CMPUT 274 — Tangible Computing I **Version 2.0.1-21**
Assignment 1 — OO Classifier

Introduction

Assignment #1 is worth 15% of your final mark.

For this assignment, you will be extending the object-oriented (OO) classifier (`ooclassifierbase.py`) program provided to you. Your modified program will be renamed `ooclassifier.py` (see Submission Guidelines). You will be adding preprocessor functionality (similar to Weekly Exercise (WE) #4), adding a new classification algorithm (based on frequent words, similar to WE #3), and adding the ability to support n-fold cross-validation.

In a change from previous work so far, this assignment:

1. Includes a substantial amount of code given to you (see eClass, and upcoming class discussion) that must be used (i.e., extending code, using OO programming)
2. Allows you to re-use some of your own code from past Weekly Exercises
3. Has marks allocated for Style and Design, including for docstrings-based documentation
4. For correctness, we will use some **new** tests and test data that are not given to you in advance (i.e., holdout)
5. For correctness, we will use `diff -w` for comparisons
6. **Some** aspects of this assignment are subject to **reasonable design decisions** (details below)
7. This assignment follows the Consultation Model (not Solo Model)

An assignment of this scope includes many unspecified details for which a **reasonable design decision** can be made (i.e., does **not** contradict an existing specification), stated explicitly (e.g., documented in a comment in the code and/or README), and implemented. The finished OO classifier must still generate output that matches the tests provided. Care and attention should be paid to designing **new** tests and input that meet the specifications.

This assignment will require you to know something about:

1. Machine-learned classifiers
2. Python programming
3. Classes and object-oriented programming in Python
4. Preprocessing and stopwords
5. Word frequency
6. Cross validation
7. Confusion matrices
8. Style and Design considerations
9. Testing strategies

The main challenge of this assignment is **not** in the total number of lines of code that has to be implemented, but rather in understanding the original code (`ooclassifierbase.py`) well enough to be able to extend it with new functionality. To be sure, there is a non-trivial amount of new Python code that must be written, but this assignment is not only about writing code.

Task I: Preprocessing in the Classifier

1. Add a new method `preprocess_words(mode='')` to class `TrainingInstance`.

   When invoked, `preprocess_words(mode='')` applies all of the preprocessing actions from Weekly Exercise #4 Text Preprocessor to all the words in that particular training instance object.

2. Add a new method `preprocess(mode='')` to class `TrainingSet`.

   When invoked, `preprocess(mode='')` will use `preprocess_words(mode='')` in class `TrainingInstance` to perform preprocessing for all training instances in a particular training dataset.

3. You may add additional methods and attributes to class `TrainingSet` and class `TrainingInstance`, as long as `preprocess(mode='')` and `preprocess_words(mode='')` work as required.

4. Do not change the interface or semantics of any of the existing methods and attributes of the existing classes.

When `preprocess(mode='')` returns, **all** of the training instances in the **entire** training dataset will have been preprocessed according to the specifications in Weekly Exercise #4 Text Preprocessor. The full description of the preprocessing steps are as described in section "Your Task #1: Full Preprocessing" of Weekly Exercise #4.

Note that, after preprocessing, the training instances still need to be explicitly re-classified. In computing science, preprocessing and processing (e.g., classifying) are usually separate steps, because there may be multiple steps or kinds of "preprocessing" (e.g., doing some but not all modes of preprocessing; see below) before the data is actually used.

Both new methods take a default argument called `mode` that is a string. If the string is empty, then it performs the full preprocessing from "Your Task #1: Full Preprocessing" of Weekly Exercise #4.

When the string `mode` is one of the values `'keep-digits'`, `'keep-stops'` or `'keep-symbols'`, the methods behave according to the description in section "Your Task #2: Optional Command Line Argument" of Weekly Exercise #4. Whereas in Weekly Exercise #4 the mode came from the command line, in this assignment the mode parameter comes from the call-site(s) of `preprocess_words()` and `preprocess()`.

Task II: Word Frequency Classification Algorithm

So far, the classification algorithm based on hard-coded target words works well enough, but there is actually no machine learning in the algorithm. The target words are hard-coded beforehand and do not change based on the input or training dataset, which is necessary for machine learning.

Therefore:

1. Create a new class `ClassifyByTopN`, which is a subclass of class `ClassifyByTarget`.

2. The class `ClassifyByTopN` includes a new method `target_top_n(tset, num=5, label='')` which replaces the current list of target words with a new list of target words based on word frequency. Details below.
3. You may add additional methods and attributes to class `ClassifyByTopN`, as long as `target_top_n(tset, num=5, label='')` works as required.
4. Do not change the interface or semantics of any of the existing methods and attributes of the existing classes.

Specifically, `target_top_n(tset, num=5, label='')` counts all of the words, in all of the training instances, whose label match the string `label`, of object `tset` which is of class `TrainingSet`.

Of course, if `tset` has been preprocessed, then the word frequency count will be based on the words remaining after the preprocessing is done. Otherwise, the word frequency count is based on whatever words are currently in the training dataset.

After the word frequency count, the top `num` most frequent words become the new target words list. The default value for `num` is 5. If there is a tie for the count at the `num`-th rank, then **all** of the words with the same count are also included in the new list of target words. Therefore, it is possible for the number of target words to be larger than `num`, due to ties in the counts.

Since class `ClassifyByTopN`, is a subclass of class `ClassifyByTarget`, it should support all of the existing methods of class `ClassifyByTarget`, such as (in particular, but not a complete list) `classify()`, `eval_training_set()`, and `print_confusion_matrix()`.

Task III: Create N-folds for Cross Validation

N-fold cross validation is an important technique in evaluating machine learning (ML) algorithms and training data, such as classifiers. If an ML model trains on data, and then that same data is used to evaluate the model, one could simply be evaluating how well the model memorizes (see memoization) the answers it has seen before. There is the related topic of *overfitting* in ML, but we do not discuss it any further here.

By analogy, it is like giving out all the questions and answers to the final exam beforehand, and then only using those questions on the actual final exam. Such an exam would be testing one's memory, not one's understanding.

In n-fold cross validation, a portion of the training data is withheld, the rest of the training data is used, then the withheld portion is used to test classifier. Specifically, the whole training dataset is divided into n different partitions or folds. When combined (e.g., set union), the n folds recreate the original training dataset (possibly with some changes in ordering).

1. Create a new method `return_nfolds(num=3)` in class `TrainingSet`, that returns a list of `num` objects of class `TrainingSet`. Each of the objects returned contains a partition or **fold** of the original training dataset. Each of the objects should contain a deepcopy of all attributes (but only for the instances of that partition or fold) of the `TrainingSet`. The default value for `num` is 3.

   The training instances in the training dataset should be assigned to the discrete partitions using a basic **round robin** (or interleaved) strategy. For example, suppose an object of class `TrainingSet`

contains exactly 7 training instances (i.e., objects of class `TrainingInstance`) (A, B, C, D, E, F, G). If `return_nfolds(num=3)` is invoked, it returns [(A, D, G), (B, E), (C, F)], which represents 3 discrete partitions of the 7 training instances, where each partition is an object of class `TrainingSet`.

Note that the first partition contains 3 training instances (instead of 2) because 3 does not divide evenly into 7. It is possible that the folds are not exactly equal in size, as they can be off-by-one depending on whether `num` divides evenly into the number of training instances.

Each of these 3 objects of class `TrainingSet` contain all of the attributes (via a deepcopy) for their respective 2 or 3 training instances (and only their training instances).

2. Create a new method `copy()` in class `TrainingSet` that returns (after making a deepcopy) an object of class `TrainingSet` that contains the same attributes (e.g., training instances) as the original object of class `TrainingSet`. This `copy()` is similar to `mylist.copy()`, except that deepcopy (instead of a shallow copy) is used.
3. Create a new method `add_training_set(tset)` in class `TrainingSet` that adds (via deepcopy) **all** the training instances of `tset` (which is of class `TrainingSet`) to an object of class `TrainingSet`.
4. You may add additional methods and attributes to class `TrainingSet`, as long as methods `return_nfolds(num=3)`, `copy()`, and `add_training_set(tset)` work as required.
5. Do not change the interface or semantics of any of the existing methods and attributes of the existing classes.

Do not be concerned with any time or space efficiency issues related to the use of deepcopy. Those concerns can be addressed by future optimizations.

In combination with class `ClassifyByTopN`, proper n-fold cross validation experiments can now be performed.

Sample Output:

Several sample unit-test drivers are already on eClass. A few more drivers and tests will be provided in the next few days. In particular, end-to-end drivers will be provided.

The drivers and tests will be discussed in class, since they are too complicated for written documentation.

Testing and Other Hints:

In addition to any hints already given elsewhere, please consider the following:

Design, write, and use your own test programs. Do not rely just on the tests provided. Write your own tests too. You can start with modified versions of the tests and drivers already provided.

Students are welcome to share their (unofficial) *tests*, **not** solution code (e.g., on eClass) as the basis for discussing the specifications. If necessary, we will clarify important ambiguities in the specifications, but we may leave other points to be decided by the programmer as **reasonable design decisions**. For example, two programmers may make different reasonable design decisions that result in different output or results for the holdout tests. If the instructors agree that the design decisions are reasonable, then marks will still be given.

Be sure to document (e.g., in the README) your design decisions.

One benefit of writing new tests is that it forces a close re-reading of the problem description to reveal any false assumptions (which can then be fixed) or additional assumptions (which can be then be documented as design decisions, if appropriate).

Double-check your submission, before and after you submit. If you submit the wrong file(s), or if eClass does not receive your submission properly, we will not accept a submission the final deadline. Re-download what you submit on eClass. Double-check it. Ideally, you will submit and double-check your submission before the deadline. But, at least check after the deadline and take advantage of late (with penalty) option to fix any mistakes.

Submission Guidelines:

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

- when your archive is extracted, it should result in exactly *one directory* called `ooclassifier` (use this exact name) with the following files in that directory:
- `ooclassifier.py` (use this exact name) contains all of your Python code and *docstrings-based documentation*. NOTE: We will be testing your code via `from ooclassifier import *` or similar.
- 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 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., `ooclassifier.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 assessment 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 (some provided on eClass). Note: The output must match the given outputs exactly (e.g., every period or character, formatting; read the description carefully) such that the Unix `diff -w` program cannot detect any differences.
  - `60/60`: Pass all of the provided and **new (i.e., holdout) test** inputs. Consideration will be made for documented **reasonable design decisions**, if the provided test inputs are handled correctly.
  - `30/60`: Pass all of the provided test inputs and exactly match the expected output
  - `10/60`: Pass at least one of the provided test inputs and exactly match the expected output
- `Style and Design`: Part marks likely, as this is subjective.
  - `20/20`: Meets all style and design requirements discussed in the *Code Submission and Style Guidelines* (on eClass)
  - `10/20`: Meets most (but not all) style and design requirements
  - `5/20`: Meets some (but not all) style and design requirements *Code Submission and Style Guidelines* (on eClass)
- `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, **consultation information**, etc. NOTE: CCID is now required and marked.
- `Submission Validator`: Pass the submission validator.
  - `15/15`: Submission validator outputs `VALIDATION SUCCEEDED`. Otherwise, 0/15.

Students are always encouraged to get feedback on design and style from a TA or instructor before a submission. However, no feedback before submission can guarantee that all aspects of the *Code Submission and Style Guidelines* are met. Ultimately, the author of the submission is responsible for meeting the *Code Submission and Style Guidelines*.

As discussed in the Course Outline, this is a **Consultation Model** assessment. Students may have high-level verbal discussions (aka consultation) with other students but may not take notes from these discussions nor write or share code in any form. **You must document in your assignment submission (e.g., in your README file) any students you consulted.**

Tools such as MOSS might be used to find code common to multiple submissions or code found on the Internet.