---

## Text Preprocessor

---

To create a good classifier using the Bag Of Words model, we need to create a *vocabulary* to feed into it. In order to train our classifier, we use a *corpus*, a set of thousands or tens of thousands of sentences. Each sentence is labelled with one of the classes we are concerned with.

In the case of sentiment analysis for "weather", this might be something like "nice weather eh" or "snow is coming".

Of course, the same machine-learned classifier should be able to be trained on data for a different sentiment. If we wanted to do sentiment analysis for "I need help", this might be something like "Need Help" or (negative example) "Do Not Need Help". Once we have our training set, we can use this to prepare our classifier.

However, using the raw text alone is problematic. It can lead to the same token being considered multiple times because of subtle differences in letter case or punctuation. For example, in a raw string of tokens, all of the following words would be considered separately in the analysis:

```
Help   help helping  Help!  Help!  HELP  helped  help.
```

Even though all of the above words indicate a similar sentiment, they have not been normalized into the single string "help".

A simple trick to solve this problem is to *transform the texts before they are used to train the classifier.* This step is also known as *preprocessing* because we are modifying (or processing) the data before using it.

The goal of this preprocessing step is to reduce the number of words in our vocabulary. This step helps to make our classifier more effective by consolidating multiple similar words into a single representative token. Some preprocessing techniques that are used to do this include:

- **Stemming**, which finds the common root of a group of words. For example, "helped", "helping", and "helper" would all be reduced to the common root "help" by removing the suffixes of words.

- **Lemmatization**, which is a process that groups together different inflections of a word into a single *lemma*. It is different from stemming because it looks at the context of a word rather than reducing using form alone.

- **Tagging different parts of speech** (labelling prepositions, adverbs, adjectives, nouns, and so on).

- **Removing numbers, punctuation, and special characters.**

- **Removing stopwords**, which are words that are ignored because they do not provide any classifying information. For example, in English, the words "the", "it", and "for" are common stopwords.

- **Converting all tokens to lowercase** so that they can be considered uniformly.

In this exercise, you will perform some of the above steps in order to perform preprocessing on an input text. To do this, you will first read in a text to process. Then, you will split the text into tokens (this will be a simple split by whitespace) and then perform actions to **normalize** each token before printing out the new word. You will also use an *optional command line argument* called `mode` to determine which preprocessing steps to complete.

### Your Task #1: Full Preprocessing

Write a program called `preprocess.py` that takes a space-separated line of words as input and performs the following preprocessing steps **in the order in which they are listed below**. Unfortunately, lemmatization, stemming, and tagging of parts of speech are all too complex for this exercise, but you are expected to implement all the others. For each word:

1. Convert to lowercase. For example, "Hello" is converted to "hello".

2. Remove all punctuation and symbols. For our purposes, this includes any non-alphanumeric character. For example, "full-time" is converted to "fulltime" and "@movie_star" is converted to "moviestar".

3. Remove all numbers UNLESS the token consists **only** of numbers. So, for example, "4real" would become "real", but "2018" would remain "2018".

4. If the word is a stopword (see the list below), remove it.

5. If the word has not been completely removed by steps 1-4, add it to a list of processed words.

When you have finished the processing steps, print the new text to the screen as a space-separated string.

You must use the `input` function to read in the text, and output the processed text using `print`. Your program should print no other information to the terminal. This means that you should not have any input prompt or any additional print formatting. When the input is exhausted (e.g., EOF), your program exits.

Below is the list of stopwords. In the template file `preprocess.py` provided, there is a global list of string literals with these words; the formatting is a bit different to maintain PEP8 compliance. You can use that global list without modification.

```
Stop_Words =
["i", "me", "my", "myself", "we", "our", "ours", "ourselves", "you", "your",
"yours", "yourself", "yourselves", "he", "him", "his", "himself", "she", "her",
"hers", "herself", "it", "its", "itself", "they", "them", "their", "theirs",
"themselves", "what", "which","who", "whom", "this", "that", "these", "those",
"am", "is", "are", "was", "were", "be","been", "being", "have", "has", "had",
"having", "do", "does", "did", "doing", "a", "an","the", "and", "but", "if",
"or", "because", "as", "until", "while", "of", "at", "by", "for", "with",
"about", "against", "between", "into", "through", "during", "before", "after",
"above", "below", "to", "from", "up", "down", "in", "out", "on", "off", "over",
"under", "again", "further", "then", "once", "here", "there", "when", "where",
"why", "how", "all", "any", "both", "each", "few", "more", "most", "other",
"some", "such", "no", "nor", "not", "only", "own", "same", "so", "than",
"too", "very", "s", "t", "can", "will", "just", "don", "should", "now"]
```

**Here are a few sample test cases:**

All of the following should be given using standard input when the program is called using `python3 preprocess.py`.

**Input #1:**

```
I need some help!
```

Output #1:

```
need help
```

**Explanation:** The words "i" and "some" are stopwords, so they are removed.

**Input #2:**

```
I am feeling fine.
```

Output #2:

```
feeling fine
```

**Explanation:** The words "i" and "am" are stopwords, and the period at the end of the sentence is removed.

**Input #3:**

```
I was born in 1968!
```

**Output #3:**

```
born 1968
```

**Explanation:** When the stopwords and punctuation are removed, only these two words remain. Notice that "1968!" becomes "1968" after removing punctuation.

## Your Task #2: Optional Command Line Argument

In this section, you will make your program recognize an optional command line argument called `mode`. The proper usage of your program, `preprocess.py`, is:

```
python3 preprocess.py <mode>
```

where mode is **optional**, meaning that it may or may not be present.

If mode is present, it can be one of:

1. "keep-digits": do not remove numbers from words, but perform all other steps.

2. "keep-stops": do not remove stopwords, but perform all other steps.

3. "keep-symbols": do not remove punctuation or symbols, but perform all other steps.

**If mode is not present,** you should complete all preprocessing steps as outlined above in the full normal version.

You may find it helpful to refer to the description of the Word Frequency exercise to recall how to process command line arguments.

**Here are a few sample test cases:**

All of the following inputs should be given using standard input. For the next three examples, the input will always be:

```
I was born in 1968! This is 4real.
```

Output #1: When the program is called using `python3 preprocess.py keep-digits`

```
born 1968 4real
```

**Explanation:** Because digits are kept, the number 4 is not removed from `4real`.

Output #2: When the program is called using `python3 preprocess.py keep-stops`

```
i was born in 1968 this is real
```

**Explanation:** The stopwords `i`, `was`, `in`, `this`, and `is` are all kept in this mode. However, digits and punctuation are still removed, and the words are still converted to lowercase.

Output #3: When the program is called using `python3 preprocess.py keep-symbols`

```
born ! real.
```

**Explanation:** Punctuation is kept, so the `1968!` becomes `!` after digits are removed.

**Submission Guidelines:**

Using git and GitHub classroom, submit all of the required files by committing and pushing out your work to GitHub. Do NOT include any extra files not required by the specifications.

You are encouraged to commit and push your work to git while working on your solution (e.g., once per day, or even once per hour). Only the last commit and push, prior to the deadline, will be marked. Be careful to **not** do any additional pushes after the deadline.

Git is an excellent tool for backing up your work and git provides a way to revert to a previous version of your file(s), if necessary.

See below for additional hints about git and (possibly) other issues.

In your final commit, and assuming your GitHub identity is **GITHUB-ID**, be sure:

- to maintain the directory structure given to you (via git clone), namely `WE04-GITHUB-ID/preprocess` (this exact name for the directory) with the following files in that directory:

- `preprocess.py` (this exact name) contains all of your Python code and *docstrings-based documentation.*

- your `README` (use this exact name) conforms with the Code Submission Guidelines.

- your `README.WSL` (use this exact name) **only if** you tested your code using **Windows Subsystem for Linux (WSL)** instead of using multipass-Ubuntu. The contents of the `README.WSL` can be blank, because the contents will be ignored.

- `Makefile` (not marked)

- No other files should be submitted.

- Do not remove any files provided via the original git clone.

Note that your files and functions must be named **exactly** as specified above.

Do **not** have any extraneous calls to `input()`, `print()`, or other I/O functions.

When your marked assignment is returned to you, there is a 7-day window to request the reconsideration of any aspect of the mark. After the window, we will only change a mark if there is a clear mistake on our part (e.g., incorrect arithmetic, incorrect recording of the mark). At any time during the term, you can request additional feedback on your submission.

**Marking Rubric:**

NOTE: The code must solve the problem algorithmically. If there is any hardcoding for the provided test cases, then zero Correctness marks will be given.

This Weekly Exercise will be marked out of 100 for:

- `Correctness:` Meets all specifications, generates no extraneous output, requires no unspecified input, and provides correct answers for the test inputs. Note: The output must match the given outputs exactly (e.g., whitespace, every period or character, formatting; read the description carefully) such that the Unix `diff` program cannot detect any differences.

  - `90/90:` Pass all of the provided test inputs
  - `60/90:` Pass any 5 (or more) of the provided test inputs
  - `40/90:` Pass any 3 (or more) of the provided test inputs
  - `20/90:` Pass any of the provided test inputs

- `Valid Submission: 10/10:` Has proper `README` file. If applicable, has `README.WSL`. All other submission requirements are satisfied

This Weekly Exercise emphasizes correctness and conformance to submission guidelines.

Although programming style and code quality (as described in the *Code Submission and Style Guidelines* document on eClass) are always important, we will not emphasize nor mark style for this exercise. Future assessments may include additional requirements for style and design. Students are always encouraged to get feedback on quality and style from a TA or instructor.

As discussed in the Course Outline, this is a Consultation Model assessment. You are allowed to discuss the exercise with fellow students.

Furthermore, going beyond basic consultation, you are allowed to help and receive help from fellow students with your code. During Office Hours or otherwise, you might see the code of fellow students and you are allowed to use that code in your own code. However, if more than 10 lines of code originate from any online resource or other person, you must acknowledge the origins of the code with a comment in the source code **and** in your README file.

These *Acknowledgements* can be considered a part of the *Notes and Assumptions* section of the README.

For example, use a comment of the form:

1. `Code from http://xyz`

2. `Code from person:  J. Smith`

3. `Code from TA: A. Singh`

4. `Code from Office Hours on September 27`

5. `Code from document/book:  Miller-Ranum textbook`

6. `Help received from person:  C. Wong`

Failure to properly acknowledge **all** sources of code will be a violation of the policies for CMPUT 274 and will result in a deduction of marks, at the discretion of the Instructor.

Explicitly disallowed is taking a full solution that someone else has written, making minor changes to it, and submitting it, even with acknowledgement. Be aware that tools such as MOSS might be used to find code common to multiple submissions or code found on the Internet.

**Other Hints and Comments::**

1. After submitting your code, create a new clone of your repository, in a different directory. Then, re-test that new clone.

   This will check if all files and updates have been pushed to GitHub, and if there are any inadvertent path dependencies in the code.

2. Be sure to acknowledge all sources of code and help as a source code comment, and in your README file.

3. Some questions on Quizzes, the Midterm, and the Final Exam will be based on the programs you write for Morning Problems, Weekly Exercises, and Assignments. Therefore, you will need to have a proper understanding of why your solution code works (and does not work).

4. Be sure the output (if any) is exactly as described and shown. As with a Python source code file itself, a single character that is wrong, a missing space, an extra space, or even a missing newline, is incorrect. Testing is automated (as in industry) and both input and output must follow the specification.