CS 210 Homework 2

Pokemon, Covid, Text Processing

Overview

You can work individually or in a group of up to 4 people.

This assignment has three parts:

- pokemon (30 points)
- covid (30 points)
- tfidf (40 points)

Please write your code in files `pokemon.py`, `covid.py`, and `tfidf.py`, respectively.

For this assignment, you may only import `math`, `collections`, `re`, and `csv`.

What to submit

Zip all of these Python files into a single file named `hw2.zip` and submit this on Canvas. Do not include any input or output files, only your Python code.

You can resubmit any number of times. The last submission will be graded.

How to test your code

Your programs should produce the results described in each problem below when run with the python interpreter. For example,

```
python pokemon.py
```

If you're running your programs on ilab, use `python3` explicitly:

```
python3 pokemon.py
```

Problem 1: Pokemon Box Dataset (30 points)

Given a CSV data file as represented by the sample file `pokemonTrain.csv`, perform the following operations on it.

1. (6 points) Find out what percentage of fire type pokemon are at or above the level 40.

   Your program should write (to a file, see below) the value as follows (replace ... with value):

   ```
   Percentage of fire type pokemon at or above level 40 = ...
   ```

   The value should be rounded off using the round() function. So, for instance, if the value is 12.3 (less than or equal to 12.5) you would print 12, but if it was 12.615 (more than 12.5), you would print 13, as in:

   ```
   Percentage of fire type pokemon at or above level 40 = 13
   ```

   Write this string to a file named `pokemon1.txt`.

2. (6 points) Fill in the missing "type" column values (given by `NaN`) by mapping them from the corresponding "weakness" values. You will see that typically a given pokemon weakness has a fixed type, but there are some exceptions. So, fill in the type column with the most common type corresponding to the pokemon's weakness value.

   For example, most of the pokemon having the weakness electric are water type pokemon but there are other types too that have electric as their weakness (exceptions in that type). But since water is the most common type for weakness electric, it should be filled in.

   In case of a tie, use the type that appears first in alphabetical order.

3. (6 points) Fill in the missing values in the Attack ("atk"), Defense ("def") and Hit Points ("hp") columns as follows:

   Set the pokemon level threshold to 40.

   For a pokemon having level above the threshold (i.e. > 40), fill in the missing value for atk/def/hp with the average values of atk/def/hp of pokemon with level > 40. So, for instance, you would substitute the missing atk value for Magmar (level 44), with the average atk value for pokemon with level > 40. Round the average to one decimal place.

   For a pokemon having level equal to or below the threshold (i.e. ≤ 40), fill in the missing value for atk/def/hp with the average values of atk/def/hp of pokemon with level ≤ 40. Round the average to one decimal place.

After performing #2 and #3, write the modified data to another csv file named `pokemonResult.csv`.

The following tasks should be performed on the `pokemonResult.csv` file.

4. (6 points) Create a dictionary that maps pokemon types to their personalities. This dictionary would map a string to a list of strings. For example:

```
{ "fire": ["docile", "modest", ...],
  "normal": ["mild", "relaxed", ...], ...}
```

Note: You can create an empty default dictionary of list with `defaultdict(list)`.

Your dictionary should have the keys ordered alphabetically, and also items ordered alphabetically in the values list, as shown in the example above.

Print the dictionary in the following format:

```
Pokemon type to personality mapping:

   normal: mild, relaxed, ...
   fire: docile, modest, ...
   ...
```

Write the dictionary to a file named `pokemon4.txt`.

5. (6 points) Find out the average Hit Points ("hp") for pokemon of stage 3.0.

Your program should print the value as follows (replace ... with value):

```
Average hit point for pokemon of stage 3.0 = ...
```

You should round off the value, as in #1 above.

Write this to a file named `pokemon5.txt`.

Problem 2: Covid-19 Dataset (30 points)

Given a Covid-19 data CSV file with 12 feature columns, perform the tasks given below. Use the sample file `covidTrain.csv` to test your code.

1. (5 points) In the age column, wherever there is a range of values, replace it by the rounded off average value. E.g., for 10-14 substitute 12. (Rounding should be done as in 1.1). You might want to use regular expressions here, but it is not required.

2. (5 points) Change the date format for the date columns (`date_onset_symptoms`, `date_admission_hospital`, and `date_confirmation`) from `dd.mm.yyyy` to `mm.dd.yyyy`. Again, you can use regexps here, but it is not required.

3. (5 points) Fill in the missing (`NaN`) latitude and longitude values by the average of the latitude and longitude values for the province where the case was recorded. Round the average to 2 decimal places.

4. (5 points) Fill in the missing city values by the most occurring city value in that province. In case of a tie, use the city that appears first in alphabetical order.

5. (10 points) Fill in the missing symptom values by the single most frequent symptom in the province where the case was recorded. In case of a tie, use the symptom that appears first in alphabetical order.

Note: While iterating through records, if you come across multiple symptoms for a single record, you need to consider them individually for frequency counts.

Careful: some symptoms could be separated by a '; ' , i.e., semicolon plus space and some by ';' , i.e., just a semicolon, even within the same record. For example:

```
fever; sore throat;cough;weak; expectoration;muscular soreness
```

After performing all these tasks, write the data back to another CSV file named `covidResult.csv`.

Problem 3: Text Processing (40 pts)

For this problem, you are given a set of documents (text files) on which you will perform some preprocessing tasks, and then compute what is called the TF-IDF score for each word. The TF-IDF score for a word is measure of its importance within the entire set of documents: the higher the score, the more important is the word.

The input set of documents must be read from a file named `tfidf_docs.txt`. This file will list all the documents (one per line) you will need to work with. For instance, if you need to work with the set `doc1.txt`, `doc2.txt`, and `doc2.txt`, the input file `tfidf_docs.txt` contents will look like this:

```
doc1.txt
doc2.txt
doc2.txt
```

Part 1: Preprocessing (20 pts)

For each document in the input set, clean and preprocess it as follows:

1. (10 points) Clean.

   Remove all characters that are not words or whitespace. Words are sequences of letters (upper and lower case), digits, and underscores.

   Remove extra whitespace between words, so that there is exactly one whitespace character between any pair of words.

   Remove all website links. A website link is a sequence of non-whitespace characters that starts with either `http://` or `https://`.

   Convert all the words to lowercase.

   The resulting document should only contain lowercase words separated by a single whitespace character.

2. (5 points) Remove stopwords.

   From the document that results after #1 above, remove "stopwords". These are the non-essential (or "noise") words listed in the file `stopwords.txt`.

3. (5 points) Stemming and Lemmatization.

   This is a process of reducing words to their root forms. For example, look at the following reductions: run, running, runs → run. All three words capture the same idea 'run' and hence their suffixes are not as important.

   (Optional: If you would like to get a better idea, you may want read Stemming and lemmatization.)

   Use the following rules to reduce the words to their root form:

   - Words ending with "ing": "flying" becomes "fly"
   - Words ending with "ly": "successfully" becomes "successful"
   - Words ending with "ment": "punishment" becomes "punish"

   These rules are not expected to capture all the edge cases of stemming in English but are intended to give you a general idea of the preprocessing steps in NLP (Natural Language Processing) tasks.

   After performing #1, #2, and #3 above for each input document, write the modified data to another text file with the prefix `preproc_`. For instance, if the input document is `doc1.txt`, the output should be `preproc_doc1.txt`.

Part 2: Computing TF-IDF Scores (20 pts)

Once preprocessing is performed on all the documents, you need to compute the "Term Frequency - Inverse Document Frequency" (TF-IDF) score for each word. TF-IDF is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus.

Resources:

- tf-idf Wikipedia Page
- TF-IDF/Term Frequency Technique

Steps:

1. (4 points) For each preprocessed document that results from the preprocessing in Part 1, compute frequencies of all the distinct words in that document only. So if you had 3 documents in the input set, you will compute 3 sets of word frequencies, one per document.

2. (4 points) Compute the Term Frequency (TF) of each distinct word (also called term) for each of the preprocessed documents:

   ```
   TF(t) = (# of times term t appears in document) / (Total # of terms in document)
   ```

   Note: The denominator, total number of terms, is the sum total of all the words, not just unique instances. So if a word occurs 5 times, and the total number of words in a document is 100, then TF for that word is 5/100.

3. (4 points) Compute the Inverse Document Frequency (IDF) of each distinct word for each of the preprocessed documents.

   IDF is a measure of how common or rare a word is in a document set (a set of preprocessed text files in this case). It is calculated as follows:

   ```
   IDF(t) = ln((Total # of documents) / (# of documents term t is in)) + 1
   ```

   where `ln` denotes the natural log. Note that since the log term will be at least 0 and we add 1 afterwards, the IDF score is guaranteed to be non-zero.

4. (4 points) Calculate TF-IDF score: `TF * IDF` for each distinct word in each preprocessed document. Round the score to 2 decimal places. (Note that the '-' in "TF-IDF" is a dash, not subtraction.)

5. (4 points) Print the top 5 most important words in each preprocessed document according to their TF-IDF scores. The higher the TF-IDF score, the more important the word. In case of ties in score, pick words in alphabetical order. You should print the result as a list of `(word, TF-IDF score)` tuples sorted in order of descending TF-IDF scores.

   Write to a file prefixed with `tfidf_`. So if the initial input document was `doc1.txt`, you should print the TF-IDF results to `tfidf_doc1.txt`.

Testing

1. You can begin with the following three sentences as separate documents against which to test your code:

   ```
   doc1 = "It is going to rain today."
   doc2 = "Today I am not going outside."
   doc3 = "I am going to watch the season premiere."
   ```

   You can match values computed by your code with this same example in the TF-IDF/Term Frequency Technique page referenced above. Look for it under "Let's cover an example of 3 documents" on this page. (Note: We are adding 1 to the log for our IDF computation.)

2. Next, you can test your code against `test1.txt` and `test2.txt`. Compare your resulting preprocessed documents with our results in `preproc_test1.txt` and `preproc_test2.txt`, and your TF-IDF results with our results in `tfidf_test1.txt` and `tfidf_test2.txt`.

3. Finally, you can try your code on these files: `covid_doc1.txt`, and `covid_doc2.txt`, and `covid_doc3.txt`. Results for these are not provided, however the files are small enough that you can identify the words that make the cut and manually compute TF-IDF.