EukProt_ProtIDs_to_UniProt

August 24, 2022

1 Obtaining UniProt IDs from disparate data sources in EukProt

---

This notebook walks through the process of taking protein IDs from EukProt and corresponding them with their respective UniProt ID wherever possible. We then use these to extract relavant metadata, namely gene ontologies, and subcellur localization. In general, EukProt is a very useful collection of proteomes, but includes data from disparate sources with varying annotation styles/resolution. For instance, the proteomes of many species in EukProt are derived from transcriptome assemblies and thus lacking specific protein accessions. In contrast, other proteomes are sourced from RefSeq or ensemble, and thus have standing protein accessions, but these can vary in format to a frustrating degree, necessitating a fair amount of coaxing into a usable format. We are using UniProt accessions as they are used by AlphaFold2 to name individual protein structure prediction files, and because these accessions are readily used to pull our other functionally relevant details about each protein.

[ ]:
```
%load_ext rpy2.ipython
```

Let's begin by loading in metadata for the taxon-set of interest, "The Comparative Set", a set of 196 Eukaryotes sampled by the authors to maximize taxonomic breadth and completeness data. In general, the "Data_Source_URL" and "Data_Source_Types" columns are very informative, and can be used to quickly determine from which source these protein IDs originated from. Additionally, if the data type is anything other than genome (e.g. transcriptome), the individual proteins will not have useful accessions/IDs.

So, first pull out the taxa for which we can most likely correspond protein IDs with UniProt accessions - those for which the data type is 'genome' and "Actions_Prior_to_Use" is 'none'.

[ ]:
```
%%R
library(tidyverse)
metadat <- as.data.frame(read_tsv('~/environment/data/EukProt/TCS/
 ↪TCS_EukProt_included_data_sets.v03.2021_11_22.txt'))

metadat <-
    metadat[which(metadat$Data_Source_Type == "genome" &
```

```
                    metadat$Actions_Prior_to_Use == "none"),]
```

[3]:
```
%%bash
# General syntax for this (and an example) is below.
# The syntax (and an example) for programatically mapping IDs between databases␣
 ↪using uniprot is the following:
curl --form 'from=RefSeq_Protein' --form 'to=UniProtKB' --form 'ids=XP_644532.
 ↪1' https://rest.uniprot.org/idmapping/run
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   408    0    52  100   356     78    535 --:--:-- --:--:-- --:--:--   613
```

```
{"jobId":"19fcde4de89e1eb4c289a2658adb1c90753a1688"}
```

[16]:
```
%%bash
# 'from' in 'from=RefSeq_Protein' describes the database where your protein ID␣
 ↪originated
# 'to' in 'to=UniProtKB is what we want to map it to (here UniProtKB)
# 'ids=XP_644532.1' is the actual ID (or IDs) from RefSeq_Protein that we want␣
 ↪to map
# 'https://rest.uniprot.org/idmapping/run' is where we are sending these␣
 ↪strings to - specifying we want to run the read mapping.

# In general, we can make these queries more programatic by storing the 'from'␣
 ↪and 'ids' as variable, and spitting the output Job ID into
# a temporary file that can then be read in as a variable. Then, we can read␣
 ↪through a set of proteins and programatically query them if
# we know their data source, and then download the results.

echo "Store in variables"
accType=RefSeq_Protein
accID=XP_644532.1

echo "Submit to UniProt for ID mapping"
curl --form "from=${accType}" --form "to=UniProtKB" --form "ids=${accID}" https:
 ↪//rest.uniprot.org/idmapping/run > $accID.txt
sleep 10

echo  "store the Job ID as a variable and then download the results, spitting␣
 ↪out to file"
jobID=$(cat $accID.txt | sed "s/.*://g" | sed "s/}//g" | sed 's/"//g')
curl -s "https://rest.uniprot.org/idmapping/uniprotkb/results/stream/$jobID" >␣
 ↪$accID.txt

cat $accID.txt
```

```
# The "rules" are described below. When the "Data_Source_URL" column of the␣
 ↪metadata contains:
# 1) "ncbi.nlm.nih.gov/genomes/all/GCF/": Then specify --form␣
 ↪'from=RefSeq_Protein'
# 2) "ncbi.nlm.nih.gov/genomes/all/GCA/": Then specify --form␣
 ↪'from=EMBL-GenBank-DDBJ_CDS'
# 3) "ensemblgenomes.org/pub/protists/": Then specify --form␣
 ↪'from=Ensembl_Genomes_Protein'
#    - unfortunately other repositories other than the protist one seems to not␣
 ↪be queryable.
# 4) Anything not from RefSeq/GenBank/Ensemble (e.g. figshare) has bespoke␣
 ↪identifiers, and either are
#    lacking annotations, or cannot readily be corresponded to other standard␣
 ↪identifiers.
```

```
Store in variables
Submit to UniProt for ID mapping
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   408    0    52  100   356     81    554 --:--:-- --:--:-- --:--:--   635
```

```
store the Job ID as a variable and then download the results, spitting out to
file
```

[17]:
```
%%bash
accID=XP_644532.1
accType=RefSeq_Protein

curl --form "from=${accType}" --form "to=UniProtKB" --form "ids=${accID}" https:
 ↪//rest.uniprot.org/idmapping/run > $accID.txt
jobID=$(cat $accID.txt | sed "s/.*://g" | sed "s/}//g" | sed 's/"//g')
rm $accID.txt

empty=1
while [ $empty == 1 ]
do
    # Wait a couple seconds
    sleep 2
    curl -s "https://rest.uniprot.org/idmapping/uniprotkb/results/stream/
 ↪$jobID" >> Species-UniProt-Protein-IDs.txt

    # Test whether the file storing the UniProt results is empty or not. If so␣
 ↪we'll keep going, otherwise we stop.
    if [[ ! -f Species-UniProt-Protein-IDs.txt || ! -s␣
 ↪Species-UniProt-Protein-IDs.txt ]]
    then
```

```
        empty=1
    else
        empty=0
    fi
done
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   408    0     52  100   356     85    582 --:--:-- --:--:-- --:--:--   666
```

[5]:
```
%%R
# So, because there are only a small number of data urls that are useful for␣
 ↪us, we can sort the remaining species into these subsets.
refseq_spp <-
    metadat[which(str_detect(metadat$Data_Source_URL, "ncbi.nlm.nih.gov/genomes/
 ↪all/GCF/")),]
refseq_spp$orig <- 'RefSeq_Protein'

genbank_spp <-
    metadat[which(str_detect(metadat$Data_Source_URL, "ncbi.nlm.nih.gov/genomes/
 ↪all/GCA/")),]
genbank_spp$orig <- 'EMBL-GenBank-DDBJ_CDS'

ensembl_spp <-
    metadat[which(str_detect(metadat$Data_Source_URL, "ensemblgenomes.org/pub/
 ↪protists/")),]
ensembl_spp$orig <- 'Ensembl_Genomes_Protein'


# Now, as I get into more details below, Ensembl IDs are the worst, and seem to␣
 ↪follow no consistent pattern as for which are readily queryable
# using UniProt's ID mapping tool. So, we will unfortunately have to manually␣
 ↪pull out the species in EukProt for which we may correspond the data
ensembl_spp <-
    ensembl_spp[which(ensembl_spp$EukProt_ID %in% c('EP00006', 'EP00473',␣
 ↪'EP00643')),]

# And while we're at it, write these out to file. We can then look through the␣
 ↪species later on to get the final list of species that we can query against␣
 ↪UniProt.
refseq_spp <- as.matrix(refseq_spp[,c('Name_to_Use', 'orig')])
genbank_spp <- as.matrix(genbank_spp[,c('Name_to_Use', 'orig')])
ensembl_spp <- as.matrix(ensembl_spp[,c('Name_to_Use', 'orig')])

dimnames(refseq_spp) <- NULL
dimnames(ensembl_spp) <- NULL
```

```
dimnames(genbank_spp) <- NULL

write.table(refseq_spp,
            file = 'TCS-RefSeq-Species.tsv', quote = F,
            col.names = F, row.names = F)
write.table(genbank_spp,
            file = 'TCS-GenBank-Species.tsv', quote = F,
            col.names = F, row.names = F)
write.table(ensembl_spp,
            file = 'TCS-Ensembl-Species.tsv', quote = F,
            col.names = F, row.names = F)
```

1.0.1 There are (annoyingly) exceptions to these rules unfortunately. So, the general format of each can be used to parse out which of these break the rules. Ensemble datasets seem to be the worst examples.

For instance:

- Phytopythium vexans's data source is the same as above (the ensemble protist dataset), but for whatever the format of the protein ID (e.g. EPrPV00000013083) isn't query-able by UniProt.

1.0.2 So, below are examples of each accession, and the patterns they should follow for things to play nicely with UniProt.

1) RefSeq_Protein: - These IDs should have easy to find prefixes - either XP_ or NP_ in the EukProt set. XM_ is also possible.

2) EMBL-GenBank-DDBJ_CDS: - These IDs follow the pattern of 3 capital letters, followed by 5 digits, and a "version number" (i.e. a suffix of ".1" or ".2"). - But, there are still cases where this naming convention isn't followed. For instance, Neovahlkampfia damariscottae has proteins named along the lines of "gene1-4252_t" etc. This doesn't correspond to actual proteins. - In another case, Carpediemonas membranifera has IDs with three letters followed by 7 digits before the accession version. These are real, and can be corresponded. - So, just check that it follows a convention along the lines of AAA00000.1, etc, otherwise ignore. It does seem that if the column "Actions_Prior_to_Use" is anything other than "none", it is unlikely that there are usable accessions - these exceptions can probably be safely be ignored.

3) Ensembl_Genomes_Protein: - This is an annoying set. Little rhyme or reason in the naming 'convention', and lots of exceptions. - Often, IDs follow a pattern of 3 capital letters followed by 5 digits - that's all. - But for Hyaloperonospora arabidopsidis, it instead is "HpaP803016" - Related to the above, protein IDs from the 'ensemblgenomes.org/pub/metazoa/' (rather than the protist set: ensemblgenomes.org/pub/protists/) data source seems to have IDs that are not queryable using UniProt. Annoying. - But, if you thought the protist database was foolproof, you're mistaken - Phytopythium vexans is in the protist set, and for this species the IDs are along the lines of 'EPrPV00000013083' - and this returns nothing when querying to UniProt. I don't know of any steadfast rules here.

In all cases, the ID is in the second "field" of the sequence name (space delimited - as in >EP00001_Species_genus_P00001 ProteinID etc etc etc), which makes things easier.

So, to summarize - data that is coming from RefSeq is very easy to reliably and programatically query against UniProt, as are proteins from EMBL-GenBank-DDBJ_CDS (just need to allow for variable number of trailing digits before the accession version). In contrast, data sourced from ensemble is…. chaos, and I'm not sure how best to handle that other than to manually curate the species for which we can query protein sequences for.

1.1 The general procedure will thus be to:

1.1.1 1) Generate the three different species sets (GenBank, RefSeq, and Ensemble) by:

a) Checking if the data type is anything other than 'genome' - exclude these.

b) Parsing the data source url to check if the data source includes

- "genomes/all/GCA/" (EMBL-GenBank-DDBJ_CDS) or
- "genomes/all/GCF/" (RefSeq_Protein).
- These two sources can be programatically queried.

1.1.2 2) Loop through each species, double checking that their protein ID follows the expected pattern (First three characters just need to be letters):

- **NOTE:** Let's assume that the species in the set are listed in a file, one species per line. Everything below would then be in a 'while read' loop like:

```
while read spp
do
    etc....
done < species-list.txt
```

a) If the source is "RefSeq_Protein":

- Store the species and ID as a variable... For example:

```
acc=$(grep ">" *${spp}*.fasta | head -n1 | cut -f2 -d' ')
```

- Check if the first three characters are one of "XP_" or "NP_":
- return 1 if yes, 0 if no
- This may be accomplished using the following:

```
prefix=${acc::3}
corr_prefix=$(if [[ "$prefix" =~ "XP_" || "$prefix" =~ "NP_" ]]; then echo 1; else echo 0 ; fi
```

- if so spit out to another file listing the species for which we may actually correspond prote

```
if [[ "$corr_prefix" == 1 ]]
then
    cat $spp >> RefSeq-UniProt-species-list.txt
fi
```

b) If the source is "EMBL-GenBank-DDBJ_CDS":

```
- Store the ID as a variable... For example:
```

```
acc=$(grep ">" example.fasta | head -n1 | cut -f2 -d' ')
```

```
- Check if the first three characters are letters: return 1 if yes, 0 if no
```

```
corr_prefix=$(if [[ ${acc::3}  =~ [0-9] ]]; then echo 0; else echo 1 ; fi)
```

```
- Check if there is an accession version (e.g. .1, .2, .3, .4, .5): return 1 if yes, 0 if no
```

```
versions=".1 .2 .3 .4 .5"
corr_acc_vers=$(if (echo "$versions" | fgrep -q "${acc: -2}"); then echo 1; else 0; fi)
```

```
- Lastly, check if the intervening characters are all numbers.
```

```
mid=${acc#"${acc::3}"}
mid=${mid%"${acc: -2}"}
corr_mid=$(if [[ $mid  =~ [0-9] ]]; then echo 1; else echo 0 ; fi)
```

```
 - Finally, check that all three conditions are true, and if so spit out to another file listi
```

```
all_true=$((corr_prefix + corr_acc_vers + corr_mid))

if [[ "$all_true" == 3 ]]
then
    echo $spp >> GenBank-UniProt-species-list.txt
fi
```

1.2 Awesome. Let's combine this now and run.

[28]:
```
%%bash
# First for RefSeq
while read spp
do
    sppFasta="/home/ubuntu/environment/data/EukProt/TCS/data/proteins/*${spp}*.
 ↪fasta"

    # Get the first accession - any will do.
    acc=$(grep ">" $sppFasta | head -n1 | cut -f2 -d' ')

    #Check if the first three characters are one of "XP_" or "NP_":
    # return 1 if yes, 0 if no
    prefix=${acc::3}
```

```
    corr_prefix=$(if [[ "$prefix" =~ "XP_" || "$prefix" =~ "NP_" ]]; then echo␣
 ↪1; else echo 0 ; fi)

    # And write to file if so.
    if [[ "$corr_prefix" == 1 ]]
    then
        echo $spp >> RefSeq-UniProt-species-list.txt
    fi
done < TCS-RefSeq-Species.tsv

# Then for GenBank
while read spp
do
    sppFasta="/home/ubuntu/environment/data/EukProt/TCS/data/proteins/*${spp}*.
 ↪fasta"

    # Get the first accession - any will do.
    acc=$(grep ">" $sppFasta | head -n1 | cut -f2 -d' ')

    # Check if the first three characters are letters: return 1 if yes, 0 if no
    corr_prefix=$(if [[ ${acc::3}  =~ [0-9] ]]; then echo 0; else echo 1 ; fi)
    # Check if there is an accession version (e.g. .1, .2, .3, .4, .5): return␣
 ↪1 if yes, 0 if no
    versions=".1 .2 .3 .4 .5"
    corr_acc_vers=$(if (echo "$versions" | fgrep -q "${acc: -2}"); then echo 1;␣
 ↪else 0; fi)

    # Lastly, check if the intervening characters are all numbers.
    mid=${acc#"${acc::3}"}
    mid=${mid%"${acc: -2}"}
    corr_mid=$(if [[ $mid  =~ [0-9] ]]; then echo 1; else echo 0 ; fi)

    # Finally, check that all three conditions are true, and if so spit out to␣
 ↪another file listing the species for which we may actually correspond␣
 ↪protein IDs to UniProt accessions.
    all_true=$((corr_prefix + corr_acc_vers + corr_mid))

    if [[ "$all_true" == 3 ]]
    then
        echo $spp >> GenBank-UniProt-species-list.txt
    fi
done < TCS-GenBank-Species.tsv

# For consistency, make an equivalent file for Ensembl, even though the species␣
 ↪set hasn't changed.
cut -f1 -d' ' TCS-Ensembl-Species.tsv > Ensembl-UniProt-species-list.txt
```

1.2.1 Great. Now, let's work through these species and pull out their protein identifiers and spit them out to file!

Basically what we'll do here is to 1) grep the sequence identifiers from their amino acid sequences and 2) pipe those into a command to pull out the second 'field' that should be the ID.

- This will be sped up using gnu-parallel to simultaneously process multiple species at once.

[2]:
```
%%bash

# Check if we've made the directory for protein ids yet, and if not create it
idDir=/home/ubuntu/environment/data/EukProt/TCS/data/protein-ids
fastaDir="/home/ubuntu/environment/data/EukProt/TCS/data/proteins"
mkdir -p $idDir

# Basically what we're doing here is running N jobs (-j N) simultaneously/in␣
 ↪parallel, where the only thing
# that is changing is the species name, which is read in from  file (e.g.␣
 ↪TCS-Ensembl-Species.tsv), where each species is on
# a line, and that species name string is specified as a variable, {}.
echo "Working on the Ensemble set. There are $(wc -l TCS-Ensembl-Species.tsv |␣
 ↪cut -f1 -d' ') species that we need to work though."
parallel -j 3 -a TCS-Ensembl-Species.tsv "grep '>' ${fastaDir}/*{}*.fasta | cut␣
 ↪-f2 -d' ' > $idDir/{}-Ensembl-ProtIDs.txt"

echo "Working on the GenBank set. There are $(wc -l␣
 ↪GenBank-UniProt-species-list.txt | cut -f1 -d' ') species that we need to␣
 ↪work though."
parallel -j 14 -a GenBank-UniProt-species-list.txt "grep '>' ${fastaDir}/*{}*.
 ↪fasta | cut -f2 -d' ' > $idDir/{}-GenBank-ProtIDs.txt"

echo "Working on the RefSeq set. There are $(wc -l RefSeq-UniProt-species-list.
 ↪txt | cut -f1 -d' ') species that we need to work though."
parallel -j 14 -a RefSeq-UniProt-species-list.txt "grep '>' ${fastaDir}/*{}*.
 ↪fasta | cut -f2 -d' ' > $idDir/{}-RefSeq-ProtIDs.txt"
```

```
Working on the Ensemble set. There are 3 species that we need to work though.
Working on the GenBank set. There are 38 species that we need to work though.
Working on the RefSeq set. There are 124 species that we need to work though.
```

1.3 Alrighty now.... Let's get some uniprot IDs!

So before, I thought I was being clever by submitting individual uniprot IDs in parallel with gnu-parallel, looping through species. Turns out that is WILDLY inefficient, and motivated by a bit of a misunderstanding of how curl works. So instead, let's give a run-down of why I though I would do this, and what we'll do instead.

1.4 *Curl: The art of submitting*

In past efforts to submit all protein accessions simultaneously for ID mapping, I (and others) ran into the dreaded error:

```
bash: /usr/bin/curl: Argument list too long
```

This is what you get when you try to submit a bash command that is obscenely long - for instance, submitting 14974 ids at once by storing them all in a variable (= ids) and submitting using '–form "ids=$ids"' This is a limitation of the shell, not curl. This can be dealt with by instead writing all of the form submissions (i.e. "from, to, and IDs") out to a file that can be specified/fed to curl using -d. So the file would read as, for example:

```
from=RefSeq_Protein&to=UniProtKB&ids=XP00001.1,XP00002.1,XP00003.1,XP00004.1,XP00005.1,XP00006
```

...and would be provided to curl and specified using -d @forms.txt. Note that when specifying a file, the file needs to be preceded by the "@" symbol.

1.4.1 Let's provide an example.

```
# Store the species as a variabel
spp=Acanthamoeba_castellanii

# read in the protein ids and store as a variable (again, one protein ID is listed per line.
ids=$(cat data/EukProt/TCS/data/protein-ids/$spp-Ensembl-ProtIDs.txt | tr '\n' ',')

# Because the last line in the protein ID file is blank, the last character will be a comma.
# You can check this:
echo ${ids: -1}

# and then easily delete
ids=${ids::-1}

# And combine the three fields and echo to a file
echo "from=Ensembl_Genomes_Protein&to=UniProtKB&ids=$ids" > $spp-form.txt

# And now submit the id mapping job with curl!
curl -d @form.txt https://rest.uniprot.org/idmapping/run
```

All the rest of the process is the same - this approach is just so, so much faster.

So, what we'll do now is to still use gnu-parallel across species, but it'll just be a single ID mapping job that we submit. The following cell generates three scripts, one for each proteome source (Ensembl, GenBank, & RefSeq), and does so by creating a unique header for each source, and then a general, source independent tail to the script. We then concatenate the source-specific headers with the source-independent tails into their full bash scripts, and clean up.

[ ]:
```
%%bash
# The below generates a set of scripts to submit the ID mapping for any␣
 ↪species, with the species name specified as a variable ("$1")

##############################################################################################
##### First Ensembl
##############################################################################################
cat << 'EOF' >> Ensembl-Header.txt
#!/bin/bash

# Store the ids as a variable:
ids=$(cat ~/environment/data/EukProt/TCS/data/protein-ids/$1-Ensembl-ProtIDs.
 ↪txt | tr '\n' ',')
ids=${ids::-1}

# Write the curl forms to a file:
echo "from=Ensembl_Genomes_Protein&to=UniProtKB&ids=$ids" > $1-form.txt

# And submit, then clean up
curl -d @$1-form.txt https://rest.uniprot.org/idmapping/run > $1.txt
jobID=$(cat $1.txt | sed "s/.*://g" | sed "s/}//g" | sed 's/"//g')
rm $1.txt
rm $1-form.txt
EOF
##############################################################################################



##############################################################################################
##### Then Genbank
##############################################################################################
cat << 'EOF' >> GenBank-Header.txt
#!/bin/bash

# Store the ids as a variable, and remove the last character (an inserted␣
 ↪comma):
ids=$(cat ~/environment/data/EukProt/TCS/data/protein-ids/$1-GenBank-ProtIDs.
 ↪txt | tr '\n' ',')
ids=${ids::-1}

# Write the curl forms to a file:
```

```
echo "from=EMBL-GenBank-DDBJ_CDS&to=UniProtKB&ids=$ids" > $1-form.txt

# And submit, then clean up
curl -d @$1-form.txt https://rest.uniprot.org/idmapping/run > $1.txt
jobID=$(cat $1.txt | sed "s/.*://g" | sed "s/}//g" | sed 's/"//g')
rm $1.txt
rm $1-form.txt
EOF
##############################################################################################



##############################################################################################
##### And lastly RefSeq
##############################################################################################
cat << 'EOF' >> RefSeq-Header.txt
#!/bin/bash

# Store the ids as a variable:
ids=$(cat ~/environment/data/EukProt/TCS/data/protein-ids/$1-RefSeq-ProtIDs.txt␣
 ↪| tr '\n' ',')
ids=${ids::-1}

# Write the curl forms to a file:
echo "from=RefSeq_Protein&to=UniProtKB&ids=$ids" > $1-form.txt

# And submit, then clean up
curl -d @$1-form.txt https://rest.uniprot.org/idmapping/run > $1.txt
jobID=$(cat $1.txt | sed "s/.*://g" | sed "s/}//g" | sed 's/"//g')
rm $1.txt
rm $1-form.txt
EOF
##############################################################################################



##############################################################################################
##### Now write the tail-end. This will be combined with the rest.
##############################################################################################

waiting=1
finished={"jobStatus":"FINISHED"}
while [ $waiting == 1 ]
do
    # Wait a few seconds
    sleep 10
    curl -i "https://rest.uniprot.org/idmapping/status/$jobID" | tail -n1 >␣
 ↪Physcomitrium_patens.status
```

```
    # Test whether the file storing the UniProt results is empty or not. If so␣
 ↪we'll keep going, otherwise we append to the list of mapped IDs and end.
    status=$(cat Physcomitrium_patens.status)
    if [[ "$status" == "$finished" ]]
    then
        waiting=1
    else
        waiting=0
        curl -s "https://rest.uniprot.org/idmapping/uniprotkb/results/stream/
 ↪$jobID?format=tsv" > Physcomitrium_patens.txt
    fi
done


cat << 'EOF' >> Mapping-check.txt
waiting=1
finished={"jobStatus":"FINISHED"}
while [ $waiting == 1 ]
do
    # Wait a few seconds
    sleep 10

    #Check if it's finished.
    curl -i "https://rest.uniprot.org/idmapping/status/$jobID" | tail -n1 > $1.
 ↪status
    status=$(cat $1.status)

    # Test whether the file storing the UniProt results is empty or not. If so␣
 ↪we'll keep going, otherwise we append to the list of mapped IDs and end.
    if [[ "$status" == "$finished" ]]
    then
        waiting=1
    else
        waiting=0
        curl -s "https://rest.uniprot.org/idmapping/uniprotkb/results/stream/
 ↪$jobID?format=tsv" > $1.txt

        mv $1.txt /home/ubuntu/environment/data/EukProt/TCS/data/protein-ids/
 ↪$1-UniProt-ProtIDs.txt
        rm $1.status
    fi
done
EOF
##############################################################################################
```

```
##############################################################################################
# Now combine them into their respective scripts
##############################################################################################
cat Ensembl-Header.txt Mapping-check.txt > Ensembl-ID-Mapping.sh
cat GenBank-Header.txt Mapping-check.txt > GenBank-ID-Mapping.sh
cat RefSeq-Header.txt Mapping-check.txt > RefSeq-ID-Mapping.sh
##############################################################################################


##############################################################################################
# Clean up the intermediate files made when generating scripts.
##############################################################################################
rm Ensembl-Header.txt
rm GenBank-Header.txt
rm RefSeq-Header.txt
rm Mapping-check.txt
##############################################################################################
```

[125]:
```
%%bash
### Let's go! First the Ensemble set

# The variable passed to gnu-parallel will be the species. We are only␣
 ↪specifying -j 3 here since there are only.... 3 species.
parallel -j 3 -a Ensembl-UniProt-species-list.txt "bash Ensembl-ID-Mapping.sh"
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  153k    0    52  100  153k     41   121k  0:00:01  0:00:01 --:--:--  121k
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  131k    0    52  100  131k     44   111k  0:00:01  0:00:01 --:--:--  111k
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 87640    0    52  100 87588     47  80819  0:00:01  0:00:01 --:--:-- 80848
```

[ ]:
```
%%bash
### And then GenBank - We are specifying -j 15 as the current instance only has␣
 ↪15 threads, and we don't want to overload the cpu.
# If you're using a more powerful instance type, go wild! just leave one or two␣
 ↪cpus free.
parallel -j 15 -a GenBank-UniProt-species-list.txt "bash GenBank-ID-Mapping.sh"
```

[ ]:
```
%%bash
### And lastly RefSeq
parallel -j 15 -a RefSeq-UniProt-species-list.txt "bash RefSeq-ID-Mapping.sh"
```

[ ]:
```
%%bash
# The human dataset exceeds the maximim number of queryable proteins using the␣
 ↪API (100,000). So, let's go on
# and break these up manually. Down the line this could be written in to the␣
 ↪code itself as a series of if-else statements.
split -l 60000 ~/environment/data/EukProt/TCS/data/protein-ids/
 ↪Homo_sapiens-RefSeq-ProtIDs.txt Human_

# Store the ids as a variable:
ids1=$(cat Human_aa | tr '\n' ',')
ids2=$(cat Human_ab | tr '\n' ',')
ids1=${ids1::-1}
ids2=${ids2::-1}

# Write the curl forms to a file:
echo "from=RefSeq_Protein&to=UniProtKB&ids=$ids1" > Homo_sapiens-set1-form.txt
echo "from=RefSeq_Protein&to=UniProtKB&ids=$ids2" > Homo_sapiens-set2-form.txt

# And submit, then clean up
curl -d @Homo_sapiens-set1-form.txt https://rest.uniprot.org/idmapping/run >␣
 ↪Homo_sapiens-set1.txt
curl -d @Homo_sapiens-set2-form.txt https://rest.uniprot.org/idmapping/run >␣
 ↪Homo_sapiens-set2.txt
jobID1=$(cat Homo_sapiens-set1.txt | sed "s/.*://g" | sed "s/}//g" | sed 's/"//
 ↪g')
jobID2=$(cat Homo_sapiens-set2.txt | sed "s/.*://g" | sed "s/}//g" | sed 's/"//
 ↪g')

# Clean up.
rm Homo_sapiens-set*txt

# We'll just write in some longer sleep statements to be sure that the ID␣
 ↪mapping finished completely before we begin downloading.
sleep 300
curl -s "https://rest.uniprot.org/idmapping/uniprotkb/results/stream/$jobID1?
 ↪format=tsv" > Homo_sapiens-set1.txt
sleep 300
curl -s "https://rest.uniprot.org/idmapping/uniprotkb/results/stream/$jobID2?
 ↪format=tsv" > Homo_sapiens-set2.txt

tail -n+2 Homo_sapiens-set2.txt > tmp
cat Homo_sapiens-set1.txt tmp > /home/ubuntu/environment/data/EukProt/TCS/data/
 ↪protein-ids/Homo_sapiens-UniProt-ProtIDs.txt

# Clean up.
rm tmp
```

```
rm Homo_sapiens-set*txt
rm Human_a*
```

1.5 Great, so in an ideal world, this all would have worked out swimmingly.

Unfortunately we don't live in an ideal world, and some of these ID mappings still didn't work out. It's currently unclear why this is, but for the time being, we just need to clean up and remove these from the list of species under future consideration. As such, generate a list of species that are still problematic and remove the uniprote query outputs.

[168]:
```
%%bash
# Find them
find /home/ubuntu/environment/data/EukProt/TCS/data/protein-ids/*UniProt* -size␣
 ↪-5k -exec ls -sd {} + | grep -v ipynb | cut -d' ' -f2 > tmp

# and remove them
for f in $(cat tmp); do rm $f; done && rm tmp
```

1.6 We actually have some broadly useful and internally consistent protein IDs now!

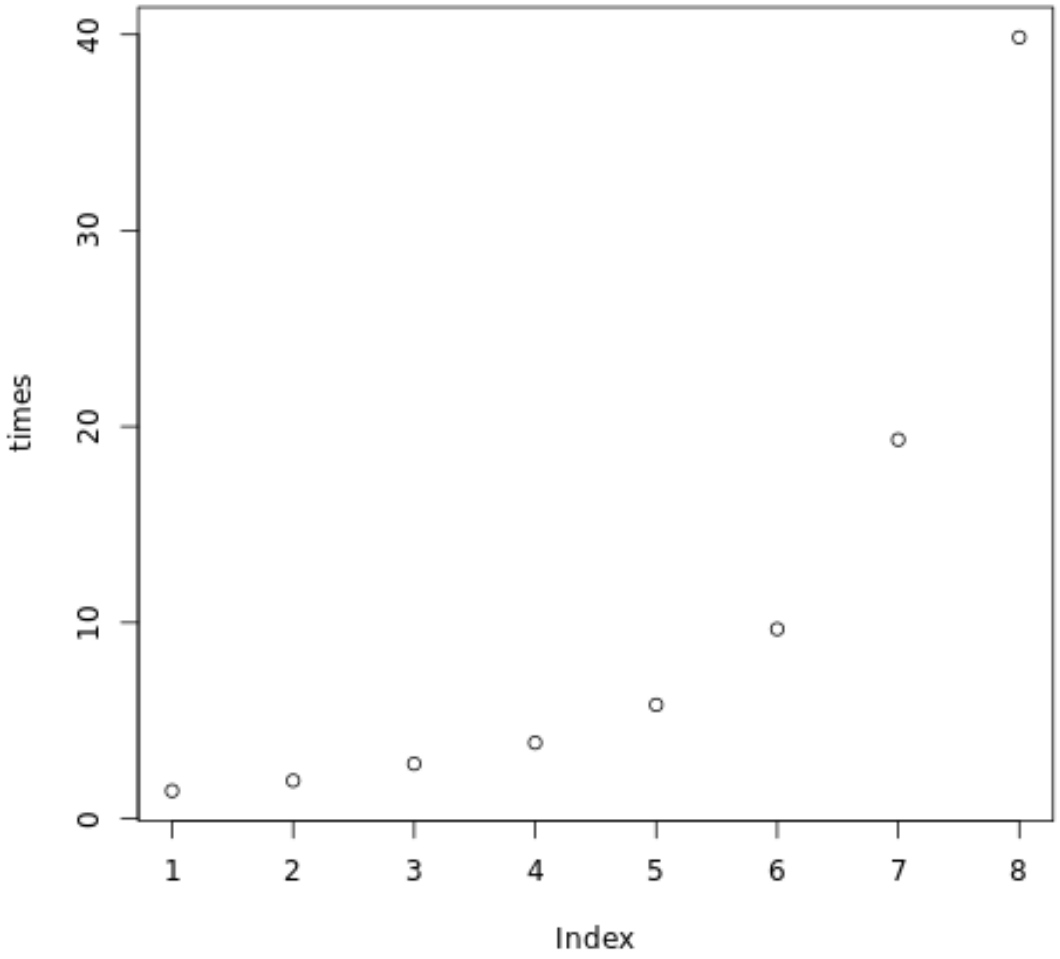
This means we can start to correspond our orthogroups and gene family trees to whatever else we'd like to, including (for instance) AlphaFold2 protein structure predictions and gene ontologies. Whereas we already have downloaded the AlphaFold predictions, and individual predictions are names according to the IDs we've just acquired, we don't yet have gene ontologies. Let's harvest these using the UniprotR R package.

[243]:
```
%%R
library(UniprotR)

# What we want to do here, is for each species, read in the list of UniProt␣
 ↪accessions and then extract their corresponding gene ontologies using the␣
 ↪GetProteinGOInfo function. First, an example using Chlamy.
dat <- read.table('~/environment/data/EukProt/TCS/data/protein-ids/
 ↪Chlamydomonas_reinhardtii-UniProt-ProtIDs.txt', sep = '\t', header = T)

# A quick look at the data:
head(dat)

# Great, so we know that the uniprot accessions are stored in the "Entry"␣
 ↪column. Now, let's do a small scale submission of these accessions to see␣
 ↪how it works.
```

```
# Note that you *can* write a csv of the results, but you can't specify the␣
 ↪name of the file, and it ends up calling it "Protein GO Info.csv", spaces␣
 ↪and all....
# So, we're going to just store that that as an object and then write to file␣
 ↪(tsv) with a name of our choosing.

# What does the GO Term output look like?
GetProteinGOInfo(ProteinAccList = head(dat$Entry))

# Okay, so that seems to work as expected - got all of the ID's themselves (in␣
 ↪the "Gene.Ontology.IDs" column, each ID separated by "; " and another␣
 ↪equivalent column ("Gene.Ontology..GO." - name followed by ID),
# as well as a column each for the different classes (i.e. biological process,␣
 ↪molecular function and cellular component).

# But - let's benchmark the efficiency of these submissions - it may be that we␣
 ↪want to submit many individual accessions in parallel, looping through␣
 ↪species.
a <- system.time(GetProteinGOInfo(ProteinAccList = head(dat$Entry,1)))
b <- system.time(GetProteinGOInfo(ProteinAccList = head(dat$Entry,2)))
c <- system.time(GetProteinGOInfo(ProteinAccList = head(dat$Entry,3)))
d <- system.time(GetProteinGOInfo(ProteinAccList = head(dat$Entry,4)))
e <- system.time(GetProteinGOInfo(ProteinAccList = head(dat$Entry)))
f <- system.time(GetProteinGOInfo(ProteinAccList = head(dat$Entry, 10)))
g <- system.time(GetProteinGOInfo(ProteinAccList = head(dat$Entry, 20)))
h <- system.time(GetProteinGOInfo(ProteinAccList = head(dat$Entry, 40)))

times <- c(a[3],b[3],c[3],d[3],e[3],f[3],g[3],h[3])
plot(times)

# Yup - horribly non-linear. We're going to want to loop through species,␣
 ↪submit each accession in parallel, and building up the go-term dataframe␣
 ↪iteratively for each species.
```

```
R[write to console]: Please wait we are processing your accessions …

R[write to console]: Please wait we are processing your accessions …

R[write to console]: Please wait we are processing your accessions …

R[write to console]: Please wait we are processing your accessions …

R[write to console]: Please wait we are processing your accessions …

R[write to console]: Please wait we are processing your accessions …

R[write to console]: Please wait we are processing your accessions …
```

```
R[write to console]: Please wait we are processing your accessions …
```



1.6.1 There are some other potentially worth-while functions to extract metadata for proteins. These are:

1) GetSubcellular_location():

- This is potentially very useful for the affinity probe project, identifying conserved proteins across diverse taxa that localize in specific organelles. I will likely end up havesting and storing these data at the tail end of this pipeline, as there is already a clear and immediate use for this information. Returned fields includes: Intramembrane, Subcellular location, Topological

Domain, and Transmembrane

2) GetExpression():

- This may be useful for certain groups at arcadia, for instance the glial origins project. Returned fields includes developmental stage, induction, and tissue specificity.

3) GetFamily_Domains():

- Lots of useful information can be obtained from this, including protein families and compositional bias - could be worth digging into eventually.

4) GetPathology_Biotech():

- Potentially some interesting stuff here - things like allergenic properties, disease involvement, mutagenesis and toxic dose. Could be useful for some.

5) GetpdbStructure():

- Not sure that these will necessarily be the most updated, but this could be a useful way to fill in the gaps in the short term, or for a quick investigation.

6) GetProteinFunction():

- Lots here - too many fields to list here, but I'm confident we can find something useful!

7) GetProteinInteractions():

- Only two fields here - "Interacts with", and "Subunit structure". As with the above, I'm confident this info will be useful for some.

8) ConvertID():

- This can be used to convert from the UniProt Accession we've just mapped everything to into whatever other format we might need down the line. Obviously we won't be using this now, but eventually this may prove useful.

1.6.2 The fields that can be queried using the functions above are all described and listed here: https://www.uniprot.org/help/return_fields

---

1.6.3 Regardless, for now we just need the GO terms.

So, let's write something that will loop through species, submitting UniProt accessions in parallel for GO-term mapping, and write out to file.

[48]:
```
%%R

# load some libraries that enable parallel computing in R
library(foreach, quietly = T)
library(doSNOW, quietly = T)
library(UniprotR, quietly = T)

# Get the list of species for which we will be doing this GO-term mapping, and␣
 ↪the paths to their corresponding UniProt accessions.
idDir <- '~/environment/data/EukProt/TCS/data/protein-ids/'
uniprots <- list.files(path = idDir, pattern = 'UniProt')

# and pull out the species names
spps <- gsub("\\-.*","",uniprots)

# Because this is going to be a fair bit of information, let's make a new␣
 ↪directory to house these outputs/gene ontologies.
goDir <- '~/environment/data/EukProt/TCS/data/protein-GOs/'
dir.create(file.path(goDir), showWarnings = FALSE) # Will only create the␣
 ↪directory if it doesn't already exist.

# Great. Now we can begin to work through the list of species in a for loop,␣
 ↪getting the go-terms for all of their uniprot accessions.
# This will involve the following steps (per species):
# 1) In parallel, submit N accessions for GO-term mapping (where accessions␣
 ↪will be drawn from their index from 1:length(accessions)) and fill in a list␣
 ↪by their index (where the list will be equal in length to the number of␣
 ↪uniprot accessions)
# 2) This list (of dataframes) will then be combined into a single data frame␣
 ↪using rlist::list.rbind(), and written to file.

# Initialize the parallel computing environment.
myCluster <- makeCluster(124) # number of cores to use, and type of cluster

registerDoSNOW(myCluster)

for(spp in spps[-c(1:75)]){
    # Who are we working on?
    print(paste0("Working on ", spp, ". This is species ", which(spp == spps),␣
 ↪" of ", length(spps), "."))

    # get the list of accessions for this species.
    accessions <- read_tsv(paste0('~/environment/data/EukProt/TCS/data/
 ↪protein-ids/', spp, '-UniProt-ProtIDs.txt'), col_types = cols())$Entry

    # Get the time taken per submission:
```

```
    a <- system.time(GetProteinGOInfo(ProteinAccList = accessions[1]))
    time <- ((a[3] * length(accessions)) / 124) / 60   # seconds per accession,␣
 ↪times the number of accessions, divided by the number of parallel jobs,␣
 ↪converted to minutes.
    print(paste0("This could take approximately ", round(time[[1]], 3), "␣
 ↪minutes. Go take a walk."))

    # And some trickery to get a progress bar to keep track of how far along we␣
 ↪are for each species
    pb <- txtProgressBar(max = length(accessions), style = 3)
    progress <- function(n) setTxtProgressBar(pb, n)
    opts <- list(progress = progress)

    go_terms <- foreach(prot = accessions, .combine = 'rbind', .options.snow =␣
 ↪opts) %dopar% {UniprotR::GetProteinGOInfo(prot)}

    # Reformat a bit
    go_terms <- data.frame(Entry = row.names(go_terms), go_terms, row.names =␣
 ↪NULL)

    # And remove rows for which no GO terms are revovered.
    go_terms <- go_terms[-which(rowSums(is.na(go_terms[-1])) ==␣
 ↪(ncol(go_terms)-1)),]

    # Write out to a tsv.
    write.table(go_terms, file = paste0(goDir, spp, '-UniProt-GO-Terms.tsv'),␣
 ↪col.names = T, row.names = F, sep = '\t', quote = F)
}

# And stop the parallel computing environment
stopCluster(myCluster)
```

```
[1] "Working on Trypanosoma_cruzi. This is species 76 of 80."
R[write to console]: Please wait we are processing your accessions …


[1] "This could take approximately 4.242 minutes. Go take a walk."
  |======================================================================|
100%[1] "Working on Ustilago_maydis. This is species 77 of 80."
R[write to console]: Please wait we are processing your accessions …


[1] "This could take approximately 1.31 minutes. Go take a walk."
  |======================================================================|
100%[1] "Working on Vitrella_brassicaformis. This is species 78 of 80."
R[write to console]: Please wait we are processing your accessions …
```

```
[1] "This could take approximately 3.065 minutes. Go take a walk."
  |======================================================================|
100%[1] "Working on Volvox_carteri. This is species 79 of 80."

R[write to console]: Please wait we are processing your accessions …


[1] "This could take approximately 2.857 minutes. Go take a walk."
  |======================================================================|
100%[1] "Working on Yarrowia_lipolytica. This is species 80 of 80."

R[write to console]: Please wait we are processing your accessions …


[1] "This could take approximately 1.295 minutes. Go take a walk."
  |======================================================================| 100%
```

[ ]:
```
%%R
# Do the same but, for subcellular localization

# Get the list of species for which we will be doing this GO-term mapping, and␣
 ↪the paths to their corresponding UniProt accessions.
idDir <- '~/environment/data/EukProt/TCS/data/protein-ids/'
uniprots <- list.files(path = idDir, pattern = 'UniProt')

# and pull out the species names
spps <- gsub("\\-.*","",uniprots)

# Because this is going to be a fair bit of information, let's make a new␣
 ↪directory to house these outputs/gene ontologies.
locDir <- '~/environment/data/EukProt/TCS/data/protein-localization/'
dir.create(file.path(locDir), showWarnings = FALSE) # Will only create the␣
 ↪directory if it doesn't already exist.

# Great. Now we can begin to work through the list of species in a for loop,␣
 ↪getting the go-terms for all of their uniprot accessions.
# This will involve the following steps (per species):
# 1) In parallel, submit N accessions for GO-term mapping (where accessions␣
 ↪will be drawn from their index from 1:length(accessions)) and fill in a list␣
 ↪by their index (where the list will be equal in length to the number of␣
 ↪uniprot accessions)
# 2) This list (of dataframes) will then be combined into a single data frame␣
 ↪using rlist::list.rbind(), and written to file.

# Initialize the parallel computing environment.
myCluster <- makeCluster(124) # number of cores to use, and type of cluster
```

```
registerDoSNOW(myCluster)

for(spp in spps){
    # Who are we working on?
    print(paste0("Working on ", spp, ". This is species ", which(spp == spps),␣
 ↪" of ", length(spps), "."))

    # get the list of accessions for this species.
    accessions <- read_tsv(paste0('~/environment/data/EukProt/TCS/data/
 ↪protein-ids/', spp, '-UniProt-ProtIDs.txt'), col_types = cols())$Entry

    # Get the time taken per submission:
    a <- system.time(GetSubcellular_location(ProteinAccList = accessions[1]))
    time <- ((a[3] * length(accessions)) / 124) / 60   # seconds per accession,␣
 ↪times the number of accessions, divided by the number of parallel jobs,␣
 ↪converted to minutes.
    print(paste0("This could take approximately ", round(time[[1]], 3), "␣
 ↪minutes. Go take a walk."))

    # And some trickery to get a progress bar to keep track of how far along we␣
 ↪are for each species
    pb <- txtProgressBar(max = length(accessions), style = 3)
    progress <- function(n) setTxtProgressBar(pb, n)
    opts <- list(progress = progress)

    prot_locs <- foreach(prot = accessions, .combine = 'rbind', .options.snow =␣
 ↪opts) %dopar% {UniprotR::GetSubcellular_location(prot)}

    # Reformat a bit
    prot_locs <- data.frame(Entry = row.names(prot_locs), prot_locs, row.names␣
 ↪= NULL)

    # And remove rows for which no GO terms are revovered.
    prot_locs <- prot_locs[-which(rowSums(is.na(prot_locs[-1])) ==␣
 ↪(ncol(prot_locs)-1)),]

    # Write out to a tsv.
    write.table(prot_locs, file = paste0(locDir, spp,␣
 ↪'-UniProt-Subcelluar-Localization.tsv'), col.names = T, row.names = F, sep =␣
 ↪'\t', quote = F)
}

# And stop the parallel computing environment
stopCluster(myCluster)
```