Prerequisites:

1. Python 2 (version 2.7.6 or higher)
2. NumPy (version 1.6.1 or higher)
3. SciPy (version 0.9 or higher)
4. Scikit-learn (version 0.18.1)
5. Java (version 1.6 or higher)
6. Preferred OS: Ubuntu 14.04 LTS or other Linux kernel

Manual on how to run ATOL Classify:

The ATOLClassify directory contains the following Python scripts and Java programs:

- baseline.py
- benchmark.py
- getPRFscores.py
- tficf.py
- GenerateWordGrp.java
- GenerateWords.java

We now briefly describe each of the Python scripts and how they can be executed.

→ baseline.py

This program takes as input tokenized text of documents, the
labeling of documents with categories (both train and test data),
a seed list of keywords for each category, and an index file having
the titles for each document. It outputs weights associated with
new keywords for each category (higher weight => keyword is more
important for category). It also computes probabilities of
categories for the test set, and computes accuracy of label
prediction based on those probabilities.

The program takes 7 arguments: (a) a train label file, which has a
list of documents for a particular category (one site per line),
(b) a word_group directory having multiple files, one per document --
each file has the count of words in that document, (c) a seed list
of keywords for each category, (d) an index file having title of
documents and existing document labelings, (e) test label file.
(f) stop word list, (g) baseline labels (for discovery).
#
Usage: python baseline.py -l [train_label_file] \
-d [wordgrp_dir] \
-k [keywords_file] \
-i [index_file] \

```
        -t [test_label_file] \
        -s [stopwords_file] \
        -b [baseline_label_file] \
        -m [mode]
```

We have created a small set of sample documents corresponding to each of the above arguments with the 20 newsgroup dataset (with categories graphics, windows, misc). The following shows an example usage with these toy samples:

Example:

```
python baseline.py \
    -l parameters.sample/train.sample \
    -d wordgrp.sample \
    -k parameters.sample/keywords.sample \
    -i parameters.sample/title.sample \
    -t parameters.sample/test.sample \
    -s parameters.sample/stopwords.txt \
    -b parameters.sample/empty.txt \
    -m "accuracy"
```

We can also replace ‘‘accuracy’’ with ‘‘discovery’’ to discover new keywords i.e. change the mode from accuracy to discovery. The discovery phase needs baseline label file. This is also known as the keyword discovery phase. Please look at the workshop paper at AAAI for more details, regarding the keyword discovery phase.

Shalini Ghosh, Phillip Porras, Vinod Yegneswaran, Ken Nitz, Ariyam Das. "*ATOL: A Framework for Automated Analysis and Categorization of the Darkweb Ecosystem*." In AAAI Workshop on Artificial Intelligence for Cyber Security *(AICS)*, 2017.

Detailed description about format of the files are given below. Please see the toy samples in case of doubt:

```
#
Format of each line of train_label_file or test_label_file:
document id/name,Category
```

#
Format of each line of each wordgrp file in the wordgrp_dir:
Word,Count,NumPages,Ratio
Also each wordgrp file must end with .onion suffix

#
Format of each line in the keywords_file:
Category, <comma-separated list of keywords>

#
Format of each line of index_file:
document id/name,Real Port,Title,Link Count,Language,First Seen,Last Seen,Last Status,Probe Status,Boolean Flag,Keys
Note: Only 1st and 3rd column are of interest.
For e.g. look at parameters.sample/title.sample, where other columns are left empty.

#
Format of stopwords file:
Word

#
Format of baseline label file:
document name/id, Category[<comma-separated list of matching keywords>]

Output: The script prints the accuracy and confusion matrices for tficf+cosine similarity keywords. The script also prints the performance of baseline keyword similarity based methods.

→ benchmark.py

This script uses Bag of Words (BoW) or Term Frequency Inverse Document Frequency (TFIDF) representation and then experiments with standard classifiers.

The script tries out the following supervised classification techniques on documents:

1. Bernoulli Naive Bayesian Classifier
2. Multinomial Naive Bayesian Classifier
3. SVM
4. Linear Classifer with SGD
5. k-Nearest Neighbor classifier
6. C4.5 Decision Tree
7. Random Forest Classifier
8. Ridge Classifier
9. Nearest Centroid classifier

How to construct input feature vectors?
→ We support the following options:

0. Bag of Words / Counting Vector / Term frequency technique
1. TFIDF model (where each term frequency is also weighted with inverse document frequency)
2. Bag of Words / Counting Vector / Term frequency technique [WITH STOP WORDS FILTERING]
3. TFIDF model (where each term frequency is also weighted with inverse document frequency) [WITH STOP WORDS FILTERING]

Usage: python benchmark.py [stopwords file] [Input path to data folder] [input vector construction option] [optional: ngram lower limit] [optional: ngram upper limit]

- The data folder should contain subfolders, each of them are named according to the categories/classes. But there is a caveat to separate train-test split, which is discussed later.
- Each subfolder contains the documents that have been manually classified as the said category.
- [input vector construction option] can be 0,1,2,3 (each representing the corresponding strategies, as discussed above).
- ngram lower and upper limits are optional; but if one of them is provided the other must be provided, otherwise we completely ignore the ngram model. If none is

provided we follow simple bag-of-words model, where position of the words does not matter.

Example: python benchmark.py parameters.sample/stopwords.txt data.sample 0 > result.bow

Output: Prints the accuracy, confusion matrix, precision, recall and weighted F1 score of the different classifiers.

Note: (1) In order to ensure that we use the same train-test split for TFICF based models in our BoW/TFIDF experiments as well, our input data folder contains twice the number of categories as subfolders. In fact each category repeats with two types of prefix ‘‘tr_’’ and ‘‘te_’’ representing the corresponding training and test instances. (2) This is to ensure that the same train and test split could be used in all experiments. (3) If you want random split, simply uncomment *line:157* and comment the block of code from *161:191*.

→ tficf.py

This script creates input feature vectors following TFICF model, as explained below:

* TFICF works as follows:

- 1. For each category, calculate the weights of the keywords (this is actually the term frequency in a category multiplied by the inverse frequency across all the categories)
- 2. Once the previous step is done, treat the document as a Bag of Words (where each word feature also includes the number of times this term is included in the document). So, basically the number of times this word appeared * [its weight from the TFICF model]

This script wants to use the input feature vectors generated from (2) for training some additional supervised ML classifiers.

The classifiers are straightforward used from python scikit package.

* USAGE: Run it exactly with the same arguments as the 'baseline' script.

Example:

```
python tficf.py \
        -l parameters.sample/train.sample \
        -d wordgrp.sample \
        -k parameters.sample/keywords.sample \
        -i parameters.sample/title.sample \
        -t parameters.sample/test.sample \
        -s parameters.sample/stopwords.txt \
        -b parameters.sample/empty.txt \
        -m "accuracy"
```

* Output:

This prints the confusion matrix and accuracy for each classifier.

Note:
If you need to calculate the precision, recall and F1 score for any of the classifiers, you can use this adhoc script 'getPRFscores.py' as follows:

USAGE: python getPRFscores.py

Then enter the confusion matrix (expects 3x3 input by default).
Prints the Precision, Recall and Weighted F1 score.

→ GenerateWordGrp.java

Compile the java program using the following command:
javac GenerateWordGrp.java

This program accepts a text file as an input argument and prints the corresponding word group format file (the wordgrp_file) required for tficf based classifiers.

Example: The below UNIX script generates the wordgrp files for the 20 newsgroup sample dataset:

```
for i in sample/t*/*;
do
        x=$(echo $i | awk -F'/' '{print $NF}');
        java GenerateWordGrp $i > wordgrp.sample/"$x".onion;
done
```

→ GenerateWords.java

Compile the java program using the following command:
javac GenerateWords.java

This program accepts a text file as an input argument and prints the corresponding words required for BoW/TFIDF based classifiers.

Example: The below UNIX script generates the files for the 20 newsgroup sample dataset:

```
for i in sample/*;
do
        mkdir -p data."$i";
done
```

```
for i in sample/t*/*;
do
        java GenerateWords $i > data."$i";
done
```

Manual on how to run ATOL Cluster:

The ATOLCluster directory contains the following Python scripts and Java code:

- full_constrained_kmeans.py
- kmeans++.py
- kobj.py
- mussco.py
- provenance_constrained_kmeans.py
- seed_constrained_kmeans.py
- seeded_kmeans.py
- seedinit_provenance_constrained_kmeans.py
- unsupervised_kmeans.py
- mussco_20news.py
- seeded_kmeans_20news.py
- simul_domain_exp.py
- simul_domain_exp_with_prob.py
- TransitiveClosure.java

We now briefly describe each of the Python scripts and how they can be executed.

→ mussco.py

Usage: python mussco.py \
-s [stopwords file] \
-k [keywords file] \
-l [seeded labels file] \
-i [input data] \
-c [number of clusters] \
-t [test file] \
-d [debug option] \
-n [noise percentage] \
-m [must link onions]

Description:

1. [stopwords file]:
- Absolute/relative path to the stopwords file, if any.

2. [keywords file]:
- Absolute/relative path to the keywords file. This must be given. The file will be formatted as follows: <Category name/Topic name> : <comma separated list of keywords/phrases>. We are using ':' as the delimiter. This file can be empty.

3. [seeded labels file]:
- Absolute/relative path to the seeded labels file. This must be given. The file will be formatted as follows: <onion name without extension> : <comma separated list of labels>. We are using ':' as the delimiter. This file can be empty.

4. [input data]:
- Absolute/relative path to the data folder. This must be given. The data folder should contain subfolders. Each of the subfolders is named according to the categories/topics. Each subfolder contains the documents/onion sites, that have been manually classified as the said category. The onion sites are stored in files which are named on the onion name, without any extension like .onion or .html

5. [number of clusters]:
- Number of clusters is to be provided as input. This must be given.

6. [test file]:
- Absolute/relative path to the test file. This is an optional argument. Follow the same format as seeded labels file.

7. [debug option]:
- Only two possible values -- ON or OFF. This is an optional argument. Default is OFF.

8. [noise percentage]:
- Any number from 0 to 100. This is an optional argument. It only makes sense to

use this with the -t option.

9. [must link onions]:
- Absolute/relative path to the must-link onions file. This must be given. The file will be formatted as follows: <onion1> : <onion2>. The above implies onion1 and onion2 must be in the same cluster ALWAYS. We are using ':' as the delimiter. This file can be empty.

Example: python mussco.py -k parameters.sample/expert_keywords.txt -l parameters.sample/seeds_5-cnt_per_category_3-clusters.txt -i data.sample -c 3 -m parameters.sample/must_link.txt

Note:
(1) If the keywords file, the seeded labels file, and the must link file are all empty, then this reduces to simple unsupervised k-means.
(2) If the keywords file & the must link file are empty, then this reduces to simple seeded k-means (minor tweak required, see main call in the code and the corresponding comments about hyper-parameter tuning).
(3) If the keywords file & seeded label file are empty, then this reduces to constrained k-means (minor tweak required, see main call in the code and the corresponding comments about hyper-parameter tuning).

CAUTION:
Label names are always CASE-SENSITIVE. Make sure they are consistent everywhere: as a directory name, in the keywords file, in the seeds labeled file.

The following scripts are variants of the original python script mussco.py and can be executed in the exact same manner.

- full_constrained_kmeans.py
- kmeans++.py
- kobj.py
- provenance_constrained_kmeans.py
- seed_constrained_kmeans.py
- seeded_kmeans.py
- seedinit_provenance_constrained_kmeans.py
- unsupervised_kmeans.py

→ unsupervised_kmeans.py

The above script performs unsupervised Kmeans clustering (with random initialization). The manual keywords file, seeded data file, must link constraint file should all be empty.

Example:
python unsupervised_kmeans.py -k input-parameters/empty.txt -l input-parameters/empty.txt -i input-data -c 3 -m input-parameters/empty.txt

---

→ kmeans++.py

The above script performs Kmeans++ clustering (using built-in Sklearn function). The manual keywords file, seeded data file, must link constraint file should all be empty.

Example:
python kmeans++.py -k input-parameters/empty.txt -l input-parameters/empty.txt -i input-data -c 3 -m input-parameters/empty.txt

---

→ kobj.py

The above script performs KMeans based clustering (but here initialization done with actual object points). The manual keywords file, seeded data file, must link constraint file should all be empty.

Example:
python kobj.py -k input-parameters/empty.txt -l input-parameters/empty.txt -i input-data -c 3 -m input-parameters/empty.txt

---

→ seeded_kmeans.py

The above script performs **seeded Kmeans** clustering (with initialization from seeded data). The manual keywords file, must link constraint file should all be empty. The seeded data file must be provided.

Ref: Sugato Basu et al. *Semi-supervised Clustering by Seeding*. ICML 2002, pages 27-34.

Example:
python seeded_kmeans.py -k input-parameters/empty.txt -l input-parameters/seeds_5-cnt_per_category_3-clusters.txt -i input-data -c 3 -m input-parameters/empty.txt

---

→ seed_constrained_kmeans.py

The above script performs seeded **Constrained Kmeans** clustering (with initialization from seeded data and additional constraint enforcement that the labels of the seeded data must not change). The manual keywords file, must link constraint file should all be empty. The seeded data file must be provided.

Ref: Sugato Basu et al. *Semi-supervised Clustering by Seeding*. ICML 2002, pages 27-34.

Example:
python seed_constrained_kmeans.py -k input-parameters/empty.txt -l input-parameters/seeds_5-cnt_per_category_3-clusters.txt -i input-data -c 3 -m input-parameters/empty.txt

---

→ provenance_constrained_kmeans.py

The above script performs **COP-Kmeans** (that enforces strict must-link constraints with random initialization of cluster centers). The manual keywords file, seeded data file should all be empty. The must link constraint file must be provided.

Ref: Kiri Wagstaff et al. *Constrained K-means Clustering with Background Knowledge*. ICML 2001, pages 577-584.

Example:
python provenance_constrained_kmeans.py -k input-parameters/empty.txt -l input-parameters/empty.txt -i input-data -c 3 -m input-parameters/must_link.txt

---

→ seedinit_provenance_constrained_kmeans.py

The above script performs a hybrid of the **seeded Kmeans** and **COP-Kmeans** (we enforce strict must-link constraints with initialization of cluster centers based on seeded data). The manual keywords file should be empty. The seeded data file, must link constraint file must be provided.

Ref [1]: Sugato Basu et al. *Semi-supervised Clustering by Seeding*. ICML 2002, pages 27-34.
Ref [2]: Kiri Wagstaff et al. *Constrained K-means Clustering with Background Knowledge*. ICML 2001, pages 577-584.

Example:
python seedinit_provenance_constrained_kmeans.py -k input-parameters/empty.txt -l

input-parameters/seeds_5-cnt_per_category_3-clusters.txt -i input-data -c 3 -m input-parameters/must_link.txt

---

→ full_constrained_kmeans.py

The above script performs a hybrid of the **Constrained Kmeans** and **COP-Kmeans** (we enforce strict must-link constraints with initialization of cluster centers based on seeded data; we also ensure that the labels of the seeded data do not change). The manual keywords file should be empty. The seeded data file, must link constraint file must be provided.

Ref [1]: Sugato Basu et al. *Semi-supervised Clustering by Seeding*. ICML 2002, pages 27-34.
Ref [2]: Kiri Wagstaff et al. *Constrained K-means Clustering with Background Knowledge*. ICML 2001, pages 577-584.

Example:
python full_constrained_kmeans.py -k input-parameters/empty.txt -l input-parameters/seeds_5-cnt_per_category_3-clusters.txt -i input-data -c 3 -m input-parameters/must_link.txt

---

The following two scripts are sample/example scripts to show how the aforementioned scripts can be modified to read datasets, where sometimes erros are thrown due to the strict parsing nature enforced in the code. The two scripts below relaxes that:

- mussco_20news.py
- seeded_kmeans_20news.py

The original mussco.py loads data using utf-8 encoding codec and raises errors, if any during encoding.

Example shown below:

```
dataset = load_files(input_dir, shuffle=False, encoding='utf-8',
decode_error='strict')
```

But mussco_20news.py ignores those errors and continues with the loading.

Example shown below:

```
dataset = load_files(input_dir, shuffle=False, encoding='utf-8',
decode_error='ignore')
```

The following two scripts are to be used when domain expertise is absent. For example, if we do not have keywords provided manually by experts, we can run these scripts to generate them (this is an attempt to simulate domain experience).

- simul_domain_exp.py
- simul_domain_exp_with_prob.py

The keyword extraction for topics (or clusters) are done using
- Non-negative Matrix Factorization and
- Latent Dirichlet Allocation

Input argument:
- Directory path, containing corpus of documents

Output:
- List of topics, each topic as set of keywords (overlapping allowed);

Input parameters (to be editted in the script):
- n_topics (number of topics) and n_top_words (number of top words for each topic).

→ simul_domain_exp.py

The above script performs prints the top keywords for each topic (probability/weights of keywords are not shown).

Example:
python simul_domain_exp.py input-data

→ simul_domain_exp_with_prob.py

The above script performs prints the top keywords for each topic (probability/weights of keywords are shown here). Use this script with very large n_top_words to determine the cut-off threshold.

Example:
python simul_domain_exp_with_prob.py input-data

→ TransitiveClosure.java

Compile the java program using the following command:
javac TransitiveClosure.java

This program accepts a must-link constraint file (text file) as an input argument and prints the transitive closure of the must-link constraints. The output format is the same as the input text format for the must-link consraint file: <point1>:<point2> means point1 and point2 must be in the same cluster.

Example: java TransitiveClosure must_link_tmp.txt > must_link.txt

---