Supplemental Material
Building better genome annotations across the tree of life
Adam H. Freedman, Timothy B. Sackton
Faculty of Arts and Sciences Informatics Group, Harvard University, Cambridge, Massachusetts 02138

Table of Contents

Supplemental Code S1

```
###### BuildUtrCdsRatioTable.py
import sys
from numpy import median
cds_lengths = open(sys.argv[1],'r')
cds_fields = cds_lengths.readline().strip().split('\t')
cds_dict = {}
for line in cds_lengths:
    line_dict =  dict(zip(cds_fields,line.strip().split('\t')))
    cds_dict[line_dict['seqid']] = int(line_dict['length'])

utr_lengths = open(sys.argv[2],'r')
utr_fields = utr_lengths.readline().strip().split('\t')
utr_dict = {}
for line in utr_lengths:
    line_dict =  dict(zip(utr_fields,line.strip().split('\t')))
    utr_dict[line_dict['tsid']] = int(line_dict['total_utr_length'])
utr2cds_ratios = []
fout = open(sys.argv[3],'w')
fout.write('tsid\tcds_length\tutr_length\tutr2cds\n')
for tsid in cds_dict:
    if tsid in utr_dict:
        fout.write('%s\t%s\t%s\t%s\n' %
(tsid,cds_dict[tsid],utr_dict[tsid],utr_dict[tsid]/cds_dict[tsid]))
        utr2cds_ratios.append(utr_dict[tsid]/cds_dict[tsid])
    else:
        fout.write('%s\t%s\t0\t0\n' % (tsid,cds_dict[tsid]))
fout.close()
print(median(utr2cds_ratios))

############
###### CalcGeneUniqueToMethodClassRate.py
methods = ['braker','cgp','lift','maker','scal','stie']
def ClassifyMethodList(methods,methodlist):
    method_classes = set()
    if 'ncbi' in methodlist:
        ncbi = True
    else:
        ncbi = False
    for method in methods:
        method_present = [method in i for i in methodlist]
        if True in method_present:
            method_classes.add(method)
        if sum(method_present) >0:
            if len(method_present) == sum(method_present):
                unique = True
                return unique,ncbi,list(method_classes)
                break
            elif len(method_present) - sum(method_present) == 1 and
'ncbi' in methodlist:
                unique = True
                return unique,ncbi,list(method_classes)
                break
```

```
    if 'unique' not in locals():
        return False,ncbi,list(method_classes)

import sys
from numpy import corrcoef
from scipy.stats import pearsonr
from collections import defaultdict
mergebed = open(sys.argv[1],'r')
species =  sys.argv[2]
method_total = defaultdict(int)
method_intergenic = defaultdict(int)
method_unique = defaultdict(int)
for method in methods:
    method_total[method] = 0
    method_intergenic[method] =0
    method_unique[method] = 0
for line in mergebed:
    methodlist = line.strip().split()[3].split(',')
    unique,ncbi,method_classes = ClassifyMethodList(methods,methodlist)
    for methodclass in method_classes:
        method_total[methodclass]+=1

    if unique == True and ncbi == False:
        method_unique[method_classes[0]]+=1
        method_intergenic[method_classes[0]]+=1
    elif unique == True and ncbi ==True:
        method_unique[method_classes[0]]+=1
    elif unique == False and ncbi == False:
        for method in method_classes:
            method_intergenic[method]+=1
fout =
open('{}_methodclass_unique_intergenic_rates.tsv'.format(sys.argv[1]),'w'
)
fout.write('species\tmethodclass\tclassunique\tclassintergenic\n')
x= []
y=[]
for methodclass in methods:
    x.append(method_unique[methodclass]/method_total[methodclass])
    y.append(method_intergenic[methodclass]/method_total[methodclass])

fout.write('{}\t{}\t{}\t{}\n'.format(species,methodclass,method_unique[me
thodclass]/method_total[methodclass],method_intergenic[methodclass]/metho
d_total[methodclass]))
fout.close()

print(corrcoef(x,y))
print(pearsonr(x,y))
#########
```

CalculateBlastpFractionRefProteinsCovered.py

```
import argparse
fields='qseqid sseqid pident length mismatch gapopen qstart qend sstart
send evalue bitscore qlen slen'.split()

if __name__=="__main__":
    parser = argparse.ArgumentParser(description='calculate fraction of
reference proteins covered by at least one predicted protein')
    parser.add_argument('-blast','--blast-
table',dest='blast',type=str,help='blast output table')
    parser.add_argument('-refn','--number-reference-
proteins',dest='refprotn',type=int,help='number of reference proteins')
    parser.add_argument('-cov','--ref-cov-
threshold',dest='coverage',type=float,help='fractional coverage
threshold')
    opts = parser.parse_args()
    refprotset=set()
    fopen = open(opts.blast,'r')
    for line in fopen:
        linedict = dict(zip(fields,line.strip().split('\t')))
        coverage = int(linedict['length'])/int(linedict['qlen'])
        if float(linedict['evalue']) <= 10**-5 and coverage
>opts.coverage:
            refprotset.add(linedict['qseqid'])
    frac_covered = len(refprotset)/int(opts.refprotn)
    fout =
open('refprotein_proportioncovered_cov_{}covthresh.txt'.format(opts.cover
age),'w')
    fout.write('proportion ref proteins with
coverage:{}\n'.format(str(frac_covered)))
    print(frac_covered)
    fout.close()
```

#########

CalculateTotalUtrLengthFromUtrBed.py

```
import sys
from collections import defaultdict
from numpy import median
ts_list = []
tscript_fasta = open(sys.argv[1],'r')
for line in tscript_fasta:
    if line[0] == '>':
        ts_list.append(line.strip().split()[0].replace('>',''))
utr_bed_open = open(sys.argv[2],'r')

utr_length_dict = defaultdict(int)
for interval in utr_bed_open:
    chrom,start,end,tsid = interval.strip().split('\t')
    utr_length_dict[tsid] += int(end)-int(start)

fout = open('%s_utrlengths.tsv' % sys.argv[1],'w')
fout.write('tsid\ttotal_utr_length\n')
utr_lengths = []
```

```
for tsid in ts_list:
    if tsid in utr_length_dict:
        fout.write('%s\t%s\n' % (tsid,utr_length_dict[tsid]))
        utr_lengths.append(utr_length_dict[tsid])
else:
        fout.write('%s\t0\n' % tsid)
        utr_lengths.append(0)
fout.close()

utr_median = open('%s_utrmedian_length.txt' % sys.argv[1],'w')
utr_median.write('%s\n' % median(utr_lengths))
utr_median.close()
```
##############

FilterOutUnsupportedBrakerAugustusAnnotations.py

```
import argparse
fields = ["seqid", "source", "type", "start",
          "end", "score", "strand", "phase", "attributes"]

def ParseBrakerGtfAttributes(attributes,cleanlabels=True):
    attribute_dict = {}
    attribute_list = attributes[:-1].replace('"','').split(';')
    for attribute in attribute_list:
        key,value = attribute.split()
        if cleanlabels == True:
            if key in ['transcript_id','gene_id'] and 'file' in value:
                value = value.split('_')[-1]
        attribute_dict[key] =  value
    return attribute_dict

def BuiltTscript2GeneDict(brakergtf,fields):
    fopen = open(brakergtf,'r')
    ts2gene = {}
    for line in fopen:
        linedict = dict(zip(fields,line.strip().split('\t')))
        if linedict['type'] == 'CDS':
            attribute_dict =
ParseBrakerGtfAttributes(linedict['attributes'],cleanlabels=False)
            ts2gene[attribute_dict['transcript_id']] =
attribute_dict['gene_id']
    fopen.close()
    return ts2gene

if __name__=="__main__":
    parser = argparse.ArgumentParser(description="Filters braker.gtf by
retaining only supported annotations, with either any or full support
gtfs produced by BRAKER selectSupportedSubsets.py script.")
    parser.add_argument('-supported','--hint-supported-
gtf',dest='supported',type=str,help='braker support eval script generated
gtf with any support')
    parser.add_argument('-gtf','--braker-
gtf',dest='braker',type=str,help='braker.gtf output file')
```

```
    parser.add_argument('-o','--output-
gtf',dest='output',type=str,help='merge of GeneMark and supported
AUGUSTUS annotations')
    parser.add_argument('-blastp','--use-blastp-
support',dest='blastp',action='store_true',help='switch indicating
whether use blastp evidence to keep annotations')
    parser.add_argument('-bp-file','--blastp-results-
file',dest='blastpfile',type=str,help='name of file with blastp results,
outputformat 6')
    opts = parser.parse_args()

    if opts.blastp == True:
        ts2gene = BuiltTscript2GeneDict(opts.braker,fields)
        blastp_supported_genes = set()
        blastp_supported_tscripts = set()
        blastp_open = open(opts.blastpfile,'r')
        for line in blastp_open:
            linelist = line.strip().split('\t')
            blastp_supported_tscripts.add(linelist[0])
            blastp_supported_genes.add(ts2gene[linelist[0]])

    supported_genes = set()
    supported_tscripts = set()
    supported = open(opts.supported,'r')
    for line in supported:
        if line[0] !='#':
            linedict = dict(zip(fields,line.strip().split('\t')))
            if linedict['type'] in ['intron','start_codon','stop_codon']:
                attribute_dict =
ParseBrakerGtfAttributes(linedict['attributes'])
                if attribute_dict['supported'] == 'True':
                    supported_genes.add(attribute_dict['gene_id'])

supported_tscripts.add(attribute_dict['transcript_id'])

    brakerin = open(opts.braker,'r')
    fout = open(opts.output,'w')
    for line in brakerin:
        if line[0] == '#':
            fout.write(line)
        elif line =='\n':
            pass
        elif "GeneMark" in line:
            if opts.blastp == True:
                linedict = dict(zip(fields,line.strip().split('\t')))
                attribute_dict =
ParseBrakerGtfAttributes(linedict['attributes'],cleanlabels=False)
                if attribute_dict['transcript_id'] in
blastp_supported_tscripts and attribute_dict['gene_id'] in
blastp_supported_genes:
                    fout.write(line)
            else:
                fout.write(line)
        else:
```

```
            linedict = dict(zip(fields,line.strip().split('\t')))
            if linedict['type'] == 'gene':
                if linedict['attributes'].split('_')[-1] in
supported_genes:
                    fout.write(line)
                elif linedict['attributes'] in blastp_supported_genes:
                    fout.write(line)
            elif linedict['type'] == 'transcript':
                if linedict['attributes'].split('_')[-1] in
supported_tscripts:
                    fout.write(line)
                elif linedict['attributes'] in blastp_supported_tscripts:
                    fout.write(line)
            elif linedict['type'] in
['exon','CDS','stop_codon','start_codon','intron']:
                attribute_dict =
ParseBrakerGtfAttributes(linedict['attributes'])
                if attribute_dict['gene_id'] in supported_genes and
attribute_dict['transcript_id'] in supported_tscripts:
                    fout.write(line)
                else:
                    attribute_dict =
ParseBrakerGtfAttributes(linedict['attributes'],cleanlabels=False)
                    if attribute_dict['gene_id'] in
blastp_supported_genes and attribute_dict['transcript_id'] in
blastp_supported_tscripts:
                        fout.write(line)
            else:
                raise ValueError('%s not in list of valid feature types'
% line_dict['type'])

    fout.close()
#########
```

GenerateFastaSeqLengthTable.py

```
from Bio import SeqIO
from numpy import median
import argparse

if __name__=="__main__":
    parser = argparse.ArgumentParser(description='args for computing
stats from assembly fiile')
    parser.add_argument('-f','--
fastain',dest='frecords',type=str,help='input transcriptome fasta file')
    parser.add_argument('-colname','--output-column-
name',dest='colname',type=str,help='column name for written table')
    opts = parser.parse_args()

    lengths=[]
    fout = open('%s_seqlengths.tsv' % (opts.frecords),'w')
    fout.write('seqid\tlength\tmethod\n')
    for record in SeqIO.parse(opts.frecords,'fasta'):
        fout.write('%s\t%s\t%s\n' %
(record.id,len(record.seq),opts.colname))
```

```
        lengths.append(len(record.seq))
    print('median: %s\n' % median(lengths))
    print('max: %s\n' % max(lengths))
    print('min: %s\n' % min(lengths))
fout.close()
##########
```

GetMedianAnnotationAlignRate.py

```
import sys
from glob import glob
from numpy import median
prefix = sys.argv[1]

logs = glob("*log")
rates = []
for log in logs:
    logopen = open(log,'r')
    logread = logopen.readlines()
    for line in logread:
        if 'overall' in line:
            rate = float(line.split()[0].replace('%',''))
            rates.append(rate)
fout = open('median_align_rate_%s.txt' % prefix, 'w')
fout.write('median annotation align rate: %s\n' % median(rates))
fout.close()
##########
```

WriteBrakerCdsTranscriptAndGeneIntervalBedFilesWithNoUTRs.py

```
import argparse
from collections import OrderedDict

fields = ["seqid", "source", "type", "start",
          "end", "score", "strand", "phase", "attributes"]
ts_dict = OrderedDict()
ts2gene = {}
gene_dict = OrderedDict()

def ParseBrakerAttributes(attributes):
    attribute_dict = {}
    attribute_list = attributes.split(';')
    for attribute in attribute_list:
        key,value = attribute.split('=')
        attribute_dict[key] =  value
    return attribute_dict

def BuildTscript2GeneforLiftoff(gff3):
    ts2gene = {}
    fopen = open(gff3,'r')
    for line in fopen:
        if line[0] != '#':
            line_dict = dict(zip(fields,line.strip().split('\t')))
            if line_dict['type'] == 'mRNA':
                attribute_dict =
ParseBrakerAttributes(line_dict['attributes'])
```

```
                ts2gene[attribute_dict['ID']] = attribute_dict['Parent']
    fopen.close()
    return ts2gene

if __name__=="__main__":
    parser = argparse.ArgumentParser(description="Generate CDS transcript
and gene interval bed without UTRs")
    parser.add_argument('-gff3','--braker-annotation-
gff3',dest='gff3',type=str,help='Braker gff3')
    parser.add_argument('-o','--output-bed-
prefix',dest='bedprefix',type=str,help='prefix for output bed files')
    parser.add_argument('-maker','--
makergff',dest='maker',action='store_true',help='indicates maker
annotation so handle gene id properly')
    parser.add_argument('-liftoff','--liftoff-
gff3',dest='liftoff',action='store_true',help='indicates liftoff
annotation')
    opts = parser.parse_args()

    if opts.liftoff == True:
        ts2gene = BuildTscript2GeneforLiftoff(opts.gff3)
    else:
        ts2gene = {}
    gffin = open(opts.gff3,'r')
    for line in gffin:
        if line[0] != '#':
            line_dict = dict(zip(fields,line.strip().split('\t')))
            if line_dict['type'] in  ['mRNA','transcript']:
                attribute_dict =
ParseBrakerAttributes(line_dict['attributes'])
                tsid = attribute_dict['ID']
                if opts.maker ==True:
                    geneid = attribute_dict['Parent']
                else:
                    try:
                        geneid = attribute_dict['geneID']
                    except:
                        geneid = attribute_dict['Parent']  # added
                if opts.liftoff == False:
                    ts2gene[tsid] = geneid
            elif line_dict['type'] == 'CDS':
                attribute_dict =
ParseBrakerAttributes(line_dict['attributes'])
                tsid = attribute_dict['Parent']
                if tsid in ts2gene: # this accounts for liftoff cases
where CDS and gene but no mRNA
                    geneid = ts2gene[tsid]
                elif 'gene' in attribute_dict['Parent']:
                    geneid = attribute_dict['Parent']

                # transcript level #
                if tsid not in ts_dict and 'gene' not in tsid:
```

```
                    ts_dict[tsid] = {'strand': line_dict['strand'],'chr':
line_dict['seqid'],'start': int(line_dict['start']), 'end':
int(line_dict['end'])}
                elif tsid in ts_dict and 'gene' not in tsid:
                    ts_dict[tsid]['start'] =
min(int(line_dict['start']),ts_dict[tsid]['start'])
                    ts_dict[tsid]['end'] =
max(int(line_dict['end']),ts_dict[tsid]['end'])
                else:
                    print('{} is also a gene feature'.format(tsid))

                # gene level #
                if geneid not in gene_dict:
                    gene_dict[geneid] = {'strand':
line_dict['strand'],'chr': line_dict['seqid'],'start':
int(line_dict['start']), 'end': int(line_dict['end'])}
                else:
                    gene_dict[geneid]['start'] =
min(int(line_dict['start']),gene_dict[geneid]['start'])
                    gene_dict[geneid]['end'] =
max(int(line_dict['end']),gene_dict[geneid]['end'])

    tsout = open('%s_CDS_transcript_interval.bed' % opts.bedprefix,'w')
    geneout = open('%s_CDS_gene_interval.bed' % opts.bedprefix,'w')

    for transcript in ts_dict:
        tsout.write('%s\t%s\t%s\t%s\t.\t%s\n' %
(ts_dict[transcript]['chr'],ts_dict[transcript]['start']-
1,ts_dict[transcript]['end'],transcript,ts_dict[transcript]['strand']))
    for gene in gene_dict:
        geneout.write('%s\t%s\t%s\t%s\t.\t%s\n' %
(gene_dict[gene]['chr'],gene_dict[gene]['start']-
1,gene_dict[gene]['end'],gene,gene_dict[gene]['strand']))

    tsout.close()
    geneout.close()
#########
```

WriteTransdecoderCdsTranscriptAndGeneIntervalBedFiles.py

```
import argparse
from collections import OrderedDict

fields = ["seqid", "source", "type", "start",
          "end", "score", "strand", "phase", "attributes"]

ts_dict = OrderedDict()
ts2gene = {}
gene_dict = OrderedDict()

def ParseTransdecoderAttributes(attributes):
    attribute_dict = {}
    attribute_list = attributes.split(';')
    for attribute in attribute_list:
        key,value = attribute.split('=')
```

```
        attribute_dict[key] =  value
    return attribute_dict

if __name__=="__main__":
    parser = argparse.ArgumentParser(description="Generate CDS transcript
and gene interval bed without UTRs")
    parser.add_argument('-tdecgff3','--transdecoder-annotation-
gff3',dest='tdecgff3',type=str,help='transdecoder genome-coordinate
gff3')
    parser.add_argument('-o','--output-bed-
prefix',dest='bedprefix',type=str,help='prefix for output bed files')
    opts = parser.parse_args()

    gffin = open(opts.tdecgff3,'r')
    for line in gffin:
        if line[0] != '#':
            line_dict = dict(zip(fields,line.strip().split('\t')))
            if line_dict['type'] == 'mRNA':
                attribute_dict =
ParseTransdecoderAttributes(line_dict['attributes'])
                tsid = '.'.join(attribute_dict['ID'].split('.')[:-1])
                geneid = attribute_dict['Parent']
                ts2gene[tsid] = geneid
            elif line_dict['type'] == 'CDS':
                attribute_dict =
ParseTransdecoderAttributes(line_dict['attributes'])
                tsid = '.'.join(attribute_dict['Parent'].split('.')[:-1])
                #try:
                geneid = ts2gene[tsid]
                #except:
                #    tsid = tsid.replace('gene','rna')
                #    geneid = ts2gene[tsid]

                # transcript level #
                if tsid not in ts_dict:
                    ts_dict[tsid] = {'strand': line_dict['strand'],'chr':
line_dict['seqid'],'start': int(line_dict['start']), 'end':
int(line_dict['end'])}
                else:
                    ts_dict[tsid]['start'] =
min(int(line_dict['start']),ts_dict[tsid]['start'])
                    ts_dict[tsid]['end'] =
max(int(line_dict['end']),ts_dict[tsid]['end'])

                # gene level #
                if geneid not in gene_dict:
                    gene_dict[geneid] = {'strand':
line_dict['strand'],'chr': line_dict['seqid'],'start':
int(line_dict['start']), 'end': int(line_dict['end'])}
                else:
                    gene_dict[geneid]['start'] =
min(int(line_dict['start']),gene_dict[geneid]['start'])
                    gene_dict[geneid]['end'] =
max(int(line_dict['end']),gene_dict[geneid]['end'])
```

```
    tsout = open('%s_transdecoder_CDS_transcript_interval.bed' %
opts.bedprefix,'w')
    geneout = open('%s_transdecoder_CDS_gene_interval.bed' %
opts.bedprefix,'w')

    for transcript in ts_dict:
        tsout.write('%s\t%s\t%s\t%s\t.\t%s\n' %
(ts_dict[transcript]['chr'],ts_dict[transcript]['start']-
1,ts_dict[transcript]['end'],transcript,ts_dict[transcript]['strand']))
    for gene in gene_dict:
        geneout.write('%s\t%s\t%s\t%s\t.\t%s\n' %
(gene_dict[gene]['chr'],gene_dict[gene]['start']-
1,gene_dict[gene]['end'],gene,gene_dict[gene]['strand']))

    tsout.close()
    geneout.close()
```

Supplemental Methods S1: Quantifying undetected CDS in UTR predictions

To determine the extent to which TransDecoder incorrectly annotates CDS exons as UTR, we do the following.

First, from a StringTie+TransDecoder or Scallop+TransDecoder gff3 file, we first extract a CDS fasta with gffread as follows:

```
gffread annotation_transdecoder.genome.gff3 -g genome.fasta -x CDS.fasta
```

Next, we use a custom python script to extract the length of the CDS in CDS.fasta:

```
python GenerateFastaSeqLengthTable.py CDS.fasta
```

which produces CDS.fasta_seqlengths.tsv.

Next, we use awk to generate a UTR interval bed file:

```
awk -F'\t' '$3 == "five_prime_UTR" || $3 == "three_prime_UTR"{print $1"\t"$4-1"\t"$5"\t"$9}' annotation_transdecoder.genome.gff3|awk -F".utr" '{print $1}' |sed 's/ID=//g' > utr.bed
```

Next, we calculate the total UTR length (i.e. the sum of 5' and 3' UTR lengths per transcript) using a custom python script:

```
python CalculateTotalUtrLengthFromUtrBed.py CDS.fasta utr.bed
```

This script produces as output a tab-separated table, CDS.fasta_utrlengths.tsv, with the first and second columns containing the transcript id and the utr length, respectively.

Next we use a custom python script to calculate the median utr-to-CDS ratio across the set of CDS-containing transcripts:

```
python BuildUtrCdsRatioTable.py CDS.fasta_seqlengths.tsv CDS.fasta_utrlengths.tsv utr2cds_ratiotable.tsv > median_utr2cdsratio.txt
```

This script produces a table of UTR-to-CDS ratios for the CDS transcripts, but also prints to standard out the median,which we redirect to a text file (as demonstrated above).

Our subsequent analysis examines whether a correlation exists between the frequency of BLASTX hits for the predicted UTRs, and the ratio of the median UTR-to-CDS ratio for the TransDecoder annotation to that derived from NCBI. A value greater than one for this ratio of ratios indicates an excess of predicted UTR relative to NCBI. If as this ratio

or ratios gets larger (and exceeds 1), the frequency of UTR BLASTX hits also increases, we take this as evidence for incorrect classification of CDS as UTRs.

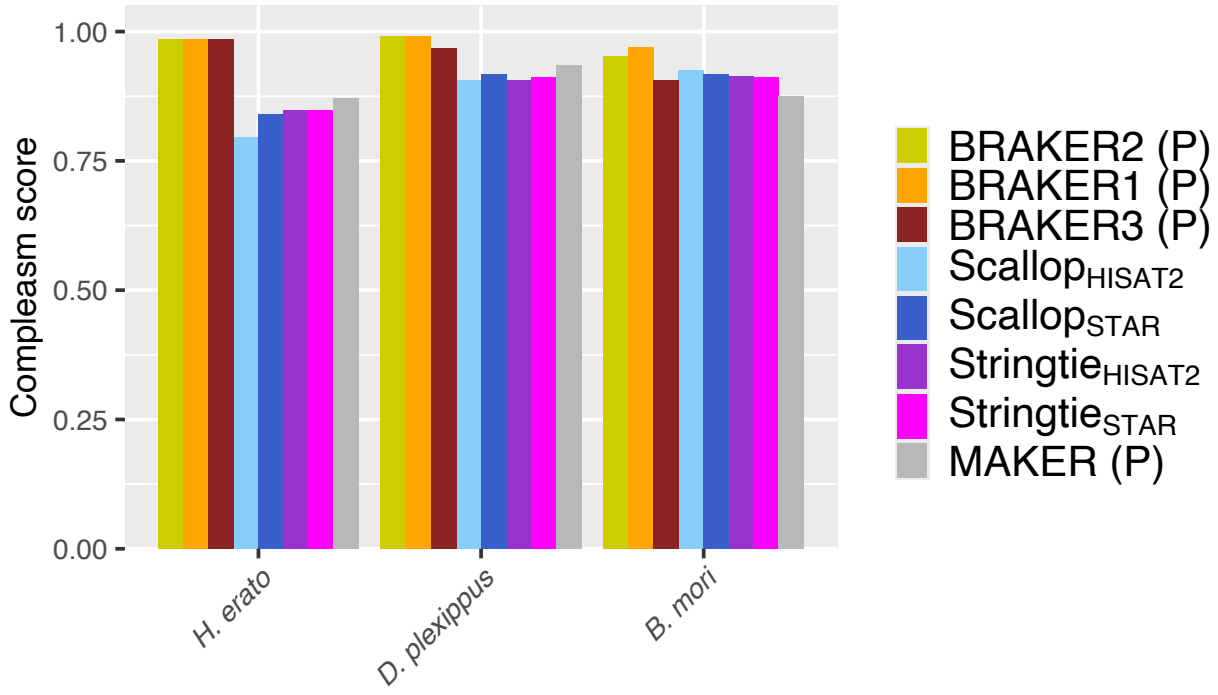
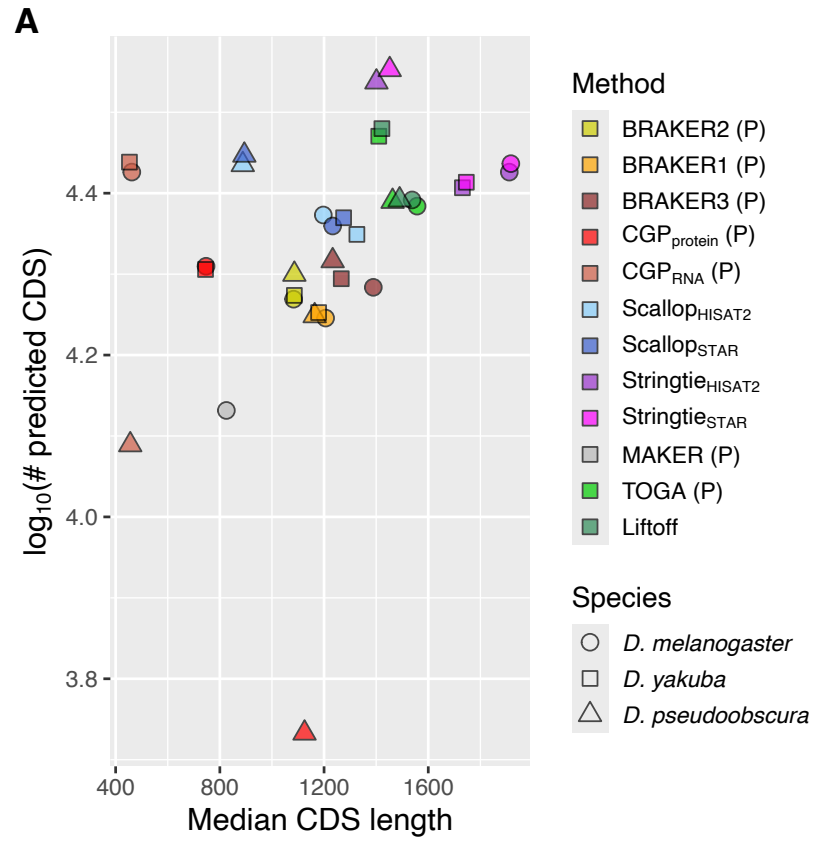
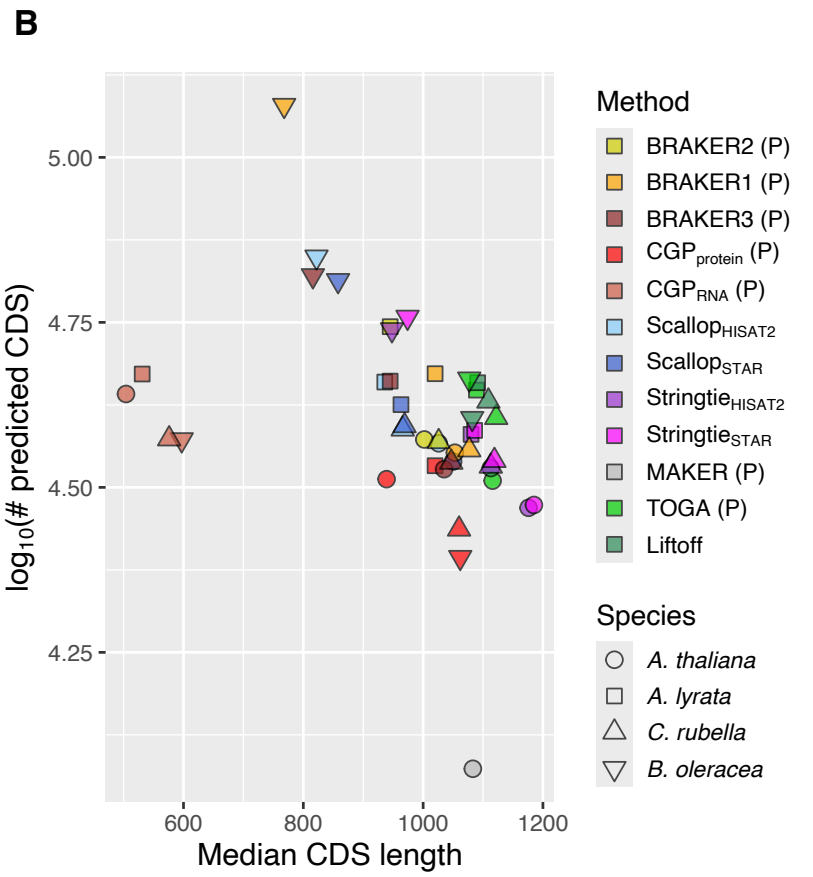
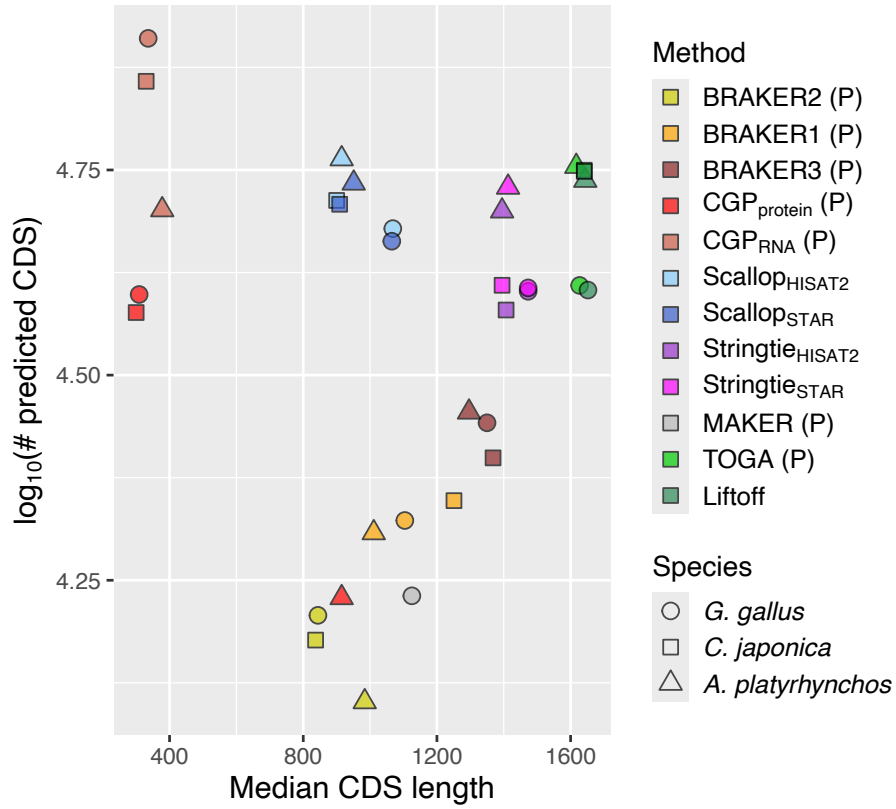


Figure S1. Proteome compleasm scores for leipidpoterans. For comparison to other taxonomic groups, see Figure 1.

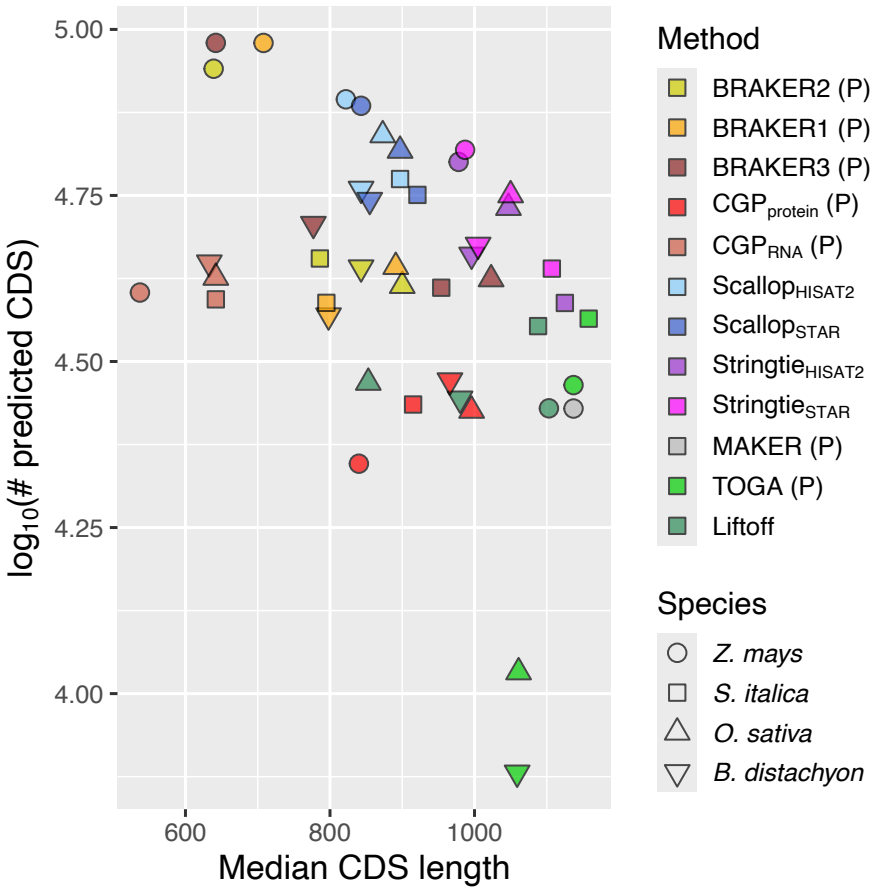

A
log10(# predicted CDS)
4.4
4.2
4.0
3.8
400
800
1200
1600
Median CDS length
Method
BRAKER2 (P)
BRAKER1 (P)
BRAKER3 (P)
CGPprotein (P)
CGPRNA (P)
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
MAKER (P)
TOGA (P)
Liftoff
Species
D. melanogaster
D. yakuba
D. pseudoobscura


B
log10(# predicted CDS)
5.00
4.75
4.50
4.25
600
800
1000
1200
Median CDS length
Method
BRAKER2 (P)
BRAKER1 (P)
BRAKER3 (P)
CGPprotein (P)
CGPRNA (P)
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
MAKER (P)
TOGA (P)
Liftoff
Species
A. thaliana
A. lyrata
C. rubella
B. oleracea

C


4.75
4.50
4.25
log10(# predicted CDS)
400
800
1200
1600
Median CDS length
Method
BRAKER2 (P)
BRAKER1 (P)
BRAKER3 (P)
CGPprotein (P)
CGPRNA (P)
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
MAKER (P)
TOGA (P)
Liftoff
Species
G. gallus
C. japonica
A. platyrhynchos

D


5.00
4.75
4.50
4.25
4.00
log10(# predicted CDS)
600
800
1000
Median CDS length
Method
BRAKER2 (P)
BRAKER1 (P)
BRAKER3 (P)
CGPprotein (P)
CGPRNA (P)
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
MAKER (P)
TOGA (P)
Liftoff
Species
Z. mays
S. italica
O. sativa
B. distachyon

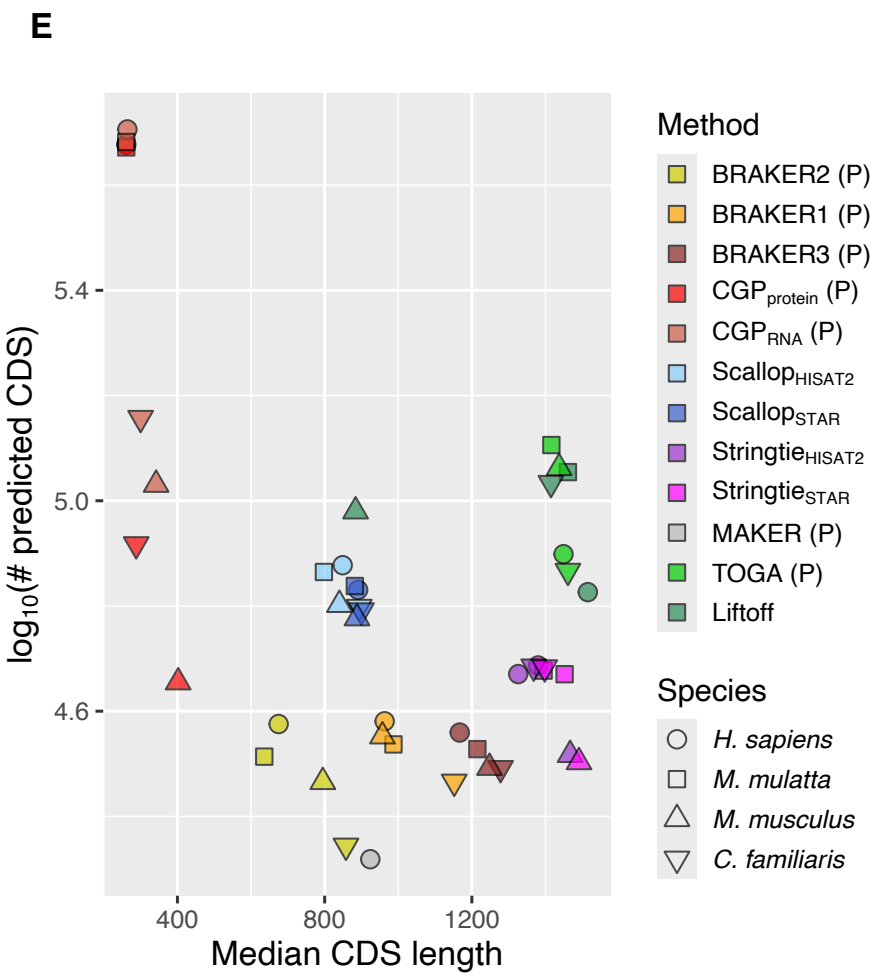
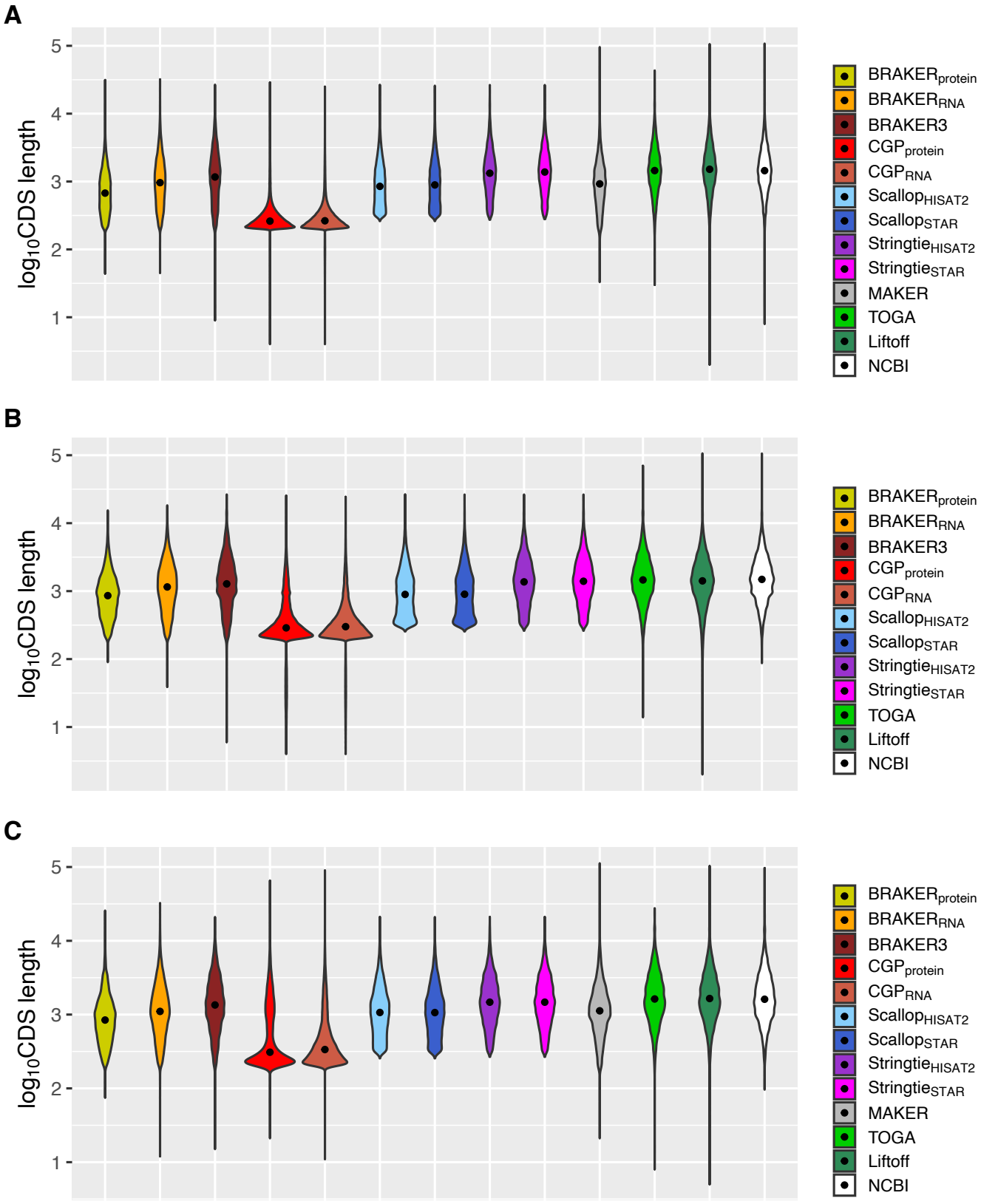
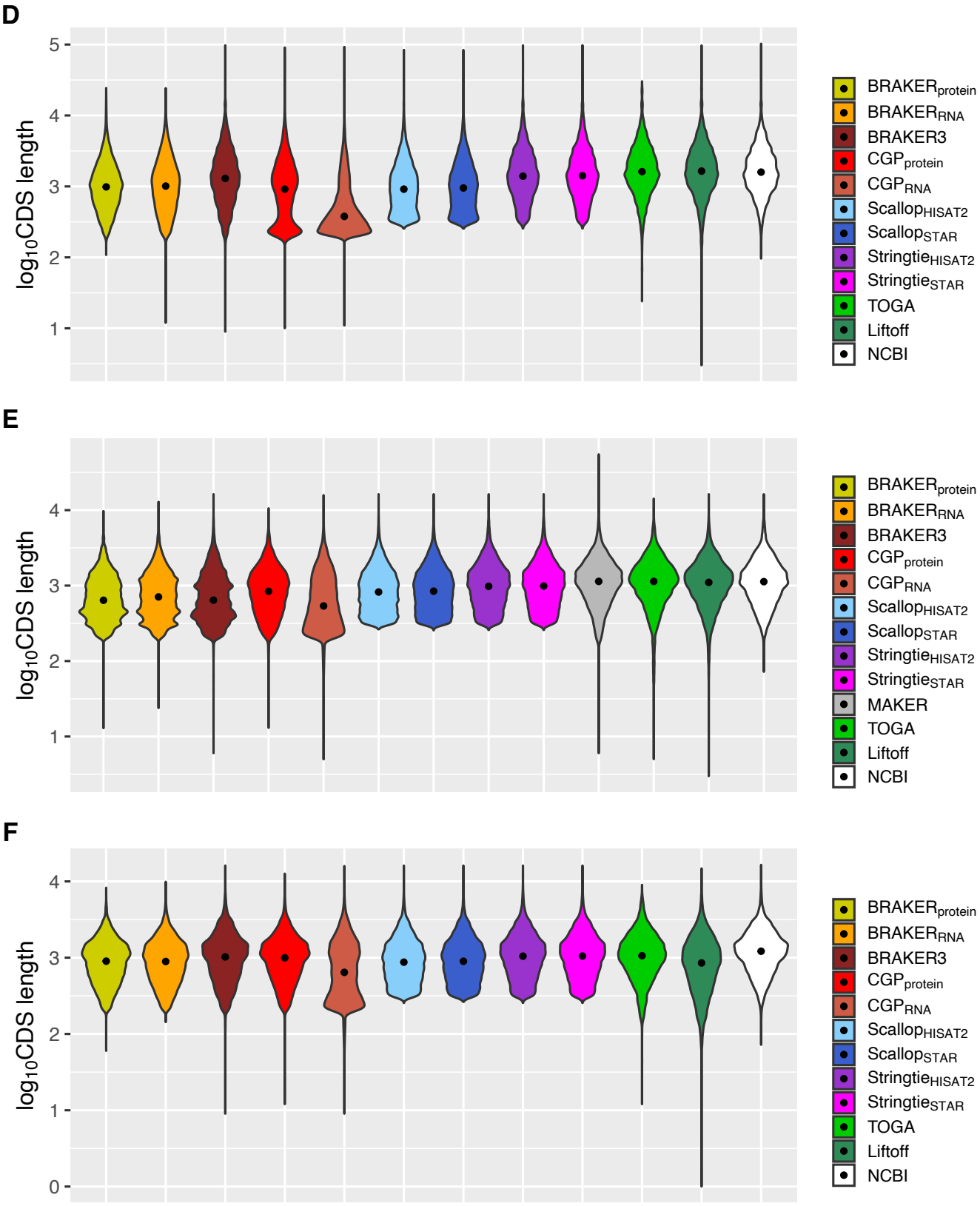
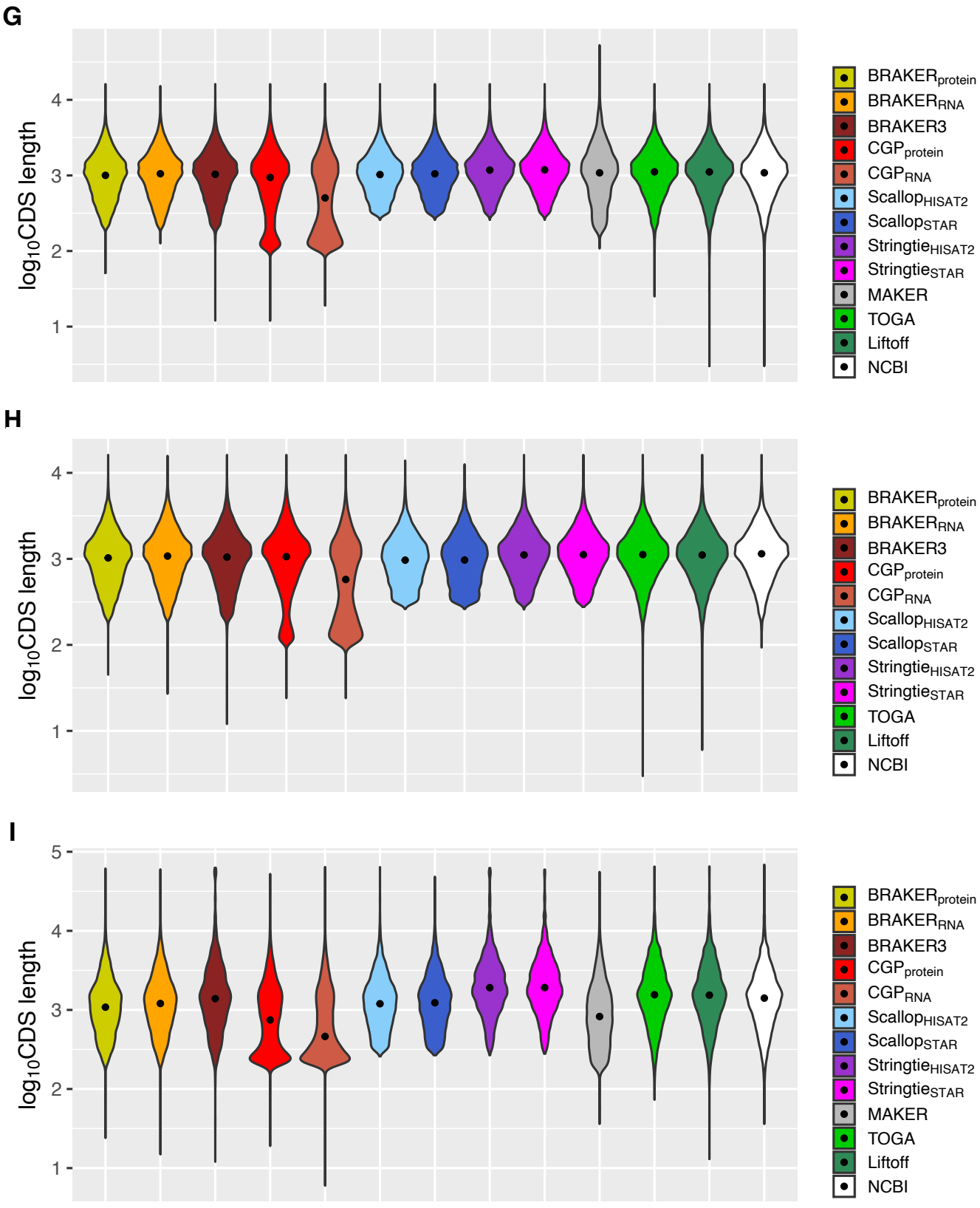


Figure S2. Joint distributions of number of predicted CDS over median predicted CDS length for (A) dipterans, (B) rosids, and (C) birds, (D) monocots and (E) mammals.


A
B
C
log10CDS length
5
4
3
2
1
BRAKERprotein
BRAKERRNA
BRAKER3
CGPprotein
CGPRNA
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
MAKER
TOGA
Liftoff
NCBI


D
E
F
log10CDS length
BRAKERprotein
BRAKERRNA
BRAKER3
CGPprotein
CGPRNA
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
MAKER
TOGA
Liftoff
NCBI


G
H
I
log10CDS length
BRAKERprotein
BRAKERRNA
BRAKER3
CGPprotein
CGPRNA
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
MAKER
TOGA
Liftoff
NCBI

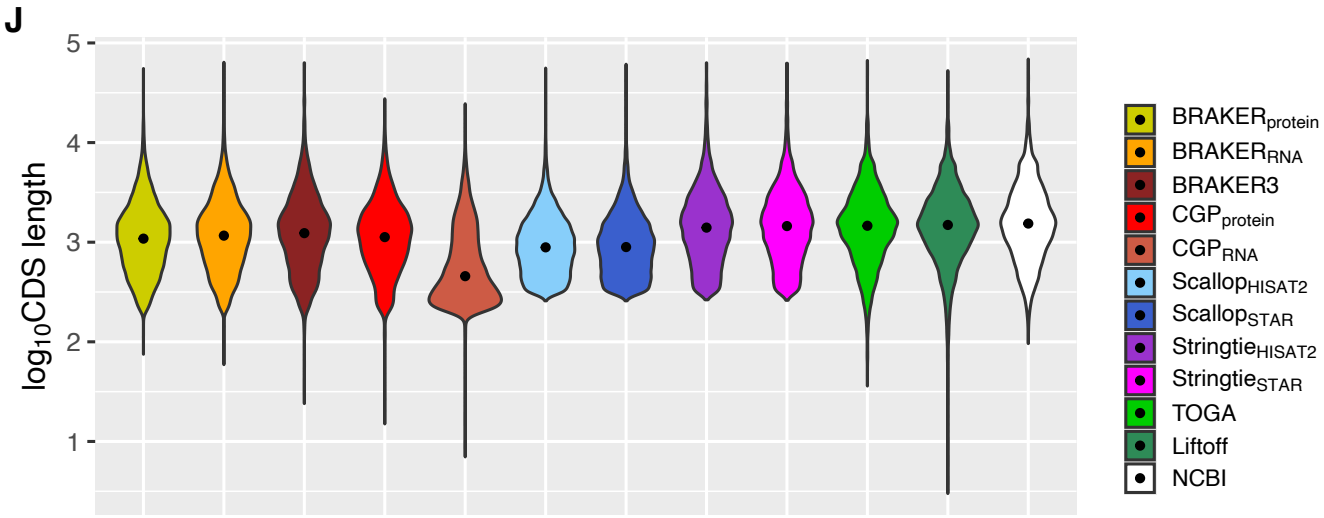


Figure S3. Violin plots of CDS length distributions for annotation methods and NCBI benchmark for (A) *H. sapiens*, (B) *C. familiaris*, (C) *G. gallus*, (D) *A. platyrhynchos*, (E) *Z. mays*, (F) *O. sativa*, (G) *A. thaliana*, (H) *C. rubella*, (I*) D. melanogaster*, and (J) *D. pseudoobscura*.

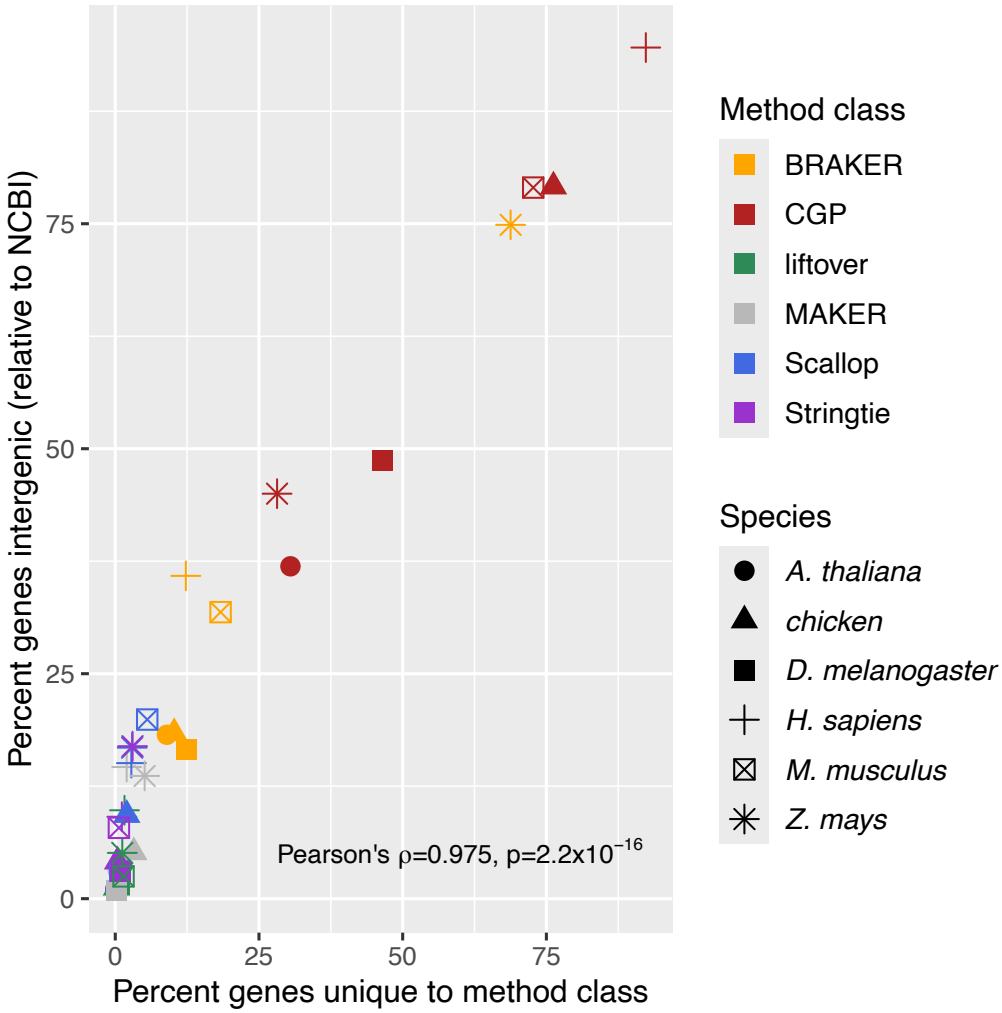
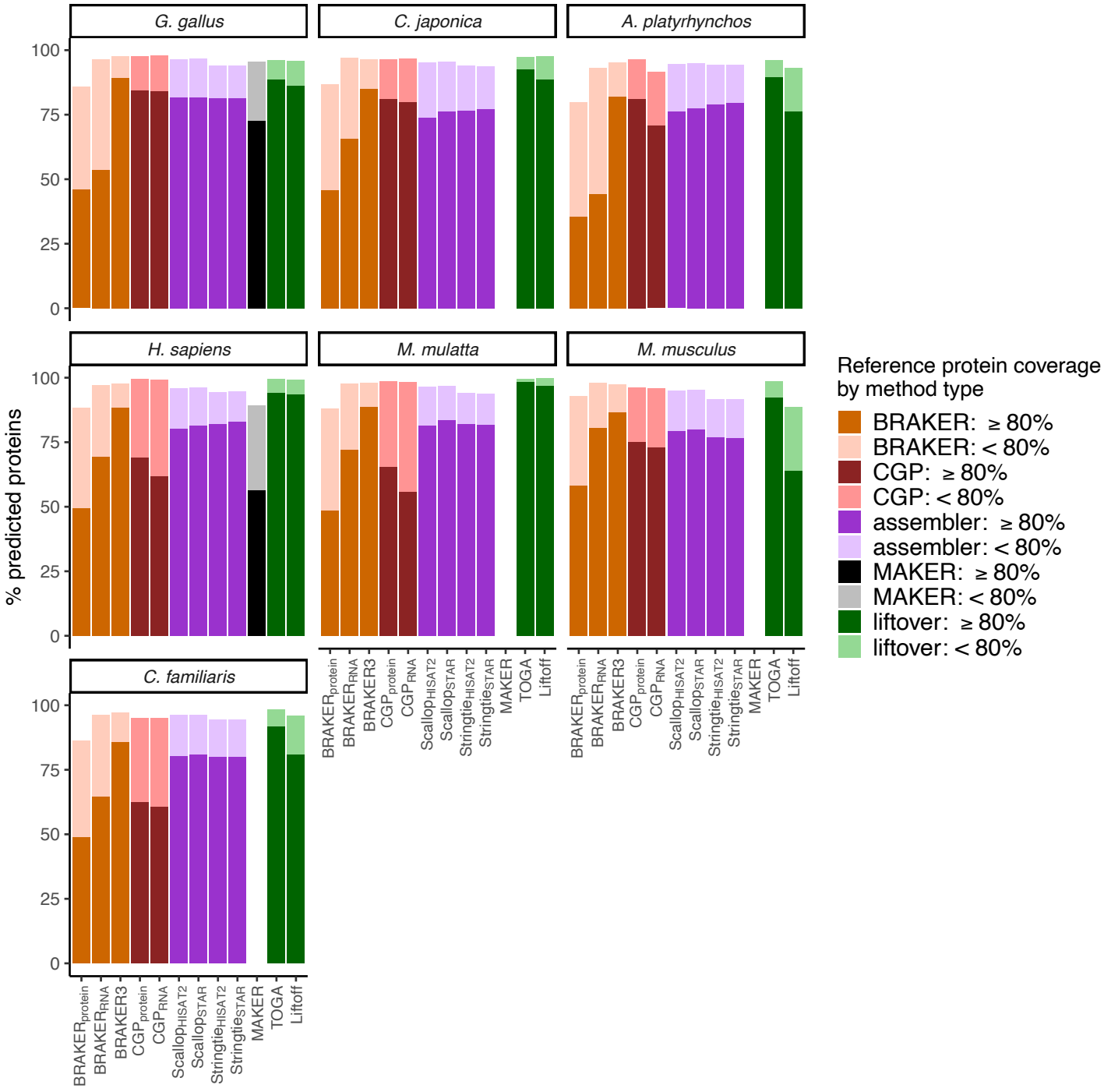
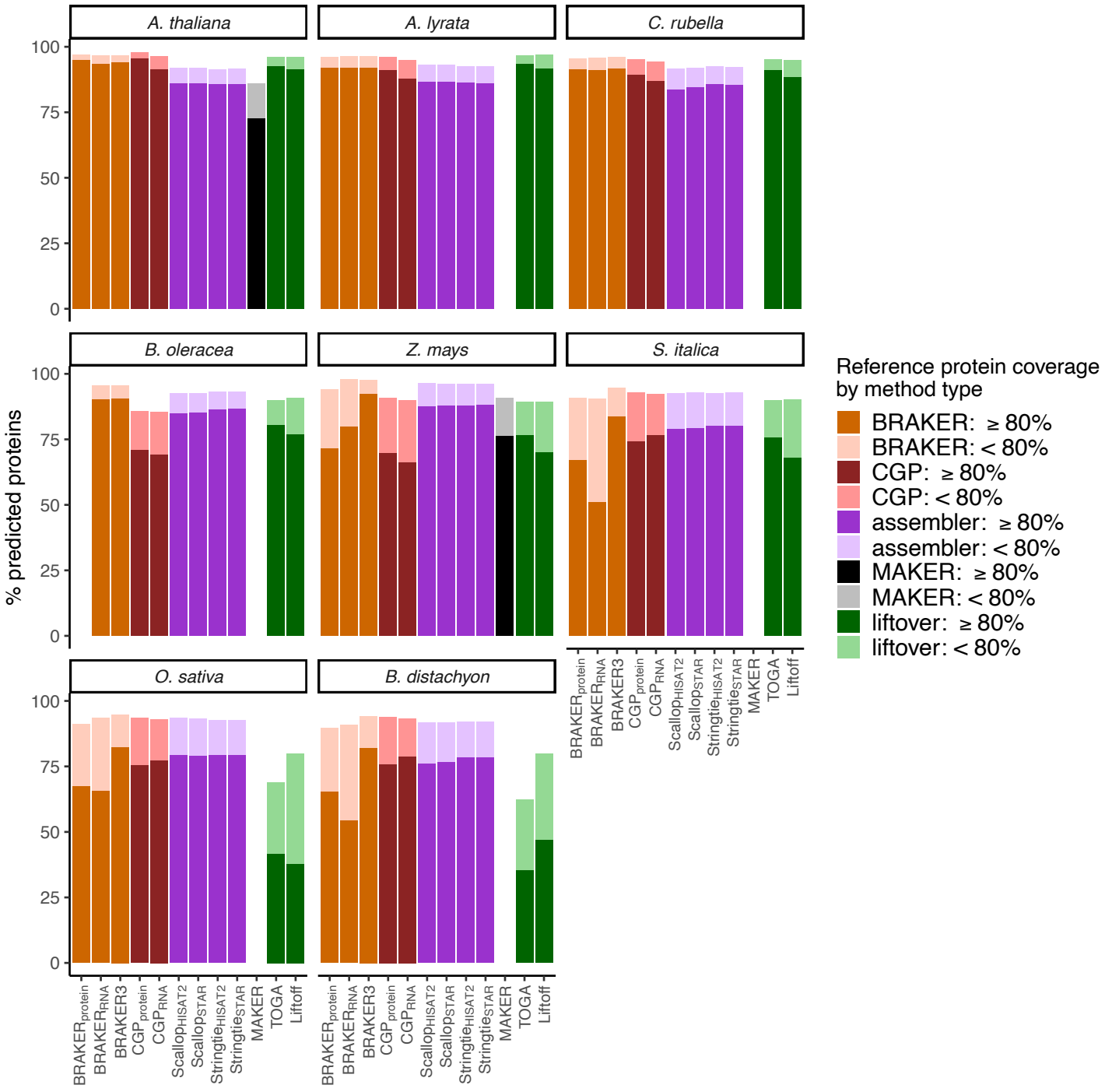


Figure S4. Correlation between the percentage of predicted genes that are unique to a particular class of annotation tool (see legend) and the proportion of genes in that class that are intergenic relative to NCBI protein-coding gene boundaries (excluding UTRs).

A


G. gallus
C. japonica
A. platyrhynchos
H. sapiens
M. mulatta
M. musculus
C. familiaris
% predicted proteins
100
75
50
25
0
BRAKER$_{protein}$
BRAKER$_{RNA}$
BRAKER3
CGP$_{protein}$
CGP$_{RNA}$
Scallop$_{HISAT2}$
Scallop$_{STAR}$
Stringtie$_{HISAT2}$
Stringtie$_{STAR}$
MAKER
TOGA
Liftoff
Reference protein coverage by method type
BRAKER: ≥ 80%
BRAKER: < 80%
CGP: ≥ 80%
CGP: < 80%
assembler: ≥ 80%
assembler: < 80%
MAKER: ≥ 80%
MAKER: < 80%
liftover: ≥ 80%
liftover: < 80%

B


A. thaliana
A. lyrata
C. rubella
B. oleracea
Z. mays
S. italica
O. sativa
B. distachyon
% predicted proteins
100
75
50
25
0
BRAKER protein
BRAKER RNA
BRAKER3
CGP protein
CGP RNA
Scallop HISAT2
Scallop STAR
Stringtie HISAT2
Stringtie STAR
MAKER
TOGA
Liftoff
Reference protein coverage by method type
BRAKER: ≥ 80%
BRAKER: < 80%
CGP: ≥ 80%
CGP: < 80%
assembler: ≥ 80%
assembler: < 80%
MAKER: ≥ 80%
MAKER: < 80%
liftover: ≥ 80%
liftover: < 80%

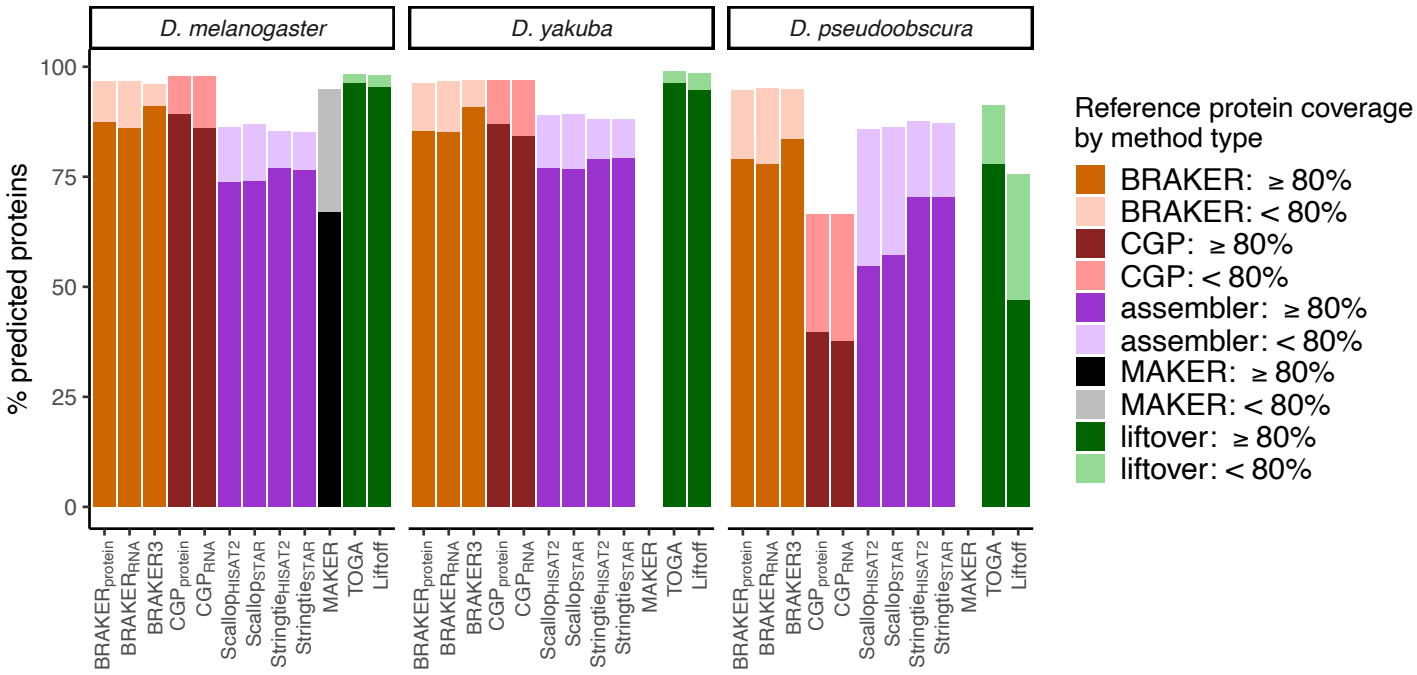


Figure S5. Blastp hit frequency by coverage of predicted proteins to NCBI proteins for (A) vertebrates, (B) plants, and (C) dipterans.

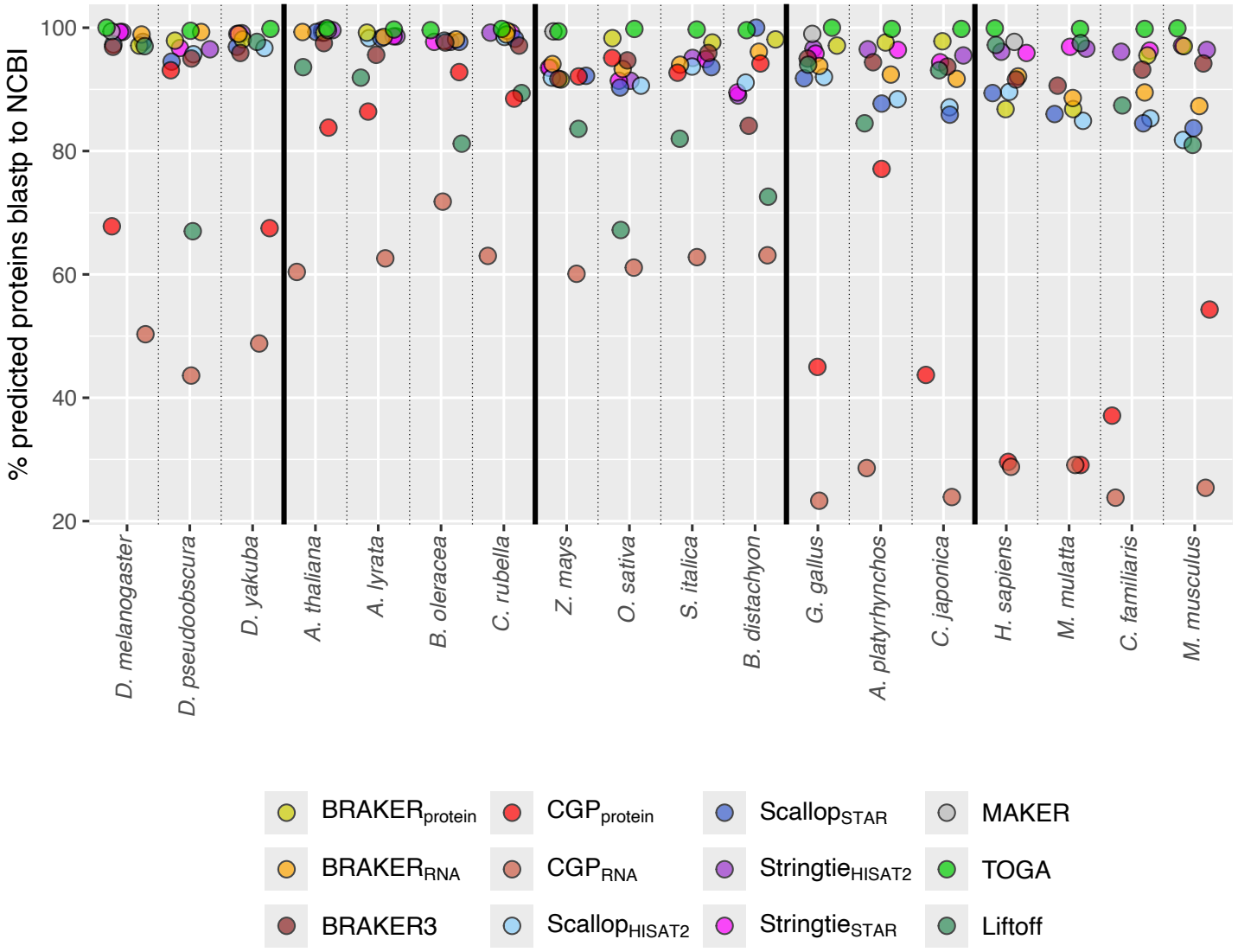
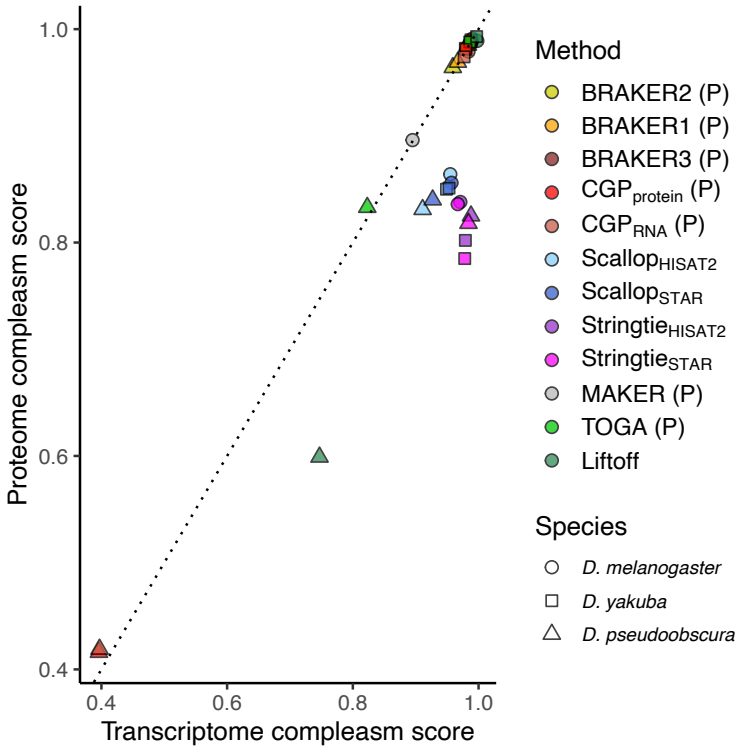
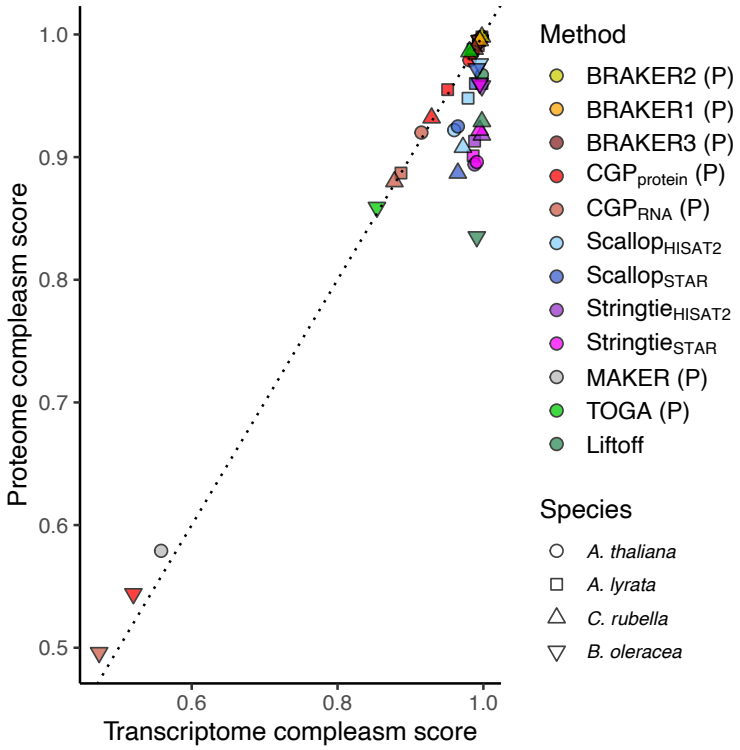


Figure S6. By species and method, percentage of predicted proteins with BLASTP hit to NCBI proteins of species in taxonomic group. Within each taxonomic group, species are ordered from left to right in order of increasing divergence from the reference species.

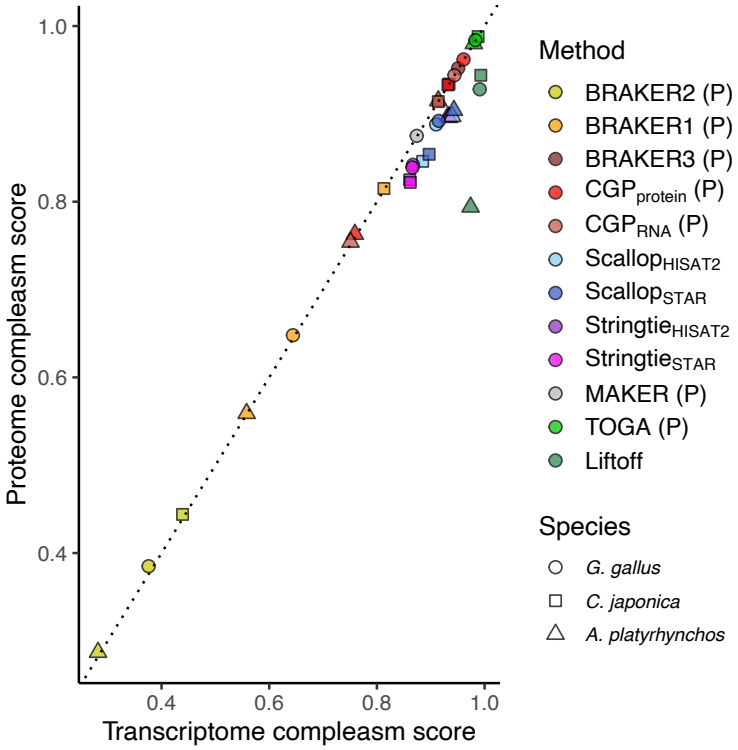
A


Proteome compleasm score
Transcriptome compleasm score
1.0
0.8
0.6
0.4
0.4
0.6
0.8
1.0
Method
BRAKER2 (P)
BRAKER1 (P)
BRAKER3 (P)
CGP protein (P)
CGP RNA (P)
Scallop HISAT2
Scallop STAR
Stringtie HISAT2
Stringtie STAR
MAKER (P)
TOGA (P)
Liftoff
Species
D. melanogaster
D. yakuba
D. pseudoobscura

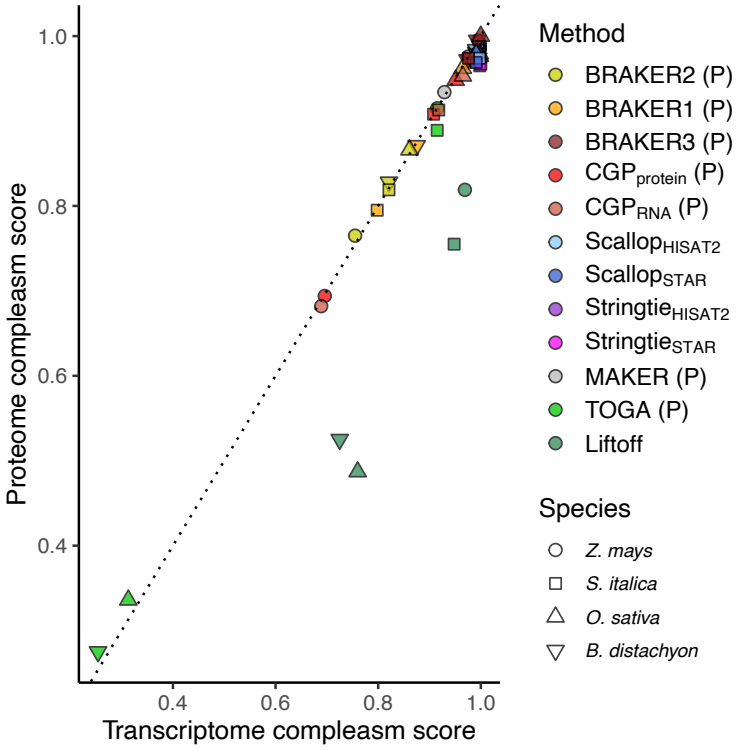
B


Proteome compleasm score
Transcriptome compleasm score
1.0
0.9
0.8
0.7
0.6
0.5
0.6
0.8
1.0
Method
BRAKER2 (P)
BRAKER1 (P)
BRAKER3 (P)
CGP protein (P)
CGP RNA (P)
Scallop HISAT2
Scallop STAR
Stringtie HISAT2
Stringtie STAR
MAKER (P)
TOGA (P)
Liftoff
Species
A. thaliana
A. lyrata
C. rubella
B. oleracea

C


Proteome compleasm score
Transcriptome compleasm score
1.0
0.8
0.6
0.4
0.4
0.6
0.8
1.0
Method
BRAKER2 (P)
BRAKER1 (P)
BRAKER3 (P)
CGP$_{protein}$ (P)
CGP$_{RNA}$ (P)
Scallop$_{HISAT2}$
Scallop$_{STAR}$
Stringtie$_{HISAT2}$
Stringtie$_{STAR}$
MAKER (P)
TOGA (P)
Liftoff
Species
G. gallus
C. japonica
A. platyrhynchos

D


Proteome compleasm score
Transcriptome compleasm score
1.0
0.8
0.6
0.4
0.4
0.6
0.8
1.0
Method
BRAKER2 (P)
BRAKER1 (P)
BRAKER3 (P)
CGP$_{protein}$ (P)
CGP$_{RNA}$ (P)
Scallop$_{HISAT2}$
Scallop$_{STAR}$
Stringtie$_{HISAT2}$
Stringtie$_{STAR}$
MAKER (P)
TOGA (P)
Liftoff
Species
Z. mays
S. italica
O. sativa
B. distachyon

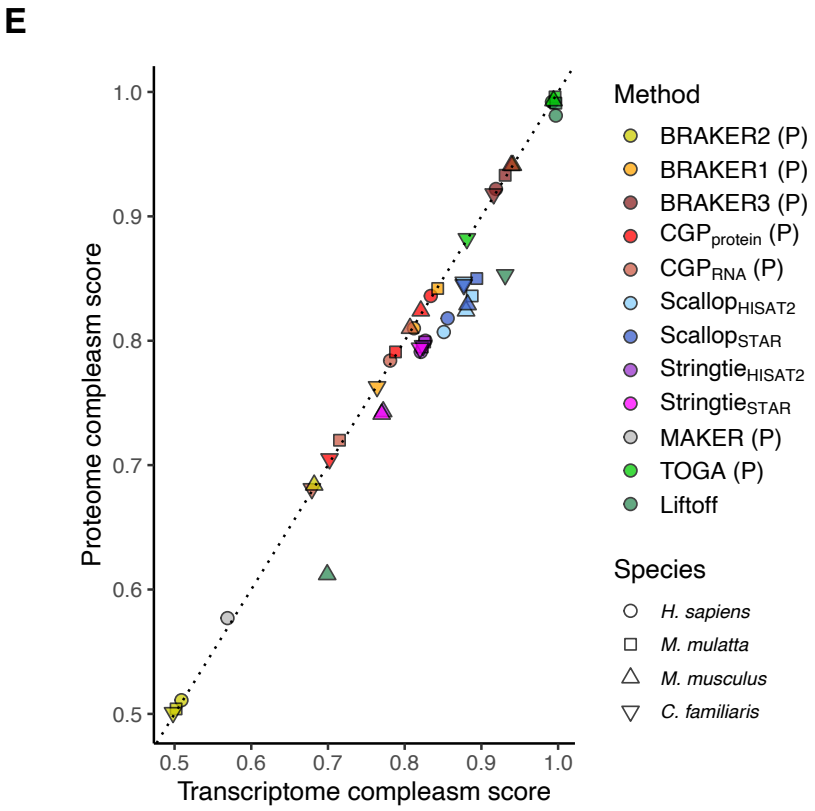
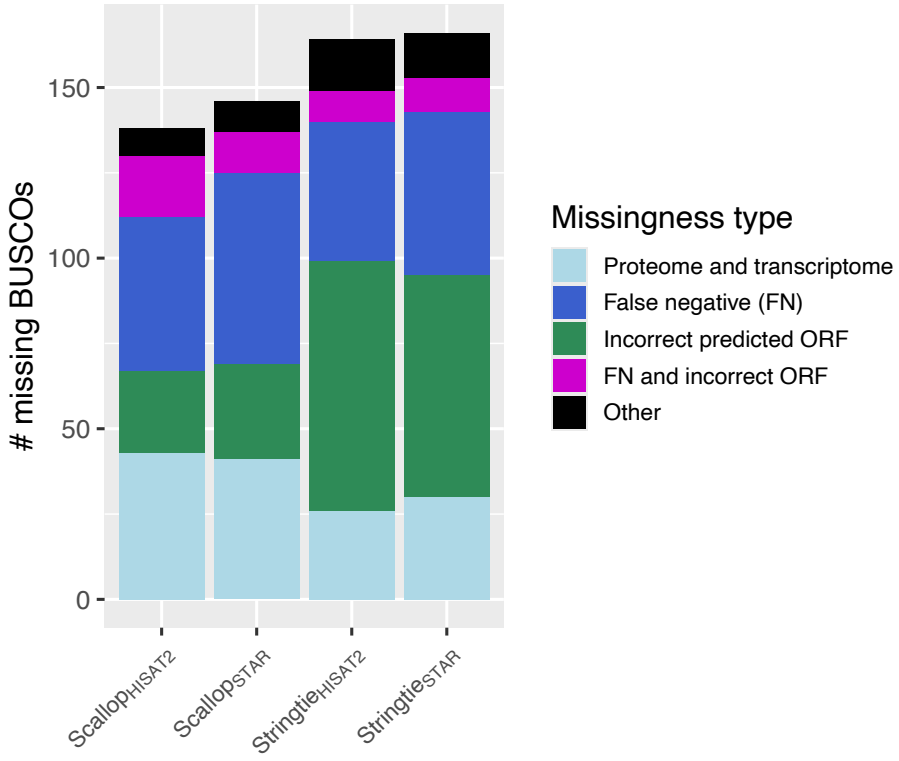
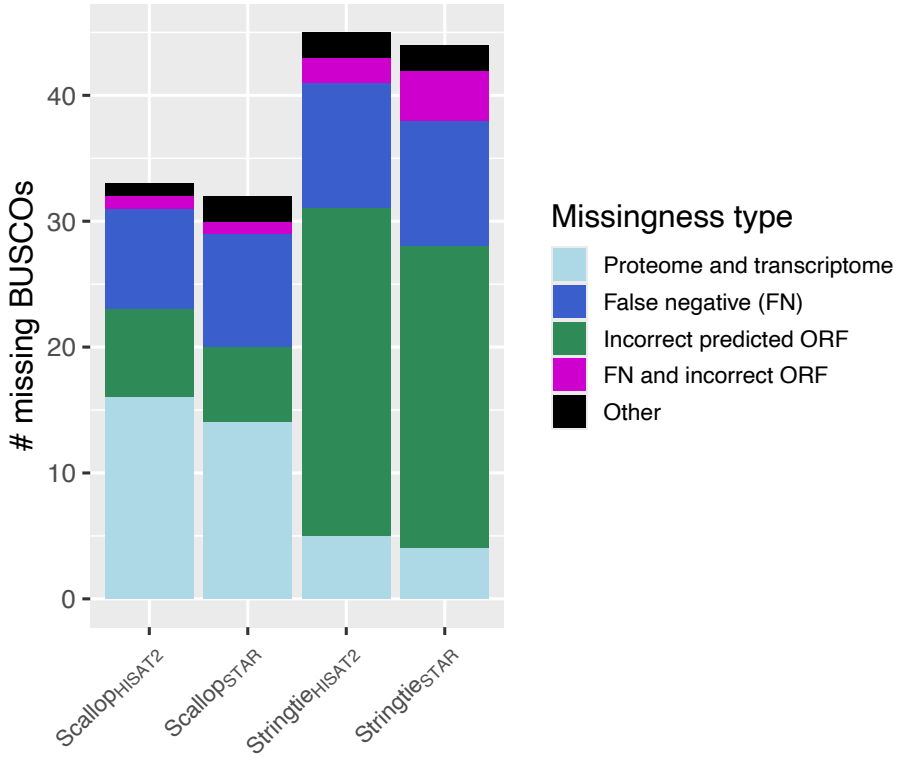


Figure S7. Correlations between proteome and transcriptome compleasm scores for (A) dipterans, (B) rosids, (C) birds, (D) monocots, and (E) mammals.

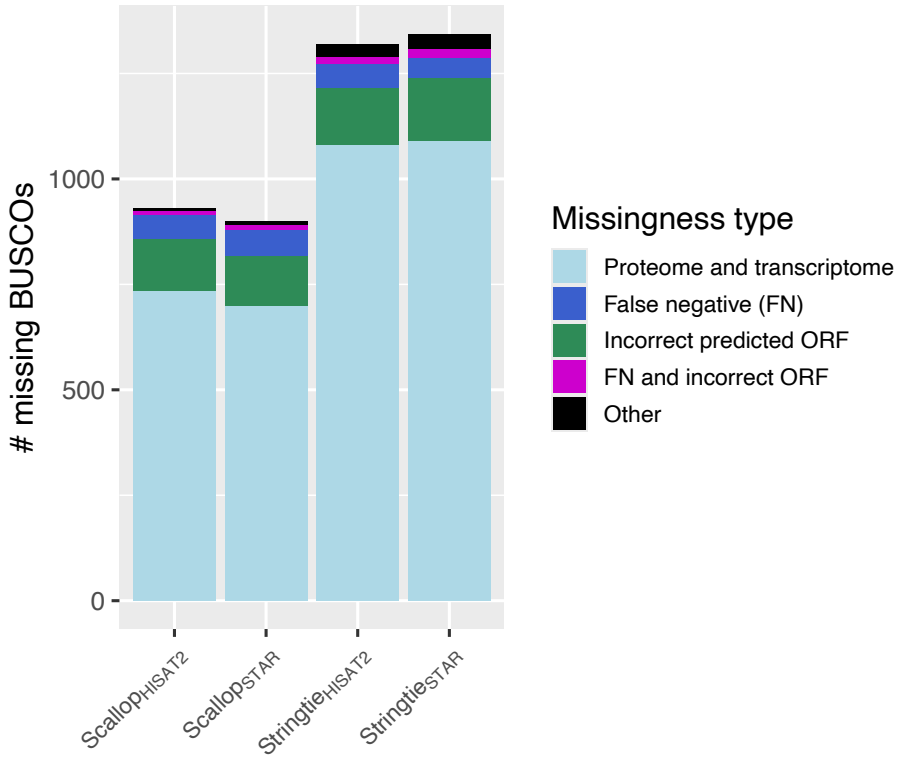
A


missing BUSCOs
150
100
50
0
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
Missingness type
Proteome and transcriptome
False negative (FN)
Incorrect predicted ORF
FN and incorrect ORF
Other

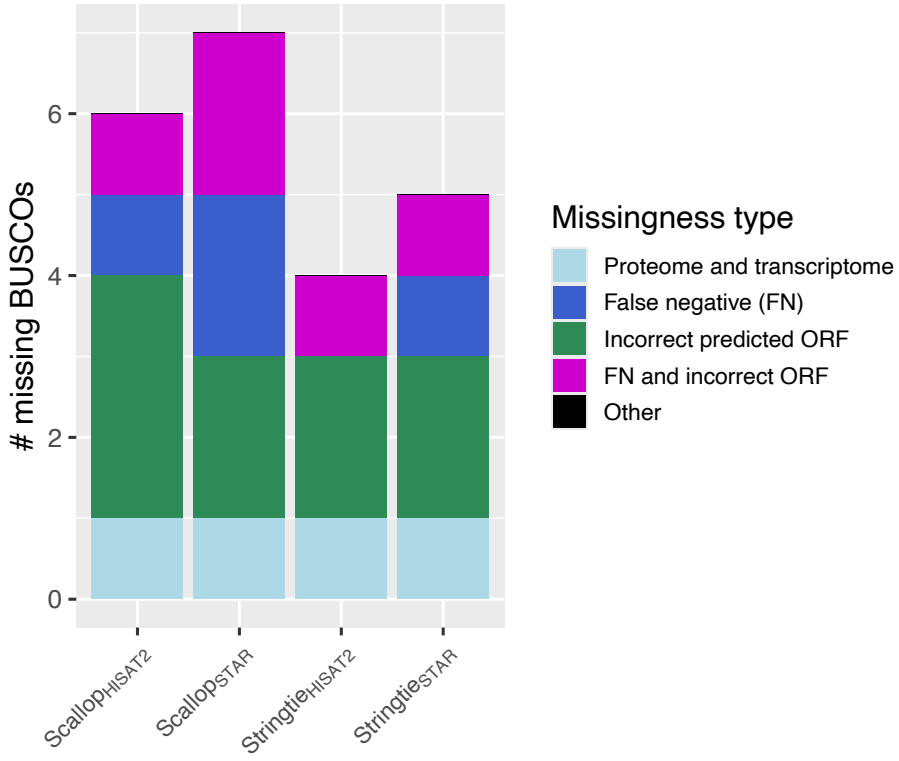
B


missing BUSCOs
40
30
20
10
0
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
Missingness type
Proteome and transcriptome
False negative (FN)
Incorrect predicted ORF
FN and incorrect ORF
Other

C


missing BUSCOs
1000
500
0
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
Missingness type
Proteome and transcriptome
False negative (FN)
Incorrect predicted ORF
FN and incorrect ORF
Other

D


missing BUSCOs
6
4
2
0
ScallopHISAT2
ScallopSTAR
StringtieHISAT2
StringtieSTAR
Missingness type
Proteome and transcriptome
False negative (FN)
Incorrect predicted ORF
FN and incorrect ORF
Other

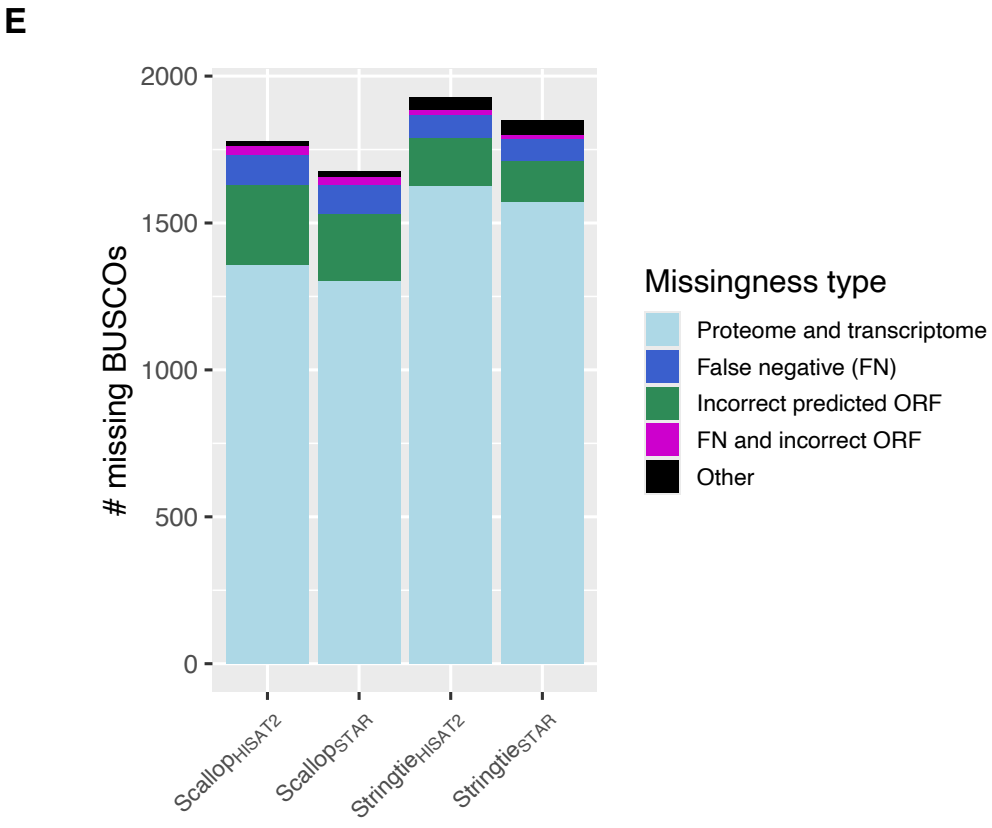


Figure S8. Sources of BUSCO missingness in RNA-seq assembler proteomes for (A) *D. melanogaster*, (B) *A. thaliana*, (C) *G. gallus*, (D) *Z. mays*, and (E) *H. sapiens*.

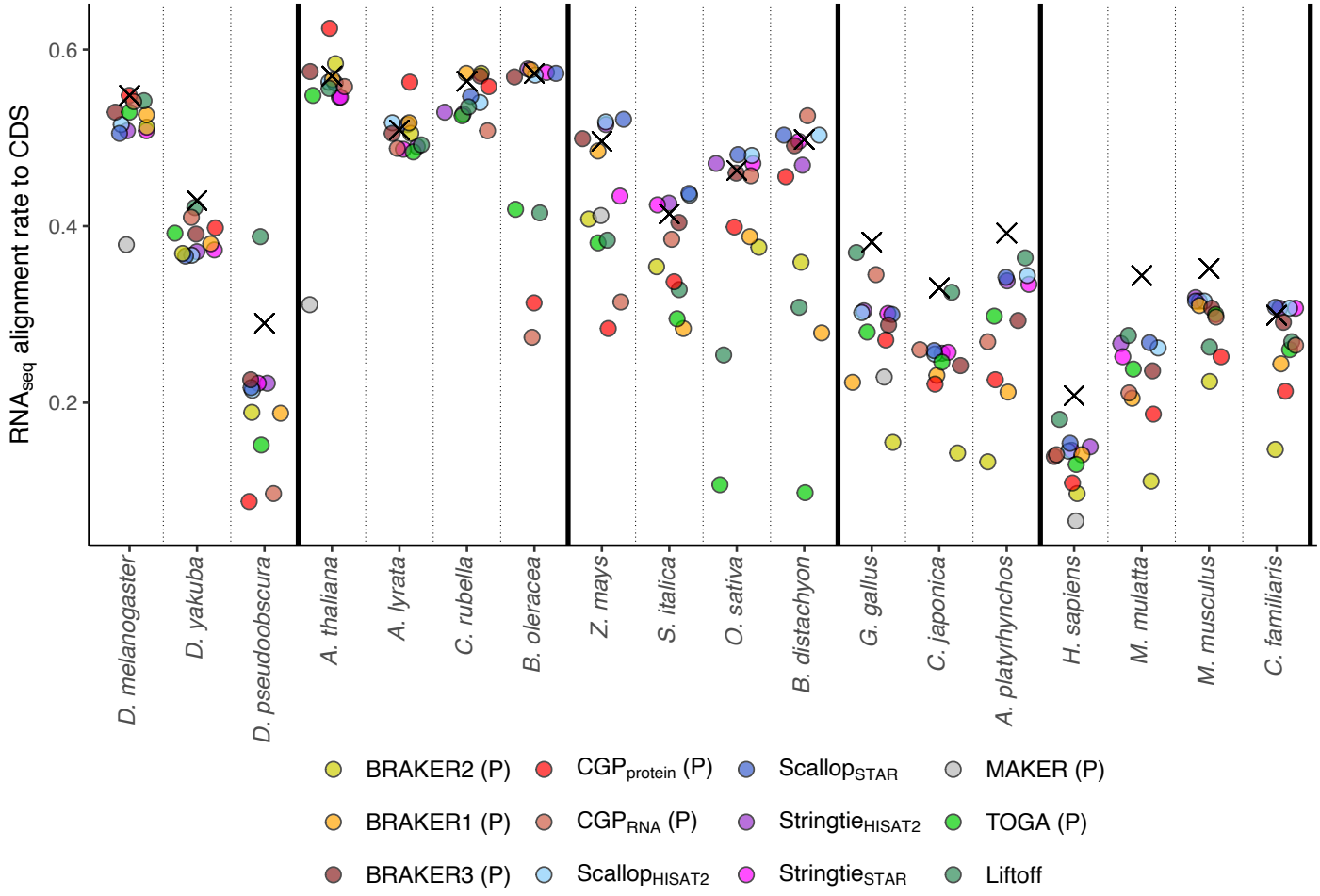


Figure S9. RNA-seq alignment rate to proteomes (i.e. CDS sequences). Species are ordered from left to right in order of increasing divergence from the reference species. Xs indicate the alignment rate to CDS of the NCBI annotation.

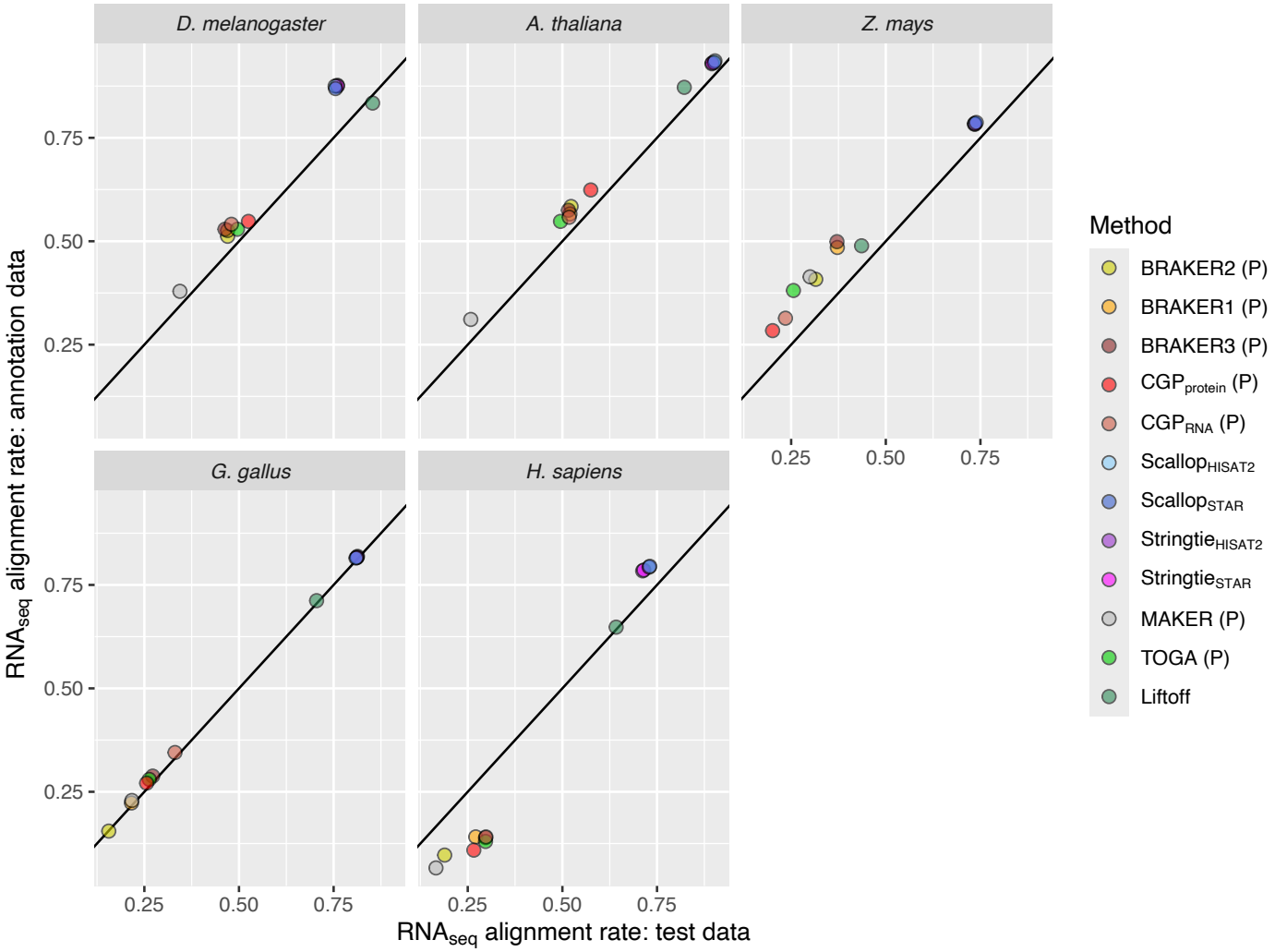


Figure S10. Correlations between RNA-seq alignment rates to annotations for samples used to generate annotations, versus a separate set of test samples that were not used in generating annotations.

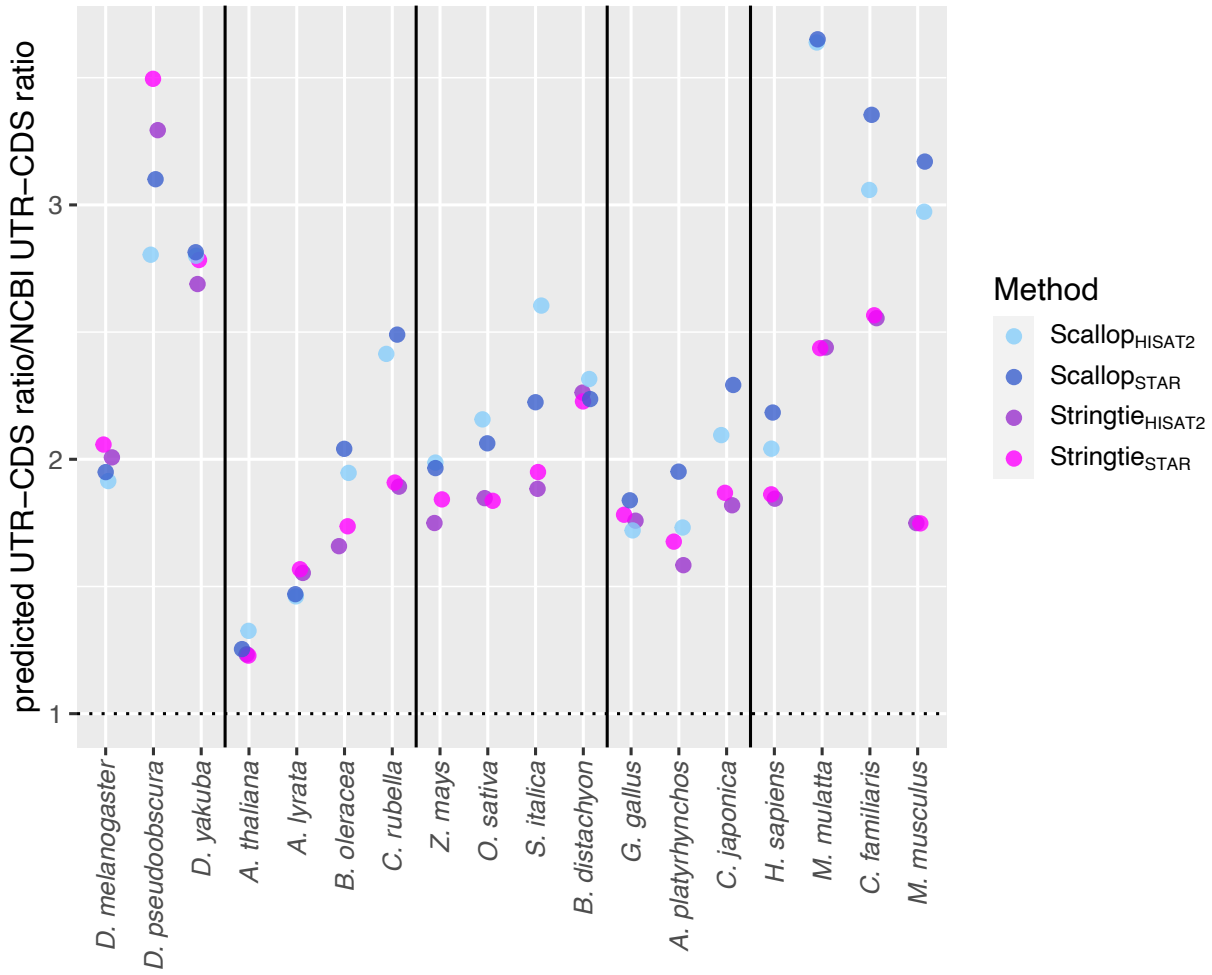


Figure S11. Ratios of the ratio of the median predicted UTR length/median predicted CDS length for RNA-seq assemblers over that for NCBI protein coding transcripts. No clear increase is observed for the most complete and curated annotations (*H. sapiens*, *D. melanogaster*, *G. gallus*, *Z. mays*, *A. thaliana*) relative to other genomes, indicating that the filtering out of cases of transcriptional readthrough (of the sort that NCBI will filter out for high quality annotations) does not explain the reduction in alignment rates for RNA-seq assemblers when UTRs are not included.

Supplemental Results S1: Support for using NCBI as truth set for calculation of intergenic false positive rate for predicted protein-coding genes

We assume that predictions that are specific to a particular class of annotation method (i.e. CGP, BRAKER, StringTie+TransDecoder, Scallop+TransDecoder, liftover, and MAKER) are more likely to be false positives than those that overlap across multiple classes of methods. Thus, if the "unique-to-method class" gene prediction frequency for our reference species (and *M. musculus*) is highly correlated with the class-level false positive rate, we would take this as evidence that the intergenic false positive rate calculated for individual methods relative to the NCBI annotation is a good approximation to the real false positive rate. Consistent with our expectations, the percent of genes unique to a method class was highly correlated with the percentage of intergenic gene predictions for that class (Pearson's ρ=0.975, p=2.2$\times$ 10^{-16}; Supplemental Fig. S4).

Supplemental Results S2: transcriptome RNA-seq alignment rate, training vs. test data

Training and test data alignment rates were strongly correlated (Supplemental Fig. S10). For RNA-seq assemblers, for four of five reference species there were modest reductions in alignment rate with test data compared to the data used to assemble transcripts, but in three of five species similar alignment rate differences were also observed for other methods including those that didn't use RNA-seq data (Supplemental Fig. S10).