CMPUT 274/5 - Tangible Computing **Version 20.0-21**
Weekly Exercise #4

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.

For the purposes of this program, here is the list of stopwords (you can find this list in plain text on eClass (file: `stopwords.txt`):

```
["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.

Your Task #3: Error Handling

Your program should not accept any values for mode other than the ones listed above ("keep-digits", "keep-stops", "keep-symbols"). If any other value is used for mode, you should:

1. Print an error message to the user.
2. Demonstrate proper usage of the program, including the acceptable values for mode.
3. Immediately exit the program.

Submission Guidelines:

Submit all of the required files (and no other files) as **one** properly formed compressed archive called either `preprocess.tar.gz`, or `preprocess.tgz`, or `preprocess.zip` (for full marks, please do **not** use `.rar`):

- when your archive is extracted, it should result in exactly *one directory* called `preprocess` (use this exact name) with the following files in that directory:
- `preprocess.py` (use 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.
- No other files should be submitted.

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.

A new tool has been developed by the TAs to help check and validate the format of your `tar` or `zip` file *prior* to submission.

To run it, you will need to download it into the VM, and place it in the same directory as your compressed archive (e.g., `preprocess.zip`).

You can read detailed instructions and more explanation about this new tool in Submission Validator: Instructions (at the top of the Weekly Exercises tab), or run:

```
python3 submission_validator.py --help
```

after you have downloaded the script to see abbreviated instructions printed to the terminal.

If your submission passes this validation process, and all validation instructions have been followed properly, you will not lose any marks related to the format of your submission. (Of course, marks can still be deducted for correctness, design, and style reasons, but not for submission correctness.)

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 (provided on eClass). 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.
  - `80/80:` Pass all of the provided test inputs
  - `50/80:` Pass any 5 (or more) of the provided test inputs
  - `30/80:` Pass any 3 (or more) of the provided test inputs
  - `20/80:` Pass any of the provided test inputs
- `README:` Part marks possible.
  - `5/5:` Correctly contains all of the required information, and in the right format, as described in the *Code Submission and Style Guidelines*, Chapter 2 (on eClass).
  - `0 to 4/5:` Deductions for missing name, CCID, student number, course name, etc. NOTE: CCID is now required and marked.
- `Submission Validator:` Pass the submission validator.
  - `15/15:` Submission validator outputs `VALIDATION SUCCEEDED`. Otherwise, 0/15.

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 will include marks for style and quality. 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 Solo Effort assessment. Tools such as MOSS might be used to find code common to multiple submissions or code found on the Internet.