Big Data Analytics Programming
Assignment 1: *Online Spam Filters*

Laurens Devos *laurens.devos@cs.kuleuven.be*

Collaboration policy
Projects are independent: no working together! You must come up with how to solve the problem independently. Do not discuss specifics of how you structure your solution, etc. You cannot share solution ideas, pseudocode, code, reports, etc. You cannot use code that is available online. You cannot look up answers to the problems online. If you are unsure about the policy, ask the professor in charge or the TAs.

1 Online Spam Filters [24pts]

The goal of this assignment is to build and evaluate four spam filter algorithms that can deal with infinite examples and features. The spam filters are supposed to operate in the following online setting: whenever the user receives an e-mail, the filter should classify it as spam or ham (a legitimate e-mail). Then, the user labels the e-mail as spam or ham and the filter should be updated with this information. The (base) features of the spam filter are the n-grams (groups of words) that appear in the e-mail. It is not known beforehand which words will appear, how many words and which ones will prove to be important. The words can even change over time, for example, when the spammers have learned that all their e-mails with "FREE TRY NOW" get blocked, they might switch to "WINNER WINNER WINNER".

You will implement two methods for spam filtering in **C++**: a naive Bayes and a perceptron classifier. In addition, you will also use two techniques to deal with infinite and unknown features: features hashing and a count-min sketch.

The following files are provided:

```
assignment1
└── src
    ├── CMakeLists.txt (CMake configuration file)
    ├── main.cpp (program entry point)
    ├── email.hpp (a representation for a single email)
    ├── base_classifier.hpp (your spam filters subclass this)
    ├── (*) perceptron_feature_hashing.hpp
    ├── (*) perceptron_count_min.hpp
    ├── (*) naive_bayes_feature_hashing.hpp
    ├── (*) naive_bayes_count_min.hpp
    └── (*) metrics.hpp
```

Make modifications to and implement the functions in the files indicated with a `(*)`.

We use CMake to generate build files. In the terminal on Unix systems:

```
mkdir build && cd build
cmake ..
make
```

Table 1: Data Characteristics

Dataset	Ham	Spam	Total
Enron	19.088	32.988	52.076
SpamAssasin	6.954	3.797	10.751
Trec2005	39.399	52.790	92.189
Trec2006	12.910	24.914	37.822
Trec2007	25.220	50.199	75.419
Total	103.571	164.686	268.257

The above will create a build folder called `build`, move into this folder, and initiate CMake. CMake will generate the necessary files for `make` to compile the program.

To run the program, execute the resulting binary:

```
build/src/bdap_assignment1 <window-size> <ngram> <output-file>
```

The arguments are:

- The metric value is reported at an interval of `window-size` emails.
- The maximum size of the n-grams is given by `ngram`.
- The recorded metric values are written to `output-file`. This file can then be read by a (Python) script to generate a learning curve. An example script is given in `plot_results.py`.

The data used for this assignment is the concatenation of five real world datasets in the given order: Enron[1], SpamAssasin[2], Trec2005, Trec2006, and Trec2007[3]. The datasets are summarized in Table 1. The emails are provided in a pre-processed format: headers are removed and the subject line and body are concatenated, stopwords and HTML tags are removed, and words are stemmed.

The data can be found in `/cw/bdap/assignment1` on the departemental computers.

1.1 Classifiers

1.1.1 Naive Bayes

Naive Bayes is a simple, standard, method to train a classifier. It was one of the original methods for spam filtering [SDHH98].

In the traditional formulation of Naive Bayes with binary features, the feature being 'true' or 'false' are considered equally informative. In our case, the features are the words that (do not) appear in the e-mail. Naively, the presence of a word in an e-mail is thus considered as informative as the absence of that word:

$$P(S|\text{text}) = \frac{P(S)\,\Pi_{w\in\text{text}}P(w|S)\Pi_{w\notin\text{text}}(1-P(w|S))}{P(\text{text})}$$

However with a large vocabulary, the absence of a specific word is not informative. Intuitively, an e-mail only contains a subset of the words that could have been used to deliver the message. Therefore, a word that does not occur in an e-mail is considered unobserved, i.e.: it may or may not be in the e-mail. Under this assumption, the Naive Bayes formula simplifies to:

$$P(S|\text{text}) = \frac{P(S)\,\Pi_{w\in\text{text}}P(w|S)}{P(\text{text})}$$

[1] `https://www.cs.cmu.edu/~./enron/`
[2] `https://spamassassin.apache.org/old/publiccorpus/`
[3] `https://plg.uwaterloo.ca/~gvcormac/spam/`

The calculations for Naive Bayes should be done in the logarithmic domain, otherwise they might suffer from underflows. Hint: sums should be calculated as follows (see `std::log1p`):

$$\log(a+b) = \log a + \log(1 + \exp(\log b - \log a))$$

The probabilities in the formula are estimated from counts in the data. However, we must avoid zero-counts. Naively, a zero-count corresponds to a zero probability. If a spam e-mail comes in with the words "free", "viagra" and "roses" in it and "roses" never occurred in spam before, then this mail would be falsely classified as ham. To prevent such situations, methods such as Laplace estimates can be used where '1' is added to every count. This is equivalent to initializing the classifier with a spam e-mail that has all possible words and a ham e-mail that has all possible words.

1.1.2 Perceptron

The perceptron classifier dates from 1958 and is one of the first artificial neural networks, laying the foundations of the current deep neural networks. For this assignment, you will implement learning with the delta rule and stochastic gradient descent.

Attention! The perceptron expects the classes to be 1 and -1, while the `Email` class implementation provides them as boolean 1 and 0. Keep this in mind when writing the perceptron code.

1.2 Dealing with Infinite Features

In this assignment you will use two ways to deal with the infinite and beforehand-unknown vocabulary used in the e-mails: feature hashing and the count-min sketch.

1.2.1 Feature Hashing

E-mails can be represented as word vectors, where every entry of the array represents a word which has value '1' if this word appears in the e-mail and '0' otherwise. With an infinite stream of e-mails, the size of this array is unknown and potentially infinite. To deal with this, feature hashing maps the word vector to an array of finite size, using a hash function. The mapped array of finite size is then used as the feature vector in the learning algorithm.

Feature hashing can be interpreted as a sketching method for classifiers with infinite features: For Naive Bayes, it approximates the count of each word with an overestimate, namely the count of all the words that hash to the same value. For the perceptron classifier, it approximates the weights for each word by the weight for all the words that hash to the same value.

In your implementation, you can use the `hash(word, seed)` function defined in `BaseClf` for hashing n-grams. However, the hash function maps n-grams to values of `size_t`. You should adjust the range to the desired range which is given as an input parameter ($2^{\texttt{log\_num\_buckets}}$).

1.2.2 Count-Min Sketch

Count-min sketch is a sketching algorithm for keeping approximate counts, or numbers in general [CM05]. It is related to feature hashing, but reduces the estimation error by using multiple hashing functions. The name "count-min" comes from the non-negative case, where the number, like a count, can only be increased. In this case the count of the feature value can only be an overestimate. By taking the minimal count accross the counts of the different hashing values for this feature, the estimation error is minimized. The general case, where the numbers can be reduced, like the weights of the perceptron, uses the median instead of the minimum.

Like for feature hashing, you can use the `hash` function in `BaseClf`, but you should adjust the hash range.

1.3 Classifier Evaluation

Because the spam filters are online learners, the algorithm should be evaluated periodically. In contrast to using a fixed test set, a classifier is evaluated against examples that will subsequently be used to update the model. The provided skeleton implementation in `main.c` uses a reporting period '`window-size`': every '`window-size`' examples, the classifier is evaluated on the next '`window-size`' examples, before adding them to the model and repeating the process.

The provided code evaluates the classifiers using accuracy (#correctly classified/#total classified). However this is not the only way to evaluate the performance of a classifier. **Implement at least two additional evaluation metrics** and discuss why these evaluation metrics are fit for evaluating a spam filter.

2 Submitting your Code

The project is due **December 16, 2022 at 17h**.

Only the following files are graded:

```
bdap-assignment1-<r-number>.zip
├── code
│   ├── perceptron_feature_hashing.hpp
│   ├── perceptron_count_min.hpp
│   ├── naive_bayes_feature_hashing.hpp
│   ├── naive_bayes_count_min.hpp
│   └── metrics.hpp
└── report.pdf
```

The C++ files should contain the complete source code for the spam filters. The report with the learning curve must be submitted in PDF. Do not upload any data or output files.

Any changes made to the other files **are not considered** during grading; we will use our own test files that expect the same interface as in the code template.

Specifically, your code should compile and run successfully on the departmental computers using the unmodified versions of the files `main.cpp`, `email.hpp`, `base_classifier.hpp`, and `CMakeLists.txt`. Make sure to follow these guidelines:

- Each of your classifiers implements an `update_` and a `predict_` method:

```
void update_(const Email& email);
void predict_(const Email& email) const;
```

- Each metric implements two `evaluate` methods and a `get_score` method:

```
template <typename Clf>
void evaluate(const Clf& clf, const std::vector<Email>& emails);

template <typename Clf>
void evaluate(const Clf& clf, const Email& email);

double get_score() const;
```

You are allowed to use the C++ standard library. You may not use any other C++ libraries.

3 Report [6 pts]

Part of this course's goal is to gain insights into the interplay between large amounts of data and algorithms. To this end, you should conduct several experiments to explore issues such as efficiency,

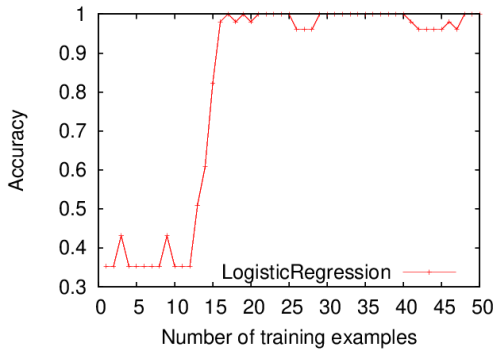

Figure 1: Example of a learning curve on zoo.

scalability, effects of parameter values, etc., to help gain insight into how the algorithms you implemented work. Please focus on posing an experimental question (e.g., What is the effect of varying parameter X), designing an experiment to test that question, and then providing a summary of the finding (e.g., increasing the value of parameter X causes...). Any valuable insight on your work will be graded accordingly. You should write a small report of at most two pages (excluding Tables or Figures) explaining your experiments and findings.

One experiment that you must perform and include in your report is generating a *learning curve*, which visualizes how a classifier's accuracy (or other evaluation metric) varies as a function of the number of training examples. Figure 1 gives an example of a learning curvre for a logistic regression classifier on the UCI zoo dataset. The learning curve in your report should show the performance of the Naive Bayes with feature hashing, Naive Bayes with count-min sketch, perceptron with feature hashing, and perceptron with count-min sketch. Failing to use the full dataset to generate the learning curve will incur a substantial penalty.

The report should also include a discussion of the different evaluation metrics that you used and the different insights that they provide.

If you added further improvements to your spam filter, you should include at most half a page extra describing the changes and how they affect the performance.

In addition, the report can include a maximum of half an extra page describing any problems (that is, bugs) in your code and what may have caused them as well as any special points about your implementation.

References

[CM05] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[SDHH98] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A bayesian approach to filtering junk e-mail. In *Learning for Text Categorization: Papers from the 1998 workshop*, volume 62, pages 98–105, 1998.