the value returned by the filename function (*Filename*) to *filename*.

void offset(int *offset*)
This function changes the position of every feature of every transcript of every gene stored by this GTF object by adding *offset* to it.

void reverse_complement(int *length*)
This function takes the length of the sequence this GTF file contains annotation for and reverse complements everything in the file. The positive strand becomes the negative strand and all positions, *p*, become *length - p*. It also updates the counts of each base returned by the sequence_count function.

Internal Functions

void _parse_gtf(fh *filehandle*)
This parses the GTF file passed to the constructor, reports any errors or warnings to *filehandle* and creates all Gene, Transcript, and Feature objects needed to store the information in the file. The *filehandle* parameter is optional and if it is not given errors and warnings are not reported.

void _rev_comp(string *string*)

string	A string of A, C, G, T, and N characters

This function returns the reverse complement of *string*.

void _update()
This function re-sorts the lists of genes, transcripts, and features of this GTF object. It is called before returning any of the lists but does nothing unless the *Modified* bit has been set by the add_gene or set_genes functions. Only re-sorting the lists before

they are returned saves time over keeping the lists in the correct order at all times since the lists are stored as simple `array` objects and would otherwise need to be re-sorted each time a new item it inserted into them..

3.2.3 Gene Object

This object stores all data for a single gene. Each gene object stores the <gene_id>, <strand>, <source>, and <seqname> fields from the GTF specification. These data are stored in the `Gene` object because they are the same for all transcripts of a given gene. All data from other fields are stored in the `Transcript` or `Feature` object. Each gene also contains a list of all transcripts which it contains.

Global Variables

`string`	*Id*	The <gene_id> of this gene.
`string`	*Seqname*	The <seqname> of this gene.
`string`	*Source*	The <source> of this gene.
`string`	*Strand*	The <strand> of this gene.
`aref`	*Transcripts*	A reference to an `array` of `Transcript` objects, containing one object for each transcript of this gene.
`pvar`	*Tag*	The value stored by the set_tag function.

Constructor

`Gene` new(`string` *gene_id*, `string` *seqname*, `string` *source*, `string` *strand*)
The constructor arguments contain data which specify the field of the same name in the GTF specification. The *strand* value must be "+", "-", or ".". The other three can be any string which contains no whitespace or "#" characters. The constructor returns a `Gene` object containing no transcripts.

Accessor Functions

`string` id()
Returns the <gene_id> of this gene (*Id*).

`string` seqname()
Returns the <seqname> field for this gene (*Seqname*).

`string` source()
Returns the <source> field for this gene (*Source*).

`int` start()
Returns the lowest start value of any transcript of this gene.

`int` stop()
Returns the highest stop value of any transcript of this gene.

`float` length()
Returns the length from the 5' most coordinate of any of its transcripts to the 3' most coordinate of any of its transcripts.

`string` strand()
Returns the <strand> field of this gene (*Strand*).

`aref` transcripts()
Returns a reference to an `array` of `Transcript` objects (*Transcripts*) containing all transcripts of this gene.

`aref` cds()
Returns a reference to an `array` of all CDS type `Feature` objects from any transcript of this gene.

`void` output_gtf(`fh` *filehandle*)
Outputs each transcript of this gene in GTF format to *filehandle*.

`void` output_gff(`fh` *filehandle*)
Outputs each transcript of this gene in GFF format to *filehandle*.

`Boolean` equals(`Gene` *compare*)
Compares *compare* to this gene and returns true if *compare* and this gene have exactly the same transcripts as each other and returns false otherwise. Transcripts are compared using the equals function of the `Transcript` object. The <gene_id> and <transcript_id> fields are ignored in this comparison.

`float` gc_percentage()
Returns the average GC percentage of all transcripts of this gene as determined by the gc_percentage function of the `Transcript` object.

`float` match_percentage()
Returns the average match percentage of all transcripts of this gene gene as determined by the match_percentage function of the `Transcript` object.

`float` mismatch_percentage()
Returns the average mismatch percentage of all transcripts of this gene gene as determined by the mismatch_percentage function of the `Transcript` object.

`float` unaligned_percentage()
Returns the average unaligned percentage of all transcripts of this gene gene as determined by the unaligned_percentage function of the `Transcript` object.

`pvar` tag()
Returns the tag value (*Tag*) as set by the set_tag function. If no tag has been set, it returns Perl's undefined value.

Modifier Functions

`void` add_transcript(`Transcript` *transcript*)
Adds *transcript* to the list of transcripts of this gene (*Transcripts*). Also sets the *Gene* field of *transcript* to be this `Gene` object.

`void` set_seqname(`string` *seqname*)
Sets the <seqname> field for this gene (*Seqname*) to *seqname*.

`void` set_source(`string` *source*)
Sets the <source> field for this gene (*Source*) to *source*.

`void` offset(`int` *offset*)
Moves the position of each transcript of this gene by *offset* bases.

`void` reverse_complement(`int` *length*)
Reverse complements each transcript in this gene given that the sequence the gene is on has length *length*.

`void` set_tag(`pvar` *tag*)
Sets the tag value (*Tag*) to be returned by the tag function to *tag*.

3.2.4 Transcript Object

This object stores the data specific to a single transcript. The transcript must be part of a `Gene` to retrieve its <strand>, <source>, <seqname>, or <gene_id>. `Transcript` objects are not intended to be used alone and should always be part of a `Gene` object. The `Transcript` object stores the <transcript_id> field and, for each of the four GTF feature types (exon, CDS, start_codon, and stop_codon), an `array` of `Feature` objects, each of which contains all features of that type of a transcript.

Global Variables

string	*Id*	The <transcript_id> of this transcript.
aref	*Exons*	A reference to an array of exon type Feature objects containing all exons of this transcript.
aref	*CDS*	A reference to an array of CDS type Feature objects containing all coding exons of this transcript.
aref	*Introns*	A reference to an array of intron type Feature objects containing all introns of this transcript. The value of this variable is not calculated until the it is requested by calling the introns function.
aref	*Starts*	A reference to an array of start_codon type Feature objects containing the start codon of this transcript. The start codon is normally contained in a single Feature object but is potentially split into two or three Feature objects.
aref	*Stops*	A reference to an array of stop_codon type Feature objects containing the stop codon of this transcript. The stop codon is normally contained in a single Feature object but is potentially split into two or three Feature objects.
Gene	*Gene*	The Gene object to which this transcript belongs.
int	*Start*	The lowest start value of any feature of this transcript.
int	*Stop*	The highest stop value of any feature of this transcript.
int	*Coding_Start*	The lowest start value of any coding exon of this transcript.
int	*Coding_Stop*	The highest stop value of any coding exon of this transcript.
int	*Coding_Length*	The sum of the lengths of all coding exons of this transcript.
Boolean	*Modified*	A Boolean specifying if changes have been made to any of the feature lists indicating that they need to be re-sorted and that some variables (*Start*, *Stop*, *Coding_Start*, *Coding_Stop*, *Coding_Length*, and *Introns*) must be recalculated.
pvar	*Tag*	The value stored by the set_tag function.

Constructor

`Transcript` new(`string` *id*)

The constructor takes a single argument which is the <transcript_id> for this transcript. The <transcript_id> should be a string containing no whitespace, quote, or "#" characters, as stated in the GTF specification. It returns a `Transcript` object containing no features and belonging to no `Gene`.

Accessor Functions

`aref` exons()

Returns a reference to an `array` of all exon type `Feature` objects of this transcript sorted by increasing start value (*Exons*).

`aref` cds()

Returns a reference to an `array` of all CDS type `Feature` objects of this transcript sorted by increasing start value (*CDS*).

`Feature` initial_exon()

If this transcript has a start codon, this function returns the 5' most coding exon of this transcript, otherwise it returns 0.

`Feature` terminal_exon()

If this transcript has a stop codon, this function returns the 3' most coding exon of this transcript, otherwise it returns 0.

`aref` introns()

Returns a reference to an `array` of all intron type `Feature` objects sorted by increasing start value (*Introns*). Each feature corresponds to an intron of this transcript. Introns are not stored explicitly in the GTF file but the `Transcript` object calculates them and builds a `Feature` object for each one.

`aref` start_codons()

Returns a reference to an `array` of all start_codon type `Feature` objects of this transcript sorted by increasing start value (*Starts*).

`aref` stop_codons()

Returns a reference to an `array` of all stop_codon type `Feature` objects of this transcript sorted by increasing start value (*Stops*).

`string` id()

Returns the <transcript_id> of this transcript (*Id*).

string gene_id()
Returns the <gene_id> of the Gene object to which this transcript belongs. This value is retrieved from the Gene object to which this transcript belongs.

Gene gene()
Returns the Gene object to which this transcript belongs (*Gene*).

string seqname()
Returns the GTF <seqname> field of this transcript. This value is retrieved from the Gene object to which this transcript belongs.

string source()
Returns the GTF <source> field of this transcript. This value is retrieved from the Gene object to which this transcript belongs.

int start()
Returns the lowest start value of any feature of this transcript (*Start*).

int stop()
Returns the highest stop value of any feature of this transcript (*Stop*).

int length()
Returns the length of the transcript from the start to the stop.

int coding_start()
Returns the lowest start value of any coding exon of this transcript (*Coding_Start*).

int coding_stop()
Returns the highest stop value of any coding exon of this transcript (*Coding_Stop*).

int coding_length()
Returns the sum of the length of all coding exons of this transcript (*Coding_Length*).

string strand()
Returns the GTF <strand> field of this transcript. This value is retrieved from the Gene object to which this transcript belongs.

float score()
Returns the sum of the score of all non-intron features of this transcript. Since introns are not explicitly stored in GTF they have no score.

void output_gtf(fh *filehandle*)
Outputs all features of this transcript to *filehandle* in GTF format.

`void` output_gff(`fh` *filehandle*)
Outputs all features of this transcript to *filehandle* in GFF format.

`Boolean` equals(`Transcript` *compare*)
Compares *compare* to this transcript and returns true if this transcript and *compare* have exactly the same features and returns false otherwise. Features are compared using the equals function of the `Feature` object. The <transcript_id> and <gene_id> fields are ignored in this comparison.

`Transcript` copy()
Returns a new `Transcript` object, with this object's transcript id, containing copies of all `Feature` objects of this transcript. The copy is not associated with any `Gene` object and should be added to a `Gene` object before being used.

`float` gc_percentage()
Returns the GC percentage of the genomic sequence of all non-intron features of this transcript.

`float` match_percentage()
Returns the match percentage of the conservation sequence of all non-intron features of this transcript.

`float` mismatch_percentage()
Returns the mismatch percentage of the conservation sequence of all non-intron features of this transcript.

`float` unaligned_percentage()
Returns the unaligned percentage of the conservation sequence of all non-intron features of this transcript.

`pvar` tag()
Returns the tag value (*Tag*) as set using the set_tag function. If no tag has been set, it returns Perl's undefined value.

Modifier Functions

`void` add_feature(`Feature` *feature*)
Adds *feature* to the proper list (*Exons*, *CDS*, *Start_Codons*, or *Stop_Codons*) of `Feature` objects stored by this `Transcript`.

`int` remove_exon(`int` *position*)
This function removes all exon type `Feature` objects stored by this `Transcript` (*Exons*) whose start value is *position* and returns the number of features removed, which should always be 0 or 1.

`int` **remove_cds**(`int` *position*)

This function removes all CDS type `Feature` objects stored by this `Transcript` (*CDS*) whose start value is *position* and returns the number of features removed, which should always be 0 or 1.

`void` **offset**(`int` *offset*)

This function changes the position of every feature of this transcript by adding *offset* to it.

`void` **reverse_complement**(`int` *length*)

This function reverse complements the transcript given that the length of the sequence that the transcript is on is *length*. The positive strand becomes the negative strand and all positions, *p*, become *length* - *p*. It also updates the counts of each base stored in each `Feature` object of this transcript.

`void` **set_tag**(`pvar` *tag*)

Sets the value returned by the tag function (*Tag*) to *tag*.

Internal Functions

`void` **_update**()

This function sorts the lists of `Feature` objects, calculates the introns, start, stop, coding start, coding stop, coding length, all of which are stored in global variables. This is called before returning any list of `Feature` objects but only runs if some modifier function has set the *Modified* bit since the last time _update was run. Only running these calculations when they are needed saves time over recalculating them each time a new feature is added.

`aref` **_all_features**()

This function returns a reference to an `array` of all non-intron `Feature` objects of this transcript, sorted by increasing start coordinate.

`void` **_set_gene**(`Gene` *gene*)

This function sets *gene* as the `Gene` object associated with this transcript (*Gene*). This function is used by the add_transcript function of the `Gene` object.

3.2.5 Feature Object

This object stores the data specific to a single feature (line) of a GTF file. The <feature>, <start>, <end>, <score>, and <frame> fields are stored in this object. Each `Feature` should be part of a `Transcript`, which, in turn, should be part of a `Gene`. `Feature` objects are not meant to be used outside of this hierarchy. The <transcript_id> field can be retrieved from the `Transcript` which contains this feature and the <source>, <seqname>, <strand>, and <gene_id> can be retrieved from the `Gene` which contains the transcript which contains this feature.

Global Variables

`string`	*Type*	The <feature> field of this feature. This must be "exon", "CDS", "start_codon", "stop_codon", or "intron". This is the <feature> field from the GTF specification.
`Int`	*Start*	The <start> field of this feature.
`Int`	*End*	The <end> field of this feature.
`float`	*Score*	The <score> field of this feature.
`string`	*Frame*	The <frame> field of this feature.
`string`	*Subtype*	The subtype of this feature. This is only used for CDS type features and should be "Initial", "Internal", "Terminal", or "Single".
`Boolean`	*Seq*	A `Boolean` specifying if the counts of A, C, G, T, and N bases in the genomic sequence of this feature have been set.
`Boolean`	*Conseq*	A `Boolean` specifying if the counts of match, mismatch, and unaligned bases in the conservation sequence of this feature have been set.
`float`	*GC*	The percentage of bases in the genomic sequence of this feature which are G or C. This is calculated and stored the first time get_gc_percentage is called. A value of -1 indicates that the A, C, G, T, and N counts have not yet been set.
`float`	*Match*	The percentage of match bases in the conservation sequence of this feature. This is calculated and stored the first time get_match_percentage is called. A value of -1 indicates that the match, mismatch, and unaligned counts have not yet been set.

float	*Mismatch*	The percentage of mismatch bases in the conservation sequence of this feature. This is calculated and stored the first time get_mismatch_percentage is called. A value of -1 indicates that the match, mismatch, and unaligned counts have not yet been set.
float	*Unaligned*	The percentage of unaligned bases in the conservation sequence of this feature. This is calculated and stored the first time get_unaligned_percentage is called. A value of -1 indicates that the match, mismatch, and unaligned counts have not yet been set.
Int	*0*	The number of mismatch bases in this feature's conservation sequence. A value of -1 indicates that this count has not yet been set.
Int	*1*	The number of match bases in this feature's conservation sequence. A value of -1 indicates that this count has not yet been set.
Int	*2*	The number of unaligned bases in this feature's conservation sequence. A value of -1 indicates that this count has not yet been set.
Int	*A*	The number of A bases in this feature's genomic sequence. A value of -1 indicates that this count has not yet been set.
Int	*C*	The number of C bases in this feature's genomic sequence. A value of -1 indicates that this count has not yet been set.
Int	*G*	The number of G bases in this feature's genomic sequence. A value of -1 indicates that this count has not yet been set.
Int	*T*	The number of T bases in this feature's genomic sequence. A value of -1 indicates that this count has not yet been set.
Int	*N*	The number of N bases in this feature's genomic sequence. A value of -1 indicates that this count has not yet been set.
Pvar	*Tag*	The value stored by the set_tag function.

Constructor

Feature new(string *type*, int *start*, int *end*, float *score*, string *frame*)

The constructor takes five arguments which correspond to the field of the same name in the GTF specification. The *type* argument must be one of "exon", "CDS", "start_codon", "stop_codon", or "intron". The *start* and *end* arguments are positive integers corresponding to the boundaries of the feature and *start* must be the lower of the two. The *frame* value must be "0", "1", "2", or ".". The constructor returns a

Feature object that does not belong to any Transcript. The user should use the add_feature function of the Transcript object to add this feature to a transcript.

Accessor Functions

string type()
Returns the <feature> field for this object (*Type*).

string subtype()
This function is only valid for CDS type Feature objects. It returns either "Initial", "Internal", "Terminal", or "Single" depending on the subtype of the CDS (*Subtype*), which is determined by the transcript which contains this feature. If this is the only CDS in this transcript, then "Single" is returned. If the transcript has a start codon and multiple exons and this is the most 5' CDS, "Initial" is returned, and if the transcript has a stop codon and multiple exons and this is the most 3' CDS, "Terminal" is returned. In all other cases "Internal" is returned.

string transcript_id()
Returns the <transcript_id> of the transcript to which this feature belongs.

Transcript transcript()
Returns the Transcript object to which this feature belongs (*Transcript*).

string gene_id()
Returns the <gene_id> of the Gene object which contains this feature.

Gene gene()
Returns the Gene object which contains this feature.

string seqname()
Returns the <seqname> field for this feature. This value is retrieved from the gene which contains this feature.

string source()
Returns the <source> field for this feature. This value is retrieved from the gene which contains this feature.

int start()
Returns the <start> field of this feature (*Start*).

int stop()
Returns the <end> field of this feature (*End*).

`int` end()
Same as the stop function above (*End*).

`int` length()
Returns the length of this feature from <start> to <end>.

`float` score()
Returns the <score> field of this feature (*Score*).

`string` frame()
Returns the <frame> field of this feature (*Frame*).

`string` strand()
Returns the <strand> field of this feature. This value is retrieved from the gene which contains this feature.

`Boolean` equals(`Feature` *compare*)
Compares *compare* to this feature and returns true if this feature and *compare* have the same <start>, <stop>, <stand>, and <feature> fields and returns false otherwise.

`Feature` copy()
Returns a new `Feature` object with the same <start>, <end>, <score>, <frame>, <feature>, base counts and conservation counts as this object.

`void` output_gtf(`fh` *filehandle*)
Outputs a single line describing this feature in GTF format to *filehandle*.

`void` output_gff(`fh` *filehandle*)
Outputs a single line describing this feature in GFF format to *filehandle*.

`float` gc_percentage()
Returns the GC percentage of the genomic sequence of this feature (*GC*).

`float` match_percentage()
Returns the match percentage of the conservation sequence of this feature (*Match*).

`float` mismatch_percentage()
Returns the mismatch percentage of the conservation sequence of this feature (*Mismatch*).

`float` unaligned_percentage()
Returns the unaligned percentage of the conservation sequence of this feature (*Unaligned*).

int get_a_count()
Returns the number of A bases in the genomic sequence of this feature (*A*).

int get_c_count()
Returns the number of C bases in the genomic sequence of this feature (*C*).

int get_g_count()
Returns the number of G bases in the genomic sequence of this feature (*G*).

int get_t_count()
Returns the number of T bases in the genomic sequence of this feature (*T*).

int get_n_count()
Returns the number of N bases in the genomic sequence of this feature (*N*).

int get_match_count()
Returns the number of match bases in the conservation sequence of this feature (*1*).

int get_mismatch_count()
Returns the number of mismatch bases in the conservation sequence of this feature (*0*).

int get_unaligned_count()
Returns the number of unaligned bases in the conservation sequence of this feature (*2*).

pvar tag()
Returns the tag value (*Tag*) as set using the set_tag function. If no tag has been set, it returns Perl's undefined value.

Modifier Functions

void set_subtype(string *subtype*)
Sets the subtype of this feature (*Subtype*) to *subtype*.

void set_start(int *start*)
Sets the <start> of this feature (*Start*) to *start*.

void set_stop(int *stop*)
Sets the <end> of this feature (*Stop*) to *stop*.

void set_frame(string *frame*)
Sets the <frame> of this feature (*Frame*) to *frame*.

void set_bases(int *a_count*, int *c_count*, int *g_count*, int *t_count*, int *n_count*)
Sets the counts of each base in this feature's genomic sequence (*A*, *C*, *G*, *T*, *N*).

`void` set_conseq(`int` *match_count*, `int` *mismatch_count*, `int` *unaligned_count*)
Sets the counts of each conservation base in this feature's conservation sequence (*0*, *1*, *2*).

`void` offset(`int` *offset*)
This function adds *offset* to the <start> and <send> fields (*Start* and *End*) of this feature.

`void` reverse_complement(`int` *length*)
This function takes the length of the sequence that this feature is on and reverse complements the feature. The positive strand becomes the negative strand and all positions, *p*, become *length* - *p*. It also updates the counts of each base stored by this feautre.

`void` set_tag(`pvar` *tag*)
Sets the value to be returned by the tag function (*Tag*) to *tag*.

3.3 Eval.pm

The Eval.pm library provides a set of functions to compute statistics on a set of `GTF` objects or compare two or more sets of `GTF` objects to each other. There are six main functions: Evaluate, General Statistics, Graph, Filter, Overlap, and Distribution. Each is described below. Most use a common set of statistics which are also described below.

All main Eval functions take `GTF_set` objects as parameters. A `GTF_set` is a data structure used to store a GTF set. Each GTF file in the set is loaded into a `GTF` object and stored in an `array` in the same position as the file occurs in the GTF set. A reference to this `array` is called a `GTF_set` object. `GTF_set` objects may contain only a single `GTF` object.

Several of the functions of the Eval library take, as input, objects that can be any of the following: `Gene`, `Transcript`, or `Feature`. A parameter of this type will be called a `GTF_obj` object.

3.3.1 Definition of Statistics

Most top-level functions use the same set of statistics and the same subroutines for GTF comparison and analysis. This allows new statistics to be added by making only one change to the initialization of the data structures and adding code to the appropriate comparison function to compute the new statistic. Statistics are organized into three levels, which from most general to most specific are: Level, Type, and Stat. A detailed description of how the statistics are organized into Levels, Types, and Stats as well as a description of all individual Levels, Types, and Stats can be found in Chapter 2.

All statistics computed are stored in a stats_struct, which is a hash with a field for each Level, each of which points to a hash containing fields for each Type of that Level, which in turn point to hash objects with fields for each Stat for this Level, each of which contains the value for this Level/Type/Stat combination.

Top-level Statistics Functions

array get_level_list()
Returns an array of string values containing the name of each Level.

hash get_list_struct()
Returns a hash with a field for each Level, indexed by the Levels name, and a field "Level" which points to an array containing the name of each Levels as a string. Each Level's field points to a hash with two fields: "Type", which is a list of all Types of this Level, and "Stat" which is a list of all Stats for this Level. This function is used to get a suggested ordering for reporting the statistics from a stats_struct.

hash get_general_list_struct()
Returns a hash similar to that of get_list_struct, but the "Stat" fields contain only General Stats.

stats_struct get_stats_struct()
Returns a stats_struct with all Stats set to 0.

Gene Level Statistics Functions

array get_gene_type_list()
Returns an array of all *Gene* Level Types.

array get_gene_stat_list()
Returns an array of all *Gene* Level Stats.

array get_gene_general_stat_list()
Returns an array of all *Gene* Level General Stats.

array get_gene_stat_type_list()
Returns an array of all *Gene* Level Comparison Stat Types.

array get_gene_substat_list()
Returns an array of all *Gene* Level Substats.

hash _get_gene_type_hash()
Returns a hash containing a field for each *Gene* Level Type. Each field is initialized to 0.

hash _get_gene_stat_hash()
Returns a hash containing a field for each *Gene* Level Stat. Each field is initialized to 0.

hash _get_gene_type_struct()
Returns a hash containing a field for each *Gene* Level Type. Each field points to a hash containing the results of _get_gene_stat_hash().

Transcript Level Statistics Functions

array get_transcript_type_list()
Returns an array of all *Transcript* Level Types.

array get_transcript_stat_list()
Returns an array of all *Transcript* Level Stats.

array get_transcript_general_stat_list()
Returns an array of all *Transcript* Level General Stats.

array get_transcript_stat_type_list()
Returns an array of all *Transcript* Level Comparison Stat Types.

array get_transcript_substat_list()
Returns an array of all *Transcript* Level Substats.

hash _get_transcript_type_hash()
Returns a hash containing a field for each *Transcript* Level Type. Each field is initialized to 0.

hash _get_transcript_stat_hash()
Returns a hash containing a field for each *Transcript* Level Stat. Each field is initialized to 0.

hash _get_transcript_type_struct()
Returns a hash containing a field for each *Transcript* Level Type. Each field points to a hash containing the results of _get_transcript_stat_hash().

Exon Level Statistics Functions

`array` get_exon_type_list()
Returns an `array` of all *Exon* Level Types.

`array` get_exon_stat_list()
Returns an `array` of all *Exon* Level Stats.

`array` get_exon_general_stat_list()
Returns an `array` of all *Exon* Level General Stats.

`array` get_exon_stat_type_list()
Returns an `array` of all *Exon* Level Comparison Stat Types.

`array` get_exon_substat_list()
Returns an `array` of all *Exon* Level Substats.

`hash` _get_exon_type_hash()
Returns a `hash` containing a field for each *Exon* Level Type. Each field is initialized to 0.

`hash` _get_exon_stat_hash()
Returns a `hash` containing a field for each *Exon* Level Stat. Each field is initialized to 0.

`hash` _get_exon_type_struct()
Returns a `hash` containing a field for each *Exon* Level Type. Each field points to a `hash` containing the results of _get_exon_stat_hash().

Nuc Level Statistics Functions

`array` get_nuc_type_list()
Returns an `array` of all *Nuc* Level Types.

`array` get_nuc_stat_list()
Returns an `array` of all *Nuc* Level Stats.

`array` get_nuc_general_stat_list()
Returns an `array` of all *Nuc* Level General Stats.

`array` get_nuc_stat_type_list()
Returns an `array` of all *Nuc* Level Comparison Stat Types.

`array` get_nuc_substat_list()
Returns an `array` of all *Nuc* Level Substats.

`hash` _get_nuc_type_hash()
Returns a `hash` containing a field for each *Nuc* Level Type. Each field is initialized to 0.

`hash` _get_nuc_stat_hash()
Returns a `hash` containing a field for each *Nuc* Level Stat. Each field is initialized to 0.

`hash` _get_nuc_type_struct()
Returns a `hash` containing a field for each *Nuc* Level Type. Each field points to a hash containing the results of _get_nuc_stat_hash();

Signal Level Statistics Functions

`array` get_signal_type_list()
Returns an `array` of all *Signal* Level Types.

`array` get_signal_stat_list()
Returns an `array` of all *Signal* Level Stats.

`array` get_signal_general_stat_list()
Returns an `array` of all *Signal* Level General Stats.

`array` get_signal_stat_type_list()
Returns an `array` of all *Signal* Level Comparison Stat Types.

`array` get_signal_substat_list()
Returns an `array` of all *Signal* Level Substats.

`hash` _get_signal_type_hash()
Returns a `hash` containing a field for each *Signal* Level Type. Each field is initialized to 0.

`hash` _get_signal_stat_hash()
Returns a `hash` containing a field for each *Signal* Level Stat. Each field is initialized to 0.

`hash` _get_signal_type_struct()
Returns a `hash` containing a field for each *Signal* Level Type. Each field points to a `hash` containing the results of _get_signal_stat_hash().

3.3.2 Evaluate Functions

array evaluate(GTF_set *ann*, aref *preds*, Boolean *verbose*)

ann	A GTF_set object containing the annotation GTF set.
pred	An array of GTF_set objects containing prediction GTF sets.
verbose	Sets the verbose mode on or off. This is an optional argument with default value false.

Returns an array of stats_struct objects where the first item corresponds to the annotation set, *ann*, and the rest correspond to the prediction sets in *pred* and occur in the same order as in the input.

The Evaluation function is used to compare one or more prediction GTF_set objects to an annotation GTF_set object. The first GTF in the annotation set is compared to the first GTF in each prediction set, the second GTF in the annotation set is compared to the second GTF in each prediction set, and so on. Statistics are calculated for each comparison and totaled, for each prediction list, over each GTF in that list. Comparisons are done using the compare_gene_lists function, which, in turn, uses the _compare_object_list function. Annotation statistics are generated using the _get_stats function described in the section 3.3.3 below.

List Comparison Functions

void compare_gene_lists(aref *anns*, aref *preds*, stats_struct *data*)

anns	A reference to an array of annotation Gene objects. This array must be sorted by increasing start position.
preds	A reference to an array of prediction Gene objects. This array must be sorted by increasing start position.
data	The results of the comparison are returned in *data*, which should have all value initialized to zero before to being passed to this function.

This function compares the gene set *anns* to the gene set *preds* and increments the appropriate counts in *data* by calling the _compare_object_list function with the appropriate function pointers.

void compare_tx_lists(aref *anns*, aref *preds*, stats_struct *data*)

anns	A reference to an array of annotation Transcript objects. This array must be sorted by increasing start position.
preds	A reference to an array of prediction Transcript objects. This array must be sorted by increasing start position.
data	The results of the comparison are returned in *data*, which should have all value initialized to zero before to being passed to this function.

This function compares the transcript set *anns* to the transcript set *preds* and increments the appropriate counts in *data* by calling the _compare_object_list function with the appropriate function pointers.

`void` compare_exon_lists(`aref` *anns*, `aref` *preds*, `stats_struct` *data*)

anns	A reference to an `array` of annotation `Feature` objects. This `array` must be sorted by increasing start position.
preds	A reference to an `array` of prediction `Feature` objects. This `array` must be sorted by increasing start position.
data	The results of the comparison are returned in *data*, which should have all value initialized to zero before to being passed to this function.

This function compares the exon set *anns* to the exon set *preds* and increments the appropriate counts in *data* by calling the _compare_object_list function with the appropriate function pointers.

`void` _compare_object_lists(`aref` *anns*, `aref` *preds*, `stats_struct` *data*, `fref` *init_func*, `fref` *clear_func*, `fref` *compare_func*, `fref` *collect_func*, `fref` *ann_collect_func*)

anns	A reference to an `array` of annotation `GTF_obj` objects. This `array` must be sorted by increasing start position.
Preds	A reference to an `array` of prediction `GTF_obj` objects. This `array` must be sorted by increasing start position.
Data	A reference to a `stats_struct` in which the results of the comparison are stored.
init_func	A reference to a function which takes a single `GTF_obj`, from *anns* or *preds*, and initializes its tag.
clear_func	A reference to a function which takes a single `GTF_obj`, from *anns* or *preds*, and clears its tag (frees the memory).
compare_func	A reference to a function which takes a `GTF_obj` from *anns* and a `GTF_obj` from *preds* (in that order), compares the objects and stores the results of the comparison in the objects' tags.
collect_func	A reference to a function which takes a single `GTF_obj` from *preds* with a filled-in tag and a `stats_struct` and copies the information from the tag into the `stats_struct`.
ann_collect_func	A reference to a function which takes a single `GTF_obj` from *anns* with a filled-in tag and a `stats_struct` and copies the information from the tag into the `stats_struct`.

This is the main function used to compare sets of `GTF_obj` objects. It moves through the sorted list of prediction objects, comparing each to all annotation objects which overlap it. The first time it compares any object it initializes that objects tag value using the *init_func* parameter. Once the object will no longer be used in any comparison (for prediction objects this means it has moved on to comparisons involving the next prediction object and for annotation objects this means that the current prediction object begins after the annotation object ends) the data from the objects tag is used to increment the values in *data* using the *collect_func* and

ann_collect_func parameters, and tag value is cleared with the *clear_func*, freeing the memory.

The tag is used to temporarily store the data from comparisons involving the object which the tag is on. This is done because if a single object matches in some way to more than one other object it should only be counted as a single match, so the results of comparisons must be stored until all comparisons of this object are complete. For example if a single prediction object overlaps two annotation objects it should be counted as one prediction overlap, and two annotation overlaps. If the fact that the prediction object already overlapped an annotation is not stored, then each overlap will be counted and a potential exists to have a prediction overlap count that is greater than the total number of predictions.

A valid tag is one of the format returned by the appropriate get_exon_tag, get_transcript_tag, or get_gene_tag function for the Level of object it is on. A filled in tag is a valid tag with all values set to the appropriate value for the object which it is on.

Object Comparison Functions

`void` _compare_genes(`Gene` *a_gene*, `Gene` *p_gene*)

This function is used to compare two `Gene` objects. This compares all overlapping transcripts of the genes using the _compare_txs function. The results of the comparisons are stored in the objects' tags, so each of the objects' tags must have been properly initialized using the _init_gene_tag function prior to calling _compare_genes.

`void` _compare_txs(`Transcript` *a_tx*, `Transcript` *p_tx*)

This function is used to compare two `Transcript` objects. It compares all overlapping coding exons and all overlapping introns of the two `Transcript` objects using the _compare_features function. The results of the comparisons are stored in the objects' tags. The tag values for the `Gene` objects that *a_tx* and *p_tx* belong to are also updated. The tags of *p_tx* and *a_tx* as well as those of the `Gene` objects they belong to must have been initialized, using _init_tx_tag and _init_gene_tag respectively, prior to calling _compare_txs.

`void` _compare_features(`Feature` *a_feature*, `Feature` *p_feature*)

This function is used to compare two CDS or intron type `Feature` objects. The results of the comparison are stored in the objects' tags. The tags must have been initialized using the _init_exon_tag or _init_intron_tag function prior to calling _compare_features.

Initialization and Clean up Functions

hash _get_gene_tag()
Returns a tag data structure for a Gene object with all fields initialized to 0.

hash _get_tx_tag()
Returns a tag data structure for a Transcript object with all fields initialized to 0.

hash _get_exon_tag()
Returns a tag data structure for a CDS or intron type Feature object with all fields initialized to 0.

void _init_gene_tag(Gene *gene*)
Sets *gene*'s tag value to the hash returned by _get_gene_tag. Any *Gene* Level Type fields that this gene qualifies for are set. It also initializes the tag of all transcripts of this gene by calling the _init_tx_tag function on each of them.

void _init_tx_tag(Transcript *tx*)
Sets *tx*'s tag value to the hash returned by _get_tx_tag. Any *Transcript* Level Type fields that this transcript qualifies for are set. It also initializes the tag of all CDS or intron type Feature objects stored by this transcript by calling the _init_exon_tag or _init_introns_tag function on each of them.

void _init_exon_tag(Feature *cds*)
Sets *cds*'s tag value to the hash returned by _get_exon_tag. Any *Exon* Level Type field that this exon qualifies for are set.

void _init_intron_tag(Feature *intron*)
Sets *intron*'s tag value to the hash returned by _get_exon_tag. The *Intron* Type field is set, and all other *Exon* Level Types are not.

void _clear_gene_tag(Gene *gene*)
Clears *gene*'s tag field.

void _clear_tx_tag(Transcript *tx*)
Clears *tx*'s tag field.

void _clear_exon_tag(Feature *exon*)
Clears *exon*'s tag field.

void _clear_intron_tag(Feature *intron*)
Clears *intron*'s tag field.

Data Collection Functions

`void` _collect_gene_stats(`aref` *genes*, `stats_struct` *data*)

genes	A reference to an `array` of `Gene` objects with valid filled in tags.
data	A `stats_struct` in which to store a summary of the data in the tags of the objects in *genes*.

Increments the appropriate *Gene* Level prediction counts in *data* for each matched value in the tag of each `Gene` object in *gene*s.

`void` _collect_ann_gene_stats(`aref` *genes*, `stats_struct` *data*)

genes	A reference to an `array` of `Gene` objects with valid filled in tags.
data	A `stats_struct` in which to store a summary of the data in the tags of the objects in *genes*.

Increments the appropriate *Gene* Level annotation counts in *data* for each matched value in the tag of each `Gene` object in *gene*s.

`void` _collect_tx_stats(`aref` *txs*, `stats_struct` *data*)

txs	A reference to an `array` of `Transcript` objects with valid filled in tags.
data	A `stats_struct` in which to store a summary of the data in the tags of the objects in *genes*.

Increments the appropriate *Transcript* Level prediction counts in *data* for each matched value in the tag of each `Transcript` object in *tx*s.

`void` _collect_ann_tx_stats(`aref` *txs*, `stats_struct` *data*)

txs	A reference to an `array` of `Transcript` objects with valid filled in tags.
data	A `stats_struct` in which to store a summary of the data in the tags of the objects in *genes*.

Increments the appropriate *Transcript* Level annotation counts in *data* for each matched value in the tag of each `Transcript` object in *tx*s.

`void` _collect_exon_stats(`aref` *exons*, `stats_struct` *data*)

exons	A reference to an `array` of `Feature` objects with valid filled in tags.
data	A `stats_struct` in which to store a summary of the data in the tags of the objects in *exons*.

Increments the appropriate *Exon* Level prediction counts in *data* for each matched value in the tag of each `Feature` object in *exon*s.

`void` _collect_ann_exon_stats(`aref` *exons*, `stats_struct` *data*)

exons	A reference to an `array` of `Feature` objects with valid filled in tags.

data	A `stats_struct` in which to store a summary of the data in the tags of the objects in *exons*.

Increments the appropriate *Exon* Level annotation counts in *data* for each matched value in the tag of each `Feature` object in *exon*s.

Statistic Calculation Functions

The following functions are used to calculate and store all Stats whose value is determined completely by the value of other Stats, such as *Sensitivity*, *Specificity*, and *Average* Stats, each of which depend on a *Count* and the total number of objects. The Stats these functions calculate are stored in *data* and the Stats which they depend upon must have been filled in in *data* prior to calling these functions.

`void` _calculate_stats(`stats_struct` *data*)
This calculates the Stats which depend on other Stats at each Level by calling each of the following five functions.

`void` _calculate_gene_stats(`stats_struct` *data*)
This calculates the Stats at the *Gene* Level which depend on other Stats.

`void` _calculate_tx_stats(`stats_struct` *data*)
This calculates the Stats at the *Transcript* Level which depend on other Stats.

`void` _calculate_exon_stats(`stats_struct` *data*)
This calculates the Stats at the *Exon* Level which depend on other Stats.

`void` _calculate_nuc_stats(`stats_struct` *data*)
This calculates the Stats at the *Nuc* Level which depend on other Stats.

`void` _calculate_signal_stats(`stats_struct` *data*)
This calculates the Stats at the *Signal* Level which depend on other Stats.

3.3.3 General Statistics Functions

`stats_struct` get_statistics (`aref` *gtfs*, `Boolean` *verbose*)

gtfs	A reference to an `array` of `GTF_set` objects for which to gather general statistics.
verbose	Sets the verbose mode on or off. This is an optional argument with default value false.

This function returns a `stats_struct` with all General Stats filled in according to the data in *gtfs*. It works by simply making a gene set for each `GTF_set`, calling the _get_stats function on it, and using the _calculate_stats function to fill in General Stats which depend on other General Stats.

stats_struct **_get_stats**(aref *genes*)

genes	A reference to an array of gene sets, where a gene set is an array of Gene objects.

Returns a stats_struct with all General Stat counts set according to the data in *genes*. It works by calling the **_get_gene_list_stats** function on each array of Gene objects in *genes*.

void **_get_gene_list_stats**(aref *genes*, stats_struct *data*)

genes	A reference to an array of Gene objects.
data	A stats_struct in which the results of this function are returned.

Increments all General Stat counts in *data* for each gene in *genes*. It works by calling the **_get_gene_stats** function on each Gene in *genes*.

void **_get_gene_stats**(Gene *gene*, stats_struct *data*)

Increments all *Gene* Level General Stat counts in *data* for *gene*. The **_get_tx_stats** function is called on each transcript of this gene to collect their stats.

void **_get_tx_stats**(Transcript *tx*, stats_struct *data*)

Increments all *Transcript* Level and some *Signal* Level (start and stop codon) General Stat counts in *data* for *tx*. The **_get_exon_stats** function is called on each CDS feature and intron feature of this transcript to collect their stats.

void **_get_exon_stats**(Feature *cds*, stats_struct *data*)

Increments all *Exon* Level, all *Nuc* Level, and some *Signal* Level (splice site) General Stat counts in *data* according to *cds*.

3.3.4 Filter Funcitons

array **filter_predictions**(GTF_set *ann*, aref *preds*, filter *filter*, Boolean *verbose*)

ann	A reference to an annotation GTF_set object.
preds	A reference to an array of prediction GTF_set objects to filter.
filter	A filter object (see below).
verbose	A Boolean value to set the verbose mode on or off. This is an optional argument with default value false.

This function takes an annotation GTF set and list of prediction GTF sets, compares the prediction sets to the annotation set, and creates, for each prediction set, a new GTF set containing only objects from the prediction set which pass the filter in *filter*. The filtered GTF sets are built using the **_filter_gene_list** function. This function returns an array of GTF_set objects, each one corresponding to the prediction GTF set at the same position in *preds*. A filter object is a reference to an array of size three. The first position is one of the following: “Check”, “Not”, “And”, or “Or”. If the value at index 0 is “Check”, then the values at the next two positions correspond to the Level

and Type of this filter in that order. Only objects at this Level of this Type will pass this filter. The Level and Type values for a "Check" filter must come from the get_filter_types function. If the first position is "And" or "Or" then the next two positions contain `filter` objects which are joined with a logical and or logical or (set intersection or set union) when checking to see if an object passes this filter. If the first position is "Not" then the next position contains another `filter` object which an object must fail to pass this filter.

`hash` get_filter_types()

Returns a `hash` listing possible filters. This just returns the results of _get_filter_type_struct.

`hash` _get_filter_type_struct()

Returns a `hash` containing all possible filters in a similar format to the get_list_struct function, described in the 3.3.1 section above. It has a field, "Levels", which lists each Level at which filters can be applied. Each Level also has a field, indexed by the Level's name, which is a list of possible filter types for this Level. Possible filter types include all Types and Comparison Stat Types for this Level.

`void` _filter_gene_lists(`aref` *anns*, `aref` *preds*, `aref` *new_genes*, `filter` *filter*)

anns	A reference to an `array` of `Gene` objects used as annotation.
preds	A reference to an `array` of `Gene` objects to be filtered
new_genes	A reference to an `array` in which `Gene` objects from *preds* which pass *filter* will be placed.
filter	A `filter` object specifying how to filter the GTF sets.

This function moves though the annotation and prediction lists in the same way as the _compare_object_lists function, making comparisons between any overlapping annotation and prediction genes. Once all comparisons have been made for a given prediction gene (and stored in that object's tag) the _filter_gene function is used to see if all or any part of the prediction gene passes *filter* and should be added to the *new_genes* list.

`Gene` _filter_gene(`Gene` *gene*, `filter` *filter*)

This function returns a new `Gene` object containing the portion of this gene that passes *filter*. The gene, as well of all of its transcripts and all of their CDS and intron features, should have a valid, filled in tag of the form returned by get_gene_tag, _get_tx_tag, or _get_exon_tag before calling this function. The _check_filter function is used to check if *gene* passes *filter*. If the gene explicitly passes the filter then a copy of *gene* is returned. If not then _filter_tx is called on each transcript and any transcript which passes the filter get placed in a new `Gene` object and returned. If the gene explicitly fails the filter or all transcripts fail the filter completely (_filter_tx returns 0) then 0 is returned.

Transcript _filter_tx(Transcript *tx*, filter *filter*)

This function returns a new Transcript object containing any part of *tx* which passed *filter*. The transcript, as well as all of its CDS and intron features, should have a valid, filled in tag of the form returned by get_tx_tag or _get_exon_tag before being passed to this function. The _check_filter function is used to check if *tx* passes *filter*. If the transcript explicitly passes the filter then a copy of *tx* is returned. If not _filter_exon is called on each coding exon of the transcript and any that pass are placed in a new Transcript object which is returned. If the transcript explicitly fails the filter or no CDS features pass the filter, then 0 is returned.

Feature _filter_exon(Feature *exon*, filter *filter*)

This function returns a new Feature, which is a copy of *exon*, if *exon* passes the filter and 0 otherwise. *exon* should have a valid, filled in tag of the form returned by get_exon_tag before being passed to this function. The _check_filter function is used to check if *exon* passes *filter*.

int _check_filter(href *info*, string *level*, filter *filter*)

info	A reference to a hash containing a tag value of the form returned by get_gene_tag, get_tx_tag, or get_exon_tag.
level	A string specifying what Level of object *info* came from.
filter	A filter object to check *info* against.

Returns -1 if the tag failed the filter, 0 if the tag neither failed nor passed the filter (the filter, or at least some part of it, is applied to a lower Level object than this), and 1 if the tag passed the filter. This function works recursively to get the value for the whole filter in cases when “And”, “Or”, or “Not” is used. Although the values returned from a call to _check_filter will always be -1, 0, or 1, its internal calls of itself may also return 2. A value of 2 means that this filter is for a Level which should already have been checked (since filter_gene_lists moves from gene to transcript to exon when checking the filter) and the filter was not failed at that level.

3.3.5 Graph Functions

array make_graphs(GTF_set *ann*, aref *preds*, aref *graphs*, href *resolution*, Boolean *verbose*)

ann	A reference to the annotation GTF_set object.
preds	A reference to array of prediction GTF_set objects.
graphs	A reference to an array of graph types to be calculated. Graph types are specified by an X-Split and a Level. Each element of the array is a hash with two fields, “split” and “level”, containing string values for the X-Split and Level respectively.

resolution	A reference to a hash containing a key for each X-Split type used in *graphs*. Each key points to another hash which specifies the bins to use for graphs of this X-Split type. This hash can contain only one of two possible keys. The first possible key is "user" and should point to an array of bins. Each bin is a hash with a "start" and "stop" field each of which should contain a float value. The array of bins should be sorted by "start" field. Bins cannot overlap and there should be no gaps (regions covered by no bin) between the "start" of the first bin and the "stop" of the last bin. The second possible key is "uniform" and should point to a hash with "min", "max", "count", "size" keys. "min" and "max" are the minimum and maximum values of the X-Split type for any bin. Any data which fall outside this range is ignored. "count" is the number of bins and "size" is the size of each bin. All four do not need to be specified. If any are skipped they are inferred from the others or from the data as needed. If all fur values are specified but are inconsistent the "count" value is ignored. Only one of the top-level keys, "user" and "uniform", should be given. If both are given the "uniform" key is ignored. A reference to a hash of the type pointed to by the X-Split keys of *resolution* is called a resolution object.
verbose	A Boolean value to set the verbose mode on or off. This is an optional argument with default value 0.

For each X-Split/Level pair in *graphs*, each prediction set in *preds* is split into bins according to *resolution* and an each bin is compared to the annotation set using the appropriate one of the compare_gene_sets, compare_transcript_sets, or compare_exon_sets functions. The return value is an array with an index for each prediction set, each of which contains the data for each X-Split/Level combination that was passed to the function in the *graphs* parameter. The format is:

return_val[*pred*]{*split*}{*level*}[*bin*] = *data*

Where *pred* is an index into the *preds* array to specify from which prediction set this data was created. *split* and *level* are hash fields specifying the X-Split/Level of the graph whose data they contain. *bin* is an index into each bin created, and *data* is a hash containing three fields:

min	The lower-bound for this bin.
max	The upper-bound for this bin.
data	An Eval report for this bin versus the whole annotation.

array get_graph_x_types()

Returns an array of string values representing all possible X-Splits.

hash **get_graph_y_types**()

Returns a hash in the same format as the return value of the get_list_struct function which specifies all Stats which can be graphed.

array **get_graph_x_levels**()

Returns an array of string values representing all possible Levels at which the data can be split.

array **_split_preds_for_graph**(GTF_set *gtfs*, aref *bins*, string *split*, string *level*)

gtf	A GTF_set object to split into bins.
bins	A reference to an array of bin objects, each of which is a hash containing three fields "min", "max", and "data" which have types float, float and stats_struct respectively. The bins should be sorted from low to high.
split	A string specifying by which property the GTF set should be split.
level	A string specifying at which Level the GTF set should be split.

Returns an array of GTF_set objects, each corresponding to a bin in *bins*, containing the *level* Level objects from *gtfs* which belong in that bin. Bins in the array returned are in the same order as the bins in the *bins* parameter.

array **_get_graph_bins**(resolution *resolution*, string *split*)

Returns an array of bin objects as specified by *resolution*. Each bin object is a hash with two fields, "min" and "max", both float values, which specify the lower- and upper-bound of the bin respectively. The *split* parameter is used only for error reporting.

float **_get_x_val**(GTF_obj *obj*, string *type*)

Returns the *type* value of *obj*, where *type* is a valid X-Split type (i.e. if *type* is GC% and *obj* is an exon it returns the GC% of that exon).

hash **_get_graph_x_val_map**()

Returns a hash mapping each X-Split type to a function that takes a GTF_obj and returns the value of the object for this X-Split.

float **_get_gc_percent**(GTF_obj *obj*)

Returns *obj*'s GC percentage.

float **_get_match_percent**(GTF_obj *obj*)

Returns *obj*'s conservation match percentage.

float **_get_mismatch_percent**(GTF_obj *obj*)

Returns *obj*'s conservation mismatch percentage.

float _get_unaligned_percent(GTF_obj *obj*)
Returns *obj*'s conservation unaligned percentage.

int _get_length(GTF_obj *obj*)
Returns *obj*'s length

3.3.6 Overlap Functions

hash get_overlap_statistics(aref *preds*, string *type*, Boolean *verbose*)

preds	A reference to an array of GTF_set objects.
type	A string specifying the type of overlap clusters to build.
verbose	Sets the verbose mode on or off. This is an optional argument with default value false.

This function is used to build overlap clusters (see Chapter 2) from one of more sets of GTF objects. The type of overlap clusters to build is specified by *type*, which must be one of the string values in the array returned by the get_overlap_mode_list function. Clusters are built using the _get_overlap_stats function. The number of clusters of each cluster type as well as the number of objects from each input set in clusters of each cluster type are counted and returned. The results are returned as a hash in exactly the format described in the _get_overlap_stats function.

array get_overlap_labels(int *count*)
Returns an array containing the first *count* letters of the alphabet. This array can be used as a mapping between the GTF sets passed to get_overlap_statistics and their labels in the value returned by that function.

array get_overlap_mode_list()
Returns an array of string values, where each string is a valid overlap type for the argument *type* to get_overlap_statistics.

Specific Overlap Type Functions

Each of the following functions takes as input a reference to an array of GTF_set objects and computes overlap clusters from the GTF_set objects by calling _get_overlap_stats with arguments which cause it to build overlap clusters using a specific overlap type. The value returned is a hash containing counts of each cluster type and is described in detail in the _get_overlap_stats section. The overlap type used to build the clusters is described for each function. All types of overlap require that the two objects are on the same strand.

hash get_tx_exact_overlap_statistics(aref *preds*)
Builds overlap clusters of transcripts which are exactly the same (identical start_codon, stop_codon, and CDS features).

hash get_tx_80p_overlap_statistics(aref *preds*)
Builds overlap clusters of transcripts whose regions overlap by at least 80% of the length of the longer region.

hash get_tx_80p_small_overlap_statistics(aref *preds*)
Builds overlap clusters of transcripts whose regions overlap by at least 80% of the length of the shorter region.

hash get_tx_coding_overlap_statistics(aref *preds*)
Builds overlap clusters of transcripts whose coding regions (CDS features) overlap each other by at least one base pair.

hash get_tx_1bp_overlap_statistics(aref *preds*)
Builds overlap clusters of transcripts whose regions overlap by at least one base pair.

hash get_tx_exact_exon_overlap_statistics(aref *preds*)
Builds overlap clusters of transcripts which match at least one exon exactly.

hash get_tx_exact_intron_overlap_statistics(aref *preds*)
Builds overlap clusters of transcripts which match at least one intron exactly.

hash get_exon_exact_overlap_statistics(aref *preds*)
Builds overlap clusters of exons which are exactly the same (<start>, <end>, and <strand> are identical).

hash get_exon_80p_both_overlap_statistics(aref *preds*)
Builds overlap clusters of exons which overlap each other by at least 80% of the length of the longer exon.

hash get_exon_80p_smaller_overlap_statistics(aref *preds*)
Builds overlap clusters of exons which overlap each other by at least 80% of the length of the shorter exon.

hash get_exon_1bp_overlap_statistics(aref *preds*)
Builds overlap clusters of exons which overlap each other by at least one base pair.

Cluster Building Functions

hash _get_overlap_stats(aref *preds*, fref *select_func*, fref *compare_func*)

preds	A reference to an array of GTF_set objects.
select_func	A reference to a function which takes a single GTF objects and returns an array of objects out of which cluster will be built.
compare_func	A reference to a function which takes two objects of the type in the array returned by *select_func* and returns 1 if they belong in the same cluster and 0 otherwise.

The return value for this function is a `hash` containing a field for each cluster type which contains data about the clusters of that type. Each prediction set in *preds* is given a one character upper-case alphabetic label. Each cluster type is given a label which is simply the alphabetic order concatenation of the labels of all prediction sets whose objects this cluster type contains. In the return value each cluster type label points to a `hash` with a "total" field and fields for each GTF set in *preds*, indexed by each sets label. The "total" field contains the total number of clusters of this type and the GTF set label fields contain the number of objects in clusters of this type which come from the GTF set with that label. A reference to a `hash` of the format described for the return value of this function will be referred to as a `cluster_count` object.

A cluster is stored internally as a `hash` with three fields: "list", "start", and "stop". The "start" field holds the lowest coordinate of any object in the cluster. The "stop" fields hold the highest coordinate of any object in the cluster. The "list" field holds a reference to an `array` of objects (of the type returned by *select_func*) which are in this cluster. Each object in the "list" `array` should have its tag value set to the label of the GTF set to which it belongs. A reference to a `hash` of this type will be referred to as a `cluster` object.

`void` _collect_cluster(`cluster` *cluster*, `cluster_count` *cluster_count*)
This function increments the counts in *cluster_count* according to the objects in *cluster*.

`void` _combine_clusters(`cluster` *c1*, `cluster` *c2*)
Takes all the objects in the *c2* cluster and places them in *c1*. Also updates the "start" and "stop" fields of *c1* as necessary. The *c2* cluster is set to be an empty cluster.

`hash` _get_overlap_map()
Returns a `hash` which has a field for each valid overlap type each of which points to the function which will compute overlap clusters for that type.

`aref` _get_genes(`GTF` *pred*)
Returns a reference to an array of all `Gene` objects in *pred*.

`aref` _get_txs(`GTF` *pred*)
Returns a reference to an array of all `Transcript` objects in *pred*.

`aref` _get_exons(`GTF` *pred*)
Returns a reference to an array of all CDS type `Feature` objects in *pred*.

Overlap Test Functions

Each of the overlap test functions described below will return false anytime the two objects are not on the same strand.

`Boolean` _exact_bounds_overlap_func(`GTF_obj` *a*, `GTF_obj` *b*)
Returns true if the start and the stop of each object is the same and returns false otherwise.

`Boolean` _80p_both_overlap_func(`GTF_obj` *a*, `GTF_obj` *b*)
Returns true if the region from the start to the stop of each object overlaps by at least 80% of the length of the shorter object and returns false otherwise.

`Boolean` _80p_smaller_overlap_func(`GTF_obj` *a*, `GTF_obj` *b*)
Returns true if the region from the start to the stop of each object overlaps by at least 80% of the length of the longer of the two objects and returns false otherwise.

`Boolean` _1bp_overlap_func(`GTF_obj` *a*, `GTF_obj` *b*)
Returns true if the regions from the start to the stop of each object overlap by at least one base pair and returns false otherwise.

`Boolean` _tx_exact_overlap_func(`Transcript` *a*, `Transcript` *b*)
Returns true if the two transcripts are identical (start and stop codons and all coding exons are the same) and false otherwise.

`Boolean` _tx_coding_overlap_func(`Transcript` *a*, `Transcript` *b*)
Returns true if at least one CDS feature in *a* overlaps a CDS feature in *b* and returns false otherwise.

`Boolean` _tx_exact_exon_overlap_func(`Transcript` *a*, `Transcript` *b*)
Returns true if the two transcripts share at least one exon (same start and stop) and false otherwise.

`Boolean` _tx_exact_intron_overlap_func(`Transcript` *a*, `Transcript` *b*)
Returns true if the two transcripts share at least one intron (same start and stop) and false otherwise.

3.3.7 Distribution functions

`array` get_distribution(`aref` *gtfs*, `aref` *distributions*, `Boolean` *verbose*)

Gtfs	A reference to an `array` of `GTF_set` objects.
distributions	A reference to an `array` of `string` values, each of which is a distribution to calculate.
Verbose	A `Boolean` value to set the verbose mode on or off. This is an optional argument with default value 0.

This function computes each distribution in *distributions* on each GTF set in *gtfs* and returns an array containing a hash for each GTF set in *gtfs*. Each hash has a field for all distributions in *distributions*, each of which points to a hash containing fields for every value an object in this prediction set takes in this distribution. Each value field points to an integer which is the number of objects which have this value. For example for an *Exons_Per_Transcript* distribution the field "4" would contain the number of transcripts which have 4 exons. A hash is used because some distributions have rare outliers which are very large. In the case of a length distribution 99% of the data could have length less than 2000 but a small number could have length between 2000 and 40000. If an array were used to store this data 40000 bins would be needed to store all the data, but many of them would be empty (have value 0). By using a hash the memory needed to store as well as the time required to iterate through most distributions is decreased.

hash get_distribution_type_hash()

Returns a hash with a field for each distribution type returned by get_distribution_type_list. All fields are initialized to 0.

array get_distribution_type_list()

Returns an array of string values containing all valid distribution types.

void _get_distribution(aref *gtfs*, aref *data*, fref *type_func*, href *dist_funcs*)

gtfs	A reference to an array of GTF_set objects.
data	A reference to an array of the form returned by get_distribution described above.
type_func	A reference to a function which takes a single GTF object and returns an array of the objects from which distributions will be made.
dist_funcs	A reference to a hash of fref objects, each of which corresponds to a distribution to calculate and takes a single object of the type in the array returned by the *type_func* function and returns the value of that object for a this distribution. The functions are indexed in the hash by the distributions name.

This function calculates all distributions which are keys to *dist_funcs* for all GTF sets in *gtfs* and places the results in *data*. This function is used by the get_distribution function to calculate all distributions.

hash _get_distribution_functions()

Returns a mapping between the distribution type, as a string, and a function to get the appropriate value for this distribution from a GTF_obj object.

int _get_exons_per(Transcript *tx*)

Returns the number of coding exons *tx* contains.

int **_get_tx_length**(Transcript *tx*)
Returns the total length of *tx*.

int **_get_coding_length**(Transcript *tx*)
Returns the coding length of *tx*.

int **_get_exon_length**(Feature *exon*)
Returns the length of *exon*.

float **_get_exon_score**(Feature *exon*)
Returns the score of *exon*.

3.3.8 General Functions and Variables

Global Variables

alphabet	An `array` of letters of the alphabet. This is used to label overlap sets.

Functions

void **print_time**(int *total_time*)
Reports to standard error that the calculations have been completed in *total_time* seconds. Time is reported in days, hours, minutes, and seconds. This function is used by all top-level Eval functions when in verbose mode.

3.4 eval.pl

3.4.1 Overview

The GUI is organized around the six top-level function of the Eval package. Each function has a frame or set of frames which is used to specify the input to and display the output from the function. When the function uses a set of frames, each frame contains buttons which allow the user to move forward and backward through the set. When started, the GUI initializes each of these frame sets and displays the Evaluate functions frame. Across the top of the screen a bar of buttons allows the user to switch from one frame set to another. When one of these buttons is pressed the currently displayed frame set is replaced with the newly selected frame set. All actual computation is done in the functions of the Eval.pm library.

The GUI uses the Tk Perl module for creating windows and display widgets such as listboxes and buttons. This module was chosen for creating the GUI because it is easy to use and allows construction of complex graphical user interfaces. A basic understanding

of the Tk module is very helpful but not necessary for understanding or modifying the code.

Data Types

`listbox`	A Tk::Listbox object.
`frame`	A Tk::Frame object.
`ann_listbox`	A `listbox` object which contains the names of all currently loaded GTF sets. Each `ann_listbox` is automatically updated when a GTF set is loaded or unloaded. Only one GTF set may be selected at a time in a given `ann_listbox`.
`pred_listbox`	A `listbox` object which contains the names of all currently loaded GTF sets. Each `pred_listbox` is automatically updated when a GTF set is loaded or unloaded. Multiple GTF sets may be selected at once in a given `pred_listbox`.

3.4.2 Constants

`int`	*MIN_WIDTH*	The minimum width the main Eval window can have.
`int`	*MIN_HEIGHT*	The minimum height the main Eval window can have.
`int`	*EVAL_FRAME_NUM*	The value of *Current_Frame* when the Evaluate frame is displayed.
`int`	*STATS_FRAME_NUM*	The value of *Current_Frame* when the GenStats frame is displayed.
`int`	*FILTER_FRAME_NUM*	The value of *Current_Frame* when the Filter frame is displayed.
`int`	*GRAPH_FRAME_NUM*	The value of *Current_Frame* when the Graph frame is displayed.
`int`	*OVERLAP_FRAME_NUM*	The value of *Current_Frame* when the Overlap frame is displayed.
`int`	*DIST_FRAME_NUM*	The value of *Current_Frame* when the Dist frame is displayed.
`string`	*HOME*	The current user's home directory.
`string`	*ACTIVE_COLOR*	The foreground color for items which are enabled.
`string`	*INACTIVE_COLOR*	The foreground color for items which are disabled.
`array`	*GENERAL_SKIP*	A list of statistics that should not be displayed when showing only General Stats.

int	*MIN_DISP_LEN*	The minimum length in characters of a Stat name when it is output to a text file. If the statistic name is shorter than this value it will be padded with spaces to make it the correct length.
int	*MIN_VAL_LEN*	The minimum length in characters of a Stat value when output to a text file. If the value's length as a string is shorter than this value it will be padded with spaces to make it the correct length.
string	*USER*	The current user's name.
array	*ALPHABET*	An `array` of letters of the alphabet.

3.4.3 Global Variables

int	*X_Pos*	X-position of the main window.
int	*Y_Pos*	Y-position of the main window.
string	*Options_File*	Full path and filename of the user's options file.
array	*Main_Buttons*	An `array` of buttons (one for each top-level Eval function) which are used to select the frame to display.
array	*Main_Frames*	An `array` of the top-level frames (one for each top-level Eval function).
array	*Ann_GTF_Lbs*	An `array` of all `ann_listbox` objects.
array	*Pred_GTF_Lbs*	An `array` of all `pred_listbox` objects.
array	*GTF_Objs*	An `array` of `GTF_set` objects containing all GTF sets which are loaded in memory.
array	*Obj_Names*	An `array` of `string` values containing the names of the GTF sets which are loaded in memory.
int	*Current_Frame*	The index in *Main_Frames* of the currently displayed top-level frame.
Boolean	*Verbose*	`Boolean` specifying whether or not "Verbose" mode is turned on.
Boolean	*Really_Verbose*	`Boolean` specifying whether of not "Really Verbose" mode is turned on.
Boolean	*No_Seq*	A `Boolean` indicating whether sequence files (if available) should be loaded for each GTF. If true, the sequence files will not be loaded.
Boolean	*List_Mode*	A `Boolean` indicating whether inputs are expected to be list files or GTF files. A value of true means list files are expected.

string	*Precision*	A C style printf format string indicating the level of precision with which to report data (the number of decimal places to report).
string	*Cwd*	Current working directory. Default for loading and saving dialog boxes.
hash	*General*	A hash of every Stat and Type for each Level. Each field contains a Boolean which is true if the Type or Stat should be displayed when reporting General Statistics and false if it should not.
hash	*Display*	A hash of every Stat and Type for each Level. Each field contains is a Boolean which is true if the Type or Stat should be displayed when reporting statistics and false if it should not.
hash	*Graph_Resolution*	A hash to store the resolution for each graph split type.

3.4.4 Functions

Initialization Functions

void init_hashes()
Initializes the *General* and *Display* variables.

void initialize_frames()
Calls each top-level frame's initialize function.

void switch_frames(int *new_frame*)
Displays the top-level frame specified by *new_frame*, which is an index into the *Main_Frames* array.

General Functions

array make_ann_pred_frame(frame *frame*)
Creates a new frame inside *frame*, which contains an ann_listbox and a pred_listbox. This function is used by frame initialization functions to get listboxes which hold all currently loaded GTF sets. By using this function the code of each frame does not need to keep track of which GTF sets are currently loaded. The newly created frame is not displayed to the screen (using the pack, grid, or place function of the frame object). The return value is an array containing the new frame, the ann_listbox and the pred_listbox in that order.

array make_pred_frame(frame *frame*)

Similar to the make_ann_pred_frame function except the newly created frame contains only a pred_listbox. The return value is an array containing the new frame and the pred_listbox in that order.

void adjust_data_precision(aref *data*)

data	A reference to the output from Eval::evaluate or Eval::get_general_statistics

This function adjusts all Stat values in *data* to have the precision (number of decimal places) specified by *Precision*.

void message_func(string *msg*)

Displays a message box containing *msg*.

void error_func(string *error_msg*, int *fatal*)

Displays a message box containing *error_msg*. The second argument is optional and if it is given and is not 0 the program exits with status *fatal*.

string get_tmp_file()

Returns the name of a temporary file which should not be in use by any other process.

Menu Functions

void open_func()

Displays a Open File dialog box which allows the user to select a new GTF or list file to load into memory. Files are loaded with the load_func function.

void load_func(string *filename*)

Takes filename of a GTF or list file, loads the file into memory, and adds it to all annotation and prediction listboxes using the create_gtf_object and add_to_display functions.

GTF create_gtf_object(string *file*, string *seq*, string *conseq*)

Takes a GTF filename and optional sequence and conservation sequence filenames, loads the file as a GTF object and returns the GTF object. If either sequence is given they are loaded into the GTF object unless *No_Seq* is true.

void add_to_display(string *name*, GTF_set *gtf*)

gtf is added to all prediction and annotation lisboxes under the name *name*. *gtf* and *name* are also added to *GTF_Objs* and *Obj_Names* respectively.

void save_func()

Opens a window which allows the user to select a GTF set to save to disk.

void _save_pred_func(GTF_set *object*)

Takes a GTF_set and opens a Save File dialog which allows the user to save *object* as either a GTF file or list file depending on whether there are one or more than one GTF objects in *object*. When *object* contains more than one GTF object it is saved as a list file and all individual GTF objects in the list are saved as GTF files in the same directory with the name "*filename.#.gtf*" where "*filename*" is the name the list file was saved to and "#" is the position of each GTF in the list.

void remove_func()

Opens a window which allows the user to select GTF sets to unload.

void _remove_preds_func(pred_listbox *plb*)

Removes all GTF sets selected in *plb* from all ann_listbox and pred_listbox objects.

void help_func()

Display the eval help system.

void about_func()

Displays an about dialog box containing general information about eval.pl.

void exit_func()

Exits the program.

Options Functions

void load_options_file()

Loads the program options from the user's .evalrc file. Options include which Types and Stats to display for each Level and the graph resolutions to use for each X-Split.

void save_options_file()

Writes the currently loaded options into the user's .evalrc file

void edit_options_func()

Loads a window that allows the user to edit the options and save them to his .evalrc file.

Eval Frame Functions

void initialize_eval_frame()

Initializes the display widgets of the Evaluate frame.

void eval_run_func(ann_listbox *alb*, pred_listbox *plb*)

Runs Eval::evaulate using the GTF set selected in *alb* as the annotation set and the GTF sets selected in *plb* as the prediction sets, and displays the results in a new window on the screen using the display_eval_results function.

`void` display_eval_results(`aref` *data*, `aref` *names*)

data	A reference to an `array` of Eval reports, as returned by Eval::evaluate
names	A reference to an `array` of `string` values containing the names of the annotation and prediction GTF setsfrom which *data* was created. The annotation set name is listed first followed by the prediction set names in the same order as their reports appear in *data*.

Displays the reports in *data* in a new window.

`void` fill_general_stats_frame(`aref` *data*, `aref` *names*, `frame` *frame*)

data	A reference to an `array` of Eval reports, as returned by Eval::evaluate
names	A reference to an `array` of `string` values containing the names of the annotation and prediction GTF sets from which *data* was created. The annotation set name is listed first followed by the prediction set names in the same order as their reports appear in *data*.
frame	A `frame` object in which the General Stats of *data* should be displayed.

Displays the General Stats from *data* in *frame*.

`void` save_eval_output(`aref` *data*, `aref` *names*)

data	A reference to an `array` of Eval reports, as returned by Eval::evaluate
names	A reference to an `array` of `string` values containing the names of the annotation and prediction GTF sets from which *data* was created. The annotation set name is listed first followed by the prediction set names in the same order as their reports appear in *data*.

Opens a Save File dialog box which allows the user to save the Eval reports in *data* to a text file.

`string` pad_string(`string` *string*, `int` *min_len*)

Appends *string* with spaces until it has length *min_len*.

`string` get_general_stats_text(`aref` *data*, `aref` *names*)

data	A reference to an `array` of Eval reports, as returned by Eval::evaluate
names	A reference to an `array` of `string` values containing the names of the annotation and prediction GTF sets from which *data* was created. The annotation set name is listed first followed by the prediction set names in the same order as their reports appear in *data*.

Returns a `string`, containing all General Stats in *data*, to be written to a file.

GenStats Frame Functions

void initialize_stats_frame()
Initializes the display widgets of the GenStats Frame.

void get_stats_run_func(pred_listbox *plb*)
Runs Eval::get_statistics on the GTF sets selected in *plb* and displays the results in a new window using the display_stats_func function.

void display_stats_func(aref *data*, aref *names*)

data	A reference to an array of Eval reports, as returned by Eval::get_statistics
names	A reference to an array of string values containing the names of the annotation and prediction GTF sets from which *data* was created. The annotation set name is listed first followed by the prediction set names in the same order as their reports appear in *data*.

The reports are displayed in a new window using the fill_general_stats_frame function.

void save_stats_output(aref *data*, aref *names*)

data	A reference to an array of Eval reports, as returned by Eval::get_statistics
names	A reference to an array of string values containing the names of the annotation and prediction GTF sets from which *data* was created. The annotation set name is listed first followed by the prediction set names in the same order as their reports appear in *data*.

Opens a Save File dialog box which allows the user to save the Eval reports in *data* to a text file.

Filter Frame Functions

void initialize_filter_frame()
Initializes the display widgets of the Filter frame.

Boolean parse_filter_string(href *filter_keys*, string *filter_text*, aref *filter*)

filter_keys	A reference to a hash which maps the single character alphabetic labels used in *filter_string* to filter they represent.
filter_string	A Filter String as described in the section 2.3.6 above.
filter	An array reference which the filter object will be returned in.

The *filter_text* string is parsed, using the parse_filter_helper function, and placed a filter object, *filter*, to be passed to the Eval::filter_predictions function. If *filter_string* is successfully parsed into *filter*, the function returns 1, otherwise it returns 0.

Boolean parse_filter_helper(href *keys*, string *text*, filter *filter*)

keys	A reference to a hash which maps each single character alphabetic label used in *text* to the filter it represents.
text	A Filter String as described in the section 2.3.6 above.
filter	An array reference which the filter object will be returned in.

This function recursively calls itself to parse *text* into *filter* using *keys*. If *text* is not a valid Filter String the function returns 0. If *text* is successfully parsed into *filter*, the function returns 1.

void filter_run_func(ann_listbox *alb*, pred_listbox *plb*, filter *filter*, string *name*)

The prediction GTF sets selected by *plb* are filtered against the GTF set selected in *alb* according to *filter* using the Eval::filter_predictions function, and the newly created GTF sets and placed in all annotation and prediction listboxes under the name of the prediction GTF set they were created from but having *name* inserted into the title.

Graph Frame Functions

void initialize_graph_frame()

Initializes the display widgets of the Graph frame.

hash get_default_graph_resolution()

Returns the default graph resolution which is used when no resolution is specified in the user's .evalrc file or the user does not have a .evalrc file.

array graph_run_func(ann_listbox *alb*, pred_listbox *plb*, listbox *glb*, resolution *resolution*)

Items in *glb* should have the format "Level::X-Split" where "Level" is the Level at which the data should be split and "X-Split" is the property by which the data should be split. The items in *glb* are then converted into the format expected by the *graphs* parameter of Eval::make_graphs. The annotation GTF set selected in *alb* and all prediction GTF sets selected in *plb* along with the *graphs* parameter and the *resolution* argument to this function are passed to Eval::make_graphs and the results of that function are returned.

void save_graph_func(ann_listbox *alb*, pred_listbox *pslb*, listbox *glb*, aref *graphs*, string *level*, string *type*, string *stat*)

This function opens a Save File dialog box which allows the user to save a graph to a text file. The graph which is saved is specified by the inputs to this function. The three listbox arguments are the same as in the graph_run_func function above. The *graphs* argument is the output from Eval::make_graphs. The three string arguments specify the Level, Type, and Stat being graphed.

`void` display_graph_func(`ann_listbox` *alb*, `pred_listbox` *pslb*, `listbox` *glb*, `aref` *graphs*, `string` *level*, `string` *type*, `string` *stat*)

Takes the same inputs as the save_graph_func function but displays the graph in a new window using the gnuplot_bar_bin function instead of saving it to a file.

`void` gnuplot_bar_bin(`aref` *graph*, `string` *title*, `string` *x_label*, `string` *y_label*)

graph	A reference to an `array` of `hash` objects. Each `hash` represents a single bar to be graphed and contains three fields: "min", "max", and "count". "min" and "max" are the lower- and upper-x-axis bounds of the bar and "count" is the height of the bar on the y-axis.
title	The title of the graph.
x_label	The label for the x-axis of the graph.
y_label	The label for the y-axis of the graph.

This function displays the graph specified by *graph* with the labels specified by the `string` arguments in a new window using gnuplot.

Overlap Statistics Frame Functions

`void` initialize_overlap_frame()

Initializes the display widgets of the Overlap frame.

`void` overlap_stats_run_func(`pred_listbox` *plb*, `listbox` *slb*)

The items in the *slb* `listbox` are `string` values which represent overlap types and exactly one of which should be selected. Using the overlap type in selected *slb* and all prediction GTF sets selected in *plb* the overlap statistics are calculated using the Eval::get_overlap_statistics function. The results are displayed in a new window using the display_overlap_stats_func function.

`void` display_overlap_stats_func(`href` *data*, `aref` *pred_names*, `string` *overlap_type*)

data	A reference to the output from Eval::get_overlap_statistics function.
pred_names	A reference to an `array` of `string` values containing the names of the prediction GTF sets from which *data* was created.
overlap_type	The type of overlap used to create *data*.

This function displays the overlap statistics in *data* in a new window.

`void` save_overlap_stats_output(`href` *data*, `aref` *pred_names*, `string` *overlap_type*)

data	A reference to the output from Eval::get_overlap_statistics function.

pred_names	A reference to an array of string values containing the names of the prediction GTF sets from which *data* was created.
overlap_type	The type of overlap used to create *data*.

This function opens a Save File dialog box which allows the user to save the overlap statistics in *data* to a text file.

string get_overlap_stats_text(href *data*, aref *pred_names*, string *overlap_type*)

data	A reference to the output from Eval::get_overlap_statistics function.
pred_names	A reference to an array of string values containing the names of the prediction GTF sets from which *data* was created.
overlap_type	The type of overlap used to create *data*.

This function returns a text version of the overlap statistics in *data*.

Dist Frame Functions

void initialize_distribution_frame()

Initializes the display widgets of the Distribution frame.

void save_distribution_data(string *gtf_name*, string *dist_name*, aref *data*)

gtf_name	The name of the GTF set from which this distribution was made.
dist_name	The type of distribution used to make data.
data	The output of bin_distribution_data.

This function opens a Save File dialog box and allows the user to save the distribution in *data* to a text file.

void graph_distribution_data(string *gtf_name*, string *dist_name*, aref *data*)

gtf_name	The name of the GTF set from which this distribution was made.
dist_name	The type of distribution used to make data.
data	The output of bin_distribution_data.

This function displays the distribution in *data* in a new window using the gnuplot_bar_bin function.

aref bin_distribution_data(aref *data*, float *max*, float *res*, Boolean *cum*)

data	A reference to the output from Eval::get_distribution.
max	The maximum upper bound for any bin.
res	The size of the bins that data should be placed in.
cum	If true then the distributions made should be cumulative. In a cumulative distribution the returned value for any bin is the number of objects which fall in it plus the sum of the number of objects in all bins below it.

This function takes the distribution in *data* and moves it into bins of size *res*, going from 0 to *max*. All data in *data* which would fall into bins above *max* are placed into a single bin that covers everything from *max* to infinity. The bins are of the form expected by the gnuplot_bar_bin function. A reference to a `array` of the bins, sorted from low to high x-axis position, is returned.

3.5 Eval Scripts

The following scripts are command line interfaces to the functions of the Eval.pm library. Most functions from the Eval scripts are not listed below. If a function is not listed it is identical to the function of the same name in the section 3.4 above.

3.5.1 evaluate_gtf.pl

`string` print_eval_output()

This function is the same as the save_eval_output function of eval.pl but it returns the text generated instead of opening a Save File dialog box and saving the text to a file.

3.5.2 get_general_stats.pl

<no new functions>

3.5.3 filter_gtfs.pl

`void` print_filter(`filter` *filter*)

Takes a `filter` object and displays it as a text string to standard out. This function is used for debugging purposes.

`void` print_filter_types()

Prints all valid filters to standard out.

`void` check_filter(`string` *level*, `string` *type*)

Takes two `string` values containing the Level and Type of a filter and exits with an error if they do not specify a valid filter.

`void` error_func(`string` *message*)

Writes *message* to standard error and exits.

`array` load_func(`string` *filename*)

Opens *filename*, which should be a GTF or list file and loads the file into a GTF_Set object which it then returns.

3.5.4 graph_gtfs.pl

`void` parse_resolution_text(`href` *res*, `array` *text*)
Each value in *text* is a line of text from a .evalrc file specifying the graph resolutions. The resolutions are parsed out of the text and placed in *res*, which is used as the *resolution* argument to the Eval::make_graphs function.

`void` print_graph_types()
Prints all valid X-Splits and Level/Type/Stat combinations for building Eval graphs to standard out.

`array` load_func(`string` *filename*)
Same as load_func in filter_gtfs.pl

3.5.5 get_overlap_stats.pl

<no new functions>

3.5.6 get_distribution.pl

`array` load_func(`string` *filename*)
Same as load_func()in filter_gtfs.pl

Chapter 4: Future Work

The Eval project is, for the most, part complete. Several small upgrades to the main functions as well as small reorganization of the code are planned.

The most general way to extend the Eval package is the addition of new Stats. If a situation arises where a new Stat is required it can easily be added to the existing code, as can a new Type. Adding a new Level would require significantly more work but could be done. The need for a new Level is not anticipated because the current set of Levels covers all types of information which can be stored in a GTF file.

The GTF validator currently outputs a list of all errors and warnings encountered when loading a GTF file. No distinction is made between an error and a warning so the user must decide the severity of the problem and whether the problem must be fixed or is acceptable to leave as it is. This should be changed so that a distinction between errors and warnings is made. Errors should be problems which must be fixed before the file can be used because they could never occur in real genes, such as in-frame stop codons. Warnings should be messages that alert the user to a situation which is abnormal but may still be correct, such as non-standard splice sites.

The graph function resolution should be extended to be more versatile. Currently there is no way to center the graph on the data. The resolution should be able to be generically specified around the minimum and maximum values in the data. The lack of this feature is a problem because graphs, even at a fixed X-Split, may have widely differing ranges of values on the x-axis. For example, an exon length graph will cover a much small and much lower range of x-axis values than a transcript length graph. If the resolution could be specified to have some specific number of bins and go from the minimum value to the maximum value, varying ranges would not be a problem.

Some of the code in eval.pl is repeated in the Eval scripts. This code belongs in some library, so that any changes that need to be made do not require altering the code in multiple places. The Eval.pm library contains code for comparing sets of GTF files so it is not the appropriate place for the repeated code, which is user interface functions, such as loading GTF files, parsing command line arguments, and displaying Eval function results in different ways. A new user interface library should be created to contain this code.

Appendix A: GTF File Format Specification

GTF stands for gene transfer format. This borrows from the GFF file format [1], but has additional structure that warrants a separate definition and format name.

The structure is similar to GFF, so the fields are:

<seqname><source><feature><start><end><score><strand><frame><attributes>

Here is a simple example with 3 translated exons. The order of the rows is not important.

```
Hs-Ch1 Twinscan CDS         380 401 . + 0 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan CDS         501 650 . + 2 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan CDS         700 707 . + 2 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan start_codon 380 382 . + 0 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan stop_codon  708 710 . + 0 gene_id "1"; transcript_id "1.a";
```

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

<seqname>
The <seqname> field contains the name of the sequence which this gene is on.

<source>
The <source> field 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 <feature> field can take four values: "CDS", "start_codon", "stop_codon", and "exon". The “CDS” feature 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” feature for the terminal exon. The “exon” feature is used to annotate all exons, including non-coding exons. The “start_codon” and “stop_codon” features should have a total length of three for any transcript but may be split onto more than one line in the rare case where an intron falls inside the codon.

<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> must fall inside the sequence on which this feature resides.

<score>
The <score> field is used to store some score for the feature. This can be any numerical value, or can be left out and replaced with a period.

<frame>
A value of 0 indicates that the first whole codon of the reading frame is located at 5'-most base. 1 means that there is one extra base before the first whole codon and 2 means that there are two extra bases before the first whole codon. Note that the frame is not the length of the CDS mod 3. 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.

<attributes>
Each attribute in the <attribute> field should have the form:
attribute_name “attribute_value”;

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). Attributes' values should be surrounded by double quotes.

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

gene_id	A unique identifier for the genomic source of the transcript. Used to group transcripts into genes.
transcript_id	A unique identifier for the predicted transcript. Used to group features into transcripts.

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

[comments]
Any line may contain comments. Comments are indicated by the # character and everything following a # character on any line is a comment. As such, all fields are prohibited from containing # characters.

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

```
Hs-Ch1 Twinscan CDS         193817 194022 . - 2 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan CDS         199645 199753 . - 2 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan CDS         200369 200507 . - 1 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan CDS         215991 216028 . - 0 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan start_codon 216026 216028 . - . gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan stop_codon  193814 193816 . - . gene_id "1"; transcript_id "1.a";
```

Note the frames of the coding exons. For example:
The first CDS (from 216028 to 215991) always has frame zero.
The frame of the first CDS is 0 and it has length 38. (38 - 0) % 3 = 2, so the frame of the second CDS is 1 (the first two bases of the codon are on the previous exon leaving one base at the start of this exon).

The frame of the second CDS is 1 and it has length 139. (139 - 1) % 3 = 0, so the frame of the third CDS is 0.
The frame of the third CDS is 0 and it has length 109. (109 - 0) % 3 = 1, so the frame of the terminal CDS is 2 (the first base of the codon is on the previous exon leaving two bases at the start of this exon).
Alternatively, the frame of terminal CDS can be calculated without the rest of the gene. The length of the terminal CDS is 206. 206 % 3 = 2, which is the frame of the terminal CDS.

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

```
Hs-Ch1 Twinscan exon        150 200 . + . gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan exon        300 401 . + . gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan CDS         380 401 . + 0 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan exon        501 650 . + . gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan CDS         501 650 . + 2 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan exon        700 800 . + . gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan CDS         700 707 . + 2 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan exon        900 997 . + . gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan start_codon 380 382 . + 0 gene_id "1"; transcript_id "1.a";
Hs-Ch1 Twinscan stop_codon  708 710 . + 0 gene_id "1"; transcript_id "1.a";
```

Appendix B: Fasta File Format

The fasta file format is a simple format for storing one or more genomic or protein sequences. Each sequence is preceded by a header line which must begin with “>”. The first word following “>” is the name of the sequence and all text following the name is a description of the sequence. All lines following the header line contain the actual sequence. All forms of whitespace in the sequence are ignored and there is no limit to the number of characters allowed on a line. Any non-nucleic or non-amino acid characters in the sequence of the file are illegal.

Fasta files with multiple sequences have multiple header lines each followed by one or more lines containing the sequence associated with that header. Here is an example of a multi sequence fasta file:

```
> seq_1 This is example sequence 1
TTCATTGTGTATTTTATCACACAAATAAGGCACAGATTTTTAAAAAATCA
TCAACTTCCTGGCTACCTATATAGACATAATTACATAGAAGCTCAACTAA
ATTTGCAAACATTCCAGAGTTTGGGTTTCCAATAATTCTTTGTGATTCTT
TAAAAGGTAAAGTATTTTTTTCCCATAAAACATAGCAACATTTAAAATCA
CCCGTAGAATGTCCTGCCATTTTTGTTTCTGTAGTTTCCTCATTTTCTGC
AAAGCCTCGCTGAGGAAATTGACTTTGAATATCCTTT
> seq_2 This is example sequence 2
TTTAGAAAGCATTGTGGTAAAACATTGAATCATCATGGTCATAAGTTCTG
TTCACATTCTTTCTTGCTTTGAATATTTTTTCCCAGTGGCCAATATTTGA
TTCTGTTGTATCATGGCTAAAAGGTAGGCATGGCAACAAAATAAAG
> seq_3 This is example sequence 3
GAAGTCTTTGGAATAAGTGATCCCATCACAATGAATCAATTTGCCATTGG
AACATATTTTTACAAAGTCACTCTTTTGAAAATATTTAGCTATGAATTAA
AACAGAGTCTGTATGGTTAATATTTTTCCTGGTCTAAGGTGAACAGCATT
TTAGAGAATGAACTCAGGACACAACCACAGCAGAAGAAAAACGTGATAAT
TAAGTTTACACATGTGTGTTACTACTGCAACAGAAAACATG
```

All fasta files used by the Eval package should contain a single genomic sequence.

Appendix C: Conservation File Format

The conservation file is used by TWINSCAN to store information about the similarities between some target sequence and some informant database of sequences. It is generated by running BLAST [3] to compare the target sequence against the informant database. All high scoring BLAST hits are incorporated into the conservation sequence in such a way that for each base in the target sequence, the conservation sequence states whether that base is matched, mismatched, or unaligned. If the base is matched, then the base is covered by some high scoring BLAST hit and is aligned to a match (i.e. A to A). If the base is mismatched, then the base is covered by a high scoring BLAST hit but is either aligned to a mismatch (i.e. A to T) or aligned to a gap. If the base is unaligned, then no high scoring BLAST hit covers it.

The conservation file format is very simple. Unlike fasta format, the sequence has no header and only a single sequence is allowed per file. The file contains a single line, which holds a string of "0", "1", and "2" characters. A "0" means that this base was mismatched, a "1" means the base is matched, and a "2" means the base was unaligned.

Appendix D: Example Eval Report

```
**Summary Stats**
Annotation:  refseq.list
Predictions:               twinscan.list

Gene Sensitivity           14.35%
Gene Specificity           6.55%
Transcript Sensitivity     12.83%
Transcript Specificity     6.55%
Exon Sensitivity           71.89%
Exon Specificity           38.58%
Nucleotide Sensitivity     83.50%
Nucleotide Specificity     41.95%

**General Stats**
Predictions:
                                         refseq.list  twinscan.list
Gene
      All
            Count                        11930.00     26119.00
            Total Transcripts            13812.00     26119.00
            Transcripts Per              1.16         1.00
Transcript
      All
            Count                        13812.00     26119.00
            Average Length               46856.60     24042.48
            Total Length                 647183295.00 627965594.00
            Average Coding Length        1491.43      1357.62
            Total Coding Length          20599564.00  35459553.00
            Average Score                0.00         215.19
            Total Score                  0.00         5620498.75
            Exons Per                    9.23         7.84
            Total Exons                  127428.00    204729.00
      Complete
            Count                        12476.00     24918.00
            Average Length               46460.42     23712.27
            Total Length                 579640144.00 590862393.00
            Average Coding Length        1480.96      1361.40
            Total Coding Length          18476415.00  33923293.00
            Average Score                0.00         217.02
            Total Score                  0.00         5407714.92
            Exons Per                    9.33         7.82
            Total Exons                  116452.00    194895.00
      Incomplete
            Count                        1336.00      1201.00
            Average Length               50556.25     30893.59
            Total Length                 67543151.00  37103201.00
            Average Coding Length        1589.18      1279.15
            Total Coding Length          2123149.00   1536260.00
            Average Score                0.00         177.17
            Total Score                  0.00         212783.83
            Exons Per                    8.22         8.19
            Total Exons                  10976.00     9834.00
Exon
      All
            Count                        109883.00    204729.00
            Average Length               164.74       173.20
            Total Length                 18102268.00  35459553.00
            Average Score                0.00         21.69
```

```
            Total Score               0.00          4440400.20
      Initial
            Count                     10164.00      22002.00
            Average Length            180.63        168.70
            Total Length              1835955.00    3711673.00
            Average Score             0.00          18.96
            Total Score               0.00          417156.27
      Internal
            Count                     87588.00      157247.00
            Average Length            143.53        154.42
            Total Length              12571788.00   24281817.00
            Average Score             0.00          22.12
            Total Score               0.00          3478941.25
      Terminal
            Count                     10670.00      21745.00
            Average Length            224.94        237.55
            Total Length              2400130.00    5165434.00
            Average Score             0.00          14.73
            Total Score               0.00          320222.35
      Single
            Count                     1488.00       3735.00
            Average Length            872.64        615.96
            Total Length              1298481.00    2300629.00
            Average Score             0.00          59.99
            Total Score               0.00          224080.33
      Intron
            Count                     97658.00      178610.00
            Average Length            5541.95       3316.52
            Total Length              541215329.00  592363950.00
            Average Score             0.00          0.00
            Total Score               0.00          0.00
Nuc
      All
            Count                     18102268.00   35459553.00
      Initial
            Count                     1835955.00    3711673.00
      Internal
            Count                     12571788.00   24281817.00
      Terminal
            Count                     2400130.00    5165434.00
      Single
            Count                     1298481.00    2300629.00
      Intron
            Count                     541215329.00  592363950.00
Signal
      Splice Acceptor
            Count                     98258.00      178992.00
      Splice Donor
            Count                     97752.00      179249.00
      Start Codon
            Count                     12656.00      25700.00
      Stop Codon
            Count                     13074.00      25335.00


**Detailed Stats**
Annotation:  refseq.list
Predictions:                        twinscan.list
Gene
      All
            Count                     26119.00
            Ann Count                 11930.00
            Total Transcripts         26119.00
```

```
            Transcripts Per             1.00
            Correct Count               1712.00
            Correct Matched             1712.00
            Correct Specificity         6.55%
            Correct Sensitivity         14.35%
            Exact Count                 1646.00
            Exact Matched               1646.00
            Exact Specificity           6.30%
            Exact Sensitivity           13.80%
            Overlap Count               12330.00
            Overlap Matched             10835.00
            Overlap Specificity         47.21%
            Overlap Sensitivity         90.82%
            Nuc Overlap Count           12172.00
            Nuc Overlap Matched         10775.00
            Nuc Overlap Specificity     46.60%
            Nuc Overlap Sensitivity     90.32%
            All Introns Count           1737.00
            All Introns Matched         1738.00
            All Introns Specificity     6.65%
            All Introns Sensitivity     14.57%
            All Exons Count             1712.00
            All Exons Matched           1712.00
            All Exons Specificity       6.55%
            All Exons Sensitivity       14.35%
            Exact Intron Count          9816.00
            Exact Intron Matched        9310.00
            Exact Intron Specificity    37.58%
            Exact Intron Sensitivity    78.04%
            Exact Exon Count            10789.00
            Exact Exon Matched          9797.00
            Exact Exon Specificity      41.31%
            Exact Exon Sensitivity      82.12%
            Start Codon Count           4636.00
            Start Codon Matched         4632.00
            Start Codon Specificity     17.75%
            Start Codon Sensitivity     38.83%
            Stop Codon Count            6930.00
            Stop Codon Matched          6920.00
            Stop Codon Specificity      26.53%
            Stop Codon Sensitivity      58.01%
            Start Stop Count            2793.00
            Start Stop Matched          2793.00
            Start Stop Specificity      10.69%
            Start Stop Sensitivity      23.41%
Transcript
      All
            Count                       26119.00
            Ann Count                   13812.00
            Average Length              24042.48
            Total Length                627965594.00
            Average Coding Length       1357.62
            Total Coding Length         35459553.00
            Average Score               215.19
            Total Score                 5620498.75
            Exons Per                   7.84
            Total Exons                 204729.00
            Correct Count               1712.00
            Correct Matched             1772.00
            Correct Specificity         6.55%
            Correct Sensitivity         12.83%
            Exact Count                 1646.00
            Exact Matched               1704.00
```

```
        Exact Specificity         6.30%
        Exact Sensitivity         12.34%
        Overlap Count             12330.00
        Overlap Matched           12599.00
        Overlap Specificity       47.21%
        Overlap Sensitivity       91.22%
        Nuc Overlap Count         12172.00
        Nuc Overlap Matched       12517.00
        Nuc Overlap Specificity   46.60%
        Nuc Overlap Sensitivity   90.62%
        All Introns Count         1737.00
        All Introns Matched       1791.00
        All Introns Specificity   6.65%
        All Introns Sensitivity   12.97%
        All Exons Count           1712.00
        All Exons Matched         1772.00
        All Exons Specificity     6.55%
        All Exons Sensitivity     12.83%
        Exact Intron Count        9816.00
        Exact Intron Matched      10771.00
        Exact Intron Specificity  37.58%
        Exact Intron Sensitivity  77.98%
        Exact Exon Count          10789.00
        Exact Exon Matched        11325.00
        Exact Exon Specificity    41.31%
        Exact Exon Sensitivity    81.99%
        Start Codon Count         4636.00
        Start Codon Matched       5230.00
        Start Codon Specificity   17.75%
        Start Codon Sensitivity   37.87%
        Stop Codon Count          6930.00
        Stop Codon Matched        7635.00
        Stop Codon Specificity    26.53%
        Stop Codon Sensitivity    55.28%
        Start Stop Count          2793.00
        Start Stop Matched        2998.00
        Start Stop Specificity    10.69%
        Start Stop Sensitivity    21.71%
Complete
        Count                     24918.00
        Ann Count                 12476.00
        Average Length            23712.27
        Total Length              590862393.00
        Average Coding Length     1361.40
        Total Coding Length       33923293.00
        Average Score             217.02
        Total Score               5407714.92
        Exons Per                 7.82
        Total Exons               194895.00
        Correct Count             1712.00
        Correct Matched           1704.00
        Correct Specificity       6.87%
        Correct Sensitivity       13.66%
        Exact Count               1646.00
        Exact Matched             1704.00
        Exact Specificity         6.61%
        Exact Sensitivity         13.66%
        Overlap Count             11759.00
        Overlap Matched           11394.00
        Overlap Specificity       47.19%
        Overlap Sensitivity       91.33%
        Nuc Overlap Count         11606.00
        Nuc Overlap Matched       11318.00
```

```
        Nuc Overlap Specificity    46.58%
        Nuc Overlap Sensitivity    90.72%
        All Introns Count          1729.00
        All Introns Matched        1594.00
        All Introns Specificity    6.94%
        All Introns Sensitivity    12.78%
        All Exons Count            1712.00
        All Exons Matched          1704.00
        All Exons Specificity      6.87%
        All Exons Sensitivity      13.66%
        Exact Intron Count         9340.00
        Exact Intron Matched       9902.00
        Exact Intron Specificity   37.48%
        Exact Intron Sensitivity   79.37%
        Exact Exon Count           10265.00
        Exact Exon Matched         10418.00
        Exact Exon Specificity     41.20%
        Exact Exon Sensitivity     83.50%
        Start Codon Count          4502.00
        Start Codon Matched        5162.00
        Start Codon Specificity    18.07%
        Start Codon Sensitivity    41.38%
        Stop Codon Count           6803.00
        Stop Codon Matched         7312.00
        Stop Codon Specificity     27.30%
        Stop Codon Sensitivity     58.61%
        Start Stop Count           2793.00
        Start Stop Matched         2998.00
        Start Stop Specificity     11.21%
        Start Stop Sensitivity     24.03%
Incomplete
        Count                      1201.00
        Ann Count                  1336.00
        Average Length             30893.59
        Total Length               37103201.00
        Average Coding Length      1279.15
        Total Coding Length        1536260.00
        Average Score              177.17
        Total Score                212783.83
        Exons Per                  8.19
        Total Exons                9834.00
        Correct Count              0.00
        Correct Matched            68.00
        Correct Specificity        0.00%
        Correct Sensitivity        5.09%
        Exact Count                0.00
        Exact Matched              0.00
        Exact Specificity          0.00%
        Exact Sensitivity          0.00%
        Overlap Count              571.00
        Overlap Matched            1205.00
        Overlap Specificity        47.54%
        Overlap Sensitivity        90.19%
        Nuc Overlap Count          566.00
        Nuc Overlap Matched        1199.00
        Nuc Overlap Specificity    47.13%
        Nuc Overlap Sensitivity    89.75%
        All Introns Count          8.00
        All Introns Matched        197.00
        All Introns Specificity    0.67%
        All Introns Sensitivity    14.75%
        All Exons Count            0.00
        All Exons Matched          68.00
```

```
            All Exons Specificity      0.00%
            All Exons Sensitivity      5.09%
            Exact Intron Count         476.00
            Exact Intron Matched       869.00
            Exact Intron Specificity   39.63%
            Exact Intron Sensitivity   65.04%
            Exact Exon Count           524.00
            Exact Exon Matched         907.00
            Exact Exon Specificity     43.63%
            Exact Exon Sensitivity     67.89%
            Start Codon Count          134.00
            Start Codon Matched        68.00
            Start Codon Specificity    11.16%
            Start Codon Sensitivity    5.09%
            Stop Codon Count           127.00
            Stop Codon Matched         323.00
            Stop Codon Specificity     10.57%
            Stop Codon Sensitivity     24.18%
            Start Stop Count           0.00
            Start Stop Matched         0.00
            Start Stop Specificity     0.00%
            Start Stop Sensitivity     0.00%
Exon
      All
            Count                      204729.00
            Ann Count                  109883.00
            Average Length             173.20
            Total Length               35459553.00
            Average Score              21.69
            Total Score                4440400.20
            Correct Count              78987.00
            Correct Matched            78992.00
            Correct Specificity        38.58%
            Correct Sensitivity        71.89%
            Overlap Count              92156.00
            Overlap Matched            92300.00
            Overlap Specificity        45.01%
            Overlap Sensitivity        84.00%
            Overlap 80p Count          84955.00
            Overlap 80p Matched        85257.00
            Overlap 80p Specificity    41.50%
            Overlap 80p Sensitivity    77.59%
            Splice 5 Count             84203.00
            Splice 5 Matched           84594.00
            Splice 5 Specificity       41.13%
            Splice 5 Sensitivity       76.99%
            Splice 3 Count             85864.00
            Splice 3 Matched           86150.00
            Splice 3 Specificity       41.94%
            Splice 3 Sensitivity       78.40%
      Initial
            Count                      22002.00
            Ann Count                  10164.00
            Average Length             168.70
            Total Length               3711673.00
            Average Score              18.96
            Total Score                417156.27
            Correct Count              4056.00
            Correct Matched            3912.00
            Correct Specificity        18.43%
            Correct Sensitivity        38.49%
            Overlap Count              6475.00
            Overlap Matched            6982.00
```

```
        Overlap Specificity       29.43%
        Overlap Sensitivity       68.69%
        Overlap 80p Count         5001.00
        Overlap 80p Matched       5063.00
        Overlap 80p Specificity   22.73%
        Overlap 80p Sensitivity   49.81%
        Splice 5 Count            4476.00
        Splice 5 Matched          4313.00
        Splice 5 Specificity      20.34%
        Splice 5 Sensitivity      42.43%
        Splice 3 Count            5863.00
        Splice 3 Matched          6395.00
        Splice 3 Specificity      26.65%
        Splice 3 Sensitivity      62.92%
Internal
        Count                     157247.00
        Ann Count                 87588.00
        Average Length            154.42
        Total Length              24281817.00
        Average Score             22.12
        Total Score               3478941.25
        Correct Count             68635.00
        Correct Matched           68806.00
        Correct Specificity       43.65%
        Correct Sensitivity       78.56%
        Overlap Count             77015.00
        Overlap Matched           76576.00
        Overlap Specificity       48.98%
        Overlap Sensitivity       87.43%
        Overlap 80p Count         72257.00
        Overlap 80p Matched       72793.00
        Overlap 80p Specificity   45.95%
        Overlap 80p Sensitivity   83.11%
        Splice 5 Count            72006.00
        Splice 5 Matched          72441.00
        Splice 5 Specificity      45.79%
        Splice 5 Sensitivity      82.71%
        Splice 3 Count            73046.00
        Splice 3 Matched          72810.00
        Splice 3 Specificity      46.45%
        Splice 3 Sensitivity      83.13%
Terminal
        Count                     21745.00
        Ann Count                 10670.00
        Average Length            237.55
        Total Length              5165434.00
        Average Score             14.73
        Total Score               320222.35
        Correct Count             5973.00
        Correct Matched           5978.00
        Correct Specificity       27.47%
        Correct Sensitivity       56.03%
        Overlap Count             7885.00
        Overlap Matched           7802.00
        Overlap Specificity       36.26%
        Overlap Sensitivity       73.12%
        Overlap 80p Count         7047.00
        Overlap 80p Matched       6752.00
        Overlap 80p Specificity   32.41%
        Overlap 80p Sensitivity   63.28%
        Splice 5 Count            7193.00
        Splice 5 Matched          7330.00
        Splice 5 Specificity      33.08%
```

```
        Splice 5 Sensitivity      68.70%
        Splice 3 Count            6454.00
        Splice 3 Matched          6366.00
        Splice 3 Specificity      29.68%
        Splice 3 Sensitivity      59.66%
    Single
        Count                     3735.00
        Ann Count                 1488.00
        Average Length            615.96
        Total Length              2300629.00
        Average Score             59.99
        Total Score               224080.33
        Correct Count             323.00
        Correct Matched           307.00
        Correct Specificity       8.65%
        Correct Sensitivity       20.63%
        Overlap Count             781.00
        Overlap Matched           960.00
        Overlap Specificity       20.91%
        Overlap Sensitivity       64.52%
        Overlap 80p Count         650.00
        Overlap 80p Matched       665.00
        Overlap 80p Specificity   17.40%
        Overlap 80p Sensitivity   44.69%
        Splice 5 Count            528.00
        Splice 5 Matched          524.00
        Splice 5 Specificity      14.14%
        Splice 5 Sensitivity      35.22%
        Splice 3 Count            501.00
        Splice 3 Matched          595.00
        Splice 3 Specificity      13.41%
        Splice 3 Sensitivity      39.99%
    Intron
        Count                     178610.00
        Ann Count                 97658.00
        Average Length            3316.52
        Total Length              592363950.00
        Average Score             0.00
        Total Score               0.00
        Correct Count             70985.00
        Correct Matched           70989.00
        Correct Specificity       39.74%
        Correct Sensitivity       72.69%
        Overlap Count             88390.00
        Overlap Matched           87982.00
        Overlap Specificity       49.49%
        Overlap Sensitivity       90.09%
        Overlap 80p Count         75066.00
        Overlap 80p Matched       75303.00
        Overlap 80p Specificity   42.03%
        Overlap 80p Sensitivity   77.11%
        Splice 5 Count            78619.00
        Splice 5 Matched          79168.00
        Splice 5 Specificity      44.02%
        Splice 5 Sensitivity      81.07%
        Splice 3 Count            78932.00
        Splice 3 Matched          79421.00
        Splice 3 Specificity      44.19%
        Splice 3 Sensitivity      81.33%
Nuc
    All
        Count                     35459553.00
        Ann Count                 18102268.00
```

```
            Correct Count              14873635.00
            Correct Matched            15115875.00
            Correct Specificity        41.95%
            Correct Sensitivity        83.50%
        Initial
            Count                      3711673.00
            Ann Count                  1835955.00
            Correct Count              1239150.00
            Correct Matched            1373641.00
            Correct Specificity        33.39%
            Correct Sensitivity        74.82%
        Internal
            Count                      24281817.00
            Ann Count                  12571788.00
            Correct Count              11095186.00
            Correct Matched            10953990.00
            Correct Specificity        45.69%
            Correct Sensitivity        87.13%
        Terminal
            Count                      5165434.00
            Ann Count                  2400130.00
            Correct Count              1891305.00
            Correct Matched            1853291.00
            Correct Specificity        36.61%
            Correct Sensitivity        77.22%
        Single
            Count                      2300629.00
            Ann Count                  1298481.00
            Correct Count              647994.00
            Correct Matched            937630.00
            Correct Specificity        28.17%
            Correct Sensitivity        72.21%
        Intron
            Count                      592363950.00
            Ann Count                  541215329.00
            Correct Count              254351879.00
            Correct Matched            260829844.00
            Correct Specificity        42.94%
            Correct Sensitivity        48.19%
    Signal
        Splice Acceptor
            Count                      178992.00
            Ann Count                  98254.00
            Correct Count              0.00
            Correct Matched            0.00
            Correct Specificity        0.00%
            Correct Sensitivity        0.00%
        Splice Donor
            Count                      179249.00
            Ann Count                  97729.00
            Correct Count              0.00
            Correct Matched            0.00
            Correct Specificity        0.00%
            Correct Sensitivity        0.00%
        Start Codon
            Count                      25700.00
            Ann Count                  12656.00
            Correct Count              0.00
            Correct Matched            0.00
            Correct Specificity        0.00%
            Correct Sensitivity        0.00%
        Stop Codon
            Count                      25335.00
```

Ann Count	13074.00
Correct Count	0.00
Correct Matched	0.00
Correct Specificity	0.00%
Correct Sensitivity	0.00%

Index

ab initio .... - 7 -
bins .... - 19 -
central dogma of molecular biology........ - 5 -
cluster .... - 20 -
codon .... - 5 -
conservation sequence........ - 9 -, - 90 -
data types
  ann_listbox .... - 73 -
  aref........ - 32 -
  array........ - 32 -
  Boolean........ - 32 -
  cluster .... - 69 -
  cluster_count .... - 69 -
  Feature........ - 46 -
  fh .... - 32 -
  filter .... - 62 -
  float........ - 32 -
  frame........ - 73 -
  fref .... - 32 -
  Gene .... - 38 -
  GTF .... - 34 -
  GTF_obj .... - 51 -
  GTF_set........ - 51 -
  hash........ - 32 -
  href .... - 32 -
  int........ - 32 -
  listbox .... - 73 -
  pred_listbox........ - 73 -
  pvar........ - 32 -
  resolution........ - 65 -
  stats_struct........ - 52 -
  Transcript .... - 40 -
de novo .... - 7 -
Distribution........ *See* Eval Functions
DNA .... - 6 -
eukaryotes........ - 5 -
Eval
  command line interface .... - 27 -
  Eval Report........ - 91 -
  Functions
    Distribution........ - 21 -, - 26 -, - 30 -, - 70 -
    Evaluate........ - 18 -, - 24 -, - 27 -, - 56 -
    Filter .... - 18 -, - 24 -, - 28 -, - 62 -
    General Statistics........ - 18 -, - 24 -, - 27 -, - 61 -

Graph .... - 19 -, - 25 -, - 28 -, - 64 -
Overlap .... - 20 -, - 26 -, - 29 -, - 67 -
GUI .... - 22 -
requirements .... - 10 -
Statistics .... - 12 -
eval.pl .... *See* programs
Eval.pm .... *See* libraries
Evaluate .... *See* Eval Functions
evaluate_gtf.pl .... *See* programs
fasta .... - 10 -, - 22 -, - 89 -
Filter .... *See* Eval Functions
filter_gtfs.pl .... *See* programs
gene .... - 5 -
gene expression .... - 5 -
gene structure .... - 5 -, - 7 -
General Statistics .... *See* Eval Functions
get_distribution.pl .... *See* programs
get_general_stats.pl .... *See* programs
get_overlap_stats.pl .... *See* programs
gnuplot .... - 10 -, - 25 -, - 81 -
Graph .... *See* Eval Functions
graph_gtfs.pl .... *See* programs
GTF .... - 7 -, - 10 -, - 51 -, - 86 -
Fields
attributes .... - 87 -
end .... - 86 -
feature .... - 86 -
frame .... - 87 -
score .... - 86 -
seqname .... - 86 -
source .... - 86 -
start .... - 86 -
GTF set .... - 12 -, - 51 -
GTF.pm .... *See* libraries
GTF.pm style objects .... - 33 -
initial exon .... - 7 -
libraries
Eval.pm .... - 51 -
GTF.pm .... - 33 -
list file .... - 22 -
mRNA .... - 5 -
Overlap .... *See* Eval Functions
overlap property .... - 20 -
primary transcript .... - 5 -
programs
eval.pl .... - 22 -, - 72 -

evaluate_gtf.pl ... - 27 -, - 83 -
filter_gtfs.pl ... - 28 -, - 83 -
get_distribution.pl ... - 30 -, - 84 -
get_general_stats.pl ... - 27 -, - 83 -
get_overlap_stats.pl ... - 29 -, - 84 -
graph_gtfs.pl ... - 28 -, - 84 -
validate_gtf.pl ... - 10 -
prokaryotes ... - 5 -
protein ... - 5 -
RNA ... - 5 -
splice sites ... - 7 -
start codon ... - 7 -
stop codon ... - 7 -
terminal exon ... - 7 -
Tk ... - 10 -, - 27 -, - 72 -
transcript region ... - 12 -
transcription ... - 5 -, - 6 -
translation ... - 5 -, - 6 -
TWINSCAN ... - 9 -, - 90 -
untranslated region (UTR) ... - 7 -
validate_gtf.pl ... *See* programs
X-Split ... - 19 -

References

1. http://www.sanger.ac.uk/Software/formats/GFF/GFF_Spec.shtml; GFF specification.
2. http://www.cs.wisc.edu/~ghost/; Ghostview.
3. http://blast.wustl.edu/; Gish, W., WU-BLAST.
4. http://www.gnuplot.info/; Gnuplot.
5. Burge, C. and S. Karlin, *Prediction of complete gene structures in human genomic DNA.* J Mol Biol, 1997. **268**(1): p. 78-94.
6. Burge, C.B. and S. Karlin, *Finding the genes in genomic DNA.* Curr Opin Struct Biol, 1998. **8**(3): p. 346-54.
7. Burset, M. and R. Guigo, *Evaluation of gene structure prediction programs.* Genomics, 1996. **34**(3): p. 353-67.
8. Fickett, J.W., *Finding genes by computer: the state of the art.* Trends Genet, 1996. **12**(8): p. 316-20.
9. Flicek, P., et al., *Leveraging the mouse genome for gene prediction in human: from whole-genome shotgun reads to a global synteny map.* Genome Res, 2003. **13**(1): p. 46-54.
10. Guigo, R., et al., *An assessment of gene prediction accuracy in large DNA sequences.* Genome Res, 2000. **10**(10): p. 1631-42.
11. Guigo, R., et al., *Prediction of gene structure.* J Mol Biol, 1992. **226**(1): p. 141-57.
12. Korf, I., et al., *Integrating genomic homology into gene structure prediction.* Bioinformatics, 2001. **17 Suppl 1**: p. S140-8.
13. Lodish, H., et al., *Molecular Cell Biology*. 4th ed. 2000, New York: W. H. Freeman and Company.
14. Mathe, C., et al., *Current methods of gene prediction, their strengths and weaknesses.* Nucleic Acids Res, 2002. **30**(19): p. 4103-17.
15. Parra, G., et al., *Comparative gene prediction in human and mouse.* Genome Res, 2003. **13**(1): p. 108-17.