*I hereby declare that this work has not been submitted for any other degree/course at this University or any other institution and that, except where reference is made to the work of other authors, the material presented is original and entirely the result of my own work at the University of Strathclyde under the supervision of Dr. Robert Atkinson."*

Signature: ..... Date: .....

| | |
|---|---|
| Title: | Time Series Analysis and Prediction |
| Student: | Bartosz Lewandowski |
| Reg.No: | 201301372 |
| Supervisor: | Dr. Robert Atkinson |
| Second Assessor: | Dr. Mark Roper |
| Date: | 31/03/2017 |


# "Time Series Analysis and Prediction"

**Bartosz Lewandowski**

A thesis presented for the degree of

Bachelor of Engineering with Honours

Department of Electrical and Electronic Engineering

University of Strathclyde

Glasgow, 31/03/2017

# Abstract

The aim of the project was to investigate and implement multiple supervised machine learning algorithms for timeseries prediction.

First stage, required implementing a data collecting application for river level and weather information data upon which predictive models will be trained. After gathering a sufficient dataset, Recurrent Neural Networks, Support Vector Regression and Gaussian Process Regression algorithms were investigated both theoretically and practically.

Predictive models capable of predicting twenty timesteps into the future, were computed using only the raw river level data for ten different rivers. Afterwards performance of the algorithms was compared using average mean absolute percentage error, for each timestep into the future, over all ten rivers.

After selecting the most accurate model, the solution aims to indicate future river levels in Scotland, in order to predict flooding events. Thus, help the authorities take appropriate action before such events occur.

# Acknowledgements

I would like to thank University of Strathclyde and Dr. Robert Atkinson for the opportunity to work on this project, and all the feedback and guidance provided.

Moreover, I'd like to thank all my fellow Computer and Electronic Systems students, especially Roman O'Dowd for the support and feedback provided throughout the project.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **ANN** | Artificial Neural Network |
| **API** | Application Programming Interface |
| **AWS** | Amazon Web Services |
| **BGD** | Batch Gradient Descent |
| **BPTT** | Backpropagation Through Time |
| **CPU** | Central Processing Unit |
| **CSV** | Comma Separeted Values |
| **FNN** | Feedforward Neural Network |
| **HTTP** | Hypertext Transfer Protocol |
| **JSON** | Javascript Object Notation |
| **LML** | Log Marginal Likelihood |
| **LSTM** | Long-short Term Memory |
| **GP** | Gaussian Process |
| **GPR** | Gaussian Process Regression |
| **GPU** | Graphics Processing Unit |
| **GRU** | Gated Recurrent Unit |
| **MAPE** | Mean Absolute Percentage Error |
| **MIMO** | Multiple-input multiple-output |
| **ML** | Machine Learning |
| **MLP** | Multi-Layer Perceptron |
| **MSE** | Mean Squarred Error |
| **ORM** | Object Relational Mapper |
| **OS** | Operating System |
| **RBF** | Radial Bias Function |

| | |
|---|---|
| **RNN** | Recurrent Neural Network |
| **SEPA** | Scottish Environmental Protection Agency |
| **SGD** | Stochastic Gradient Descent |
| **SSH** | Secure Shell |
| **SVR** | Support Vector Regression |
| **VPN** | Virtual Private Network |

# Chapter 1

# Introduction

In this chapter the overall project description and goals will be outlined. Furthermore, a short summary of the report describing each chapter will be provided.

## 1.1 Project Description

The project task is to develop a predictive model to indicate future river levels in Scotland. The project involves collecting river level data on selected rivers and current weather data. Based on the data a predictive model will be developed using various supervised machine learning algorithms for solving regression problems. Indicated future river levels will be used by Scottish authorities for flood prediction. The results will be published on a web-based solution, and aim to increase situational awareness of the authorities and help take appropriate action, in flooding events such as evacuating nearby citizens. Even though the project focuses on environmental issues, the learning outcomes could be easily transferred to other similar data analysis tasks involving regression supervised machine learning problems.

## 1.2 Project Aims

The project aims could be split into four points listed below:

1. Design and implement suitable PostgreSQL database tables for storing river level and weather information data.
2. Implement a data collecting application in Python, that will pull data from external sources and store it in a local database.
3. Investigate and utilize number of different machine learning algorithms in order to build a predictive model.
4. Evaluate and select best performing model, following good practice machine learning techniques.
5. Implement a web based approach to publish the predictions made by the model in real time.

Throughout the project, the emphasis will be on investigating different supervised learning algorithms. In order to do so it is crucial to collect the data first, however publishing the web based approach is of minor importance.

## 1.3 Report Outline

The whole report outlines the work process throughout the project. Chapter 2 includes introduction to Machine Learning and some related basic concepts. Furthermore, it explains some of the good practises for data preprocessing and comparison of machine learning models. Chapter 3 describes the first stage of the project, which involves collecting river level and weather information data, that will be used for training the predictive models. It describes database tables that the data will be stored in and a basic outline of the data collecting application. The use of Recurrent Neural Networks to create a predictive model is described in Chapter 4. The concept of Neural Networks is introduced, with all related concepts that will be used for training a Recurrent Neural Network. Afterwards, an experimental procedure is described in more detail. Finally the model will

be evaluated. Chapter 5 and 6 will follow a similar structure to Chapter 4. In Chapter 5 the use of Support Vector Regression will be investigated for indicating future river levels. While in Chapter 6, Gaussian Process Regression will be explored. Chapter 7 will describe further work that could be done in order to extend the project. Eventually in Chapter 8 a final conclusion and comparison of the algorithms will be performed. Appendices will contain the code for chapters 3 to 6. Appendix A contains the code for data collecting application, appendix B contains the code used for developing a RNN model, appendix C contains code for investigating SVR algorithm and finally, appendix D contains the code used for GPR.

# Chapter 2

# Machine Learning

In this chapter, an introduction to machine learning and supervised machine learning is provided. Additionally, some general concepts and problems faced by machine learning algorithms are described. More specifically, the strategy for computing and evaluating a predictive model will be outlined, followed by descriptions of concepts such as generalisation and data standardisation. In section 2.6, different strategies for performing multiple steps ahead timeseries prediction are introduced. And finally the technological stack used for implementing machine learning algorithms is described.

## 2.1 Machine Learning

Machine Learning is a subfield of artificial intelligence, that involves development of self learning algorithms that gain knowledge from data, in order to make predictions on new or future data [13]. Due to the amounts of available data in the modern world, utilising these algorithms became extremely promising. Machine Learning algorithms allow capturing patterns in data very efficiently, without the need of humans to manually derive rules, in order to make data driven decisions. The applications of Machine Learning are endless, starting with email spam filter, natural language processing, time series analysis for stock exchange, up to self-driving cars. There are three different types of machine learning: supervised learning, unsupervised learning and reinforcement learning. This project focuses

on the supervised learning techniques for timeseries prediction.

## 2.2 Supervised Machine Learning

Supervised Machine Learning is a technique of training an algorithm from a labelled training dataset, in order to make predictions of unseen or future data. The training set is organised into an input vector $X$ and an output variable $Y$, which can be a label, a scalar or a vector. The algorithm produces a mapping function between the input and the output such that:

$$Y = f(X) \tag{2.1}$$

Simplifying, the algorithm learns from a set of examples given the answers, and learns the dependencies to predict future values.

Supervised Learning can be split into two problems: classification and regression. A classification problem is when the algorithm is meant to predict categorical class labels. A regression problem is when the algorithm is meant to predict continues outcomes. One of the most common problems for regression is time-series analysis, which is the subject of this project.

## 2.3 Machine Learning Process

Choosing an appropriate machine learning algorithm for a particular problem requires comparing performance of at least a handful of different algorithms. The "No Free Lunch" [16] theorem states that no algorithm preforms best for each possible problem. Model performance may differ on the number samples, number of features or amount of noise in the data. Thus, performance of an algorithm, depends heavily on the underlying data available for learning [13]. It is vital to follow a proper process in order to find the best algorithm, and can be summarised in the following steps:

1. Data Preprocessing: This step involves feature selection and scaling, dimensionality reduction and sampling. Moreover, the data has to be split

into training and testing subsets for model evaluation. The split is crucial in order to avoid overfitting.

2. Learning: This step involves choosing an algorithm and a suitable performance metric. Also hyperparameter tuning will be performed in order to maximise models performance.

3. Evaluation: At this stage, the models performance has to be evaluated using the right cross-validation technique such as holdout cross-validation or k-fold cross validation. The model is evaluated on a test subset, by comparing predicted values, to the actual measurements in the dataset.

4. Prediction: Now the model is ready to make prediction on new unseen data.

## 2.4 Generalisation

The ability to generalise is one of at most importance for a predictive model. That is, the model's capability to make accurate predictions on unseen data. A predictive model can either suffer from underfitting (high bias) or overfitting (high variance). The former means that the model is too simple, and is missing the relevant relation between features and target outputs. While, the latter means that the model is too complex for the underlying data set, this results in sensitivity to small fluctuations and modelling random noise in the training data, rather than intended outputs. Hence, the goal is to minimise both sources of error.

## 2.5 Data Standardization

Standardization is the process of rescaling the features of a dataset so that they have properties of standard normal distribution with zero mean and unit variance. Standardizing features is not only crucial for comparing features with different units, it is a general requirement for many machine learning algorithms. An example could be optimization algorithms, without standardizing features, certain weights may update faster than others as feature values plays a role in weight

updates [12]. The process of standardising data can be described in few simple steps. Firstly the mean has to be calculated:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.2}$$

Where n is the number of samples. Afterwards the standard deviation is calculated:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2} \tag{2.3}$$

Finally, Standardized values are calculated as follows:

$$\hat{x_i} = \frac{x_i - \mu}{\sigma} \tag{2.4}$$

In equation 2.4 $\hat{x_i}$ is the standardized value of variable $x_i$.

## 2.6 Time Series Prediction

Time series prediction involves using a set of past observations in order to estimate future values of a variable of interest. The most trivial case, *one step prediction*, predicts a value immediately following the latest observation. In other words for a set of samples observed at times $t_0, t_1, t_2, ..., t_n$, value at time $t_{n+1}$ is estimated.

Predicting two or more steps into the future is called *multi-step-ahead* prediction. It is significantly more challenging than one step prediction due to increasing number of uncertainties such as accumulation of error or lack of information [17]. Thus, selecting the right strategy is crucial for modelling a time series.

The two most commonly used strategies for multi-step-ahead forecasting are *iterated strategy* and *direct strategy.*

The iterated strategy, involves build a one step ahead predictor, and using the predicted values as input to forecast further into the future. For $n$ steps ahead prediction, the process has to be repeated $n$ times. Unfortunately, this strategy is susceptible to the error accumulation problem i.e errors commited in the past are propagated into the future [3].

In contrast to iterated strategy, the direct strategy consists of constructing a separate model for predicting each iteration into the future. Hence, to predict $n$ steps into the future, separate models for times t+1, t+2, t+3, ... , t+n would be trained. The main advantage of this approach is that not only one step ahead error is minimized, but also the error for each step into the future. However, this may involve training a huge number of models and hence, will be very time consuming [17].

In order to overcome these problems a *multiple-input multiple-output (MIMO)* strategy was introduced by Bontempi [2]. In this case, instead of producing a single scalar, a vector of future estimate values is produced. This approach allows preservation of stochastic dependency characterizing time series, which facilitates to model the underlying dynamics of the time series [17]. This fact makes this strategy incredibly promising and should be able to outperform the other two. However not every algorithm is able to utilize the MIMO strategy, therefore different strategies will be used for different algorithm.

## 2.7 Python Scientific Stack

Python is one of the most popular programming languages for machine learning and data science in general. In contrast of Matlab, Python is free to use therefore is rapidly gaining popularity. Moreover Python has a big community of open source developers that provide a huge number of useful libraries. This sections describes in a little more depth all the libraries that will be used throughout the project.

### 2.7.1 Scipy and NumPy

SciPy and NumPy were developed in order to increase the performance for computationally-intensive tasks in Python. They provide fast vectorised operations on multidimensional arrays by abstracting implementation in lower level languages such as C and Fortran.

### 2.7.2 Pandas

Pandas is a library for data manipulation and analysis. It is built on top of NumPy and provides higher level data manipulation tools for working with tabular data more conveniently. It will be mainly used for importing and initial manipulation of data imported from spreadsheets (.csv files).

### 2.7.3 Matplotlib

Matplotlib is a library providing an object-oriented API for embedding plots. This tool will be used to visualise the predictions made by machine learning algorithms, for initial, intuitive evaluation of performance.

### 2.7.4 Scikit-learn

Scikit-learn is the most popular open source, machine learning library for Python. It provides a user-friendly API, with number of highly optimised implementation of for both classification and regression algorithms. Moreover it provides functionality for data preprocessing, feature selection, model evaluation and hyperparameter tuning. It is built on top of NumPy, SciPy and matplotlib.

### 2.7.5 Jupyter Notebook

Jupyter Notebook is server-client application that allows editing and running notebook documents via web browser. It can be installed on a local server and accessed without the Internet connection or on remote server and accessed through the Internet. Each "notebook" is a document that contains both computer code and rich text elements. This fact makes the Jupyter Notebook extremely useful tool as it can contain both executable data analysing code, and human-readable description and results in the form of tables or figures. All these features made this tool extremely popular in the Python machine learning community.

The fact that Jupyter Notebook can be accessed via web browser allows running

multiple notebooks in parallel. This features will be utilised by using Amazon Web Services (AWS) as training times start to increase.

### 2.7.6 TensorFlow

TensorFlow is an open source library created by Google for conducting machine learning and deep neural network research. It provides numerical computation resources using data flow graphs. Each node is a mathematical operation, while edges represent multidimensional data array or Tensors. Moreover it provides functionality for utilizing both CPU and GPU architectures.

### 2.7.7 Keras

Keras is a Python library for deep learning built on top of TensorFlow or Theano. It was started in 2015 and by now is one of the most popular libraries that faciliates neural network training. The main goal of Keras is to allow easy and fast prototyping, while being able to utilise both CPU and GPU architectures for fast training times. It provides very minimalisting, user-friendly API, that allows building a neural networks with just a few simple lines of code. Keras provides a number of neural layers, cost functions, optimisers, initialisation algorithms, activation function and regularisation schemes modules that can be easily linked together with as little restriction as possible. Moreover, it is easily extendable, user can provide their own implementations for each module listed above.

# Chapter 3

# Data Harvesting

The first stage of the project, requires collecting river level and weather data from external sources upon which, the predictive model will be trained. In this chapter the process of data collection and storage will be described in detail. Data will be collected from The Scottish Environmental Protection Agency (SEPA), which publishes real time measurements river level data, and OpenWeatherMap for current weather information. Eventually, the data will be stored in a PostgreSQL database. The code for the data collecting application is available in Appendix A.

## 3.1 Database Tables

For data storage PostgreSQL (Postgres) database will be used. Postgres is a object-relational database, which is a database management system similar to relational database with addition of object-oriented support. That means, object, classes and inheritance are supported withing the database schemas and queries.

The data is organised into four distinct tables, with primary and foreign keys to build relations between them. The four tables are:

1. **RiverStation**: This table represents measurement collecting station for river levels. It contains the name, primary key and location of the station

both in the easting/northing and latitude/longitude format.

2. **RiverLevel**: This table represents each measurement collected by a station. It contains a primary key, sampling time, archivisation time, the actual river level measurement in meters and a foreign key representing a RiverStation at which the measurement was taken.

3. **WeatherStation**: This table represents a meteorological station at which weather information was collected. This table contains the name of the station, it's longitude and latitude and a primary key.

4. **CurrentWeather**: This table represents an actual weather information collected at each meteorological station. It contains information about sampling time, archivisation time, precipitation, temperature, wind direction and speed and a short description of the weather. Moreover it contains a primary key and a foreign key which represents the meteorological station at which the information was collected.

By providing foreign keys in both RiverLevel and CurrentWeather tables relations are built. Hence there exists a one to many relationship between RiverStation and RiverLevel and one to many relationship between WeatherStation and CurrentWeather tables. This relations and the actual information stored in each table are shown below on figure 3.1.

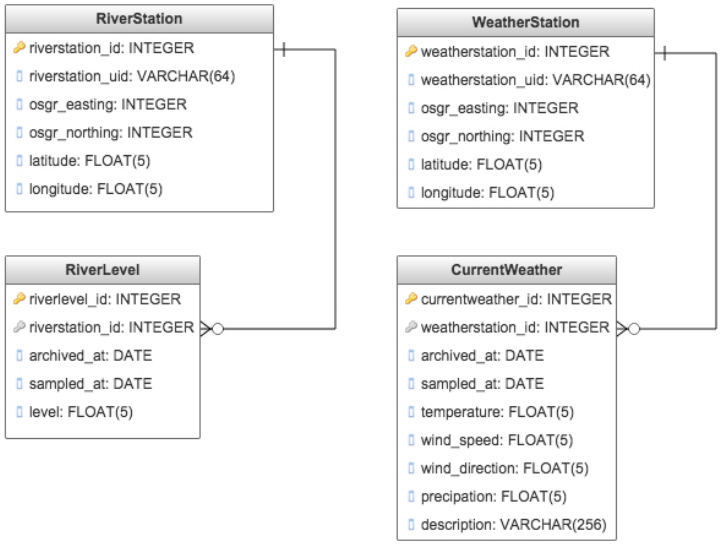


Figure 3.1: SQL database tables for river level and weather data

## 3.2 Data Collecting Application

The data collecting application is set up as a *Flask* application, for the purpose of further extension to display the results as a web based solution.

Flask is a Python web micro-framework based on Werkzeug toolkit. It is designed for quick development and extensibility. It provides a solid core for building web application, while allowing the developer to choose their own packages for more sophisticated functionality.

The application is making use of *SQLAlchemy* package for working with PostgreSQL database. SQLAlchemy is a Object Relational Mapper (ORM), which means it allows the developer to represent data as objects which are then translated to database rows. An ORM provides high-level object-oriented operations into low-level database instructions [8].

The application is also making use of *APScheduler* package. Which allows to schedule jobs that run in the background. The data collection is implementing as jobs, that run a selected script at a specified time interval. A simple schema of the application is shown below in figure 3.2

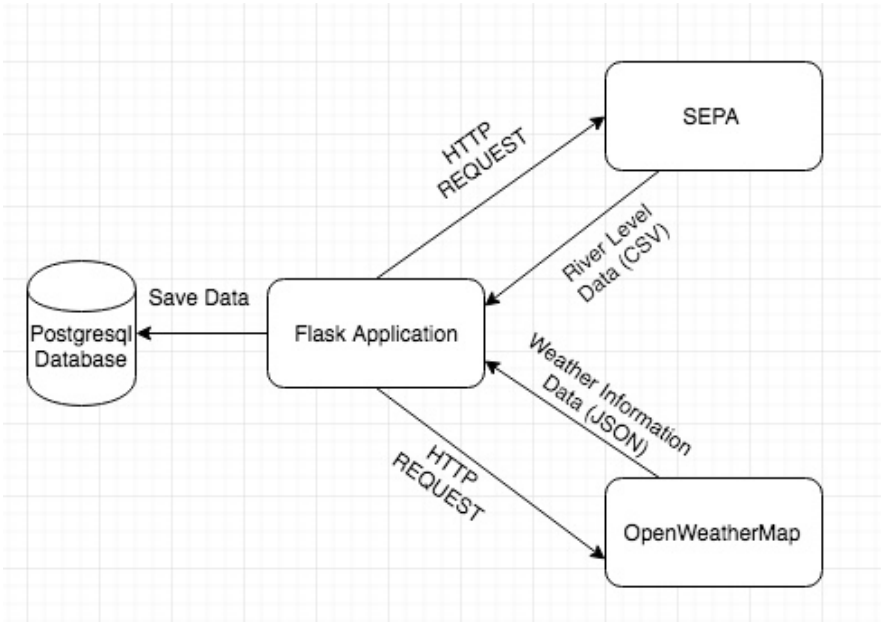


Figure 3.2: A simple schema of the data collecting application.

The river level data is collected from Scottish Environmental Protection Agency

(SEPA) website as a spreadsheet that has to be preprocessed in order to extract the required information. The spreadsheet is collected from the SEPA website using a simple HTTP request and then processed. The script preprocessing the data makes sure that entries are not repeated. Furthermore, SEPA provides location only in the form of easting and northing, thus a separate script translating those to longitude and latitude is implemented.

The weather information data is collected from OpenWeatherAPI provieded by IBM. Collection of weather information is set up as a separate job, and analogically the data is collected via a HTTP request. Weather information is returned in *JavaScript Object Notation(JSON)* format which is very easy to work with and does not require additional preprocessing.

## 3.3 Deployment

The data collection application was deployed onto a Linux server provided by the University. The process of deploying the application is described in this section.

### 3.3.1 Connecting to the server

The server is running inside the University, and hence is being protected from external access. The first step requires accessing the University Virtual Private Network (VPN). A VPN is an extension of a private network which enables access through the Internet. This approach allows the network to benefit from functionality, security and management of the private network, while allowing external access for a certain group of users, in this case students and University employees. In order to connect to Strathclyde VPN from a Mac OS operating system the following steps need to be followed:

1. Go to *System Preferences* and choose the *Network* section.
2. Click on the + symbol on the bottom and select a VPN interface.

3. Set VPN type to *Cisco IPSec*.
4. Click *Create* to add a new VPN connection to the list.
5. Specify the server address to 'vpn1.cc.strath.ac.uk'.
6. Fill in the username and password.
7. Go to *Authentication Settings* and specify the shared secret to 'point1' and the group name to 'entry'.
8. Click 'Connect' to connect to the VPN.

After successfully connecting to the University VPN, the server can be accessed using *Secure Shell (SSH)*. SSH is a cryptographic protocol that allows secure access to remote server, through insecure networks. In order to access the server the following command is used:

```
ssh <username>@<ip address of the server>
```

The user will then be prompted for a password. At this point remote access to the server is established.

### 3.3.2 Installing required packages

In order to run the application, there are a few Python packages that have to be installed. Due to University restriction root access is required to do so. In order to switch to root the "su" command is used, the user will be then prompted for the password to the root user.

Afterwards the package repository has to be updated using the following command:

apt-get update

Now, all the required packages can be installed using the "apt-get install" command. Packages to be installed include: python-pip, python-dev, build-essentials, git and postgresql.

apt-get install python-pip python-dev build-essentials git postgresql

Once pip is installed, the virtualenv package has to be installed. Pip is a python package manager, while virtualenv is a Python package for creating virtual environments. Virtual environment is a private copy of Python interpreter, which allows to install python packages privately, so that no clutters and version conflicts occur with the system's Python interpreter. In order to install virtualenv the following command is used:

pip install virtualenv

At this stage all the required packages are installed, now the application has to be imported, the database has to be set up and the application has to be run on the server.

### 3.3.3 PostgreSQL setup

Once PosgreSQL is installed, the database has to be set up. In order to do so, user has to be switched to *postgres*.

sudo -i -u postgres

Firstly a new role has to be created by typing the following command which will prompt the user for all required information:

createuser –interactive

Once the role is added, a database has to be created using the following command:

createdb $< name >$

where, $< name >$ is the name of the database. Now in order to initialise all tables,*flask_script* package is used. The following commands have to be executed in order:

./run.py db init ./run.py db migrate ./run.py db upgade

The database now contains all the tables and is ready to be used by the application.

### 3.3.4 Importing the Application

First a new virtual environment has to be initialised, using the "virtualenv" command and specifying the name to "flask":

virtualenv flask

Importing the application will be done using git from the remote github repository. In order to do so the following command is used, with url set to the url of github repository:

git clone $< url >$

Now, all the dependacies required by the application have to be installed. This is done by firstly activating the virtual environment:

source flask/bin/activate

The dependancies are specified in the *requirements.txt* file, and can be installed using pip package managed by using the following command:

pip install -r requirements.txt

### 3.3.5 Running the application

In order to run the application in the background after exiting the ssh session, *tmux* package is installed.

apt-get install tmux

*tmux* is a terminal multiplexer, which allows running multiple programs in the same terminal. After exiting a tmux session the program keeps running in the background. Firstly a new tmux session has to be created by using the "tmux" command. The application is then run using the following command:

./run.py runserver

Afterwards to exit the tmux session, "ctrl+b", followed by "d" has to be pressed. Now the ssh session can be ended, and the application will keep running in the background.

In order to access the tmux session again the following command is used:

```
tmux attach
```

At this point the application is up and running, collecting the data for training the predictive models.

# Chapter 4

# Recurrent Neural Networks

In this chapter the use of artificial neural networks for time series prediction will be investigated. More accurately, Recurrent Neural Networks with Gated Recurrent Units. Firstly, some theoretical aspects of neural networks will be explained. Afterwards an experiment will be carried out, using Keras deep learning library.

## 4.1 Background

This section contains an introduction to both regular artificial neural networks and recurrent neural networks. Afterwards some additional concepts that, related to neural network training, that will be used in the experiment section are explained. All these concepts are crucial for understanding the training process and aim to improve the performance of the network.

### 4.1.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) can be described as computational models, that aim to mimic biological brain. They consist of a set of artificial neurons, also referred to as nodes or units, and a set of edges between them. Edges are equivalent to synapses in biological neuron. Each edge has a weight associated with it, i.e each input to a neuron has a corresponding weight coefficient.

The simplest ANN is called *Multi Layer Perceptron (MLP)*, which is an example of a *Feed Forward Network (FNN)*. Such network does not allow cycles, and the information is passed only in one direction. Input is passed to the input layer, then propagated through each hidden layer respectively, towards the output layer. ANNs with cycles are referred to as feedback, recursive, or recurrent neural networks [7] and will be described in section 4.1.2. A simple MLP is shown below, in figure 4.1:

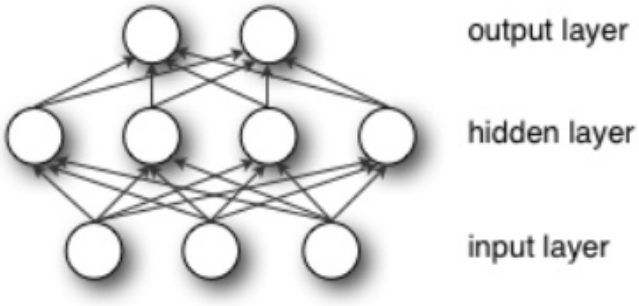


Figure 4.1: Multi Layer Perceptron with one hidden layer. Source: [10]

As shown in Figure 4.1, in a feedforward network, neurons are organised into layers, each containing a variable number of neurons. A network with one hidden layer is called a single layer network, even though it actually consists of two layers: hidden and output layer. Any network with more than one hidden layer is considered a *Deep Neural Network* [13].

The process of propagating information through the network is known as the *forward pass*. The activation of a hidden unit $h$ is calculated by applying an activation function to a weighted sum of inputs as shown in equation 4.1.

$$a_h = \phi(\sum_{i=1}^{I} w_{ih} x_i) \tag{4.1}$$

In equation 4.1 $a_h$ denotes activation of unit $h$, $\phi(*)$ is an activation function such as hyperbolic tangent or logistic sigmoid, $w_{ih}$ denotes as weight from unit $i$ to unit $h$ and $x_i$ is the input vector. For hidden/output layer the input would be an activation passed from previous layer of the network.

In order to train an ANN a suitable, differentiable loss function has to be introduced. In case of a regression problem a Mean Squared Error is an example of such and will be introduced in detail in section 4.1.6. The network is then trained by minimising a loss function using *gradient descent algorithm*. The idea is to differentiate the loss function with respect to each weight, and adjust the weights in the direction of negative slope [7]. Gradient Descent will be described in section 4.1.7. For efficient training of a network an algorithm called *backpropagation* is used, which can be simply described as repeated application of chain rule for partial derivatives, that computes the values for weight updates.

### 4.1.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) can be considered feedforward networks with feedback loops [13]. Activation of each node in the hidden layer is influenced both by the current external input, and the hidden layer activation from the previous time step [7]. Recurrent connections allow the information about previous time steps to persist in the network's internal state and hence influence the output. More formally the output of a recurrent node can be calculated by applying equation 4.2:

$$a_h^t = \phi(\sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} a_{h'}^{t-1}) \tag{4.2}$$

In equation 4.2, $a_h^t$ represents activation of node $h$ at time $t$, $x_i^t$ is the $i^{th}$ input at time $t$, I is the number of input units, H is the number of hidden units.

In order to calculate the complete sequence of hidden activations 4.2 has to be applied recursively starting at t=1, incrementing t at each timestep [7].

For training an RNN an algorithm called *backpropagation through time (BPTT)* has been introduced by Williams and Zipser in 1995. As with FNN, BPTT consists of repeated application of the chain rule. The difference is that the loss function depends on the activation of the hidden layer not only through its influence on the output layer, but also through its influence on the hidden layer at the next timestep [7].

An interesting aspect of Recurrent Neural Networks is their ability to operate over sequences of vectors. Both input and output can be a vector of consecutive values overtime. As shown on Figure 4.2 Recurrent Neural networks can map variable size input to variable size output. In case of time series prediction, many to one relationship would be used in iterative and direct strategy, while the fourth from the left (many to many) variant, would be used for the MIMO strategy.

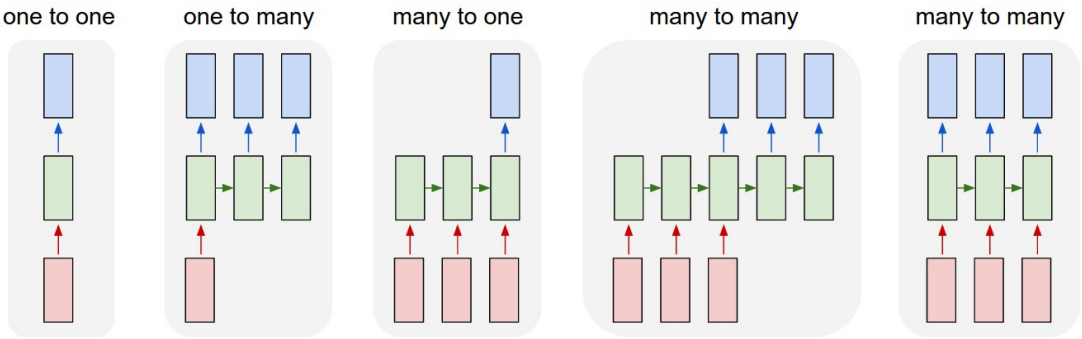


Figure 4.2: Different variants of recurrent neural network input/output relations.

A major obstacle for to recurrent neural networks performance is the vanishing gradient problem [9], discovered in the early 1990s. Vanishing gradient problem could be simply described as a difficulty to capture long-term dependencies [5]. The problem arises due to the fact that the influence of an input on both hidden layer and output, decays or blows up exponentially as it cycles through the network. The problem is shown in figure 4.3.

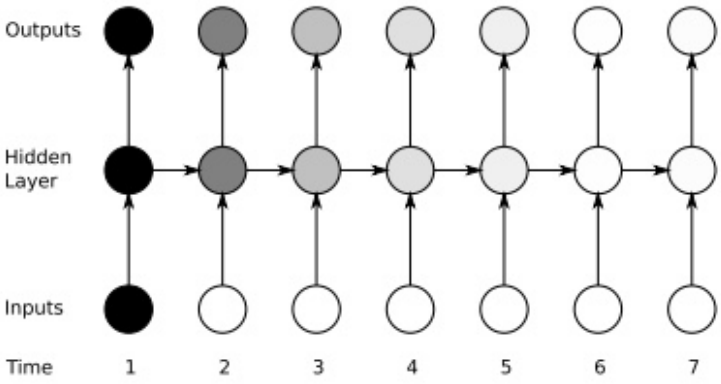


Figure 4.3: The vanishing gradient problem for RNN. Source: [7]

In Figure 4.3 the shading indicates the sensitivity to the input at time one. The

influence decays overtime, as new inputs overwrite the activations of the hidden layer [7].

There are two most common solutions to this problem. The first called *Long-Short Term Memory (LSTM)* was introduced by Hochreiter and Schmidhuber [9]. LSTM allows the long term dependencies to persist in it's internal state by having containing a memory cell. LSTM unit contains gating system, that decides what information will be passed to the internal state. The first gate is called a *forget gate* and it decides which information will be removed from the internal state. The input gate decides, which values of the internal state will be updated. While, the output gate decides what information from the internal state will be passed as the output of the unit. he second described in the used in this chapter is a *Gated Recurrent Unit* (GRU) [4]. GRU could be simply described as LSTM without the output gate, thus all the information from internal state is passed to the output. Due to better performance the GRU gate was chosen over LSTM during the experimentation.

### 4.1.3 Gated Recurrent Unit

Gated Recurrent Unit (GRU) was proposed in order to overcome the vanishing gradient problem i.e allow the network to capture long term dependancies.GRU uses gating units that modulates the information flow inside the unit. A simple schema of a GRU unit is shown on figure 4.4.

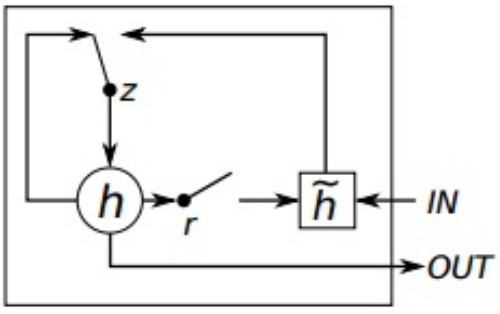


Figure 4.4: Gated Recurrent Unit. Source: [5]

Activation in each node at time t, is a linear interpolation between the previous

activation and a *candidate activation*.

$$a_h^t = (1 - z_h^t)a_h^{t-1} + z_h^t \bar{a}_h^t \tag{4.3}$$

In equation 4.3, $a_h^t$ describes activation of node $h$ at time $t$. $z_h^t$ is an update gate, that decides how much a node updates its activation or content.

The update gate for unit $h$ can be computed using the following equation:

$$z_h^t = \sigma(W_z x_t + U_z h_{t-1})^h \tag{4.4}$$

The candidate activation function $\bar{a}_h^t$ is computed as follows:

$$\bar{a}_h^t = tanh(W x_t + U(r_t \odot h_{t-1}))^j \tag{4.5}$$

where, $r_t$ is a set of reset gates and $\odot$ is an element-wise multiplication. So if $r_h^t$ is close to zero, the reset gate 'forgets' the previous state and effectively treats the input as a first sample of a sequence. The reset gate $r_h^t$ is computed similarly yo update gate:

$$r_h^t = \sigma(W_r x_t + U_r h_{t-1})^h \tag{4.6}$$

### 4.1.4 Activation functions

Activation function is a function that is applied to the net input of a neuron in order to produce its output. It is vital for backpropagation algorithms that the activation function is differentiable. A few important activation functions, that will be used in this chapter are:

1. Identity function (Linear) : $f(x) = x$
2. Hyperbolic Tangent : $f(x) = \frac{2}{1+e^{-2}} - 1$
3. Hard Sigmoid : linear approximation of sigmoid function $f(x) = \frac{1}{1+e^{-x}}$

### 4.1.5 Weight Initialization

It is common for gradient descent algorithms to require weight initialization, usually to small random values. The algorithm used in this chapter was described by Glorot [6]. It uses uniform distribution scaled by the sum of number of inputs and number of outputs of a node.

### 4.1.6 Objective (Loss) Function

Defining a right objective function that will be optimized during the learning process is crucial for supervised machine learning algorithms. One of the most common cost functions for regression problems is *Mean Squared Error (MSE).* Hence it will be used as a loss function during neural network training in this chapter. MSE can be defined as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - Pred_i)^2 \tag{4.7}$$

Where $Y$ is a vector of observed values, and *Pred* is a vector of predicted values.

### 4.1.7 Gradient Descent

Gradient Descent is an algorithm used to minimize an objective function of a neural network. Weights are updated the opposite way to the gradient of an objective function $\nabla J(\theta)$. So the weight update can be described as:

$$w = w + \Delta w \tag{4.8}$$

Where $\Delta w$ is defined as the negative gradient multiplied by the learning rate $\eta$ as shown in equation 4.9:

$$\Delta w = -\eta \nabla J(w) \tag{4.9}$$

There are three different variant of gradient descent. The difference between them is the amount of data being used to compute the gradient of the objective function. The three variants are:

1. Batch Gradient Descent (BGD): This variant uses the whole dataset to perform one weight update by accumulating the error over all samples. Batch Gradient Descent is guaranteed to converge to the global minimum for convex error surfaces and to local minimum for non-convex surfaces.

2. Stochastic Gradient Descent (SGD): This variant performs a weight update after each single training example. By updating the weights much more frequently, this SGD converges much faster than BGD. Moreover due to the error surface being much noisier than with BGD, this approach is also able to escape local minima much easier [13]. SGD can be used for online learning. Whenever new data arrives, the model is updated "on-the-fly".

3. Mini-batch Gradient Descent: It is a compromise between the two previously described approaches. The weight update is done after each mini-batch of n samples. It converges much faster than BGD due to more frequent updates, while also improving the computational efficiency of SGD by replacing a for-loop with a vectorised operation [13]. This approach is most common for neural network training.

Gradient descent algorithm faces many challenges that prevent it from good convergence. For instance choosing a proper learning rate or getting trapped in numerous suboptimal local minima, when the error function is highly non-convex. In order to to address these issues optimisation algorithms were introduced by the deep learning community. The recommended optimisation algorithm by Keras for Recurrent Neural Networks is RMSProp, therefore it will be used during model training.

### 4.1.8 Holdout cross validation

Cross-validation is a useful technique for estimating model's generalization error, i.e how well the model performs on unseen data. Particular type of cross validation that will be used in this chapter is called *Holdout Cross Validation*. This method involves splitting the data set into three parts: a training set, a validation set and test set. The purpose of a training set is to train the models. Validation

set is used for model selection, i.e selecting a model with most optimal hyperparameters. Finally, the test data set is used to estimate the model's generalization error. That approach reduces the likelihood of overfitting that could be the effect of reusing the same test set for both model selection and final evaluation ([13]).

A main drawback of this technique is the fact that part of the training data has to be sacrificed for validation. This can lead to reduced performance on small data sets ([7]).

### 4.1.9 Early Stopping

Another technique for improving generalisation that is used in this chapter is called *Early Stopping*. Early stopping requires, existence of a validation data set, thus goes hand in hand with *Holdout cross-validation* technique described in section 4.1.8 . After each epoch the performance of the model is evaluated by applying an appropriate error function to the validation data set. In case of the problem faced in this report, the Mean Squared Error function is used. Then, the 'best' set of weight is chosen, i.e the one that minimizes the error function.

While training a model, the error on all data sets usually decreases at first on all three subsets: train, validation and test. However, after a while the error on train data set continues to decrease, while the validation and test error increases ([7]). This is when the model starts to overfit. Early stopping prevents the occurrence of this scenario.

## 4.2 Experimental Procedure

In this section a predictive model will be build using Recurrent Neural networks implemented using Keras deep learning library described in section 2.7.7. The procedure described in section 2.3 will be followed, each step in a separate section. For now, only the raw river level data will be used for model training, the weather data will be omitted. Once a predictive model is selected, training and prediction

stage will be repeated for ten different monitored rivers. Hence, final evaluation will be done by comparing results for all of these rivers. The experiment will be carried out in a virtual environment using Jupyter Notebook. The nature of recurrent neural networks allows the implementation of MIMO strategy described in section 2.6, and hence it will be used as the most promising strategy. The code for this section is available in Appendix B.

### 4.2.1 Data Preprocessing

The first step of a machine learning process involves data preprocessing. As the performance of machine learning algorithms is highly dependant on the underlying data, the same dataset has to be used for training, so that model selection is done solely by comparing algorithms. Hence, the content of the database is exported to a spreadsheet file and then the data is loaded into the Jupyter Notebook environment from it. Note that only raw river level data is loaded and weather information data is omitted.

Once the data is loaded, it has to be divided into separate variables, each containing only the river levels for a particular river. This is achieved by selecting river data with a specific *riverstation_id* property. Furthermore, as only raw river data is used, the *river_level* property from each sample has to be chosen.

Afterwards, the data has to be standardized. That is all features will be rescaled so that they have properties of standard normal distribution with zero mean and unit variance. This is achieved using scikit-learn's *scale()* function available in the preprocessing module. The mean and standard deviation for each river have to be stored in order to rescale the data to original values after the model training process.

As the weather data is omitted, there is no need to perform dimensionality reduction. In the next step the data will be split into train and test subsets. 80% of the data will be used for training, while the remaining 20% will be used for model evaluation. Part of the training set will be used as validation dataset, as described in section 4.1.8. Hence data for each river is split.

Afterwards, both subsets have to be split into input and output vectors. The input vector will be a windows of 64 previous values, based on which 20 following timesteps will be predicted. The window size is an arbitrary choice. Hence, for some timestep $t$, the input sequence would contain river level values:

$$[level_{t-64}, level_{t-63}, ..., level_t] \tag{4.10}$$

And the output sequence would contain the following values:

$$[level_{t+1}, level_{t+2}, ..., level_{t+20}] \tag{4.11}$$

Where, *level* is the measured river level at a specific timestep.

Finally, Keras recurrent layers require a particular input format - [samples, time steps, features]. In order to ensure that Keras will treat our window of values as consecutive timesteps of one feature, the size of the window is passed as "time steps". If the size of the window was passed as features, Keras RNNs would treat timesteps as separate features. As a time series is being analysed the former approach seems as a more accurate formulation of the problem. A NumPy *reshape* method is used in order to transform the data into the right format

At this stage the data is in the right format to be used for model training.

### 4.2.2 Model Construction and Learning

As the MIMO strategy is being investigated in this experiment, a model capable of mapping a sequence input to sequence output has to be constructed. The model has to produce an output as shown in the fourth case from left in the figure 4.2. As Keras provides a very simple and minimalistic API, the model construction is very simple. Sequential model will be used to combine Dense, GRU, Dropout, Activation and RepeatVector layers into a deep neural network. First a Sequential model is initialised. Afterwards the following layers will be added:

1. An encoder layer consisting of 16 GRU units: This layer will encode the information held by our data into the network state. Note, the input di-

mension has to be set to 1, as the input vector was defined as a set of consecutive timesteps of one feature, rather than separate features.

2. A RepeatedVector layer: The RepeatVector repeatedly provides the last hidden state of the network for each timestep to the *decoder* layers. This layer takes the number of repeats as an argument. In this case, it is set to 20.

3. Two decoder layers, each consisting of 8 GRU units: These layers will take the input from the RepeatVector layer, and decode the values in order to produce the actual output sequence. The *return_sequences* parameter of these layers has to be set to True, as the output of these layers has to remain a vector.

4. TimeDistributed layer: This layer splits the output into separate timesteps with a Dense layer applied to each in order to produce the actual output. By default the activation function of Dense layer is set to the identity function, hence a continues output for each timestep is produced.

Finally the model has to be compiled in order to link all the layers together. The loss function is specified to Mean Squared Error described in 4.1.6. The optimized for gradient descent algorithm is set to RMSProp as suggested in the Keras documentation for RNN. It has been described in section 4.1.7 The initialization algorithm by default is Glorot's Uniform described in 4.1.5.

The model is then trained on the training subset. The model will be trained for 20 epochs, performing mini-batch training, described in section 4.1.7 with the mini-batch size of 10. An epoch is a full training cycle over the training dataset. The holdout cross-validation is used by specifying a *validation_split* property to 0.1. Hence, the validation data set is derived from the training set as the last 10% of samples. Early stopping, described in 4.1.9, is also used using a module available in Keras. It monitors the value of objective function (Mean Squared Error) on the validation data set. Min_delta parameter specifies a threshold, a minimum change of the monitored value classified as improvement, patience specifies number of steps through which there was no improvement, before stopping.

### 4.2.3 Model Evaluation

At this stage the model is trained and ready for final evaluation. All the parameters were chosen arbitrary. Due to very good performance, hyperparameter tuning is skipped. However, by performing a grid search over window size, number of GRU units in each layer, batch size and the Early stopping parameter, the performance of the algorithm could be improved.

Prediction are made using the *predict* method of Keras Sequential model. For each test sample, the model produces a vector of estimated future values from time $t + 1$ to time $t + 20$ as specified by the training dataset. Predictions are mode on the test subset in order to perform proper model evaluation and checked for possible over/under fitting.

Predictions were made on scaled data, thus the results are also scaled. Hence, both predictions and original observed data points have transformed to original values before they will be plotted.

The whole process was then repeated for ten monitored rivers. At first the results are plotted for visual evaluation. The results of four chosen rivers are shown below. Each Figure depicts four subplots, comparing values of original and predicted values, at times t+1, t+4, t+8 and t+16. Also a Mean Absolute Percentage Error (MAPE) is used as a performance metric. The code below produces the desired plots.

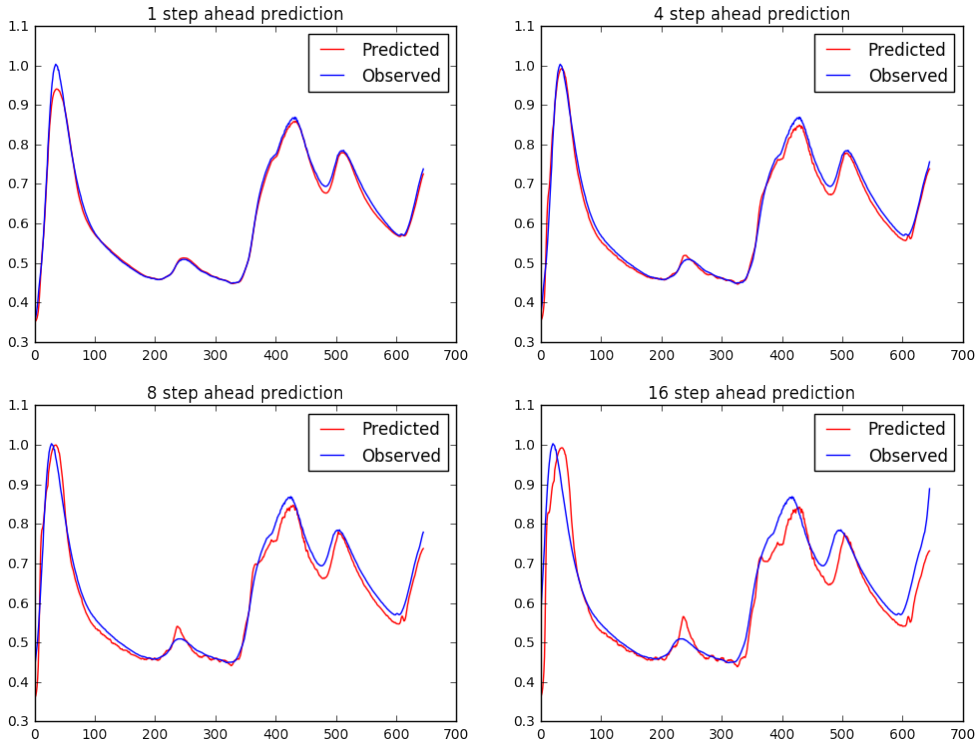


Figure 4.5: Plots depicting original and predicted data, for the test subset of ... river.

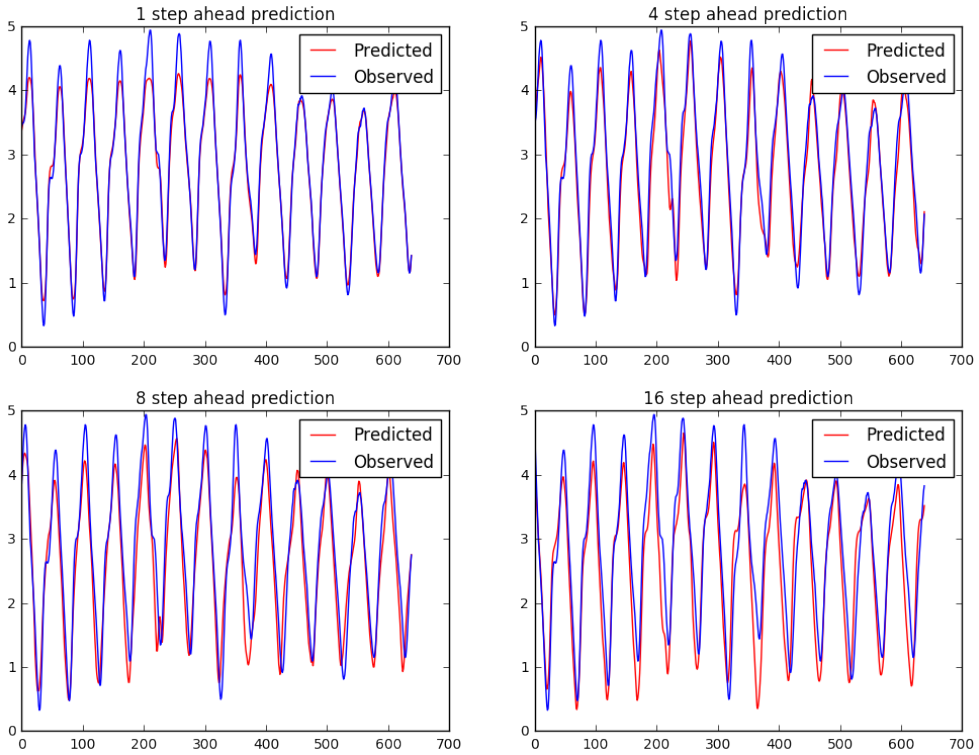


Figure 4.6: Plots depicting original and predicted data, for the test subset of ... river.

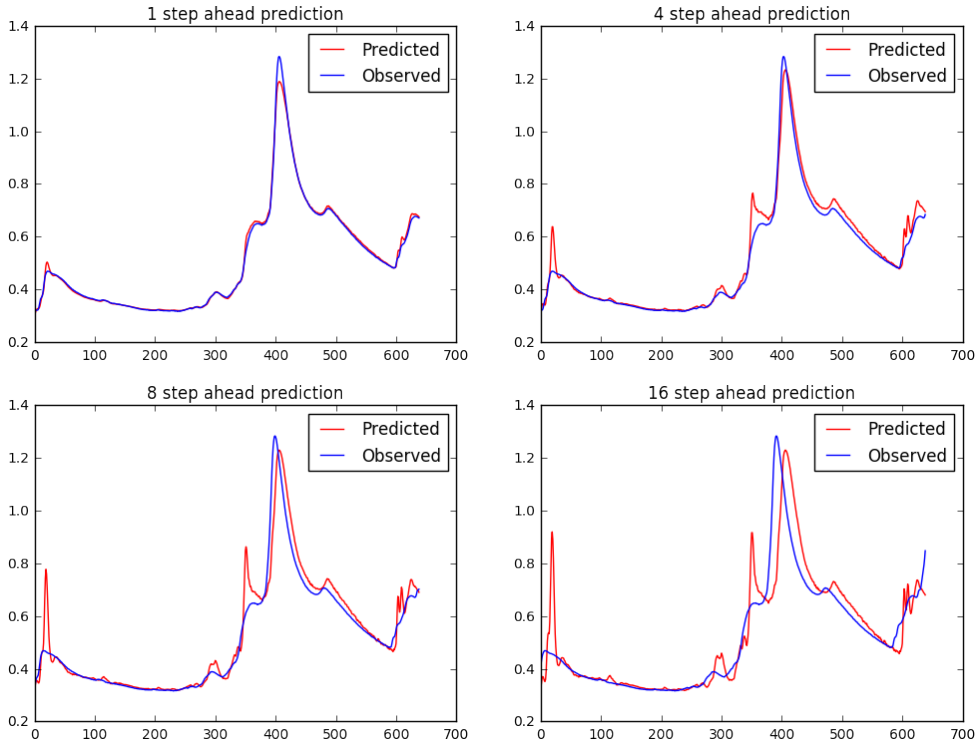


Figure 4.7: Plots depicting original and predicted data, for the test subset of ... river.

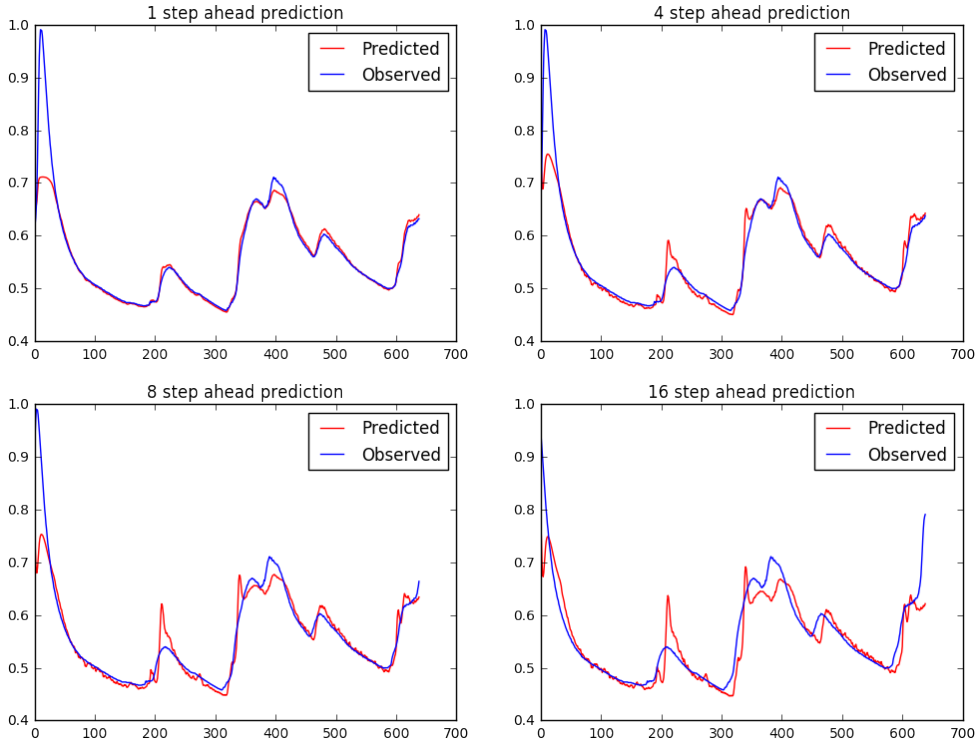


Figure 4.8: Plots depicting original and predicted data, for the test subset of ... river.

The model predicts the output sequences of the test subset very accurately. The error is very small and the modelled shape of the signal seems to be very accurate. An interesting observation can be made by inspecting the graphs in figure 4.8. The model doesn't seem to model the first maximum. In order to deduce the cause of this error, training data was examined. The training data did not include any river level measurement as high as the maximum in the test data. Therefore, it depicts very accurately how machine learning models are dependant on the training data.

| **Steps Ahead** | **Average MAPE over all rivers** |
|---|---|
| 1 | 2.156 % |
| 2 | 2.437 % |
| 3 | 2.74 % |
| 4 | 3.121 % |
| 5 | 3.516 % |
| 6 | 3.886 % |
| 7 | 4.209 % |
| 8 | 4.487 % |
| 9 | 4.744 % |
| 10 | 5.008 % |
| 11 | 5.297 % |
| 12 | 5.603 % |
| 13 | 5.91 % |
| 14 | 6.211 % |
| 15 | 6.522 % |
| 16 | 6.83 % |
| 17 | 7.134 % |
| 18 | 7.454 % |
| 19 | 7.807 % |
| 20 | 8.209 % |

Table 4.1: Average MAPE over all monitored rivers, for different number of steps ahead prediction using Recurrent Neural Networks.

Final evaluation is done by comparing MAPE for each predicted timestep into the future, for each monitored river. The average MAPE among all rivers for each timestep is computed and will be used for comparing algorithms. The results are shown in the table 4.2.3. Also a graph depicting the increase in error for each step further into the future is drawn.

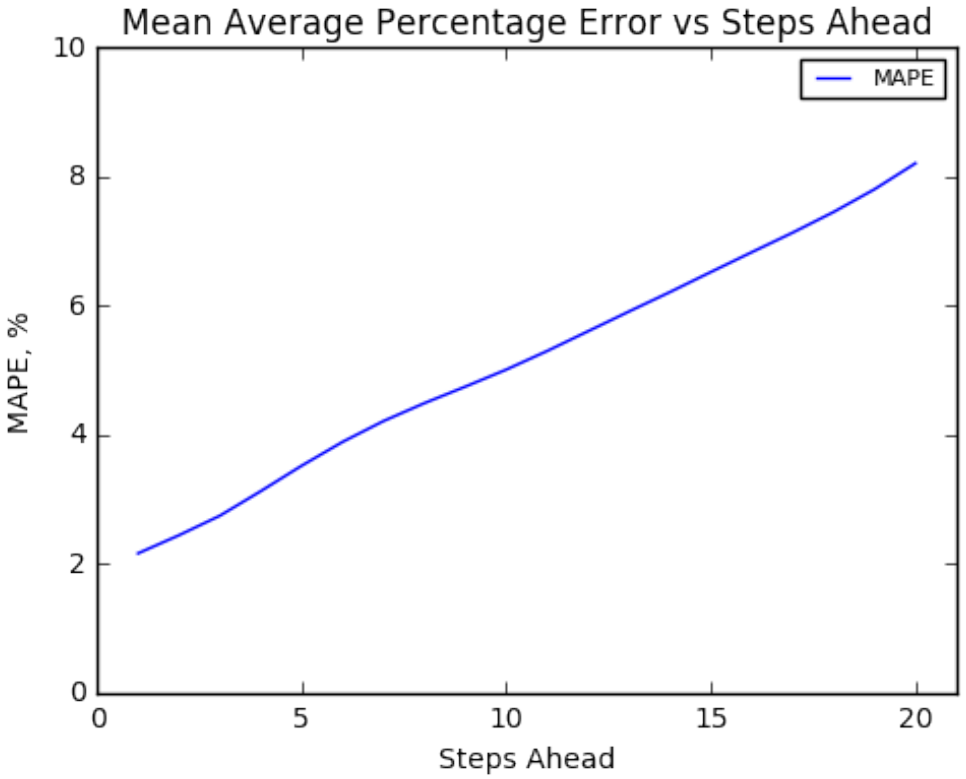


Figure 4.9: Plot of average MAPE over all rivers versus number of steps ahead, for predictions made using recurrent neural network.

As shown in table and figure 4.9, the error is increasing for each predicted step further into the future. The increase in error seems to be almost linear.

## 4.3 Conclusion

Recurrent Neural Networks algorithms has shown very good performance in the river level prediction. It is able to naturally model the time dependency in the underlying data, which makes it extremely powerful for timeseries modelling. Moreover, RNNs are capable of sequence to sequence modelling which allows the

use of MIMO strategy for timeseries prediction. This approach allows a fewer number of models, than the direct strategy, and thus increases the time required for predicting twenty consecutive timesteps.

An interesting finding was made while examining the graphs of predicted value. The importance of underlying data was proven, as one of the trained models was unable to predict values not seen in the training dataset.

Finally, the averages in table 4.2.3, clearly show the difficulty in predicting values further into the future. For each timestep into the future the error increases. Interestingly, the error seems to be increasing almost linearly.

# Chapter 5

# Support Vector Regression

In this chapter the use of Support Vector Regression (SVR) algorithm with radial basis function kernel will be investigate for timeseries prediction of river levels. Similarly to RNN, only river level data will be used for prediction. This chapter involves explaining the general idea behind SVR and associated concepts. Afterwards an experiment will be carried out. A predictive model will be constructed and evaluated leading to a final conclusion over the performance of the algorithm.

## 5.1 Background

In this section the general idea behind Support Vector Regression will be explained. Due to complexity of mathematical derivation and inclusion of quadratic programming, the mathematical concepts will be skipped. Hence, only the conceptual idea will be explained. The complete mathematical context can be found in [15].

### 5.1.1 General Idea

Support Vector Regression is one of the most common applications of Support Vector Machines and was proposed by Vapnik in 1997. SVR has found application in various fields including timeseries and financial prediction with great

successes. Opposing to traditional regression algorithms, which aim to produce a function $f(x)$ that minimizes the observed training error, Support Vector Regression (SVR) attempts to minimize the generalization error bound in order to achieve good generalization performance. Bound is influenced both by training error and regularization term in order to control the complexity of the hypothesis space [1]. SVR algorithm creates a model based only on a limited subset of the training data, due to the cost function, that ignores every training sample which lies within a threshold $\epsilon$ to the model prediction. Basically, given a training data $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, where n is the size of the dataset. The goal of a $\epsilon - SVR$ is to find a function $f(x)$ that has at most $\epsilon$ deviation from the actual measurements $y_i$ for the whole training data, that is additionally as flat as possible [15]. In other words, error larger than $\epsilon$ is not acceptable, every smalled error is ignored.

The issue with this approach is that a function meeting these constrains may not exists, i.e there may not exists a function that approximates all pairs $(x_i, y_i)$ with $\epsilon$ precision. Thus, a *soft-margin loss function* was developed by introducing slack variable $\zeta$. The slack variable is a way of relaxing the constraint of $\epsilon$ precision, under certain cost penalization while calculating the loss function. Figure 5.1 shows the use of slack variable graphically.

This penalization is also multiplied by a constant $C > 0$, which basicly determines the trade-off between the flatness of function $f$ and the amount by which the deviation of $\epsilon$ is tolerated [15]. Large values of C result in large error penalties for data points outside the threshold $\epsilon$, while small C results in small penalty. Therefore, C can be used to tune the bias-variance trade-off [13].

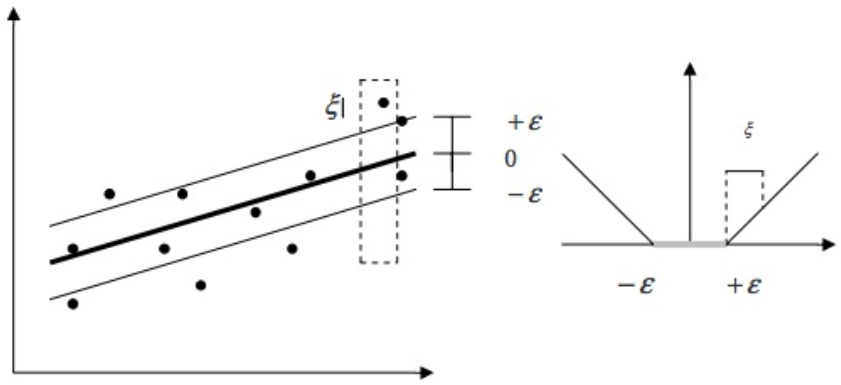


Figure 5.1: Soft-margin loss in SVR.

### 5.1.2 Kernel Trick

Another reason for the success of support vector regression algorithm is the kernelization, which enables application of the algorithm in order to solve non-linear problems. In order to solve non-linear problems, the a transformation function $\phi$ need to be defined, that maps the feature space into higher dimensionality, where the standard SVR algorithm can be applied. This is process is called the *Kernel Trick* and the transformation function is called a *kernel function.* The kernel function can be interpreted as a*similarity function* between a pair of samples.

### 5.1.3 Radial Bias Function (RBF)

One of the most popular and widely used kernel function is the *Radial Basis Function Kernel (RBF)* or Gaussian Kernel. Thus, it will be used in this chapter while carrying out the experimental procedure. The RBF can be defined as follows:

$$k(x^{(i)}, x^{(j)}) = exp(-\frac{||x^{(i)} - x^{(j)}||^2}{2\sigma^2}) \tag{5.1}$$

And is often simplified to:

$$k(x^{(i)}, x^{(j)}) = exp(-\gamma||x^{(i)} - x^{(j)}||^2 2\sigma^2) \tag{5.2}$$

$||x^{(i)} - x^{(j)}||^2$ is recognised as squared euclidean distance. Here, $\gamma = \frac{1}{2\sigma^2}$, which is a free parameter that has to be optimized. It can be understood as a cut-off parameter for the Gaussian sphere. Increasing *gamma* results in softer decision boundary. The RFB kernel function produces a similarity score that falls into range between 1 (for exactly similar samples) and 0 (for very non-similar samples) [13].

## 5.2 Experimental Procedure

In this section an experiment will carried out using Support Vector Regression algorithm implementation provided by scikit-learn library with RBF non-linear kernel. The procedure described in section 2.3 will be followed. The procedure will be repeated for each of the ten monitored rivers for which data was collected. Separate predictive models will be constructed for up to 20 steps ahead using the direct strategy described in section 2.6. The results will be presented in the form of graphs for 1,4,8 and 16 steps ahead for visual evaluation. For final evaluation a table of average MAPE for each timestep into the future over all monitored rivers will be presented. The code for this section is available in Appendix C.

### 5.2.1 Data Preprocessing

Similarly as for RNN experimental procedure, the first step involves importing the dataset into Jupyter Notebook environment. The same data as used for RNN will be imported from a spreadsheet, in order to be able to accurately compare the performance of both algorithms. Analogically, the weather information data is omitted, only raw river data is used for model training.

Furthermore, again, the data has to be divided into separate rivers, by selecting samples containing the same riverstation_id field.

Afterwards, the data is standardized using the *scale* method provided by scikit-learn's preprocessing module. The mean and standard deviation is calculated and stored for each river for the purpose of reverse scaling the data before presenting

the actual results. NumPy provides *mean()* and *std()* functions that will be used for that purpose.

The data is now split into the train and test subsets for cross-validation. Similarly as for RNN, however in this case regular cross-validation split will be done, rather than the holdout cross-validation. First 70% of samples will be used for training, while the remaining 30% will be used for testing.

And finally the data is transformed into the input vector and output scalar. The window of previous values used for prediction is set to 32. Thus, the input vector for a timestep $t$ would contain river level values:

$$[level_{t-32}, level_{t-31}, ..., level_t] \tag{5.3}$$

While the output scalar would contain the value:

$$level_{t+n} \tag{5.4}$$

Where, $n \in \{1, 2, ..., 20\}$. Depending which timestep into the future is being predicted. As specified by the direct strategy for predicting multiple timesteps into the future, a separate model has to be constructed for each timestep. Thus a different dataset will have to be constructed for each of these models. This finishes the preprocessing stage of the data.

### 5.2.2 Model Contruction and Learning

Again, as the SVR algorithm is not capable of producing a vector output, the direct strategy is used predicting 20 consecutive steps into the future. This strategy is chosen over the iterative strategy due to the accumulation of error that occurs in the latter and the speed of training an SVR model which is extremely fast. The model is constructed using the *SVR* class from scikit-learn's *svm* module. The parameters for the SVR are as C=1.0 which is a penalty parameter of the error term, epsilon is set to 0.01 specifies the epsilon tube, within which no penalty is associated in the training loss function. Kernel by default is set to radial basis function (RBF). Gamma, being the kernel coefficient, is set to 'auto' which is $\frac{1}{no\_features}$, so for our window of 32 values $gamma = \frac{1}{32}$. The use of shrinking

heuristic is set to True by default. Stopping criterion is by default set to 0.001. Maximum number of iterations is set to no limit by default.

The training of the model will be repeated for each of 10 monitored rivers and each of 20 consecutive timesteps. Thus 200 models will need to be created in total.

### 5.2.3 Model Evalutation

At this stage the models are ready, and each models performance will be evaluated on the test sunset. All the hyperparameters were chosen arbitrary, hence the performance can still be improved. For each of model, the predictions are made on the test subset using the *predict* function provided by scikit-learn SVR class.

As the dataset has been scaled, the predicted results have to be rescaled back to it's original values before being plotted. Also a MAPE error is calculated for each model, as an evaluation metric.

The whole process was repeated for all ten monitored rivers and graphs depicting predictions for 1,4,8 and 16 steps into the future, for selected rivers (the same as in RNN) are presented below.

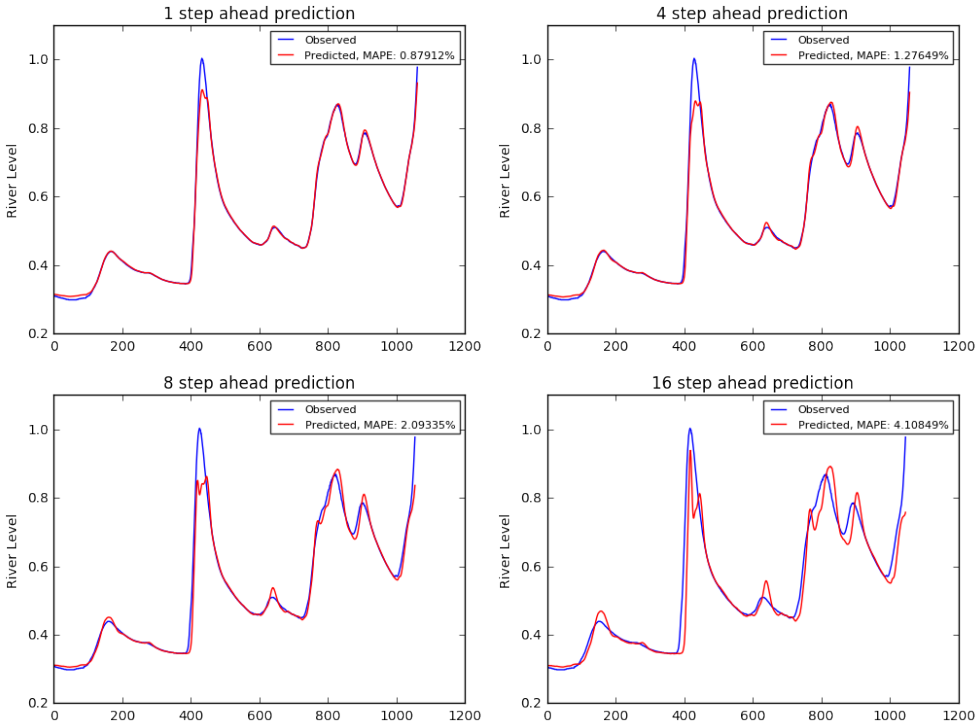


Figure 5.2: Plots showing predicted results for 1,4,8 and 16 steps ahead predictions using SVR algorithm.

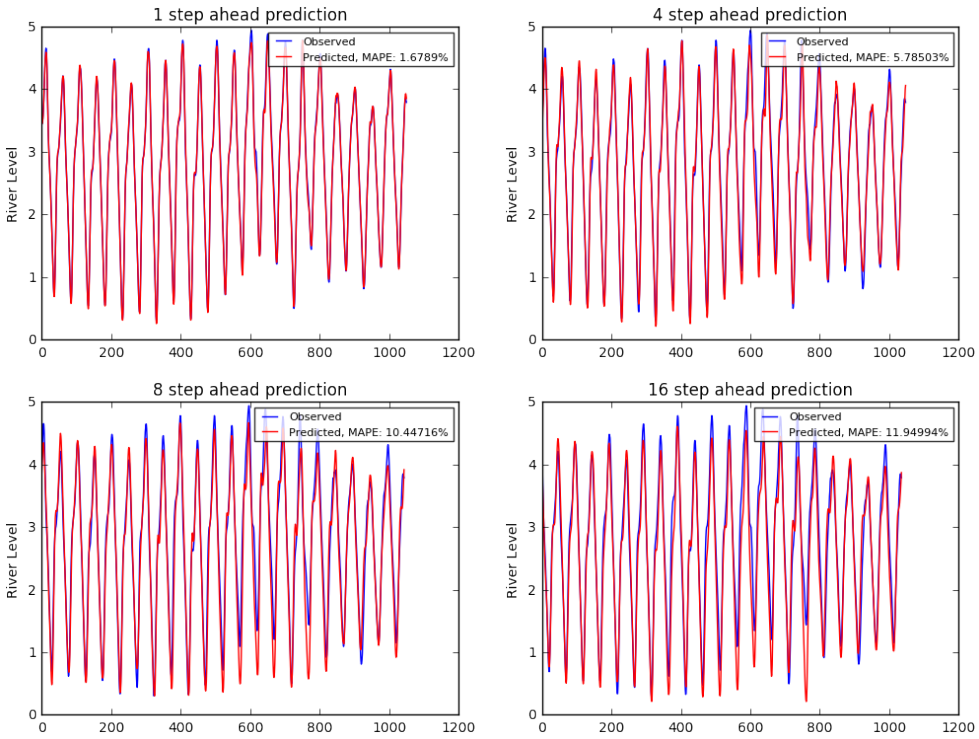


Figure 5.3: Plots showing predicted results for 1,4,8 and 16 steps ahead predictions using SVR algorithm.

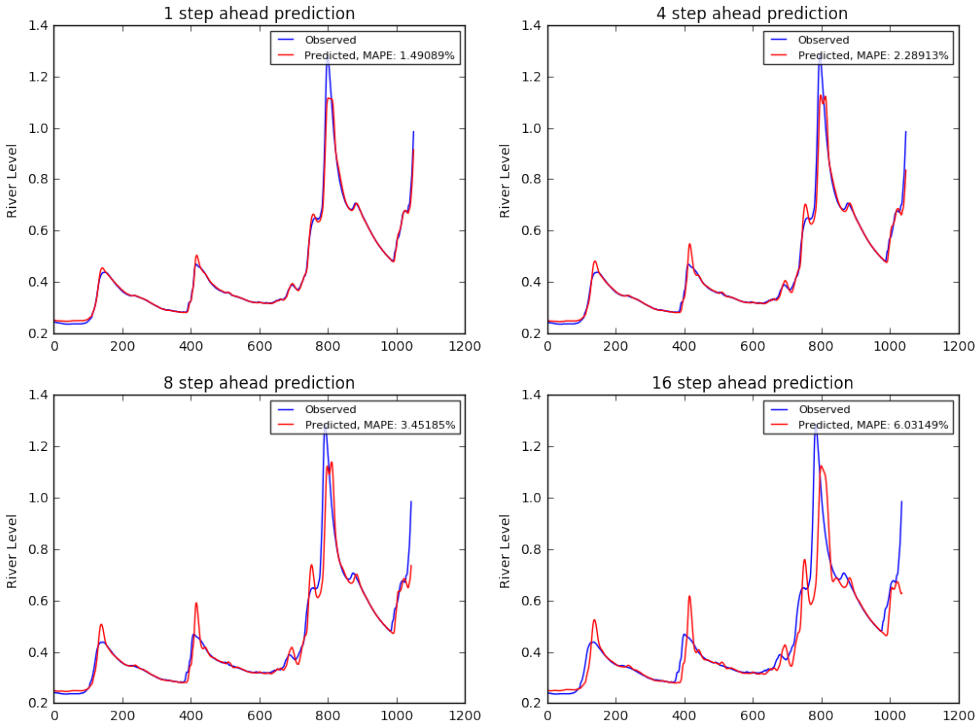


Figure 5.4: Plots showing predicted results for 1,4,8 and 16 steps ahead predictions using SVR algorithm.

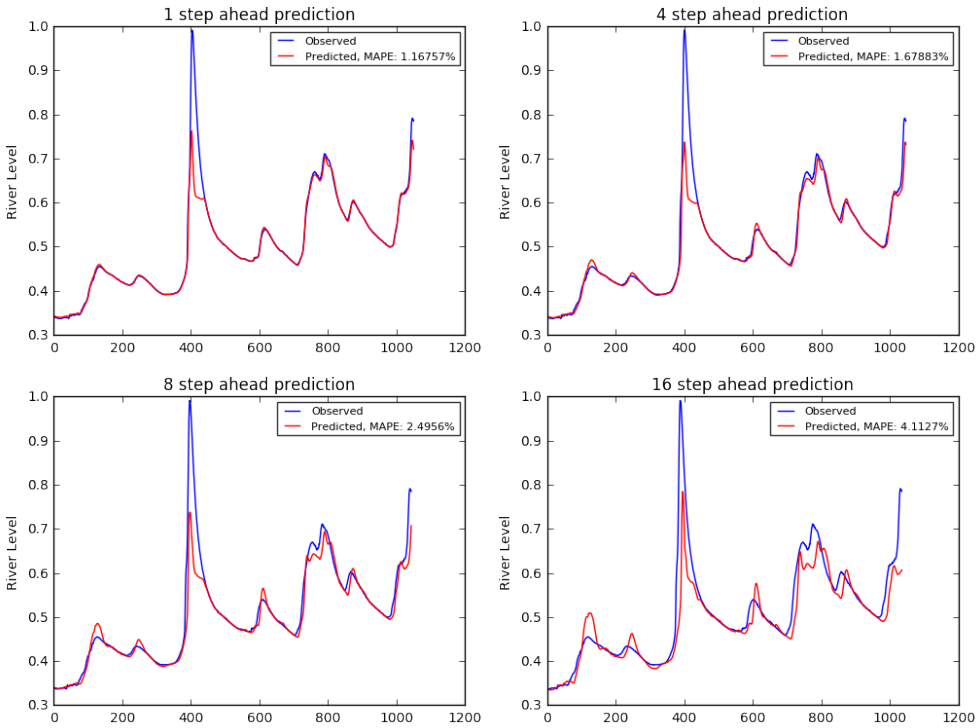


Figure 5.5: Plots showing predicted results for 1,4,8 and 16 steps ahead predictions using SVR algorithm.

The SVR algorithm seems to have similar performance to RNN. Furthermore, this algorithm encountered the same issue as RNN river levels not encountered in the training dataset.

Finally, the average MAPE over all rivers was computed for predictions made for each timestep into the future and the results are presented in table 5.2.3. The averages are then plotted on a graph (figure 5.6), to visualise the increase in error.

| **Steps Ahead** | **Average MAPE over all rivers** |
|---|---|
| 1 | 1.294 % |
| 2 | 1.591 % |
| 3 | 1.916 % |
| 4 | 2.258 % |
| 5 | 2.593 % |
| 6 | 2.939 % |
| 7 | 3.276 % |
| 8 | 3.579 % |
| 9 | 3.88 % |
| 10 | 4.15 % |
| 11 | 4.406 % |
| 12 | 4.655 % |
| 13 | 4.897 % |
| 14 | 5.134 % |
| 15 | 5.396 % |
| 16 | 5.649 % |
| 17 | 5.905 % |
| 18 | 6.155 % |
| 19 | 6.412 % |
| 20 | 6.653 % |

Table 5.1: Average MAPE over all monitored rivers, for different number of steps ahead prediction using Support Vector Regression.

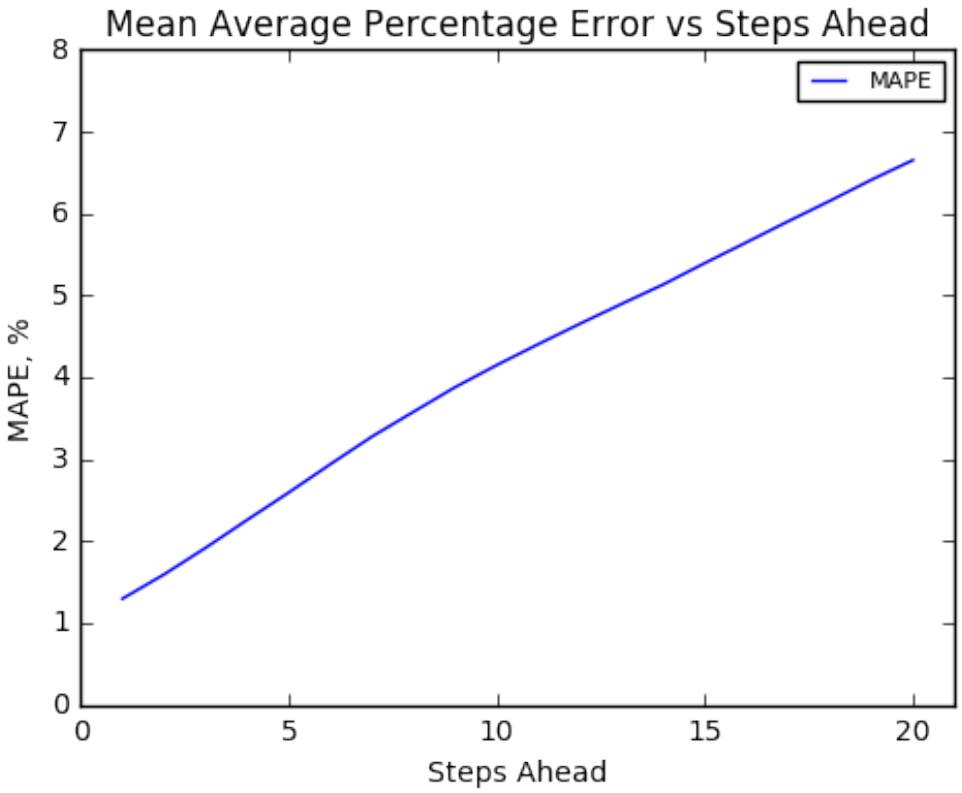


Figure 5.6: Plot of average MAPE over all rivers versus number of steps ahead, for predictions made using support vector regression algorithm.

## 5.3 Conslusion

Support Vector Regression algorithm has also proven a very good performance in timeseries prediction of river levels. The average results seem to be slightly better than in RNN. Due to very short training times for SVR models, using the direct strategy was not an issue, even though 200 different models were to be computed.

Similarly to RNN, the error increases for predicting timesteps further into the future almost linearly. Moreover, the algorithm is also strongly dependant on the training data, as it is unable to predict previously not encountered river levels.

A very interesting aspect of SVR, is the $\epsilon$ parameter, which allows us to control the error. Thus, the performance of the algorithm could be controlled, if a specific

threshold for error was specified beforehand. However, there might be a possibility that the model wouldn't be able to achieve the planned performance.

As no hyperparemeter tuning was done, the models performance could still be improved. A specific guide for hyperparameter tuning of SVR algorithm can be found in the following paper [11].

# Chapter 6

# Gaussian Process Regression

In this chapter the use of Gaussian Process Regression algorithm will be investigated for training a predictive model for river level indication. Similarly to SVR, the direct strategy described in section 2.6, will be used, due to a single scalar value being returned by the algorithm, and very quick training times. Section 6.2 describes the experimental process for building and evaluating a predictive model. Graphs for selected rivers and timesteps into the future are presented for visual evaluation. Eventually, final evaluation is presented in a table by gathering average Mean Absolute Percentage Errors for each of twenty consecutive timesteps into the future, along ten monitored rivers.

## 6.1 Background

In the section the basic idea of using Gaussian Process (GP) models for formulating Bayesian framework for regression will be explained. GP differs from previously examined algithms as it produces a non parametric model. The background is explained by following the tutorial provided in [14].

### 6.1.1 Prior Gaussian Process

A *Gaussian process* can be defined as a collection of random variables, any finite number of which have consistent *joint Gaussian distribution* [14]. Gaussian

process can be treated as a natural generalisation of the Gaussian distribution, and can be specified by a mean function $m(x)$ and a covariance function $k(x, x')$. While the distribution is defined by a mean vector and covariance matrix. Hence, function f can be defined as follows:

$$f \sim GP(m, k) \tag{6.1}$$

Equation 6.1 means: function f is distributed as a GP with mean function $m$ and covariance function $k$ [14].

In Gaussian distribution individual random variables are indexed by their position in the vector. In Gaussian process it is the argument $x$ of a random function $f(x)$ that plays the role of index set. For every input $x$ there exists a random variable $f(x)$, representing a value computed by a stochastic function $f$ at that location.

In order to work with finite quantities, the value of $f$ is only requested at a distinct finite number of n locations. Given a the input set x, the vector of means and covariance matrix can be computed in order to define a Gaussian distribution:

$$\mu_i = m(x_i), i = 1, ..., n \tag{6.2}$$

$$\Sigma_{ij} = k(x_i, x_j), i, j = 1, ..., n \tag{6.3}$$

Now a random vector can be generated from that distribution. The coordinates of the vector correspond to values of function $f(x)$.

$$f \sim N(\mu, \Sigma) \tag{6.4}$$

In equation (6.4), $N$ represent the normal (gaussian) distribution given the mean vector $\mu$ and covariance matrix $\Sigma$.

### 6.1.2 Posterior Gaussian Process

In the previous section defining distributions over functions using GPs was defined and will be used as a prior for Bayesian inference. The prior only specifies some initial properties of the functions. Computing a posterior involves updating the

properties of the prior based on the training data to be able to make predictions on unseen data. Let $f$ be the function values of the training subset and $f_*$ be the function values of the test subset $X_*$. The joint distribution can be written as:

$$\begin{bmatrix} f \\ f_* \end{bmatrix} \sim N(\begin{bmatrix} \mu \\ \mu_* \end{bmatrix}, \begin{bmatrix} \Sigma & \Sigma_* \\ \Sigma_*^T & \Sigma_{**} \end{bmatrix}) \tag{6.5}$$

As in previous section $\mu = m(x_i)$, i = 1,...,n and analogically for the test mean $\mu_*$. $\Sigma$ is training covariance, $\Sigma_*$ is training-test covariance and $\Sigma_{**}$ is test covariance. In order to make predictions a conditional distribution is now to be computed in order to make predictions on the test subset.

$$f_*|f \sim N(\mu_* + \Sigma_*^T\Sigma^{-1}(f-\mu).\Sigma_{**}\Sigma^{-1}\Sigma_*) \tag{6.6}$$

Equation 6.6 was derived from equation 6.5 using a formula for conditioning a joint Gaussian distribution and represents posterior distribution for a specific set of test cases. From 6.6 the central equations for Gaussian process prediction are be derived.

$$f|D \sim GP(m_D, k_D) \tag{6.7}$$

$$m_D(x) = m(x) + \Sigma(X,x)^T\Sigma^{-1}(f-m) \tag{6.8}$$

$$k_D(x,x') = k(x,x') - \Sigma(X,x)^T\Sigma^{-1}\Sigma(X,x') \tag{6.9}$$

Note that posterior covariance is equal to prior covariance minus a positive term, which depends on the training dataset.

Additionally, it is common for regression observations to contain noise in the observed data. Thus, this issue has to be addressed while training a model. By assuming that the noise is Gaussian, it can be easily taken into account. The effect is that every f(x) has an extra covariance with itself. Thus, the covariance function of a noisy process can be represented as the sum of the input signal covariance and noise covariance.

$$y(x) = f(x) + \epsilon, \epsilon \sim N(0, \sigma_n^2) \tag{6.10}$$

$$f \sim GP(m,k), y \sim GP(m, k + \sigma_n^2\delta_{ii'}) \tag{6.11}$$

### 6.1.3 Training a Gaussian Process

Computing the posterior Gaussian distribution updates the prior with respect to the training data. However this is useful if the prior mean and covariance functions can be specified accurately from the dataset. Unfortunately, in most machine learning applications this is not the case, and a process of selecting the right mean and covariance functions is crucial. This process could be referred to as training a GP model. The mean and covariance functions are parametrised in terms of hyperparameters which will be then adjusted during training. Selecting the right mean and covariance function types allows specifying vague prior information about the dataset.

The hyperapameters are tuned by computing and optimising the *log-marginal-likelihood*, which in other words is the probability of the given data given the hyperparameters, assuming the distribution of the data is Gaussian. The optimisation is done using gradient based methods. Note that the model produced by Gaussian process is non-parametric, there are no weighting parameters that need to be set. This fact significantly simplifies training, as the Occam's razor is automatically incorporated by the marginal likelihood. In machine learning Occam's razor could be interpretted as favouring simpler hypothesis over complex ones.

A mathematical context of the training process can be found in [14].

## 6.2 Experimental Procedure

In this section an experiment will be carried out using Gaussian Process Regression (GPR) algorithm implementation in scikit-learn library. As GPR is another kernel based algorithm, similarly to SVR, the RBF kernel will be used. GPR is also only capable of producing a single scalar output, thus the direct strategy will be used. Separate predictive models will be constructed for each of 20 steps ahead predictions. The process will be repeated for 10 monitored rivers. Afterwards the final evaluation will be done by computing average MAPE for each timestep

over all monitored rivers. The full code for this section is available in Appendix D.

### 6.2.1 Data Preprocessing

The data preprocessing process, is almost identical to SVR. Firstly, the data is imported from the same spreadsheet into the Jupyter Notebook environment, to evaluate the model's performance more accurately. Analogically weather information data will be omitted.

The data is then divided into variables containing data about each river, by selecting a specific *riverstation_id*. However, the standardization process is skipped. The model seems to perform better with the original values.

The data is then spread into training and test subset as in SVR, with the same proportion 70% of the dataset is used for training and remaining 30% is used for testing.

Afterwards the data transformed into input vector and output scalar, again analogically as in SVR, with the difference being the window size of 64 previous timesteps.

### 6.2.2 Model Construction and Learning

The model is constructed using scikit-learn's *GaussianProcessRegressor* module, that implement Gaussian Process for regression based on Algorithm 2.1 proposed by Rasmussen and Williams. The prior mean is specified to training data mean by setting the *normalize_y* parameter to True. The prior covariance is specified by the kernel parameter and by default is set to $1.0 * RBF(1.0)$, where RBF is the Radial Bias Function kernel. The hyperparameters of the kernel are automatically optimized by maximizing the log-marginal-likelihood (LML), based on a specified optimizater. By default the optimization algorithm is set to *fmin_l_bfgs_b* and hence, is used as no further investigation into optimization algorithms was made. In order to avoid local optima of LML, the optimizer can be restarted multiple

times by specifying the *n_restart_optimizer* parameter. The optimizer is run multiple times with randomly chosen hyperparameters. The number of restarts was chosen arbitrary to equal 10. The noise level is specified by the *alpha* parameter, and is set globally to 0.1.

A separate model for each of 20 consecutive timesteps into the future, for each of the 10 monitored rivers is trained. Thus, 200 models were required before the final evaluation was done.

### 6.2.3 Model Evaluation

After the models are constructed, predictions are made for the test subset using the *predict* method provided by the *GaussianProcessRegressor* module. Again, all parameters were chosen arbitrary, so the model's performance could be still improved by performing hyperparameter tuning.

Afterwards, a mean absolute percentage error (MAPE) was calculated to evaluate each models performance. No rescaling was required as the step of standardization was omitted.

Predictions for 1,4,8 and 16 steps into the future were plotted using matplotlib library for visual evaluation. Figures 6.1, 6.2, 6.3, 6.4 show graphs for selected rivers (the same ones as for RNN and SVR).

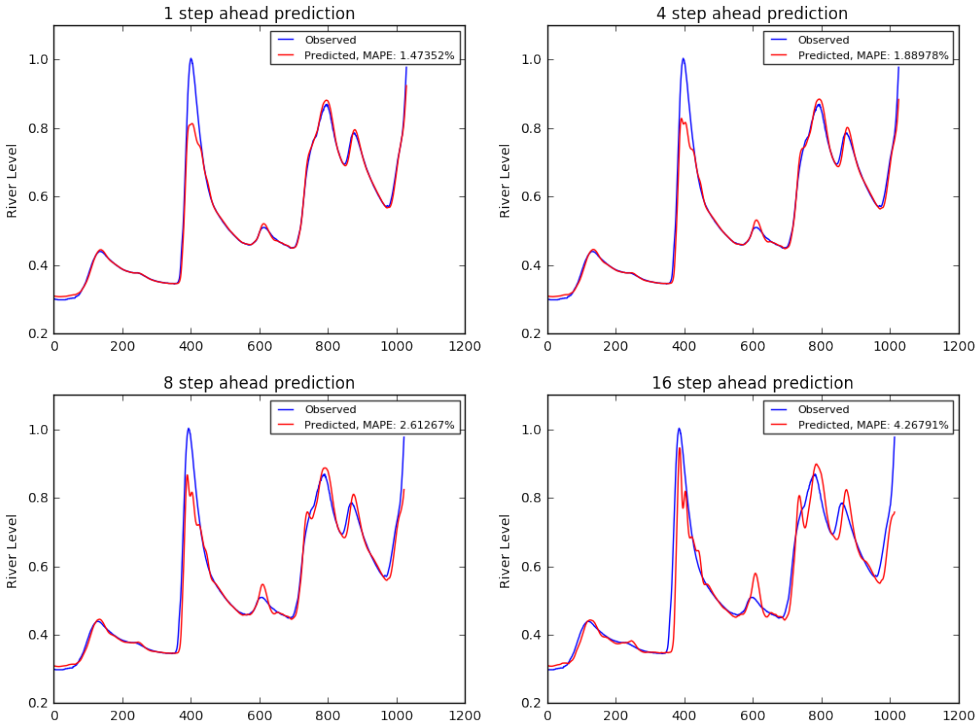


Figure 6.1: Plots showing predicted results for 1,4,8 and 16 steps ahead predictions using GPR algorithm.

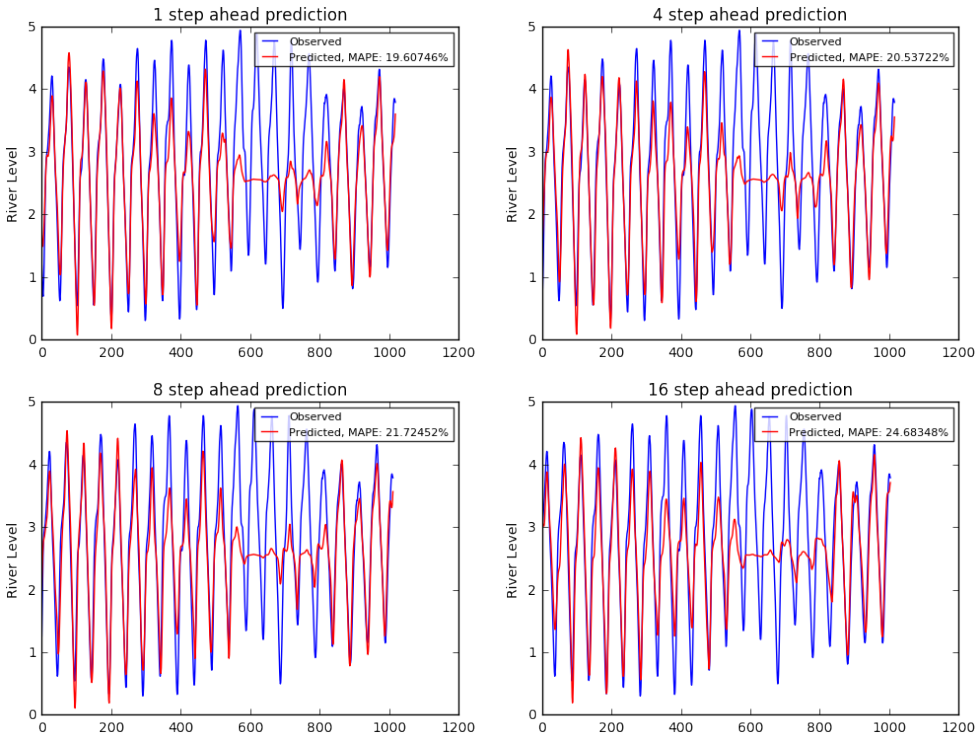


Figure 6.2: Plots showing predicted results for 1,4,8 and 16 steps ahead predictions using GPR algorithm.

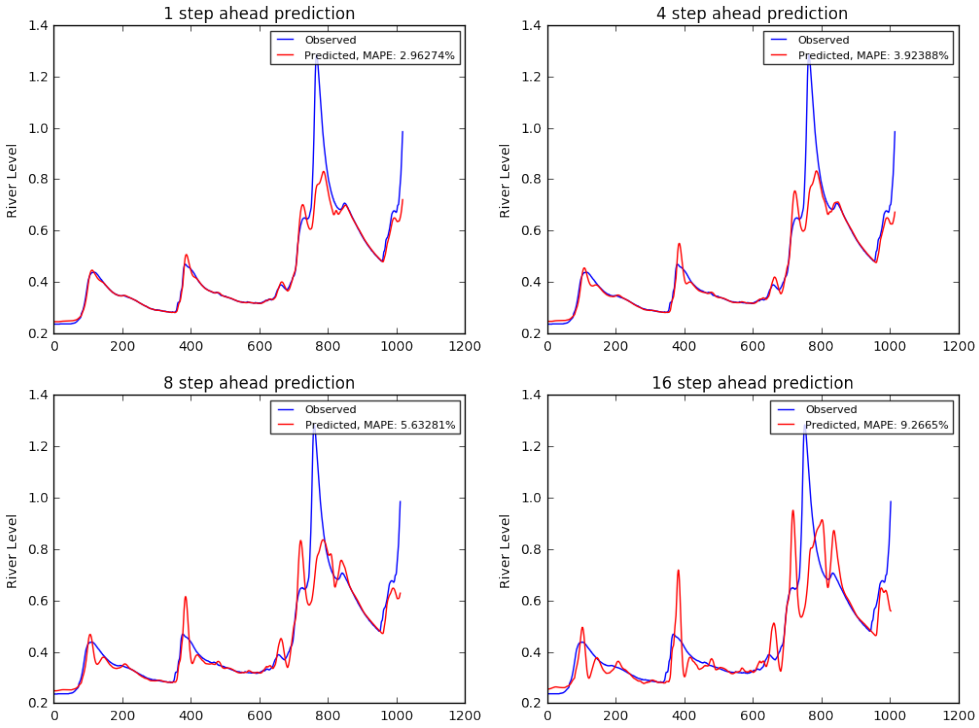


Figure 6.3: Plots showing predicted results for 1,4,8 and 16 steps ahead predictions using GPR algorithm.

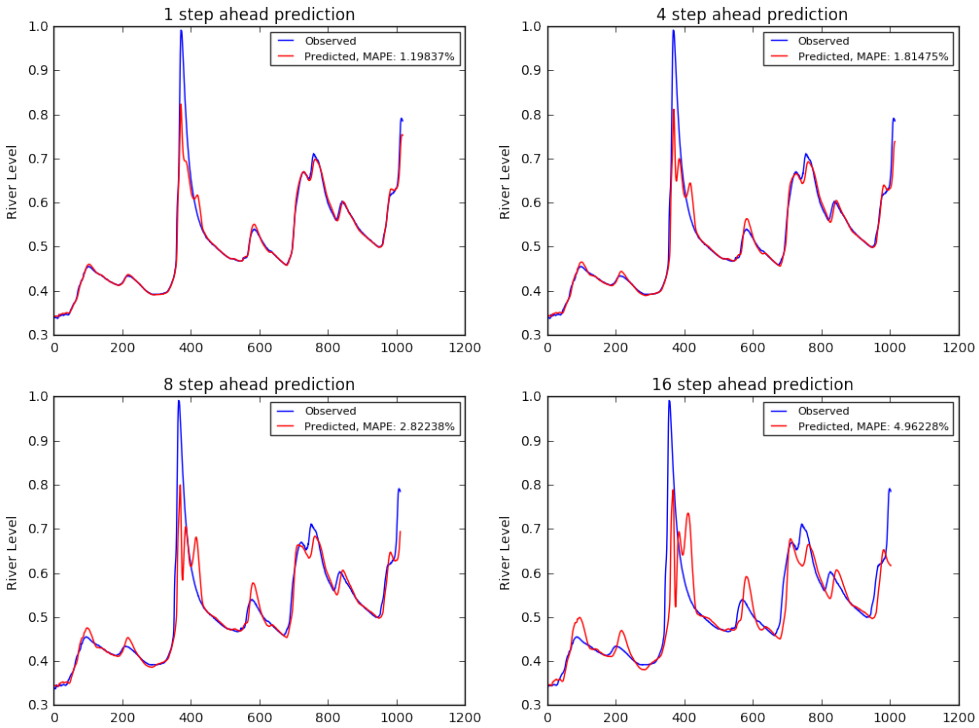


Figure 6.4: Plots showing predicted results for 1,4,8 and 16 steps ahead predictions using GPR algorithm.

The visual evaluation of predictions show that the algorithm performs relatively well. However the produced signal is less accurate than the ones produced by SVR and RNN. The model performs particularly bad in case of the second river as shown in figure 6.2.

Average MAPE for each step into the future was calculated over 10 monitored rivers, and presented in the table below for final evaluation of the algorithms performance.

| **Steps Ahead** | **Average MAPE over all rivers** |
|---|---|
| 1 | 3.743 % |
| 2 | 3.96 % |
| 3 | 4.199 % |
| 4 | 4.451 % |
| 5 | 4.709 % |
| 6 | 4.981 % |
| 7 | 5.267 % |
| 8 | 5.561 % |
| 9 | 5.861 % |
| 10 | 6.164 % |
| 11 | 6.47 % |
| 12 | 6.777 % |
| 13 | 7.082 % |
| 14 | 7.385 % |
| 15 | 7.684 % |
| 16 | 7.977 % |
| 17 | 8.265 % |
| 18 | 8.542 % |
| 19 | 8.807 % |
| 20 | 9.059 % |

Table 6.1: Average MAPE over all monitored rivers, for different number of steps ahead prediction using Gaussian Process Regression.

The results presented in table 6.2.3 were then plotted on a graph presented in figure 6.5.

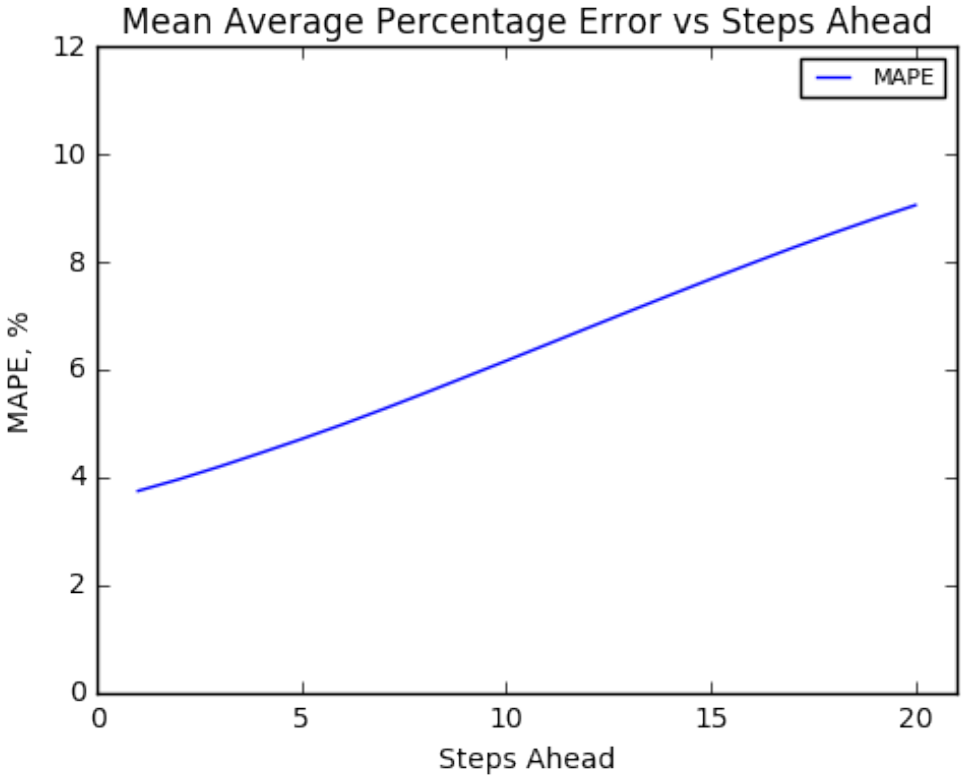


Figure 6.5: Plot of average MAPE over all rivers versus number of steps ahead, for predictions made using Gaussian Process Regression algorithm.

## 6.3 Conclusion

It is certain the Gaussian Process Regression is capable of modelling non-linear regression problems. The model performs relatively well in most cases, as supported by the results presented in table 6.2.3. The interesting property of the algorithm is the fact that it tunes most of its hyperparameters automatically, and applies the Occam's razor.

Moreover, as a non parametric model it seems to be able to model river levels that it has not encountered in the training data, more accurately than RNN and SVR.

Analogically, the error for predicting timesteps further into the future increases

almost linearly.

# Chapter 7

# Future Work

This chapter describes the additional work that could be performed in order to extend the project or achieve better performance for predictive models.

## 7.1 Weather Data

During model training only raw river level data was used for simplicity. However, the model could achieve better performance by including the weather information data collected during the data gathering process. Adding weather information might include feature selection process using techniques such as Random Forests in order to select the features that influence the river levels. Afterwards, these features would be added to each sample of the data. On the other hand, by adding too many features the model could suffer from the *Curse of Dimensionality* and performance might not improve.

## 7.2 Connected Rivers

Another way of adding additional features might include adding river level data from different river stations, that are placed on the same river, or connected rivers. Similarly to adding weather information, that might not necessarily improve the model performance.

## 7.3 Hyperparameter Tuning

Hyperparameters are free variables that have to be adjusted during model training to achieve the most optimal performance. The hyperparameter tuning process was omitted, while investigating each algorithms due to very good performance of the models. However by adjusting window sizes of previous values and all model parameters, an improvement in algorithms performance would be achieved. However, the improvement might not be significant. The hyperparameter tuning could be done using the *Pipeline* module provided by scikit-learn, which performs a grid search through each possible setting of hyperparameters provided in an array. At first sparse hypereparameters would be used, and the process would be repeated narrowing down the deviation of hyperparameters.

## 7.4 More Training Data

Machine Learning algorithms performance is very sensitive to the training data it is provided with. Predictive models are unable to predict scenarios that they have not encountered in the training data, thus by increasing the number of training samples a better performance will be achieved. The data collecting process has been continued, during investigation of the algorithms, however the dataset has not been updated. Hence, more data is available on the University server, however has not been used.

## 7.5 Additional Rivers

While comparing different algorithms, the performance of predictive models for ten different rivers was used. By creating predictive models on a higher number of rivers a more accurate comparison could be made. Only ten rivers were chosen due to time required for training the predictive models.

## 7.6 Copulas

Another algorithm that would be interesting to investigate, is the Copula based approach. This algorithm has been widely used for stock market forecasting, thus might achieve excellent performance, as problem faced in this project is analogous. This algorithm has been omitted in exchange of investigating Support Vector Regression and Gaussian Process Regression as implementations of these algorithms is provided by the scikit-learn library, in contrast of copulas.

# Chapter 8

# Conclusion

In this section, a final conclusion for the project will be performed. The performance for each investigated algorithm will be compared to each other for final evaluation and selection of the best performing model.

## 8.1 Comparison of investigated algorithms

During the project development three different machine learning algorithms, for solving regression problem were investigated. The algorithms are: Recurrent Neural Networks, Support Vector Regression and Gaussian Process Regression. Each of these algorithms seemed to evince successful application in similar problems, such as timeseries prediction in stock markets. The resulting average mean absolute percentage errors collected while evaluating each model were combined into one table 8.1, and then a combine plot was produced in figure 8.1. The initial models,were based solely on raw river level data, without the weather information data and the same number of samples. The best performing one seem to be the Support Vector Regression. However further investigation, by increasing the size of the dataset and performing hyperparameter tuning would need to be performed, in order to select the truly most accurate model. For instance, by greatly increasing the dataset size and number of layers in RNN, this algorithm could potentially outperform the SVR. Furthermore, additional techniques that could improve the performance of each model was described in more detail in the

*Further Work* chapter.

| **Steps Ahead** | **RNN** | **SVR** | **GPR** |
|---|---|---|---|
| 1 | 2.156 % | 1.294 % | 3.743 % |
| 2 | 2.437 % | 1.591 % | 3.96 % |
| 3 | 2.74 % | 1.916 % | 4.199 % |
| 4 | 3.121 % | 2.258 % | 4.451 % |
| 5 | 3.516 % | 2.593 % | 4.709 % |
| 6 | 3.886 % | 2.939 % | 4.981 % |
| 7 | 4.209 % | 3.276 % | 5.267 % |
| 8 | 4.487 % | 3.579 % | 5.561 % |
| 9 | 4.744 % | 3.88 % | 5.861 % |
| 10 | 5.008 % | 4.15 % | 6.164 % |
| 11 | 5.297 % | 4.406 % | 6.47 % |
| 12 | 5.603 % | 4.655 % | 6.777 % |
| 13 | 5.91 % | 4.897 % | 7.082 % |
| 14 | 6.211 % | 5.134 % | 7.385 % |
| 15 | 6.522 % | 5.396 % | 7.684 % |
| 16 | 6.83 % | 5.649 % | 7.977 % |
| 17 | 7.134 % | 5.905 % | 8.265 % |
| 18 | 7.454 % | 6.155 % | 8.542 % |
| 19 | 7.807 % | 6.412 % | 8.807 % |
| 20 | 8.209 % | 6.653 % | 9.059 % |

Table 8.1: Average MAPE over all monitored rivers, for different number of steps ahead prediction for each investigated algorithm.

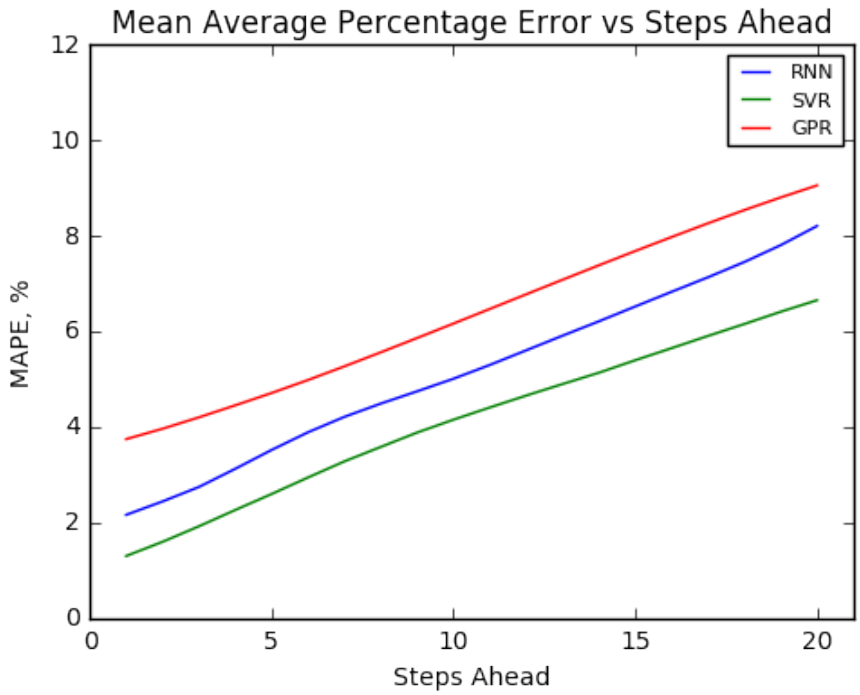


Figure 8.1: Plot of average MAPE over all rivers versus number of steps ahead, for all investigated algorithms.

## 8.2 Project Conclusion

Overall, the development of the project can be considered successful. Firstly, a data collecting application was implemented in python, based on the Flask micro-framework for future extension. Initial dataset for building a predictive model was collected. Afterwards, investigation of various supervised machine learning algorithms for timeseries prediction was performed. Furthermore, additional research for multistep timestep prediction was carried out, and three different strategies were discovered. The investigated algorithms along with the strategies used are:

1. Recurrent Neural Networks for timeseries prediction using the Multiple-input Multiple-output strategy was explored.

2. Support Vector Regression algorithm was investigated using the direct strategy.

3. Non parametrized model was constructed using the Gaussian Process Regression algorithm, again using the direct strategy.

Each algorithms performance was evaluated using mean absolute percentage error metrics. The best performing model seems to be the SVR. However, further investigation should be made, after performing hyperparameter tuning on each of the algorithms for more accurate model selection.

As the focus was put on investigating different algorithms, a web based approach for publishing the results was not implemented. However, work on a similar application has already been done by the EEE department at University of Strathclyde and the initially best performing algorithm will be integrate into the already existing application for the final presentation.

# Appendix A

# Data Collecting Application Code

__init__.py

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_script import Manager, Shell
from flask_migrate import Migrate, MigrateCommand
from flask_apscheduler import APScheduler
import logging

logging.basicConfig()

app = Flask(__name__)
app.config.from_object('config')

manager = Manager(app)

db = SQLAlchemy(app)

from app import views, models, jobs

migrate = Migrate(app, db)

def make_shell_context():
    return dict(app=app,
```

```
                    db=db,
                    RiverStation=models.RiverStation,
                    RiverLevel=models.RiverLevel,
                    WeatherStation=models.WeatherStation,
                    CurrentWeather=models.CurrentWeather)
manager.add_command("shell", Shell(make_context=make_shell_context))
manager.add_command('db', MigrateCommand)

scheduler = APScheduler()
scheduler.init_app(app)
scheduler.start()
```

grid_to_ne.py

```
import pyproj

def GR_to_NE( gr ):
  gr = gr.strip().replace( ' ', '' )
  if len(gr) == 0 or len(gr) % 2 == 1 or len(gr) > 12:
      return None, None

  gr = gr.upper()
  if gr[0] not in 'STNOH' or gr[1] == 'I' :
    return None, None

  e = n = 0
  c = gr[0]

  if c == 'T' :
    e = 500000
  elif c == 'N' :
    n = 500000
  elif c == 'O' :
    e = 500000
    n = 500000
  elif c == 'H':
    n = 10000000

  c = ord(gr[1]) - 66
  if c < 8: # J
```

```
        c += 1;

    e += (c % 5) * 100000;
    n += (4 - c/5) * 100000;

    c = gr[2:]
    try :
        s = c[:len(c)/2]
        while len(s) < 5 :
            s += '0'

        e += int( s )

        s = c[-len(c)/2:]
        while len(s) < 5 :
            s += '0';

        n += int( s )

    except Exception:
        return None,None;

    return e,n

def get_ne_lat_long( grid ):
    e,n = GR_to_NE( grid )
    bng = pyproj.Proj(init='epsg:27700')
    wgs84 = pyproj.Proj(init='epsg:4326')
    lon,lat = pyproj.transform(bng,wgs84, e, n)
    return e,n,lon,lat
```

jobs.py

```
import requests
import csv
from app import app, db
from app.models import RiverStation, RiverLevel, WeatherStation,
    CurrentWeather
import os
from grid_to_ne import get_ne_lat_long
```

```
import datetime

def test():
    print 'test'

def pull_river_level_data(urls):
    if os.environ.get("WERKZEUG_RUN_MAIN") == "true":
        for url in urls:
            r = requests.get(url)
            if r.status_code != 404:
                l = csv_to_list(r)
                data_index = get_measurment_index(l) + 1
                station_id = get_or_create_station(l)
                for m in l[data_index:]:
                    if m:
                        add_measurment(m, station_id)
                        db.session.commit()
                print "Fetched River Level for :" + str(station_id)
            else:
                print "404 Something is wrong with the url: " + url


def add_measurment(m, station_id):
    sampled_at = datetime.datetime.strptime(m[0], '%d/%m/%Y %H:%M:%S
    ')
    riverlevel = RiverLevel.query.filter_by(
        riverstation_id=station_id,
        sampled_at=sampled_at).first()
    if len(m) == 2 and  riverlevel== None:
        meas = RiverLevel()
        meas.riverstation_id = station_id
        meas.sampled_at = datetime.datetime.strptime(m[0], '%d/%m/%Y
    %H:%M:%S')
        meas.level = m[1]
        db.session.add(meas)


def csv_to_list(r):
    text = r.iter_lines()
```

```
    reader = csv.reader(text, delimiter=",")
    l = []
    for m in reader:
        l.append(m)
    return l

def get_measurment_index(result):
    for index, row in enumerate(result):
        if row[0] == 'Date/Time':
            return index

def get_or_create_station(result):
    station_name = get_station_name(result)
    station = RiverStation.query.filter_by(riverstation_uid=
   station_name).first()
    if station == None:
        station = RiverStation()
        station.riverstation_uid = station_name
        e,n,lon,lat = get_coords(station_name)
        station.osgr_easting = e;
        station.osgr_northing = n;
        station.longitide = lon;
        station.latitude = lat;
        db.session.add(station)
        db.session.commit()
    return station.riverstation_id

def get_location_code(result):
    for row in result:
      if row[0] == 'Location Code':
          return row[1]

def get_office(result):
    for row in result:
        if row[0] == 'Office':
            return row[1]

def get_station_name(result):
    for row in result:
```

```
        if row[0] == 'Station_Name':
            return row[1]

def get_coords(station_name):
    url = 'http://apps.sepa.org.uk/database/riverlevels/
   SEPA_River_Levels_Web.csv'
    r = requests.get(url)
    l = csv_to_list(r)
    name_i = station_name_index(l[0])
    ngr_i = ngr_index(l[0])
    for row in l:
        if row and station_name == row[name_i]:
            return get_ne_lat_long(row[ngr_i])

def station_name_index(row):
    for index, field in enumerate(row):
        if(field=="STATION_NAME"):
            return index

def ngr_index(row):
    for index, field in enumerate(row):
        if(field=="NATIONAL_GRID_REFERENCE"):
            return index


####### WEATHER JOB ########

def get_weather(cities):
    api_key = "32f94d7321018228f4a37b517df35858"
    for city in cities:
        url = "http://api.openweathermap.org/data/2.5/weather?q="+
   city+"&appid=" + api_key
        r = requests.get(url)
        weather = r.json()
        weather_station_id = get_or_create_weather_station(weather)
        current_weather = CurrentWeather()
        current_weather.weatherstation_id = weather_station_id
        current_weather.precipation = weather['rain']['3h'] if 'rain
   ' in weather else 0
```

```
        current_weather.temperature = weather['main']['temp']
        current_weather.wind_speed = weather['wind']['speed']
        current_weather.wind_direction = weather['wind']['deg']
        current_weather.description = weather['weather'][0]['
    description']
        current_weather.sampled_at = datetime.datetime.fromtimestamp
    (
                                                int(weather['dt'])
                                            ).strftime('%d/%m/%Y %H:%M:%
    S')
        db.session.add(current_weather)
        db.session.commit()
        print 'Weather fetched ' + str(current_weather.
    weatherstation_id)

def get_or_create_weather_station(weather):
    weather_station = WeatherStation.query.filter_by(
    weatherstation_uid=weather['name']).first()
    if not weather_station:
        weather_station = WeatherStation()
        weather_station.weatherstation_uid = weather['name']
        weather_station.longitude = weather['coord']['lon']
        weather_station.latitude = weather['coord']['lat']
        db.session.add(weather_station)
        db.session.commit()
    return weather_station.weatherstation_id
```

models.py

```
from app import db
from sqlalchemy.sql import func

class RiverStation(db.Model):
  __tablename__ = 'RiverStation'
  riverstation_id = db.Column(db.Integer, primary_key=True, nullable
   =False)
  riverstation_uid = db.Column(db.String(64), unique=True, index=
   True, nullable=False)
  osgr_easting = db.Column(db.Integer, nullable=False)
  osgr_northing = db.Column(db.Integer, nullable=False)
```

```
    latitude = db.Column(db.Float, nullable=False)
    longitide = db.Column(db.Float, nullable=False)
    river_level = db.relationship('RiverLevel', backref='RiverStation'
    )

    def __repr__(self):
        return '<Station id:%d, name:%r' % (self.riverstation_id, self.
        riverstation_uid)

class RiverLevel(db.Model):
    __tablename__= 'RiverLevel'
    riverlevel_id = db.Column(db.Integer, primary_key=True, nullable=
    False)
    riverstation_id = db.Column(db.Integer, db.ForeignKey('
    RiverStation.riverstation_id'), nullable=False)
    sampled_at = db.Column(db.DateTime, nullable=False)
    archived_at = db.Column(db.DateTime, default=func.now(), nullable=
    False)
    level = db.Column(db.Float, nullable=False)

    def __repr__(self):
        return '<Measurment id:%d, date:%c, value:%0.3f' % (self.
        _riverlevel_id, self.sampled_at, self.level)

class WeatherStation(db.Model):
    __tablename__= 'WeatherStation'
    weatherstation_id = db.Column(db.Integer, primary_key=True,
    nullable=False)
    weatherstation_uid = db.Column(db.String(64), unique=True,
    nullable=False)
    latitude = db.Column(db.Float, nullable=False)
    longitude = db.Column(db.Float, nullable=False)

class CurrentWeather(db.Model):
    __tablename__= 'CurrentWeather'
    currentweather_id = db.Column(db.Integer, primary_key=True,
    nullable=False)
    weatherstation_id = db.Column(db.Integer, db.ForeignKey('
    WeatherStation.weatherstation_id'), nullable=False)
```

```
    archived_at = db.Column(db.DateTime, default=func.now(),
    nullable=False)
    sampled_at = db.Column(db.DateTime, nullable=False)
    precipation = db.Column(db.Float, nullable=False)
    temperature = db.Column(db.Float, nullable=False)
    wind_speed = db.Column(db.Float, nullable=False)
    wind_direction = db.Column(db.Float, nullable=False)
    description = db.Column(db.String(256), nullable=False)
```

config.py

```
import os

SQLALCHEMY_DATABASE_URI = 'postgresql:///bartosz'
SQLALCHEMY_COMMIT_ON_TEARDOWN = True
SQLALCHEMY_TRACK_MODIFICATIONS = True

DEBUG = True

SEPA_RIVER_LEVEL_URLS = [
  'http://apps.sepa.org.uk/database/riverlevels/133099-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/133138-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/133074-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/133093-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/133118-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/133061-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/133080-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/133119-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/133113-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/133114-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/504722-SG.csv',
  'http://apps.sepa.org.uk/database/riverlevels/133182-SG.csv'
]

CITIES = [ 'Glasgow,uk', 'Moffat,uk']

JOBS = [
  {
    'id': 'river_level',
    'func': 'app.jobs:pull_river_level_data',
```

```
    'args': [SEPA_RIVER_LEVEL_URLS],
    'trigger': 'interval',
    'hours': 12
  },
  {
    'id': 'current_weahter',
    'func': 'app.jobs:get_weather',
    'args': [CITIES],
    'trigger': 'interval',
    'minutes': 15
  }
]

WEATHER_APIKEY = "32f94d7321018228f4a37b517df35858"
```

run.py

```
#!flask/bin/python
from app import manager
manager.run()
```

requirements.txt

```
alembic==0.8.8
APScheduler==3.3.0
click==6.6
Flask==0.11.1
Flask-APScheduler==1.5.0
Flask-Migrate==2.0.0
Flask-Script==2.0.5
Flask-SQLAlchemy==2.1
funcsigs==1.0.2
futures==3.0.5
itsdangerous==0.24
Jinja2==2.8
Mako==1.0.4
MarkupSafe==0.23
numpy==1.11.2
pandas==0.19.0
psycopg2==2.6.2
```

```
pyproj==1.9.5.1
python-dateutil==2.5.3
python-editor==1.0.1
pytz==2016.7
requests==2.11.1
six==1.10.0
SQLAlchemy==1.1.2
tzlocal==1.3
Werkzeug==0.11.11
```

# Appendix B

# Recurrent Neural Networks Code

```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, LSTM, GRU, SimpleRNN, Dropout,
    Activation, RepeatVector, TimeDistributed
from keras.regularizers import l1
from keras.callbacks import EarlyStopping
from sklearn.metrics import mean_squared_error
from sklearn import preprocessing
import math

# 2 following lines are required by AWS EC2 instance, for some
    reason
plt.switch_backend('TkAgg')
%matplotlib inline

all_levels = pd.read_csv('./csv/rivers.csv')

np.random.seed(7)

# Get current size
```

```
fig_size = plt.rcParams["figure.figsize"]

# Set figure width to 12 and height to 9
fig_size[0] = 12
fig_size[1] = 9
plt.rcParams["figure.figsize"] = fig_size

def train_test_split(dataset, train_frac):
    train_size = int(len(dataset)*train_frac)
    return dataset[:train_size, :], dataset[train_size: ,:]

def create_datasets(dataset, look_back=1, look_ahead=1):
    data_x, data_y = [], []
    for i in range(len(dataset)-look_back-look_ahead+1):
        window = dataset[i:(i+look_back), 0]
        data_x.append(window)
        data_y.append(dataset[i + look_back:i + look_back +
   look_ahead , 0])
    return np.array(data_x), np.array(data_y)

def build_seq2seq_model(look_ahead = 1):
    m = Sequential()
    # encoder
    m.add(GRU(16, input_dim = 1))
    # repeat for the number of steps out
    m.add(RepeatVector(look_ahead))
    # decoder
    m.add(GRU(8, return_sequences=True))
    m.add(GRU(8, return_sequences=True))
    # split the output into timesteps
    m.add(TimeDistributed(Dense(1)))
    m.compile(loss='mse', optimizer='rmsprop')
    #m.summary()
    return m

def reverse_scale(data, mean, std):
    for x in np.nditer(data, op_flags=['readwrite']):
        x[...] = x*std + mean
    return data
```

```
def calculate_error(train_y, test_y, pred_train, pred_test):
    test_score = math.sqrt(mean_squared_error(test_y, pred_test))
    train_score = math.sqrt(mean_squared_error(train_y, pred_train))
    return train_score, test_score

def mean_absolute_percentage(y, y_pred):
    return np.mean(np.abs((y - y_pred) / y)) * 100

def root_mse(pred_test, test_y):
    t = []
    for i in range(20):
        score = math.sqrt(mean_squared_error(pred_test[:,i,:],
   test_y[:,i,:]))
        t.append(score)
        print i+1, "  ->  ", score

    return score

def plot_4_errors(pred_test, test_y, er1, er2, er3, er4):
    plt.subplot(221)
    plt.plot(test_y[:,0,:], label="Observed")
    plt.plot(pred_test[:,0,:], color="red", label="Predicted, MAPE: "
   "+ str(round(er1, 5))+"%")
    plt.title("1 step ahead prediction")
    plt.ylabel("River Level")
    plt.legend(loc=1, fontsize = 8, framealpha=0.8)

    plt.subplot(222)
    plt.plot(pred_test[:,3,:], color="red", label="Predicted, MAPE: "
   "+ str(round(er2, 5))+"%")
    plt.plot(test_y[:,3,:], label="Observed")
    plt.title("4 step ahead prediction")
    plt.legend(loc=1, fontsize = 8, framealpha=0.8)

    plt.subplot(223)
    plt.plot(pred_test[:,7,:], color="red", label="Predicted, MAPE: "
   "+ str(round(er3, 5))+"%")
    plt.plot(test_y[:,7,:], label="Observed")
```

```
    plt.title("8 step ahead prediction")
    plt.legend(loc=1, fontsize = 8, framealpha=0.8)

    plt.subplot(224)
    plt.plot(pred_test[:,15,:], color="red", label="Predicted, MAPE:
    "+ str(round(er4, 5))+"%")
    plt.plot(test_y[:,15,:], label="Observed")
    plt.title("16 step ahead prediction")
    plt.legend(loc=1, fontsize = 8, framealpha=0.8)

    plt.show()


def do_everything(look_back, look_ahead, riverstation_id, split,
    epochs, batch_size):
    river = all_levels[all_levels['riverstation_id'] ==
    riverstation_id]['level'].values
    river_mean, river_std = river.mean(), river.std()
    river = preprocessing.scale(river).reshape(len(river), 1)

    #split data into train and test subsets
    train, test = train_test_split(river, split)
    train_x, train_y = create_datasets(train, look_back, look_ahead)
    test_x, test_y = create_datasets(test, look_back, look_ahead)

    #reshape the data to match Keras LSTM gate input [samples, time
    steps, features]
    train_x = np.reshape(train_x, (train_x.shape[0], train_x.shape
    [1], 1))
    train_y = np.reshape(train_y, (train_y.shape[0], train_y.shape
    [1], 1))

    test_x = np.reshape(test_x, (test_x.shape[0], test_x.shape[1],
    1))
    test_y = np.reshape(test_y, (test_y.shape[0], test_y.shape[1],
    1))

    model = build_seq2seq_model(look_ahead)
```

```
    model.fit(train_x,
              train_y,
              nb_epoch=epochs,
              batch_size=batch_size,
              verbose=2,
              validation_split = 0.1,
              callbacks = [
                EarlyStopping(monitor='val_loss', min_delta=0.001,
    patience=3, verbose=2, mode='auto')
              ]
             )

    pred_train = model.predict(train_x)
    pred_test = model.predict(test_x)

    pred_train = reverse_scale(pred_train, river_mean, river_std)
    pred_test = reverse_scale(pred_test, river_mean, river_std)
    test_y = reverse_scale(test_y, river_mean, river_std)
    train_y = reverse_scale(train_y, river_mean, river_std)

    errors = []
    for i in range(20):
        errors.append(mean_absolute_percentage(test_y[:,i,:],
    pred_test[:,i,:]))

    plot_4_errors(pred_test, test_y, errors[0], errors[3], errors
    [7], errors[15])
    #root_mse(pred_test, test_y)

    errors = []
    for i in range(20):
        errors.append(mean_absolute_percentage(test_y[:,i,:],
    pred_test[:,i,:]))

    return errors

scores = []
for i in range(1,11):
    err = do_everything(look_back=64,
```

```
                            look_ahead=20,
                            riverstation_id=i,
                            split=0.8,
                            epochs=50,
                            batch_size=10)
    scores.append(err)

averages = []
for i in range(20):
    sum = 0
    for r in scores:
        sum+=r[i]
    averages.append(sum/10)
    print i+1, " & ",round(sum/10, 3), "\% \\\\"
    print '\hline'

plt.subplot(221)
plt.plot(range(1,21),averages, label="MAPE")
plt.title("Mean Average Percentage Error vs Steps Ahead")
plt.ylabel("MAPE, %")
plt.xlabel("Steps Ahead")
plt.legend(loc=1, fontsize=8)
plt.ylim([0,10])
plt.xlim([0, 21])
plt.show()
```

# Appendix C

# Support Vector Regression Code

```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn.svm import SVR
from sklearn import preprocessing
from sklearn.metrics import mean_squared_error
import math

all_levels = pd.read_csv('./csv/rivers.csv')

# Get current size
fig_size = plt.rcParams["figure.figsize"]

# Set figure width to 12 and height to 9
fig_size[0] = 12
fig_size[1] = 9
plt.rcParams["figure.figsize"] = fig_size

def train_test_split(dataset, train_frac):
    train_size = int(len(dataset)*train_frac)
    return dataset[:train_size, :], dataset[train_size:,:]
```

```
def create_datasets(dataset, look_back=1, look_ahead=1):
    data_x, data_y = [], []
    for i in range(len(dataset)-look_back-look_ahead+1):
        window = dataset[i:(i+look_back), 0]
        data_x.append(window)
        data_y.append(dataset[i + look_back + look_ahead -1, 0])
    return np.array(data_x), np.array(data_y)

def reverse_scale(data, mean, std):
    for x in np.nditer(data, op_flags=['readwrite']):
        x[...] = x*std + mean
    return data

def mean_absolute_percentage(y, y_pred):
    return np.mean(np.abs((y - y_pred) / y)) * 100

split = 0.7
look_back = 32
results = []
for i in range(1, 11):
    river_results = []
    for j in range(1, 21):
        look_ahead = j
        riverstation_id = i

        river = all_levels[all_levels['riverstation_id'] ==
   riverstation_id]['level'].values
        river = river.reshape(len(river), 1)

        #standardize data
        river_mean, river_std = river.mean(), river.std()
        river = preprocessing.scale(river).reshape(len(river), 1)

        train, test = train_test_split(river, split)
        train_x, train_y = create_datasets(train, look_back,
   look_ahead)
        test_x, test_y = create_datasets(test, look_back, look_ahead
   )
```

```
        clf = SVR(C=1.0, epsilon=0.01)
        clf.fit(train_x, train_y)
        pred_train = clf.predict(train_x)
        pred_test = clf.predict(test_x)

        #reverse scale
        pred_train = reverse_scale(pred_train, river_mean, river_std
    )
        pred_test = reverse_scale(pred_test, river_mean, river_std)
        test_y = reverse_scale(test_y, river_mean, river_std)
        train_y = reverse_scale(train_y, river_mean, river_std)

        score = mean_absolute_percentage(test_y, pred_test)
        river_results.append(score)

        if(j == 1):
            plt.subplot(221)
            plt.plot(test_y, label="Observed")
            plt.plot(pred_test, color="red", label="Predicted, MAPE:
    "+ str(round(score, 5))+"%")
            plt.title("1 step ahead prediction")
            plt.ylabel("River Level")
            plt.legend(loc=1, fontsize = 8, framealpha=0.8)


        if(j == 4):
            plt.subplot(222)
            plt.plot(test_y, label="Observed")
            plt.plot(pred_test, color="red", label="Predicted, MAPE:
    "+ str(round(score, 5))+"%")
            plt.title("4 step ahead prediction")
            plt.ylabel("River Level")
            plt.legend(loc=1, fontsize = 8, framealpha=0.8)


        if(j == 8):
            plt.subplot(223)
            plt.plot(test_y, label="Observed")
```

```
            plt.plot(pred_test, color="red", label="Predicted, MAPE:
    "+ str(round(score, 5))+"%")
            plt.title("8 step ahead prediction")
            plt.ylabel("River Level")
            plt.legend(loc=1, fontsize = 8, framealpha=0.8)


        if(j == 16):
            plt.subplot(224)
            plt.plot(test_y, label="Observed")
            plt.plot(pred_test, color="red", label="Predicted, MAPE:
    "+ str(round(score, 5))+"%")
            plt.title("16 step ahead prediction")
            plt.legend(loc=1, fontsize = 8, framealpha=0.8)
            plt.ylabel("River Level")
            plt.show()

    results.append(river_results)

averages = []
for i in range(20):
    sum = 0
    for r in results:
        sum+=r[i]
    averages.append(sum/10)


plt.subplot(221)
plt.plot(range(1,21),averages, label="MAPE")
plt.title("Mean Average Percentage Error vs Steps Ahead")
plt.ylabel("MAPE, %")
plt.xlabel("Steps Ahead")
plt.legend(loc=1, fontsize=8)
plt.ylim([0,8])
plt.xlim([0, 21])
plt.show()
```

# Appendix D

# Gaussian Process Code

```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn.svm import SVR
from sklearn import preprocessing
from sklearn.metrics import mean_squared_error
import math
from sklearn.gaussian_process import GaussianProcessRegressor

all_levels = pd.read_csv('./csv/rivers.csv')

# Get current size
fig_size = plt.rcParams["figure.figsize"]

# Set figure width to 12 and height to 9
fig_size[0] = 12
fig_size[1] = 9
plt.rcParams["figure.figsize"] = fig_size

def train_test_split(dataset, train_frac):
    train_size = int(len(dataset)*train_frac)
    return dataset[:train_size, :], dataset[train_size:,:]

def create_datasets(dataset, look_back=1, look_ahead=1):
    data_x, data_y = [], []
```

```
    for i in range(len(dataset)-look_back-look_ahead+1):
        window = dataset[i:(i+look_back), 0]
        data_x.append(window)
        data_y.append(dataset[i + look_back + look_ahead -1, 0])
    return np.array(data_x), np.array(data_y)

def mean_absolute_percentage(y, y_pred):
    return np.mean(np.abs((y - y_pred) / y)) * 100


results = []

for i in range(1, 11):
    river_results = []
    for j in range(1, 21):
        look_ahead = j
        riverstation_id = i

        river = all_levels[all_levels['riverstation_id'] ==
    riverstation_id]['level'].values
        river = river.reshape(len(river), 1)

        train, test = train_test_split(river, split)
        train_x, train_y = create_datasets(train, look_back,
    look_ahead)
        test_x, test_y = create_datasets(test, look_back, look_ahead
    )
        gpr = GaussianProcessRegressor(n_restarts_optimizer = 10,
    alpha = 0.1, normalize_y=True)
        gpr.fit(train_x, train_y)
        pred_train = gpr.predict(train_x)
        pred_test = gpr.predict(test_x)

        score = mean_absolute_percentage(test_y, pred_test)
        river_results.append(score)

        if(j == 1):
            plt.subplot(221)
            plt.plot(test_y, label="Observed")
```

```
            plt.plot(pred_test, color="red", label="Predicted, MAPE:
 "+ str(round(score, 5))+"%")
            plt.title("1 step ahead prediction")
            plt.ylabel("River Level")
            plt.legend(loc=1, fontsize = 8, framealpha=0.8)


        if(j == 4):
            plt.subplot(222)
            plt.plot(test_y, label="Observed")
            plt.plot(pred_test, color="red", label="Predicted, MAPE:
 "+ str(round(score, 5))+"%")
            plt.title("4 step ahead prediction")
            plt.ylabel("River Level")
            plt.legend(loc=1, fontsize = 8, framealpha=0.8)


        if(j == 8):
            plt.subplot(223)
            plt.plot(test_y, label="Observed")
            plt.plot(pred_test, color="red", label="Predicted, MAPE:
 "+ str(round(score, 5))+"%")
            plt.title("8 step ahead prediction")
            plt.ylabel("River Level")
            plt.legend(loc=1, fontsize = 8, framealpha=0.8)


        if(j == 16):
            plt.subplot(224)
            plt.plot(test_y, label="Observed")
            plt.plot(pred_test, color="red", label="Predicted, MAPE:
 "+ str(round(score, 5))+"%")
            plt.title("16 step ahead prediction")
            plt.legend(loc=1, fontsize = 8, framealpha=0.8)
            plt.ylabel("River Level")
            plt.show()

    results.append(river_results)
```

```
averages = []
for i in range(20):
    sum = 0
    for r in results:
        sum+=r[i]
    averages.append(sum/10)

plt.subplot(221)
plt.plot(range(1,21),averages, label="MAPE")
plt.title("Mean Average Percentage Error vs Steps Ahead")
plt.ylabel("MAPE, %")
plt.xlabel("Steps Ahead")
plt.legend(loc=1, fontsize=8)
plt.ylim([0,12])
plt.xlim([0, 21])
plt.show()
```

# Bibliography


[1] Debasish Basak et al. "Support vector regression". In: *Neural Information Processing Letters and Reviews*. 2007, pp. 203–224.

[2] G. Bontempi. "Long term time series prediction with multi-input multi-output local learning." In: (2008).

[3] Haibin Cheng et al. "Multistep-Ahead Time Series Prediction". In: *Advances in Knowledge Discovery and Data Mining: 10th Pacific-Asia Conference, PAKDD 2006, Singapore, April 9-12, 2006. Proceedings*. Ed. by Wee-Keong Ng et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 765–774. ISBN: 978-3-540-33207-7. DOI: 10.1007/11731139_89. URL: http://dx.doi.org/10.1007/11731139_89.

[4] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). URL: http://arxiv.org/abs/1406.1078.

[5] Junyoung Chung et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555* (2014).

[6] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Aistats*. Vol. 9. 2010, pp. 249–256.

[7] Alex Graves. *Supervised sequence labelling with recurrent neural networks*. Studies in Computational intelligence. Heidelberg, New York: Springer, 2012. ISBN: 978-3-642-24796-5. URL: http://opac.inria.fr/record=b1133792.

[8] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. 1st. O'Reilly Media, Inc., 2014. ISBN: 1449372627, 9781449372620.

[9] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[10] University of Montreal Lisa lab. "Deep Learning Tutorial Release 0.1". In: (2015).

[11] Longfei Lu. "Optimal $\gamma$ and C for $\epsilon$-Support Vector Regression with RBF Kernels". In: *CoRR* abs/1506.03942 (2015). URL: http://arxiv.org/abs/1506.03942.

[12] Sebastian Raschka. *About Feature Scaling and Normalization*. 2014. URL: http://sebastianraschka.com/Articles/2014_about_feature_scaling.html.

[13] Sebastian Raschka. *Python Machine Learning*. Birmingham, UK: Packt Publishing, 2015. ISBN: 1783555130.

[14] CE. Rasmussen and CKI. Williams. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. Cambridge, MA, USA: MIT Press, Jan. 2006, p. 248.

[15] Alex J. Smola and Bernhard Schölkopf. "A Tutorial on Support Vector Regression". In: *Statistics and Computing* 14.3 (Aug. 2004), pp. 199–222. ISSN: 0960-3174. DOI: 10.1023/B:STCO.0000035301.49549.88. URL: https://doi.org/10.1023/B:STCO.0000035301.49549.88.

[16] D. H. Wolpert and W. G. Macready. "No Free Lunch Theorems for Optimization". In: *Trans. Evol. Comp* 1.1 (Apr. 1997), pp. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893. URL: http://dx.doi.org/10.1109/4235.585893.

[17] Zhongyi Hu Yukun Bao Tao Xiong. "Multi-Step-Ahead Time Series Prediction using Multiple-Output Support Vector Regression." In: (2013).