UnPlag

Course project for **CS 251: Software Systems Lab**

Table of Contents

1. Getting Started
2. Core Logic
3. UnPlag Backend API Documentation
4. Angular Frontend Routes Documentation
5. Command Line Interface

Getting Started

Backend

```
cd to UnPlag/
python3 -m venv UnPlag
source UnPlag/bin/activate
pip install -r requirements.txt
cd unplag
python manage.py makemigrations account plagsample organization
python manage.py migrate
python manage.py runserver
```

Frontend

```
cd to UnPlag/
cd frontend
npm install
ng serve
```

CLI

```
cd to UnPlag/
cd cli
npm install
npm link
```

Core Logic

Experimentations/Approaches tried:

We document and discuss the various models, implementations and approaches we tried throughout the course of this project. We also discuss the problems faced, results and drawbacks.

Character-level LSTM based approach:

For: `c/c++` We tried implementing this paper. We tried to train an character-level LSTM on the sequence prediction task on the entire Linux kernel source code (`.c, .cpp` and `.h` files).
If this was successful, we would've fine-tuned this model on less complicated C files, and then use the output of the last layer as our learned features, which would eventually be passed into a SVM classifier for ternary classification, each class signifying a different degree of plagiarism.

Problems faced while implementation:

- **Lack of computational power:** Google Colab was being used for training the LSTM. After concatenation of all relevant files from the Linux source code, the resultant text file was extremely large (440 MB for only `.c` files). We tried training a 3-layered LSTM each layer having 512 units but Colab's RAM kept exploding as it couldn't handle the large number of 100-character-long sequences for training the model.
- **Unavailability of a labelled dataset:** Even if we somehow managed to train the LSTM model using other ways to feed the sequences, training of the SVM classifier would be infeasible due to the lack of a properly labelled dataset. The authors of the paper have made their own dataset for this purpose using the submissions from an actual college course, which have been properly annotated by their teaching assistants and cross-verified with MOSS scores. Unfortunately, they haven't released this dataset, and we couldn't find any other such labelled dataset containing a good number of files. Creating our own dataset (without the access to MOSS or TAs), and manually annotating each file would take ages (and also may cause copyright issues). Due to the above reasons and a shortage of time, we decided to not go on with this approach. However, this approach seems very promising, and the authors claim that this model performs much better than MOSS on their labelled dataset.

Using unsupervised learning on source code metrics

For: `c/c++` We tried implementing this paper.
In this paper, the authors try using 55 source code metrics extracted using Milepost GCC as features, and then use a clustering algorithm based on the Euclidean distance between the feature vectors to identify similar groups of files.

Problems faced while implementation:

- **Lack of documentation:**: We couldn't find any proper documentation of how to use Milepost GCC to extract the static program features. Even the installation was creating problems.
- **Integration with the backend:** This was a bigger issue, Milepost GCC seemed to be command line tool working mainly using Bash and C files, none of us knew how to make this work with the Django REST backend. We were also uncertain about how well this would actually work, since again, the authors hadn't released the dataset which they were using to report the results obtained in the paper. Furthermore, this approach was being tested along with the AST based approach (which we are finally using). Since we were quite satisfied with the results obtained by the AST based approach, we didn't want to experiment more, and instead improve on the AST based approach. Due to the above reasons and a lack of time, we decided to not go on with this approach.

For: `python3`

Taking inspiration from the previous approach for `c/c++` files, we decided try the same for `python3` files. We used `radon` to extract a total of 21 source code metrics from python files to be used as 21-dimensional features. The metrics are:

- Raw metrics:
  - **loc**: The number of lines of code (total)
  - **lloc**: The number of logical lines of code
  - **sloc**: The number of source lines of code (not necessarily corresponding to the LLOC)

  - **comments**: The number of Python comment lines
  - **multi**: The number of lines which represent multi-line strings
  - **single_comments**: The number of lines which are just comments with no code
  - **blank**: The number of blank lines (or whitespace-only ones)
- Halstead Metrics:
  - **h1**: the number of distinct operators
  - **h2**: the number of distinct operands
  - **N1**: the total number of operators
  - **N2**: the total number of operands
  - **h**: the vocabulary, i.e. h1 + h2
  - **N**: the length, i.e. N1 + N2
  - **calculated_length**: h1 _ log2(h1) + h2 _ log2(h2)
  - **volume**: V = N * log2(h)
  - **difficulty**: D = h1 / 2 * N2 / h2
  - **effort**: E = D * V
  - **time**: T = E / 18 seconds
  - **bugs**: B = V / 3000 - an estimate of the errors in the implementation
- Cyclomatic Complexity
- Maintainability Index

(These are all possible metrics which Radon can compute)

For each file in the collection provided, we compute a 21-dimensional vector. We center and standardize this feature-wise. For similarity computation, we used cosine similarity. For testing, we cloned this GitHub repository containing 576 files, flattened it, and ran our program on all these files. Ideally, we would want the similarity to be low for all pairs, since each file was a different algorithm. On running, and keeping the threshold for cosine similarity as `0.998` , we obtain a high similarity between:

```
base32.py base85.py
find_max_recursion.py find_min_recursion.py
gaussian_naive_bayes.py random_forest_classifier.py
gaussian_naive_bayes.py random_forest_regressor.py
randomized_heap.py skew_heap.py
random_forest_classifier.py random_forest_regressor.py
remove_duplicate.py test_prime_check.py
sol1.py sol5.py
```

We see that most of these pairs (apart from maybe `random_forest_classifier.py and random_forest_regressor.py`) are indeed similar. The program takes only `7s` for 567 files (On `WSL-2`).Thus the efficiency is better when compared to our detector for `c++` files.

The above dataset didn't contain any actual cases of plagiarism, so we created a 16-file dataset containing python code taken from various free sources like GeeksForGeeks, Javatpoint, GitHub, etc. The dataset description is as follows: - `00.py` : Python program to create a Circular Linked List of n nodes and display it in reverse order - `01.py` : Same as above but with variables changed, extra useless comments added - `02.py` : Order of functions, classes, declarations has been changed - `03.py` : Small changes in logic - `while` changed to `for` , `if-else` blocks reversed, completely new `main` function, extra useless functions added - `04.py` : Blocks of useless/repeated code inserted throughout the file - `05.py` : Different approach for the same problem - `06.py` : Another different approach for the same problem - `07.py` : Somewhat related to `00.py` , contains an implementation of circular linked lists - `08.py` : Another implementation of circular linked lists, related to `07.py` -

`09.py` : Completely unrelated, a program for topological sorting - `10.py` : Unrelated, an implementation of heapsort - `11.py` : Unrelated, Dijkstra's algorithm - `12.py` : Unrelated, Radix sort - `13.py` : Recursive Fibonacci - `14.py` : Dynamic programming Fibonacci - `15.py` : Fibonacci with memoization

Keeping the threshold as 0.7, we obtain high similarity between the following files:

```
00.py 01.py
00.py 02.py
00.py 09.py
01.py 02.py
06.py 09.py
10.py 12.py
13.py 14.py
13.py 15.py
```

Drawbacks:

- The threshold used in the previous case was 0.998, this is extremely high, and shows how sensitive the program is to thresholds. Even reducing the threshold to 0.995 results in many new pairs of files popping up as plagiarized, many are actually similar, while many are also false positives. This shows that the distribution of the similarities is heavily skewed towards 1. Thus, it can get quite complicated for the end user to select an appropriate threshold.
- The tests on the second, more carefully created dataset provides some valuable insights into the detector's behavior. We do expect a high similarity between `00.py, 01.py, 02.py` and between `13.py, 14.py, 15.py` , but the remaining pairs are false positives. More importantly, little similarity is detected between `00.py and 03.py,04.py` whereas ideally, this should've been reported.

These drawbacks are quite significant, thus we decided not to use/integrate this detector with the backend.

TF-IDF on Abstract Syntax Trees

(This approach is currently being used) For: `c++` We first parse the given file using clang and create the AST. We then traverse the tree in pre-order and use the list of nodes (as `clang.cindex.Cursor` objects) for further processing. We then process each node according to it's "kind" (`clang.cindex.CursorKind`), we have pre-defined rules for each kind. The preprocessed node is added as a token.

Assumptions made while pre-processing and tokenizing:

- Comments are ignored
- Compound assignment operators (eg: `+=`) are treated as binary operators
- "Free" functions (functions not defined inside classes) and member functions are treated the same.
- All numeric data types are treated the same, else, changing all `int` s to `long long int` s would decrease the similarity considerably.
- Range based loops (eg: `for(auto i: ...)`) are treated the same as vanilla for loops
- All identifiers are ignored, and are stored as the token " `var` ". For example, declaring a variable, say `string a` is stored as the token `string_var` , and whenever the variable `a` is used later, eg: `cout << a` is stored as `string_used`
- Explicit type casting and functional type casting are treated the same, eg: `a = (int)b` and `a = int(b)` are assumed to be equivalent

We then create a vocabulary using unigrams (single tokens) and bigrams (two consecutive tokens) and apply the TF-IDF weighting scheme using sub-linear TF. Cosine similarity is used for computing the similarity between the resultant weight vectors.

Testing dataset:

We create our own 16-file dataset containing code taken from various free online sources. Description of the data:

- `00.cpp` : Directed graph implementation
- `01.cpp` : Variables and types changed
- `02.cpp` : Order of blocks, declaration, etc changed
- `03.cpp` : Addition of useless code between the original code
- `04.cpp` : Moderately plagiarized (Different `main` , `while` to `for` , etc)
- `05.cpp` : Heavily plagiarized (Taken from a different online source, which apparently plagiarized from the first source)
- `06.cpp` : A different implementation
- `07.cpp` : Another very different implementation
- `08.cpp` : Depth-first search
- `09.cpp` : Breadth-first search
- `10.cpp` : Quicksort
- `11.cpp` : Mergesort
- `12.cpp` : Recursive Fibonacci
- `13.cpp` : Matrix-based Fibonacci
- `14.cpp` : Fibonacci using memoization
- `15.cpp` : Fibonacci using dynamic programming

Results:

This takes about 15s to execute. We obtain the following results (indexed according to filenames): | | | | | | | | | | | | | | | | | |--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------| | 1 | 1 | 0.891 | 0.7905 | 0.689 | 0.9652 | 0.4048 | 0.1667 | 0.3583 | 0.3012 | 0.3858 | 0.3732 | 0.1842 | 0.2615 | 0.1583 | 0.1676 | | 1 | 1 | 0.891 | 0.7905 | 0.689 | 0.9652 | 0.4048 | 0.1667 | 0.3583 | 0.3012 | 0.3858 | 0.3732 | 0.1842 | 0.2615 | 0.1583 | 0.1676 | | 0.891 | 0.891 | 1 | 0.7244 | 0.6203 | 0.8608 | 0.378 | 0.1647 | 0.3461 | 0.2991 | 0.3708 | 0.3504 | 0.1788 | 0.2481 | 0.1438 | 0.1618 | | 0.7905 | 0.7905 | 0.7244 | 1 | 0.5662 | 0.7629 | 0.3162 | 0.1693 | 0.318 | 0.2642 | 0.5507 | 0.3882 | 0.5241 | 0.3 | 0.31 | 0.2538 | | 0.689 | 0.689 | 0.6203 | 0.5662 | 1 | 0.6736 | 0.2065 | 0.1729 | 0.3184 | 0.2485 | 0.3026 | 0.3227 | 0.1586 | 0.1885 | 0.1243 | 0.1304 | | 0.9652 | 0.9652 | 0.8608 | 0.7629 | 0.6736 | 1 | 0.4047 | 0.1655 | 0.3538 | 0.2988 | 0.3745 | 0.3735 | 0.1716 | 0.2659 | 0.161 | 0.1671 | | 0.4048 | 0.4048 | 0.378 | 0.3162 | 0.2065 | 0.4047 | 1 | 0.2946 | 0.2201 | 0.217 | 0.1854 | 0.2162 | 0.1458 | 0.1956 | 0.0736 | 0.1493 | | 0.1667 | 0.1667 | 0.1647 | 0.1693 | 0.1729 | 0.1655 | 0.2946 | 1 | 0.1618 | 0.1778 | 0.0974 | 0.0711 | 0.0899 | 0.0653 | 0.0719 | 0.106 | | 0.3583 | 0.3583 | 0.3461 | 0.318 | 0.3184 | 0.3538 | 0.2201 | 0.1618 | 1 | 0.7749 | 0.2673 | 0.2721 | 0.1705 | 0.1526 | 0.097 | 0.133 | | 0.3012 | 0.3012 | 0.2991 | 0.2642 | 0.2485 | 0.2988 | 0.217 | 0.1778 | 0.7749 | 1 | 0.2147 | 0.2074 | 0.1381 | 0.1733 | 0.0982 | 0.1404 | | 0.3858 | 0.3858 | 0.3708 | 0.5507 | 0.3026 | 0.3745 | 0.1854 | 0.0974 | 0.2673 | 0.2147 | 1 | 0.5356 | 0.279 | 0.2999 | 0.3435 | 0.296 | | 0.3732 | 0.3732 | 0.3504 | 0.3882 | 0.3227 | 0.3735 | 0.2162 | 0.0711 | 0.2721 | 0.2074 | 0.5356 | 1 | 0.3 | 0.2591 | 0.2915 | 0.4656 | | 0.1842 | 0.1842 | 0.1788 | 0.5241 | 0.1586 | 0.1716 | 0.1458 | 0.0899 | 0.1705 | 0.1381 | 0.279 | 0.3 | 1 | 0.2374 | 0.3875 | 0.3199 | | 0.2615 | 0.2615 | 0.2481 | 0.3 | 0.1885 | 0.2659 | 0.1956 | 0.0653 | 0.1526 | 0.1733 | 0.2999 | 0.2591 | 0.2374 | 1 | 0.2772 | 0.2623 | | 0.1583 | 0.1583 | 0.1438 | 0.31 | 0.1243 | 0.161 | 0.0736 | 0.0719 | 0.097 | 0.0982 | 0.3435 | 0.2915 | 0.3875 | 0.2772 | 1 | 0.3128 | | 0.1676 | 0.1676 | 0.1618 | 0.2538 | 0.1304 | 0.1671 | 0.1493 | 0.106 | 0.133 | 0.1404 | 0.296 | 0.4656 | 0.3199 | 0.2623 | 0.3128 | 1 |

These results are precisely what we expect. The detector is not fooled by variable name changes, variable type changes, reordering, dead code injections, and detects moderate/heavy plagiarism accurately. It also nicely segregates different approaches to the same problem effectively and doesn't report them as plagiarized. Furthermore, the similarity values are nicely distributed between 0 and 1, thus easing threshold selection.

TF-IDF's superiority compared to Cosine Similarity and Jaccard Similarity

We also tried the following approaches/metrics:

- Unigrams with cosine similarity as the metric between frequency vectors
- Unigrams with Jaccard similarity as the metric between frequency vectors

The results are as follows:

- Cosine similarity:

1	1	0.9978	0.9816	0.8508	0.9949	0.8206	0.3538	0.849
1	1	0.9978	0.9816	0.8508	0.9949	0.8206	0.3538	0.849
0.9978	0.9978	1	0.9759	0.8286	0.9918	0.8286	0.3639	0.8436
0.9816	0.9816	0.9759	1	0.8797	0.9754	0.7843	0.3792	0.8445
0.8508	0.8508	0.8286	0.8797	1	0.8461	0.5085	0.3903	0.7492
0.9949	0.9949	0.9918	0.9754	0.8461	1	0.827	0.3552	0.8371
0.8206	0.8206	0.8286	0.7843	0.5085	0.827	1	0.2931	0.7619
0.3538	0.3538	0.3639	0.3792	0.3903	0.3552	0.2931	1	0.3537
0.849	0.849	0.8436	0.8445	0.7492	0.8371	0.7619	0.3537	1
0.8334	0.8334	0.8313	0.8196	0.7161	0.8179	0.7495	0.3553	0.9767
0.8734	0.8734	0.8537	0.9075	0.8444	0.8664	0.704	0.2775	0.861
0.8149	0.8149	0.7828	0.8434	0.8519	0.8171	0.6216	0.1927	0.7858
0.737	0.737	0.7194	0.8159	0.6728	0.7248	0.6592	0.2485	0.6944
0.7146	0.7146	0.7267	0.663	0.3941	0.7112	0.7001	0.1442	0.6653
0.6911	0.6911	0.6774	0.7414	0.5987	0.6918	0.5762	0.1968	0.6302
0.7877	0.7877	0.7759	0.8062	0.6851	0.7866	0.6753	0.2418	0.7497

As it is clearly seen, the distribution is not uniform between 0 and 1, thus selection of a proper threshold becomes quite difficult. Also, this metric cannot segregate different approaches properly, as files `00.cpp` and `06.cpp` report a similarity of 0.82, even the last 2 different implementations of Fibonacci numbers is incorrectly reported as similar. Many more false positives are also clearly visible.

- Jaccard similarity:

1	1	0.9684	0.7085	0.6319	0.8742	0.3195	0.1525	0.4149
1	1	0.9684	0.7085	0.6319	0.8742	0.3195	0.1525	0.4149

0.9684	0.9684	1	0.6861	0.6312	0.8903	0.3293	0.157	0.4262
0.7085	0.7085	0.6861	1	0.4646	0.6205	0.2308	0.1255	0.351
0.6319	0.6319	0.6312	0.4646	1	0.6316	0.227	0.1756	0.4211
0.8742	0.8742	0.8903	0.6205	0.6316	1	0.3667	0.1698	0.4335
0.3195	0.3195	0.3293	0.2308	0.227	0.3667	1	0.1935	0.3622
0.1525	0.1525	0.157	0.1255	0.1756	0.1698	0.1935	1	0.2031
0.4149	0.4149	0.4262	0.351	0.4211	0.4335	0.3622	0.2031	1
0.3957	0.3957	0.4066	0.3198	0.3882	0.4211	0.3659	0.2213	0.7881
0.485	0.485	0.4747	0.4836	0.436	0.484	0.275	0.142	0.4615
0.3992	0.3992	0.3852	0.4295	0.3122	0.3831	0.2018	0.0779	0.3348
0.1686	0.1686	0.1737	0.1928	0.2276	0.1883	0.3012	0.1266	0.2276
0.2715	0.2715	0.2718	0.2832	0.1722	0.2662	0.194	0.0795	0.2167
0.2804	0.2804	0.2811	0.2582	0.28	0.3099	0.2417	0.1111	0.3061
0.2428	0.2428	0.25	0.2017	0.2791	0.271	0.3261	0.1444	0.3306

Jaccard reports a similarity of 0.4536 between 2 different approaches for Fibonacci numbers, this value is quite moderate in terms of Jaccard values. It even reports a very high value of 0.7881 between DFS and BFS. Injection of dead code also decreased the similarity drastically to 0.7085. This shows that TF-IDF is better at detecting different approaches to the same problem, and is less sensitive to "tricks" like dead code injection.

TF-IDF on preprocessed textual data:

Our model for detecting similarity in textual data is similar to the TF-IDF based approach for `c++` files. Here, the major difference is the preprocessing. We apply the following steps for preprocessing a file:

1. Convert to lowercase
2. Remove all punctuation
3. Remove non-ascii characters
4. Tokenize into words
5. Remove (English) stopwords
6. Use the Porter Stemmer for "removing the commoner morphological and inflexional endings from words in English." Example:

   ```
   connect
   connected
   connecting
   connection
   connections
   ```

   are all stemmed down to `connect`

Similar to the latter half of `c++` approach, we create a vocabulary using unigrams (single tokens) and bigrams (two consecutive tokens) and apply the TF-IDF weighting scheme using sub-linear TF. Cosine similarity is used for computing the similarity between the resultant weight vectors.

Documentation for the code

Note: For the core logic part, we have not used any explicit functional/OOP logic. The files are simple well-commented python scripts meant to be used directly by using certain commands/options. They have been integrated into the Django REST backend in a similar way.

Dependencies

We require the following libraries for proper execution. Run `pip install -r requirements.txt`, you may need to perform some additional steps for installing `clang`

- `os, glob`: For reading the data, applying the given RegEx
- `argparse`: For processing the command line arguments
- `numpy`: General scientific computation
- `tqdm`: For the command line based progress bar
- `unidecode`: For translating all unicode objects into ASCII
- `nltk`: For tokenizing the words, the Porter Stemmer and english stop-words
- `matplotlib, seaborn`: Data visualization
- `sklearn`:
  - `linear_kernel`: For computing dot products between large sparse matrices.
  - `TfIdfVectorizer`: For building the n-gram based vocabulary, computing the TF-IDF weights
- `clang`: For parsing `c++` files, creating and traversing the AST, and tokenizing it. *(Note: `clang` as used by us works only on `linux` based machines. You also may need to install it using `sudo apt-get`, and create symbolic links)*

Usage (for Textual files)

- **Main file:** `models/txt/tfidf.py`
- **Direct usage:** `python3 tfidf.py -d <dataset> -r "<RegEx>" -out <output>`
  - `<dataset>`: This must contain the path to directory containing all the files which you want to check. The files are expected to be in English.
  - `<regEx>`: This must contain the regular expression for testing some subset of the files inside `<dataset>`. For example, if you have Java, C++ and text files in your dataset, and you want to check for plagiarism only on the text files, then you `<regEx>` will be `*.txt`.
  - `<output>`: This contains a string which determines where the output `.csv` file will be saved. For example, if `<output> = my_file.csv` then the output csv file is saved at `txt/results/tfidf_my_file.csv`
- **Output:** Running the code generates:
  - A `.csv` file at `txt/results/tfidf_<output>` This contains the pairwise similarities between the pairs of files. The axis is indexed as `0,...,n-1` where `n` is the number of files, the index to file mapping can be found via running `ls -U` in the `<dataset>`.
  - A heatmap corresponding to the `.csv` file, at `txt/plots/tfidf_heatmap.png`, same indexing is followed as in the csv file.
  - A progress bar will also been shown on the terminal, this indicates the progress made in reading, processing and tokenizing the files.
  - *Note: Analyzing this output for a large number of files may be cumbersome, so it is advised to use the front-end interface which contains much nicer interactive plots.*

Usage (for C++ files)

- **Main file:** `models/cpp/tfidf.py`

- **Direct usage:** `python3 tfidf.py -d <dataset> -r "<RegEx>" -out <output>`
  - `<dataset>` : This must contain the path to directory containing all the files which you want to check. The files are expected to be `C++` files.
  - `<regEx>` : This must contain the regular expression for testing some subset of the files inside `<dataset>`. For example, if you have Java, C++ and text files in your dataset, and you want to check for plagiarism only on the `C++` files, then you `<regEx>` will be `*.cpp`.
  - `<output>` : This contains a string which determines where the output `.csv` file will be saved. For example, if `<output> = my_file.csv` then the output csv file is saved at `cpp/results/tfidf_my_file.csv`
- **Output:** Running the code generates:
  - A `.csv` file at `cpp/results/tfidf_<output>` This contains the pairwise similarities between the pairs of files. The axis is indexed as `0,...,  n-1` where `n` is the number of files, the index to file mapping can be found via running `ls -U` in the `<dataset>`.
  - A heatmap corresponding to the `.csv` file, at `cpp/plots/tfidf_heatmap.png`, same indexing is followed as in the csv file.
  - A progress bar will also been shown on the terminal, this indicates the progress made in reading, processing and tokenizing the files.
  - *Note: Analyzing this output for a large number of files may be cumbersome, so it is advised to use the front-end interface which contains much nicer interactive plots.*

UnPlag Backend API Documentation

API Endpoints Implemented (Links given to the detailed documentation)

Token API Endpoints :

1. '/api/token/'
2. '/api/token/refresh/'

Account API Endpoints :

3. '/api/account/signup/'
4. '/api/account/profile/'
5. '/api/account/update/'
6. '/api/account/upassword/'
7. '/api/account/pastchecks/'

Plagsample API Endpoints :

8. '/api/plagsample/upload/'
9. '/api/plagsample/download/int:id/'
10. '/api/plagsample/info/int:id/'

Organization API Endpoints :

11. '/api/organization/makeorg/'
12. '/api/organization/get/int:id/'
13. '/api/organization/update/int:id/'
14. '/api/organization/joinorg/'

Detailed API Documentation

Token

`ENDPOINT : '/api/token/' | REQUEST TYPE : POST`

Returns an 'access' and a 'refresh' **JWT token** for a given valid 'username' and 'password'

```
Format :

@[in body] username, password
@[JSON response] username, userid, access, refresh
```

Token Refresh

`ENDPOINT : '/api/token/refresh/' | REQUEST TYPE : POST`

Returns an 'access' token for a given valid 'refresh' token

```
Format:

@[in body] refresh
@[JSON response] access
```

User Signup

`ENDPOINT : 'api/account/signup/' | REQUEST TYPE : POST`

Returns a 'username', 'userid' and the 'access' and 'refresh' JWT tokens, for a given valid 'username', 'password', 'password2'

```
Format:

@[in body] username, password, password2
@[JSON response] response(string), username, userid, access, refresh
```

Profile Details

`ENDPOINT : 'api/account/profile/' | REQUEST TYPE : GET (Authenticated Endpoint)`

Returns profile details of the current authenticated user

```
Format:

@[in header] “Authorization: Bearer <access>”
@[JSON response] id(profile id), user(user id), username, nick, orgs: [{org_id,
org_name},...]
```

Profile Update

`ENDPOINT : 'api/account/update/' | REQUEST TYPE : PUT (Authenticated Endpoint)`

Updates the profile with the given input data

```
Format:

@[in header] “Authorization: Bearer <access>”
@[in body] nick(optional and it's the only field as of now)
@[JSON response] id(profile id), user(user id), username, nick
```

Password Update

`ENDPOINT : 'api/account/upassword/' | REQUEST TYPE : PUT (Authenticated Endpoint)`

Updates the user password

```
Format:

@[in header] “Authorization: Bearer <access>”
@[in body] old_password, new_password (required fields)
@[JSON response] status : ‘success’, message : ‘Password updated successfully’
```

Get Past PlagChecks

`ENDPOINT : 'api/account/pastchecks/' | REQUEST TYPE : GET (Authenticated Endpoint)`

Returns a list of past plagiarism check IDs by the user along with the uploaded filename

```
Format:

@[in header] “Authorization: Bearer <access>”
@[in body]
@[JSON response] // Sorted by org_id and then date_posted.
{
   "pastchecks": [
       {
           "filename": "Outlab5-Resources.tar_e3Ce4OJ.gz",
           "file_type": "txt"
           "id": 2,
           "name": "Outlab5-Resources",
           "timestamp": "2020-11-26 20:15:30",
           "org_id": "1",
           "org_name": "scriptographers",
       },
       ...,
       ...
   ]
}
```

Upload Files

`ENDPOINT : 'api/plagsample/upload/' | REQUEST TYPE : POST (Authenticated Endpoint)`

Returns a plagiarism check id for the uploaded compressed file Supplied org_id must be valid and the user must be in it

This method processes the uploaded compressed file on a separate thread, so as to keep the backend open to further uploads.

```
Format:

@[in header] “Authorization: Bearer <access>”
@[in body] name, org_id, file_type(must, available choices : [“txt”, “cpp”]),
plagzip (Filefields) (As of now zip, tar.gz, rar are allowed)
@[JSON response] id(plagsample id), name, file_type, plagzip(name of the files),
user(user id), date_posted, outfile (name of output csv)
```

Download CSV

`ENDPOINT : 'api/plagsample/download/<id>' | REQUEST TYPE : GET (Authenticated Endpoint)`

Returns the processed CSV file as a JSON file attachment response blob(If the authentication details match correctly: User needs to be a part of the organization to which the uploaded sample belongs)

```
Format:

@[in header] “Authorization: Bearer <access>”
@[out]  CSV is returned as a file attachment in the body(as a file Blob).
Name of the file can be found under the "Content-Disposition" header.
@[out in case of error] JSON form of error is returned along with correct HTTP error
code.
Throws 415_UNSUPPORTED_MEDIA HTTP Error if no files of give file_type is
found after extracting the compressed ball.
```

Plagsample Info

`ENDPOINT : 'api/plagsample/info/<int:id>/' | REQUEST TYPE : GET (Authenticated Endpoint)`

Returns details of a particular plag check Supplied id must correspond to a valid plagsample and the user must be in the organization to which it belongs.

```
Format:

@[in header] “Authorization: Bearer <access>”
@[in body]
@[JSON response] id, name , filename, file_type, timestamp, org_id, org_name,
uploader, uploader_id, file_count
```

Create New Organization

`ENDPOINT : 'api/organization/makeorg/' | REQUEST TYPE : POST (Authenticated Endpoint)`

Signs up a new organization with the currently logged in user as its first and only member.

```
Format:

@[in header] “Authorization: Bearer <access>”
@[in body] name(required), title(optional description)
@[JSON response] id(organization id), creator(name of creator), title, date_created,
unique_code
```

Organization Info

`ENDPOINT : 'api/organization/get/<int:id>/' | REQUEST TYPE : GET (Authenticated Endpoint)`

Returns details of the inquired organization. Inquiring user must be a member of the organization.

```
Format:

@[in header] “Authorization: Bearer <access>”
```

```
@[in body]
@[JSON response] id(org id), name, creator, title, unique_code, date_created,
members : [{“id” : 1, “username” : “ardy”}, {...}, {...}] (sorted according to
user_id),
pastchecks : [{filename, id, file_type, timestamp}, ...]
```

Update Organization

`ENDPOINT : 'api/organization/update/<int:id>/' | REQUEST TYPE : PUT (Authenticated Endpoint)`

A user belonging to the organizaation can update the title.

```
Format:

@[in header] “Authorization: Bearer <access>”
@[in body] title
@[JSON response] id(org id), name, title, creator, date_created
```

Join Organization

`ENDPOINT : 'api/organization/joinorg/' | REQUEST TYPE : POST (Authenticated Endpoint)`

Given a unique_code adds the user to the org(unless its a personal organization)

```
Format:

@[in header] “Authorization: Bearer <access>”
@[in body] unique_code
@[JSON response] id(org id), creator, name, title, date_created, unique_code,
members : [{“id” : 1, “username” : “ardy”}, {...}, {...}] (sorted according to
user_id)
```

Angular Frontend Routes Documentation

Routes Implemented (Links given to the detailed documentation)

User Account Routes :

1. '/register'
2. '/login'

Dashboard Routes :

3. '/dashboard'

Profile Routes :

4. '/profile/changepwd'
5. '/profile/view'
6. '/profile/edit'

Organization Routes :

7. '/org/create'
8. '/org/join'
9. '/org/view/:id'

10. '/org/edit/:id'

Plagsample Routes :

11. '/upload'
12. '/report/:id'

Detailed API Documentation

Token

Register

- Create a new user
- Accessible only if the user is not logged in
- Redirects to dashboard if successful

Login

- Login the user
- Accessible only if the user is not logged in
- Redirects to dashboard if successful

Dashboard

- Landing page after login
- Displays list of organizations and uploads belonging to each sorted by latest.
- Accessible only if the user is logged in and if the token has not expired

Change Password

- Change(update) the password
- Accessible only if the user is logged in and if the token has not expired
- Redirects to dashboard if successful

View Profile

- Contains the profile details and list of organizations
- Accessible only if the user is logged in and if the token has not expired

Edit Profile

- Update the profile
- Accessible only if the user is logged in and if the token has not expired
- Redirects to the profile page if successful.

Create an Organization

- Create a new organization
- Accessible only if the user is logged in and if the token has not expired
- Redirects to the organization's page if successful.

Join an Organization

- Join an existing organization
- Accessible only if the user is logged in and if the token has not expired
- Redirects to the organization's page if successful.

View Organization

- Contains the organization details and list of members and samples uploaded
- Accessible only if the user is logged in and is part of the organization and if the token has not expired

Edit Organization

- Update the organization details
- Accessible only if the user is logged in and is part of the organization and if the token has not expired
- Redirects to the organization's page if successful.

Upload Sample

- Upload sample to the organization (only zip, tar, gz, rar are allowed, content type - _.txt or _.cpp)
- Accessible only if the user is logged in and if the token has not expired
- Redirects to the report if the upload is successful.

Display Report

- Users can view the result here.
- Accessible only if the user is logged in and is part of the organization and if the token has not expired
- User can download csv file.

Command Line Interface

Usage

```
unplag-cli <command>

Commands:
  unplag-cli download [save_loc]       Download csv
  unplag-cli upload [file_loc] [name]  Upload compressed folder

Options:
  --version  Show version number                      [boolean]
  --help     Show help                                [boolean]
```
