p5-ml

EECS 280 Project 5: Machine Learning

Project Due Friday, 7 Dec 2018, 8pm

In this project, you will write a program that uses natural language processing and machine learning techniques to automatically identify the subject of posts from the EECS 280 Piazza. You will gain experience with recursion, binary trees, templates, comparators, and the map data structure. Another goal is to prepare you for future courses (like EECS 281) or your own independent programming projects, so we have given you a lot of freedom to design the structure of your overall application.

This project will be autograded for correctness, comprehensiveness of your test cases, and programming style. See the style checking tutorial for the criteria and how to check your style automatically on CAEN.

You may work alone or with a partner. Please see the syllabus for partnership rules.

Academic Integrity

As a reminder, you may not share any part of your solution outside of your partnership. This includes both code and test cases.

Do not post code or pseudocode on Piazza. While we encourage you to continue to help your classmates on Piazza, please don't post function implementation details.

Table of Contents

- Project Roadmap
- Project Introduction
- Project Essentials
- The BinarySearchTree ADT
  - Testing `BinarySearchTree`
- The Map ADT
  - Testing Map
- The Piazza Datasets

- Classifying Piazza Posts with NLP and ML
  - The Bag of Words Model
  - Training the Classifier
  - Predicting a Label for a New Post
  - Implementing Your Top-Level Classifier Application
  - Classifier Application Interface
  - Output
  - Results
- Appendix A: Map Example
- Appendix B: Splitting a Whitespace-Delimited String

Project Roadmap

1. **Set up your IDE**

   Use the tutorial from project 1 to get your visual debugger set up. Use this `wget` link `https://eecs280staff.github.io/p5-ml/starter-files.tar.gz`.

   Before setting up your visual debugger, you'll need to rename each `.h.starter` file to a `.h` file.

   ```
   $ mv BinarySearchTree.h.starter BinarySearchTree.h
   $ mv Map.h.starter Map.h
   ```

   You'll also need to create these new files and add function stubs.

   ```
   $ touch main.cpp
   ```

   These are the executables you'll use in this project:

   - `BinarySearchTree_compile_check.exe`
   - `BinarySearchTree_public_test.exe`
   - `BinarySearchTree_tests.exe`
   - `Map_compile_check.exe`
   - `Map_public_test.exe`
   - `main.exe`

   If you're working in a partnership, set up version control for a team.

2. **Read the Project Introduction and Project Essentials**

   See the first sections below for an introduction to the project as well as essential instructions for successfully completing the project.

3. **Test and implement the BinarySearchTree data structure**

   We've provided header files with comments. Test and implement those functions. Be sure to use recursion and tail recursion where the comments require it.

4. **Test and implement the Map data structure**

   Implement and test a Map ADT that internally uses your BinarySearchTree to provide an interface that works (almost) exactly like `std::map` from the STL! Appendix A has an example.

5. **Test and implement the Piazza Classifier Application**

   This specification describes the interface for the overall application, but it's up to you how to separate it into functions and data structures.

   Appendix B has tips and tricks for this part.

Submit to the Autograder

Submit the following files to the autograder.

- `BinarySearchTree.h`
- `Map.h`
- `main.cpp`
- `BinarySearchTree_tests.cpp`

Project Introduction

The goal for this project is to write an intelligent program that can **classify** Piazza posts according to topic. This task is easy for humans - we simply read and understand the content of the post, and the topic is intuitively clear. But how do we compose an algorithm to do the same? We can't just tell the computer to "look at it" and understand. This is typical of problems in artificial intelligence and natural language processing.

We know this is about Euchre, but how can we write an algorithm that "knows" that?

With a bit of introspection, we might realize each individual word is a bit of evidence for the topic about which the post was written. Seeing a word like "card", "spades", or even "bob" leads us toward the Euchre project. We judge a potential label for a post based on how likely it is given all the evidence. Along these lines, information about how common each word is for each topic essentially constitutes our classification algorithm.

But we don't have that information (i.e. that algorithm). You could try to sit down and write out a list of common words for each project, but there's no way you'll get them all. For example, the word "lecture" appears much more frequently in posts about exam preparation. This makes sense, but we probably wouldn't come up with it on our own. And what if the projects change? We don't want to have to put in all that work again.

Instead, let's write a program to comb through Piazza posts from previous terms (which are already tagged according to topic) and learn which words go with which topics. Essentially, the result of our program is an algorithm! This approach is called (supervised) machine learning. Once we've trained the classifier on some set of Piazza posts, we can apply it to new ones written in the future.

Authors

This project was developed for EECS 280, Fall 2016 at the University of Michigan. Andrew DeOrio and James Juett wrote the original project and specification. Amir Kamil contributed to code structure, style, and implementation details.

Project Essentials

The project consists of three main phases:

1. Implement and test the static `_impl` member functions in `BinarySearchTree`.

2. Implement and test `Map` by using the has-a pattern on top of `BinarySearchTree`.
3. Design, implement, and test the top-level classifier application.

The focus of part 1 is on working with recursive data structures and algorithms. The framework and some of the implementation for `BinarySearchTree` is provided for you, but you must implement the core functionality in several static member functions. Be mindful of requirements for which implementations must use certain kinds of recursion.

Part 2 should not require a lot of additional implementation code. Make sure to reuse the functionality already present in `BinarySearchTree` wherever possible.

For your top-level application, you must use `std::map` in place of `Map`. This means a bug in parts 1 or 2 will not jeopardize your ability to complete part 3. Additionally, the implementation of `BinarySearchTree` (and consequently `Map`) we have you write will not be fast enough for the classifier.

Requirements and Restrictions

DO	DO NOT
Put all top-level application code in `main.cpp`.	Create additional files other than `main.cpp`.
Create any ADTs or functions you wish for your top-level classifier application.	Modify the `BinarySearchTree` or `Map` public interfaces
Use any part of the STL for your top level classifier application, including `map` and `set`.	Use STL containers in your implementation of `BinarySearchTree` or `Map`.
Use any part of the STL except for containers in your `BinarySearchTree` and `Map` implementations.	Use your `Map` implementation for the top level application. It will be too slow.
Use recursion for the BST `_impl` functions.	Use iteration for the BST `_impl` functions.
Follow course style guidelines.	Use static or global variables.

Starter Files

The following table describes each file included in the starter code. As you begin development, rename files to remove `.starter`.

Filename	Description
`BinarySearchTree.h.starter`	Defines an ADT for a binary search tree.
`BinarySearchTree_tests.cpp`	Add your BST tests to this file.
`BinarySearchTree_public_test.cpp`	A public test for `BinarySearchTree`
`BinarySearchTree_compile_check.cpp`	A compilation test for `BinarySearchTree.h`
`TreePrint.h`	Auxiliary file to support printing trees. You do not need to look at this file. Do not modify it.
`Map.h.starter`	`Map` ADT
`Map_public_test.cpp`	A sample test for `Map` . You are encouraged to write map tests, but do not submit them.
`Map_public_test.out.correct`	Correct output for the `Map` public test.
`Map_compile_check.cpp`	A compilation test for `Map.h` .
Piazza Datasets (Four `.csv` files)	Piazza post data from several past EECS 280 terms in Comma Separated Value (CSV) format.
`csvstream.h`	A library for reading data in CSV format.

`train_small.csv`
`test_small.csv`
`test_small.out.correct`
`test_small_debug.out.correct`
	Sample input training and testing files for the classifier application, as well as the corresponding correct output when run with those files.

`sp16_projects_exam.csv`
`w14-f15_instructor_student.csv`
`w16_instructor_student.csv`
`w16_projects_exam.csv`
`instructor_student.out.correct`
`projects_exam.out.correct`
	Actual Piazza data from past terms, as well as the corresponding correct output when run with those files.

`Makefile`	Used by the `make` command to compile the executable.
`unit_test_framework.h` `unit_test_framework.cpp`	The unit test framework you must use to write your test cases.

The BinarySearchTree ADT

A binary search tree supports efficiently storing and searching for elements.

Template Parameters

`BinarySearchTree` has two template parameters:

- `T` - The type of elements stored within the tree.
- `Compare` - The type of comparator object (a functor) that should be used to determine whether one element is less than another. The default type is `std::less<T>`, which compares two `T` objects with the `<` operator. To compare elements in a different fashion, a custom comparator type must be specified.

No Duplicates Invariant

In the context of this project, duplicate values are NOT allowed in a BST. This does not need to be the case, but it avoids some distracting complications.

Sorting Invariant

A binary search tree is special in that the structure of the tree corresponds to a sorted ordering of elements and allows efficient searches (i.e. in logarithmic time).

Every node in a well-formed binary search tree must obey this sorting invariant:

- It represents an empty tree (i.e. a null `Node*`).

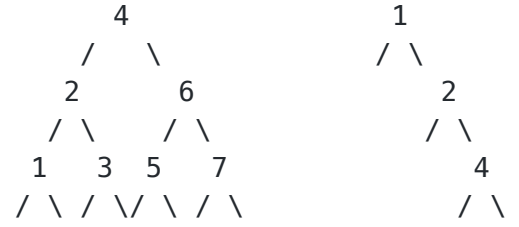
- OR -

- The left subtree obeys the sorting invariant, and every element in the left subtree is less than the root element (i.e. this node).

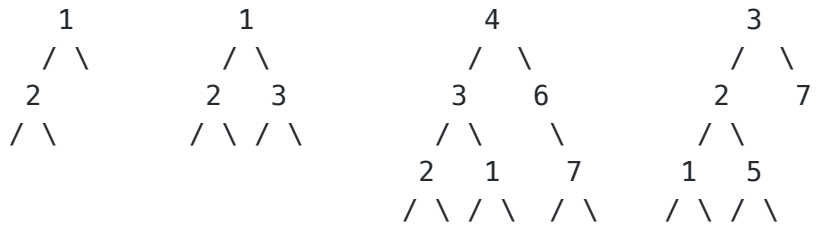
  - AND -

  The right subtree obeys the sorting invariant, and the root element (i.e. this node) is less than every element in the right subtree.

Put briefly, go left and you'll find smaller elements. Go right and you'll find bigger ones. For example, the following are all well-formed sorted binary trees:



While the following are not:



Data Representation

The data representation for `BinarySearchTree` is a tree-like structure of nodes similar to that described in lecture. Each `Node` contains an element and pointers to left and right subtrees. The structure is self-similar. A null pointer indicates an empty tree. **You must use this data representation. Do not add member variables to `BinarySearchTree` or `Node`.**

Public Member Functions and Iterator Interface

The public member functions and iterator interface for `BinarySearchTree` are already implemented in the starter code. DO NOT modify the code for any of these functions. They delegate the work to private, static implementation functions, which you will write.

Implementation Functions

The core of the implementation for `BinarySearchTree` is a collection of private, static member functions that operate on tree-like structures of nodes. **You are responsible for writing the implementation of several of these functions.**

To disambiguate these implementation functions from the public interface functions, we have used names ending with `_impl`. (This is not strictly necessary, because the compiler can differentiate them based on the `Node*` parameter.)

There are a few keys to thinking about the implementation of these functions:

- **The functions have no idea that such a thing as the `BinarySearchTree` class exists**, and they shouldn't. A "tree" is not a class, but simply a tree-shaped structure of `Node`s. The parameter node points to the root of these nodes.
- A recursive implementation depends on the idea of **similar subproblems**, so a "subtree" is just as much a tree as the "whole tree". That means you shouldn't need to think about "where you came from" in your implementation.
- Every function should have a **base case**! Start by writing this part.

- You only need to think about one "level" of recursion at a time. Avoid thinking about the contents of subtrees and **take the recursive leap of faith**.

We've structured the starter code so that the first bullet point above is actually enforced by the language. Because they are `static` member functions, they do not have access to a receiver object (i.e. there's no `this` pointer). That means it's actually impossible for these functions to try to do something bad with the `BinarySearchTree` object (e.g. trying to access the `root` member variable).

Instead, the implementation functions are called from the regular member functions to perform specific operations on the underlying nodes and tree structure, and are passed only a pointer to the root `Node` of the tree/subtree they should work with.

The `empty_impl` function must run in constant time. It must must be able to determine and return its result immediately, without using either iteration or recursion. The rest of the implementation functions must be recursive. There are additional requirements on the kind of recursion that must be used for some functions. See comments in the starter code for details. **Iteration (i.e. using loops) is not allowed in any of the `_impl` functions**.

Using the Comparator

The `_impl` functions that need to compare data take in a comparator parameter called `less`. Make sure to use `less` rather than the `<` operator to compare elements!

The `insert_impl` Function

The key to properly maintaining the sorting invariant lies in the implementation of the `insert_impl` function - this is essentially where the tree is built, and this function will make or break the whole ADT. Your `insert_impl` function should follow this procedure:

1. Handle an originally empty tree as a special case.
2. Insert the element into the appropriate place in the tree, keeping in mind the sorting invariant. You'll need to compare elements for this, and to do so make sure to use the `less` comparator passed in as a parameter.
3. Use the recursive leap of faith and call `insert_impl` itself on the left or right subtree. Hint: You **do** need to use the return value of the recursive call. (Why?)

Important: When recursively inserting an item into the left or right subtree, be sure to replace the old left or right pointer of the current node with the result from the recursive call. This is essential, because in some cases the old tree structure (i.e. the nodes pointed to by the old left or right pointer) is not reused. Specifically, if the subtree is empty, the only way to get the current node to "know" about the newly allocated node is to use the pointer returned from the recursive call.

Technicality: In some cases, the tree structure may become unbalanced (i.e. too many nodes on one side of the tree, causing it to be much deeper than necessary) and prevent efficient operation for large trees. You don't have to worry about this.

Testing `BinarySearchTree`

You must write and submit tests for the `BinarySearchTree` class. Your test cases MUST use the unit test framework, otherwise the autograder will not be able to evaluate them. Since unit tests should be small and run quickly, you are limited to **50** `TEST()` items per file, and your whole test suite must finish running in less than 5 seconds. Please bear in mind that you DO NOT need 50 unit tests to catch all the bugs. Writing targeted test cases and avoiding redundant tests can help catch more bugs in fewer tests.

ProTip: When writing tests for functions that return a `size_t` (which is an unsigned integer type), compare against an unsigned literal. For example:

```
BinarySearchTree<int> b;
ASSERT_EQUAL(b.height(), 0u);
```

How We Grade Your Tests

We will autograde your `BinarySearchTree` unit tests by running them against a number of implementations of the module. If a test of yours fails for one of those implementations, that is considered a report of a bug in that implementation.

We grade your tests by the following procedure:

1. We compile and run your test cases with a **correct** solution. Test cases that pass are considered **valid**. Tests that fail (i.e. falsely report a bug in the solution) are invalid. The autograder gives you feedback about which test cases are valid/invalid. Since unit tests should be small and run quickly, your whole test suite must finish running in **less than 5 seconds**.
2. We have a set of intentionally **incorrect** implementations that contain bugs. You get points for each of these "buggy" implementations that your **valid** tests can catch.
3. How do you catch the bugs? We compile and run all of your **valid** test cases against each buggy implementation. If **any** of these test cases fail (i.e. report a bug), we consider that you have caught the bug and you earn the points for that bug.

The Map ADT

The `Map` ADT works just like `std::map`. `Map` has three template parameters for the types of keys and values, as well as a customizable comparator type for comparing keys. The most important functions are `find`, `insert`, and the `[]` operator. The RMEs and comments in `Map.h` provide the details, and appendix A includes an example.

Note: Although you must implement `Map`, use `std::map` instead in your top-level application. Our implementation of `Map` is not fast enough for the classifier.

Building on the BST

The operation of a map is quite similar to that of a BST. The additional consideration for a map is that we want to store key-value pairs instead of single elements, but also have any comparisons (e.g. for searching) only depend on the key and be able to freely change the stored values without messing up the BST sorting invariant. We can employ the has-a pattern using a `BinarySearchTree` as the data representation for Map:

- BST template parameter: `T`

  Instantiate with: `Pair_type`

  We've provided a using declaration in the starter code for `Pair_type`:

  ```
  using Pair_type = std::pair<Key_type, Value_type>;
  ```

  `std::pair` is basically like a struct that stores two objects together. `Key_type` and `Value_type` are whatever template parameters were used to instantiate `Map`.

- BST template parameter: `Compare`

  Instantiate with: `PairComp`

  You'll need to define your own comparator by declaring a functor type called `PairComp` (or whatever you want to call it) in your `Map` class. The overloaded `()` operator should accept two objects of `Pair_type` and return whether the key of the LHS is less than the key of the RHS (according to `Key_compare`).

Finally, we can even reuse the iterators from the BST class, since the interface we want (based on `std::map`) calls for iterators to yield a key-value pair when dereferenced. Since the element type `T` of the BST is our `Pair_type`, BST iterators will yield pairs and will work just fine. We've provided this `using` declaration with the starter code to make `Map::Iterator` simply an alias for iterators from the corresponding BST:

```
using Iterator = typename BinarySearchTree<Pair_type, PairComp>::Iterator;
```

Testing Map

You are encouraged to write tests for the `Map` ADT, but they are not required for the project submission. Do not submit them to the autograder.

The Piazza Datasets

For this project, we retrieved archived Piazza posts from EECS 280 in past terms. We will focus on two different ways to divide Piazza posts into labels (i.e. categories).

- By **topic**. Labels: "exam", "calculator", "euchre", "image", "recursion", "statistics"

  Example: Posts extracted from `w16_projects_exam.csv`

label	content
exam	will final grades be posted within 72 hours
calculator	can we use the friend class list in stack
euchre	weird problem when i try to compile euchrecpp
image	is it normal for the horses tests to take 10 minutes
recursion	is an empty tree a sorted binary tree
statistics	are we supposed to have a function for summary
...	...

- By **author**. Labels: "instructor", "student"

  Example: Posts extracted from `w14-f15_instructor_student.csv`

label	content
instructor	disclaimer not actually a party just extra OH
student	how can you use valgrind with calccpp
student	could someone explain to me what the this keyword means
...	...

The Piazza datasets are Comma Separated Value (CSV) files. The label for each post is found in the "tag" column, and the content in the "content" column. There may be other columns in the

CSV file; your code should ignore all but the "tag" and "content" columns. **You may assume all Piazza files are formatted correctly, and that post content and labels only contain lowercase characters, numbers, and no punctuation.** We recommend using the `csvstream.h` library (see https://github.com/awdeorio/csvstream for documentation) to read CSV files in your application. The `csvstream.h` file itself is included with the starter code.

Your classifier should not hardcode any labels. Instead, it should use the exact set of labels that appear in the training data.

Appendix B contains code for splitting a string of content into a set of individual words.

We have included several Piazza datasets with the project:

- `train_small.csv` - Made up training data intended for small-scale testing.
- `test_small.csv` - Made up test data intended for small-scale testing.
- `w16_projects_exam.csv` - (Train) Real posts from W16 labeled by topic.
- `sp16_projects_exam.csv` - (Test) Real posts from Sp16 labeled by topic.
- `w14-f15_instructor_student.csv` - (Train) Real posts from four terms labeled by author.
- `w16_instructor_student.csv` - (Test) Real posts from W16 Piazza labeled by author.

For the real datasets, we have indicated which are intended for training vs. testing.

Classifying Piazza Posts with NLP and ML

At a high level, the classifier we'll implement works by assuming a probabilistic model of how Piazza posts are composed, and then finding which label (e.g. our categories of "euchre", "exam", etc.) is the most probable source of a particular post.

All the details of natural language processing (NLP) and machine learning (ML) techniques you need to implement the project are described here. You are welcome to consult other resources, but there are many kinds of classifiers that have subtle differences. The classifier we describe here is a simplified version of a "Multi-Variate Bernoulli Naive Bayes Classifier". If you find other resources, but you're not sure they apply, make sure to check them against this specification.

This document provides a more complete description of the way the classifier works, in case you're interested in the math behind the formulas here.

The Bag of Words Model

We will treat a Piazza post as a "**bag of words**" - each post is simply characterized by which words it includes. The ordering of words is ignored, as are multiple occurrences of the same word.

These two posts would be considered equivalent:

- “the left bower took the trick”
- “took took trick the left bower bower”

Thus, we could imagine the post-generation process as a person sitting down and going through every possible word and deciding which to toss into a bag.

Background: Conditional Probabilities and Notation

We write to denote the probability (a number between 0 and 1) that some event will occur. denotes the probability that event will occur given that we already know event has occurred. For example, . This means that if a Piazza post is about the euchre project, there is a 0.7% chance it will contain the word bower (we should say “at least once”, technically, because of the bag of words model).

Training the Classifier

Before the classifier can make predictions, it needs to be trained on a set of previously labeled Piazza posts (e.g. `train_small.csv` or `w16_projects_exam.csv`). Your application should process each post in the training set, and record the following information:

- The total number of posts in the entire training set.
- The number of unique words in the entire training set. (The **vocabulary size**.)
- For each word , the number of posts in the entire training set that contain .
- For each label , the number of posts with that label.
- For each label and word , the number of posts with label that contain .

Predicting a Label for a New Post

Given a new Piazza post , we must determine the most probable label , based on what the classifier has learned from the training set. A measure of the likelihood of C is the **log-probability score** given the post:

Important: Because we’re using the bag-of-words model, the words w_1, w_2, ..., w_n in this formula are only the unique words in the post, not including duplicates! To ensure consistent results, make sure to add the contributions from each word in alphabetic order.

The classifier should predict whichever label has the highest log-probability score for the post. If multiple labels are tied, predict whichever comes first alphabetically.

is the **log-prior** probability of label and is a reflection of how common it is:

is the **log-likelihood** of a word given a label , which is a measure of how likely it is to see word in posts with label . The regular formula for is:

However, if was never seen in a post with label in the training data, we get a log-likelihood of -∞, which is no good. Instead, use one of these two alternate formulas:

(Use when does not occur in posts labeled but does occur in the training data overall.)

(Use when does not occur anywhere at all in the training set.)

Implementing Your Top-Level Classifier Application

For submission to the autograder, your top-level application code must be entirely contained in a single file, `main.cpp` . However, the structure of your classifier application, including which procedural abstractions and/or ADTs to use for the classifier, is entirely up to you. Make sure your decisions are informed by carefully considering the classifier and top-level application described in this specification.

We **strongly** suggest you make a class to represent the classifier - the private data members for the class should keep track of the classifier parameters learned from the training data, and the public member functions should provide an interface that allows you to train the classifier and make predictions for new piazza posts.

Here is some high-level guidance:

1. First, your application should read posts from a file (e.g. `train_small.csv`) and use them to train the classifier. After training, your classifier abstraction should store the information

mentioned in the "Training the Classifier" section above.

2. Your classifier should be able to compute the log-probability score of a post (i.e. a collection of words) given a particular label. To predict a label for a new post, it should choose the label that gives the highest log-probability score.
3. Read posts from a file (e.g. `test_small.csv`) to use as testing data. For each post, predict a label using your classifier.

Some of these steps have output associated with them. See the "output" section below for the details.

You must also write RMEs and appropriate comments to describe the interfaces for the abstractions you choose (ADTs, classes, functions, etc.). You should also write unit tests to verify each component works on its own.

You are welcome to use any part of the STL in your top-level classifier application. In particular, `std::map` and `std::set` will be useful.

Classifier Application Interface

Here is the usage message for the top-level application:

```
$ ./main.exe
Usage: main.exe TRAIN_FILE TEST_FILE [--debug]
```

The main application always requires files for both training and testing, although the test file may be empty. You may assume all files are in the correct format.

Use the provided small-scale files for initial testing and to check your output formatting:

```
$ ./main.exe train_small.csv test_small.csv
$ ./main.exe train_small.csv test_small.csv --debug
```

Correct output is in `test_small.out.correct` and `test_small_debug.out.correct`. The output format is discussed in detail below.

Error Checking

The program checks that the command line arguments obey the following rules:

- There are 3 or 4 arguments, including the executable name itself (i.e. `argv[0]`).
- The fourth argument (i.e. `argv[3]`), if provided, must be `--debug`.

If any of these are violated, print out the usage message and then quit by returning a non-zero value from `main`. **Do not use the `exit` library function, as this fails to clean up local objects.**

```
cout << "Usage: main.exe TRAIN_FILE TEST_FILE [--debug]" << endl;
```

If any file cannot be opened, print out the following message, where `filename` is the name of the file that could not be opened, and quit by returning a non-zero value from `main`.

```
cout << "Error opening file: " << filename << endl;
```

You do not need to do any error checking for command-line arguments or file I/O other than what is described on this page. However, you must use precisely the error messages given here in order to receive credit. (**Just literally use the code given here to print them.**)

As mentioned earlier, you may assume all Piazza data files are in the correct format.

Output

This section details the output your program should write to `cout`, using the small files mentioned above as an example. Some lines are indented by two spaces. Output only printed when the `--debug` flag is provided is indicated here with "(DEBUG)".

Add this line at the beginning of your main function to set floating point precision:

```
cout.precision(3);
```

First, print information about the training data:

- (DEBUG) Line-by-line, the label and content for each training document.

```
training data:
  label = euchre, content = can the upcard ever be the left bower
  label = euchre, content = when would the dealer ever prefer a card to the upcar
  label = euchre, content = bob played the same card twice is he cheating
  ...
  label = calculator, content = does stack need its own big three
  label = calculator, content = valgrind memory error not sure what it means
```

- The number of training posts.

```
trained on 8 examples
```

- (DEBUG) The vocabulary size (the number of unique words in all training content).

```
vocabulary size = 49
```

- An extra blank line

If the debug option is provided, also print information about the classifier trained on the training posts. Whenever classes or words are listed, they are in alphabetic order.

- (DEBUG) The classes in the training data, and the number of examples for each.

```
classes:
  calculator, 3 examples, log-prior = -0.981
  euchre, 5 examples, log-prior = -0.47
```

- (DEBUG) For each label, and for each word that occurs for that label: The number of posts with that label that contained the word, and the log-likelihood of the word given the label.

```
classifier parameters:
  calculator:assert, count = 1, log-likelihood = -1.1
  calculator:big, count = 1, log-likelihood = -1.1
  ...
  euchre:twice, count = 1, log-likelihood = -1.61
  euchre:upcard, count = 2, log-likelihood = -0.916
  ...
```

- (DEBUG) An extra blank line

Finally, use the classifier to predict classes for each example in the testing data. Print information about the test data as well as these predictions.

- Line-by-line, the "correct" label, the predicted label and its log-probability score, and the content for each test. Insert a blank line after each for readability.

```
test data:
  correct = euchre, predicted = euchre, log-probability score = -13.7
  content = my code segfaults when bob is the dealer

  correct = euchre, predicted = calculator, log-probability score = -12.5
  content = no rational explanation for this bug

  correct = calculator, predicted = calculator, log-probability score = -13.6
  content = countif function in stack class not working
```

- The number of correct predictions and total number of test posts.

```
    performance: 2 / 3 posts predicted correctly
```

The last thing printed should be a newline character. The output for this example can be found in `test_small.out.correct` and `test_small_debug.out.correct`. Use `diff` to compare against these files and check your formatting.

Results

In case you're curious, here's the performance for the large datasets. Not too bad!

`./main.exe w16_projects_exam.csv sp16_projects_exam.csv`	245 / 332
`./main.exe w14-f15_instructor_student.csv w16_instructor_student.csv`	2602 / 2988

Appendix A: Map Example

```
#include <iostream>
#include <string>
#include "Map.h"
using namespace std;

int main () {
  // A map stores two types, key and value
  Map<string, double> words;

  // One way to use a map is like an array
  words["hello"] = 1;

  // Maps store a std::pair type, which "glues" one key to one value.
  // The CS term is Tuple, a fixed-size heterogeneous container.
  pair<string, double> tuple;
  tuple.first = "world";
  tuple.second = 2;
  words.insert(tuple);

  // Here's the C++11 way to insert a pair
  words.insert({"pi", 3.14159});

  // Iterate over map contents using a C++11 range-for loop
```

```
  // This is the equivalent without C++11:
  // for (Map<string, double>::Iterator i=words.begin();
  //      i != words.end(); ++i) {
  for (auto i : words) {
    auto word = i.first; //key
    auto number = i.second; //value
    cout << word << " " << number << "\n";
  }

  // Check if a key is in the map.  find() returns an iterator.
  auto found_it = words.find("pi");
  if (found_it != words.end()) {
    auto word = (*found_it).first; //key
    auto number = (*found_it).second; //value
    cout << "found " << word << " " << number << "\n";
  }

  // When using the [] notation, an element not found is automatically created.
  // If the value type of the map is numeric, it will always be 0 "by default".
  cout << "bleh: " << words["bleh"] << endl;
}
```

Appendix B: Splitting a Whitespace-Delimited String

We've provided two versions that use `istringstream`. They do the same thing, so use whichever you like in your code.

```
// EFFECTS: Returns a set containing the unique "words" in the original
//          string, delimited by whitespace.
set<string> unique_words(const string &str) {
 istringstream source(str);
 set<string> words;
 string word;

 // Read word by word from the stringstream and insert into the set
 while (source >> word) {
   words.insert(word);
 }
 return words;
}
```

```
// EFFECTS: Returns a set containing the unique "words" in the original
//          string, delimited by whitespace.
set<string> unique_words(const string &str) {
```

```
    // Fancy modern C++ and STL way to do it
    istringstream source{str};
    return {istream_iterator<string>{source},
            istream_iterator<string>{}};
}
```