An Infrastructure to Detect and Analyse Seismic Events as Strings

A dissertation submitted in partial fulfilment of the requirements for the MSc in Data Analytics

Thomas Taylor

September 2017

Department of Computer Science and Information Systems
Birkbeck College, University of London

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

List of Figures

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 would be possible 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
6. To evaluate the suitability of SAX or SAX-VSM for comparing the patterns of seismic events.

These objectives are slightly altered from the original proposal. This is discussed in section 9.1.

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.

There are three main factors that will affect the final signal recorded at a station. The first is the fault geometry itself, this means that depending on the angle of the fault. The phase and amplitude of the originating signal depend on the angle of the station relative to the fault as shown in fig. 1. The effects on amplitude should be eliminated by normalisation but the phase may require special care in pattern matching.

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 types of stations. It is also not possible to directly infer the strength of a quake from a seismogram without empirical evidence from previous events recorded at the same station.

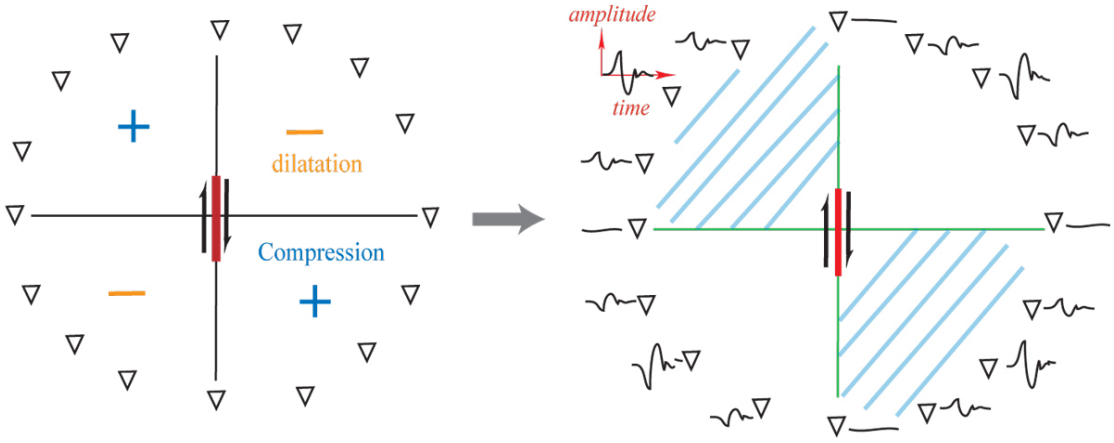


Figure 1: Effect of fault geometry on amplitude and phase (Garnero, 2017)

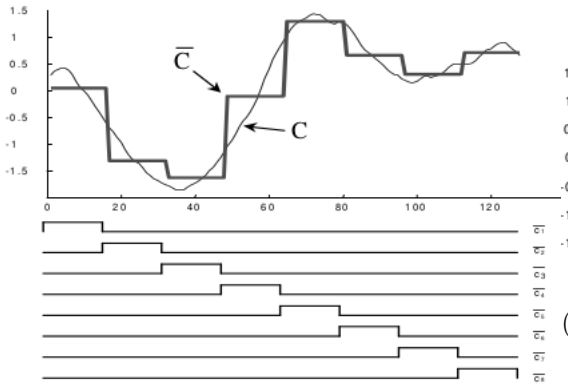
The second factor is the path effect. This is what happens to the signal as it passes through various substrates and geological boundaries. It produces a not easily predictable effect on the velocity, amplitudes and frequencies of a wave as it passes through them. In addition, the boundaries can cause reflection and refraction of the signals further distorting the recorded signal at a station.

The final factor is the frequency response and characteristics of the station itself. In the main dataset from the Nabro Volcano (Goitom et al., 2015), this can be largely ignored as the stations are all of the same make and model however it should be considered when looking at datasets involving many differing stations.

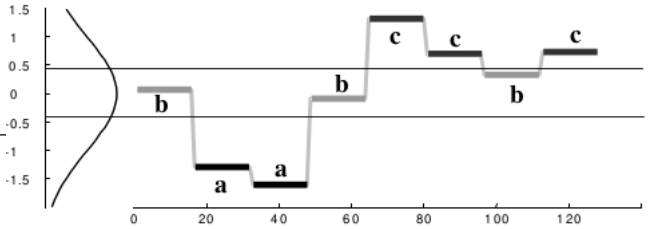
2.2 Symbolic Aggregate Approximation (SAX)

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 that could then be used for searching or 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 4.4.

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.

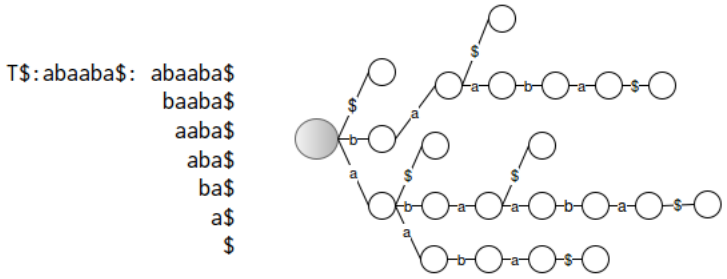


Figure 3: Example Suffix Trie

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.

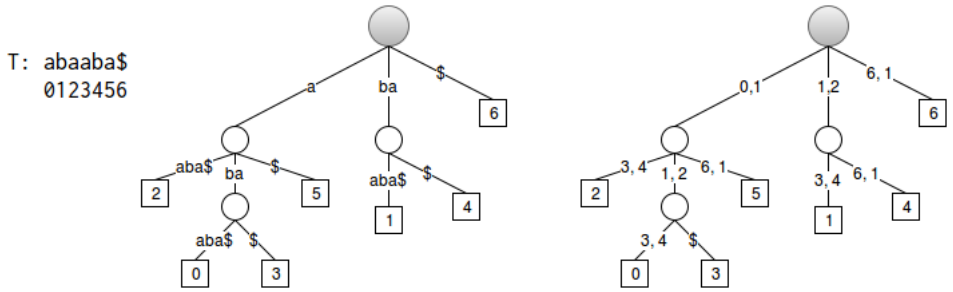


Figure 4: Example Suffix Tree (edges shown as text and as positions/offsets)

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.

2.4 Text Similarity

The following section outlines two of Similarity Measures that will be tested to try and identify patterns in the waveforms.

2.4.1 Jaccard Similarity

The Jaccard Similarity Coefficient is a measure of similarity between two sets of features. In the case of text processing, these features often words. It is defined as the intersection of features divided by their union. As a pre-processing step, the words would be converted to sets therefore removing the frequency of a word as a feature. Then common words (often called stop words) such as "the", "at", "and" are stripped out.

$$J = \frac{|A \cap B|}{|A \cup B|}$$

As an example (using integers in place of words):

$$A = \{0, 2, 3, 5, 7, 9\}, \quad B = \{0, 1, 2, 5, 6\}$$

$$A \cap B = \{0, 2, 5\}, \quad |A \cap B| = 3$$

$$A \cup B = \{0, 1, 2, 3, 5, 6, 7, 9\}, \quad |A \cup B| = 8$$

$$J_{AB} = \frac{3}{8} = 0.375$$

2.4.2 TF-IDF and the Vector Space Model

In a similar way to that of Jaccard Similarity, TF-IDF when combined with a cosine similarity can be used to give a measure of the similarity of two documents. Firstly the text is pre-processed in the same way as was done with Jaccard. That is to normalise the case, remove stop words and split on white space to produce bags of words per document. The resulting split is a *bag of words* for a given document.

The *TF-IDF* can then be calculated for a given term t by counting its occurrence in a document *tf* and multiplying by the inverse document frequency *idf* which is calculated as the number of documents N over the number of documents in the corpus containing the term N_t logarithmically scaled to dampen the effect.

$$\text{idf}_{(t,d)} = \log \frac{N}{N_t}$$

$$\text{tfidf}_{(t,d)} = \text{tf}_{(t,d)} \cdot \text{idf}_{(t,d)}$$

To produce a similarity measure, all of the *bags of words* are combined to produce a *corpus* of all of the documents in a collection and a *term frequency matrix* is calculated.

TODO

3 Plan

The following sections follow the objectives as defined in section 1.2

3.1 Importing and Storing Data

Initially the observation data is available as SAC or Miniseed files. These contain the raw observations along with metadata about the station that recorded the data. This data can be parsed with the Obspy Python library (Beyreuther et al., 2010). Use of this library produces an Python object allowing simplified access to the data and metadata from the file. 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 will be wrapped in to a Data Access Object (DAO). This is to add to or simplify many of the commonly used operations such as down-sampling, time slicing and passing through a bandpass filter to remove high and low frequencies.

For simplicity when working with many files, a service will be written to extract the metadata and store it in a database and the raw observation file will be stored in an object store for later retrieval.

3.2 Generate SAX Strings

A library and associated service will be written to carry out the normalisation of data, and to calculate PAA and to generate and return the SAX strings. In itself, this service would not persist the output but return it to the caller to either be rendered or stored.

3.3 Event Detection

Event detection will call on data provided by the SAX service and apply rules or heuristics to determine the onset (and potentially the duration) of events. This information will be persisted in a database along with the observation metadata.

3.4 Event Similarity

Two commonly used text similarity measures on the SAX generated strings will be evaluated to test similarity between events. The methods are the Jaccard Index and TF-IDF with Cosine similarity (Vector Space Model).

3.5 User Interface & Graphical Representation

An HTML/JavaScript interface will be rendered from user facing service utilising the other services and passing data back and forth using HTTP requests and JSON payloads.

The interface will facilitate the importing and storing of data, searching and rendering results. It will also allow for the submission of deferred or long running tasks.

Visuals will be rendered client-side in the browser. One or more JavaScript libraries will be utilised to show the various stages of the processing and results.

4 Data Processing

4.1 Raw Data

The seismic data is provided in either SAC or Miniseed formats that can be parsed with the Obspy Python library (Beyreuther et al., 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.

A library was written (`seismic.observations`) to wrap around the Obspy functionality combined with some additional operations specific to the project such as downsampling for rendering graphs in a browser and providing simpler interfaces to operations such as filtering on frequencies (using a Butterworth bandpass) and slicing events. The core component of the library is a class called `ObservationDAO`. This class produces an object that stores the Observation trace and metadata and provides the aforementioned methods.

4.2 Sample Data

The first dataset is from an event in California on June 22nd 1987 at around 11:00 UTC recorded from 9 nearby stations. fig. 5 shows a map of the stations extrapolated from the latitude and longitude in their metadata. This dataset will be used to train the algorithm for detecting events as they have varying amounts of signal to noise ratio.

Figure 6 shows the raw unprocessed traces from these stations (the X-axis shows seconds from 11:10:06 UTC). As can be seen from these plots, there is a high variation of background noise and little obvious correlation between the shapes of the event as observed from different stations.

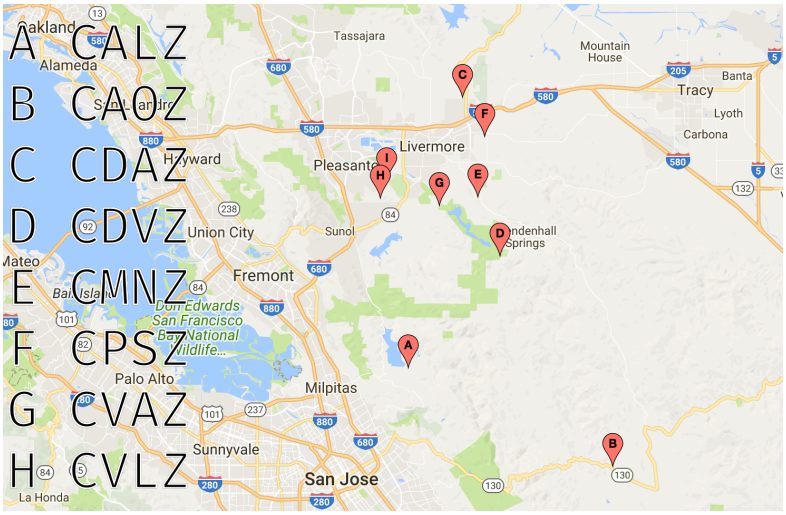


Figure 5: Map of Seismic Stations

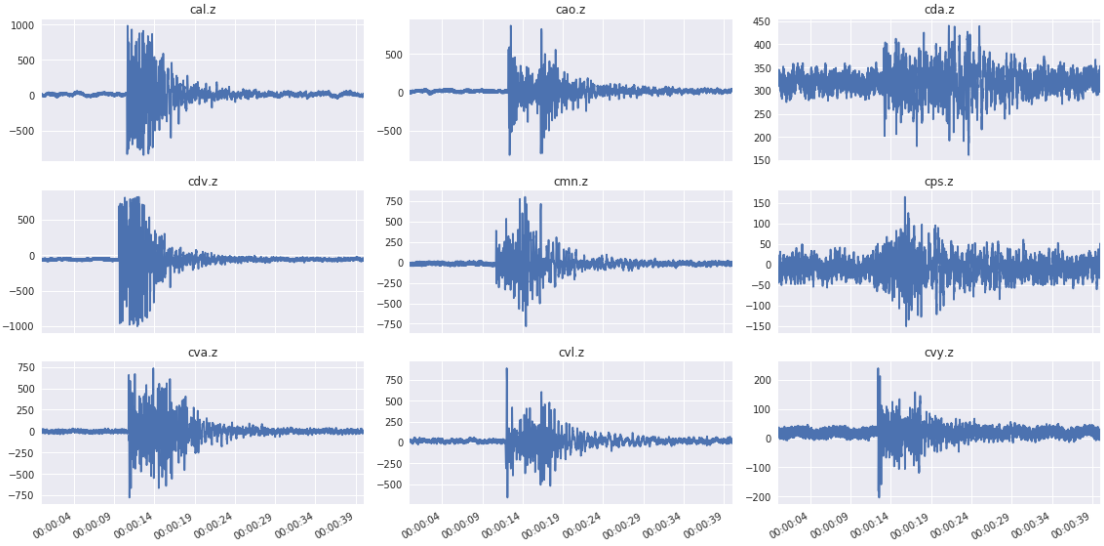


Figure 6: Raw Observation Data

The second dataset is from the Nabro volcano in Eritrea (Goitom et al., 2015). It consists of one years worth of observations from stations situated around the volcano. Volcanoes sometimes produce similar waves when the event source is the same. Within this dataset, known similarities have been provided between some events. These events (totalling 138) will be used to evaluate the similarity measures.

4.3 Frequency filtering (Butterworth Bandpass)

As seen in section 4.2 and fig. 6, raw seismic traces contain a lot of background noise. This consists of both high frequency and low frequency interference which can be caused by many unrelated events such as wind, temperature variations and human events such as traffic and construction work. A commonly used technique (Bormann, 2012) is to perform a Butterworth Filter (Butterworth, 1930) on the data, eliminating frequencies that are outside of a given range. The most significant frequencies from a seismic event tend to be between 5 and 10Hz so these values are used going forwards to clean the data. The bandpass filter is provided by the Obspy library and implemented as a method in the `ObservationDAO` class.

The effects of applying the bandpass filter can be seen in fig. 7. This results a much cleaner event profile. However, there is unfortunately still little obvious visual correlation beyond timing between the observations.

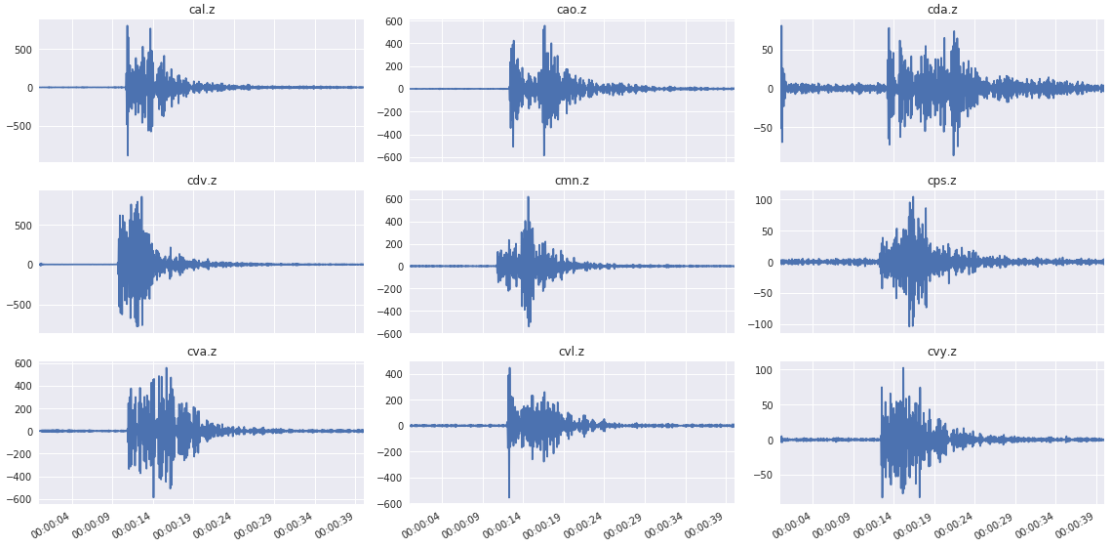


Figure 7: Frequency Filtered Observation Data (5-10Hz)

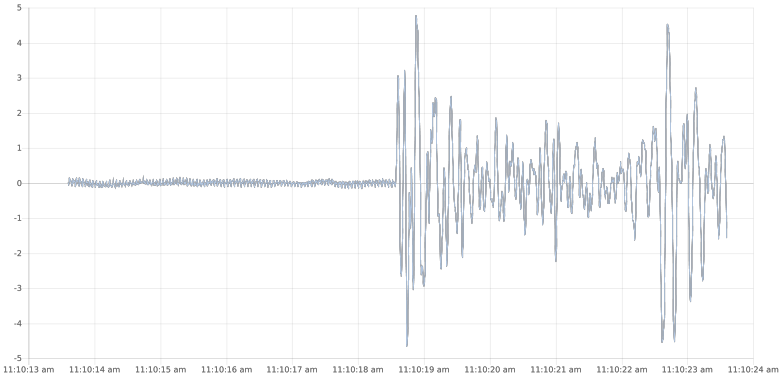
4.4 Normalisation

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.



(a) With background noise

(b) Without background noise

Figure 8: Comparison of normalisation with and without background

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. 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. 8 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.

4.5 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.

4.5.1 Data Distribution

An assumption that is made by the SAX algorithm is that data is close to normally distributed once it is *Z-normalised* (section 4.4) to get an appropriate distribution of the symbols. As can be seen from fig. 9 which shows the distribution of data points over the whole observation in blue against a normal distribution in red, the whole data set is not a good fit for a normal distribution.

However, when data is only considered during an event as seen in fig. 10 it can be seen that the data is a much closer fit. Automated event detection is discussed in section 5.

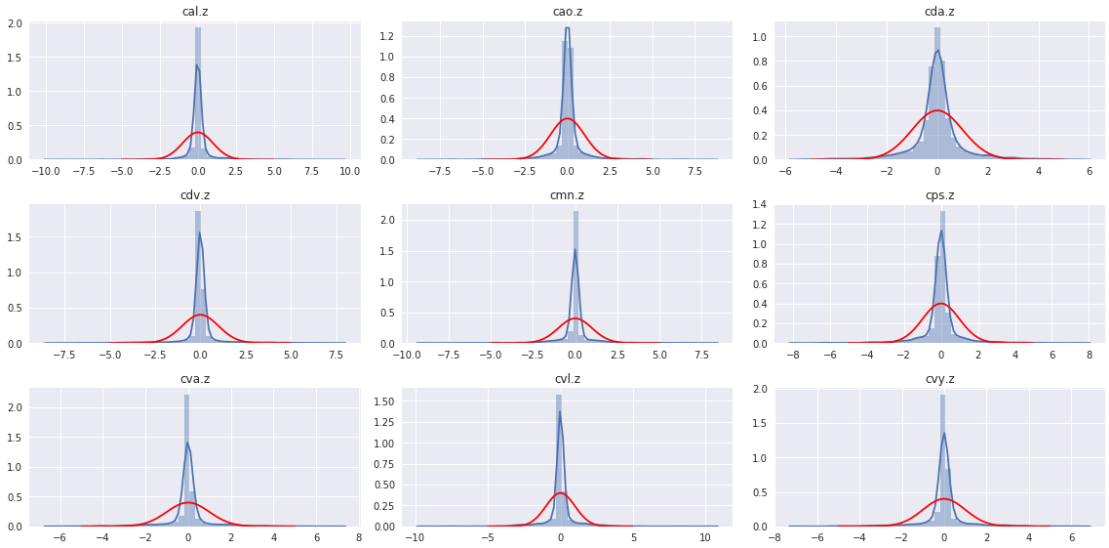


Figure 9: Distribution of Observation Data as a whole

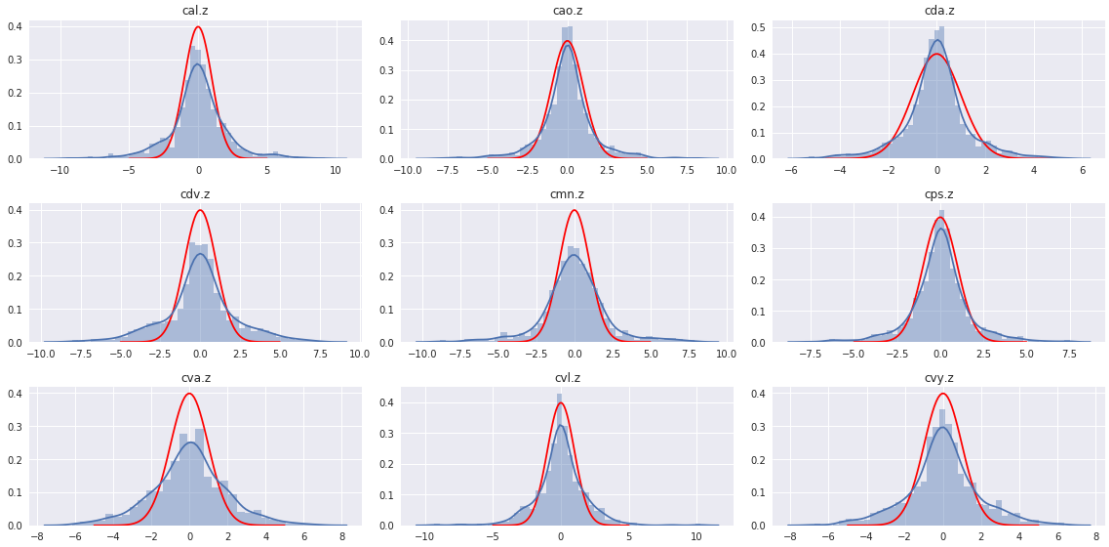


Figure 10: Distribution of Observation Data During an Event

The disparity between the distributions can be attributed to the full observation traces including many data points during quiet periods that would naturally tend towards zero. As the fit is significantly closer when only looking at data points during an event only, it suggests that a measure of the fit of a normal distribution may be a predictor to an event occurring.

4.5.2 PAA

A Python class was written to be callable to convert a Data-frame 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 by passing `normalise=False` as a parameter).

```
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").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 4.4.

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
```

4.5.3 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)]

    def to_string(self, alphabet):
        return "".join([i for i in self.__call__(alphabet)])
```

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. The length of the alphabet 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 call to print the characters is demonstrated 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)
```

4.6 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 so again a Butterworth Bandpass (Butterworth, 1930) was applied eliminating frequencies outside of the 1-20Hz range. Code was then written to interpolate the modulation in frequency by means of differentiating the phase inversions. The code consisted of three functions, *frequency*, *phase_inversions* and *zero_intersect*.

The *frequency* function takes two parameters, the first being a series (or array) of observation values and the second being the interval in millisecods between the observations (it assumes evenly timed datapoints). The series is fed through the phase inversions function that scans the series for the value moving from positive to negative or vice versa. When it finds an inversion, it passes the Z values from either side of the inversion to *zero_intersect* which liniearly interpolates the point at which it crossed the x-axis. This is the passed back up to the frequency function which divides 500 (to convert a half cycle in milliseconds to cycles per second) by the distance between the two observations

to estimate the frequency and returns this value along with the midpoint between the timestamps.

The results of this for four events recorded at four stations are shown below (events in columns, stations in rows with the x-axis being the time in milliseconds since the detected start and the y-axis showing interpolated frequency in Hz.)

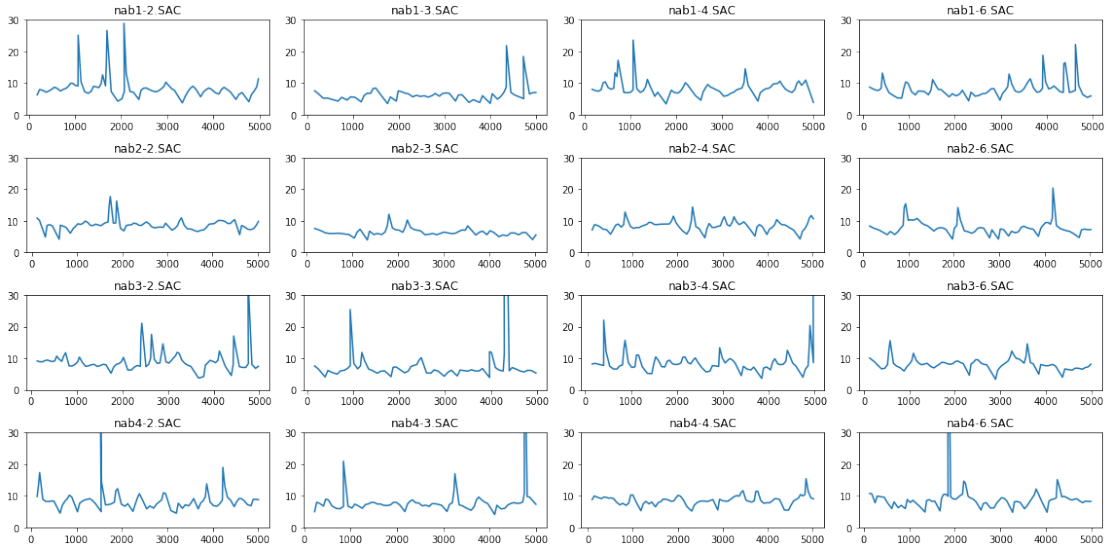


Figure 11: Interpolated Freqencies

As when looking at the amplitudes in section 4.2, there is no obvious correlation between the stations for the same event. This was further validated when applied to the same tests as the SAX analysis of the raw signal in sections 6.2 and 6.3. It was decided not to investigate further.

5 Event Detection

The primary intent of the project is to be able to use the data generated by the *SAX* algorithm to detect events in a measurement over a period of time. One existing common technique for doing this is STA/LTA (Short Term Average/Long Term Average) (Bormann, 2012). STA/LTA is a sliding window technique where the mean of the absolute amplitudes are compared ofver two different sized windows. If the STA exceeds the LTA by a preset threshold then an event is triggered, if it drops below another threshold the trigger is ended. The event is considered the duration between these two triggers.

In order to evaluate a SAX based detection algorithm, an implementation of STA/LTA was required for comparison Traditionally STA/LTA thresholds are set manually by experienced seismologists, in this case however, the author does not have the required experience to set these values so an adaptive method was required.

Both event detector algorithms were written against a common abstract base class to ensure that they resulted in similar interfaces and provide common methods and exceptions. The listing for this interface is too long for inclusion but can be found in the module at `seismic.detector.detectorbase`.

5.1 Adaptive STA/LTA Algorithm

N.B. This implementation has only been validated on a relatively small sample set and is not recommended for real-world use without further testing.

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

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

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

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 algorithm is considered adaptive by using the number of standard deviations as opposed to pre-set trigger values.

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 code is listed in appendix B.1.

One 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. It also struggles to detect an event when there is significant background noise.

5.2 SAX Detection Algorithm

The *SAX Detection Algorithm* works by firstly calculating a SAX string for the whole observation. It requires a minimum *trigger on* length in milliseconds (to eliminate spikes), a quiet duration (again in milliseconds) and a distance from the centre value of the alphabet used to produce the SAX string. This string is then iterated over one character at a time filling a ring buffer that is at least the length of the *trigger on* or the *quiet duration* that is used to establish whether trigger requirements are met. It is effectively an implementation of Step Detection using the breakpoints defined in the SAX process.

The code is listed in appendix B.2.

5.3 Results

Both algorithms were run against the California earthquake data as show in fig. 6 in section 4.2. The following table shows the event detection results of the StaLta algorithm compared to the SAX detect algorithm.

Observation	StaLta Detect	SAX Detect	Difference(s)
CAL.Z	11:10:16.981	11:10:17.056	0.075
CAO.Z	11:10:18.599	11:10:18.656	0.057
CDA.Z	n/a	11:10:19.556	
CDV.Z	11:10:15.973	11:10:16.056	0.083
CMN.Z	11:10:17.083	11:10:17.206	0.123
CPS.Z	11:10:18.619	11:10:18.656	0.037
CVA.Z	11:10:17.194	11:10:17.256	0.062
CVL.Z	11:10:18.324	11:10:18.406	0.082
CVY.Z	11:10:18.762	11:10:18.806	0.044

On one occasion, the StaLta detector failed to detect an event. On all of the others, the SAX detector found the same event albeit marginally later. The mean difference

was 0.07s and was always in favour of the STALTA algorithm. Over longer observation periods with multiple events, the SAX based algorithm failed to find many events. This is presumably because larger events will dampen the effect of the smaller ones which will then be disregarded as background noise.

5.4 Approximate Detection of Events by Distribution

Following on from the discussion in section 4.5.1 and based on a hypothesis that background noise can be considered random, with a high enough number of samples would it follow the Central Limit Theorem, it follows that only when a sample contains both background noise and an event would its normalised distribution deviate significantly from a normal one.

By applying the SAX algorithm to a sample of data, it has effectively been bucketed against a normal or Gaussian distribution. It therefore leads that the distribution of counts of each symbol in the sample provide an approximate measure of how close the sample is to being normally distributed.

A function was written to sample a window of length w, perform PAA and SAX on it and calculate the standard deviation normalised by the number of samples over the number of symbols. The window is then shifted to the right by $\frac{w}{2}$ places run again. This is repeated for the whole observation. The effectiveness of this method is shown in fig. 12. The top plot shows a raw trace over a 24 hour period for the station YW.NAB1 at the Nabro Volcano in Eritrea on 2011-08-27 (Goitom et al., 2015) and the bottom plot shows the aforementioned sliding window SAX distribution function with a w of 300s, a PAA interval of 50ms and an alphabet of length 7 with a trigger value set to 0.25.

It can be seen from the two plots imposed next to each other that the spikes from the SAX distribution function correlate almost perfectly with the visible events on the trace, albeit at a much lower resolution (2.5 minutes as opposed to 0.1 seconds in this case).

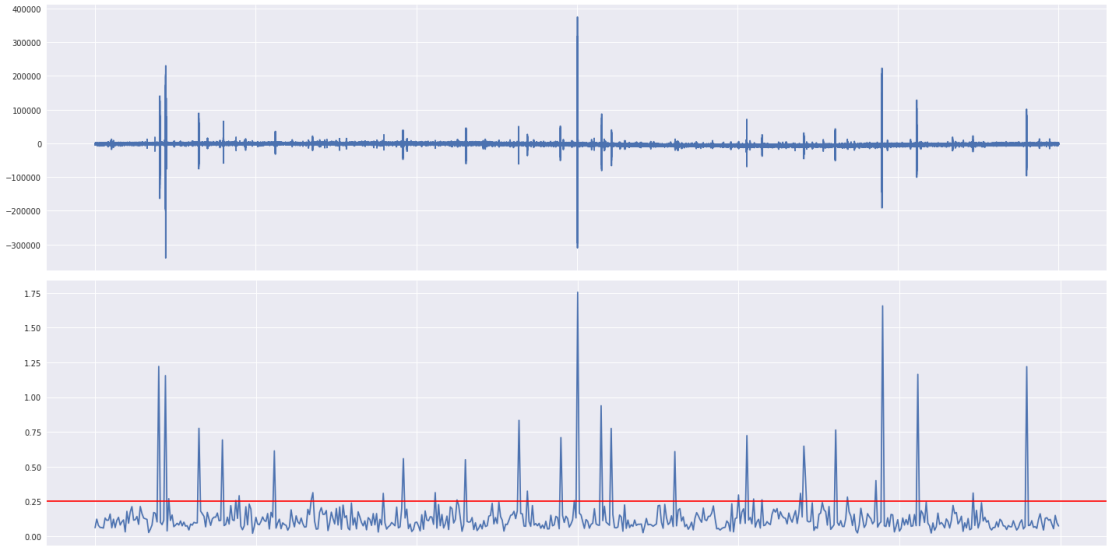


Figure 12: Using SAX distribution for approximate event finding

5.5 Combining the Approximate Detection with the SAX detection

The SAX distribution approximation function returns probable events in a 5 minute window. These windows can then be passed to the SAX Detection Algorithm described in section 5.2 which has been shown to be effective over shorter windows. Computationally, the combined process takes approximately 10 seconds to run over a 24 hour period with a sampling rate of 100Hz on a single core of an Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz once the whole observation is loaded in to memory as a Pandas Series object. Although by the nature of Python's threading model it cannot be optimised for multiple cores for a single observation, there is no reason why it cannot be run in parallel against multiple observations. This is the model that the worker process described in section 7.5 takes.

It is difficult to accurately measure the effectiveness of the combined algorithm without more domain specific knowledge. The algorithm did pick up many events (137 over a 24 hour period) but was also easily confused by lower energy longer duration events as was common with the volcanic data. Also there were some events that were not detected.

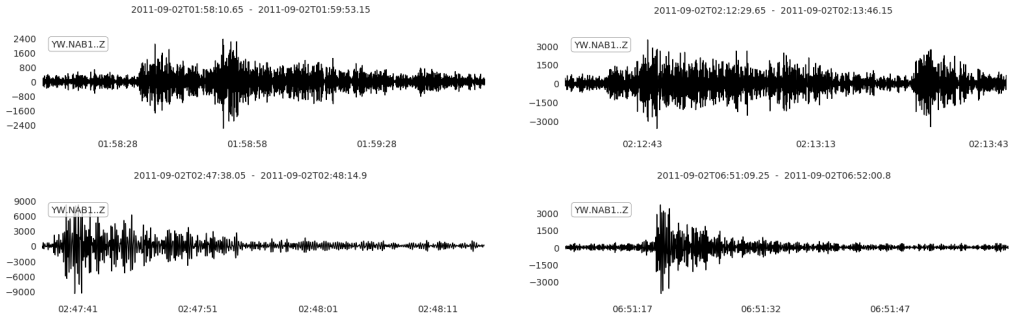


Figure 13: A Selection of Detected Events by the Combined Algorithm

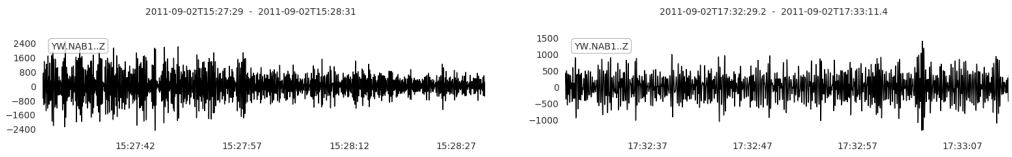


Figure 14: A Selection of False positives by the Combined Algorithm

6 Event Similarity

6.1 Bag of Words

Many text similarity models rely on having a collection of words to be treated as features. They often delimit text by white-space, removing punctuation and normalising to a single case. The output strings from the SAX algorithm have none of these features and are effectively a single long word so this approach of splitting isn't appropriate

One approach to this is to take a sliding window of length w to build to vocabulary (Keogh et al., 2007). This was implemented in Python as follows:

```
def bag_of_words(s, w):
    """
    Takes a string s and returns a sliding window array of substrings of
    length w

    Args:
        s: string
        w: length of word

    Returns:
        List
    """
    bow = [None] * (len(s) - w + 1)
    for i in range(0, len(s) - w + 1):
        bow[i] = s[i:i + w]
    return bow
```

For example, given an input string s of `abcdefghijkl` and a word length $w = 5$, the function returns the list:

```
['abdce', 'bdcef', 'dcefg', 'cefgh', 'efghi', 'fghij', 'ghijk',
'hijkl']
```

Test cases for this function are included in appendix C.1.

6.2 Jaccard Similarity

6.2.1 Implementation

The Jaccard Similarity Coefficient was implemented in Python as follows (with the case two sets with a cardinality of 0 returning 1).

```
def jaccard(a, b):
    """
    Calculate the Jaccard Similarity Coefficient of two input sets

    Args:
        a: First set
        b: Second Set

    Returns:
        Float
    """
    a = set(a)
    b = set(b)
    union = float(len(a.union(b)))
    intersection = float(len(a.intersection(b)))
    return intersection / union if (union + intersection) > 0 else 1
```

Test cases are included in appendix C.2. The inputs to the function are the bag-of-words produced from two observations.

6.2.2 Results

The Jaccard similarity was run against 138 events with known similarities from the Nabro 2011 data for 4 stations (NAB1..4). For each event at a given station, a SAX string was produced with the length of the alphabet and the PAA interval as varying parameters between runs. This was then run through the *bag_of_words* function with varying word lengths. In differing runs, the PAA Interval was set to 5, 10, 25 and 50ms, the alphabet was varied from a length of 4 to 9 characters and the length of the words were set to 5 to 10.

For each run, a similarity matrix was produced comparing every pair of events and the similarities ranked. This was then aggregated into a single data-frame and the ranks and average ranks compared to whether this was a known similar event. The results for the following parameters are shown below (these parameters showed the best case for matching known events):

```
paa_int = 10
alphabet = "abcde"
word_len = 5
```

	Jaccard Index					Rank in Cohort				
Match	NAB1	NAB2	NAB3	NAB4	μ	NAB1	NAB2	NAB3	NAB4	μ
0	0.38	0.31	0.28	0.33	0.33	2	3	4	1	2.5
1	0.46	0.40	0.50	0.32	0.42	1	3	6.5	2	3.125
0	0.33	0.43	0.50	0.36	0.40	6	1	6.5	1	3.625
1	0.46	0.40	0.50	0.32	0.42	2	3	6	4	3.75
0	0.33	0.43	0.50	0.36	0.40	8	1	5	1	3.75
0	0.28	0.44	0.62	0.29	0.41	5	2	2	7	4
0	0.37	0.34	0.29	0.23	0.31	2	1	4	10	4.25
0	0.38	0.31	0.28	0.33	0.33	6	1	9	2	4.5
1	0.22	0.58	0.14	0.25	0.30	1	1	15	1	4.5
0	0.38	0.38	0.36	0.43	0.39	6	2	10	1	4.75
0	0.34	0.35	0.38	0.23	0.33	10	2	4	4	5
0	0.43	0.29	0.27	0.23	0.30	2	9	5	5	5.25
1	0.36	0.32	0.29	0.24	0.30	3	5.5	5	8	5.375
0	0.12	0.30	0.23	0.17	0.20	2	4	13	3	5.5
1	0.22	0.58	0.14	0.25	0.30	1	1	19	1	5.5
0	0.29	0.43	0.67	0.23	0.40	3	2	3	14	5.5
0	0.33	0.28	0.27	0.36	0.31	6	8	7	2	5.75
0	0.28	0.26	0.23	0.25	0.25	1	4	6	15	6.5
0	0.23	0.38	0.33	0.27	0.30	14	2	5	5	6.5
0	0.37	0.35	0.40	0.19	0.33	2	1	5.5	18	6.625

The table shows the 20 most similar pairs of events sorted by their average rank across the four stations. The *Match* column shows whether or not the event was a known match. As can be seen, while some of the matched events score quite highly, there are also many highly scored false positives. The rankings are of a maximum of 137. The mean rank for a known match was 48.8 ($\sigma = 24.7$) and a known non-match was 69.3 ($\sigma = 23.7$) of 137. Approximately 69 would be the expected average rank for a non match as it is close to the median rank. While matched events were more likely to be considered of higher similarity using this method, the effect is only marginal and therefore not a suitable predictor by itself.

6.3 Vector Space Model (TF-IDF)

6.3.1 Implementation

To implement the functionality of TF-IDF and the Cosine similarity functions in a high level language such as Python and in a performant way would have taken a significant amount of work. It has also already implemented and well tested in many open source libraries and for performance reasons is normally constructed in a lower level language such as C. It was decided for this project to use an open source library to use a library called Gensim (Řehůřek and Sojka, 2010) that has a fairly sensible API and decent documentation.

The following code snippet shows this library being used with the same similarity matrix as the one produced to evaluate the *Jaccard Index* in section 6.2. The full code can be found in a Python Notebook under `/notebooks/TFIDF.ipynb` where an aggregate matrix of all of the stations is produced to calculate the combined ranking.

```
# Get a list of events
keys = sorted(sax[st].keys())
# Create a Dict of Matrices per station
tm = {s: SimilarityMatrix(siml.keys()) for s in stations}

for st in stations:
        raw_corpus = [sax[st][doc] for doc in keys]
        dictionary = corpora.Dictionary(raw_corpus)
        corpus = [dictionary.doc2bow(t) for t in raw_corpus]
        lsi = models.LsiModel(corpus, id2word=dictionary, num_topics=2)

        for i in range(len(keys)):
                qry = dictionary.doc2bow(raw_corpus[i])
                vec_qry = lsi[qry]
                index = similarities.MatrixSimilarity(lsi[corpus])
                sims = index[vec_qry]
                for j in range(len(sims)):
                        tm[st].put(keys[i], keys[j], sims[j])
```

As with the *Jaccard Index*, a full range of PAA intervals, alphabet and word lengths were tested.

6.3.2 Results

The following shows a table produced in the same way as that for the *Jaccard Index*. The best scoring parameters were found to be:

```
paa_int = 10
alphabet = "abcde"
word_len = 7
```

	Cosine Similarity					Rank in Cohort				
Match	NAB1	NAB2	NAB3	NAB4	μ	NAB1	NAB2	NAB3	NAB4	μ
1	0.999999	1.000000	0.999999	1.000000	1.00	8	2	5	2	4.25
1	0.999999	1.000000	0.999999	1.000000	1.00	6	4	6	2	4.5
0	0.999454	0.998098	0.999893	0.999951	1.00	5	5	4	5	4.75
0	0.999454	0.998098	0.999893	0.999951	1.00	13	3	4	7	6.75
0	0.999502	1.000000	0.990480	0.999772	1.00	8	2	11	14	8.75
0	0.999944	0.997661	0.979703	0.999969	0.99	4	7	20	6	9.25
0	0.967376	0.999938	0.999362	1.000000	0.99	16	6	15	1	9.5
0	0.999905	0.998278	0.986593	0.998055	1.00	5	13	18	2	9.5
0	0.982158	0.999804	0.999969	0.999996	1.00	23	9	3	4	9.75
0	0.999905	0.998278	0.986593	0.998055	1.00	10	13	16	1	10
0	0.999944	0.997661	0.979703	0.999969	0.99	3	6	18	13	10
0	0.999501	0.999821	0.999996	0.995503	1.00	1	18	4	18	10.25
0	0.982158	0.999804	0.999969	0.999996	1.00	26	10	2	5.5	10.875
0	0.991812	1.000000	0.999172	0.999994	1.00	10	2	14	17.5	10.875
0	0.999949	0.996722	0.999966	0.999383	1.00	19	12	2	11	11
0	0.967376	0.999938	0.999362	1.000000	0.99	4	2	38	2	11.5
0	1.000000	0.892858	0.999272	0.999998	0.97	1.5	35	5	5	11.625
0	0.999991	0.985588	0.981926	0.999999	0.99	15	11	18	3	11.75
0	0.982587	1.000000	0.998375	0.999994	1.00	22	1	12	12	11.75
0	0.996410	0.957936	0.999225	0.972952	0.98	6	19	3	20	12

Interestingly, the performance of the Vector Space Model was even worse than the Jaccard Index in terms of accuracy. Computationally it also took approximately 6 times more CPU time. The mean ranking for a match was 57.0 ($\sigma = 22.0$) and a non match was 69.1 ($\sigma = 19.4$). Again, on average, the rank of a known match was higher but it was again marginal so wouldn't be suitable as a predictor by itself.

7 Application Design

7.1 Discrete Services

The application was decided to be split into discrete components with an all of the data processing bundled together behind a single API exposed via HTTP with mostly JSON payloads (the observation data is transferred and stored in its original binary format). The interface is a separate service that is mostly an HTML and JavaScript powered application that speaks to the API on the HTTP/JSON interface. A third instance of the application runs to carry out long running or deferred tasks and is communicated with via Redis as a message queue. Additionally data persistence services (see section 7.3) run separately.

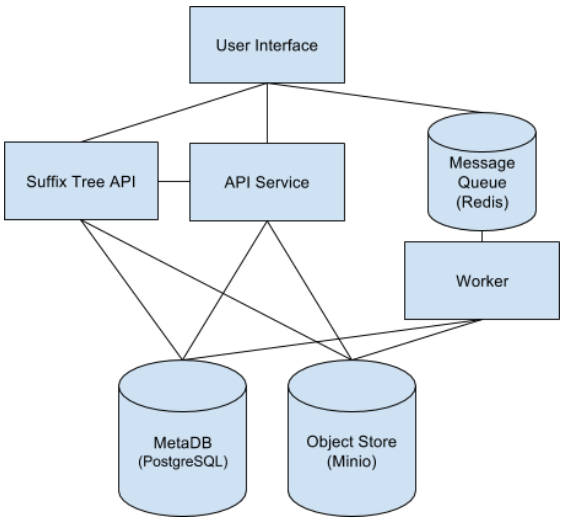


Figure 15: Application Architecture

7.2 Data processing API

As previously mentioned, the data processing functions are bundled in to a single API process broken in to separate namespaces for the different parts.

The *observations* namespace used to manipulate observation files. It supports a REST like interface for CRUD (Create, Read, Update and Delete) operations and RPCs (Remote Procedure Calls) for running event detection and retrieving results.

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

The HTTP/JSON APIs were written using Flask-Restplus (Haustant, 2016), 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 and provides a Swagger interface. Swagger is a commonly used standard for defining APIs. It 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. 16.

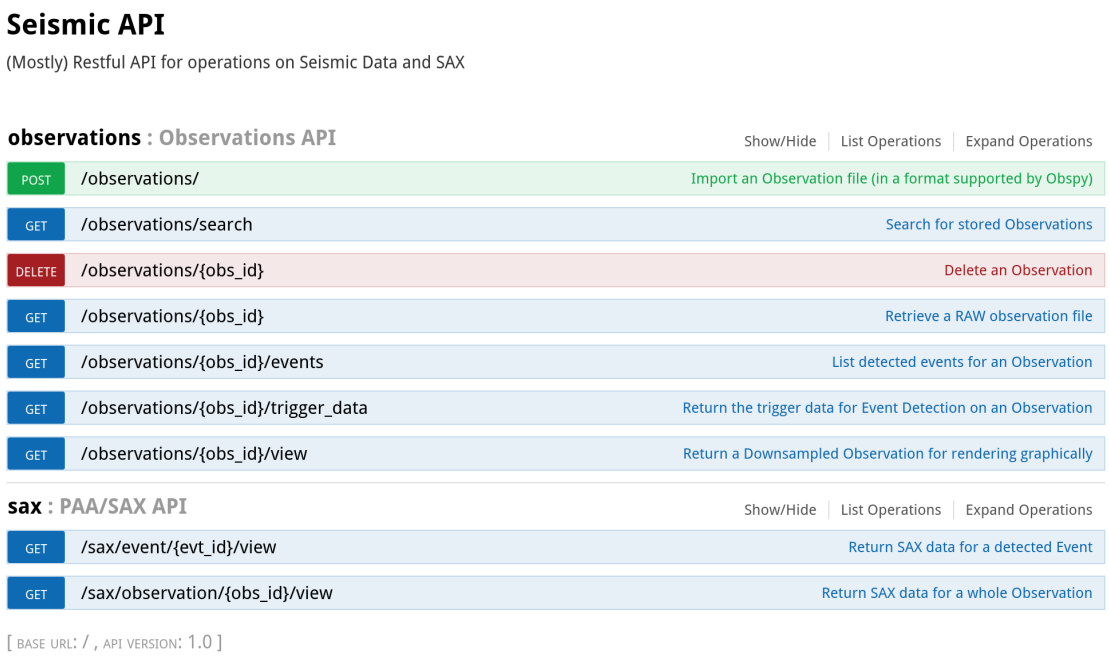


Figure 16: Swagger interface for Data Processing API

7.2.1 Observations API Namespace

TODO

7.2.2 SAX API Namespace

TODO

7.3 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 Rela-

tional 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 substituted easily by another object or file store.

7.4 User Interface

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.

As previously mentioned, 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. It also facilitates the potential sharing of data between users.

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 Twitter's Bootstrap) and JavaScript (using AngularJS, Charts.js and c3.js). AngularJS allows for two way data binding between the browser's 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.

7.5 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.

7.6 Suffix API

The Suffix API runs as a separate service as the majority of the code base is taken from previous work on suffix trees at Birkbeck (Harris et al., 2016). This was done mainly to separate it from the original work on this project. The only additions were an HTTP API written in the same style as the Data Processing API so that it could be interacted with. The API operations are summarised in fig. 17.

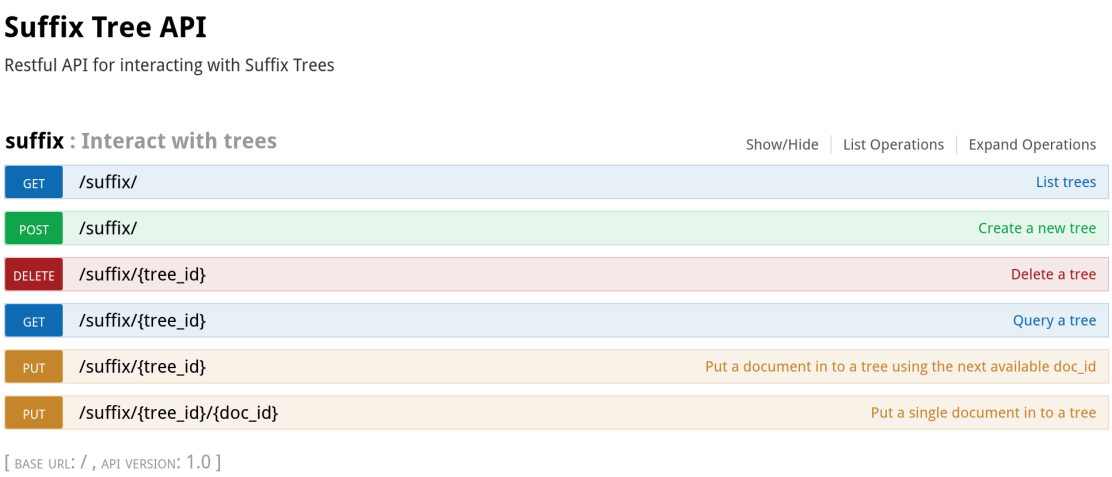


Figure 17: Swagger interface for Suffix Tree API

8 Conclusions

8.1 Event Detection

The SAX detection algorithm performed quite well on an observation with a single event, often within a few milliseconds of the known onset. Applied naively to a longer time period with multiple events it was much less reliable. When combined with the distribution function its performance was fair, it detected many events but still missed quite a few and picked up many false positives.

There are not sufficient patterns in the produced SAX string beyond the step change and the change in character distribution that can be used to reliably determine when an event is happening.

While the analysis of the strings is fast, the pre-computation during the PAA phase (the normalisation and aggregation) must still be factored in with the consideration that the string is only to be used once and then discarded as the PAA needs to be recalculated once the event has been isolated.

In conclusion, it is only performing a step change algorithm with the steps being predefined by the Gaussian break points as determined by the SAX algorithm. It is not felt that it has anything to add over existing step change based algorithms such as *Short Term Average Long Term Average* (STALTA) already in use in the field.

8.2 Event Similarity

Ultimately SAX was found not to be suitable for detecting similarities between known similar seismic events. This is presumably down to the factors listed in section 2.1, specifically the path effect. That is the reflection and refraction of the waves as they pass through different substrates and geological boundaries causes a significant amount of noise that distorts the waveform beyond it being easily recognisable. While the string similarity measures (Jaccard and TF-IDF) showed on average a slightly higher score for known matched events, the highly scoring false positives rendered the positive results statistically insignificant.

9 Critical Evaluation

9.1 Divergence from Original Objectives regarding Suffix Trees

The original objectives had focussed on building SAX strings suitable for storage and interpretation using Suffix Trees. The difficulty in finding patterns between different observations of the same event (as discussed in the latter part of section 2.1) was not picked up during the early research stages. Once this had been established, it was felt that Suffix Trees would not allow enough flexibility in detecting similarities in event data and the text similarity models were considered instead.

9.2 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 observations and in theory provided fast arbitrary access to data.

While this should have been the case, none of the databases investigated 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 stumbling 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 resulting in storage requirements many times bigger than the original data and not sufficiently fast access. 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 metadata about them in an RDBMS.

9.3 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.

9.4 Approach

I feel I started work on the supporting application far too early without having first established a minimum viable prototype with regards to the event detection and the similarity measures. This was probably down to a lack of experience with the statistical and graphing libraries in Python and I had felt that the only way to visualise the data as I wanted to see it was to write the application to support it first. In hindsight, all of the tooling I needed was there all along. Had I focussed more time on the models and algorithms and established the problems earlier on, I would have spent less time designing the application and possibly found more effective techniques.

9.5 Domain Specific Knowledge

I went in to this project with only a passing knowledge of seismic waves. The reasons behind the lack of patterns as described at the end of section 2.1 was something that I only discovered towards the end of the project. Had this been known at the start then I may well have taken a completely different approach or been based on another subject entirely.

9.6 In conclusion

I learned a lot from working on the project, not just about Geophysics and the propagation of seismic waves, but also about development practices. Test driven development (TDD) only works when you already have an expectation of an outcome and is not necessarily suited to experimental work. It is important to focus on a working prototype or minimal viable product before attempting to piece together an entire project.

A Building and Running the Application

A.1 Building

This step is not strictly necessary to run the application as it is available as pre-built docker images but is included for completeness. To build (probably) requires a Unix like environment (e.g. Linux or Mac OS), Python 3.5+ with pip, GNU make and docker.

```
$ cd seismic-detector
$ python3 -m virtualenv .
$ source bin/activate
$ pip install numpy
$ make clean sdist docker
```

A.2 Running

To run the application pre-built requires docker and docker-compose. It is hosted on the Docker Hub so it does not need to be built first.

From the root of the project (on writeable media such as a hard disk), invoke `docker-compose up -d`.

```
$ docker-compose up -d
docker-compose up -d
Creating network "seismic_default" with the default driver
Creating seismic_api_1
Creating seismic_minio_1
Creating seismic_postgres_1
Creating seismic_redis_1
Creating seismic_worker_1
Creating seismic_interface_1
```

The application should now be available via a browser on `http://localhost:8080`. To terminate, call `docker-compose down`.

B Detectors

B.1 STA/LTA

```
from logging import debug, info
import numpy as np
import pandas as pd

from .detectorbase import DetectorBase


class StaLtaDetect(DetectorBase):
    def __init__(self, trace, sampling_rate):
        """
        Create a new Detector object

        Args:
            trace (np.ndarray): raw trace
            sampling_rate (int): Sample rate of trace in Hz
        """
        super().__init__(trace, sampling_rate)
        self.trigger_values = None

    def detect(self, short, long, nstds=1, trigger_len=5000):
        """
        Run a hackish STA—LTA like detection on trace. Looks for short
        window means above nstds standard deviations from the long window
        mean for more than trigger_len observations.

        Args:
            short (int): number of ms for short window
            long (int): number of ms for long window
            nstds (float): number of standard deviations from long window
                mean to trigger on
            trigger_len (int): length in ms of short mean observations
                being above nstds to trigger on

        Yields:
            (time from start, long window mean, short window mean)
        """
        trigger_values = []

        long_win_len = int(long / self.interval)
```

```
short_win_len = int(short / self.interval)
iter_len = 1
# Convert from ms to number of obs
trigger_len = int(trigger_len / self.interval)
debug("Window␣lengths:␣{},␣{}␣({}s,␣{}s)".format(
    long_win_len, short_win_len, long/1000, short/1000))
i = long_win_len + short_win_len
triggered = False
off_threshold = 0
triggered_obs = 0
while i + short_win_len < len(self.trace):
    long_window = self.abs[
        i - long_win_len - short_win_len:
        i - short_win_len
    ]
    long_win_mean = np.mean(long_window)
    long_win_std = np.std(long_window)
    short_win_mean = np.mean(
        self.abs[i - short_win_len:i])
    trigger_values.append((
        i * self.interval,
        short_win_mean,
        long_win_mean,
        long_win_std * nstds
    ))
    if not triggered:
        trigger_val = long_win_mean + long_win_std * nstds
        if short_win_mean > trigger_val:
            off_threshold = long_win_mean
            triggered = True
            triggered_obs = 1
    else:  # if triggered
        triggered_obs += iter_len
        if short_win_mean < off_threshold:  # trigger over
            triggered = False
            if triggered_obs > trigger_len:
                yield(
                    int(i - triggered_obs - (short_win_len / 2)) *
                        self.interval,
                    i * self.interval
                )
            triggered_obs = 0
    i += iter_len
# Get trigger values
```

```
self.trigger_values = pd.DataFrame(
    trigger_values,
    columns=("t", "sm", "lm", "trigger"),
)
```

B.2 SAX

```
import types
from collections import deque
import numpy as np
import pandas as pd

from seismic.sax import Paa, PaaError, Sax, SaxError
from .detectorbase import DetectorBase
from .exceptions import DetectorError


class SaxDetect(DetectorBase):
    def __init__(self, trace, sampling_rate):
        """

        Args:
            trace:
            sampling_rate:
        """
        super().__init__(trace, sampling_rate)

    def detect(self, alphabet, paa_int, off_threshold=5000, min_len=5000):
        p = Paa(self.series)
        s = Sax(paa=p(paa_int))
        for start, end in sax_detect(s(alphabet), alphabet, paa_int,
            off_threshold, min_len):
            yield start, end


def near_centre(alphabet, centre, max, s):
    """
    Returns whether a character is less than max from the centre value of
        a string

    Args:
        alphabet (str): full alphabet to compare from
        centre (str): centre value
        max (int): maximum distance to consider
        s (str): search character

    Returns:
        bool
    """
```

```
    if centre not in alphabet:
        raise DetectorError("{} was not found in {}".format(centre,
            alphabet))
    if s not in alphabet:
        raise DetectorError("{} was not found in {}".format(s, alphabet))
    return s == centre or abs(alphabet.find(centre) - alphabet.find(s)) <=
         max


def sax_detect(stream, alphabet, paa_int, off_threshold=5000, min_len
    =5000):
    """

    Args:
        stream (types.GeneratorType): Generator of SAX string
        alphabet (str): Alphabet used to create SAX string
        paa_int (int): PAA Window size in ms
        off_threshold: Quiet period to consider event finished
        min_len: Minimum length of an event to yield

    Yields:
        (start_ms, end_ms)
    """
    if not isinstance(alphabet, str):
        raise DetectorError(
            "SaxDetect alphabet expects a str, got {}".format(
                type(alphabet)))
    if len(set(alphabet)) != len(alphabet):
        raise DetectorError("Alphabet must only contain unique characters"
            )
    if len(alphabet) % 2 == 0:
        raise DetectorError(
            "SaxDetect requires an odd length of alphabet to have a centre
                ")
    # Convert the next two values from ms to number of elements
    min_len = int(min_len / paa_int)
    off_threshold = int(off_threshold / paa_int)
    # Ring buffer for looking back at recent values
    buffer = deque([], maxlen=off_threshold)
    # Centre value to calculate distance from
    centre = alphabet[int(len(alphabet) / 2)]
    i = 0  # current pos
    t_on = 0  # trigger on value
    triggered = False
```

```
for s in stream:
    buffer.appendleft(s)
    if not triggered:
        if not near_centre(alphabet, centre, 1, s):
            triggered = True
            t_on = i
    elif triggered:
        if near_centre(alphabet, centre, 1, s):
            for j in range(1, off_threshold):
                if not near_centre(alphabet, centre, 1, buffer[j]):
                    break
            else:
                triggered = False
                if (i - off_threshold) - t_on >= min_len:
                    yield t_on * paa_int, (i - off_threshold) *
                        paa_int
    i += 1
```

C Sample Tests

C.1 Bag of Words

Test cases for the *bag_of_words* function described in section 6.1.

```
import unittest
from .. import bag_of_words


class BagOfWordsTest(unittest.TestCase):
    def setUp(self):
        self.sample = "abdcefghijkl"

    def test_w_1(self):
        bow = bag_of_words(self.sample, 1)
        self.assertEqual(bow[-1], "l")
        self.assertEqual(len(bow), len(self.sample))

    def test_w_3(self):
        bow = bag_of_words(self.sample, 3)
        self.assertEqual(bow[-1], "jkl")
        self.assertEqual(len(bow), 10)

    def test_w_5(self):
        bow = bag_of_words(self.sample, 5)
        self.assertEqual(bow[-1], "hijkl")
        self.assertEqual(len(bow), 8)
```

C.2 Jaccard

Test cases for the Jaccard Index function in section 6.2.

```
import unittest

from ..jaccard import jaccard


class JaccardTest(unittest.TestCase):
    def test_len_zero(self):
        a = []
        b = []
        self.assertEqual(jaccard(a, b), 1)

    def test_similar(self):
        a = [1, 2, 3]
        b = [3, 2, 1]
        self.assertEqual(jaccard(a, b), 1)

    def test_dissimilar(self):
        a = [1, 2, 3]
        b = [4, 5, 6]
        self.assertEqual(jaccard(a, b), 0)

    def test_score(self):
        a = [0, 1, 2, 5, 6]
        b = [0, 2, 3, 5, 7, 9]
        self.assertEqual(jaccard(a, b), 0.375)

    def test_score_reversed(self):
        a = [0, 2, 3, 5, 7, 9]
        b = [0, 1, 2, 5, 6]
        self.assertEqual(jaccard(a, b), 0.375)

    def test_strings_with_repetition(self):
        a = "abcfggg"
        b = "aacdfhj"
        self.assertEqual(jaccard(a, b), 0.375)
```

C.3 Detector

Test cases for both the SAX detector in section 5.2 and the STA-LTA implementation in section 5.1.

```
import unittest

from seismic.observations import ObservationDAO
from seismic.detector import SaxDetect, StaLtaDetect
from seismic.detector.sax import sax_detect


def gen_from_string(ss):
    for s in ss:
        yield s


class SaxDetectTest(unittest.TestCase):
    def setUp(self):
        self.paa_int = 50          # ms of intervals for following samples
        self.alphabet = "abcdefg"  # alphabet for following samples

        self.sax_str_1 = (
            'ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd'
            'ddddddddddddddddddddddddddddddddddddddddddddddacgagdbabbgcgagebgab'
            'gfagddbcbgfgacecfgabgadagaebggeggaadgaagadgfaadggbafggabaefg'
            'abgeabgaafgefgbdbbbagggfeaecaadgggffbbbdccdgeddeaacdffecccbd'
            'fdefcbcbbegfdcccddeedbceeefddceeddeefdecbddddeeddcddeccdeecc'
            'ddeddeeddccbdddccdcdddefdccdcdeedddcdedededdeeedccdddeedddcc'
            'ccddddddccddccddddddcddcddddeedddccdedcddccddddddddddddeeede'
            'eddcddddddddddddddddddddddddddddeddddddddddddddddddddddddccc'
            'ddddddddddddddddddddddeddddddddddddddddddddddddddddddddddddd'
            'ddddddddcccdcddddddddddddddddddddddddddddddddddddddddddddddd'
            'd')

        self.sax_str_2 = (
            'ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd'
            'ddddddddddddddddddddddddddddddddddddddddddddfgacaaggaadggaabcgadb'
            'feaffdbebfebbffeecbcfebedgbddedccegdbbbffceccddeecfacfgebggc'
            'aggaaeggaagfacedeageagfgaegbaefgeeaaggbaaagfcbfecffeacefcbgb'
            'ddbeadgdddeedebddbddaeedfbefgdccacccegfcdcdfdeeffdcbaadegeec'
            'accdffecdeedeeccbbcceeeddddedcdeddcceecdeddedcdeedcbeddddcdd'
            'dddeecdeecdddcbeeddddddeedccdeddeddddddddddcccddddeddeeddddcbc'
            'dedcddddccdddeddddeedddeddddeeddddddcdccdeedddeddddddcccddddddd'
```

```
        'ddddcdddedeedddcdeedccddddddeeddddddddddddddddddddcddddddddd'
        'ddddddddddddddddddddccddddddddddddcddddddddddddddedddddeeeddd'
        'd')

def test_sax_detect(self):
    """
    Tests two known events using sax_detect algorithm with a minimum
    event length of 5s to trigger on and a minimum quiet period of 5s
    to trigger off.

    These are Unknown.CALZ and Unknown.CAOZ respectively.
    """
    # String 1
    t = list(sax_detect(gen_from_string(self.sax_str_1),
                        self.alphabet, self.paa_int, 5000, 5000))
    self.assertEqual(len(t), 1, "Only␣one␣event␣should␣be␣found")
    duration_ms = (t[0][1] - t[0][0])
    self.assertTrue(10000 <= duration_ms <= 15000,
                    "Event␣was␣between␣8␣and␣15␣seconds,␣got␣{}"
                    .format(duration_ms))
    self.assertAlmostEqual(t[0][0] / 1000, 5, 1,
                           "Event␣was␣approximately␣5000ms␣from␣the␣
                               start"
                           ",␣got␣{}".format(t[0][0]))
    # String 2
    t = list(sax_detect(gen_from_string(self.sax_str_2),
                        self.alphabet, self.paa_int, 5000, 5000))
    self.assertEqual(len(t), 1, "Only␣one␣event␣should␣be␣found")
    duration_ms = (t[0][1] - t[0][0])
    self.assertTrue(10000 <= duration_ms <= 20000,
                    "Event␣was␣between␣10␣and␣20␣seconds,␣got␣{}"
                    .format(duration_ms))
    self.assertAlmostEqual(t[0][0] / 1000, 5, 1,
                           "Event␣was␣approximately␣5000ms␣from␣the␣
                               start,"
                           "␣got␣{}".format(t[0][0]))

def test_full_sax_detector_class_1(self):
    o = ObservationDAO("../../../sample/_all/cal.z")
    o.bandpass(5, 10)
    det = SaxDetect(o.stream[0].data, o.stats.sampling_rate)
    s, e = det.detect(self.alphabet, self.paa_int).__next__()
    self.assertTrue((10.5 <= s / 1000 <= 11),
```

```
                    "Event␣start␣was␣10.5—11␣seconds␣from␣start,␣got␣
                        {}"
                    .format(s / 1000))
    self.assertTrue((20.0 <= e / 1000 <= 30.0),
                    "Event␣end␣was␣20␣to␣30␣seconds␣from␣start,␣got␣{}
                        "
                    .format(e / 1000))

def test_full_sax_detector_class_2(self):
    o = ObservationDAO("../../../sample/_all/cao.z")
    o.bandpass(5, 10)
    det = SaxDetect(o.stream[0].data, o.stats.sampling_rate)
    s, e = det.detect(self.alphabet, self.paa_int).__next__()
    self.assertTrue((11.5 <= s / 1000 <= 12.5),
                    "Event␣start␣was␣11.5␣to␣12.5␣seconds␣from␣start,␣
                        "
                    "got␣{}".format(s / 1000))
    self.assertTrue((25.0 <= e / 1000 <= 35.0),
                    "Event␣end␣was␣20␣to␣30␣seconds␣from␣start"
                    ",␣got␣{}".format(e / 1000))

def test_full_sax_detector_class_3(self):
    o = ObservationDAO("../../../sample/_all/cps.z")
    o.bandpass(5, 10)
    det = SaxDetect(o.stream[0].data, o.stats.sampling_rate)
    s, e = det.detect(self.alphabet, self.paa_int).__next__()
    self.assertTrue((11.5 <= s / 1000 <= 12.5),
                    "Event␣start␣was␣11.5␣to␣12.5␣seconds␣from␣start"
                    ",␣got␣{}".format(s / 1000))

def test_full_stalta_detector_class_1(self):
    o = ObservationDAO("../../../sample/_all/cal.z")
    o.bandpass(5, 10)
    det = StaLtaDetect(o.stream[0].data, o.stats.sampling_rate)
    s, e = det.detect(
        short=50, long=5000, nstds=3, trigger_len=5000).__next__()
    self.assertTrue((10.5 <= s / 1000 <= 11),
                    "Event␣start␣was␣10.5␣to␣11␣seconds␣from␣start,␣"
                    "got␣{}".format(s / 1000))
    self.assertTrue((20.0 <= e / 1000 <= 30.0),
                    "Event␣end␣was␣25␣to␣35␣seconds␣from␣start,␣"
                    "got␣{}".format(e / 1000))

def test_full_stalta_detector_class_2(self):
```

```
o = ObservationDAO("../../../sample/_all/cao.z")
o.bandpass(5, 10)
det = StaLtaDetect(o.stream[0].data, o.stats.sampling_rate)
s, e = det.detect(
    short=50, long=5000, nstds=3, trigger_len=5000).__next__()
self.assertTrue((11.5 <= s / 1000 <= 12.5),
                "Event␣start␣was␣11.5␣to␣12.5␣seconds␣from␣start"
                ",␣got␣{}".format(s / 1000))
self.assertTrue((25.0 <= e / 1000 <= 35.0),
                "Event␣end␣was␣25␣to␣35␣seconds␣from␣start"
                ",␣got␣{}".format(e / 1000))
```

Bibliography

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

Bormann, P.
2012. New manual of seismological observatory practice.

Butterworth, S.
1930. On the theory of filter amplifiers. *Experimental Wireless and the Wireless Engineer*, 7.

Garnero, E. J.
2017. "seismology 101" images. [Online; accessed August, 2017].

Goitom, B., C. Oppenheimer, J. O. S. Hammond, R. Grandin, T. Barnie, A. Donovan, G. Ogubazghi, E. Yohannes, G. Kibrom, J.-M. Kendall, S. A. Carn, D. Fee, C. Sealing, D. Keir, A. Ayele, J. Blundy, J. Hamlyn, T. Wright, and S. Berhe
2015. First recorded eruption of nabro volcano, eritrea. *Bulletin of Volcanology*.

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

Haustant, A.
2016. Flask-RESTPlus fully featured framework for fast, easy and documented api development with flask.

Keogh, E., J. Lin, and A. Fu
2007. Hot sax: Efficiently finding the most unusual time series subsequence. *5th IEEE International Conference on Data Mining (ICDM 2005)*.

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*.

Řehůřek, R. and P. Sojka
2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, Pp. 45–50, Valletta, Malta. ELRA. `http://is.muni.cz/publication/884893/en`.

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*.