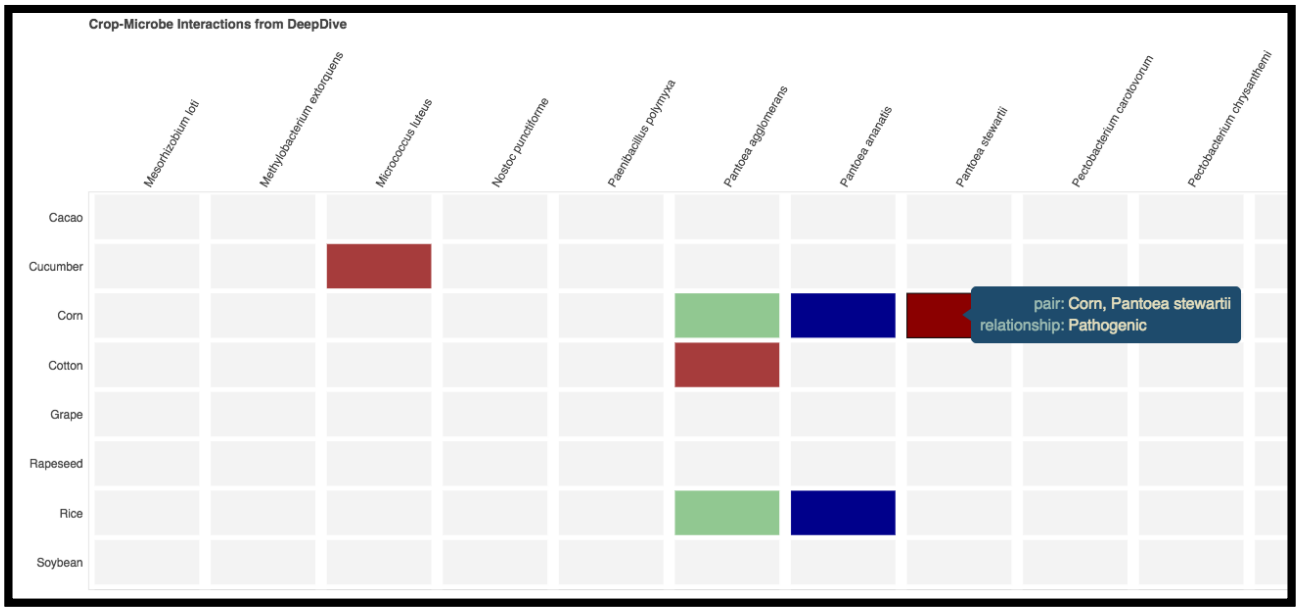
Text mining applications with DeepDive

A tutorial

Jahir M. Gutierrez – University of California, San Diego



- Example of Crop-Microbe interactions predicted by DeepDive using PubMed abstracts

At a glance

This document contains specific instructions to:

- Build and configure DeepDive applications
- Test, run, and debug deployed applications

Table of Contents

Introduction

DeepDive is a computer program for extracting entities and facts from very large text corpora. Recently, we have used DeepDive to find plant-endophyte and plant-pathogen pairs across ~60,000 documents including full-text journal articles and PUBMED abstracts. DeepDive was designed to make it easier for users to define features, linguistic and grammatical rules without the need to worry about the actual machine learning algorithms necessary to process language and obtain meaningful predictions. DeepDive was developed by computer scientists at Stanford University and is hosted at http://deepdive.stanford.edu (full documentation and tutorials can be found in this website). In the following sections, we will describe how to build and run a DeepDive application in Linux that finds plant-pathogen pairs in full-text journal articles. All terminal commands are indicated using the blue `consolas` font and it is assumed that the reader will be running these on an Ubuntu machine.

0. DeepDive Installation

Before installing DeepDive, some dependencies must be installed. Please note that we will be using the home directory **"**`home/ubuntu/`**"** as the reference location to illustrate the installation process step by step.

1. **Scala**
    a. In the terminal, enter the command
    `wget "http://downloads.lightbend.com/scala/2.11.8/scala-2.11.8.zip"`
    b. Unzip the file
    `unzip scala-2.11.8.zip`
    c. Confirm that a folder called "scala-2.11.8" has been created with location "`home/ubuntu/scala-2.11.8`"
    d. Open the .bashrc file with
    `nano home/Ubuntu/.bashrc`
    and add the following line at the end of the file
    `PATH=/home/ubuntu/scala-2.11.8/bin:$PATH`
    Save and close the .bashrc file. Finally, enter
    `source ./bashrc`
    to run this file.
2. **Git**
    a. Enter `sudo apt-get install git`
3. **Curl**
    a. Enter `sudo apt-get install curl`

We can now install DeepDive and a compatible version of Postgresql. In the home directory enter:
`bash <(curl -fsSL deepdive.stanford.edu/install)  postgres deepdive`
Open the .bashrc file again with and add the following line at the end of the file
`PATH=/home/ubuntu/local/bin:$PATH`
Save and close the .bashrc file. Source the .bashrc file as we did before with
`source ./bashrc`
Finally, we can test our DeepDive installation by typing the following command
`bash <(curl -fsSL deepdive.stanford.edu/install)  run_deepdive_tests`
If all the above steps were done correctly, the last command will run 125 test, 120 of which will pass and the remaining 5 will be skipped.

1. Building the scaffold of the DeepDive application

Every time we build a DeepDive application we need to create a folder for it. Let us create this folder in the home directory with

`mkdir MY_FIRST_DD_APP`

Now, we need to create 5 basic components **inside** the MY_FIRST_DD_APP folder (i.e. navigate to this folder with `cd /home/Ubuntu/MY_FIRST_DD_APP/)` :

1. A folder called **input** (`mkdir input`)

   This folder will contain the files that DeepDive will read as input. For example, in this tutorial we will be building an application that reads full-text articles from the Biomed Central dataset and thus the **input** folder will contain the files from this dataset.

2. A folder called **udf** (`mkdir udf`)

   The name of this folder stands for **User Defined Functions** and it will contain all the python functions that will tell DeepDive the criteria, features and keywords we are looking for in the input text.

3. A file named **app.ddlog** (`touch app.ddlog`)

   This file is the heart of our application. All the structure of the relational database to be constructed with DeepDive is defined here as well as the structure and format of our input files and the statistical model we will use to make predictions.

4. A file named **db.url** (`touch db.url`)

   This file contains a single line of text. This is the URL of the database that will be constructed with DeepDive and Postgresql. It always looks like this:

   postgresql://localhost/<name_of_deepdive_app>_$USER

   Where we will replace the <name_of_deepdive_app> by MY_FIRST_DD_APP

5. A file named **deepdive.conf** (`touch deepdive.conf`)

   This file configures some parameters in DeepDive such as the holdout fraction of the test/training dataset. Please read the documentation at http://deepdive.stanford.edu for more details or check the **deepdive.conf** file in the **Supplementary Folder B-Demo_DeepDive_application**.

With the MY_FIRST_DD_APP folder ready, we can now start to modify/add files to the 5 basic components to run our first DeepDive application.

2. Preparing the text corpus and training examples

The http://deepdive.stanford.edu/opendata/ website contains ready-to-use datasets for working with DeepDive. Table 1 below shows the size and description of those datasets. In this SOP, we will be using the Biomed Central dataset (BMC) to search for plant-pathogen pairs and catalogue them with respect to the microbe type (fungus or bacterium).

Table 1 – Size and number of documents in the DeepDive open datasets

	Number of documents	Size
PubMed Central Open Access	359,324 full-text articles	70 GB
Biomed Central	70,043 full-text articles	21 GB
Public Library of Science	125,378 full-text articles	70 GB
Google Patents	2,437,00 patents	428 GB

The first step is to download the BMC dataset as a compressed file from the DeepDive website and extract the contents into a new folder.

1. **Download and extract the contents in the BMC dataset**
    a. In the home directory, create a new folder called BMC

    ```
    mkdir BMC
    ```

    Navigate to that folder by entering

    ```
    cd BMC/
    ```

    b. Download the BMC dataset

    ```
    wget
    "http://i.stanford.edu/hazy/opendata/bmc/bmc_full_dddb_20150927_965
    1bf4a468cefcea30911050c2ca6db.tar.bzip2"
    ```

    c. Extract its contents

    ```
    tar -xvjf
    bmc_full_dddb_20150927_9651bf4a468cefcea30911050c2ca6db.tar.bzip2
    ```

After the extraction step, a folder called `bmc_full_dddb` is created. This folder contains about 30 files in the tab-separated value (.tsv) format. We will work with the `sentences-0.tsv` file only for now, which will help us illustrate all the steps required to build and deploy a DeepDive application.

2. **Label the BMC sentences-0.tsv file with the names of plant and microbial organisms**
    a. Copy the "sentences-0.tsv" file into the **Supplementary Folder A-Labeling_sentences**

b. Navigate to the Supplementary Folder A and enter the following command:

```
python labelORGANISMS.py sentences-0.tsv
```

This last command will run a python script that finds and tags all the words in the sentences-0.tsv file that correspond to the names of PLANTS, FUNGI or PROKARYOTES (bacteria and archaea). The resulting file is called "sentences-0_LABELED.tsv".

3. **Prepare the sentences file for the DeepDive application**
   a. Move the "sentences-0_LABELED.tsv" into the **input** folder inside the MY_FIRST_DD_APP directory

      ```
      mv sentences-0_LABELED.tsv /home/ubuntu/MY_FIRST_DD_APP/input/
      ```

   b. Navigate to the input folder and rename the sentences file

      ```
      cd /home/ubuntu/MY_FIRST_DD_APP/input/
      mv sentences-0_LABELED.tsv sentences.tsv
      ```

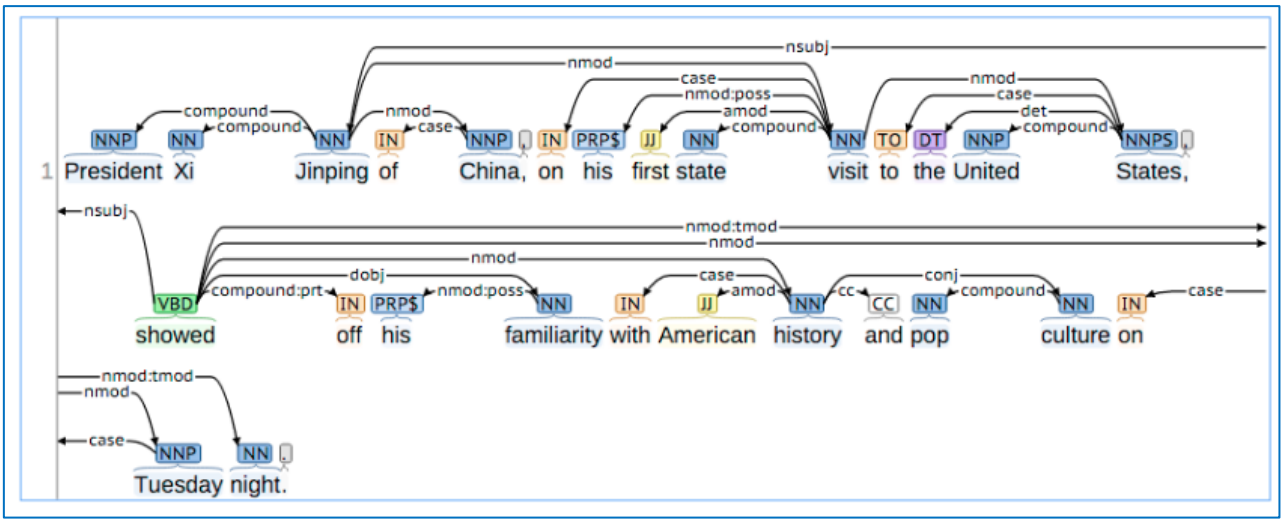
What we just did is creating a file with all the text that DeepDive will read for us. Before we move on, we need to add one more input file inside the same **MY_FIRST_DD_APP/input** directory. Namely, this file contains a list of **example** plant-pathogen pairs that we will use as a training set for DeepDive to infer the rules behind the predictive model that will find new examples of plant-pathogen pairs. This file is called **plant_pathogen_pairs.csv** and it can be found in the **Supplementary Folder B-Demo_DeepDive_application** under the /**input** directory. We just need to copy this .csv file and add it to the **MY_FIRST_DD_APP/input** directory.

3. Understand the content of the sentences.tsv file

If we take a look at our **sentences.tsv** file (which is in our MY_FIRST_DD_APP/input directory now) we will see that each line in this file represents a sentence. Let us explain now what each column in this sentences table represents.

1. The first column indicates the **document identifier**. That is, what journal articles does the sentence come from? In this case, the identifier is `BMC_Bioinformatics_2008_Feb_19_9_104` which means that this sentence is from the February 19th 2008 article in the BMC Bioinformatics Journal, volume 9, issue 104.
2. The second column indicates the **sentence number** in the article.
3. The third column indicates the **word index** of all words in the sentence.
4. The fourth column contains the **words** as they appear in the original text.
5. The fifth column indicates the **Parts of Speech** (POS) tags for each word. For example, if the word "John" appears in column 4, the corresponding POS tag for "John" will be "NNP" which stands for "proper noun". The meaning of all POS tags can be found at https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html
6. The sixth column contains the **Name Entity Recognition** (NER) tags for each word in the sentence. Continuing with our previous example, the corresponding NER tag for the word "John", if it appeared in column 4, would be "PERSON". DeepDive recognizes PERSON, NUMBER, ORGANIZATION, PERCENTAGE, LOCATION and TIME. Note, however, that when we ran the **labelORGANISMS.py** with the original BMC sentences-0.tsv file, we actually added the NER tags PLANT, FUNGUS and PROKARYOTE for all those words that matched the names of the corresponding organism. Finally, a NER tag 'O' (capital letter "o") means the word does not represent an entity.
7. The seventh column contains the stemmed version of the original words, also called **lemmas**. That is, each word in column 4 appears again here but in lower case (except with proper nouns and acronyms), in singular form and in the infinitive tense (in case of verbs). For example, if column 4 contained the words {"John","And","I","Are","Friends"}, the corresponding lemmas in column 7 would be {"John","and","I","be","friend"}.
8. The eighth column contains the **dependency function type** of each word in the sentence. This is a very useful way to represent a sentence as a tree where the words are linked by grammatical functions. Excellent information about this structure and the meaning of each dependency can be found at http://nlp.stanford.edu/software/dependencies_manual.pdf as well as at

https://en.wikipedia.org/wiki/Dependency_grammar. The figure below shows an example



9. The ninth column contains the **dependency tokens** of each word in the sentence. These are closely related to the dependency types of column 8 and detailed information of their meaning can be found in sections 4 and 5 of following document http://nlp.stanford.edu/software/dependencies_manual.pdf. The Stanford dependency parser is also a very useful resource to learn more about linguistic dependencies http://nlp.stanford.edu:8080/parser/
10. Finally, column 10 indicates the **document offsets** for each word in the sentence. That is, it tells the start and end positions of the word in the full article.

4. Initialize Database and load /input files

We are now about to start **writing** our first DeepDive application. Our text dataset and the training data are ready and, as described in the documentation (http://deepdive.stanford.edu/development-cycle), it is time to open our **app.ddlog** file. First, take into account that DeepDive will create a **relational Database** and this Database will thus contain tables with different types of data. The first thing we will write to our app.ddlog file will thus be the **schema** of the tables that will store the information in each of the files in our **/input** folder. For more information about how the DDLOG language works, please consult the documentation at (http://deepdive.stanford.edu/writing-dataflow-ddlog)

1. Define schema for the **sentences** file
    a. Since the name of our input file is **sentences**.tsv, we need to name the table schema with the same name so that DeepDive knows what file to load from the /**input** folder. In the app.ddlog file type the following

    ```
    # DEFINE TABLE SCHEMA FOR SENTENCES.TSV
    sentences().
    ```

    The first line starts with a (#) symbol and is just a comment (DeepDive will not read it). The second line indicates the name of the table we will create in the DeepDive database which will store the sentences from the sentences.tsv file in our /input folder.

    b. Now, inside the round brackets we will declare the name of the **headers** of each column in the sentences.tsv file (see **Section 3**). The name of each column header must be followed by the type of data contained under the corresponding column. The different types are **text** (for columns that contain a single text string per row), **int** (for columns that contain a single integer value), **text[]** (for lists of words) and **int[]** (for lists of integer numbers).
    Thus, following the same format of our sentences.tsv file, we will declare its columns such that our app.ddlog file looks as follows:

    ```
    # DEFINE TABLE SCHEMA FOR SENTENCES.TSV
    sentences(
       doc_id          text,
       sentence_index  int,
       word_id         int[],
       tokens          text[],
       pos_tags        text[],
       ner_tags        text[],
       lemmas          text[],
       dep_types       text[],
       dep_tokens      int[],
       doc_offsets     text[]
    ).
    ```

    We will now do the same for our **plant_pathogen_pairs.csv** file

2. Define schema for **plant_pathogen_pairs**.csv
    a. Just as we did with our sentences file, we will add the following code to our app.ddlog file right after our sentences schema declaration. Again, remember that the name of the table schema must be the same as the name of the file in the **/input** folder.

```
# DEFINE TABLE SCHEMA FOR PLANT_PATHOGEN_PAIRS.CSV
plant_pathogen_pairs(
   plant_name      text,
   pathogen_name   text
).
```

    Here we note that the plant_pathogen_pairs.csv file only contains two columns: the first column is the plant name and the second column contains the pathogen name (which could be either a fungus or a bacterium).
    b. Save and close the app.ddlog file
3. Initialize DeepDive Database
    a. In the terminal, while inside the MY_FIRST_DD_APP directory, enter the following command:

```
deepdive compile
```

    This will read and compile our app.ddlog file and it will make sure there are no syntax errors in it. If the message "COMPILED" appears then everything is OK in our app.ddlog and we are done. However, if the message "ABORTED" is displayed then it means that we have a syntax error somewhere.

```
2016-08-31 17:04:14.571302 'run/compiled' -> 'run/compiled~'
2016-08-31 17:04:14.572043 'run/compiled' -> '20160831/170409.986249109'
ubuntu@ip-172-22-4-35:~/B-Demo_DeepDive_application$
```

    b. Now, let us tell DeepDive to load the sentences table onto the database with the "do" command:

```
deepdive do sentences
```

    This will display a document with the instructions DeepDive will follow to create a sentences table in the Database. Just input the command `:wq` and hit Enter (as you would do to save and quit in vim). DeepDive will show the progress of the upload and display "COMPLETED" when it's done. Note that a new folder called /**run** has been created inside our MY_FIRST_DD_APP directory. This folder contains files that keep track of all the runs we've done since we created our application.
    c. We will do the same for the plant_pathogen_pairs table. In the terminal enter

```
deepdive do plant_pathogen_pairs
```

    Again, type `:wq` and hit Enter when the instructions file is displayed. This will be necessary to do every time we run the **deepdive do** command.

```
2016-08-31 17:06:52.587125 + deepdive load sentences
loading sentences: 0:00:01   17k [13.2k/s] ([13.2k/s])t/sentences.tsv (tsv format)
loading sentences: 0:01:06  707MiB [10.7MiB/s] ([10.7MiB/s])
loading sentences: 0:01:06  521k [7.85k/s] ([7.85k/s])
2016-08-31 17:07:59.744776 mark_done process/init/relation/sentences
2016-08-31 17:07:59.757649 ############################################################
####################
2016-08-31 17:07:59.757685
2016-08-31 17:07:59.757699
2016-08-31 17:07:59.757711 ## data/sentences ########################################
####################
2016-08-31 17:07:59.757722 # Done: N/A
2016-08-31 17:07:59.757733 # no-op
2016-08-31 17:07:59.757744 mark_done data/sentences
2016-08-31 17:07:59.762439 ############################################################
####################
2016-08-31 17:07:59.762469
2016-08-31 17:07:59.762483
'run/FINISHED' -> '20160831/170649.787264504'
ubuntu@ip-172-22-4-35:~/B-Demo_DeepDive_application$
```

4. Confirm that these two first tables have been created in our new database
    a. Open the DeepDive Database created for this application in Postgres with the command
    `deepdive sql`
    Note that this will open Postgres in the database with the url indicated in our **db.url** file. It is also possible to access this same database by typing the alternative command
    `psql postgresql://localhost/MY_FIRST_DD_APP_$USER`
    b. Enter `\dt` to display the tables in the new database. You should see both a table named "sentences" and a table named "plant_pathogen_pairs".
    c. You can query this database with SQL commands. For example, we can count the number of sentences in the sentences table
    `SELECT COUNT(*) FROM sentences;`
    We can see that there are 521,383 sentences in this table
    d. Close Database and exit Postgres by entering `\q`

```
ubuntu@ip-172-22-4-35:~/B-Demo_DeepDive_application$ deepdive sql
psql (9.3.13)
SSL connection (cipher: DHE-RSA-AES256-GCM-SHA384, bits: 256)
Type "help" for help.

MY_FIRST_DD_APP_ubuntu=# \dt
                List of relations
 Schema |         Name          | Type  | Owner
--------+-----------------------+-------+--------
 public | plant_pathogen_pairs  | table | ubuntu
 public | sentences             | table | ubuntu
(2 rows)

MY_FIRST_DD_APP_ubuntu=# \q
ubuntu@ip-172-22-4-35:~/B-Demo_DeepDive_application$
```

5. Map all organism mentions across sentences

We will now find all the **sentences where a plant or a microbe (fungus or prokaryote) is mentioned**. This will be very useful as it will allow us later to map candidate sentences where pairs of organisms appear in the same sentence (see **Section 6**).

To do so, we will **declare the schema of three new tables** that will be stored in our relational database:

1. A table to store **the information of a plant mention**. That is, we will store all those sentences where a word tagged as "PLANT" in the 6th column of the sentences file appears. The information we want to have in this new table called **plant_mention** includes
    a. A ***mention identifier*** (a unique barcode that tells me where that mention is inside my sentences file)
    b. The ***mention text*** (that is, the actual words in the plant's name)
    c. The ***document identifier*** (so that we know in what specific BMC article the plant mention was found)
    d. A ***sentence index*** (the sentence number where it was found)
    e. The ***start*** and ***end indices*** (so that we know where in the sentence the mention of the plant begins and ends)

    Thus, we will write the following schema declaration in our app.ddlog file:

```
# DECLARE SCHEMAS OF ORGANISM MENTIONS
plant_mention(
    mention_id       text,
    mention_text     text,
    doc_id           text,
    sentence_index   int,
    begin_index      int,
    end_index        int
).
```

2. A table to store **the information of a fungus** mention. That is, we will store all those sentences where a word tagged as "FUNGUS" in the 6th column of the sentences file appears. The information we want to have in this new table called **is the same as that above**. In our app.ddlog file we will write:

```
fungus_mention(
    mention_id       text,
    mention_text     text,
    doc_id           text,
    sentence_index   int,
    begin_index      int,
    end_index        int
).
```

3. A table to store **the information of a prokaryote** mention. That is, we will store all those sentences where a word tagged as "PROKARYOTE" in the 6th column of the sentences file appears. In our app.ddlog file we will write:

```
prokaryote_mention(
    mention_id      text,
    mention_text    text,
    doc_id          text,
    sentence_index  int,
    begin_index     int,
    end_index       int
).
```

Now, how are we going to fill in the rows of the tables whose schemas we have just declared?
This will be accomplished by writing **Python** functions in our **/udf** folder. These functions will take the data from the sentences table, they will search for the "PLANT", "FUNGUS" and "PROKARYOTE" tags, and will output the information as we specified it in our table schemas above. In DeepDive, **functions** must be declared in the app.ddlog file and the scripts running the actual functions must be stored in the **/udf** folder. To map all organism mentions we will first need to tell DeepDive that we want to use a function and that it is stored in our **/udf** folder. See the documentation at (http://deepdive.stanford.edu/writing-udf-python) for more details on the syntax for function declarations. Here, we will create three functions, one function for mapping each of the organism types (plant, fungus, prokaryote). In our app.ddlog file add the following:

```
# DECLARE FUNCTIONS TO MAP ORGANISM MENTIONS
function map_plant_mention over(
        doc_id         text,
        sentence_index int,
        tokens         text[],
        ner_tags       text[]
    ) returns rows like plant_mention
    implementation "udf/map_plant_mention.py" handles tsv lines.


function map_fungus_mention over(
        doc_id         text,
        sentence_index int,
        tokens         text[],
        ner_tags       text[]
    ) returns rows like fungus_mention
    implementation "udf/map_fungus_mention.py" handles tsv lines.


function map_prokaryote_mention over(
        doc_id         text,
        sentence_index int,
        tokens         text[],
        ner_tags       text[]
    ) returns rows like prokaryote_mention
    implementation "udf/map_prokaryote_mention.py" handles tsv lines.
```

Note that in each function declaration, we are telling DeepDive to use the corresponding python **implementation** in our **/udf** folder. The corresponding python functions for mapping the organisms in the sentences can be found under **B-Demo_DeepDive_application/udf**. Just copy the map_**_mention.py files into the **MY_FIRST_DD_APP/udf** directory.
Finally, we will add the following code to our app.ddlog file to tell DeepDive how we are going to fill in the plant_, fungus_ and prokaryote_mention tables with the output from the Python functions. Note that the "_" is a place holder and indicates a column whose data is not going to be used in a function.

```
## APPLY MAPPING FUNCTIONS TO FILL IN MENTION TABLES ##
plant_mention += map_plant_mention(
    doc_id, sentence_index, tokens, ner_tags
) :-
    sentences(doc_id, sentence_index, _, tokens, _, ner_tags, _, _, _, _).

fungus_mention += map_fungus_mention(
    doc_id, sentence_index, tokens, ner_tags
) :-
    sentences(doc_id, sentence_index, _, tokens, _, ner_tags, _, _, _, _).

prokaryote_mention += map_prokaryote_mention(
    doc_id, sentence_index, tokens, ner_tags
) :-
    sentences(doc_id, sentence_index, _, tokens, _, ner_tags, _, _, _, _).
```

Here, we are telling DeepDive that, for example, the **plant_mention** table should be filled in with the output from **map_plant_mention** whose input is the **sentences** table.
Now, save and close the app.ddlog file. In the terminal, run the following commands:

```
deepdive compile
deepdive do plant_mention
deepdive do fungus_mention
deepdive do prokaryote_mention
```

We can now check the status and size of these new tables in our database:

```
deepdive sql
\dt
SELECT COUNT(*) FROM plant_mention;
SELECT COUNT(*) FROM fungus_mention;
SELECT COUNT(*) FROM prokaryote_mention;
\q
```

```
MY_FIRST_DD_APP_ubuntu=# \dt
                List of relations
 Schema |         Name          | Type  | Owner
--------+-----------------------+-------+--------
 public | fungus_mention        | table | ubuntu
 public | plant_mention         | table | ubuntu
 public | plant_pathogen_pairs  | table | ubuntu
 public | prokaryote_mention    | table | ubuntu
 public | sentences             | table | ubuntu
(5 rows)

MY_FIRST_DD_APP_ubuntu=# SELECT COUNT(*) FROM plant_mention ;
 count
-------
  7379
(1 row)

MY_FIRST_DD_APP_ubuntu=# SELECT COUNT(*) FROM fungus_mention ;
 count
-------
  1952
(1 row)

MY_FIRST_DD_APP_ubuntu=# SELECT COUNT(*) FROM prokaryote_mention ;
 count
-------
  7866
(1 row)

MY_FIRST_DD_APP_ubuntu=#
```

6. Map the candidate sentences with plant-microbe pairs

Now that DeepDive has found all the instances of organism mentions, it will be really easy to find candidate plant-microbe pairs that appear in the same sentence. These will be our candidate pairs to extract the interactions from since we are assuming that there must be words in between a plant mention and a microbe mention that could indicate their host-pathogen relationship.

1. First, note that there might be sentences where there are multiple instances of PLANT and FUNGUS/PROKARYOTE mentions. Thus what we are going to do is define a set of variables to keep track of the **number** of times the NER tag PLANT, FUNGUS or PROKARYOTE appear in a given sentence. In the app.ddlog file, write the following lines of code:

```
# COUNT THE NUMBER OF PLANTS IN EACH SENTENCE

num_plants(doc_id, sentence_index, COUNT(a)) :-
      plant_mention(a, _, doc_id, sentence_index, _, _).
```

   What the code above basically says is:
   - Create a table called **num_plants** which has three columns (document id, sentence index, and number of plants)
   - The rows of the **num_plants** table will be filled in with information from the **plant_mention** table
   - The letter "a" represents the plant_mention's identifier. Thus **num_plants** will take all the sentences in the same document with different identifiers (given by "a") and it will **count** the total

   We can do the same operation for counting the number of fungi and prokaryotes in each sentence:

```
# COUNT THE NUMBER OF FUNGI AND PROKARYOTES IN EACH SENTENCE

num_fungi(doc_id, sentence_index, COUNT(b)) :-
    fungus_mention(b, _, doc_id, sentence_index, _, _).

num_prokaryotes(doc_id, sentence_index, COUNT(c)) :-
    prokaryote_mention(c, _, doc_id, sentence_index, _, _).
```

2. Now, we can easily declare the schema of a **Plant-Fungus** and a **Plant-Prokaryote** candidate pair tables. Again, in the app.ddlog file, add the following code:

```
# CREATE AND FILL IN THE PLANT-FUNGUS CANDIDATE PAIR TABLE
plant_fungus_candidate(plant_id, plant_name, fungus_id, fungus_name) :-
       num_plants(same_doc, same_sentence, a),
       num_fungi(same_doc, same_sentence, b),
       plant_mention(plant_id, plant_name, same_doc, same_sentence, _,
_),
```

```
        fungus_mention(fungus_id, fungus_name, same_doc, same_sentence, _,
_),
        a < 5,
        b < 5.
```

This code basically says:

- Create a table called **plant_fungus_candidate** with four columns: the plant mention identifier, the plant name, the fungus mention identifier and the fungus name
- Fill in the rows of the plant_fungus_candidate table with data from the **num_plants**, **num_fungi**, **plant_mention** and **fungus_mention** tables.
- A row in the **plant_fungus_candidate** table is one for which (1) the plant_mention and the fungus_mention appear in the *same document* and *same sentence* and (2) there are at most 4 plant mentions and at most 4 fungus mentions in the same sentence.

We do the same for finding all the **Plant-Prokaryote** candidate pairs. In our app.ddlog file we add:

```
# CREATE AND FILL IN THE PLANT-FUNGUS CANDIDATE PAIR TABLE

plant_prokaryote_candidate(plant_id, plant_name, prokaryote_id,
prokaryote_name) :-
        num_plants(same_doc, same_sentence, a),
        num_prokaryotes(same_doc, same_sentence, c),
        plant_mention(plant_id, plant_name, same_doc, same_sentence, _,
_),
        prokaryote_mention(prokaryote_id, prokaryote_name, same_doc,
same_sentence, _, _),
        a < 5,
        c < 5.
```

3. We can now save and close our app.ddlog file and run the following commands:

```
deepdive compile
deepdive do plant_fungus_candidate
deepdive do plant_prokaryote_candidate
```

Finally, let us check the status and size of these new tables in our database:

```
deepdive sql
\dt
SELECT COUNT(*) FROM plant_fungus_candidate
SELECT COUNT(*) FROM plant_prokaryote_candidate;
\q
```

```
ubuntu@ip-172-22-4-35:~/B-Demo_DeepDive_application$ deepdive sql
psql (9.3.13)
SSL connection (cipher: DHE-RSA-AES256-GCM-SHA384, bits: 256)
Type "help" for help.

MY_FIRST_DD_APP_ubuntu=# \dt
                List of relations
 Schema |            Name             | Type  | Owner
--------+-----------------------------+-------+--------
 public | fungus_mention              | table | ubuntu
 public | plant_fungus_candidate      | table | ubuntu
 public | plant_mention               | table | ubuntu
 public | plant_pathogen_pairs        | table | ubuntu
 public | plant_prokaryote_candidate  | table | ubuntu
 public | prokaryote_mention          | table | ubuntu
 public | sentences                   | table | ubuntu
(7 rows)

MY_FIRST_DD_APP_ubuntu=# SELECT COUNT(*) FROM plant_fungus_candidate;
 count
-------
   111
(1 row)

MY_FIRST_DD_APP_ubuntu=# SELECT COUNT(*) FROM plant_prokaryote_candidate;
 count
-------
   195
(1 row)

MY_FIRST_DD_APP_ubuntu=# \q
ubuntu@ip-172-22-4-35:~/B-Demo_DeepDive_application$
```

7. Extract the generic features of all candidate sentences

Since we now have identified all candidate Plant-microbe pairs, the next step is to look at the features of each candidate. DeepDive can then learn those features that correlate well with correct predictions (i.e. a correct plant-pathogen pair). Here, we will compute the **generic features** from the DDLIB python module. A detailed description of this module and the generic features can be found here (http://deepdive.stanford.edu/gen_feats).

1. **Define the schema for the feature tables**
   As always, we need to define the schema of the tables that will store the features of each Plant-Microbe pair. We will define three columns in the features table: the plant mention identifier, the microbe mention identifier, and the associated feature. Because we have both Plant-Fungus and Plant-Prokaryote pairs, we will define two schemas in our **app.ddlog** file, one for each type of pair.

```
# DEFINE FEATURES TABLE SCHEMA

# Plant-Fungus features
pathogenic_fungus_feature(
   plant_id   text,
   fungus_id  text,
   feature    text
).

# Plant-prokaryote features
plant_prokaryote_feature(
   plant_id   text,
   prokaryote_id  text,
   feature    text
).
```

2. **Define a function schema to extract the features from each plant-microbe pair**
   We will define two function schemas (again, one for plant-fungus pairs and one for plant-prokaryote pairs). The two functions will be declared in our **app.ddlog** file and the associated user-defined function must be stored inside the **/udf** directory.
   The python script to extract the features from each pair is called **extract_features.py** and can be found under the **B-Demo_DeepDive_application/udf** directory.
   Note that the two feature-extracting functions are declared in app.ddlog in very similar ways. Both functions even call the **same python implementation.** The only difference then is the "**return rows like**" part as well as the column names for each output table.

```
# DEFINE FEATURE-EXTRACTING FUNCTIONS

# Plant-fungus features
function extract_pathogenic_fungus_features over (
     plant_id            text,
     fungus_id           text,
     plant_begin_index   text,
     plant_end_index     text,
     fungus_begin_index  text,
     fungus_end_index    text,
     doc_id              text,
     sent_index          int,
     tokens              text[],
     pos_tags            text[],
     ner_tags            text[],
     lemmas              text[],
     dep_types           text[],
     dep_tokens          int[]
) returns rows like pathogenic_fungus_feature
  implementation "udf/extract_features.py" handles tsv lines.

# Plant-prokaryote features
function extract_pathogenic_prokaryote_features over (
     plant_id                text,
     prokaryote_id           text,
     plant_begin_index       text,
     plant_end_index         text,
     prokaryote_begin_index  text,
     prokaryote_end_index    text,
     doc_id                  text,
     sent_index              int,
     tokens                  text[],
     pos_tags                text[],
     ner_tags                text[],
     lemmas                  text[],
     dep_types               text[],
     dep_tokens              int[]
) returns rows like pathogenic_prokaryote_feature
  implementation "udf/extract_features.py" handles tsv lines.
```

3. **Call functions to fill in the rows in the feature tables**
   We will add the following code to our app.ddlog right below the function definitions we have just written. This will add (thus the += operator) the output of the feature extraction function as rows of the pathogenic_fungus_feature.

```
# EXTRACT FEATURES AND FILL IN FEATURE TABLES

pathogenic_fungus_feature += extract_pathogenic_fungus_features(
plant_id, fungus_id, plant_begin_index, plant_end_index, fungus_begin_index,
fungus_end_index, doc_id, sent_index, tokens, pos_tags, ner_tags, lemmas,
dep_types, dep_tokens) :-
plant_mention(plant_id, _, doc_id, sent_index, plant_begin_index,
plant_end_index),
fungus_mention(fungus_id, _, doc_id, sent_index, fungus_begin_index,
fungus_end_index),
sentences(doc_id, sent_index, _, tokens, pos_tags, ner_tags, lemmas,
dep_types, dep_tokens, _).

pathogenic_prokaryote_feature += extract_pathogenic_prokaryote_features(
plant_id, prokaryote_id, plant_begin_index, plant_end_index,
prokaryote_begin_index, prokaryote_end_index, doc_id, sent_index, tokens,
pos_tags, ner_tags, lemmas, dep_types, dep_tokens) :-
plant_mention(plant_id, _, doc_id, sent_index, plant_begin_index,
plant_end_index),
prokaryote_mention(prokaryote_id, _, doc_id, sent_index,
prokaryote_begin_index, prokaryote_end_index),
sentences(doc_id, sent_index, _, tokens, pos_tags, ner_tags, lemmas,
dep_types, dep_tokens, _).
```

4. **Compute features and update DeepDive database**
   Save and close the app.ddlog file and run the following commands:

```
deepdive compile
deepdive do pathogenic_fungus_feature
deepdive do pathogenic_prokaryote_feature
```

   Again, you can check the status of the database:

```
deepdive sql
\dt
SELECT COUNT(*) FROM pathogenic_fungus_feature
SELECT COUNT(*) FROM pathogenic_prokaryote_feature;
\q
```

```
MY_FIRST_DD_APP_ubuntu=# \dt
                  List of relations
 Schema |             Name              | Type  | Owner
--------+-------------------------------+-------+--------
 public | fungus_mention                | table | ubuntu
 public | pathogenic_fungus_feature     | table | ubuntu
 public | pathogenic_prokaryote_feature | table | ubuntu
 public | plant_fungus_candidate        | table | ubuntu
 public | plant_mention                 | table | ubuntu
 public | plant_pathogen_pairs          | table | ubuntu
 public | plant_prokaryote_candidate    | table | ubuntu
 public | prokaryote_mention            | table | ubuntu
 public | sentences                     | table | ubuntu
(9 rows)

MY_FIRST_DD_APP_ubuntu=# SELECT COUNT(*) FROM pathogenic_fungus_feature;
 count
-------
 72991
(1 row)

MY_FIRST_DD_APP_ubuntu=# SELECT COUNT(*) FROM pathogenic_prokaryote_feature;
 count
--------
 155313
(1 row)

MY_FIRST_DD_APP_ubuntu=# \q
ubuntu@ip-172-22-4-35:~/B-Demo_DeepDive_application$ deepdive sql
```

8. Label all candidate pairs as positive, null or negative examples

Labeling each candidate pair as positive, null or negative is useful for **Distant Supervision**. This will provide a noisy set of labels for candidate pairs, with which we will train DeepDive's machine learning model.

We will utilize two basic categories of labeling:

A. Labeling examples that match those in our **plant_pathogen_pairs.csv** file. This is, we are using secondary data for distant supervision
B. Labeling examples based on **user-defined rules**. These rules can include linguistic rules as well as syntactic rules as we will show below.

At the end, we will simply use a majority-vote approach to resolve multiple labels per candidate. The final score (i.e. the sum of all scores) will tell DeepDive whether a candidate should be considered positive, null or negative example for training the model.

1. **Define the schema of the candidate labels table**
   We will define a table schema for storing the labels of all plant-fungus candidates and another table for plant-prokaryote candidates. Each table will have 4 columns, the plant mention identifier, the microbe identifier, the label (which is a number: -1 for negative example, 0 for null example and 1 for positive example) and a rule identifier (this will remind us the rule used to assign the given label).
   In app.ddlog, add the following code:

```
# DEFINE TABLE SCHEMAS FOR LABELS

pathogenic_fungus_label(
plant_id    text,
fungus_id   text,
label       int,
rule_id     text
).

pathogenic_prokaryote_label(
plant_id          text,
prokaryote_id     text,
label             int,
rule_id           text
).
```

2. **Make sure all candidates can be considered unsupervised examples**
   By assigning a null label to all candidates, we ensure all examples in our candidate pool can be considered in the training of the model. To do so, add the following lines of code in app.ddlog:

```
# Assign a NULL label to all candidates for unsupervised label

pathogenic_fungus_label(plant_id, fungus_id, 0, NULL) :-
      plant_fungus_candidate(plant_id, _, fungus_id, _).

pathogenic_prokaryote_label(plant_id, prokaryote_id, 0, NULL) :-
      plant_prokaryote_candidate(plant_id, _, prokaryote_id, _).
```

   The code above uses the ":-" operator to express assignment. This can be interpreted as follows:

   - Create a row in the **pathogenic_fungus_label** with a *label* equal to 0 and a *rule_id* equal to null for all *plant_id* and *fungus_id* in the **plant_fungus_candidate** table.

3. **Label candidates for distant supervision with the pairs in plant_pathogen_pairs.csv**
   We will now take the known plant-pathogen pairs from the **plant_pathogen_pairs.csv** file to label the corresponding candidates as positive examples. In other words, for each candidate pair found in our dataset, we will see whether that pair is already in the **plant_pathogen_pairs.csv** file. If it is, then we will give it a label of 1 and we will call the rule_id as *from_known_pathogens_csv* to help us remember where that label came from. The code in app.ddlog is as follows:

```
# Label positive examples from plant_pathogen_pairs.csv

pathogenic_fungus_label(plant_id, fungus_id,1,"from_known_pathogens_csv") :-
plant_fungus_candidate(plant_id, plant_name, fungus_id, fungus_name),
plant_pathogen_pairs(plant, pathogen), [ lower(plant) = lower(plant_name),
lower(pathogen) = lower(fungus_name)].

pathogenic_prokaryote_label(plant_id,
prokaryote_id,1,"from_known_pathogens_csv") :-
plant_prokaryote_candidate(plant_id, plant_name, prokaryote_id,
prokaryote_name), plant_pathogen_pairs(plant, pathogen), [ lower(plant) =
lower(plant_name), lower(pathogen) = lower(prokaryote_name)].
```

   The code above can be interpreted as follows:

   - Assign a label of 1 to all plant-fungus candidates where the *lowercase* form of the plant and fungus names match the lowercase form of any pair in the **plant_pathogen_pairs** table.
   - Name this rule as *"from_known_pathogens_csv"* so that we remember where that label=1 came from.

4. **Define a function for labeling candidates according to user-defined supervision rules**
   We need to define a function that will assign a *label* and a *rule_id* to each candidate pair given a set of user-defined rules. These rules need to be specified in a Python script inside the **/udf** folder. The python script to label each pair is called **supervise_pathogenic.py** and can be found under the **B-Demo_DeepDive_application/udf** directory. The rules in the Python script and that we will use for labeling each candidate are as follows:
   - If a sentence with a candidate pair contains keywords associated to a negative effect (e.g. "rot", "mildew", "necrosis"), label that candidate as 1 and name this rule_id as "pathogenic:negativeEffect_between"
   - If a sentence contains very long words in between the plant and the microbe, it suggests that there must be some noisy words in the text (e.g. citations). Therefore, label such candidate as -1 and name this rule_id as "negative:very_long_words_between"
   - If the phrases "resistant/susceptible against" or "resistant/susceptible to" appear immediately before the microbe, it suggests that the sentence refers to a pathogen. Thus label that with a 1 and name that rule_id as "pathogenic:ResistantAgainst_before_microbe"
   - If there are more than 10 words between the plant and the microbe, they are too far apart in the sentence and thus is unlikely that they are related. Label this candidate with a -1 and name this rule_id "negative:too_far_apart"
   - If the name of the plant is merged with that of a raw material for culture media (e.g. the plant "potato" might appear in a sentence as "potato dextrose", a supplement for culturing microbes), then it is unlikey this is a plant-pathogen pair. Therefore, label this candidate with a 1 and name the rule_id as "negative:culture_media_between"

   Now, in order to define the supervision function that will call the above Python script, we will add the following code to our app.ddlog file:

```
# Define supervision function

function supervise_pathogenic over (
  plant_id text, plant_begin int, plant_end int,
  microbe_id text, microbe_begin int, microbe_end int,
  doc_id         text,
  sentence_index int,
  word_id        int[],
  tokens         text[],
  lemmas         text[],
  pos_tags       text[],
  ner_tags       text[],
  dep_types      text[],
  doc_offsets    int[]
) returns (
  plant_id text, microbe_id text, label int, rule_id text
)
    implementation "udf/supervise_pathogenic.py" handles tsv lines.
```

5. **Fill in rows in Label Tables**

   Now that are function is declared in app.ddlog, we can fill in the rows of the label tables with the following code:

```
# FILL IN ROWS IN LABEL TABLES BY CALLING FUNCTION

# Pathogenic fungus label
pathogenic_fungus_label += supervise_pathogenic(
       plant_id, plant_begin, plant_end,
       fungus_id, fungus_begin, fungus_end,
       doc_id, sentence_index, word_id,
       tokens, lemmas, pos_tags, ner_tags, dep_types, doc_offsets) :-
plant_fungus_candidate(plant_id, _, fungus_id, _),
plant_mention(plant_id, plant_text, doc_id, sentence_index, plant_begin,
plant_end),
fungus_mention(fungus_id, fungus_text, doc_id, sentence_index,
fungus_begin, fungus_end),
sentences(doc_id, sentence_index, word_id, tokens, pos_tags, ner_tags,
lemmas, dep_types, doc_offsets, _).

# Pathogenic prokaryote label
pathogenic_prokaryote_label += supervise_pathogenic(
       plant_id, plant_begin, plant_end,
       prokaryote_id, prokaryote_begin, prokaryote_end,
       doc_id, sentence_index, word_id,
       tokens, lemmas, pos_tags, ner_tags, dep_types, doc_offsets) :-
plant_prokaryote_candidate(plant_id, _, prokaryote_id, _),
plant_mention(plant_id, plant_text, doc_id, sentence_index, plant_begin,
plant_end),
prokaryote_mention(prokaryote_id, prokaryote_text, doc_id,
sentence_index, prokaryote_begin, prokaryote_end),
sentences(doc_id, sentence_index, word_id, tokens, pos_tags, ner_tags,
lemmas, dep_types, doc_offsets, _).
```

6. **Compute the sum of all labels to get a final score**

   Given that a sentence with a candidate plant-microbe pair might have more than one label, we need to add up all the labels to get a final score. We will create a new **resolved** label variable which will be equal to the *sum* of the individual labels. In order to do so, we need to add the following code in our app.ddlog file:

```
# Compute the sum of all labels

# Plant-Fungus labels
pathogenic_fungus_label_resolved(plant_id, fungus_id, SUM(vote)) :-
pathogenic_fungus_label(plant_id, fungus_id, vote, rule_id).

# Plant-Prokaryote labels
pathogenic_prokaryote_label_resolved(plant_id, prokaryote_id,
SUM(vote)) :- pathogenic_prokaryote_label(plant_id, prokaryote_id, vote,
rule_id).
```

7. **Declare the schema of the output variable**
   The **resolved** labels contain the sum of all labels for a given plant-microbe pair. We can now use this final score to determine whether a plant-microbe pair is a TRUE plant-pathogen pair or not. We can finally declare the schema of this **output** variable in our app.ddlog file (note the "?" symbol in the declaration). The output table should contain plant and microbe identifiers of all candidate pairs, but they will be assigned a TRUE or FALSE value based the resolved labels. In app.ddlog add the following code:

```
# Declare random variables to predict as the output
is_pathogenic_fungus?(
   plant_id   text,
   fungus_id  text
).
is_pathogenic_prokaryote?(
   plant_id       text,
   prokaryote_id  text
).
```

8. **Assign TRUE or FALSE based on total votes (t)**
   In app.ddlog, add the following code:

```
# Assign TRUE or False based on final score
is_pathogenic_fungus(plant_id,fungus_id) = if t > 0 then TRUE else if t <
0 then FALSE else NULL end :-
pathogenic_fungus_label_resolved(plant_id,fungus_id,t).
is_pathogenic_prokaryote(plant_id,prokaryote_id) = if t > 0 then TRUE
else if t < 0 then FALSE else NULL end :-
pathogenic_prokaryote_label_resolved(plant_id,prokaryote_id,t).
```

9. **Run DeepDive commands to update database**

   With all our labels resolved and our output variables in place, we will now update the DeepDive database to include the labels for all candidates as well as the TRUE/FALSE/NULL labels for all candidates in the output variables. Save and close the app.ddlog file and run the following commands inside the MY_FIRST_DD_APP directory:

   ```
   deepdive compile
   deepdive do pathogenic_fungus_label
   deepdive do pathogenic_prokaryote_label
   ```

10. **Check the database to confirm that these tables have been created**

    ```
    deepdive sql
    \dt
    SELECT COUNT(*) FROM pathogenic_fungus_label;
    SELECT COUNT(*) FROM pathogenic_prokaryote_label;
    \q
    ```

```
MY_FIRST_DD_APP_ubuntu=# \dt
                  List of relations
 Schema |              Name               | Type  | Owner
--------+---------------------------------+-------+--------
 public | fungus_mention                  | table | ubuntu
 public | pathogenic_fungus_feature       | table | ubuntu
 public | pathogenic_fungus_label         | table | ubuntu
 public | pathogenic_fungus_label__0      | table | ubuntu
 public | pathogenic_prokaryote_feature   | table | ubuntu
 public | pathogenic_prokaryote_label     | table | ubuntu
 public | pathogenic_prokaryote_label__0  | table | ubuntu
 public | plant_fungus_candidate          | table | ubuntu
 public | plant_mention                   | table | ubuntu
 public | plant_pathogen_pairs            | table | ubuntu
 public | plant_prokaryote_candidate      | table | ubuntu
 public | prokaryote_mention              | table | ubuntu
 public | sentences                       | table | ubuntu
(13 rows)

MY_FIRST_DD_APP_ubuntu=# SELECT COUNT(*) FROM pathogenic_fungus_label;
 count
-------
   236
(1 row)

MY_FIRST_DD_APP_ubuntu=# SELECT COUNT(*) FROM pathogenic_prokaryote_label;
 count
-------
   466
(1 row)

MY_FIRST_DD_APP_ubuntu=# \q
ubuntu@ip-172-22-4-35:~/B-Demo_DeepDive_application$
```

9. Declare inference rules and run statistical model

Since we now have positive examples and labeled candidates according to linguistic rules, we can create an inference rule and ask DeepDive to utilize all the labeled examples and training data to calculate a statistical model and make predictions. For that, we will declare a **weight** function that will compute the weight of all **features** connecting the plants and microbes in each candidate pair. Then, DeepDive will compute the probability of that pair to be a TRUE example of what we are looking for based on the examples and label we have given to it previously. Further details about this step can be found in (http://deepdive.stanford.edu/writing-model-ddlog) and (http://deepdive.stanford.edu/ops-model)
In app.ddlog add the following:

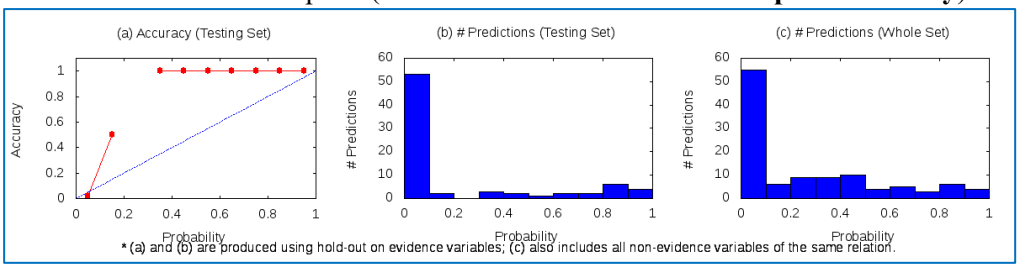
```
# Declare inference rules
@weight(f)
is_pathogenic_fungus(plant_id, fungus_id) :-
    plant_fungus_candidate(plant_id, _, fungus_id, _),
    pathogenic_fungus_feature(plant_id, fungus_id, f).

@weight(p)
is_pathogenic_prokaryote(plant_id, prokaryote_id) :-
    plant_prokaryote_candidate(plant_id, _, prokaryote_id, _),
    pathogenic_prokaryote_feature(plant_id, prokaryote_id, p).
```

Finally, save and close the app.ddlog file and run the following commands in the terminal:

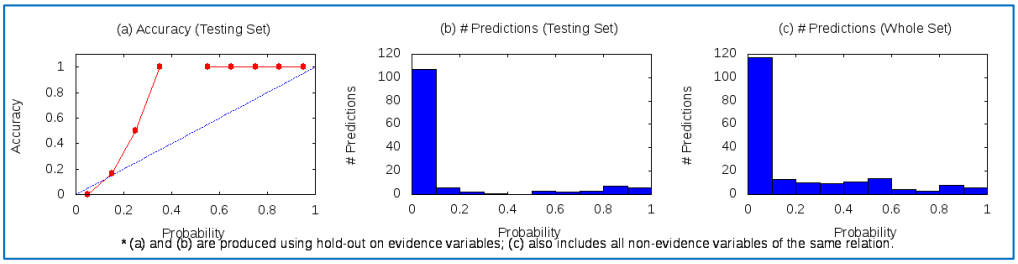
```
deepdive compile
deepdive do is_pathogenic_fungus
deepdive do is_pathogenic_prokaryote
deepdive do probabilities
deepdive do calibration-plots
```

This will compute the weights "f" and "p" of all features and create a statistical model from it. This model is then used to predict how likely (i.e. expectation) it is that a given plant-microbe pair is indeed a plant-pathogen pair. Below we show the calibration plots (stored in **run/model/calibration-plots directory**).

is_pathogenic_fungus?



is_pathogenic_prokaryote?



10. Look at results from predictions and provide feedback with PyTagger

In this section we will use a tool we built called **PyTagger** to look at the predictions in more detail and provide feedback to DeepDive to improve the model in subsequent iterations. Here is what happens when one uses PyTagger:

- First of all, PyTagger will display a *sentence* on the terminal with the plant and microbe names highlighted in green and blue, respectively.
- PyTagger will also show the expectation DeepDive calculated for that candidate. The higher the expectation value, the more likely the candidate is to be a plant-pathogen pair according to DeepDive
- PyTagger will let you read the sentence and decide if the prediction is correct, incorrect or it can't be determined from the context (null).
- After labeling all the sentences, a .tsv file is generated that can be fed back to DeepDive for improving the predictions in subsequent iterations.

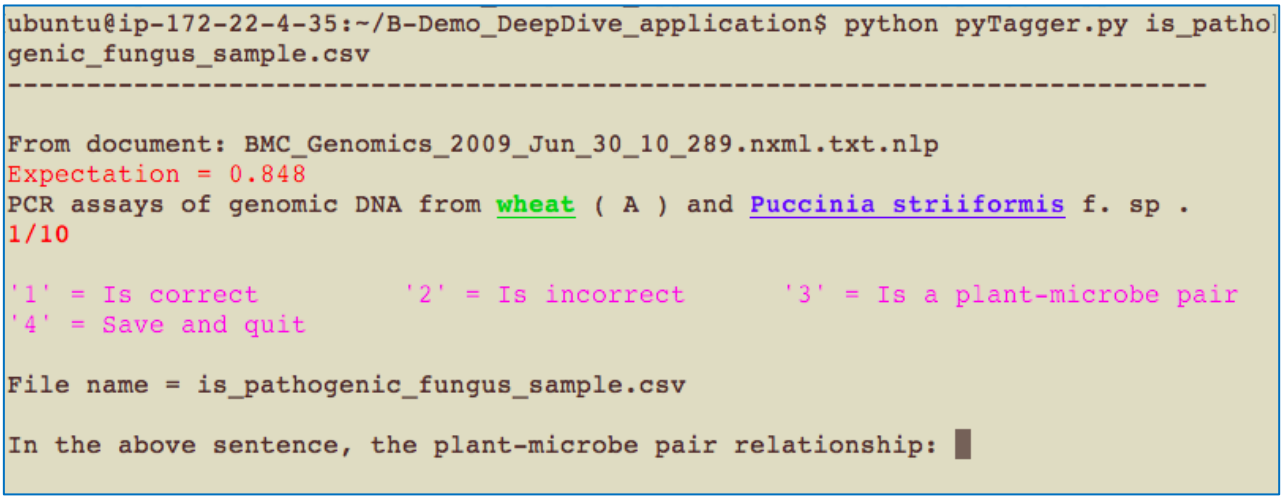
1. **Create a sample of DeepDive predictions**
   Let us query the DeepDive database to get a **random sample of 100 sentences** with plant-fungus pairs with an **expectation of at least 80%**. In the terminal, enter the following SQL query:
   ```
   deepdive sql eval "SELECT ipm.plant_id, ipm.fungus_id, s.doc_id,
   s.sentence_index, ipm.label, ipm.expectation, s.tokens,
   plant_mention.mention_text AS plant_text, plant_mention.begin_index AS
   plant_start, plant_mention.end_index AS plant_end,
   fungus_mention.mention_text AS fungus_text, fungus_mention.begin_index AS
   fungus_start, fungus_mention.end_index AS fungus_end FROM
   is_pathogenic_fungus_label_inference ipm, plant_mention plant_mention,
   fungus_mention fungus_mention, sentences s WHERE ipm.plant_id =
   plant_mention.mention_id AND plant_mention.doc_id = s.doc_id AND
   plant_mention.sentence_index = s.sentence_index AND ipm.fungus_id =
   fungus_mention.mention_id AND fungus_mention.doc_id = s.doc_id AND
   fungus_mention.sentence_index = s.sentence_index AND expectation >= 0.8
   ORDER BY random() LIMIT 100" format=csv header=1 >
   is_pathogenic_fungus_sample.csv
   ```
   This will create the file **is_pathogenic_fungus_sample.csv** which will be used as input for PyTagger. This file has the following columns: plant_id, fungus_id, doc_id, sentence_index, label, expectation, tokens, plant_text, plant_start, plant_end, fungus_text, fungus_start, fungus_end.
   The python script to generate samples for both plant-fungus and plant-prokaryote pairs ca be found in **Supplementary Folder B** and it's called **makeSamples.py**. To run it, simply enter `python makeSamples.py`. Make sure makeSamples.py is in the same folder as app.ddlog.

2. **Initialize PyTagger**

   To initialize PyTagger, simply run:

   `python pyTagger.py is_pathogenic_fungus_sample.csv`

   The following picture shows what pyTagger looks like on the terminal



   The first row printed on the screen is the **document_**id, the second row shows the expectation (a value from 0.8 to 1 in our case), the third is the actual **sentence**, the fourth line says there are 10 sentences in the sample and that we are taking a look at the first one now (1/10). Options are printed in pink in the fifth line. Finally, the name of the file that is being revised in printed in the sixth line.

3. **Label and save tagged examples**

   To tag each example in the sample, this is what you should do:

   - Enter (1) if the plant-microbe pair is indeed a plant-pathogen candidate given the context of the sentence where it appears.
   - Enter (2) if the plant-microbe is not a plant-pathogen candidate
   - Enter (3) if the plant-microbe pair cannot be determined to be plant-pathogen with the text in the sentence
   - Enter (4) to save the tagged examples and quit pyTagger

   After going over all sentences in the sample, PyTagger will save the results as **is_pathogenic_fungus_sample_tagged.csv**. The columns in this file are: plant_id, microbe_id, plant_name, microbe_name, doc_id, is_correct.

4. **Add tagged examples to the /input folder**

   Run `mv is_pathogenic_fungus_sample_tagged.csv ./input`

5. **Declare schema of this file in app.ddlog**

   Because we want to feed these tagged examples back into our DeepDive application, it is important that we declare its schema before we use it as feedback. Open app.ddlog and add the following:

```
is_pathogenic_fungus_sample_tagged.csv(
plant_id text, microbe_id text, plant_name text, microbe_name text,
doc_id text, is_correct text).
```

6. **Label candidate pairs according to the new training examples**
   We can now create new labels for each candidate pair by adding the following code to our app.ddlog (note the similarity between this code and that of section 8.2)

```
# Assign a "1" label to all candidates matching the tagged examples

pathogenic_fungus_label(plant_id, fungus_id, 1,
"from_tagged_examples") :-
      plant_fungus_candidate(plant_id, _, fungus_id, _),
is_pathogenic_fungus_sample_tagged(plant_id,fungus_id,_,_,_,is_correct),
is_correct = "true".
```

7. **Re-run DeepDive application and iterate for improving**
   Lastly, we can re run the labeling and the predictions with the new examples to see how the predictions improved (note the "redo" command instead of "do").

```
deepdive compile
deepdive redo pathogenic_fungus_label
deepdive redo is_pathogenic_fungus
deepdive redo probabilities
deepdive redo calibration-plots
```

   The exact approach can be used to refine predictions for **is_pathogenic_prokaryote**. Note also that the dataset we used in this tutorial is very small. Better results are obtained by increasing the size of the text corpus. Alternatively, an easier way to re-run the entire DeepDive application with fewer commands is as follows:

```
deepdive compile
deepdive db init
deepive run
```

   An alternative method for labeling examples using a Graphic User Interface called **Mindtagger** is described in **Appendix D**.

Appendix A. Converting PUBMED abstracts to an appropriate sentences.tsv file

Here we will explain how to use the scripts in **Supplementary Folder C** to convert PUBMED abstracts to a suitable sentences.tsv file for deploying DeepDive apps using the CoreNLP capabilities of DeepDive.

Let's suppose that we want to expand the text corpus of our Plant-Pathogen app by incorporating PUBMED abstracts from different journals. The problem is that the text in PUBMED abstracts is plain and it is not in the form parsed sentences like in the BMC, PMC and PLOS deepdive open datasets. However, we can convert the raw text of PUBMED abstracts to a sentences.tsv file by following these steps:

1. **Download abstracts from PUBMED**
    - Go to www.ncbi.nlm.nih.gov/pubmed
    - Enter the search terms in the search bar at the top of the page. For this example, we will enter the search term "plant pathogen"
    - From the results (22,317 hits in PUBMED as of September 12th 2016), filter out those items that lack an abstracts by clicking "Abstract" under **Text availability** in the left menu
    - Click "**Send to**" on the upper right side of the page. Then check **File** under "Choose destination" and select **XML** under "Format" and click **Create File**.
    - Place the downloaded .xml file (usually it's called pubmed_result.xml) inside the **Supplementary Folder C-Converting_PUBMED_abstracts**
    - Navigate to the Supplementary Folder C in the terminal and run
      `python getPUBMEDabstractsFROMxml <your-downloaded-file>.xml`
    - This will output a tab-separated-value file with the same name as <your-downloaded-file>. In our case for this example, we downloaded the ten first hits from PUBMED as pubmed_result.xml and created an output file called pubmed_result.tsv (see Supplementary Folder C).
2. **Create sentences from pubmed abstracts**
    - Place the .tsv file from step 1 inside the **/input** directory under **Supplementary Folder C**.
    - Change the name of the .tsv file to **pubmed_result.tsv**
    - Exit the /input folder by entering `cd ..`
    - Enter the following commands under the Supplementary Folder C directory:
        - `deepdive compile`
        - `deepdive do pubmed_result`
        - `deepdive do pubmed_sentences`
    - If all is done correctly, DeepDive will produce a table inside PostgreSQL database with all the sentences from the pubmed abstracts. Note that this step might take a long time depending on the number of downloaded abstracts due to the NLP processing time of DeepDive
3. **Export sentences table and save it as an input**

We can now create a .tsv file of our sentences by entering:

```
deepdive sql
```

Once inside PostgreSQL, enter:

```
\COPY pubmed_sentences TO pubmed_sentences.tsv
```

We now have a .tsv file with DeepDive-compatible sentences that we can use in our Plant-Pathogen app to expand our corpus.

Appendix B. Reusing an existing DeepDive database as template for new applications

Suppose you are working with a large sentence dataset like the full-text articles from PLOS (~70 GB). Doing so requires one to have the plos_sentences.tsv file inside the /input folder, define the appropriate schema in app.ddlog and then load it onto the DeepDive databse with `deepdive do plos_sentences`. Because there's a 70 GB file in our input folder and a sentences table is created in PostgreSQL with the same data in it, just loading the sentences in our DeepDive application alone would require about 140GB or disk space! You can start to see how problematic this would be if you are to use the PLOS dataset in more than one DeepDive application.

To go around this problem, we have three alternatives: we can either (A) have a single plos_sentences.tsv file stored in a master directory which will be loaded every time we create a new DeepDive app or (B) we can clone an existing DeepDive database with the plos_sentences table in it and reuse it in a new application or (C) we can move the **tablespace** of the database to a file location that as much more available space.

A. **Load a large dataset from a directory that is not /input**
   1. Save the large dataset in a master directory. For instance, save the PLOS dataset (the file is called plos_sentences.tsv) in a directory called PLOS_CORPUS under the **home** folder (i.e. /home/PLOS_CORPUS)
   2. Create a blank file (.tsv or .csv) inside the /input folder of your application. The name of this blank file should correspond to that of the sentences schema defined in app.ddlog. In our example, we will create a file called **plos_sentences.tsv** inside the /input folder with nothing written in it (just do `touch plos_sentences.tsv` in the /input folder).
   3. Run the following commands:
      ```
      deepdive compile
      deepdive create table plos_sentences
      deepdive load plos_sentences /home/PLOS_CORPUS/plos_sentences.tsv
      deepdive mark done plos_sentences
      ```
   4. More information can be found here http://deepdive.stanford.edu/ops-data

B. **Clone an existing DeepDive database to use it in a new app**
   1. Open Postgres from within a DeepDive app folder with `deepdive sql`
   2. Run the following command while replacing the variables (<>) by the appropriate names for your specific case
      ```
      CREATE DATABASE <newDB> WITH TEMPLATE <originalDB> OWNER <$USER>;
      ```

C. **Move the tablespace of a DeepDive database to a location with more available space**
   1. Create a folder within the location in the computer that contains more space (e.g. a mounted hard disk, file share system, etc.). Here we will assume we have a mounted disk under /data/deepdive/ so, we do: `mkdir /data/deepdive/new_tablespace`
   2. Change the owner of this new tablespace to **postgres** with `chown postgres /data/deepdive/new_tablespace`

3. Open Postgres within a deepdive application **different from the one we will move** and run the following:
   ```
   CREATE TABLESPACE new_tablespace
   LOCATION '/data/deepdive/new_tablespace';
   ```
4. Move the DeepDive application to the new tablespace (to see the name of your deepdive databases simply run `\l`):
   ```
   ALTER DATABASE <deepdive_database_name> SET TABLESPACE new_tablespace
   ```

Appendix C. Changing the location of the PostgreSQL server

1. Configure a password for the **postgres** user (here we will set the password to *deepdivepass*)

   ```
   sudo passwd postgres
   Enter new UNIX password: deepdivepass
   Retype new UNIX password: deepdivepass
   passwd: password updated successfully
   exit
   ```

2. Create and configure folder where you would like to host the new PostgreSQL server and databases (here we will create a directory under /my-data/ called DATABASE)
   NOTE: We are using PostgreSQL 9.3
   ```
   cd /my-data
   mkdir DATABASE
   /usr/lib/postgresql/9.3/bin/initdb –D /my-data/DATABASE
   ```

3. Stop the current PostgreSQL server
   ```
   sudo service postgresql stop
   ```

4. Update the file **/etc/postgresql/9.3/main/postgresql.conf**
   Open the file and change **data_directory = '/var/lib/postgresql/9.3/main'** to **data_directory = '/my-data/DATABASE'.**
   Save and exit the text editor.

5. Restart the PostgreSQL server
   ```
   sudo service postgresql start
   ```

6. Create superuser so that we can continue using DeepDive commands as normal in this new server
   ```
   sudo –u postgres createuser –superuser $USER
   ```

The instructions in this post are based on the following post:

http://climber2002.github.io/blog/2015/02/07/install-and-configure-postgresql-on-ubuntu-14-dot-04/

Appendix D. Initializing a web server for Mindbender and Mindtagger

Mindbender
Mindbender is an interactive search interface for browsing DeepDive-generated results and input data in a web page. In order to run mindbender for a DeepDive application, it is highly advisable that the reader understands the material in this tutorial as well as in http://deepdive.stanford.edu/browsing before proceeding.

We have prepared a ready-to-run DeepDive application with appropriate Mindbender files under the **Supplementary Folder D-mindbender_example_app**. This folder contains a DeepDive application that predicts whether two persons are spouses. Note that this app contains a folder called **/mindbender** in addition to the usual **/input** and **/udf** folders. The details of the files inside the **/mindbender** folder can be read in http://deepdive.stanford.edu/browsing.
To initialize the Mindbender application and browse the contents of the DeepDive database on the web, we must navigate to the **spouse-app** folder inside **Supplementary Folder D** and enter the following commands in the Terminal:

(Give appropriate permissions to all files in udf)

```
chmod a+x ~/spouse-app/udf/*
```

(Run the spouse-app application to create a DeepDive database along with its predictions)
```
deepdive compile
deepdive run
```
(Enter ":wq" to save and run the vi file when prompted)

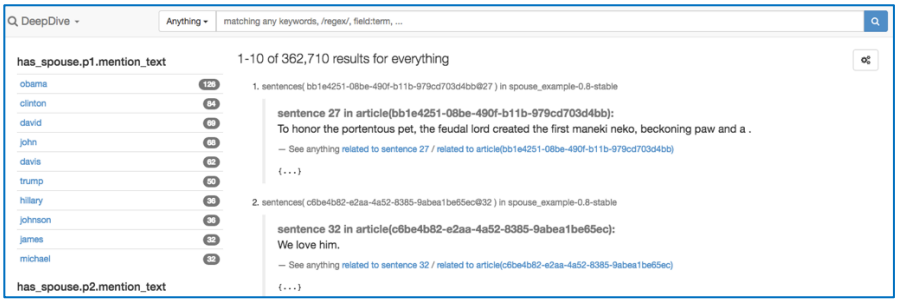
(Initialize Mindbender search engine to load the database onto Mindbender, this step might take a significantly long time to complete based on the size of the Database)
```
mindbender search update
```

(Initialize a web server to open Mindbender in a web browser outside the EC2 instance)
```
mindbender search gui
```

This will print the IP address of the EC2 instance along with the port being used by the server (usually 8000), in our example it would look like this:

```
ubuntu@ip-172-22-4-35:~/spouse_example-mindbender$ mindbender search gui
Launching Elasticsearch for http://localhost:9200 from /home/ubuntu/spouse_example-mindbender/search
20 Sep 17:36:12 - Loaded 0 Mindtagger tasks:
20 Sep 17:36:12 - Mindbender GUI started at http://ip-172-22-4-35:8000/
```

Thus, on a web browser, we go to http://172.22.4.35:8000/ (note that we replaced the dashes by dots in the IP address). Note that if you would like to run this from a different port you can do:
```
PORT=12345 mindbender search gui
```



Mindtagger

Mindtagger is a very useful tool to label DeepDive predictions on a web browser instead of doing it in the Terminal as we did with PyTagger (see section 10). Mindtagger provides a full Graphic User Interface and requires many less files to work in contrast to Mindbender. Additionally, the loading time is quite short and it can be easily deployed for multiple instances using one single EC2 instance.

Here, we will create a Mindtagger application using the results from our Plant-Pathogen app. Please read the Mindtagger documentation in http://deepdive.stanford.edu/labeling

We have included a folder called **/labeling** under the **Supplementary Folder B** which contains all the files necessary to run a Mindtagger task for labeling Plant-Fungi and Plant-Bacteria pairs for training the machine learning system. Note that the .csv files inside each task were generated with **makeSamples.py**

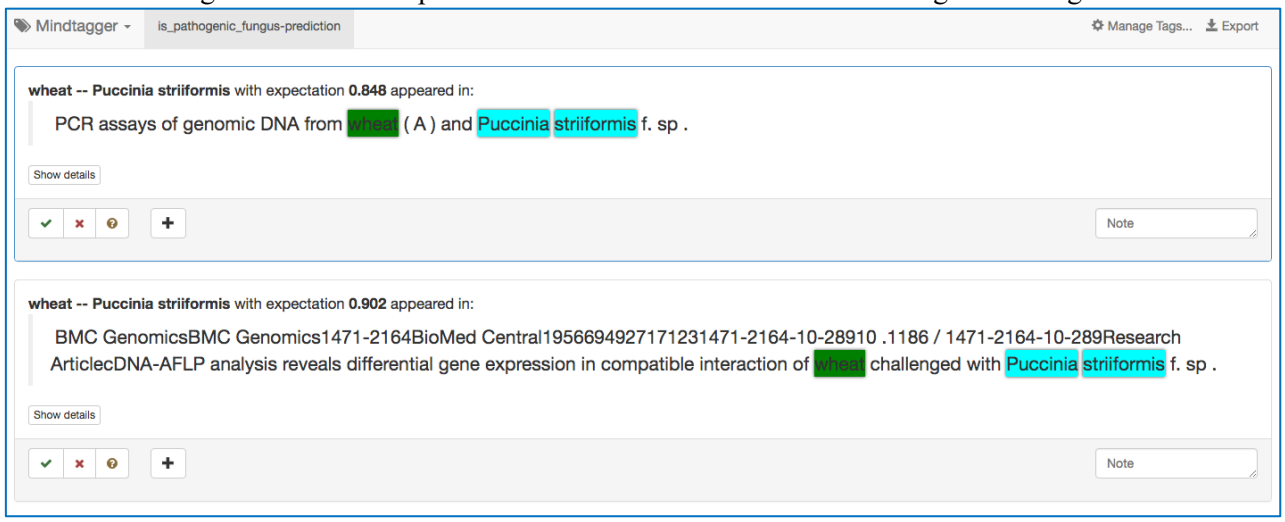
Basically, the way we develop a Mindtagger application is as follows:

1. First, we take a sample from our predictions using the **makeSamples.py** script
2. Second, we create a /**labeling** folder in our main DeepDive app directory (i.e. in the same location as the /input and /udf folders)
3. Inside the /labeling folder, we create two subdirectories: one for the plant-fungus predictions and one for the plant-prokaryote predictions (called "is_pathogenic_fungus-prediction" and "is_pathogenic_prokaryote-prediction" in our Supplementary Folder B/labeling folder)
4. Each subdirectory will have three files:
    a. **is_pathogenic_fungus(prokaryote).csv** (this is the sample file created from the DeepDive predictions with **makeSamples.py**)
    b. **mindtagger.conf** (dictates the name of the .csv file containing the data that will be loaded)
    c. **template.html** (determines how the text will be displayed in the browser. It is important to pay special attention to how the name of the columns in the .csv file must correspond to the item calls in the template.html file)
5. Once the files are ready, we navigate to our main app folder and we enter the following command in the Terminal:
   ```
   mindbender tagger ./labeling/is_pathogenic_fungus-prediction/mindtagger.conf
   ```
6. This will display an url in the screen which we will navigate to from our web browser

```
ubuntu@ip-172-22-4-35:~/B-Demo_DeepDive_application$ mindbender tagger labeling/is_pathogenic_fungus-prediction/mindtagger.conf
20 Sep 18:06:44 - Mindbender GUI started at http://ip-172-22-4-35:8000/
20 Sep 18:06:44 - Loaded Mindtagger task is_pathogenic_fungus-prediction
20 Sep 18:06:44 - Loaded 1 Mindtagger tasks: is_pathogenic_fungus-prediction
```

Mindtagger can now be opened in http://172.22.4.35:8000/

Again, see the documentation in http://deepdive.stanford.edu/labeling for more details about exporting results and adding labels to the sample data. The window should look something like the figure below



The results can be exported to .tsv or .csv file and fed back to our DeepDive application using the same procedure outlined in Section 10.