Abstract

In the seventeenth century, Philosophers such as Leibniz and Descartes put forward proposal for codes to relate words between languages. The first patents for ”translating machines” were applied for in the mid-1930s. Up to the 1980s, most Natural Language Processing (NLP) systems were based on complex sets of hand-written rules. At that time however, the introduction of machine learning algorithms for language processing revolutionized NLP.[5]

In 2008, Collobert and Weston exhibited the power of pre-trained word embeddings in a paper called A unified architecture for natural language processing. Here, word embeddings is highlight for its ability in downstream tasks. They also discuss a neural network architecture that many of todays approaches are built upon. In 2013, Mikolov created word2vec, a toolkit that enabled the training and use of pre-trained embeddings. In 2014, Pennington introduced GloVe, a competitive set of pre-trained embeddings.

Starting off, a single word or group of words can be converted into a vector. This vector can be created using the Skip gram method, which predicts the possible words nearby, the LSTM-RNN method, which forms semantic representations of sentences by learning more about the sentence as it iterates through a sentence, using single convolution neural networks, and several other methods.

Using these theories, we are trying to build a Similarity Engine which provides machine learning based content search and classification of data.
Acquia

Acquia provides the leading cloud platform for building, delivering, and optimizing digital experiences.

Global organizations use the Acquia Platform to create a single, powerful digital foundation for delivering world-class customer experiences. With the Acquia Platform, organizations can realize the vision of personalized, contextual, and real-time customer engagement for the right person, at the right time, on the right device. These experiences can be easily deployed and managed at scale, for one to thousands of sites, driving transformative business results at breakthrough speed.[2]

In short, Acquia aims to provide software as a service (SaaS) with a focus on providing services and support for the Drupal Content Management System.[16]
Acknowledgments

Thank you to my direct manager, Radhika Chippada (Journey Team Manager at Acquia), who has offered to supervise my thesis at Acquia and help with my work.

Thank you to Krista Williams, the UI developer I worked with through all the front end development.

Thank you to the Data Science team (Katherine Bailey, Sunny Chopra and Jenny Heffernan) without whom we would have nothing to build around.

Lastly, thank you to Aleksander Madry, NBX Career Development Assistant Professor of Computer Science in the MIT EECS Department who will be overseeing my thesis on the MIT front.
Contents

1 Introduction .......................................................... 13
   1.1 Work Structure ................................................. 14

2 Back End Development ............................................. 15
   2.1 RESTful API ..................................................... 15
      2.1.1 Checking the Requests .................................... 16
      2.1.2 Responses ................................................ 16
   2.2 Processing requests ........................................... 17
      2.2.1 Cache ...................................................... 17
      2.2.2 Mathematical Processing ................................. 17
   2.3 Infrastructure .................................................. 18
      2.3.1 Amazon Web Services ..................................... 18
      2.3.2 SQL ......................................................... 18
   2.4 Logging Errors .................................................. 19

3 Back End Testing ................................................... 21
   3.1 Unit Tests ....................................................... 21
   3.2 Git Pre-Commit Restrictions ................................. 23
   3.3 Continuous Integration Testing with Travis ................. 23

4 REST API Spec using RAML ........................................ 27
   4.1 Writing a spec .................................................. 32
      4.1.1 RAML2HTML .............................................. 32
4.2 Testing the spec
   4.2.1 Manually Hitting the Endpoints
   4.2.2 Testing with Abao

5 Front End Development
   5.1 New Features
   5.2 Stylistic Design

6 Front End Testing
   6.1 Headless Browsing
   6.2 Git Pre-Commit Restrictions

7 Next Steps
   7.1 Short Term
   7.2 Long Term
## List of Figures

- **3-1** Unit Tests ........................................ 22
- **3-2** Travis ............................................ 24
- **3-3** Travis Continued ................................. 25
- **4-1** RAML Landing Page ............................... 28
- **4-2** RAML Request Spec .............................. 29
- **4-3** RAML Response Spec ............................ 30
- **4-4** RAML Response Spec Continued ............... 31
- **4-5** Postman ........................................... 33
- **5-1** Home Page ........................................ 39
- **5-2** Batches Page ...................................... 40
- **5-3** Documents Page ................................... 41
- **5-4** Document Modal .................................. 42
- **5-5** Classify Warning Modal ......................... 43
- **5-6** Classifier Page ................................... 44
- **5-7** Assign Classifications Page .................... 45
- **5-8** Select Category Page ............................. 46
- **5-9** Add New Category Page ......................... 46
- **5-10** Review Classifications Page ................... 47
- **5-11** Classified Documents Page .................... 48
Chapter 1

Introduction

Starting August 14th 2017, I started an internship with Acquia as part of the VI-A program. I joined a newly formed team under the product scope; we are building a similarity engine that can find the most similar text within a batch of texts to a piece of text. Moreover, it can categorize a large batch of documents with several categories after a few documents have been categorized. This system is open-ended and could potentially allow a large number of platforms to use its software. I will explore the potential later. When I joined this team, there were four including myself; as of now, the team is at seven.

The possibilities of such a system are endless. We are constantly looking for opportunities to bucket and group various items. While such a system is primarily used for text, it is extremely scalable.

As mentioned before, the advertising world acts on the information that we give it to recommend other similar information for us to use. Examples of this include Facebook recommending articles, Twitter suggesting people to follow, generic ads on most web articles and pages, etc.

However, we can see classifications and recommendations in other areas too: music on Spotify and Apple Music, videos on YouTube, moves and TV shows on Netflix, etc. However, majority of these are manually categorized and based on other user interactions. As we get better with representing music, videos and other forms of media in vector format, we should be able to classify and recommend those using the
likes of Euclidean distance and other mathematic comparators.

We are looking to build a scalable system that can be used across these other forms of media.

1.1 Work Structure

From August 2017 till February 2018, I worked at Acquia 40 hours a week and switched to part-time for 12 hours a week from February till now.

We worked in two-week sprints which kicked off with a discussion of the next priorities and a potential assignment of tickets that are to be achieved that sprint. The sprints culminated in a demo of the work and some feedback on how to be more efficient.

We used JIRA to help us with our project management;

Jira is a proprietary issue tracking product, developed by Atlassian. It provides bug tracking, issue tracking, and project management functions.[6]
Chapter 2

Back End Development

Our back-end is primarily written in Python with SQL calls to pull information from the database quickly. We are using Amazon’s Web Services (AWS) to host the server and store data.

We also used Flask to host the Python application.

2.1 RESTful API

We spent a good amount of time ensuring that the naming convention for the routes are intuitive to use.

The four types of requests that we catered to include GET, POST, PUT and DELETE.

Simply put, GET requests are to retrieve information, POST requests are to add new data to the database, PUT requests are to edit existing data in the database and DELETE is used to remove information from the database.

I wrote a lot of the endpoints currently being used from scratch.

In order to document our API well, I wrote the RESTful API Modeling Language (RAML) documents. I discuss that in more depth in the next chapter.
2.1.1 Checking the Requests

The first step that we carry out is to check the request arguments to see if the necessary ones are there. In our current implementation, we are simply checking that there is an account ID and that it corresponds to one in the database. If either of those are incorrect, it returns a bad request response.

Depending on the endpoint, we then check that the body matches the desired schema and return a bad request response if it is not.

I contributed by writing bits of code that would check the requests and validate the inputs.

2.1.2 Responses

In our system we had five main status responses:

- the 200 OK response is used for all requests where the request was met successfully

- the 202 ACCEPTED response is used for all asynchronous requests where the request is not completed yet

- the 400 BAD REQUEST response is used for requests where the necessary arguments are not present or incorrect

- the 404 NOT FOUND response is used for requests where the desired database item is not found

- the 500 INTERNAL SERVER ERROR response is used in the case where there is any other error

I standardized responses across all endpoints so that they were consistent for the types of requests.

With regards to responses bodies, I tried to minimize the size of responses by giving only the necessary information in each request.
2.2 Processing requests

Once the request is validated, the next step is to call the functions that parse the requests, manipulate the format of the information, validate it, and then connect it to the storage functions.

I wrote a large number of these functions, and these were updated on an iterative basis.

2.2.1 Cache

During this step, the cache is usually checked to see if the same request was made recently and if so, that response can be returned. If however it is not in the cache, the information may be stored in the cache during this step.

We primarily used Redis for this: Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker.[14]

2.2.2 Mathematical Processing

A lot of mathematical calculations are carried during these steps; such as vectorizing texts and performing operations on the vectors.

For this, we used NumPy: NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.[8]

For the machine learning aspect, we occasionally used Scikit. Scikit-learn (formerly scikits.learn) is a free software machine learning library for the Python programming language.[15]

We also used Pandas: pandas is a software library written for the Python programming language for data manipulation and analysis.[9]
2.3 Infrastructure

Given that the similarity engine is built on the premise that it compares information to large sets of data. We are using AWS to store this data, and SQL to make calls to the database.

2.3.1 Amazon Web Services

Amazon Relational Database Service (Amazon RDS) is a web service that makes it easier to set up, operate, and scale a relational database in the cloud.[3]

One of the issues with have large data sets is the time taken for the transfer of information. For example, retrieving the contents of a batch may take several seconds. One of my tasks was to convert all the get requests from the storage to be in an iterative sense; thus an attempt to get a data set with 10,000 rows may make five calls of getting 2000 rows. However, the AWS may have its own restrictions on the size of the data that can be transferred in a single call - so the implementation of that had to check how many documents were actually received and make extra calls if that was less than the value of documents requested - this may happen easily since some documents are magnitudes larger than other documents.

Another aspect we realize that might be an issue is the uploading of large sets of data. We have so far only enabled the use of CSV files as a method of uploading data. But upon uploading a CSV file, we found that the time taken to parse the rows into the databases in our storage took too long. Thus I created a system where the CSV file is initially uploaded to an S3 bucket and then another function is triggered that pulls the file from the bucket in the background and processes it.

2.3.2 SQL

We use the Python library, MySQL, which is an open-source relational database management system (RDBMS) [7], for making SQL calls.
2.4 Logging Errors

One of the other major things I worked on was the logging of the system. In preparation for when we release this, we needed to ensure that when users encounter errors that it is logged so that we can inspect the issue.

While we have used an external library for the main aspect of logging, the main things we had to consider included:

- Descriptive log statements
- Pointer to where in the code the issue was encountered
- Which platform this was encountered: development or production
Chapter 3

Back End Testing

In order to ensure that we have built a stable system that can constantly be updated without mistakes seeping through the cracks, we started with a rather extensive testing plan.

This can be broken down into three major categories; unit tests, putting in Git pre-commit restrictions and Travis testing on pushes.

There is another component: the continuous integration (CI) tests which in effect is testing the endpoints. However, I will cover that in the next section.

3.1 Unit Tests

Unit tests are used to test the intermediary functions between the end points and the database. In order to test it, the functions needs to be called with test data and the response tested with an assertion to be equal to some expected value.

While writing the unit tests, we aimed to get good code coverage by trying empty, valid and invalid inputs as well as an assortment of inputs for the optional inputs.
Figure 3-1: Unit Tests

As you can see, one can run a large number of unit tests in a short period of time and using this, we can quickly narrow down which function has the issue.
Using this, it is easy to isolate the issue if there is an error in the code.

### 3.2 Git Pre-Commit Restrictions

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. The pre-commit hook is run first, before you even type in a commit message. It’s used to inspect the snapshot that’s about to be committed, to see if you’ve forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code.\[10\]

We used the pre-commit for two purposes:

1. To ensure that developers don’t commit directly to the master branch.
2. Ensure that the committed code follows a certain code style

### 3.3 Continuous Integration Testing with Travis

This tool can be used to automatically run tests on a branch on each commit and merge. We set it to test any branch that is committed and send an email update on the results to the person who submitted the code. We also set it to run the same tests on master every time a branch is merged into master.
Figure 3-2: Travis

This is the interface of the tool. This is an example of what it looks like when the test is complete.
Figure 3-3: Travis Continued

If you expand the results of the test, you can see the detail of the tests as would show in the command line.
Chapter 4

REST API Spec using RAML

"RESTful API Modeling Language (RAML) makes it easy to manage the whole API lifecycle from design to sharing. It’s concise - you only write what you need to define - and reusable. It is machine readable API design that is actually human friendly.”[11]
Figure 4-1: RAML Landing Page

If you navigate to the HTML version of the RAML, you will get something like this. This is a user friendly interactive version of the RAML where you can see what the endpoints are and how to hit them.
Figure 4-2: RAML Request Spec

Clicking on one of the endpoint functions will pull up a modal like this which describes the spec of the request, possibly with an example.
Figure 4-3: RAML Response Spec

Navigating to the response tab, you can view the spec of the response.

```json
{
  "batch_id_1": {
    "num_docs": 23
  },
  "batch_id_2": {
    "num_docs": 1776
  }
}
```
HTTP status code 400

This can happen if no account ID passed in or the account ID provided is not in the list of supported account IDs.

Body
Type: application/json

Schema:

```json
{
   "$schema": "http://json-schema.org/schema",
   "id": "error_response.json",
   "title": "Error Response",
   "type": "object",
   "properties": {
      "error": {
         "type": "string"
      }
   },
   "required": [
      "error"
   ]
}
```

Example:

```json
{
   "error": "Some error message"
}
```

Figure 4-4: RAML Response Spec Continued
4.1 Writing a spec

One of my tasks was to write the spec of the endpoints so that other teams within Acquia that were looking to use our similarity engine would know how to interact with it in while we were building out the user interface.

This is typically done as part of good documentation and is meant to help developers new to the code base quickly understand how the system works and for other people interacting with it to know how to make appropriate calls.

This involved writing large chunks of code spec and cross checking that they match the code. During this time, I was also able to identify other gaps in the code and help improve the input checking at the endpoints.

4.1.1 RAML2HTML

In order to make this visually pleasing, I used a script called raml2html [12] which takes the RAML file that I wrote and converts it into a user friendly HTML page. In fact, due to compatibility issues with this, I had to write the entire spec in RAML 0.8 and then convert it to RAML 1.0 which has a good number of nuance differences.

4.2 Testing the spec

4.2.1 Manually Hitting the Endpoints

While working on the routes, we wanted to be able to test each of the endpoints individually. For the most part, this could be done using Curl calls. I also used a tool called Postman.
Figure 4-5: Postman

Using this tool I was able to configure the type of requests, the body of the requests, the headers and was able to parse all the responses as well.
4.2.2 Testing with Abao

In order to ensure that the endpoints had good input checking and consistently threw the correct exceptions based on requests and server processing, we decided to build tests to test the endpoints.

To do this, I used a REST API automated testing tool called Abao [1]. Abao takes a RAML file and generates a set of empty tests (in the form of a JacaScript hook file) that expect each of the status responses for each end point. In order to run this, the server needs to be running. In order to check that the exact parts of the code are hit to get the necessary status response, I had to customize the empty JavaScript hooks with specific information such that it would hit the expected code. Moreover, in order to check the value of the responses, I would write assertions to check if the body value was equivalent to the expected value.

Here is a sample test:

```javascript
hooks.before('DELETE /classifiers/{batch_id} -> 200',
  function (test, done) {
    test.request.params = {
      batch_id: 'second_batch'
    };
    test.request.query = {
      account_id: 'acquia-demo'
    };
    done();
  });

hooks.after('DELETE /classifiers/{batch_id} -> 200',
  function (test, done) {
    assert('status' in test.response.body);
    assert.equal(test.response.body.status, 'Deleted');
    done();
  });
```
});
Chapter 5

Front End Development

For the front end, we built the application using Facebook’s React infrastructure. "React is a declarative, efficient, and flexible JavaScript library for building user interfaces.”[13]

The two major components of building out the front end was incorporating means for the user to access all the functionality available from the back end and making the user interface super friendly.

5.1 New Features

There were numerous features that I worked on with regards to the user interface:

- Display document modal
- Displaying only enabled navigation buttons based on the account ID
- Infinite scrolling which gets more documents in batches once the bottom of the page has been reached
- Allowing users to download a CSV file of the classification predictions
- Error logging and handling
- Sorting of documents based on confidence scores
5.2 Stylistic Design

There were a ton of stylistic design aspects I worked on too - while this had some CSS, majority of my work consisted in making it easier for users to navigate around the page and easily interact with data:

- Display of batches and document lists
- More navigation buttons in appropriate places
- Home page display look
- Wording and phrasing of buttons and navigation
- Better explanations on error handling
- Refreshing when necessary
- Warning when multiple users are classifying the same batch to prevent overlap of requests
Figure 5-1: Home Page

This is the landing page after you login. From this page you can see all the enabled features for the account and clicking on either the top navigation buttons or the squares on the right will take you to the respective pages.
Figure 5-2: Batches Page

This page lists all the batches in the account. For each batch, you can see the number of documents in the batch and can easily add a new batch, delete an existing batch or navigate to the list of documents in the batch by clicking on the row.
This page lists all the documents in the batch. One can easily classify the documents in the batch by clicking on the 'classify' button or view a document by clicking on the document row.
If you click a document row on the Documents Page, a modal with the document content will pop-up over the page. We tried to maintain external consistency by matching with the likes of Google Documents.
For the design principle of safety, we have put checks on requests where we ask for confirmation before an action is performed - this check happens in a custom modal. Similar modals appear when there is an error processing a request.
On this page you can select the categories that you would like to use to classify a batch with.
On this page you can assign classifications to a page. To assign a document to a category, you click on the notepad symbol on the document row. After some documents have been assigned to categories (at least one document must be assigned to each category), pressing 'Next' automatically classifies the remaining documents.
In the popover that appears when selecting a category, a document can be assigned to a category by clicking the category.

Figure 5-9: Add New Category Page

In the popover, there is also the option to create a new category. Adding this category automatically assigns the document to the new category. Pressing enter locks it in.
After the documents have been automatically classified, if there are documents where the confidence level of their automatic classification is less than 60%, then the system asks the user to review the classifications.
Figure 5-11: Classified Documents Page

This page is an extension of the Documents Page, which lists each document’s category and confidence score as well. If the batch is classified, the ‘classify’ button is now seen as 're-classify’. Moreover, there would be a ‘download’ button which downloads the results of the classification as a CSV spreadsheet.
Chapter 6

Front End Testing

In order to ensure that we can build a user interface that uses scalable code where components can be used in multiple places and can be customized for certain pages while maintaining its functionality elsewhere, we thought of building an automated set of navigational steps using headless browsing.

As with the back end, we also had to incorporate a custom Git pre-commit to ensure that code style was held and there were no faulty commits.

6.1 Headless Browsing

In order to quickly test the UI, we did some experimenting with Chrome’s headless browser. It’s a way to run the Chrome browser in a headless environment. Essentially, running Chrome without chrome![4]

A sample test would be testing the login: this would include focusing on the user name and password inputs, trying various combinations of correct and incorrect information and seeing if the page redirects fall in line with the expectation - this would be done by checking for a page redirect and checking if the other pages have specific components.
6.2 Git Pre-Commit Restrictions

As before, this was used to ensure that nobody committed directly to master and that committed code followed a certain code style.
Chapter 7

Next Steps

As one can see, this engine has a lot of scope and could be used in a myriad of ways across a vast number of media types.

we have recently integrated the back-end into Acquia’s Lift personalization; a beta release + the alpha release of the UI for content classification. After our beta release is tested, we will open it to the general public. During this time we are working on the authentication component, expanding the language component and building out some other features. As soon as we receive the feedback, we will iterate on that version and release it to the larger public.

We also have some long term goals that are a bit far out that I will explain in more detail.

7.1 Short Term

In the short term we plan on enabling multilingual support for content recommendations. At a high level, we are looking into using word embeddings for other languages (using FastText) to support “other” languages. The problems with this are that we are limited to languages offered using FastText and the varying strength of the set of word embeddings for each language. We are also considering translating documents from other languages into English. The general problems with this would include the loss of information in translation and the time taken for translation.
The purpose of this being to support customers who host their content in multiple languages or other than English

7.2 Long Term

After building the back-end to handle the multilingual support in content classification, we plan to build the front-end to enable users to tap into that functionality. We also plan on working on the back and front end components of multi-label classification and supervised classification.
Bibliography


