ECE250 Project 3: Hierarchical Text Classification

Due: Monday, November 18, 2024 at 11:00pm to the dropbox on Learn

Overview

A very common problem in natural language processing is to classify an input text according to some set of possible classes. In recent years, classifying text has become much easier by leveraging the power of language models as text classifiers. However, even language models do poorly if the number of classes is very large. For this reason, hierarchical classification is often used, where we begin by classifying text into a broad base class, then refine the classification based on which base class the text belongs to. For example, the text "apple tree" might belong to the class "living things". We might refine that class further by saying it is a "living thing that is a plant". We might refine it yet further by saying it is a "living thing that is a plant and is a tree" and so forth. This refinement lends itself well to a tree implementation, in which each level of the tree represents a more specific set of classes that are specific to the parent.

In this project, you will implement a trie data structure designed for hierarchical text classification. The trie is an N-ary tree, where each level of the tree represents a more specific class in the hierarchy. The root node represents the most general class. If the k^{th}subtree is not null, it represents a class that is a refinement of the parent class with the k^{th} child node corresponding to the more specific classification. Each node in the trie may additionally indicate that it is a terminal node, which signifies that it represents a final classification level. For the purposes of this assignment, it is *not* possible for an internal node to be a terminal node. In any trie the number of possible child nodes is fixed. For our purposes in this project, we will fix N = 15, indicating that no class will ever have more than 15 subclasses.

The trie you create will only store classes and their subclasses. We have set up a language model classifier for you to interact with, and have written a library of code that allows you to do so. You must include the library in your own cpp files and compile it using your makefile. The library is best used on Linux or Mac. Remember, everyone has access to a graphical Ubuntu environment, and instructions to access it were sent out via email from Hugh.

Program Design and Documentation

You must use proper Object-Oriented design principles to implement your solution. You will create a design using classes which have appropriately private/protected data members and appropriately public services (member functions). It is not acceptable to simply make all data members public. **Structs are not permitted.** Write a short description of your design (three pages) to be submitted along with your C++ solution files for marking according to the template posted on LEARN. **You may use the vector class and the "algorithm" library, but no others from the STL**.

Input/Output Requirements

In this project, you must create a test program that must read commands from standard input and write the output to standard output. The commands are described below.

Command	**Parameters**	**Description**	**Expected Runtime**	**Output**
LOAD	filename	Load the classes stored in the file filename into your trie. The format of this file is <base_class>,<subclass>,<subsubclass>… For instance: living living,animal nonliving,artificial,human-made object If a class is already in the trie, do nothing for that line of the file	**N/A**	**success** ***Note:*** This command should output "success" no matter what.

INSERT	classification	Insert a new classification into the trie. This classification will follow the comma-separated values format that the input files do, such as: nonliving,natural,mineral,gemstone	**O(n)** *where n is the number of classes in the classification*	**success** if the insertion was successful **failure** if the classification is already in the trie **illegal argument** (see below the table)
CLASSIFY	input	Classifies the input phrase using the classifier and the trie. This requires you to traverse your trie and call the classifier function at each level, providing the new class options as you do. Note: other than illegal arguments, you may assume that this command always successfully provides a classification, even if that classification is objectively incorrect. ***The LLM can make mistakes***. We know this and will ensure that grading is not affected by LLM errors.	**O(N)** *where N is the number of classes in the trie. Do not consider the runtime of the classifier. It's awful.*	**<base_class>,<subclass>,<subsubclass>…** **illegal argument** (see below the table)
ERASE	classification	Erase the entire classification in the trie. Classifications will be given in the comma-separated format above.	**O(n)** *where n is the number of classes in the classification*	**success** if the classification is in the trie and was erased **failure** if the classification is not in the trie, or the trie is empty **illegal argument** (see below the table)
PRINT		Prints all classifications in the trie on a single line. Classifications are to be printed in the comma-separated format, but each classification will be separated by the others by an underscore. Like this: living,plant_nonliving,natural,gemstone_nonliving,human-made object,electronic,computer	**O(N)** *where N is the number of classifications in trie*	**classification1_classification2_classification3…** There should be a new line after the last classification, but otherwise all classifications should be printed on a single line. Do not worry about sorting the classifications, we will handle that during autograding. **trie is empty** if the trie is empty
EMPTY		Checks if the trie is empty.	**O(1)**	**empty 1** if the trie is empty **empty 0** if the trie is not empty
CLEAR		Deletes all classifications from the trie.	**O(N)** *where N is the number of nodes in trie*	**success** if the trie was already empty, this command should print "success" anyway

Command	Parameters	Description	Expected Runtime	Output
SIZE		Prints a message indicating the number of classifications in the trie.	**O(1)**	**number of classifications is** *count* where "*count*" is the number of classifications. Count may be 0 if the trie is empty.
EXIT		Last command for all input files.		This command does not print any output.

Illegal arguments: For the commands *INSERT, CLASSIFY,* and *ERASE,* you must handle invalid input. If the input contains any upper-case English letters your code must throw an illegal_argument exception, catch it, and output **"illegal argument"** (without quotes) if it is caught. You should read the entire line of input even if it has illegal arguments. Afterward, the code should continue parsing input if any remains. You may assume that the only illegal argument will be upper-case English letters, and otherwise all inputs will be valid as specified above. To do this, you will need to:

a. Define a class for this exception, call it **illegal_exception**
b. Throw an instance of this class when the condition is encountered using this line:

```
throw illegal_exception();
```

c. Use a try/catch block to handle this exception and print the desired output of the command

You must analyze the expected runtime of your algorithms in your design document. Prove that your implementation achieves these runtimes.

Valgrind and Memory Leaks, Formatting, and Commenting

10% of the grade of this project will be allocated to memory leaks. We will be using the Valgrind utility to do this check. The expected behaviour of Valgrind is to indicate 0 errors and 0 leaks possible, with all allocated bytes freed. A penalty of 10% will be applied for poor commenting or code organization. A penalty of 15% will be applied if non-permitted header files or structs are used.

Test Files

Learn contains some examples input files with the corresponding output. The files are named *test01.in*, *test02.in* and so on with their corresponding output files named *test01.out* etc.

All lines in the input files end with a UNIX newline character (\n), and there are no spaces before this character.

Submitting your Program

Once you have completed your solution and tested it comprehensively on your own computer or the lab computers, you should transfer your files to the eceUbuntu server and test there. We perform automated tested on this platform, so if your code works on your own computer but not on eceUbuntu it will be considered incorrect. **A makefile is required for this project since the exact source structure you use will be unique to you. A video has been posted on Learn about how to create it.**

Once you are done your testing you must create a compressed file in the tar.gz format, that contains:

- A typed document, maximum of three pages, describing your design. Submit this document in PDF format. The name of this file should be xxxxxxxx_design_pn.pdf where xxxxxxxx is your maximum 8-character UW user ID (for example, I would use my ID "mstachow", not my ID "mstachowsky", even though both are valid UW IDs), and n is the project number. In my case, my file would be mstachow_design_p3.pdf.
- A test program, trietest.cpp, that reads the commands and writes the output.
- Required header files that you created.
- Any additional support files that you created.

- A makefile, named Makefile, with commands to compiler your solution and creates an executable. Do not use the -o output flag in your makefile. The executable's name must be a.out.

The name of your compressed file should be **xxxxxxxx_p3.tar.gz**, where xxxxxxxx is your UW ID as above.