An Infrastructure to Store and Analyse Seismic Data as Suffix Trees

MSc Data Analytics - Project Report

Tom Taylor

September 2017

This report is substantially the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service. I have read and understood the sections on plagiarism in the programme Handbook and the College website.

Contents

1 Introduction

1.1 Abstract

The purpose of this project is to develop an infrastructure and tool set for converting raw seismic time series data into a searchable string using SAX (**S**ymbolic **A**ggregate appro**X**imation) and then to store this data as a suffix tree for fast searching and analysis. An interface will then be developed to enable the searching of these suffix trees and provide visualisation of the data. Primarily this will be with the aim of being able to identify the start of an event with reasonable accuracy. Additionally it could ideally be used to search for similar patterns over time or between stations after an event.

1.2 Objectives

1. To facilitate the importing and storage of raw seismic data and associated meta-data
2. To calculate and store SAX strings of a whole observation period or an event
3. To be able to determine the onset of an event in an observation
4. To provide graphical representations of the analysis
5. To provide a web based user interface for all of the above

2 Background

2.1 Seismic Waves

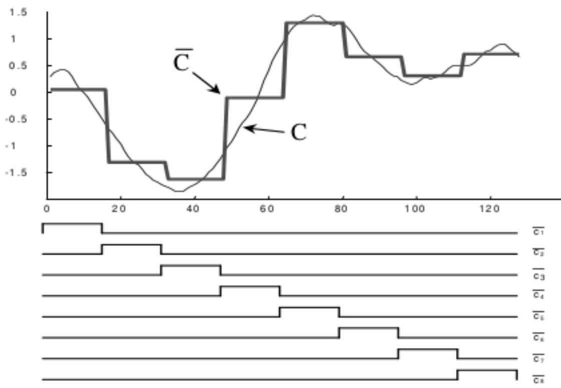
Seismic waves take on two main forms, body waves and surface waves. Body waves are those that travel through the interior of the earth and are the fastest travelling. The body waves are comprised of **P** (primary) waves which are compressional waves, travel fastest and thus arrive first. **S** (secondary) waves are shear waves and travel more slowly, thus arrive later. The separation between the phases is related to the distance of the earthquake and the local velocity structure. The surface waves travel only along the earth's crust and, as they are confined to shallow depths where seismic velocities are slow, will normally arrive much later than the body waves.

A seismic station records movement over three axis: vertical (**z**) alongside horizontal in terms of north-south (**n**) and east-west (**e**). Due to seismic velocities generally increasing with depth, P waves arrive at a seismic station close to the vertical axis. As a result, P waves can be measured as a simple metric of displacement along the Z axis. S waves follow similar ray paths, but have their particle motion perpendicular to the direction of propagation. As a result, they manifest on a seismogram as movement on both the n and e axis. The geometry of the fault tends to have a bearing on the orientation of the displacement so the two horizontal axis of movement cannot be easily combined in to a single metric for time series analysis.

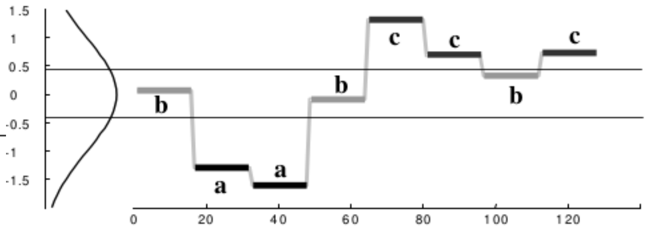
2.2 Symbolic Aggregate Approximation (SAX)

SAX (**S**ymbolic **A**ggregate appro**X**imation) (Lin et al., 2003) is a technique where by a single dimension of a time series is reduced to a string of symbols for pattern matching. The technique involves first transforming the normalised time-series in to a Piecewise Aggregate Approximation (**PAA**) which is then represented by a fixed number of symbols. The normalisation technique is *Z-normalisation* which is discussed in section 3.2.

For the PAA, the data is first divided into equal sized time frames (see diagram below), then the mean deviation from zero of each frame is calculated. An appropriate number of breakpoints symmetrical along the x-axis are created so that they follow a Gaussian distribution and a symbol assigned to each range between the breakpoints. Then for each frame, a symbol is assigned based on which range the mean falls in to. The symbols assigned to each frame are then concatenated in to a string and it is this string that gives the SAX representation of that data. The width of the time frame and the number of discrete regions would be two parameters passed to this process alongside the data.



(a) Calculating PAA



(b) Diagrams showing PAA to SAX on a time-series (taken from Lin et al. (2003))

2.3 Suffix Trees

Suffix trees (Weiner, 1973) are a representation of a string designed for performing search related algorithms. They are based on a suffix trie of a given string **T**. The trie is constructed using all of the suffixes of **T** with a terminator character not used in the alphabet appended to the end of **T**. The terminator is deemed to be lexicographically of lower priority than all of the other characters to ensure that a prefix of a string would appear before that string (such as in *as* before *ash*). In the following example, the suffixes of *T: abaaba* would produce the following suffixes with **$** applied as a terminator.

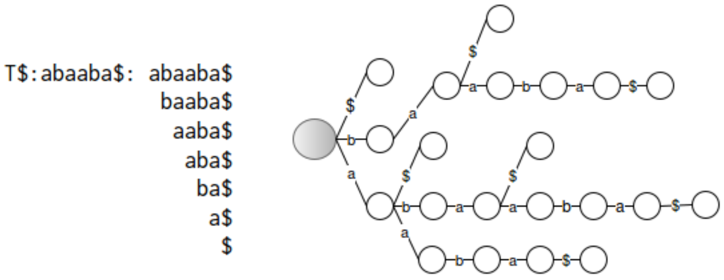


Figure 2: Example Suffix Trie

This allows for efficient searching and counting substrings of **T** by following the trie from the root node. Each substring **S** of **T** is a prefix of a path starting at the root. It allows for counting occurrences of the substring by finding the number of leaf nodes that can be reached from the end of the substring. This format is not particularly space efficient as it has an upper bound storage in the order of $O(n^2)$.

A far more efficient way to store and query the trie is to store it as Suffix Trees. In order to produce a suffix tree from a trie, the non-branching nodes are firstly reduced (or coalesced) in to a single edge. The original string **T** is then stored along side the tree and the labels further reduced to a starting position and offset within **T**. The leaf nodes become labels to the offset of the suffix in the string. This reduces the upper bound storage to $O(n)$ and results in a tree for our example **T** as seen in section 2.3.

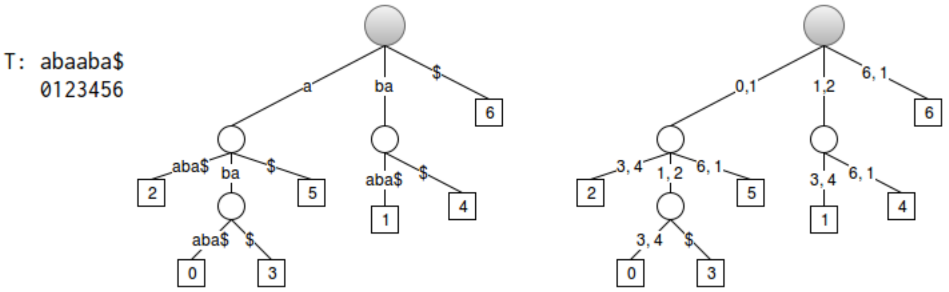


Figure 3: Example Suffix Tree (edges shown as text and as positions/offsets)

This technique of building the tree is considered naive though as it is again very inefficient (being computationally in the order of $O(n^3)$). It is far more desirable to achieve time-linear *O(n)* construction of the tree. An example of this is the Ukkonens Algorithm (Ukkone, 1995). The algorithm is too long for inclusion but effectively starts with the implicit suffix tree of a string of length 1 and conditionally extends the tree on each iteration of adding a character.

At Birkbeck, an infrastructure has been developed to load and query Suffix Trees (Harris et al., 2016). At a high level it is based on language models but should be extendable to time series by the use of SAX. There is current work in progress to utilise external memory (in this case Solid State Drives) to back the loaded Suffix Trees. This would massively increase the maximum size of a tree to be queryable. The current libraries to utilise this are written in Python however there is currently a refactor happening to C due to Python not being particularly computationally efficient because of its interpreted nature. The trees in this implementation are stored as k-truncated trees for practical reasons.

3 Data Analysis

3.1 Raw Data

The seismic data is provided in either SAC or Miniseed formats that are can be parsed with the Obspy Python library (M. Beyreuther and Wassermann, 2010). Use of this library produces an object containing the raw measurements along with metadata about the station it was produced from. The library also contains many methods for post-processing operations such as signal filtering and commonly used seismic analysis tools.

In the project, this object is wrapped in to a Data Access Object to add or simplify many of the commonly used operations such as downsampling (for rendering in a browser), time slicing and passing through a bandpass filter to remove high and low frequencies.

Due to the widely varying types and models of instrument deployed, the values reported from a Seismic Station are often relative only to themselves. This means while the profile of an event could be established from the seismogram, amplitudes are potentially meaningless when being compared between stations. It is also not possible to directly infer the strength of a quake from a seismogram without empirical evidence from previous events.

3.2 Normalisation

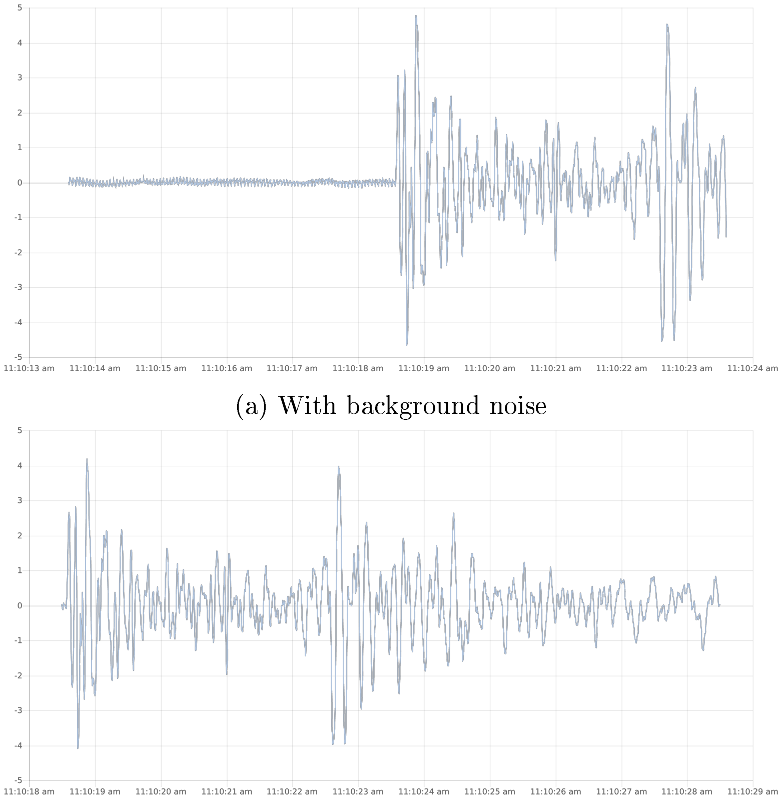
The preprocessing technique for normalisation suggested for SAX (Lin et al., 2003) is *Z-normalisation.* That is to normalise the data so that it has a mean ($\bar{x}$) of zero and a standard deviation (σ) of one. This is achieved by subtracting the mean from each value and then dividing by the standard deviation.

$$z = \frac{x - \bar{x}}{\sigma} \tag{1}$$

In a closed set of values, this technique often works perfectly fine. Unfortunately in this case, the vast majority of data points on seismic recordings are made up of background noise that are often indistinguishable from the events themselves apart from by amplitude. This means that if background noise is included in the sample from which the mean and standard deviation are taken, then values during the actual event will be over amplified and therefore unsuitable for *SAX* processing.

A further problem introduced by unsupervised normalisation is the tailing off of events. In the same way that introducing otherwise quiet periods amplifies the peaks of the event, so does including too much of the tail.

In view of these issues, and because *z-normalisation* is an important part of *SAX processing*, it is essential that only significant data be presented to be normalised for analysis.



(a) With background noise

(b) Without background noise

Figure 4: Comparison of normalisation with and without background

To meet this requirement, it is important that the start of the event be relatively accurately estimated and only the first few seconds are treated. This is also important because we are only interested in profiling *P-waves*. The *S-waves* following a few seconds later interfere with the Z-axis recordings. This is demonstrated in fig. 4 where if earlier background noise is included in the sample (a) the peaks are normalised a little over 4 where if only the event is included, the peaks are nearer to 5. When looking at longer time ranges, this effect is amplified significantly rendering the event data meaningless.

3.3 PAA & SAX

As described in the background section section 2.2 on SAX, there are two steps. PAA (Peicewise Aggregate Approximation) is applied before symbols are calculated based on equal breakpoints following a Gaussian distribution.

3.3.1 PAA

A Python class was written to be callable to convert a Dataframe into an array of the aggregated values. To reduce the number of steps involved in using the class, it performs the normalisation step unless told not to.

```
import numpy as np
import pandas as pd


class PaaError(Exception):
    pass


class Paa(object):
    def __init__(self, series=pd.Series, normalise=True):
        """
        Prepare a PAA (Piecewise Aggregate Approximation) object to calculate
        PAA of a given dataset.  Will perform z-normalisation by default on
        data     before interpolating linearly to an interval of 1ms.

        Args:
            series (DataFrame): pandas DataFrame with time as index
            normalise (bool): Whether or not to normalise
        """
        if not isinstance(series, pd.Series):
            raise PaaError("series␣should␣be␣a␣pandas␣Series")
        series = series.resample("1L", how="mean").interpolate(method="time")
        if normalise:
            std = np.std(series)
            mean = np.mean(series)
            series = (series - mean) / std
        self.series = series

    def __call__(self, window=int):
        """
        Return a PAA of the DataFrame

        Args:
            window (int): Number of milliseconds in window

        Returns:
            pandas.Series
        """
        if not isinstance(window, int):
            raise PaaError("Window␣should␣be␣an␣integer")
        df = self.series.copy()
        return df.resample("{}L".format(window)).mean().interpolate(method="
            time")
```

When the class is first instantiated, a copy of the series is stored locally in the object. Unless the normalise parameter is explicitly set to false, the numpy library is used to calculate the mean and standard deviation of the series and then $(x - \bar{x})/\sigma$ is calculated for the whole series as described in section 3.2.

This returns a callable object with the window size (in milliseconds) as a parameter. When called, the object uses the resample feature of Pandas to return a series of mean values. It should be defined and called as follows:

```
        p = Paa(series=d)  # where d is a Pandas dataframe with a time index
        paa_out = p(50)        # performs PAA on d with a window size of 50ms
```

3.3.2 SAX

Similar to the PAA class, the SAX class was written to produce a callable object.

```
from scipy.stats import norm
import numpy as np
from pandas import Series


class SaxError(Exception):
    pass


class Sax(object):
    def __init__(self, paa=Series):
        """
        Provides a generator for SAX data from PAA
        Args:
            paa (pd.Series): result from calling Paa
        """
        if not isinstance(paa, Series):
            raise SaxError("paa␣should␣be␣a␣pandas.Series,␣got␣{}".format(
                type(paa)))
        self.paa = paa

    def __call__(self, alphabet=str):
        """
        Generate SAX string from PAA as a pandas.Series

        Args:
            alphabet (str): alphabet for SAX

        yields:
            str
        """
        if not isinstance(alphabet, str):
            raise SaxError("alphabet␣should␣be␣a␣str,␣got␣{}".format(type(
                alphabet)))
        # Generate gaussian breakpoints
        thresholds = norm.ppf(
            np.linspace(1 / len(alphabet), 1 - 1 / len(alphabet), len(
                alphabet) - 1)
        )
```

```
for i in self.paa:
    yield alphabet[np.searchsorted(thresholds, i)]
```

On instantiation, copy of the original series is stored in the object with no additional pre-processing. The object is then called with the desired alphabet passed as the only parameter. Then length of the string is used to determine the number of breakpoints to calculate against a normal distribution and these are stored as the *thresholds*. A Python generator is then returned that uses the numpy *searchsorted* method to establish between which breakpoints a value falls and then return the corresponding character. The generator can then be used to iterate over the values one at a time by the calling function. A simple use to print the characters is shown below:

```
s = Sax(paa_out)           # instantiate a callable Sax object
for val in s("abcdefg"):   # iterate over the object
    print(val, end="")     # print each value (with no newline)
```

3.4 Frequency Analysis

Another option for analysing the stream was to disregard the amplitudes and concentrate on the frequency domain. That is to convert to a function of frequency against time.

Many of the frequencies that occur within the streams are considered noise, especially those of a higher frequency that are likely caused by wind.

4 Event Detection

The initial idea of the project was to be able to use the SAX data to detect events in a measurement over a long period of time. This ultimately proved to be too difficult to achieve due to the normalisation problems mentioned in section 3.2.

As such, an algorithm was needed to achieve this on unprocessed data.

4.1 Algorithm

The algorithm works by iterating through the data on two sliding windows. calculating a short and long term mean of the absolute values of the amplitude.

l = number of datapoints in LTA
s = number of datapoints in STA

$$\overline{LTA} = \sum_{i=1}^{l} \frac{|v(t-i)|}{l} \tag{2}$$

$$\overline{STA} = \sum_{i=1}^{s} \frac{|v(t-i)|}{s} \tag{3}$$

For each iteration, the STA is compared to a number of standard deviations (typically 3) from the LTA, if it exceeds this value, the event is considered triggered on. The current value of the LTA is then stored.

The iteration then continues calculating STA with the triggered value set to true until the STA drops below the original LTA value when the event was triggered.

If the event duration is above a pre-set threshold (typically 5s) then the event is recorded, if not the it is discarded as a probable spike.

The downside to this approach is that it cannot normally detect an event within the period of the LTA from the start of the observation window.

5 Application Design & Development

5.1 Specification

The application should be sectioned in to individual services following a microservices style architecture. This allows for development of a single component independently from others and allows the mocking of APIs that are not yet implemented. This also allows for faster iteration and integration testing. In a production-like environment, this means that development and bug-fixing of components can be done locally unlike with a monolithic application.

The application should also allow for deferred, batched or long-running tasks to be performed without interfering with the user experience.

Data should be persistent and not susceptible to race conditions and access times should be minimised.

It is a requirement that as much of the code is done in Python as possible. This is to allow potential later utilisation of the code by others in the Department.

5.2 Architecture Overview

Given the microservices style of application development chosen, it made sense to bundle each of the services in Docker containers. Containers are similar to Virtual Machines (VMs) in that they allow for isolation of filesystems, memory and network between processes or groups of processes but by sharing an Operating System Kernel it removes the need to run a hypervisor.

This also means development could utilise a tool called Docker Compose which allows for the running of a group of containers with their own virtual network on a workstation. It also allows for the running of community containers such as PostgreSQL, Minio and Redis which are dependencies of the application without having to install and administer them.

The user entry point to the application is served by the *interface* service. This provides a web based interface to the various backend components and rendering of graphs as well as acting as a reverse proxy to the various APIs behind the application.

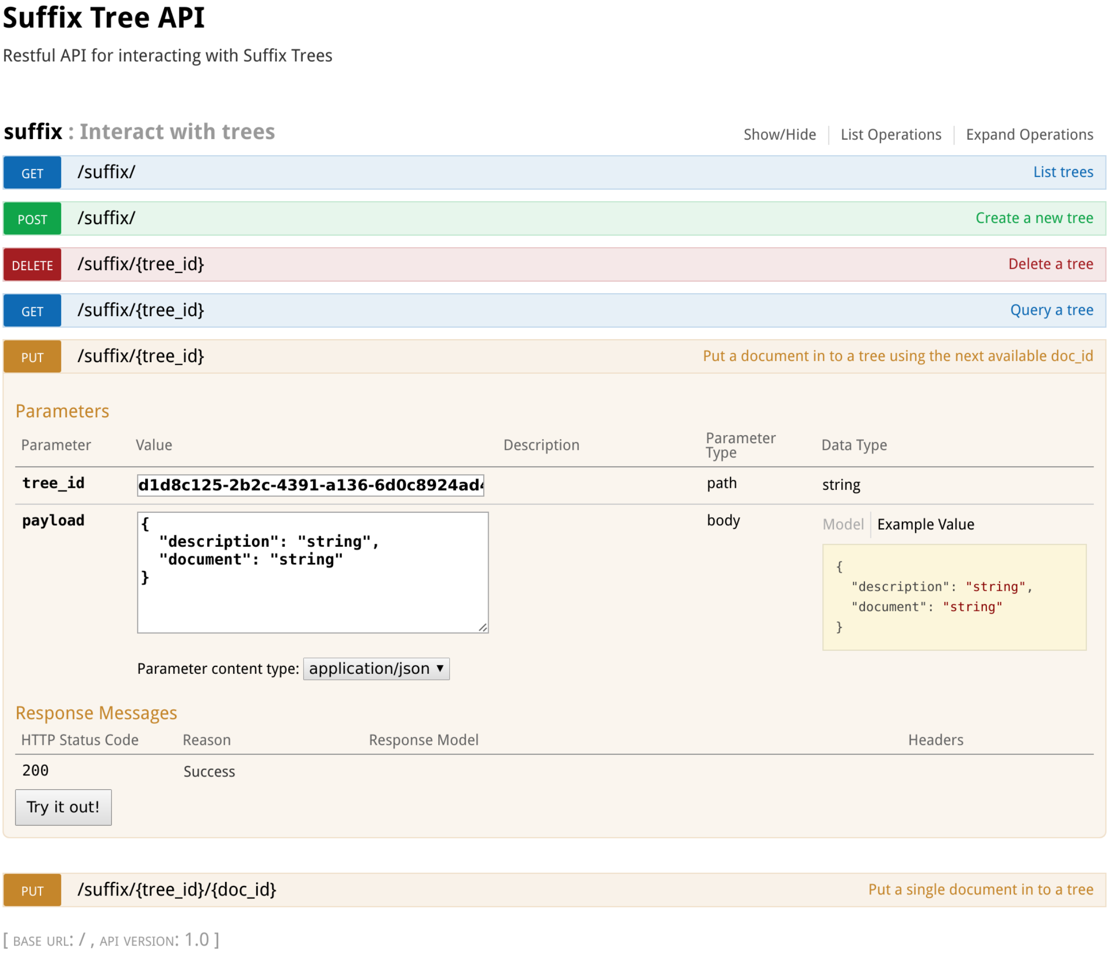
The *observations* service is exposed via a mostly JSON based HTTP API. It's primary responsibility is the storing accessing of raw observation files and events, as well as downsampling for rendering in a web browser.

The *SAX* service is exposed via a JSON based HTTP API. Its purpose is to perform PAA and SAX operations on a given event and return data for rendering visualisations as well as the produced string from the SAX calculation.

The *Suffix* service is also behind a JSON based HTTP API. It exists to store and query data with Suffix Trees.

The *worker* service exists to perform long running or compute heavy tasks asynchronously from the interface. It is exposed via a message queue running on Redis (an in-memory key-value database) where the interface or other services can queue tasks.

The HTTP/JSON APIs were written using Flask-Restplus, a Python framework based on Flask that allows for request and response definition using Python Decorators around classes and methods. The framework allows for dynamic generation of a swagger.json a commonly used format for defining APIs. Swagger also provides a user friendly HTML based interface for testing methods and calls during development and also doubles as API documentation. An example of Swagger used on the Suffix Tree API is shown in fig. 5.



Suffix Tree API

Restful API for interacting with Suffix Trees

suffix : Interact with trees

GET	/suffix/	List trees
POST	/suffix/	Create a new tree
DELETE	/suffix/{tree_id}	Delete a tree
GET	/suffix/{tree_id}	Query a tree
PUT	/suffix/{tree_id}	Put a document in to a tree using the next available doc_id

Parameters

Parameter	Value	Description	Parameter Type	Data Type
tree_id	d1d8c125-2b2c-4391-a136-6d0c8924ad		path	string
payload	{ "description": "string", "document": "string" }		body	Model | Example Value { "description": "string", "document": "string" }

Parameter content type: application/json ▾

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

PUT	/suffix/{tree_id}/{doc_id}	Put a single document in to a tree

[BASE URL: / , API VERSION: 1.0]

Figure 5: Swagger interface for Suffix Tree API

5.2.1 Interface

As mentioned previously, the interface to the application is web-based. This serves two purposes; firstly it provides a unified experience across operating systems and secondly some of the processing can be memory and CPU intensive so is better suited for running on server hardware.

The interface is served using Flask for Python. Flask allows for dynamic request handling by converting URL paths directly in to function calls in Python. It also supports HTML templating via Jinja2. An module was written to act as a reverse proxy to expose various sections of the backend APIs to the browser.

The view and control components of the interface were written in HTML (using Twitters Bootstrap) and Javascript (using AngularJS, Charts.js and c3.js). AngularJS allows for two way data binding between the Broswers DOM and elements such as form inputs and renders with minimal additional code. It also provides a mechanism for sending requests to the APIs exposed by the aforementioned Proxy module. Chart.js and c3.js are two open source Javascript charting libraries that utilise features included in HTML5 used in this case for rendering observation data.

5.2.2 Data Persistence

For the persistence of Metadata about observations, detected events and Suffix Trees, PostgreSQL was selected. PostgreSQL is a performant and mature open-source Relational Database Management System (RDBMS). Being a fully featured RDBMS means it brings ACID guarantees for data resilience and transaction management.

For the persistence of Binary objects (such as Raw Observation Files and Suffix Trees), Minio was selected. Minio is an open-source object storage application written in Golang designed to emulate the abilities of Amazons Simple Storage Service (S3). It allows for arbitrary binary objects to be stored and retrieved remotely from *buckets* of grouped resources. In each bucket, an object has a unique name that can also emulate a filesystem path (e.g. *bucket*/somedir/somefile).

A wrapper class was written around Minio that provides Get, Put and Delete operations. This interface was intentionally left simple so that it could be replaced easily by another object or file store.

5.2.3 Observations API

5.2.4 SAX API

5.2.5 Suffix API

5.2.6 Deferred & Batch Jobs

For the running of deferred and batch jobs, Celery (another Python based project) was selected. It uses a message queue for receiving and dispatching jobs. Celery allows for the definition of jobs as functions which can then be imported by other processes (in this case, mostly the user interface). The application uses Redis as the message queue. It also allows for the chaining together of tasks to ensure order (e.g. event detection before SAX analysis) and the passing of results on to the next function in a functional programming style.

6 Conclusions

7 Critical Evaluation

7.1 Mistakes

7.1.1 Time-series Databases

Early on in the project, I had thought that it would be beneficial to store the raw data in a Time-Series based database such as InfluxDB, OpenTSDB or KairosDB. This would have allowed for the chaining together of events and in theory provided fast arbitrary access to data.

While this should have been the case, none of the databases mentioned were suitable. All of the open-source time-series databases seemed to be primarily focussed on the gathering of system metrics from a variety of sources and were not suitable for the large amounts of high frequency data points that came with the Seismic Data. The main sticking point with all three was that they were built for storing irregular data so each data point was stored with a timestamp meaning bytes per point and not bits.

None of my research was able to find a suitable time-series database and ultimately I settled on storing the observations in their raw format and then keeping meta data about them in an RDBMS.

7.1.2 Dataset

Initially I was provided with a large and mostly unsorted dataset (approximately 180GiB) spanning a years worth of observations from the Nabro Volcano in Eritrea. I boldly and subsequently fool-heartedly proceeded to attempt to analyse the dataset in full and with insufficient background knowledge to really understand what I was working with. This was also before I had realised the problems around normalising large datasets where most of the measurements were background noise.

Ultimately I was provided with a much smaller dataset of events to work from and this allowed my to get an actual prototype of the application running in its entirety.

Bibliography

Harris, M., M. Levene, D. Zhang, and D. Levene
2016. The anatomy of a search and mining system for digital archives.

Lin, J., E. Keogh, S. Lonardi, and B. Chiu
2003. A symbolic representation of time series, with implications for streaming algorithms. *Data Mining and Knowledge Discovery.*

M. Beyreuther, R. Barsch, L. K. T. M. Y. B. and J. Wassermann
2010. Obspy: A python toolbox for seismology.

Ukkone, E.
1995. On-line construction of suffix trees. *Algorithmica.*

Weiner, P.
1973. Linear pattern matching algorithms. *14th Annual IEEE Symposium on Switching and Automata Theory.*