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

Maaike Van Roy
maaike.vanroy@kuleuven.be

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.

Beyond what is provided, you may not use any other code that is available online. The use of language models such as ChatGPT or any other AI-based systems for completing this programming assignment is strictly prohibited. If you are unsure about the policy, ask the professor in charge or the TAs.

1 Online Spam Filters [14pts]

The goal of this assignment is to build and evaluate two 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 the naive Bayes method for spam filtering in **C++**. In addition, you will also use two techniques to deal with infinite and unknown features: feature hashing and count-min sketch.

The following files are provided:

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

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. CMake is cross-platform, and should work on Mac and Windows as well. Only the `make` is going to be different (e.g., it will likely use the Visual Studio Compiler on Windows). You can always use the departmental computers over SSH if you are having troubles on your personal computer.

To run the program, execute the resulting binary:

```
build/src/bdap_assignment1
```

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 departmental computers.

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

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.

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 Dealing with Infinite Features

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.

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.

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 ($2^{\texttt{log\_num\_buckets}}$).

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.

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

1.2 Evaluation Metrics

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.cpp` 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.

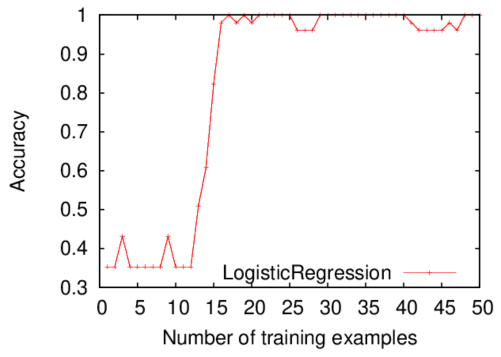

Figure 1: Example of a learning curve on zoo.

2 Experimenting with the Classifiers [6pts]

Rigorous experimentation is a crucial aspect of working with large datasets. Moreover, an important skill in both industry and research is the ability analyze and interpret model performance in a principled way. The goal of experimentation is multifaceted. First, it helps show that your implementation works. Second, there multiple different ways that all learning pipelines can be configured and performance can vary substantially based on the configuration. Third, one makes choices when designing and implementing algorithms, and experimentation can help justify these decisions.

Experimentation is therefore also a central aspect of this assignment (and the course in general). You must perform two types of experiments in this assignment.

First, you must generate a *learning curve*, which visualizes how a classifier's accuracy varies as a function of the number of training examples. Figure 1 gives an example of a learning curve for a logistic regression classifier on the UCI zoo dataset. You should perform this experiment for different `window-sizes`. This must be done on the complete dataset.

Second, you must explore how the different design choices and configuration options (e.g., hyperparameter choices) affect the performance of the Naive Bayes classifiers that you have developed. You should be able to show that you can design such experiments and perform them in a systematically and sound way. Hence, you should come up with your own experimental setups (i.e., select the design and configuration options to evaluate, define how performance will be measured, and decide on any meaningful comparisons). Moreover, you must be able to present and interpret the results correctly. Note that this does NOT necessarily require generating a learning curve for each experiment.

Your experiments must be fully reproducible. Use `main.cpp` to run the experiments and write the results to a file. Create a Python script (`plot.py`) to collect and visualize them. The Python script must at least include a function to visualize the learning curve, and additionally, in case the results of your self-designed experiments need visualization, any functions needed to do this.

Because this is a class about big data, running experiments could take a long time. You can do this easily on the departmental machines: it takes several minutes to start the experiments, and then you check back later to see the results.

2.1 Report

You must write a concise report of at most two pages, including tables and figures, in which you present and discuss the results of your experiments. The report should also justify your choice of evaluation metrics. **Remember to include your name and student number in the report!**

In summary, the report must include:

- The generated learning curve showing the accuracy for Naive Bayes with feature hashing and Naive Bayes with count-min sketch, using different window-sizes. Failing to use the full dataset will incur a substantial penalty.

- For each experiment you came up with, explicitly state the experimental question you aimed to answer, clearly describe how you conducted the experiment, and include a brief interpretation and an appropriate visualization of its results.
- A brief justification of 1 sentence per metric stating why it is appropriate for evaluating a spam filter.

3 Submitting

Deadline The project is due **Friday, December 12, 2025 at 17h**. Upload your submission to Toledo.

Deliverables You must upload an archive (**.zip** or **.tar.gz**) containing the report (as **PDF**) and the source files, according to the following file hierarchy:

```
bdap-assignment1-<r-number>.zip
 └─ src
 │   ├─ naive_bayes_feature_hashing.hpp
 │   ├─ naive_bayes_count_min.hpp
 │   ├─ metric.hpp
 │   ├─ main.cpp
 │   ├─ plot.py
 │   └─ script.sh
 └─ report.pdf
```

To run your code, you must create a script, `script.sh`, which includes the commands to (1) compile your project, (2) run your project, and (3) run the Python script that visualizes the results. The C++ files should contain the complete source code for the spam filters. The report must be submitted in PDF, for which you **must** use the provided LaTeX template on Toledo.

Please follow the outlined submission format. Do not upload any additional template code files, do not add new source files (i.e., implement everything in the provided files), and do not upload data or output files (e.g., other files generated by LaTeX that are not the PDF, build files from CMake, etc.). Points will be deducted if you fail to follow this guideline.

This also means that 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. Do not change the constructors of the classifiers and code marked as "DO NOT CHANGE".

Specifically, your code should compile and run successfully on the departmental computers using the unmodified versions of the files `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.

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.