# Report

## Authors

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

## Table of Contents

i. What we have implemented
ii. Applied techniques and frameworks
iii. Design and implementation choices
iv. Some of the problems encountered and how we have solved them
v. Descriptions of the development (and testing) of the most interesting parts
vi. Explanation of some of the tools used

## What we have implemented

We implemented **yasa** (Yet Another Sentiment Analyser), a containerized command line tool that make 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 feeding it on 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, moreover files and directories used must be bound to the container. This can be achieved with docker volumes or docker bind mounts.

For example in order to train the underneath model with all the text files in the directories `"$PWD"/example/pos` and `"$PWD"/example/neg`, you can use the following command:

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

If you want to classify e.g. the `"$PWD"/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,target=/example,readonly \
    --rm --tty yasa classify -f example/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 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 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.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: when a bug in a component caused tests of other components to fail. 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 `Dictionary` interface.

# 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.

# 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.

## CodeCov

Codecov provides an online tool to visualize automatically code coverage report(s) easly. 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 library (sometimes we also call it a "framework" to make it sound cool) for creating mock classes and using them. It does to C++ what jMock/EasyMock does to Java (well, more or less). gMock is bundled with googletest.

## 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 that 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.)

Your program will normally use a real implementation of this interface. In tests, you can use a mock implementation instead. This allows you to easily check what drawing primitives your program is calling, with what arguments, and in which order. Tests written this way are much more robust (they won't break because your new machine does anti-aliasing differently), easier to read and maintain, and run much, much faster.

Using the Dictionary interface, here are the simple steps you need to follow:

- derive `MockDictionary` from `Dictionary`;
- take a virtual function from `Dictionary`;
- in the `public:` section of the child class, write `MOCK_METHOD()`;
- take the function signature, cut-and-paste it into the macro, and add two commas:
  - one between the return type and the name,
  - another between the name and the argument list;
- if you're mocking a const method, add a 4th parameter containing `(const)` (the parentheses are required);
- 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_METHOD(int, positivesCount, (const std::string& word), (override));
    MOCK_METHOD(int, negativesCount, (const std::string& word), (override));
    ...
};
```

Once you have a mock class, using it is easy. The typical work flow is: 1. import the gMock names from the testing namespace such that you can use them unqualified; 2. create some mock objects; 3. specify your expectations on them; 4. exercise some code that uses the mocks; 5. when a mock is destructed, gMock will automatically check whether all expectations on it have been satisfied.

Here's an example:

```
#include "path/to/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);  // #4
  std::vector<std::string> words{};
  classifier.classify(words);  // #5
}
```

The key to using a mock object successfully is to set the right expectations on it. If you set the expectations too strict, your test will fail as the result of unrelated changes. If you set them too loose, bugs can slip through. You want to do it just right such that your test can catch exactly the kind of bugs you intend it to catch. gMock provides the necessary means for you to do it "just right".

---