iLabs Research
Summer 2017

Jaideep Patel
jaideep.ptl@gmail.com

Query Spell Check and NER Model

Contents:

1. Knowledge Discovery:

We began exploring the possible problems to work on by referring to Daniel Tunkelang's query understanding blog. I learnt about the techniques and tools available for performing spell correction.

There are 2 potential high-level routes towards building the spell check correction models:

- Method1: Building a word context model using historically "correct" annotated queries.
  - This uses the Bayes theorem to score the spelling correction candidates and rank them based on their probability.
- Method2: Using the approximate string matching Hunspell model.
  - Here use make use of one or more English word dictionaries and affix files for English rule to provide suggestions for the identified misspelled words.

Challenges/drawbacks of spell check models:

- Method1:
  - It is too expensive and time consuming to pay a linguist to manually identify and annotate the historical user search queries as correctly and incorrectly spelled.
  - It is difficult to find train data from historical user logs that provides the potential corrections that the users previously manually made on their search queries, which will help us design a robust probabilistic model.
- Method2:

- Not all words are available in English dictionaries.
- It uses a simple string matching Levenshtein edit distance algorithm, with no smarter modeling or any historically available train data.

2. Proposed Method:

Based on the positives and limitations of the methods in hand we moved ahead with the Hunspell spell check library. We use the Hunspell 1.6.2 spell check library.

Hunspell Overview:

Hunspell is a spell checker and morphological analyzer library and program designed for languages with rich morphology and complex word compounding or character encoding. Hunspell interfaces: Ispell-like terminal interface using Curses library, Ispell pipe interface, C++ class and C functions.

3. Setting up Development Environment for Hunspell library:

We use the ilabs-gpus Linux machine to install the Hunspell library and build the spell check models.

Instructions to install Hunspell and pyhunspell on Linux machine:

Hunspell: (http://hunspell.github.io/)

Pyhunspell: https://github.com/blatinier/pyhunspell

Hunspell is the spell checker of LibreOffice, OpenOffice.org, Mozilla Firefox 3 & Thunderbird, Google Chrome, and it is also used by proprietary software packages, like macOS, InDesign, memoQ, Opera and SDL Trados.

This link explains the Hunspell format for Hunspell dictionaries and affix files.

Software Requirements:

Operating system: Linux Environment

Software: Python (version 2.7 or later) / anaconda (recommended).

Installing hunspell:

Step1: Install python on your Linux environment.

Anaconda provides the required python resources. Use the link to download and install anaconda on Linux.

After installing anaconda on Linus use the below command to install the python-dev package:

```
sudo apt-get install python-dev
```

Run the following commands to update conda’s c++ library (if using anaconda):

```
$ conda install libgcc
```

Step2: Steps for compiling Hunspell library on Linux:

Download the latest release of the Hunspell library from the link here.

Extract the zip of the tar file to your work directory.

Run the following commands on your linux terminal to install required packages:

```
autoconf

automake

autopoint

libtool

g++
```

Navigate to the Hunspell folder and run the following commands

```
sudo apt-get install autoreconf -vfi

./configure

make

sudo make install

sudo ldconfig
```

Install libhunspell-dev package

```
sudo apt-get install libhunspell-dev
```

Step4: Now install the hunspell library using the command:

```
sudo apt-get install pip

pip install hunspell
```

Note: If using a later release of pyhunspell library from another sources, where the library is not included in the pypi package index, please refer the corresponding documentation for the instructions. You need to use the python setup.py install command.

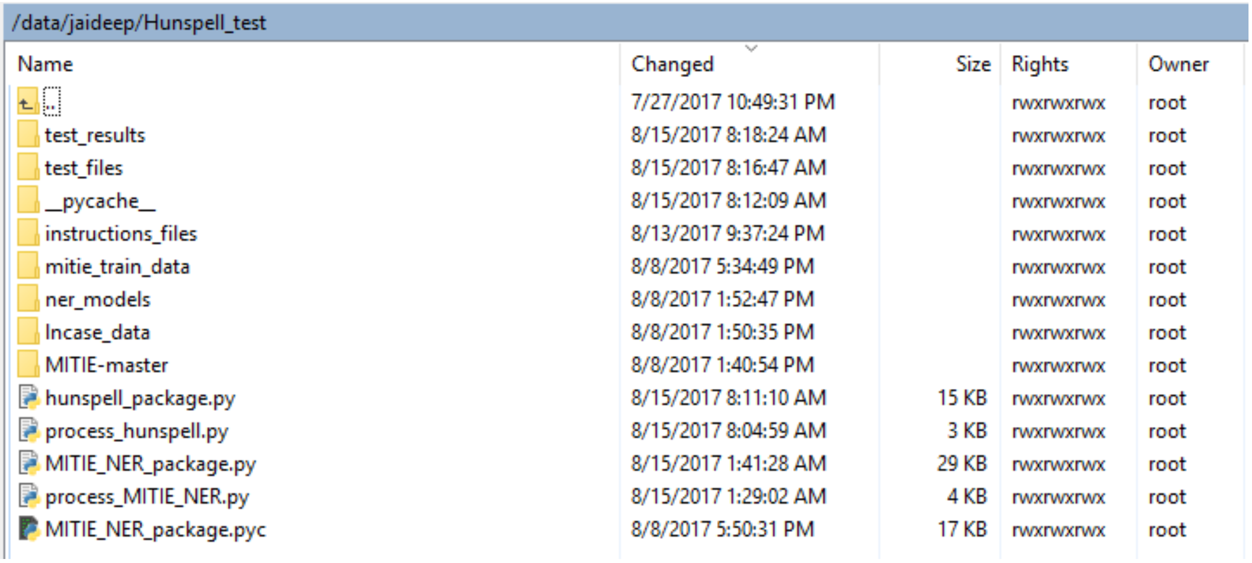
Step5: Now open the python environment and import the pyhunspell library:

```
import hunspell
```

This completes the hunspell library installation process.

4. Code details and Instructions:

Below is a screenshot of the code directory:



/data/jaideep/Hunspell_test

Name	Changed	Size	Rights	Owner
..	7/27/2017 10:49:31 PM		rwxrwxrwx	root
test_results	8/15/2017 8:18:24 AM		rwxrwxrwx	root
test_files	8/15/2017 8:16:47 AM		rwxrwxrwx	root
__pycache__	8/15/2017 8:12:09 AM		rwxrwxrwx	root
instructions_files	8/13/2017 9:37:24 PM		rwxrwxrwx	root
mitie_train_data	8/8/2017 5:34:49 PM		rwxrwxrwx	root
ner_models	8/8/2017 1:52:47 PM		rwxrwxrwx	root
lncase_data	8/8/2017 1:50:35 PM		rwxrwxrwx	root
MITIE-master	8/8/2017 1:40:54 PM		rwxrwxrwx	root
hunspell_package.py	8/15/2017 8:11:10 AM	15 KB	rwxrwxrwx	root
process_hunspell.py	8/15/2017 8:04:59 AM	3 KB	rwxrwxrwx	root
MITIE_NER_package.py	8/15/2017 1:41:28 AM	29 KB	rwxrwxrwx	root
process_MITIE_NER.py	8/15/2017 1:29:02 AM	4 KB	rwxrwxrwx	root
MITIE_NER_package.pyc	8/8/2017 5:50:31 PM	17 KB	rwxrwxrwx	root

Folder details:

test_results: All the test results will be saved to this folder. The result file name will be same as the test file name.

test_files: All the test files are to be uploaded in this folder. The test files could be individual files or subfolder if needed to test of multiple test files.

Instructions_files: There are respective instruction files for processing Hunspell and NER tests.

File name: hunspell_instructions.txt

File format: file_name, dictionaries list, top-n suggestions, expected results or not (Y or N), expected results format (span, nonspan, full-exp)

Specifications:

File_name: Enter the full file name (Example: dymWithoutClicks.csv)

List of dictionaries: give all dictionary names separated by <space> (Example: En_US.dic, Black.dic)

Top-n suggestions: This parameter is to return the top n suggestions from Hunspell for each test query.

Expected results: This parameter mentions if the test file contains expected results. This will enable the code to find the accuracy and precision values.

Expected results format: There are 3 different types of formats of expected results;

Span format: When the misspelled words in the expected results column are encoded within the span tags. Example:

Query: verdict upheld for 21 year old female struck by vehicle knee allograft surgery

Expected results: verdict upheld for 21 year old female struck by vehicle knee <span>allograph</span> surgery.

Non-span format: When the misspelled words in the expected results column are separated by '#'. Example:

Query: court denie without addressing "Cross motion" for summary judgment

Expected results: denies # judgement

Full-exp format: When the expected results is a full clean query. Example:

Query: verdict upheld for 21 year old female struck by vehicle knee allograft surgery

Expected results: verdict upheld for 21 year old female struck by vehicle knee allograph surgery.

File format of the test file:

Column names for csv file:

"Query": Column name for queries.

"Expected Results": Column for expected results.

Txt file or dat file format:

Every line in the text file is a query with no headers.

If the text file also contains the expected results, in this case the current code expects the test file to be saved as a csv file with col names as "Query" and "Expected Results".

Important Note: All the test files must be in '**utf-8**' encoding.

Code Overview:

The hunspell_pachage.py file contains all the functions needed to process and run the spell checker on the test file.

The process_hunspell.py file is the trigger file to read the instructions and call the functions in the hunspell_package.py file to process the Hunspell suggestions on the test queries.

For in-depth details about the functions in the Hunspell package please refer to the code for comments and directions.

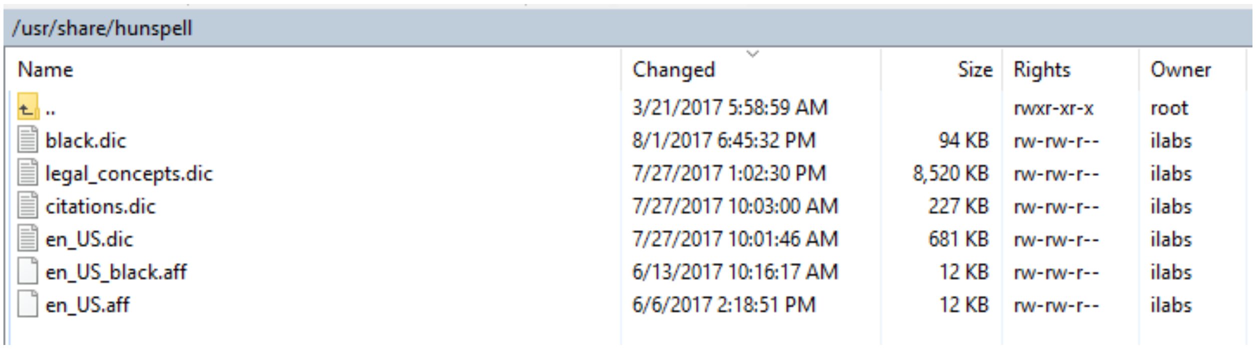
Prerequisites:

1. Hunspell library needs to be install on the working environment.
2. Install the python bindings of Hunspell either using the pyhunspell git repository or by using the standard python pip installer. (https://github.com/blatinier/pyhunspell)

Instructions to add additional dictionaries to the Hunspell library:

Locate the repository where the Hunspell libraries are loaded. In our case, for Linux environments the general path is: /usr/share/Hunspell

Users can add additional dictionaries to this folder and add them to the Hunspell object during runtime.



/usr/share/hunspell

Name	Changed	Size	Rights	Owner
..	3/21/2017 5:58:59 AM		rwxr-xr-x	root
black.dic	8/1/2017 6:45:32 PM	94 KB	rw-rw-r--	ilabs
legal_concepts.dic	7/27/2017 1:02:30 PM	8,520 KB	rw-rw-r--	ilabs
citations.dic	7/27/2017 10:03:00 AM	227 KB	rw-rw-r--	ilabs
en_US.dic	7/27/2017 10:01:46 AM	681 KB	rw-rw-r--	ilabs
en_US_black.aff	6/13/2017 10:16:17 AM	12 KB	rw-rw-r--	ilabs
en_US.aff	6/6/2017 2:18:51 PM	12 KB	rw-rw-r--	ilabs

5. Testing Hunspell:

All the test files are in the test_files folder and the corresponding results are available in the test_results folder.

The Query files are tested using the stock Hunspell dictionary and affix files with extracting the top 3 suggestions for every test token from the query

The stock Hunspell dictionary is a list of 62,154 English words + most of the combinations of their possible prefix and suffix extension words.

Test on Hyphen words:

Testing on dot words:

Testing on Concatenate word forms:

Key findings:

The Accuracy values are good when the queries contains regular misspelled English words. The Hunspell library performs poorly when there is a high percentage of words like person names, citations, section names, non-english words, abbreviations and specific legal entity words.

Two Big Challenges:

From the results above we see that the Hunspell library produces high volumes of "**False positives**" primarily because it does not recognize specific entity names and citation, case names, section words. Handling this problem will improve the precision of the spell check system.

The challenge of **finding the best suggestion** among the top suggestions from Hunspell. Handling this problem will improve the relevance/accuracy of the spell check system.

Overview of Code details:

The Hunspell library is first install on the linux – ilabs-gpus machine.

Install the standard pyhunspell library.

Python code to preprocess the query files and run through the Hunspell library.

Python code processing time:

- Total Queries: 1890
- Total time taken to preprocess, spell check and save the accuracy and precision results to a csv file: 38 seconds.
- It takes less than 0.02 seconds to process a query and provide to Hunspell suggestions.

6. Potential ideas for future scope:

Abbreviations:

Build annotated query training data with queries having abbreviations to identify abbreviations as entities with a robust entity recognizer. Like Google we could build a large collection of abbreviations and their corresponding expansions that could be used as a dictionary map of use the abbreviations dictionary in Hunspell.

Unseen English words in the dictionaries:

Add the corresponding word to the Hunspell dictionaries by tracing user logs and reading user corrections for a few initial first instances. Build a context based model for example a word2vec that returns the context of a word. In our case we use the words surrounding the misspelled words to predict the incorrect word by training the model on large case volumes law documents.

Non-English terms:

Build non-English Hunspell dictionaries.

Improvements to Spell check model using NER model

As seen earlier, we notice that the Hunspell fails when the queries contain high percentage of entities as a result the Hunspell flags most of the tokens as 'Misspelled' and tries to suggestions. This result is high percentage of 'False positives'. To encounter this problem, it is very important to build a robust entity recognition model that identifies a set of legal/non-legal entities in the queries and skip those words from being processed by the Hunspell model.

We initially looked at a few pre-trained entity recognizer available as a free source and there are many available on the web which perform with high accuracy but have these **prerequisites**:

- They assume that the text has proper English syntax and grammar. Queries as we know lack structure and grammar.
- High percentage of entity recognition models use POS tagging which in itself is a challenging problem in queries.
- Many pretrained models are not designed to identify entities

Some other data challenges:

- We lack annotated (entity recognized) historical query data that could be used for training an entity recognition model.

7. Proposed NER Model:

MITIE is built on top of dlib, a high-performance machine-learning library.

MITIE makes use of several state-of-the-art techniques including the use of distributional word embeddings and Structural Support Vector Machines.

Advantages of Using MITIE:

- It does not use POS tagging.
- It uses a robust structural SVM and sequential labeling algorithm using custom random fields.

References:

- T. Joachims, T. Finley, Chun-Nam Yu, Cutting-Plane Training of Structural SVMs, Machine Learning, 77(1):27-59, 2009.
- Paramveer Dhillon, Dean Foster and Lyle Ungar, Eigenwords: Spectral Word Embeddings, Journal of Machine Learning Research (JMLR), 16, 2015.
- Davis E. King. Dlib-ml: A Machine Learning Toolkit. Journal of Machine Learning Research 10, pp. 1755-1758, 2009.

8. Setting up Development Environment and MITIE library:

Software Requirements:

Operating system: Linux Environment

Software: Python (version 2.7 or later) / anaconda (recommended).

Installing MTIE NER package:

MITIE package is directly available among the collection of packages in the pypi package index, bet we recommend to pull the latest version from the official GitHub repository using the below command:

If you are using a UNIX system, you can also install mitie package direcly from github:

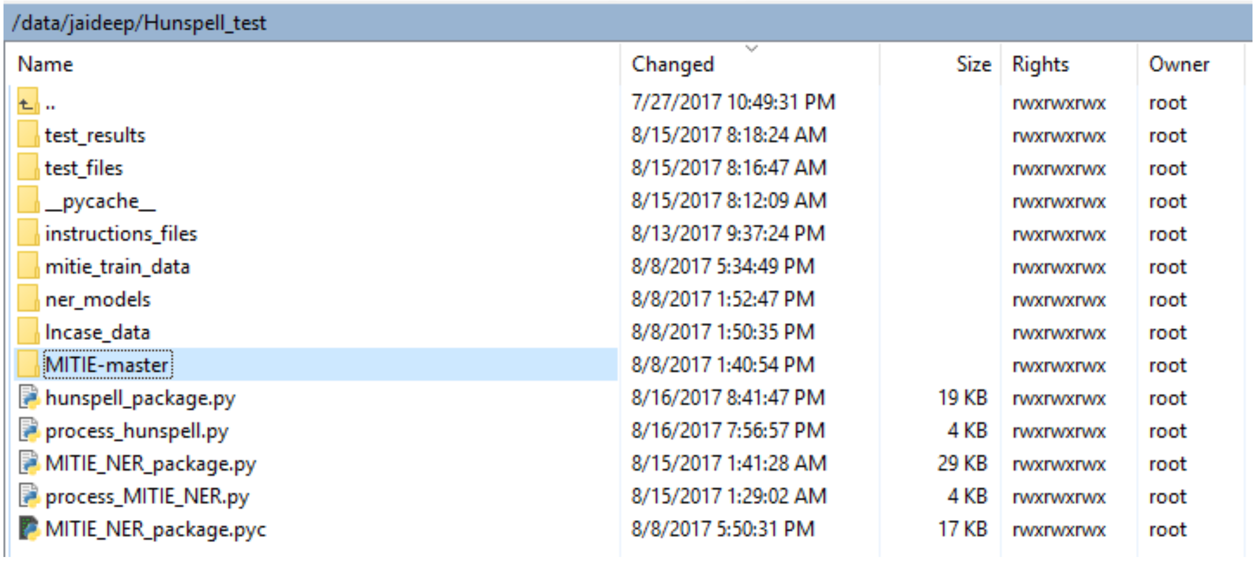
pip install git+https://github.com/mit-nlp/MITIE.git

Recommended:

Download the MITIE GitHub repository, extract the root MITIE-Master folder and copy it to the working directory.

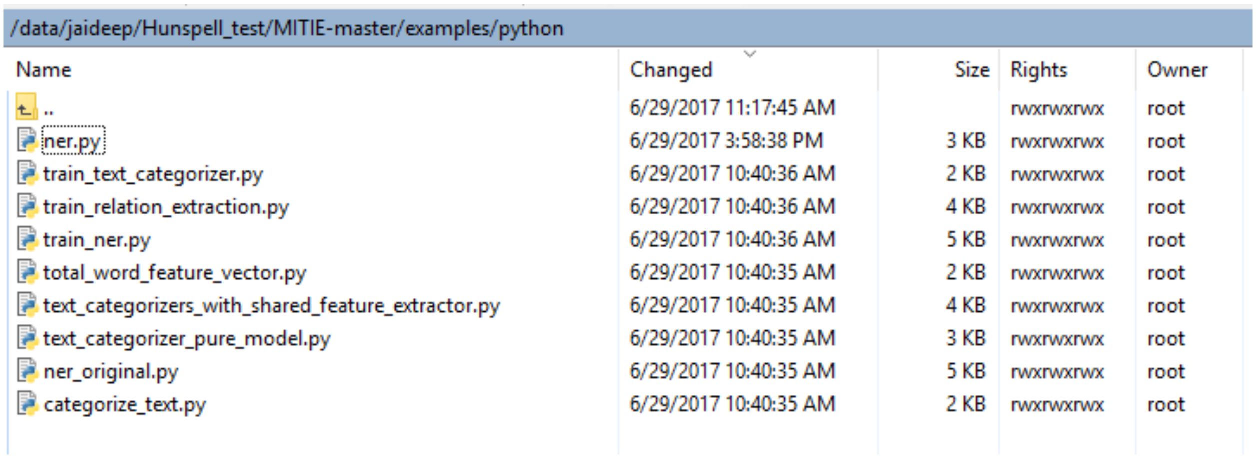
There are two reasons to do so:

1. The package contains a pretrained NER model that could be used to perform some initial cases and understand how the pretrained model performs of your test data.
2. The examples folder (MITIE-Master -> examples -> python) contains example python files for reference on how to train and test the MITIE NER model.
3. We need to look up the 'total_word_feature_extraction.py' file to perform the feature extraction on the train data while training the NER model. You will find additional details about the usage of it in the project code.



/data/jaideep/Hunspell_test

Name	Changed	Size	Rights	Owner
..	7/27/2017 10:49:31 PM		rwxrwxrwx	root
test_results	8/15/2017 8:18:24 AM		rwxrwxrwx	root
test_files	8/15/2017 8:16:47 AM		rwxrwxrwx	root
__pycache__	8/15/2017 8:12:09 AM		rwxrwxrwx	root
instructions_files	8/13/2017 9:37:24 PM		rwxrwxrwx	root
mitie_train_data	8/8/2017 5:34:49 PM		rwxrwxrwx	root
ner_models	8/8/2017 1:52:47 PM		rwxrwxrwx	root
Incase_data	8/8/2017 1:50:35 PM		rwxrwxrwx	root
MITIE-master	8/8/2017 1:40:54 PM		rwxrwxrwx	root
hunspell_package.py	8/16/2017 8:41:47 PM	19 KB	rwxrwxrwx	root
process_hunspell.py	8/16/2017 7:56:57 PM	4 KB	rwxrwxrwx	root
MITIE_NER_package.py	8/15/2017 1:41:28 AM	29 KB	rwxrwxrwx	root
process_MITIE_NER.py	8/15/2017 1:29:02 AM	4 KB	rwxrwxrwx	root
MITIE_NER_package.pyc	8/8/2017 5:50:31 PM	17 KB	rwxrwxrwx	root



/data/jaideep/Hunspell_test/MITIE-master/examples/python

Name	Changed	Size	Rights	Owner
..	6/29/2017 11:17:45 AM		rwxrwxrwx	root
ner.py	6/29/2017 3:58:38 PM	3 KB	rwxrwxrwx	root
train_text_categorizer.py	6/29/2017 10:40:36 AM	2 KB	rwxrwxrwx	root
train_relation_extraction.py	6/29/2017 10:40:36 AM	4 KB	rwxrwxrwx	root
train_ner.py	6/29/2017 10:40:36 AM	5 KB	rwxrwxrwx	root
total_word_feature_vector.py	6/29/2017 10:40:35 AM	2 KB	rwxrwxrwx	root
text_categorizers_with_shared_feature_extractor.py	6/29/2017 10:40:35 AM	4 KB	rwxrwxrwx	root
text_categorizer_pure_model.py	6/29/2017 10:40:35 AM	3 KB	rwxrwxrwx	root
ner_original.py	6/29/2017 10:40:35 AM	5 KB	rwxrwxrwx	root
categorize_text.py	6/29/2017 10:40:35 AM	2 KB	rwxrwxrwx	root

9. Code details and Instructions MITIE:

Folder details:

test_results: All the test results will be saved to this folder. The result file name will be same as the test file name.

test_files: All the test files are to be uploaded in this folder. The test files could be individual files or subfolder if needed to test of multiple test files.

lncase_data: This folder contains all the documents that needs to be used to train the NER model. This could contain a single file or a directory containing multiple files to be used to train the model.

ner_models: This folder is used to save the NER trained models as a .dat file.

mitie_train_data: This folder contains the sentence entity MITIE format train data files that will be directly used to train the NER model.

Instructions_files: There are respective instruction files for processing NER training and testing:

1. **Preprocessing case law documents to MITIE format and train NER model:**
   Instruction file name: mitie_train_instruction.txt
   File format: file_name, directory_name, file format
   File_name: Enter the full file name (Example: 2000cases.xml)
   Dir_name: Enter the directory name under the data folder to preprocess the case law files in the respective directory (Example: ner_train_data)
   File_type: Mention if the data file is in xml, dat format (Example: xml)

Note: Either pass the file name of the directory name – but not both at the same time.

2. **Preprocessing case law test documents to MITIE format:**
   Instruction file name: mitie_process_test_instructions.txt
   File format: The format is the same as the mitie_train_instruction.txt format.
3. **Testing NER:**
   Instruction file name: mitie_test_instructions.txt
   File format: test_file_format, file_type, file_name, dir_name, exp_value, model_name
   Example: mitie, txt, ner_test_files, T, ner_models/2000_cases_model.dat
   Test_file_format: Options are – 'DOCUEMENT', 'SENT', 'QUERY'
   Document – This indicates that the test file contains a set pf paragraph text of a collection of sentences just like a regular text document.
   Sent – This indicates that the test file contains each sentence in every line of the test (.txt) file.
   Query - This indicates that the test file contains one query in every line of the test (.txt, .csv) file.
   File_type: Options are txt and csv
   Txt file – This indicates that the test file does not have any expected values and the file format is one among the above-mentioned formats.
   Csv file – We use the csv file format only when the test file contains the expected results.
   Columns in csv file:
   'Query' – Contains the test query, sentence.
   'Expecrted Results' – Contains the expected entity results for the test record.
   File_name: This is the complete file name (Example: entity_test.txt)
   Dir_name: This indicates the directory under the test_files folder that contains the test files for which we program need to perform the entity recognition.
   Note: One can only have either a file name of the dir name but not both.
   Exp_value: This is a flag that mentions if the test files contain the expected results for not. If the value is 'T' this also means that the test file is a csv file with 'Expected Results' columns.
   Model_name: Provide the pre-trained model name that needs to be used to perform the entity recognition on the test files.

Code Overview:

The MITIE_NER_package.py file contains all the functions needed to process and run the NER training and testing.

The process_MITIE_NER.py file is the trigger file to read the instructions and call the functions in the MITIE_NER_package.py file.

For in-depth details about the functions in the MITIE NER package please refer to the code for comments and directions.

Prerequisites:

1. MITIE library needs to be install on the working environment.
2. MITIE_Master folder need to be added to the working directory.

10. Testing the MITIE trained model:

We have performed the below testing:

1. Test the MITIE models on another set of documents.
2. Test the MITIE model to identify entities in user queries and pass the NER processed queries in the Hunspell spell checker and get predictions for non-NER tokens. (Primary objective is to reduce the False positive rate).

Conclusion…. A few thoughts …

A large collection of Hunspell dictionaries will improve the accuracy of the spell check results.

Using a probabilistic model on historical query correction data to choose the best suggestion amongst the top n Hunspell suggestion will improve the accuracy and relevance quality of the spell check

Implementing a high quality of NER model will reduce the False positives in user queries and greatly improves the precision quality of the spell check model.