VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
DEPARTMENT OF COMPUTATIONAL AND DATA MODELING

Bachelors Thesis

Implementation of application for visualization of regularities and randomness in data

Done by:
Audrius Baranauskas signature

Supervisor:
Ph. D. Tadas Meškauskas

Vilnius
2021

Contents

Keywords

Pateikiamas terminų sąrašas (jei reikia)

Abstract

The goal of this project was to create a web application for generating and classifying recurrence plots that are used to visualize data regularities and randomness. The application features a responsive design allowing access from desktop and mobile users. Classification feature is implemented by training a convolutional neural network. It effective distinguishes chaotic, period and trending data types. The paper dives into the application development process. It also details the convolutional nerual network data generation, training and testing procdures. A concolusion was reached that a convolutional neural network can identify data characteristic by analyzing a recurrence plot image.

Santrauka

Aplikacijos kūrimas dėsningumų ir atsitiktinumų duomenyse vizualizavimui

Šio projekto tikslas buvo sukurti interneto aplikaciją, kuri sugebėti sukurti bei kategorizuoti rekurencines diagramas naudojamas duomenų duomenų dėsningumų ir atsitiktinų vizualizavimui. Aplikacija sugeba reaguoti į varotojo įrenginio dydį. Dėl šios priežasties ją galima naudotis tiek mobiliuoju telefonu, tiek namų kompiuteriu. Duomenu kategorizacija yra įgyvendinta panaudojant kovoliucinius neuroninius tinklus. Neuroninis tinklas gali tiksliai klasifikuoti chaotinius, periodinius bei trendą turinčius duomenis. Ši publikacija detaliai aprašo aplikacijos vystymo procesą. Ji taip pat nupasakoja kaip generuojami duomenys kovoliucinio neuroninio tinklo mokymams bei testavimui. Galiausiai, buvo prieita išvada, kad konvoliucinis neuroninis tinklas gali išmokti kategorizuoti rekurentines diagramas.

Introduction

Humans are not particularly good at dealing with large quantities of data, especially when it is expressed in a numeric value. We are visual creatures, as the common expression states *I Won't Believe It Until I See It*. This has lead to an explosion in visualization tools and methods. Depending on what the illustration is intented to represent, unique techniques are used. These techniques help us see the data from a unique perspective by emphasising certain aspects of data. For example in order to display possible logical relations between a colletion of data sets one might choose the venn diagram [20].

A task at hand is the visualization of data randomness and regularities. The common method used for exactly this job is a recurrence plot. This data analysis tool evaluates a stream of data and produces and image. For an untrained eye the visualization might seem trivial, but after seing a few examples one does not require a great deal of effort to learn how to read the recurrence plot. The algorithm used to generate this image has its peculiarities but the paper seeks to clarify them.

On the other hand, for a long time visual tasks were performed exclusively by humans. The underlying principles for machine learning have been around for quite some time, but were limited by hardware. Rapid increase in affordable computational power coupled with open sourced software is enabling the widespread adoption of this technology. This is an attempt to apply the current advances in machine learning in order to tackle a task of classifying characteristics of a recurrence plot. For the model to be utilized, a web application infrastructure is build allowing one to explore the recurrence plot and its properties.

1 Data randomness and regularities

This section covers what is the expected input and output of data analysis. It will explore a method for identifying data characteristics and its limitations. Finally it will dive deeper into innerworkings of the given tool used for this task.

1.1 Signal data

A signal is a function that conveys information about the behaviour of a system or attributes of some phenomenon [24]. For example, measuring the time taken between a weight-driven pendulum clock's ticks produces a signal. This hypthetical singal would consist of a series of numbers identifying how long each pendulum swing took. In this case such a signal could be expressed as a one dimensional stream of data. For the scope of this paper every singal we analyze will be a one dimensional series of numbers. When refering to singals we will use terms **signal** and **data** interchangeably.

1.2 Recurrence plot

To tackle the problem of identifying randomness and regularities in signals a **recurrence plot** can be utilized. A recurrence plot is a an illustration that can help determine the non-triviality of a given data series. It visualizates when a given signal repeats for each moment in a time series.

Given a `data` array of deciman numbers, the very primitive python implementation of the algorithm would look similar to this:

Listing 1. Primitive recurrence plot algorithm

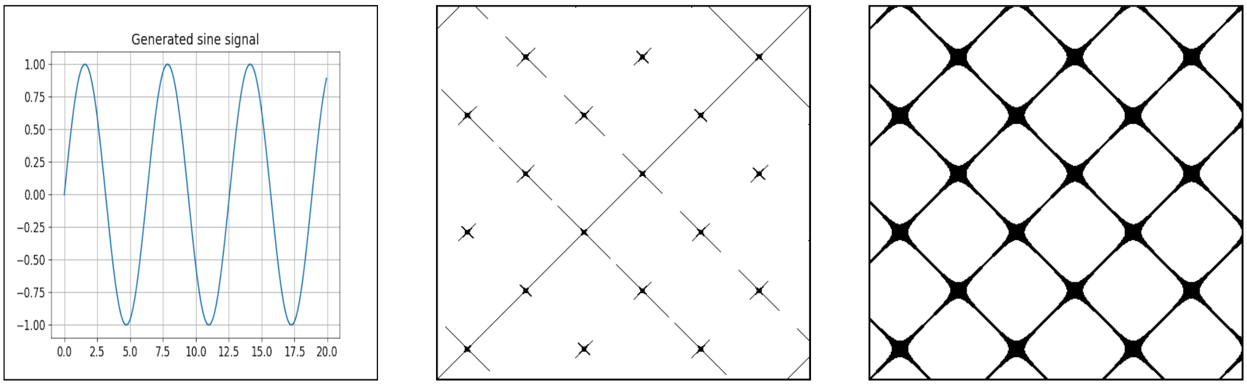
```python
def generate_recurrence_plot(data:list, r:int):
  recurrences = []

  for i in range(len(data)):
    for j in range(len(data)):
      if (abs(data[i] - data[j]) <= r):
        recurrences.append([i, j])

  return recurrences
```

The example above defines a method `generate_recurrence_plot` that takes in parameters `data` and `r`, of types *list* and *int* respectively. The method compares each element in `data` to every element including itself. If the difference of these elements is no larger than a given threshold `r` - they are considered similar and their index positions are added to the recurrences array. The method then returns an array of index pairs when data values were similar. These pairs can be represented as pixel possitions in an image and plotted to form a recurrence plot.

In figure 1 we can see a sine wave data on the left (a). The recurrence plot in the middle (b) is generated with a small value for `r` and produces a pattern with cross signs. Meanwhile the graph on the right (c) has a higher `r` value and displays a grid pattern. Recurrence plot (c) has noticably more black pixels. From observations in figure 1, we can conclude that the value for `r` ir proportional to the amount of similarities found when generating a recurrence plot.



(a) (b) (c)

Figure 1. (a) Sine wave data; (b) Recurrence plot D=1, d=1, r=0.0017; (c) Recurrence plot D=1, d=1, r=0.06

1.3 Recurrence plot parameters

Now that we know how a basic version of the algorithm behaves we can dive deeper into the recurrence plot algorithm. It is important to note that not all signal have such predictable patterns. Measuring the similarity of two values helps us understand where data *values* are similar. However, two sequential signal values being similar does indicate that the following values are similar as well. A better way to approach detecting similarities is comparing signal *states* instead of *values*.

Signal state is a collection of signal values commonly represented as a vector. A linear signal's state can be represented as a pair of two consecutive signal values. The number of consecutive values that make up a signal state is arbitraty and can be increased to tripplets, quadruplets, D-plets. For example, recurrence plots generated in figure 1 used D a parameter of 1, therefore it is also correct state that the algorithm in listing 1 was used to compare one dimensional vectors. For example, given a signal of length N, and a D parameter, one can divide the signal sequence into a list of signal state vectors.

$$y_0 = (f_0, f_1, ..., f_{D-1}), \quad y_1 = (f_1, f_2, ..., f_D), \quad ..., \quad y_{N-D} = (f_{N-(D-1)-1}, f_{N-(D-0)-1}, ..., f_{N-1})$$

It is also worth metioning that a signal state not necessarily contain sequential values. Signal values can also be have a delay step d. The delay step indicates that only every d-th element is to be used to compose a given signal state. This translates into the following series of elements, given a signal length of N, D and d parameters.

$$y_0 = (f_0, f_d, ..., f_{(D-1)*d}), \quad y_1 = (f_1, f_{d+1}, ..., f_{D*d}), \quad \longrightarrow$$
$$..., \quad y_{N-(D-1)*d} = (f_{N-(D-1)*d}, f_{N-(D-0)*d}, ..., f_{N-1})$$

It is apparent that the number of associated signal states decreases in relation to D and d. The total number of signal states is denoted with M. From the expression above we can extrapolate that the M is calculated by using the following expression.

$$M = N - (D - 1) * d$$

We can conclusively state for a given signal Y of length N, the generated recurrence plot is always of size M^2.

1.4 Comparing signal states

Now that we have familiarize with recurrence plot parameters let us go back to out primitive implementation in listing 1. We are no longer comparing values, but rather - vectors.

A distance between two vectors v_i and v_j is denoted as $||v_i - v_j||$. We can assign this vector distance to a D - dimensional point $p = (p_1, p_2, ..., p_D)$.

The most common way of comparing vector length is by utilizing the *Euclidean metric* also known as the L_2 *norm*. L_2 norm calculates the distance between a vector in a D - dimensional space and the start of the coordinate plane. This norm can be derrived by using the Pythagorean theorem:

$$||v|| = \sqrt{v_1^2 + v_2^2 + v_3^2 + ... + v_D^2}$$

Alternative solutions for calculating a vector distance are the Manhattan norm

$$||v|| = |v_1| + |v_2| + |v_3| + ... + |v_D|$$

and the Maximum norm.

$$||v|| = max(|v_1|, |v_2|, |v_3|. + ...+, |v_D|)$$

From a computational perspective it becomes quite apparent that the Maximum norm requires the least effort. For the scope of this paper we have compared the performance characteristics

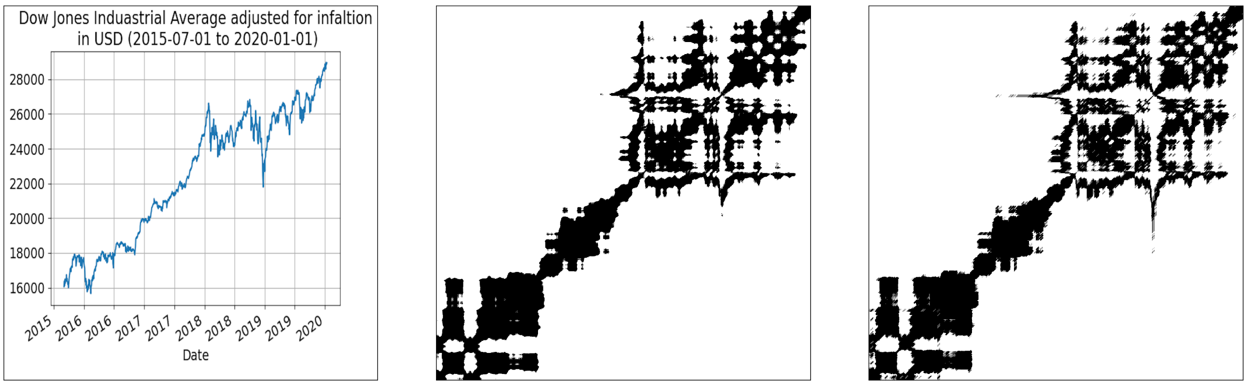


Figure 2. (a) The Dow Jones Industrial Average adjusted for inflation 2015-2020; (b) Recurrence plot using the Euclidean norm D=4, d=2, r=842, T=3s; (c) Recurrence plot using the Euclidean norm D=4, d=2, r=1969, T=0.5s; Source: yahoo-finance [1]

of the Euclidean and the Maximum norms. Figure 2, graph on the left 2a illustrates stock price fluctuations of the Dow Jones Industrial average. Recurrence plot in the middle 2b is generated using the Euclidean norm, while plot on the right 2c utilizes the Maximum norm. Only a miniscule differences can be observed in the generated plots. However, as capture indicates, the Maximum norm performed about 6 times faster than the Euclidean, taking 0.5 and 3 seconds to complete respectively.

1.5 Performance improvements

One giant leap in performance was the utilization of the Maximum norm instead of the Euclidean norm as documented in section 1.4. The recurrence plot algorithm can be further optimised by only calculating one half of the recurrence plot. Refering back to listing 1, we could make simple code modification to the looping system as shown in listing 2.

Listing 2. Improved looping system

```
for i in range(len(data)):
  for j in range(i, len(data)):
    #Signal state comparison
```

Note that the second level loop now iterates an ever decreasing number of times. If plotted, the recurrence plot would only have one half if the image divided by the symetry line. This can be easy overcome by populating the similarities array with the same data, just switched positions of indices. This effectively reduces the processing time in half.

1.6 Calculating the pixel percentage

A key characteristic of a recurrence graph is the diagonal line of symetry spanning across the image. This line is always present because the algorithm iterates in a nested loop. Looking back at listing 1 we can identify that when the first level loop iterator `i` is equal to the second level loop iterator `j` - the algorithm compares a distance between the same sygnal states as `data[i]` and `data[j]` use the identical index. Regardless of the vector D dimensionality and the norm used to find the distance - it always remains zero. As r has to be a positive number, we can conclude that a distance of zero is always less than the threshold. Therefore in a given image of size M the similarities array returned by a recurrence plot algorithm always includes similarities at positions (i, j), when $i = j$.

Consequently, calculating pixel percentage of a recurrence plot requires us to exclude the diagonal line. The total number of pixels in a recurrence plot is M^2. A diagonal line has one pixel in each row, therefore the line pixel count is M.

Lets denote the number of pixels inside the graph as p and the pixel percentage as P. Taking into account a graph utilizing performance improvements, the pixel percentage can be calculated with the following expression:

$$P = (p - M)/(M^2 - M) * 2 * 100$$

Note that we subtract the diagonal line from both sides of the equation, then multiply everything by a factor of 2 to factor in that we are only dealing with half the plot data. Finally we multiply by 100 to get the ratio in percentages.

1.7 Calibrating the treshold parameter

As previously mensioned, when generating a recurrence plot, a parameter r determines a treshold for the maximum allowed distance between a pair of signal states. Looking back at figure 1 we can see that a higher r value produces a more densly populated recurrence plot.

A recurrence plot with a low pixel percentage can be difficult to read as it lacks data. Similarly, a plot too densely populated with pixels can have the same effect. Usually a healthy percentage target is between 15 and 20 percent. The algorithm does not provide an easy way to determine r value. A human can guess a treshold value, generate a plot, check the pixel percentage and adjust r. This is to be repeated until desired pixel percentage is reached.

It is prefered to tackle such tasks programatically. Lets denote t as the target pixel percentage and a as the allowed pixel percentage deviation. We first define the minimum and maximum values for r. Assuming we use the Maximum norm, we can deduce that $r_{min} = 0$ and $r_{max} = max(y)$, where y is array of input data. A fair guess for r could be calculated by multiplying subtracting the maximum and minimum treshold values and scaling by the pixel percentage target

$$r = (r_{max} - r_{min}) * t/100$$

It is unlikely that the pixel percentage is estimated correctly, but it is a starting point. A full programatic implementation of the algorithm is provided in appendix A. Refering to it we can see that the current and previous value for r and P are stored. Each iteration the sum of r and r_{last} is compared to P and P_{last}. This ratio is then scaled to the target pixel percentage p. This is algorithm is repeated until a given pixel percentage target is reached. Using this programatic solution can find r, where $P \approx t$ in 2 to 3 iterations.

2 Web application development

This project is aimed at creating a web application allowing one to interact with the recurrence plot algorithm in a user friendly manner. The project offers a feature of classifying data based on the generated plot using convolutional neural networks. This is an effort to further spread the popularity of this algorithm and help users intuitively grasp how it behaves.

2.1 Analysis of analogous tools

As of the date of publishing, only one tool was located capable of generating a recurrence plot online [23]. There are multiple implementations of the recurrence plot in Python as well as other languages, but none offer the ability to classify data based on the generated image.

It is noteworthy, that the aforementioned implementations require at least a minimal undertanding of software programming, a computing machine and specific software to compile and run the code. This is laborious and is not likely to attract new users to experiment with algorithm. Based on these factors, a decision was made to create a web based application that requires as little user knowledge to get started with the algorithm as possible.

2.2 Architecture

This project consists of three microservices. Microservices are small autonomous services deployed independently, with a single and clearly defined purpose [21]. This design approach was chosen due to the flexibility and scalability associated with the architecture. The nature of microservices allows one to easily test, modify or out right replace each one of the components giving more freedom to the developer. The project ecosystem consists of the following microservices:

1. Front end web application
2. Back end for the web application
3. Python web server for plotting operations

Communication between microservices is performed via HTTP requests. In general, a query with JSON body is sent to a service and a JSON response along side an image attachment is returned. Figure 3 illustrates the microservice architecture of the project and data flow among services.

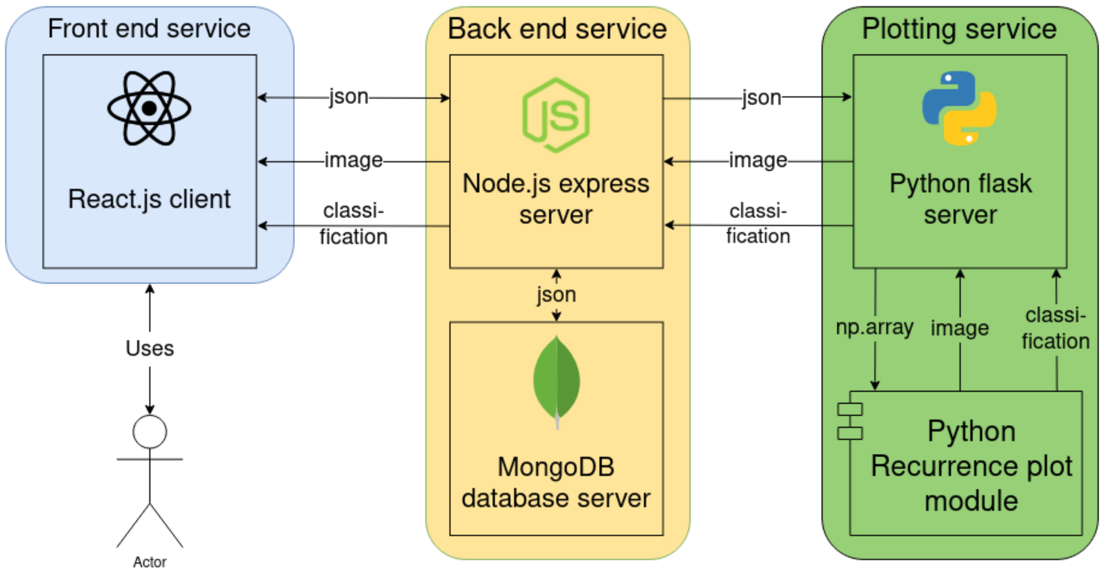


Figure 3. Microservice architecture structure

2.2.1 Project structure

The project is structured so that each microservice resides in an independent directory:

- app/
  - src/
    - components/
  - public/
- server/
  - db/
  - public/
  - utils/
- plotter/
  - jupyter/

As the name suggests - `/app/` directory contains the front end ReactJS application. The NodeJS express[2] server resides inside the `/server/` directory. Meanwhile, `/plotter/` contains all of the Python source code. That includes the flask [3] web server, the recurrence plot module, jupyter notebooks for convolutional neural network model development and scripts for model training data generation.

2.2.2 Project workflow

We will now cover an example workflow of the application as per figure 3.

When first opening the app, a request is sent to the back end to fetch a list of existing plot data. The user selects an entry from the list and fills in remaining parameters for generating a recurrence plot. A request with select data ID is sent to the back end microservice. The back end service fetches data from the database and forwards it to the plotting service. The plotting service generates an image, then runs the image through a convolutional neural network to get classification data. Finally, the plotting service sends the image along with classification data back to the back end service, which in turn forwards it to the front end service. The front end service displays the image and classification data.

2.3 Microservices

The tools used for microservice development were largely open-sourced and relatively modern. Front end and Back end services were written in javascript based environments - React JS and Node JS respectively. These choices were made due to the widespread use of javascript in modern web application development providing a large pool of open-sourced libraries and tools.

On the other hand python was the tool used to develop the plotting service. It is known to perform better on data handling and machine learning than javascript alternatives [22]. Both Python and Node JS have certain strengths and thus have appropriate community driven libraries and modules to reinforce their leverages in appropriate operations.

2.3.1 Front end microservice

The front end service is developed using React - A JavaScript library for building user interfaces [19]. SASS is used for styling the apllication due to the intuitive syntax it provides [14]. The microservice utilizes the Node Package Manager [11]. From the NPM registry, two open sourced libraries are used:

- node-fetch - A module that brings window.fetch to Node.js [10].
- query-string - a tool for building HTTP query string [13].

These libraries were used to facilitate communication via HTTP requests with the back end server.

Following the best practices of React development, the app is broken down into reuseable components. Figure 4 indicates the application structure denoting components with the standard JSX component notation `<component />`. We will be using this notation to refeter to JSX components.

Figure 4 indicates that a root `<App />` component wraps the whole application. Initially, only the `<Header />`, `<Selector />` and `<Loader />` components are visible to the user. The `<Selector />` component sends an HTTP GET request to the backend service to retrieve a list of available plot data. This list is displayed inside the `<Selector />` for the user to pick from. A user must select a data entry and may add optional plotting parameters. Submitting the `<Selector />` form sends an HTTP GET request to the back end service. The backend service returns a JSON with the location of the generated recurrence plot image and additional parameters. After handling the server response - the `<Loader />` component is replaced by the `<Image />`. During any further plot requests, the `<Image />` is briefly replaced by the `<Loader />` component to indicate that a request is being processed.

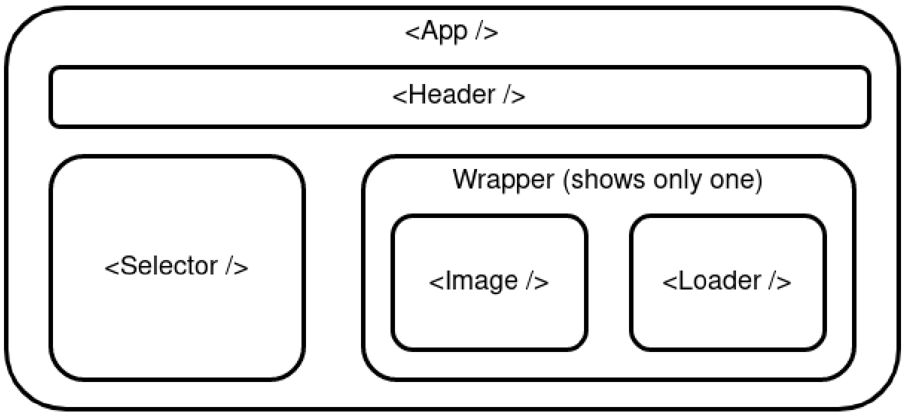


Figure 4. React app component structure

2.3.2 Back end microservice

The backend microservice also utilizes libraries provided by the Node Package Manager. The service runs on an Node JS express server [2]. The server handles all requests from the front end service. Server endpoints cover the following operations:

- CRUD operations for plot data stored inside the MongoDB database
- Requests to generate a recurrence plot using the plotting service

The express server communicates with the database server by making use of an open sourced MongoDB object modeling library - mongoose [9]. The service itself does not generate any plot data, but merely acts as an intermediary between the front end service, the MongoDB database and the plotting service.

2.3.3 Plotter microservice

The plotter microservice handles requests to generate and classify recurrence plots. The service utilizes numpy[12], scipy[15] and matplotlib[7] open sourced python libraries.

The service consists of 3 main parts:

1. Flask - a python web framework
2. Recurrence plot module
3. Data classification model

The flask service handles HTTP requests with JSON data as input. The service processes the input and generates an image using the recurrence plot module. Image is passed through the convolutional neural network to get the image classification. An HTTP response is then sent containing the classification data and the generated image as an attachment.

2.4 Recurrence plot module

The recurrence plot module is a Python implementation of the algorithm used to generate a recurrence plot. The module creates a `RecurrencePlot` object. The object takes in several parameters as input allowing one to customize the following features of the generated plot:

- D - signal dimension
- d - signal delay
- compare_mode - evaluation metric: euclidean and maximum
- target - Prefered pixel percentage of the recurrence plot
- deviation - allowed deviation for final pixel percentage

As output the module returns the name of the asset in the local storage. The object can also be manipulated to retrieve various other metrics about the recurrence plot. The module implements all of the best practices described in sections 1.4 - 1.7.

3 Data classification model

A recurrence plot reveals certain information about the singal. After some practice a human can identify whether a given signal exhibits signs of periodity and / or stationarity, has a trend or seems to be random in nature. The goal of this model is to determine some the aforementioned characteristics of a signal by analyzing the reucrrence plot generated by it.

3.1 Tools and libraries

The convolutional neural network along with assets was developed using the python programming language. Libraries for asset generation, image preprocessing and the training of CNN are mostly open sourced. They are as follows: Numpy [12], TensorFlow [16], Keras [4], scikit-learn [8] and matplotlib [7].

3.1.1 Hardware specifications

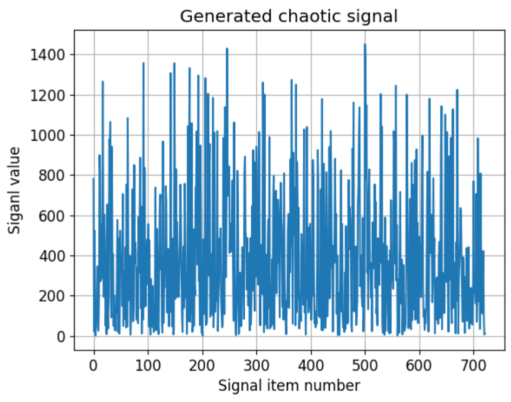
Machine learning is a resource intensive task. TensorFlow supports both: the Central Processing Unit and Graphics Processing Unit for training neural networks. For this project, GPU accellerated learning was used. The GPU device used: GTX 1070 Ti with 8 Gigabytes of on-board memory.

3.2 Generating training data

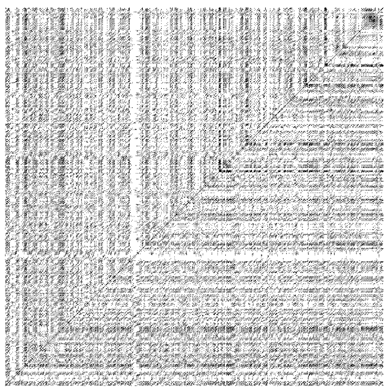
It is common knowledge that one requires data to train a convolutional neural network. The accuracy of a data model heavily weighs on the quality of training data and labeling. After a brief search for publicly available, labeled and categorized data that is suitable for training a recurrence plot categorising model, a decision was made for this data to be generated synthetically.

3.2.1 Training data methodologies

All of the following signals and their graphs are generated by utilizing the aforementioned libraries and the recurrence plot module. For every signal a complementary graph image is generated to help visualize the data which is depicted in a given recurrence plot. Due to the module flexibility every chaotic and periodic asset had randomised values for D - Dimension and d - delay. In addition, most aspects of each graph had a randomised flaoting point number be added or subtracter. This



(a) Chaotic signal



(b) Recurrence plot. Pixel target=17.5%, D=2, d=1

Figure 5. Chaotic signal and recurrence plot generated by it

aided in generating a more diverse set of training assets. The number of assets chosen was arbitraty - 1000 sygnals for each training label. That amounts to a total of 3000 recurrence plot images used for training the CNN. The source code for generating assets and associated graphs is inside the `/plotter/generate-cnn-assets.py` file.

3.2.2 Choosing classification features

Before generating the data it is important to recognize what the convolutional neural network is expected to learn from it. The *what can it learn?* aspect of a CNN is mostly limited by the labeled data we can provide. A choice was made to classify plot images into one of the three groups:

- Plotted signal is chaotic
- Plotted signal has a trend
- Plotted signal has is periodic

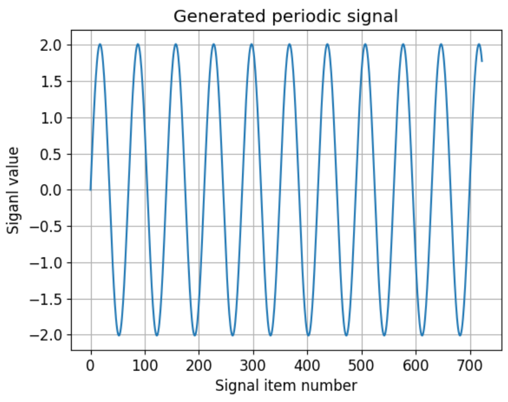
We will explore the reasoning of this decision by exploring the limitations of generating labeled signals.

3.2.3 Chaotic signal

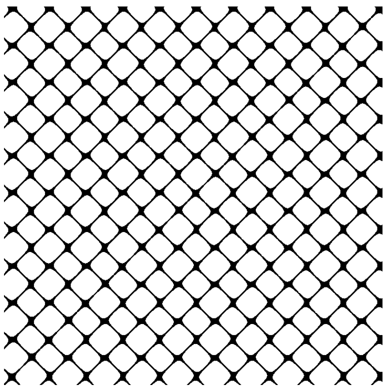
A relatively simple signal to identify is a chaotic signal. Chaotic, means it has no distinguishable pattern - random. This signal can be generated rather easily - by calling a random number generator for each entry in the signal. It is also advantageous, that chaotic signal does not trigger any other feature attribute except for stationarity. A chaotic signal tends to have a high level of stationary meaning it is dispersed fairly evenly across the recurrence plot. Figure 5 displays a signal (a) and the generated recurrence plot (b) from the training set. This signal is labeled as chaotic and is one of the signals used for training the convolutional neural network.

3.2.4 Periodic signal

For a human, identifying a signal with a periodic characteristic is not too strenuous either. Unfortunately we cannot consider the periodicity of a signal without considering the stationarity of it.

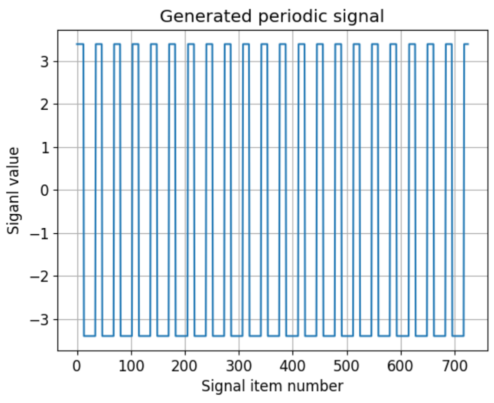


(a) Sine wave signal

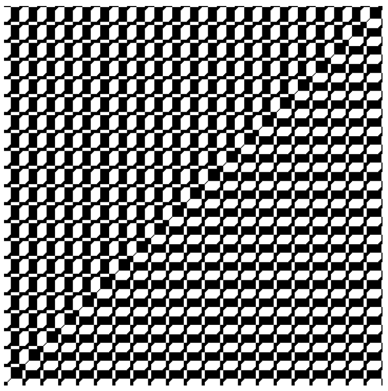


(b) Recurrence plot. Pixel target=17.5%, D=3, d=2

Figure 6. Periodic sine wave form signal and recurrence plot generated by it



(a) Square wave signal

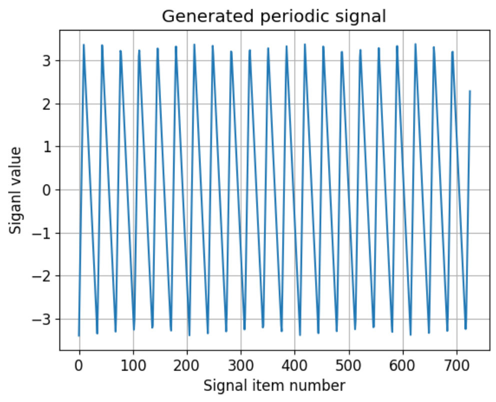


(b) Recurrence plot. Pixel target=17.5%, D=4, d=2

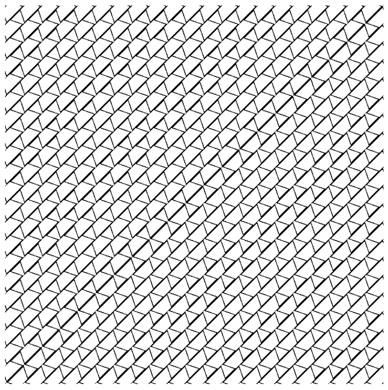
Figure 7. Periodic square wave form signal and recurrence plot generated by it

Stationarity is generally observed by the homogeneity of a signal. In layman terms - how evenly the pixels are spread across the recurrence plot. Looking back at figure 5 we can see that a chaotic signal is highly stationary as the pixels are spread fairly evenly. Figure 6 illustrates a signal (a) from the periodic training set and the recurrence plot (b) that is generated from it. We can see that this siganl forms a grid pattern. The pattern stretches across the whole image therefore this signal is also stationary. It would be difficult to generate periodic data that is not stationary, but the same applies to chaotic data. This is the reason why stationarity is not one of the attributes measured by the model. Determining the level of periodicity is beyond the scope of this particular neural netowrk.

A periodic signal wave can have difference forms. The signal in figure 6 is generated from a sine wave. To provide the model with more diverse training data - two additional distinct wave functions were used to generate the training data. Figure 7 illustrates a signal with a square wave. See the signal wave (a) on the left and generated recurrence plot on the right (b). The final wave-form to be used for periodic images is the sawtooth waveform. Figure 8 illustrates the wave signal (a) and the recurrence plot generated (b) using it.

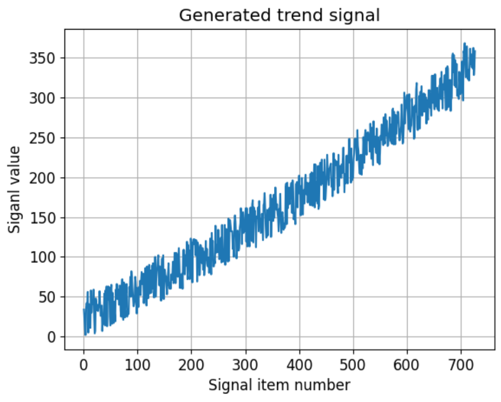


(a) Sawtooth wave signal

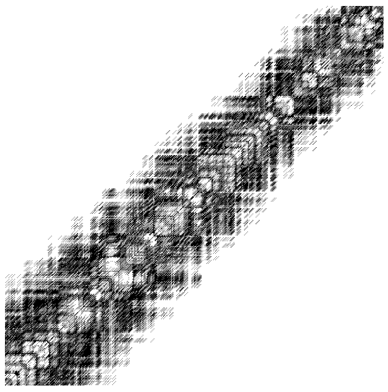


(b) Recurrence plot. Pixel target=17.5%, D=4, d=2

Figure 8. Periodic sawtooth wave form signal and recurrence plot generated by it



(a) Trend signal



(b) Recurrence plot. Pixel target=17.5%, D=4, d=2

Figure 9. Trend signal and recurrence plot generated by it

3.2.5 Signal with a trend

A trend signal is quite distinguishable by it's tendency to be less stationary than the previous two. Recurrence plot of a non synthetic trend signal appears to center aruond the diagonal symetry line and tends to have an increasing width towards either side of the image. Figure 9 illustrates an example trend signal (a) and the recurrence plot generated using it (b). This is one of the labeled samples used in training the convolutional neural netowrk.

These signals are generated by using a randomly generated data starting point. Then generating an integer within a given range to simulate increasing or decreasing data. Finally slightly increasing the average signal value by a flat value multiplied by an exponent. This allows synthetic trend data to either increase linearly or exponentially. Looking closely to figure 9 image on the left we can see that the trend data exhibits an exponentially increasing in values. The exponent is picked to be small so that the synthetic signal appears more natural.

3.3 Data preprocessing

Assets used for training a model need to be properly prepared before they can be used by the convolutional neural network. But first, it is vital to determine the form of the CNN input data.

There are two obvious ways for tackling this problem.

One way is for the signal to be passed as a two dimensional array of binary data. At the corresponding pixel location the array of binary data would containing 0 if a pixel is white and a 1 otherwise. The array data size could further be reduced if only one half of the symetrical image was taken.

Another way of dealing with this is by using the whole image as an input. An andantage of this approach is that the model would be less dependent on the recurrence plot implementation and would be easier to use by a third party. A downside to this approach is that training this model would require more computing power. It was decided to use the whole image as an input.

The complete pool of assets is divided into istributed into three groups:

- Training data - used for the training of the model. This group contains 85% of all training assets.
- Validation data - used to validate the model accuracy after each training epoch completes. This group contains 10% of all training assets.
- Testing data - used for manual testing of the final model. This group contains 5% of all training assets.

Images in each group are preprocessed using the vgg16 preprocessor [25], labeled and divided into batches of 10. Utilizing the vgg16 preprocessor, image is scaled down to the size of 224 by 224 pixels. This is implemented in order to reduce the number of parameters in the CNN.

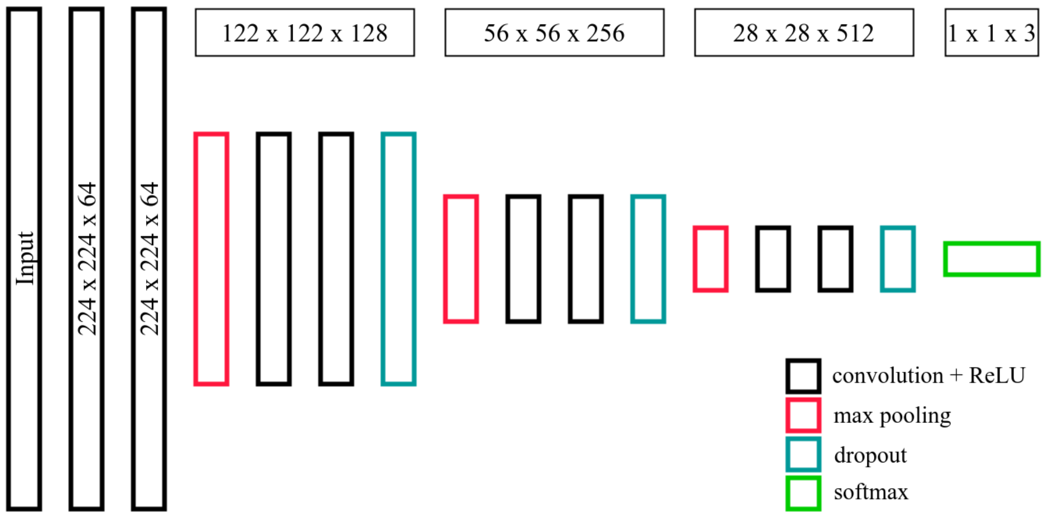


Figure 10. Convolutional neural network layer structure

3.4 Convolutional neural network

A convolutional neural network is a type of artificial neural network that has been successfully applied in computer vision tasks [18]. Typically, a CNN consists of convolutional pooling and traditional fully connected layers. An important aspect of a CNN, is to obtain abstract features when input propagates toward the deeper layers [17]. This is exactly what we expect the CNN to pick up on. For example, we expect a CNN to recognize grid patterns in figure 6 and associate them with a periodic attribute.

To achieve this goal the CNN is inspired by the famous vgg16 model [25]. The model proposed earlier is excellent for spacial objects recognitions. Unfortunately it did not achieve desired results for our task. The CNN would perform well on the training samples, but poorly when faced with novel images. This indicates an overfitting problem. Overfitting is a fundamental issue in supervised machine learning which prevents a neural network from generalizing the models to perform well on training data, as well as unseen testing sets [26]. It is usually caused by lack of training data or a neural network with too many parameters.

A common and effective way to combat overfitting is the introduction of dropout layers. A dropout layers randomly sets input units to 0 with a given frequency rate at each step during training time [5]. Adding a dropout layer with the frequency rate of 25% after every pooling solved the overfitting issue. This is illustrated in figure 10.

3.4.1 CNN layers structure

In total, a given input goes through 15 layers. Using the Keras Sequential class, layers are grouped into a linear stack [6]. The CNN has four pairs of convolutional layers illustrated as black rectangles in figure 10. Excluding the first two layers, each pair of convolutional layers has max pooling layer marked before them and a dropout layer after them. In figure 10, the max pooling and dropout layers are depicted in red and blue respectively. A fully connected layer with the softmax activation acts as the output layer.

3.4.2 Training and testing the CNN

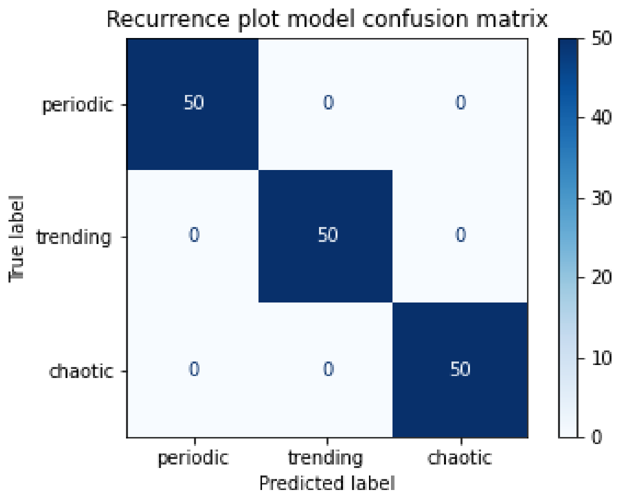


Figure 11. Confusion matrix for predicting test data, n=150

The Keras Sequential model is trained by compiling it and invoking the fit function. The model was compiled using the following parameters: Adam - optimizer function of choice, learning rate - 0.001, objective function - crossentropy objective, metrics - accuracy.

Model was trained in 10 epochs, each taking about 27 seconds to train, running on the aforementioned GPU accellerated hardware 3.1.1.

After 7 our of 10 epochs the model seems to be performing at a 100% accuracy. Testing the model with novel data yields the same results. Figure 11 confusion matrix illustrates the results of predicting the charactestics of recurrence plot data before unseen to the model. Looking at the first column and first row of the matrix, we can identify that 50 out of 50 images were classified as periodic. The same apllies for trending and chaotic samples. This indicates that the model performs very well on this type of data.

Conclusions and Recommendations

During the research and development phases of this project, the following goals were achieved:

- The recurrence plot was analysed to evaluate what can be determinen from the visualization. Several observable characteristics were distinguished. A local implementation of the recurrence plot was developed and utilized throughout the application.
- A convolutional neural network, capable of identifying the characteristics of a recurrence plot, was developed and trained. The trained data model performed with high accuracy.
- An web application for generating recurrence plots was developed. A microservice architecture was used develop the application providing a modern, scalable solution. The application utilizes the aforementioned CNN to provide information about the generated, that a novel user might not distinguish.

The application functionality can be yet improved. Microservice could be containerized to provide easier deployment. The front end part of the application could support full CRUD operations for managing of data in the database. So far this this can only be done by accessing the backend service via an exposed API.

References

[1] Yahoo finance - dow jones industrial average historical data.
https://finance.yahoo.com/quote/2020. [Online; accessed 14-Nov-2021].

[2] Expressjs - nodejs web application frameowrk.
https://github.com/expressjs/expressjs.com, 2021. [Online; accessed 2-Jan-2021].

[3] Flask - python web application framework.
https://flask.palletsprojects.com/en/1.1.x/, 2021. [Online; accessed 2-Jan-2021].

[4] Keras - an api designed for human beings, not machines.
https://github.com/keras-team/keras, 2021. [Online; accessed 4-Jan-2021].

[5] Keras dropout layer - applies dropout to the input.
https://keras.io/api/layers/regularization_layers/dropout/, 2021. [Online; accessed 4-Jan-2021].

[6] Keras sequential groups a linear stack of layers into a tf.keras.model.
https://keras.io/api/models/sequential/, 2021. [Online; accessed 4-Jan-2021].

[7] Matplotlib - a comprehensive library for creating static, animated, and interactive visualizations in python.
https://github.com/matplotlib/matplotlib, 2021. [Online; accessed 3-Jan-2021].

[8] Matplotlib - a python module for machine learning built on top of scipy and is distributed under the 3-clause bsd license.
https://github.com/scikit-learn/scikit-learn, 2021. [Online; accessed 4-Jan-2021].

[9] Mongoose - a mongodb object modeling tool designed to work in an asynchronous environment.
https://github.com/Automattic/mongoose, 2021. [Online; accessed 3-Jan-2021].

[10] Node fetch - a module that brings window.fetch to node.js.
https://github.com/node-fetch/node-fetch, 2021. [Online; accessed 2-Jan-2021].

[11] Npm - node package manager.
https://github.com/npm/, 2021. [Online; accessed 2-Jan-2021].

[12] Numpy - the fundamental package needed for scientific computing with python.
https://github.com/numpy/numpy, 2021. [Online; accessed 3-Jan-2021].

[13] Query-string - an http query string building module.
https://github.com/sindresorhus/query-string, 2021. [Online; accessed 3-Jan-2021].

[14] Sass - a preprocessor scripting language that is interpreted or compiled into cascading style sheets.
https://github.com/sass, 2021. [Online; accessed 3-Jan-2021].

[15] Scipy (pronounced "sigh pie") is open-source software for mathematics, science, and engineering.
https://github.com/scipy/scipy, 2021. [Online; accessed 3-Jan-2021].

[16] Tensorflow - an end-to-end open source platform for machine learning. https://github.com/tensorflow/, 2021. [Online; accessed 4-Jan-2021].

[17] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1--6, 2017.

[18] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354 -- 377, 2018.

[19] Facebook Inc. React is a javascript library for building user interfaces. https://github.com/facebook/react, 2020. [Online; accessed 2-Jan-2021].

[20] Hans A. Kestler, André Müller, Thomas M. Gress, and Malte Buchholz. Generalized Venn diagrams: a new method of visualizing complex genetic set relations. *Bioinformatics*, 21(8):1592--1595, 11 2004.

[21] L. Krause. *Microservices: Patterns and Applications: Designing Fine-Grained Services by Applying Patterns*. Lucas Krause, 2015.

[22] Miśtal Krzysztof. Performance comparison: Javascript vs. python for machine learning. https://dlabs.ai/blog/performance-comparison-javascript-vs-python-for-machine-learning/, 2020. [Online; accessed 2-Jan-2021].

[23] Norbert Marwan. Recurrence plots and cross recurrence plots. http://recurrence-plot.tk/online/index.php. [Online; accessed 2-Jan-2021].

[24] R. Priemer. *Introductory Signal Processing*. Advanced Series In Electrical And Computer Engineering. World Scientific Publishing Company, 1990.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[26] Xue Ying. An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 1168:022022, feb 2019.

Appendices

The paper contains appendicy A. It displays a programatic solution to calibrating pixel percentage implemented in Python programming language.

A Callibrating recurrence plot threshold r

Listing 3. Pixel percentage calibration

```
def calibrate_r(input_data, pixel_percentage_target, allowerd_deviation)
    :
  # Initialize variables
  r = min(input_data)
  pixel_percentage = 0
  r_last = max(input_data) - min(input_data)
  pixel_percentage_last = 100

  # Loop until calibrated
  while (abs(pixel_percentage-pixel_percentage_target) >
      allowed_deviation):
    # Compare the sum of current and previous R
    # to the sum of current and previous pixel percentages
    # Then scale R value based on pixel percentage target

    r = (r+r_last)/(pixel_percentage+pixel_percentage_last)*
        pixel_percentage_target

    # Calculate recurrences
    calculate_recurrences()

    # Update last pixel percentage and r
    pixel_percentage_last = pixel_percentage
    r_last = r
  return r
```