Report

Authors

Emilio Cecchini & **Lorenzo Palloni**

Table of Contents

What we implemented

We implemented **yasa** (Yet Another Sentiment Analyser), a containerized command line tool that make easy the application of Sentiment Analysis. It allows to train an underneath model of sentiment analysis with text files labeled as positives or negatives. After the training the model is able to predict the label of an unseen text file into positive/negative.

The management of this model is done trough a command line interface (CLI). The model can be trained giving folder containing positives or negatives text using the command
`$ docker yasa train --positives <positives-folder> --negatives <negatives-folder>`

After trained, the model is able to classify a text file using the command
`$ docker yasa classify --file <file-to-classify>`

Applied techniques and frameworks

The software is implemented in C++ programming language and in order to make it reproducible and portable we use docker. To store all the information that the underneath model needs, we use SQLite that is a database management system (DBMS) based on a relational model.

- **Test ->** Google Test
- **Mock ->** Google Mock
- **Code Covarage ->** Gcov
- **Code Quality ->** LGTM
- **Linter + Formatter ->** Clang-tidy
- **Build Automation ->** CMake + Docker
- **Continuous Integration->** Travis CI
- **VCS ->** GitHub
- **IDE ->** Eclipse
- **DBMS ->** SQLite

Design and implementation choices

The main calls ArgumentParser class that parses the input arguments given through the CLI. Then there are three possibilities:

- train;
- classify;
- display usage message.

The training is handled by the Trainer class that:

i. reads all the text files in the given directories with TextFileReader class;
ii. tokenizes the words previously found with the extractWords function (in preprocessing source code file);
iii. updates the underneath SQLite database using the SqliteDictionary class.

We wrapped the C-written API of SQLite in the class SqliteHandler to make easier the usage. The classifying is handled by a Classifier class that:

i. reads the given text file with TextFileReader;
ii. tokenizes the words previously found with the extractWords function (in preprocessing source code file);
iii. classifies the text querying the database using SqliteDictionary class.

Usage messages to displayed are managed by the ArgumentParser class.

Possible problems encountered and how you solve them

Management of SQLite database was one of the biggest challenges because its implementation in C language did not fit well with our other C++ classes. We implemented a C++ wrapper in the SqliteHandler class to encapsulate the functionalities that we needed to solve this problem.

We initially missed the concept of isolation for unit-tests: we understood this while a new unit-test was running and another unit-test regarding another SUT did not pass. Then Google Mock came into rescue!

- Google Mock requires a virtual implementation of the class to mock. Both original and mocked classes will inherit from their virtual base class. For example, in our case the mocked class MockDictionary and the original class SqliteDictionary both inherit from the interface Dictionary.

Descriptions of the development (and testing) of the most interesting parts

Initially the subject of our project was a preprocessing tool for natural language processing. We changed almost completely the project due to give more sense to the application of a database and in order to have a more concrete goal.

One of the most difficult parts to test was the main because it has a lot of logic branches. We delegates much of the responsabilities of the main to the ArgumentParser class.