# Report

## Authors

**Emilio Cecchini** **Lorenzo Palloni**

## Table of Contents



## What we have implemented

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

The management of this model is done through a command line interface (CLI). The model can be trained by feeding it with a folder containing positives text and/or folder containing negatives text. The following command is an example of usage:

```
$ Docker run \
    --mount type=bind,source=<path-to-yasa.db>,target=/yasa.db \
    --mount type=bind,source=<path-to-pos-dir>,target=/<pos-dir>,readonly \
    --mount type=bind,source=<path-to-neg-dir>,target=/<neg-dir>,readonly \
    --rm --tty yasa train -p <pos-dir> -n <neg-dir>
```

After trained, the model is able to classify a text file using the following command:

```
$ Docker run \
    --mount type=bind,source=<path-to-yasa.db>,target=/yasa.db \
    --mount type=bind,source=<path-to-file-to-classify>,target=/<file-to-classify>,readonly \
    --rm --tty yasa classify -f <file-to-classify>
```

The `yasa.db` file must exists (you can create it with `$ touch yasa.db`), moreover files and directories used must be bound to the container, which can be achieved with Docker volumes or Docker bind mounts. The training phase will update this file `yasa.db`, which can be stored and used later for the classification phase or for another training.

For example in order to train the underneath model with all the text files in the directories `example/pos` and `example/neg`, you can use the following command (it can take about 5 minutes):

```
$ Docker run \
    --mount type=bind,source="$PWD"/yasa.db,target=/yasa.db \
    --mount type=bind,source="$PWD"/example/pos,target=/pos,readonly \
    --mount type=bind,source="$PWD"/example/neg,target=/neg,readonly \
    --rm --tty yasa train -p pos -n neg
```

If you want to classify e.g. the `example/neg.txt` file with the model trained just before, you can use the following command:

```
$ Docker run \
    --mount type=bind,source="$PWD"/yasa.db,target=/yasa.db \
    --mount type=bind,source="$PWD"/example/neg.txt,target=/neg.txt,readonly \
    --rm --tty yasa classify -f neg.txt
```

## Applied techniques and frameworks

The software is implemented in C++ 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 + LCOV + CodeCov
- **Code Quality ->** LGTM
- **Linter ->** clang-tidy
- **Formatter ->** clang-format
- **Build Automation ->** CMake + Docker
- **Continuous Integration ->** Travis CI
- **VCS ->** GitHub
- **IDE ->** Eclipse
- **DBMS ->** SQLite

# Design and implementation choices

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

- train;
- classify;
- display version;
- 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.cc` source code file);
iii. updates the underneath SQLite database using the `SqliteDictionary` class.

We wrapped the C-written API of SQLite with the class `SqliteHandler` to make the usage easier.
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.cc` source code file);
iii. classifies the text querying the database using `SqliteDictionary` class.

Usage messages to displayed are managed by the `ArgumentParser` class.

# Some of the problems encountered and how we have solved them

The 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 have solved this problem encapsulating the necessary functionality of the API wrapping it in our `SqliteHandler` C++ class.

We initially missed the concept of isolation for unit-tests: we realized it when a bug in a component caused tests of other components to fail. Then Google Mock came into rescue!

# 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 delegate much of the responsibilities of the main to the `ArgumentParser` class.

At some point during the development of this project, there was a folder `tests/resources` that contained examples of text files. It allowed us to test `Trainer` and `Classifier` classes, but we understood that with a mock object on the text files we would achieve a more isolated unit-tests for both `Trainer` and `Classifier` classes.

The shell script `run_example.sh` can be considered as an **end-to-end test**, because it executes both the training and classifying phases with real files and with a real database. Since the training process can take time, it is executed in Travis CI.

Test pyramid:

```
               /\
              /  \
             /    \
            /      \
           /        \
          /    1     \
         / end-to-end \
        /==============\
       /                \
      /        16        \
     / integration tests  \
    /======================\
   /                        \
  /            50            \
 /         unit tests         \
/==============================\
```

## Explanation of some of the tools used

### LGTM

LGTM is a code analysis platform that automatically checks your code for real Common Vulnerabilities and Exposures (CVEs). By combining deep semantic code search with data science insights, LGTM ranks the most relevant results to show you only the alerts that matter.

LGTM processes the contents of software development projects whose source code is stored in public Git repositories hosted on such as GitHub.com and among the currently supported languages there is C++.

If the project you're interested in isn't already on LGTM, simply log in and add it directly from your Project lists page. In this way you enable automated code review for your project's pull requests.

### GCOV / LCOV

In order to get code coverage from C++ programs you have to add the compile flag `--coverage`. This flag allows GCOV to create `.gcda` e `.gcno` files after the execution of the program. LCOV is an extension of GCOV which makes easy the creation of human-friendly reports.

Here are the commands that we run to generate the coverage report after the execution of the unit tests.

To generate a report file called `coverage.info` related to the whole project:

```
$ lcov --capture --directory . --output-file coverage.info
```

This command includes in the report all the source files covered by the previous execution including the tests and the standard library. The following command excludes the source files that we are not interested to:

```
$ lcov --remove coverage.info \
    '/usr/include/*' \
    '/usr/local/include/*' \
    ${YASA_ROOT_DIR}'/tests/*' \
    --quiet --output-file coverage.info
```

Finally, to display the coverage results:

```
$ lcov --list coverage.info
```

### CodeCov

Codecov provides an online tool to visualize automatically code coverage report(s) easily. Codecov delivers or "injects" coverage metrics directly into the modern workflow to promote more code coverage, especially in pull requests where new features and bug fixes commonly occur.

We use CodeCov following these simple steps:

- sign up on codecov.io and link our GitHub.com account;
- once linked, Codecov will automatically sync all the repositories to which you have access;
- select our repository at https://github.com/ceccoemi/yasa;

- run our normal test suite to generate code coverage reports through Travis CI;
- use in `.travis.yml` the bash uploader to upload our coverage report(s) to Codecov.

In `.travis.yml` the bash uploader is the following script command:

```
wget -S -O - https://codecov.io/bash | bash
```

## Google Test

Google Test is a *C++ Testing Framework* developed and maintained by Google. In the last years it became the standard for testing C++ applications. Since Google Test is based on the popular xUnit architecture, it is very straightforward to use it if you are used to test software with JUnit.

### Installation

To write a test program using Google Test, you need to compile Google Test into a library and link your test with it. In our application, since it is built in a Docker environment with Ubuntu 18.04, you have to install it with

```
$ sudo apt-get install libgtest-dev
```

After this is done, you have to manually compile it

```
$ cd /usr/src/gtest
$ cd build-aux
$ cmake -j$(nproc) ..
$ make -j$(nproc)
$ sudo make install
```

### Compile code with Google Test

Now it must be linked to the main function of the tests. CMake provides the handy function `find_package(GTest REQUIRED)` to locate the Google Test library in the system. If it is found then a variable called `GTEST_BOTH_LIBRARIES` containing `libgtest` and `libgtest-main` is created. Finally that variable `GTEST_BOTH_LIBRARIES` can be linked to the executable with `target_link_libraries`. In our `CMakeLists.txt` located in the test directory we have

```
find_package(GTest REQUIRED)

...

file(GLOB sourceFiles *.cc)
add_executable(RunAllTests ${sourceFiles})
target_link_libraries(
    RunAllTests ${GTEST_BOTH_LIBRARIES} ${TEST_LIB}
)
```

### Usage

When using Google Test, you start by writing assertions, which are statements that check whether a condition is true. An assertion’s result can be success, nonfatal failure, or fatal failure. If a fatal failure occurs, it aborts the current function; otherwise the program continues normally.

Tests use assertions to verify the tested code’s behavior. If a test crashes or has a failed assertion, then it fails; otherwise it succeeds.

Google Test assertions are macros that resemble function calls. You test a class or function by making assertions about its behavior. When an assertion fails, Google Test prints the assertion’s source file and line number location, along with a failure message. You may also supply a custom failure message which will be appended to Google Test’s message.

The assertions come in pairs that test the same thing but have different effects on the current function. `ASSERT_*` versions generate fatal failures when they fail, and abort the current function. `EXPECT_*` versions generate nonfatal failures, which don’t abort the current function. In our project we only use `ASSERT_*` functions.

Here an example extracted from `preprocessingTest`:

```
#include <gtest/gtest.h>
```

```
#include <preprocessing.h>
#include <string>
#include <vector>

TEST(PreprocessingTest, emptyTest) {
  std::vector<std::string> actual = extractWords("");
  std::vector<std::string> expected{};
  ASSERT_EQ(actual, expected);
}

TEST(PreprocessingTest, oneLetterText) {
  std::vector<std::string> actual = extractWords("O");
  std::vector<std::string> expected{"O"};
  ASSERT_EQ(actual, expected);
}

TEST(PreprocessingTest, extractWordsWithSpaces) {
  std::vector<std::string> actual =
      extractWords("  yasa is   the\nbest\r\nsoftware\tin\vthe\fworld   ");
  std::vector<std::string> expected = {"yasa", "best", "software", "world"};
  ASSERT_EQ(actual, expected);
}
```

Each test function is identified by

```
TEST(test_case_name, test_name) {
 ... test body ...
}
```

As you can see, the usage is very similar to JUnit. You can also create a *Test Fixture* to use the same data configuration for multiple tests, for example in `SqliteHandlerTest`:

```
class SqliteHandlerDbTest : public ::testing::Test {
 protected:
  virtual void SetUp() {
    dbFileName = fs::temp_directory_path() / fs::path("test.db");
  }

  virtual void TearDown() { fs::remove(dbFileName); }

  std::string dbFileName;
};

TEST_F(SqliteHandlerDbTest, throwRuntimeErrorWhenFilenameIsInvalid) {
  ASSERT_THROW(SqliteHandler("/"), std::runtime_error);
}

TEST_F(SqliteHandlerDbTest, createDbInAFile) {
  ASSERT_FALSE(fs::is_regular_file(dbFileName));
  SqliteHandler db(dbFileName);
  ASSERT_TRUE(fs::is_regular_file(dbFileName));
}

TEST_F(SqliteHandlerDbTest, operateOnTheSameFile) {
  SqliteHandler db1(dbFileName);
  SqliteHandler db2(dbFileName);
  db1.query(
      "CREATE TABLE testTable("
      "id int PRIMARY KEY NOT NULL,"
      "name VARCHAR(30) NOT NULL);");
  QueryResult result =
      db2.query("SELECT name FROM sqlite_master WHERE type = 'table';");
  ASSERT_EQ(result["name"], std::vector<std::string>{"testTable"});
}
```

Here the fixture class is `SqliteHandlerDbTest`, which stores the database file name. A temporary file is created in `SetUp` before the execution of each test and it is destroyed in `TearDown` when each test ends.

## Google Mock

[gMock](#) is a C++ library for creating mock classes and using them. gMock is bundled with Google Test and don't need to download it if you have Google Test. However, you must link gMock through CMake among the tests to use it.

### Usage

First of all we have to define an interface and we took a piece of the `Dictionary` interface from our project as an example:

```
#pragma once

#include <string>

class Dictionary {
 public:
  virtual ~Dictionary() {}
  virtual int positivesCount(const std::string& word);
  virtual int negativesCount(const std::string& word);
  ...
};
```

Note: the destructor of `Dictionary` must be virtual, as is the case for all classes you intend to inherit from. Otherwise the destructor of the derived class will not be called when you delete an object through a base pointer, and you'll get corrupted program states like memory leaks.

Using the `Dictionary` interface, here are the simple steps you need to follow to build a `MockDictionary`:

- derive `MockDictionary` from `Dictionary`;
- take a virtual function from Dictionary;
- in the `public:` section of the child class, write `MOCK_METHODn()`;
- `MOCK_METHODn()`, has two parameters described in the following two steps:
  - first argument: the identifier of the method to mock;
  - second argument: `<return-type>(<argument-type> <argument-identifier>, ...)`;
- change `n` in `MOCK_METHODn()` based on the number of arguments of the function to mock;
- repeat until all virtual functions you want to mock are done.

After the process, you should have something like:

```
#include "gmock/gmock.h"
#include "Dictionary.h"
#include <string>

class MockDictionary : public Dictionary {
  public:
    MOCK_METHOD1(positivesCount, int(const std::string& word));
    MOCK_METHOD1(negativesCount, int(const std::string& word));
    ...
};
```

Once you have a mock class, using it is easy. The typical work flow is:

i. import the gMock names from the testing namespace such that you can use them unqualified;
ii. create some mock objects;
iii. specify your expectations on them;
iv. exercise some code that uses the mocks;
v. when a mock is destructed, gMock will automatically check whether all expectations on it have been satisfied.

Here's an example:

```
#include "MockDictionary.h"
#include "Classifier.h"
#include "gmock/gmock.h"
#include "gtest/gtest.h"
#include <vector>
#include <string>

using ::testing::_;  // #1

TEST(Classifier, ClassifiyWithNoWords) {
  MockDictionary dictionary;  // #2
  EXPECT_CALL(dictionary, positivesCount(_)).Times(0);  // #3
  EXPECT_CALL(dictionary, negativesCount(_)).Times(0);  // #3
```

```
  Classifier classifier(&dictionary);
  std::vector<std::string> words{};
  classifier.classify(words);  // #4
}  // #5
```