SentiTomo

As part of our Master Team Project named "Topic Monitoring in the Pharmaceutical Industry" we wanted to develop an application to incorporate our findings and different Machine Learning scripts to show how a production environment can look like. The name of this application is "SentiTomo", a combination of the words "Sentiment Analysis" and "Topic Monitoring".

Idea

The basic idea of accomplish such an task was to create a client-server architecture. The server part will be responsible for crawling social media data from Twitter and Facebook, as well as executing different machine learning tasks, such as sentiment analysis (classification), topic detection and trend detection (clustering). Besides this, the results of those tasks are saved in an database to be available through an Application Programming Interface (API). This ensures that the results are persisted over time and can be analyzed.

Problem statement and way to go

Throughout our work on this project we tried out variuous algorithms for machine learning (ML) and implemented them in different programming languages. After filtering all good performing algorithms we ended up with three programming languages: R, Python and Java. The resulting problem now was to find a common platform for the server to work with those languages. The inital attempt to solve this issue was to choose one those language as the primary one and let it work with the others. But this had one drawback. If we would have chosen one of the ML languages as the primary one we would have not been able to replace this language anymore in favor of another one in which some models may perform better. This would lead to an abandoned machine learning language just serving as a wrapper language. So we were searching for a language which not perform any machine learning but is capable of working with all the existing languages. This ensures that the different languages, and machine learning scripts can easily be switched out or updated without influencing other parts of the application because those will always interact with the wrapper language which copes with the different languages.
Because of the rising popularity and the possibility to be run on server and client side we chose JavaScript as our wrapper language. With the open-source and community driven framework called 'Node.js' it is possible to create powerful server and client applications. Furthermore it is possible to easily spawn child processes, invoking other programming languages, and communicate with them. This made JavaScript our way to go wrapper language.
Based on this, we want to provide you an overview about the main technologies

which we used to built up the application, how the application is structured and how it eventually can be used in a production environment.

Fundamental technologies

To give you an idea which technologies were used throughout the development process of SentiTomo this chapter gives an overview about the most prominent ones. We tried to only use frameworks which experience a high support from the developer community as well as developed by reliable sources. This section should not give an exhaustive explanation of all the used framework. Much more it should give a hint on why we choosed the different solutions for our application.

Node.js (Server)

Node.js is an open-source, cross-platform framework written in C, C++ and JavaScript, which makes it possible to run JavaScript code on the server-side. The initial release was on May the 27th, 2009 and it was written by Ryan Dahl. Primarily it was built because the most common web server at this time, Apache HTTP Server, had troubles with a lot of concurrent connections and normally used blocking code executions which led to poor server performance.
The idea behind Node is to utilize a simplified event-driven programming paradigm where the program flow is determined by events (user clicks, messages from other methods etc.) to let so called callback functions take care of the result of method calls. With this structure the main thread of a Node.js application is not blocked by method executions. Normally a Node based application is only running on one thread, but with non-blocking method calls it will never gets stucked at one point. This makes it easy to build highly scalable applications without the need of mulitple threads, which could result into poor performance. But if it is needed Node can also spawn different threads and is not limited to only one.
Hand in hand to Node comes a package manager called `npm` which stands for `Node Packaging Manager`. It is used to install, update and remove third party Node.js programs which are listed in the npm registry. npm enables developers to easily share and distribute Node.js code, so it can be used in other projects. All installed dependencies are listed inside a file called `package.json` which is mandatory for every Node based project. It contains all neccessary information about the different packages and their version numbers. These are installed inside a folder called `node_modules`, which is accessed at runtime to load the different dependencies of a Node application.

Yarn (Server and Client)

Yarn is an additional Node package manager built by Facebook based on npm, which improves npm in some important parts. One of the biggest flaws of `npm` is that it stores the different packges inside the `node_modules` in a non deterministic way. That means that the order of packages inside `node_modules` can differ from user to user depending on the order of installation. This can lead to bugs like 'Works on my machine' arise very quickly which makes debugging and hunting bugs down very frustrating. Also the the actual dependency tree can differ from the `node_modules` directory because duplicate dependencies are merged by Node itself. All these issues are resolved with Yarn by installing dependencies in a deterministic way and organizing duplicate ones in a better manner. Every projects that uses Yarn file called 'yarn.lock' is created. This file is used to execute the installation processes on different machines in the exact same way which makes bugs like 'Works on my machine' very unlikely.

Express.js (Server)

Express.js is a JavaScript framework built with Node.js and today the de-facto standard to build a web-server application with Node. It is an MIT licensed, open-source published directly from the developers of Node.js.
In its fundamental form it is very lightweight and only offers the minimum functionalities to build a web-server. But it offers great ways to integrate all kind of plugins, like logging, templating engines, server side rendering, and even more. This makes it very versatile and the number one solution for developing a web server with Node.js. Therefore we chose it as the framework to build our server architecture.

h20 (Server)

h20 is a software platform for big-data analysis and machine learning tasks. It can be run in Java, R and Python and offers APIs for building and using models for detection patterns in data. Inside SentiTomo it is used in the R versions and serves as a platform for running some machine learning tasks on it. Primiarly these are sentiment analysis tasks.

GraphQL (Server API)

Because we wanted to integrate an Application Programming Interface for easy data sharing between server and client we searched for a state of the art solution and came up with GraphQL. It is the definition for the combination of two things. On the one hand it is a query language for any existing API/connection to a database and on the other hand it is an server-side runtime for executing queries

that are defined by a type system based on data from any backend storage. It is actively maintained by the the open-source community, created by Facebook and said to be next big thing after REST.
The major key part of GraphQL is that it can be set up on any database management system (DBSM), like on SQL-based or document based ones. It is not bound to any specific programming language nor to any server implementation. It is completely decoupled from those, which makes it easy to integrate in any existing system. Another advantage is, that it does not dictate your backend storage option. This makes it possible to easily switch the backend without affecting the existing API. But one of the most impressing parts is, that it solves a big issue we had with REST. When querying an instance in REST, this very often ended up with multiple queries to different endpoints, while with GraphQL this can be accomplished with only one request.
For example in our use case: A query for an tweet and the corresponding author of the tweet in REST would normally result in two calls. One to the tweet endpoint and one the author endpoint. With an GraphQL based API the request can be fulfilled with only one call:

A GraphQL service is defined by its `type definitions`. A sample one would look like the follwoing:

```
type Query {
  tweet: Tweet
}

type Tweet {
  id: ID
  message: String
  author: Author
}

type Author {
  id: ID
  name: String
}
```

Type definitions are like a schema for the API. They define which type requests are accepted by API.

So called `resolvers` then take care of the actual database call to get the instances needed.

```
function Query_tweet(request) {
  return database.getTweet();
}

function Tweet_message(tweet) {
  return tweet.getMessage();
```

```
}

function Tweet_author(tweet) {
  return tweet.getAuthor();
}
```

An example request to the API could be:

```
{
    tweet{
        id
        message
        author{
            id
            name
        }
    }
}
```

Respone:

```
{
    data: {
        tweet: {
            id: 123456789
            message: "This is a test"
            author: {
                id: 1
                name: "John Doe"
            }
        }
    }
}
```

With this example it is very clear to see that it is possible to easily request only the data we want and do this with only one API request. This and the fact that GraphQL can work with any programming language, any server implementation and the option to completely switch out the DBMS if needed, we chose it as our standard API runtime.

React (Client - Front-End)

To develop our front-end client side we wanted to use a modern, well supported and written framework and found the best solution in React. React is an open source front-end JavaScript library for dynamically creating user interfaces. It is very actively maintained by Facebook and the open source community and offers a variety of additional packages which can be used to extend the functionalities

of it. Currently it is available in version 15 with version 16 at beta state. A lot of big companies are building their web application based on it. One of the most prominent ones are AirBnB, Netflix and Facebook itself.

React uses a sepcial JavaScript syntax called `.jsx`. With that it is possible to write HTML code inside a JavaScript file. A simple hello world `.jsx` example file looks like this:

```
import * as React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
               <p>Header</p>
               <p>Content</p>
               <p>Footer</p>
         </div>
      );
   }
}

export default App;
```

As we see inside the `render` method of this class we can write pure HTML code without breaking the code. React in the end renders this HTML dynamically to the Virtual Document Object Model (Virtual DOM) which is another notable feature of React. Instead of rendering directly to the HTML DOM it caches all changes inside the virtual DOM and updates the browsers displayed DOM accordingly in a efficient way.
React always tries to work with a component oriented way of structuring a front-end. For example a sidebar navigation is one component. Inside this component multiple link components are nested. This makes React projects very structured and good to maintain if some features need to be added.
React is in our eyes one of the most versatile tool for developing dynamic front-ends which makes it the best fit for creating the client part of SentiTomo.

The Application

Due to the fact that SentiTomo is built up with different technologies, there are some requirements that need to be fulfilled when you are trying to set up the application. To clarify the different environment settings and the overall structure the next section tries to discuss all of those facts.

Overview

The directory structure of the application can be seen in the following: (*without files except package.json and yarn.lock*):

SentiTomo is divided into two parts, client and server. The main entry point of the application (`/server/server.js`) lies in the server directory which is starting up the Node.js server process and serving files form the build directory of the client. Before we have a deeper look into the different directories we first introduce you the installation process of SentiTomo.

Installation

The main steps to install SentiTomo are:

1. Install Node.js and npm
2. Install Yarn (optional)
3. Install dependencies
4. Install Java Version 6 an 8
5. Install Python Version 3
6. Install R
7. Set up environment variables
8. Start the server

1. Install Node.js and npm

Node.js always comes in combination with it's packaging manager npm . It can be installed on any common OS, and therefore you can use nearly any server operating system you like. SentiTomo was built on MacOS X Sierra Version 10.12.6 so we suggest using a Unix based server here. To install Node you can either use the downloads provided by the Node website or using one of the following aproaches.

MacOS:

Using homebrew:

```
$ brew install node
```

Linux:

On Linux there are plenty of different installation possibilities but here we suggest to use the ones provided on the Node website.

Install Yarn (optional)

After installing Node it comes to the decision to either stay with npm as your desired dependency manager or additionally install Yarn. If you decide to go with yarn this has the advantage that you will get exact the same dependency version as we had while testing SentiTomo. This is ensured with the `yarn.lock` files. This is a non- mandatory step but when you are using npm it can be the case that you will get a slightly different dependency tree than with npm.

Yarn is installed via npm with the following command.

```
$ sudo npm install -g yarn # use sudo to ensure yarn is on your path
```

Install dependencies

Now that either npm or Yarn is set up we can use it to install the dependencies. The following commands assume that you are in the top level `sentitomo/` of the application.

npm:
$ # install server dependencies
$ cd server
$ npm install

```
$ # install client dependencies
$ cd ../client
$ npm install
```

Yarn:

```
$ # install server dependencies
$ cd server
$ yarn install

$ # install client dependencies
$ cd ../client
$ yarn install
```

Install Java Version 6 and 8

Some of the R scripts we use for machine learning need both versions of Java to work. To install the versions just follow the basic instructions of your specified distribution. You just have to ensure that the user who executes the Node process has access to both libraries.

Install Python and dependencies

The Python scripts for topic detection are based on Pyhton version 2 and 3. Follow the common ways to install it on your server OS and ensure that `python`

is the command to execute Pyhton 2 and `python3` to execute Pyhton 3 on the terminal.

Dependencies
Before you can run the Python files you have to ensure that all of the following modules are installed. Install module via `pip3 installl <moduleName>` for Python 3 modules and `pip2 installl <moduleName>` for Python 2 modules

Python3 modules

```
$ pip3 freeze
boto==2.48.0
bz2file==0.98
certifi==2017.4.17
chardet==3.0.4
cycler==0.10.0
datetime-truncate==1.0.1
funcy==1.8
future==0.16.0
gensim==2.3.0
gnip-trend-detection==0.5
idna==2.5
Jinja2==2.9.6
joblib==0.11
MarkupSafe==1.0
matplotlib==2.0.2
nltk==3.2.4
numexpr==2.6.2
numpy==1.13.1
pandas==0.20.3
py==1.4.34
pyLDAvis==2.1.1
pyparsing==2.2.0
pytest==3.1.3
python-dateutil==2.6.1
pytz==2017.2
requests==2.18.2
scikit-learn==0.18.2
scipy==0.19.1
six==1.10.0
sklearn==0.0
smart-open==1.5.3
textblob==0.12.0
urllib3==1.22
```

Python2 modules

```
$ pip freeze
```

```
altgraph==0.10.2
astroid==1.5.3
backports.functools-lru-cache==1.4
bdist-mpkg==0.5.0
bonjour-py==0.3
boto==2.48.0
bz2file==0.98
certifi==2017.4.17
chardet==3.0.4
configparser==3.5.0
enum34==1.1.6
gensim==2.3.0
idna==2.5
isort==4.2.15
lazy-object-proxy==1.3.1
macholib==1.5.1
matplotlib==1.3.1
mccabe==0.6.1
modulegraph==0.10.4
nltk==3.2.4
numpy==1.13.1
pandas==0.20.3
py2app==0.7.3
pylint==1.7.2
pyobjc-core==2.5.1
pyobjc-framework-Accounts==2.5.1
pyobjc-framework-AddressBook==2.5.1
pyobjc-framework-AppleScriptKit==2.5.1
pyobjc-framework-AppleScriptObjC==2.5.1
pyobjc-framework-Automator==2.5.1
pyobjc-framework-CFNetwork==2.5.1
pyobjc-framework-Cocoa==2.5.1
pyobjc-framework-Collaboration==2.5.1
pyobjc-framework-CoreData==2.5.1
pyobjc-framework-CoreLocation==2.5.1
pyobjc-framework-CoreText==2.5.1
pyobjc-framework-DictionaryServices==2.5.1
pyobjc-framework-EventKit==2.5.1
pyobjc-framework-ExceptionHandling==2.5.1
pyobjc-framework-FSEvents==2.5.1
pyobjc-framework-InputMethodKit==2.5.1
pyobjc-framework-InstallerPlugins==2.5.1
pyobjc-framework-InstantMessage==2.5.1
pyobjc-framework-LatentSemanticMapping==2.5.1
pyobjc-framework-LaunchServices==2.5.1
pyobjc-framework-Message==2.5.1
```

```
pyobjc-framework-OpenDirectory==2.5.1
pyobjc-framework-PreferencePanes==2.5.1
pyobjc-framework-PubSub==2.5.1
pyobjc-framework-QTKit==2.5.1
pyobjc-framework-Quartz==2.5.1
pyobjc-framework-ScreenSaver==2.5.1
pyobjc-framework-ScriptingBridge==2.5.1
pyobjc-framework-SearchKit==2.5.1
pyobjc-framework-ServiceManagement==2.5.1
pyobjc-framework-Social==2.5.1
pyobjc-framework-SyncServices==2.5.1
pyobjc-framework-SystemConfiguration==2.5.1
pyobjc-framework-WebKit==2.5.1
pyOpenSSL==0.13.1
pyparsing==2.0.1
python-dateutil==1.5
pytz==2013.7
requests==2.18.2
scikit-learn==0.18.2
scipy==0.19.1
singledispatch==3.4.0.3
six==1.10.0
smart-open==1.5.3
textblob==0.12.0
urllib3==1.22
wrapt==1.10.10
xattr==0.6.4
yapf==0.16.3
zope.interface==4.1.1
```

During the execution a proportion of our scripts throw deprecation warnings. These are only thrown for information purposes, they will not influence the execution of the code.

Some of the Python scripts are using the `Natural language toolkit (NLTK)` to preprocess messages, detect the sentiment or the topic of those, based on corpora, tokenizers, lexica etc. . In order to use this it is mandatory to copy the folder `nltk_data` (can be found in `sentitomo/`), to the home directory of the user who is running the application. Afterwards it can be removed from the `sentitomo/` directory.

On MacOS:

`/usr/<username>/nltk_data`

On Linux:

`/home/<username>/nltk_data`

On Windows Vista,7,8,10

```
<root>\Users\<username>\nltk_data
```

Install R

MacOS:

On MacOS we suggest using the `.pkg` file from the r-project site to install R.

Linux:

The same way applies for Linux, please have a look at the r-project site to install R.

Windows:

The same way also applies for Windows, please have a look at the r-project site to install R.

Set up environment variables

SentiTomo is using some environment variables to connect to different services which are necessary to run the application. **Be aware to never add `.env` to your VCS**

Database

Our application is using a MySQL database as it's backend. To use your database you have to modify the `.env` file in the root of `sentitomo/server`. This file is used for setting up some configuration settings. Just fill out the following key/value pairs to connect your database.

```
DB_NAME=yourDBName
DB_USER=yourDBUser
DB_PASS=userPassword
DB_HOST=hostURL
```

When the application is connected to the first time it will create the mandatory table structure in your database automatically.

Twitter

In order have access to the Twitter Streaming API you have to obtain your client credentials by creating a new Twitter APP on the Twitter Dev Website. After doing so, also save your credentials in the `.env`file:

```
TWITTER_CONSUMER_KEY=yourConsumerKey
TWITTER_CONSUMER_SECRET=yourConsumerSecret
TWITTER_ACCESS_TOKEN_KEY=yourAccessTokenKey
TWITTER_ACCESS_TOKEN_SECRET=yourAccessTokenSecret
```

It is also possible to change the filters which are used to crawl the Twitter API here.
A `,` indicates an OR and a whitespace an AND concatenation.

```
TWITTER_STREAMING_FILTERS="YOURFILTERS"
```

Examples:

- `"humira,abbvie"` crawls all tweets that contain `humira OR abbvie`
- `"humira abbvie"` crawls all tweets that contain `humira AND abbvie`

Facebook

To have access to the Facebook API you need to set the app credentials inside the `.env` file. To create a Facebook app visit: Facebook Dev Website. After setting up your APP copy the `APP ID` and the `APP Secret` into following line:

```
FACEBOOK_APP_ID=1234567890yourAPPID
FACEBOOK_APP_SECRET=123abcde123yourAPPSecret
```

SentiTomo is able to manually crawl Facebook pages based on keywords. To narrow down the results of those crawls to the pharmaceutical context there is a category filter set in `.env`. To change it just edit the following line:

```
FACEBOOK_PAGE_CATEGORY_FILTER="Medical Company,Pharmaceuticals,Biotechnology Company,Medical
```

Start the server

After installing all necessary software fragments you just need to start the server with the following command, assuming you are in the `sentitomo/server`directory:

```
$ npm start
#or
$ yarn start
```

After starting the server is listening on Port:8080.
If you test locally then visit:

- Front-End: localhost:8080/app/dashboard
- GraphQL Endpoint: localhost:8080/graphql
- Endpoint for testing the API: localhost:8080/graphiql

If you want to access the server from a remote destination just switch out localhost with your server IP or domain.

Oncethe server is started four methods are executed:

- `JavaShell('h20.jar').call()`
- `SentimentWorker.start()`
- `TopicWorker.start()`
- `TwitterCrawler.start()`

The first one will start the h20 platform which is needed by some machine learning tasks. It starts a second webserver at `localhost:54321`.

The second and third one will continously search for tweets which do not have sentiment or topic assigned to them, and will detect those. The last one will crawl the Twitter streaming API based on the keywords specified inside the `.env` file and save them into the database. The whole startup sequence can also be found displayed in the sequecene diagrams sections. All of these workers are executed asynchronously, so they do not infere or block each other.

In the next part we will have a deeper look into the database and the directories of SentiTomo.

Database

By default SentiTomo is using a MySQL database, but theoretically it can be used with any other DBMS. We chose a MySQL database because for us it was the easiest and fastest way DMBS to set up. When the application is initially connected to the database it is creating all necessary tables and foreign keys automatically. In the following the different CREATE statements of the tables are listed.

TW_Tweets holds all raw information of the tweets.

```
CREATE TABLE `TW_Tweet` (
`id` varchar(255) NOT NULL,
`keywordType` varchar(255) DEFAULT NULL,
`keyword` varchar(255) DEFAULT NULL,
`created` datetime DEFAULT NULL,
`createdWeek` int(11) DEFAULT NULL,
`toUser` varchar(255) DEFAULT NULL,
`language` varchar(255) DEFAULT NULL,
`source` varchar(255) DEFAULT NULL,
`message` varchar(255) DEFAULT NULL,
`hashtags` varchar(255) DEFAULT NULL,
`messagePrep` varchar(255) DEFAULT NULL,
`latitude` varchar(255) DEFAULT NULL,
`longitude` varchar(255) DEFAULT NULL,
`retweetCount` int(11) DEFAULT NULL,
`favorited` tinyint(1) DEFAULT NULL,
`favoriteCount` int(11) DEFAULT NULL,
`isRetweet` tinyint(1) DEFAULT NULL,
`retweeted` int(11) DEFAULT NULL,
`createdAt` datetime NOT NULL,
`updatedAt` datetime NOT NULL,
`TWUserId` varchar(255) DEFAULT NULL,
PRIMARY KEY (`id`),
```

```
KEY `TWUserId` (`TWUserId`),
CONSTRAINT `TW_Tweet_ibfk_1` FOREIGN KEY (`TWUserId`) REFERENCES `TW_User` (`id`) ON DELETE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

TW__Users contains information about the tweet authors.

```
CREATE TABLE `TW_User` (
`id` varchar(255) NOT NULL,
`username` varchar(255) DEFAULT NULL,
`screenname` varchar(255) DEFAULT NULL,
`createdAt` datetime NOT NULL,
`updatedAt` datetime NOT NULL,
`followercount` bigint(20) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

TW__Sentiment contains information about the sentiments of different tweets.

```
CREATE TABLE `TW_Sentiment` (
`id` varchar(255) NOT NULL,
`sarcastic` double DEFAULT NULL COMMENT 'Sarcasm probability',
`sentiment` varchar(255) DEFAULT NULL COMMENT 'Final sentiment result',
`emojiSentiment` double DEFAULT NULL COMMENT 'sentiment of emojis in the message',
`emojiDesc` varchar(255) DEFAULT NULL COMMENT 'emojis in the message',
`rEnsemble` varchar(255) DEFAULT NULL COMMENT 'Result from R_algos',
`pythonEnsemble` varchar(255) DEFAULT NULL COMMENT 'Result from Python_algos',
`createdAt` datetime NOT NULL,
`updatedAt` datetime NOT NULL,
PRIMARY KEY (`id`),
CONSTRAINT `TW_Sentiment_ibfk_1` FOREIGN KEY (`id`) REFERENCES `TW_Tweet` (`id`) ON DELETE (
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

TW__Topic contains information about the topics of different tweets.

```
CREATE TABLE `TW_Topic` (
`id` varchar(255) NOT NULL,
`topicId` int(11) DEFAULT NULL,
`topicContent` varchar(255) DEFAULT NULL,
`probability` double DEFAULT NULL,
`createdAt` datetime NOT NULL,
`updatedAt` datetime NOT NULL,
PRIMARY KEY (`id`),
CONSTRAINT `TW_Topic_ibfk_1` FOREIGN KEY (`id`) REFERENCES `TW_Tweet` (`id`) ON DELETE CASCA
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

FB__Profile contains information about different Facebook profiles.

```
CREATE TABLE `FB_Profile` (
`id` varchar(255) NOT NULL,
`keyword` varchar(255) DEFAULT NULL,
```

```
`name` varchar(255) DEFAULT NULL,
`category` varchar(255) DEFAULT NULL,
`likes` int(11) DEFAULT NULL,
`type` varchar(255) DEFAULT NULL,
`createdAt` datetime NOT NULL,
`updatedAt` datetime NOT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

FB_Post contains information about posts on Facebook.

```
 CREATE TABLE `FB_Post` (
`id` varchar(255) NOT NULL,
`message` varchar(255) DEFAULT NULL,
`lang` varchar(3) DEFAULT NULL,
`story` varchar(255) DEFAULT NULL,
`likes` int(11) DEFAULT NULL,
`link` varchar(255) DEFAULT NULL,
`created` datetime DEFAULT NULL,
`createdAt` datetime NOT NULL,
`updatedAt` datetime NOT NULL,
`FBProfileId` varchar(255) DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `FBProfileId` (`FBProfileId`),
CONSTRAINT `FB_Post_ibfk_1` FOREIGN KEY (`FBProfileId`) REFERENCES `FB_Profile` (`id`) ON DE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

FB_Comment contains information about comments on Facebook posts.

```
CREATE TABLE `FB_Comment` (
`id` varchar(255) NOT NULL,
`message` text,
`lang` varchar(3) DEFAULT NULL,
`createdAt` datetime NOT NULL,
`updatedAt` datetime NOT NULL,
`FBPostId` varchar(255) DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `FBPostId` (`FBPostId`),
CONSTRAINT `FB_Comment_ibfk_1` FOREIGN KEY (`FBPostId`) REFERENCES `FB_Post` (`id`) ON DELET
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

FB_Sentiment contains information about the sentiment of Facebook posts.

```
CREATE TABLE `FB_Sentiment` (
`id` varchar(255) NOT NULL,
`sarcastic` double DEFAULT NULL,
`sentiment` varchar(255) DEFAULT NULL,
`emojiSentiment` double DEFAULT NULL,
`emojiDesc` varchar(255) DEFAULT NULL,
```

```
`rEnsemble` varchar(255) DEFAULT NULL,
`pythonEnsemble` varchar(255) DEFAULT NULL,
`createdAt` datetime NOT NULL,
`updatedAt` datetime NOT NULL,
PRIMARY KEY (`id`),
CONSTRAINT `FB_Sentiment_ibfk_1` FOREIGN KEY (`id`) REFERENCES `FB_Post` (`id`) ON DELETE CA
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

FB_Topic contains information about the topic of Facebook posts.

```
CREATE TABLE `FB_Topic` (
`id` varchar(255) NOT NULL,
`topicId` int(11) DEFAULT NULL,
`topicContent` varchar(255) DEFAULT NULL,
`probability` double DEFAULT NULL,
`createdAt` datetime NOT NULL,
`updatedAt` datetime NOT NULL,
PRIMARY KEY (`id`),
CONSTRAINT `FB_Topic_ibfk_1` FOREIGN KEY (`id`) REFERENCES `FB_Post` (`id`) ON DELETE CASCAI
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Server

All the backend server logic is placed inside the `server`directory. We won't cover every single file or directory here, but we want to introduce the most important ones which are the `ML` and `data` directories and the `server/service/TwitterCrawler.js` file.

ML

Inside the `ML` directory we placed all files which are related to the machine learning tasks, like sentiment analysis, topic detection and trend detection. It is divided in three subdirectories `Java`, `Python` and `R` to seperate the different programming languages. In the top level of `ML` you can find the wrapper files for incorporating the different programming languages with Javascript, `ml_wrapper.js` and `preprocess.js`. To let the different programming files work together we use Node's opportunity to spawn child processes and capture the output of these. To do so, we wrote a small file for calling the foreign code in a more convenient way, than directly spawning the child processes in the `ml_wrapper.js`. Those methods can be found in `/server/util/foreignCode.js`.
With that procedure we can spawn the machine learning tasks asynchronously to the main process, which leads to the fact that the main thread is not blocked or influenced by executing foreign code. In the following we want to explain how we integration works in detail.

Preface (Important Notice)

It is not possible to pass complex data types and structures from Javascript to R, Python or Java. It can either be a **plain string** or a **JSON encoded string** which then needs to be parsed by the executed file. The same applies for the output. If the foreign code wants to output an complex object it is the best practice to convert it to a JSON representation so that the Node.js proceess can easily work with it.

Possible packages and modules to use with JSON

- **R** with RJSONIO
- **Pyhton** with JSON Module
- **Java** with org.json library

Paths

When the foreign code needs access to some other files inside a directory it is mandatory to know that all files are executed in the scope of `./server/`. For example if a Python file needs to load a model from the Python directory it has to do it with the relative path `./ML/Python/filename.bin`.

Examples

All examples in JavaScript are written with ES6. To have a look the wrapper function for calling the foreign code, have a look at `/server/wrapper/codeWrapper.js`.

h2o

SentiTomo is using h2o to execute some machine learning tasks for sentiment analysis. To do so it spawns a Java child process which is setting up a h2o server on port 54321. This second server is then used to execute some algorithms on it. It is gracefully shutdown when you terminate the SentiTomo process.

Important:

The R package version and the Java h20 server verison have to exactly match, if not the sentiment detection will not execute. We use the h2o version **3.10.5.3**.

If it is needed to have a look on the h2o server front end just visit localhost:54321. Then you can have a look at the different models we used for h2o.

R

For integrating R with the server we were at first using a package called r-script package. It ships with a handy R function called `needs()`. This is basically a combination of `install()` and `require()`. This ensured that the different packages which are required by our scripts are installed and loaded in the correct way. Therefore every R file has to use `needs()` instead of `install.package("packageName")` and `require("package")/load("package")`. Also it is recommended to place all

functions at the top of the R files.
At the end of our project we faced some problems with the package so we decided to write our own implementation of calling R files, as mentioned in the introduction of `ML`. We followed the same approach as the `r-script` package did, included the `needs.R` file but simplified the process a little bit, so it could work with our application,
To send data to the R process from and back to the Javascript we can call the R file from Javascript like the follwoing:

JavaScript

```
import { RShell } from './util/foreignCode'

RShell("example/test.R")
  .data([tweet.message])
  .call(function(err, d) {
    if (err) throw err;
    console.log(d);
  });
```

One thing to mention is that the implementation of `RShell` reads the console output from the R files. So if a file wants to a pass a value back to the JavaScript, for example the output of a classification task it **SHOULD NOT ASSIGN IT TO VARIABLE** just print it the console. The best way to do it is to write:

```
cat(yourVariable)
```

With `cat` it is possible to remove the `[X]` identifier in front of the output. If you want to print out a named variable the best approach is to use

```
cat(unname(yourVariable))
```

R

```
source(needs.R)
needs(dplyr) # require every library so
args = commandArgs(trailingOnly=TRUE) # read command line

# Here comes all your function
# function 1
# function 2
# *****

# assign the retrieved value to a local one, this comes from the Javascript code
# It was passed like this {message: tweet.message}. It has the same name, 'message'.
out <- args[1] # first command line argument
out <- gsub("ut","ot",out) # do something with it
out # last line of the script should always print the value which you want to return to the
```

Example for converting JSON to data.frame in R

See this great answer onStackoverflow

JavaScript

```
import { RShell } from './util/foreignCode'


const message = {message: "[{"name":"Doe, John","group":"Red","age (y)":24,"height (cm)":182
    {"name":"Doe, Jane","group":"Green","age (y)":30,"height (cm)":170,"wieght (kg)":70.1,"s
    {"name":"Smith, Joan","group":"Yellow","age (y)":41,"height (cm)":169,"wieght (kg)":60,"
    {"name":"Brown, Sam","group":"Green","age (y)":22,"height (cm)":183,"wieght (kg)":75,"sc
    {"name":"Jones, Larry","group":"Green","age (y)":31,"height (cm)":178,"wieght (kg)":83.9
    {"name":"Murray, Seth","group":"Red","age (y)":35,"height (cm)":172,"wieght (kg)":76.2,"
    {"name":"Doe, Jane","group":"Yellow","age (y)":22,"height (cm)":164,"wieght (kg)":68,"sc

RShell("example/test.R")
  .data([message])
  .call(function(err, d) {
    if (err) throw err;
    console.log(d);
  });
```

R

```
needs(RJSONIO)
args = commandArgs(trailingOnly=TRUE) # read command line
json <- fromJSON(args[1])   # comes from Javascript {message: variable}
json <- lapply(json, function(x) {
  x[sapply(x, is.null)] <- NA
  unlist(x)
})
do.call("rbind", json)
```

Outcome is a data.frame

```
     name           group    age (y) height (cm) wieght (kg) score
[1,] "Doe, John"    "Red"    "24"    "182"       "74.8"      NA
[2,] "Doe, Jane"    "Green"  "30"    "170"       "70.1"      "500"
[3,] "Smith, Joan"  "Yellow" "41"    "169"       "60"        NA
[4,] "Brown, Sam"   "Green"  "22"    "183"       "75"        "865"
[5,] "Jones, Larry" "Green"  "31"    "178"       "83.9"      "221"
[6,] "Murray, Seth" "Red"    "35"    "172"       "76.2"      "413"
[7,] "Doe, Jane"    "Yellow" "22"    "164"       "68"        "902"
```

Python

For Python we initially used a package called python-shell to execute single Python files. But as our Python version switched from 2 to 3 we had

some issues to make this package work with the newer version. So in the end we decided to add the the spawning of the child process for Pyhton to `/server/util/foreignCode.js`. With our implementation we were able to set the Python version manually. Additionally the files are able to retrieve command line arguments which makes the communication between JS and Python possible. This is again based on reading the console prints of the Python file. One advice is, to make sure to not heavily use the console for prints, because the main process only needs to know the final result of the script.

A small example with passing JSON forth and back:

Javascript

```
import { PythonShell } from './util/foreignCode';

/**
 * @function test
 * @param  {String} message  Message to the file
 * @param  {Function} callback Function to handle the sentiment result
 * @description Test function to show the Python procedure
 * @return {String} Result of the
 */
var test = function(message, callback) {
    PythonShell('./ML/Python/test.p').data([message]).call(result => {
        callback(result.trim());
    })
}

test('{"message": "This is my message"}', result => {
    console.log(result);
});
```

Python

```
import sys, json

# simple JSON echo script
for line in sys.argv[1:]:
  print json.dumps(json.loads(line))


#Plain decode JSON
json.loads(argv[1])
```

Output in the Javascript

```
{"message": "This is my message"}
```

Java

For Java we followed the same approach and added the spawning process of the files to our existing file for spawning foreign code. It is the exact same process like with Python, the only thing which is different that you have to execute the .jar instead of the Python file.

Javascript

```
import {JavaShell} from './wrapper/codeWrapper';';

 /**
     * @function test
     * @param  {String} message  Message to the file
     * @param  {Function} callback Function to handle the sentiment result
     * @description Test function to show the Java procedure
     * @return {String} Result of the
     */
    var test = function(filename, callback) {
        JavaShell("./ML/Java/test.jar").data([message]).call(result => {
            callback(result.trimt());
        })
    }

test("Hello world", result => {
    console.log(result)
});
```

Java

```
public class HelloWorld {

    public static void main(String[] args) {
        // Prints the command line argument to the terminal window.
        System.out.println(args[0]);
    }

}
```

data

Inside the `data` directory the connection to the database and the `GraphQL` schema defininiton are expressed.
For connecting to the database we use a package called Sequlize.js. It offers the opportunity to work on a higher level way with databases. It provides methods which just define the database schema and the resulting SQL queries for inserting, updating and deleting records are build dynamically and fully handled by the package itself. Sequelize supports PostgreSQL, MySQL, SQLite

and MSSQL dialects. It was also used in the resolver function for the GraphQL API (`sentitomo/server/data/resolvers.js`).

Another main part of the `data` directory is the set up of GraphQL which is written in `sentitomo/server/data/resolvers.js`. It handles all the request to the API whose schema is defined in `sentitomo/server/data/schema.js`. For this part we used a Node.js package called apollo-server. It is a great, easy to use open-source implementation of GraphQL on the server-side.
In the following we want to provide a sample request to the API and depict what the response looks like:

Request sent to localhost:8080/graphql:

```
{
  tweet(id: "881026061806571520"){
    id
    message
    sentiment {
      id
      sentiment
    }
    author{
      id
      screenname
      username
    }
  }
}
```

Response:

```
{
  "data": {
    "tweet": {
      "id": "881026061806571520",
      "message": "New2Trip: Rheumatoid arthritis-specific cardiovascular risk scores are not
      "sentiment": {
        "id": "881026061806571520",
        "sentiment": "negative"
      },
      "author": {
        "id": "2199494760",
        "screenname": "TripPrimaryCare",
        "username": "Trip Primary Care"
      }
    }
  }
}
```

What is happening ?

At first the request from the client is piped through the resolvers of `resolvers.js` and the `tweet(_,args)` method is invoked because the request accessed the `tweet` endpoint. The `args` object contains all arguments provided to the endpoint. In this example only the `id` value of the tweet. To get the result from the database we used the database model object of sequlize.js called `Tweet` which is defined in the `connectors.js` file. It returns the exact tweet 'where tweet.id = args.id'. In the end a new data object is constructed which contains all the information that were requested.

`resolvers.js` (truncated)

```
import {
    TweetAuthor,
    Tweet,
    TweetSentiment,
    TweetTopic,
    FacebookProfile,
    FacebookPost,
    FacebookComment
} from './connectors';
import moment from 'moment';
import GraphQLMoment from './scalar/GraphQLMoment';



const resolvers = {
    Date: GraphQLMoment,
    Query: {
        ...,
        tweet(_, args) {
            return Tweet.find({
                where: args
            });
        },
        ...
    },
    Author: {
        tweets(author) {
            return author.getTweets();
        },
    },
    Tweet: {
        author(tweet) {
            return tweet.getTW_User();
        },
```

```
        sentiment(tweet) {
            return tweet.getTW_SENTIMENT();
        },
        topic(tweet) {
            return tweet.getTW_TOPIC();
        }
    },
};
```

`schema.js` (Provides the different API queries and the available fields)

```
/**
 * @constant typeDefinitions
 * @type {String}
 * @description Type definition schema for the GraphQL API. here all types, queries and muta
 */
const typeDefinitions = `

  scalar Date
  type Tweet {
    id: String
    keywordType: String
    keyword: String
    created: String
    createdWeek: Int
    toUser: String
    language: String
    source: String
    message: String
    messagePrep: String
    latitude: String
    longitude: String
    retweetCount: Int
    favorited: Boolean
    favoriteCount: Int
    isRetweet: Boolean
    retweeted: Int
    author: Author
    sentiment: Sentiment
    topic: Topic
  }

  type Author {
    id: String
    username: String
    screenname: String
    tweets: [Tweet]
```

```
  }

  type Sentiment {
    id: String
    sentiment: String
  }

  type Topic {
    id: String
    topic1Month: String
    topic1Month_C: String
    topic3Month: String
    topic3Month_C: String
    topicWhole: String
    topicWhole: String
  }

  type Query {
    tweet(id: String): Tweet
    sentiment(id: String): Sentiment
    topic(id: String): Topic
    author(username: String): Author
    tweets(limit: Int, offset: Int, startDate: Date, endDate: Date): [Tweet]
    count(startDate: Date, endDate: Date): Int
  }

  schema {
      query: Query
  }
';
export default [typeDefinitions];
```

Client

SentiTomo comes pre-shipped with an own front-end implementation written with the React framework and build with the create-react-app configuration. This makes it possible to integrate the app into any static HTML site you want.

We created it to show a way of how to integrate a fully customizable front-end to our server. The fact that the React app is converted into a static `.js` file makes it very easy to integrate it into the express Node.js server. Express.js is able to send a static HTML file down to the client if some specific URL is visited. We configured the server to use serve the app to the client if the `/app/*` URL is visited. Furthermore we added a static endpoint for serving CSS and web fonts which are used in the React app.

```
server.use('/static', express.static(path.join(__dirname + '/../client/build/static')));
server.use('/static', express.static(path.join(__dirname + '/../client/build/fonts')));

server.get('/app/*', (req, res) => {
    res.sendFile(path.join(__dirname + '/../client/build/index.html'));
});
```

The client uses websockets to open a bidirectional communication channel to the server when it is opened at the first time.

```
import io from 'socket.io-client'
const socket = io(`http://localhost:8080`, { reconnectionDelay: 4000 });
export default socket;
```

```
 socket.on('connect', data => {
    NotificationManager.success('Connected to the server', 'Success', 3000);
})
```

This channel is primarily used for initiating server methods which are

Because of the GraphQL API it is even possible to totally switch out the entire client directory and develop another solution.
When the application is started, the front-end is available at localhost:8080/app/.
It offers two different views right now:

- /dashboard rudimentary dashboard
- /toolbox different options to initate machine learning tasks dynamically

Sequence Diagrams

Server start:

Conclusion

With this application we wanted to show one way of using our findings and algorithms in a production-like environment. We showed how it is possible to built a server application which is capable of integrating different programming languages and use them for machine learning in an efficient way. With Node.js it is an ease to set up an highly scalable server which can handle asynchronous execetuions of foreign code very well. With the help spawning child processes a reliable communicate with those files is possible. With the help of GraphQL it was possible to set up a very modern and reliable API which any front-end can use to retrieve the result, which are produced by the the machine learning algorithms. If it is necessary to switch out the DBMS in the future the API endpoints will not change and the front-end system would not need to be heavily modified.

The client side was implemented in React, which makes it highly dynamical from a user point of view and well structured from a programmer's point of view. With the help of different packages it was possible to built a easy to use and good looking user interface.
All in all we hope that we could show you one possible way of how to build a good solution for monitoring topics and trends in social media today.

Questions

If you any question arise while looking trough the docs, the code or while testing the application, please do not hesitate and contact us.

Note

We did not included application tests inside, because this is more like a showcase how the integration of such an application can be done rather than providing a fully working and mature application.