PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
GRADUATE PROGRAM IN COMPUTER SCIENCE

AN ARCHITECTURE FOR IMPROVED DEMAND FORECASTING

STEPHAN CHANG

Thesis presented as partial requirement for obtaining the degree of Master in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Felipe Rech Meneguzzi

Porto Alegre
2017

REPLACE THIS PAGE
WITH THE LIBRARY
CATALOG INFORMATION

REPLACE THIS PAGE
WITH THE
PRESENTATION TERM

AN ARCHITECTURE FOR IMPROVED DEMAND FORECASTING

RESUMO

Há um problema no domínio da logística que está relacionado à distribuição precisa e pontual de suprimentos em uma rede de localizações. Este problem surge da incerteza da demanda futura e causa o desperdício de recursos na produção de suprimentos que acabam inutilizados, enquanto locais de demanda sofrem com baixos estoques. Uma das formas de reduzir o impacto desse problema é utilizar modelos de predição para prever a demanda que está por vir, de forma a utilizar essa informação para coordenar o processo de produção e distribuição de suprimentos. Fazer previsões acuradas, no entanto, é uma tarefa desafiadora pois para treinar modelos de predição depende-se não apenas de uma grande quantidade de dados, mas também da qualidade dos exemplos de treino, especialmente quando consideramos que há muitos locais de demanda e uma grande variedade de produtos envolvidos nas demandas. Nesta dissertação, construímos uma arquitetura que utiliza conceitos de agentes e modelos de aprendizado de máquina para melhorar a qualidade das previsões de demanda feitas neste tipo de cenário. Para a experimentação, analisamos duas bases de dados: uma que possui dados limitados, e outra que possui dados com melhor qualidade, para que possamos estudar a performance dos algoritmos em cada caso. Ao final, comparamos os resultados da experimentação dos modelos avulsos com os resultados obtidos ao utilizar a nossa arquitetura.

Palavras-Chave: previsão de demanda, modelos de predição, sistemas multiagente, inteligência artificial, aprendizado de máquina.

AN ARCHITECTURE FOR IMPROVED DEMAND FORECASTING

ABSTRACT

In the logistics domain, there is a problem in supply and demand which concerns the accurate and punctual distribution of supplies in a network of locations. This problem stems from the uncertainty of future demand, and often causes the supplier to waste resources on unneeded supplies and demanding parties to suffer from under supply. Prediction models to forecast demand can minimise the impact of this situation by using predictions to coordinate both production and distribution strategies. Making accurate forecasts, however, is difficult because it depends on the quality of the data available to train predictive models, specially when considering multiple demanding locations and a variety of products to choose from. In this dissertation, we develop an architecture that uses agents and machine learning models to improve demand forecasting in such domains. We evaluate two datasets, one with limited data and the other with richer data, and experiment with Machine Learning algorithms to solve the demand forecasting problem on these datasets. We then compare the results obtained from these experiments and those obtained with our architecture.

Keywords: demand forecasting, prediction models, multi-agent systems, artificial intelligence, machine learning.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ACRONYMS

AI Artificial Intelligence

ANN Artificial Neural Network

BDI Beliefs, Desires and Intentions

BMA Best Method Agent

CoV Coefficient of Variation

LR Linear Regression

MAE Mean Absolute Error

MAPE Mean Absolute Percentage Error

MASE Mean Absolute Scaled Error

MAS Multi-Agent System

ML Machine Learning

MLP Multi-Layer Perceptron

NN Neural Network

RBF Radial Basis Function

ReLU Rectified Linear Units

RMSE Root Mean Squared Error

SSE Sum of Squared Errors

SVM Support Vector Machine

SVR Support Vector Regression

CONTENTS

1. INTRODUCTION

Logistics is the discipline of coordinating events that involve agents, goods and facilities. In supply and demand, we employ logistics to manage the production of goods and their distribution to locations in need of such goods. There is a problem within this domain, however, that concerns the accurate and punctual distribution of supplies among these locations. Distribution accuracy refers to delivering the correct items in their correct quantities to the correct places, while punctuality refers to delivering items within a pre-defined time frame, which is usually negotiated between supplier and demander. Failure to meet the demander's expectations might cause the waste of resources on the production and transportation of unconsumed supplies, which then causes the satisfaction rate of demanding parties to decrease as a result. When a supply cycle ends with some locations holding more supplies than they need, and others lacking enough supplies, we say there is an **imbalance in supply**. The supply imbalance problem shares similarities with Vehicle Routing Problem (VRP) [12, 4], an optimisation problem in which we have a set of supplies that must be distributed among a network of cities. The difference between these problems is the optimisation goal that we seek. Whereas in VRP we wish to optimise the route by limiting the number of cities each truck visits and the distribution of supplies among available trucks, in the supply imbalance problem we want to optimise production costs and the time to delivery.

To optimise for production costs and delivery time, we can use forecasting methods to predict future demand and set production to meet predictions. This stems from two previous attempts to manage supply inventories, one in which suppliers overstocked items to have them ready for eventual demand, and one in which suppliers await for actual demand before stocking items. The disadvantage of the former is the high risk of having stale inventories and the associated waste of resources, while the latter, though successful in resource saving, makes up for a slow delivery process, therefore risking punctuality. However, if we are able to predict demand, we may still save resources by stocking only what has a good chance of being demanded, and speed up delivery time by having supplies available at the time of demand.

Demand forecasting helps deciding whether to produce items before they are purchased, to reduce delivery times, or to withhold production and avoid having items stuck in warehouses. This is an important challenge that many businesses face when assessing their sales strategy [22], and many efforts address the problem of demand forecasting in different sectors, such as energy demand, ATM cash demand and product retailer demands [30, 25, 6]. In such work, the authors use time-series analysis on historical demand data to predict demand values for future time periods. Although effective, these methods only take into consideration temporal information related to the period in which demand occurred. Accordingly, the predictions are also given in terms of time only. This means that, although we may predict the demand that originates at a given time, we may not know from where demand originates, unless we work with several time series, each pertaining to a different location. A feasible approach, but one that may introduce certain disadvantages, like

reducing the amount of historical data available for training prediction models for locations with fewer demand patterns.

In recent years, researchers have attempted to use Machine Learning [28] methods to solve the problem of demand forecasting [5, 9, 23]. Such work shows us that we can use Neural Networks, Support Vector Regression [7] or even unsupervised learning to predict sales from historical data with results similar to or better than statistical methods. The advantage of using Machine Learning instead of classical time series analysis methods is that Machine Learning methods naturally account for multiple predictors.

Chang and Meneguzzi (2017, to appear) [8] run a benchmark between Gradient Tree Boosting [16, 17], Linear Regression [38], Multi-Layer Perceptron [18] and Support Vector Regression [36, 7], and one classical forecasting technique called Naïve Forecasting on two datasets. In this dissertation, we continue this work by envisioning an architecture that combines agent technology and Machine Learning algorithms for forecasting demand. We provide a formalisation of the Supply and Demand domain, which includes the *time-to-delivery* element and other demand related costs, and then evaluate our architecture using a selection of machine learning algorithms to predict sales based on limited temporal data, while also considering geographical data with respect to demand origin points. We run experiments with and without our architecture on the same datasets, and discuss the results we have obtained with each model and how our architecture helps improve these results.

This document is organised as follows. In Chapter 3, we discuss the specifications of the Demand Forecasting problem, outlining its elements and dynamics. In Chapter 2, we overview some of the concepts and theories we use to develop our work through the dissertation. We describe our main contribution, an architecture for improved demand forecasting, in Chapter 4. To validate our architecture, we run a series of experiments, which we detail in Chapter 5. In Chapter 6 we discuss a few works that are similar in objective to ours, and compare their results with the ones we have obtained. Finally, in Chapter 7, we summarise the main points of this dissertation, discussing our results, our limitations and possibilities of future work.

2. BACKGROUND

In this chapter, we discuss the theories and algorithms that we use to develop our work. We divide the chapter into two main sections: Software Agents and their use in solving distributed problems; and Machine Learning methods for prediction. Within the Machine Learning section, we discuss different types of Machine Learning, as well as the algorithms that we use.

2.1 Software Agents

Agents are entities that can receive perceptions via sensors and react to these perceptions by acting through actuators [34]. Humans and machines are therefore examples of what can be computationally modelled as agents, but this definition is open enough to include other entities in it as well, such as computer programs. We can think of the inputs of a computer program as its perception, to which it reacts by executing a series of instructions. It does not matter if the program has an output, so much as it *acts* by executing tasks. This abstraction is interesting because it allows us to program software based on agent models in order to imitate all sorts of behaviour seen in the real world: from very simple models to more complex ones. In Figure 2.1 we see an illustration of how an agent interacts with its environment.

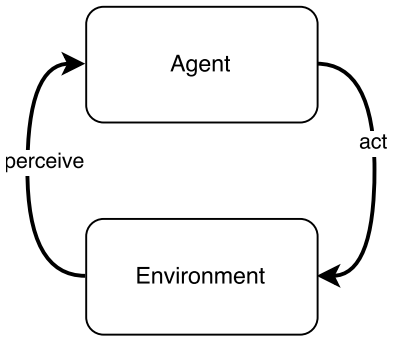

Figure 2.1 – An agent acts in an environment and perceives events that happen in it.

Software agents can be programmed using either everyday-use programming languages, such as Java, C, Python, etc. or agent specialised programming languages [35], such as AgentSpeak(L) [31], Jadex [29] and 2APL [13]. Choosing to work with an agent-oriented programming language implies the use of an agent architecture. Agent architectures determine the flow of information from the agents' sensors to its actuators: how the agent perceives the information, the steps it takes to treat that information and act based on it, and so on. Thus, the advantage of choosing this type of programming language is that the architecture part of the agent is already implemented, and so the developer may fully focus on programming the agent's behaviour. General-purpose programming languages, on the other hand, are useful for prototyping agents with simple behaviour, which do not need a complex architecture to perform their jobs well.

An agent can be purely reactive and execute tasks following a set of rules that are applied to the input, *e.g.* "If the coffee has finished brewing, send an alert to the writer.". The subsumption architecture by Brooks (1985) [3] is an example of such an architecture, in which it combines multiple modules that decide on the next steps of a robot based on its perceptions. Modules are stacked on top of one another, and the higher the modules are, the higher their priority over the others, thus making the agent always opt for the rule with the highest priority. But there are also more sophisticated architectures, such as Beliefs, Desires and Intentions (BDI) [32], which is based on the concepts of practical reasoning proposed by Bratman (1987) [2]. In this architecture, agents have an internal representation of their *beliefs*, facts about their environment that they believe to be true and will use as base for deciding on an appropriate course of action. Along with it, we have the notion of a *desire*, or *goal*, a state of affairs that the agent wishes to bring about in its environment. When the agent's beliefs support the pursuing of one or more of its desires, it commits to an *intention* and executes a sequence of tasks that ideally will take it to its goal.

An important distinction we must make when working with agents is between **rational agents** and **intelligent agents**. For an agent to be rational, it must be able to always choose the actions that maximise its performance. Intelligent behaviour, however, requires an agent to be proactive, reactive and social [39, 20], it does not involve making optimal choices. We have already covered reactive agents: they respond to perceptions from their environment. Proactive agents, on the other hand, have goals of their own that they try to achieve without an explicit call from the external world. So, even if we do not ask them to do anything, they can still work towards their own goals independently. Lastly, an agent is social when it is able to interact with other agents, either by exchanging messages or by delegating tasks. This has important implications in solving complex problems, as it allows us to use strategies to simplify solutions using multiple agents, where using a single computer program would be either too time consuming or too troublesome to program.

In most cases, agents do not act alone, and the very requirement that they should be social makes sure that they do not. As such, systems in which multiple agents interact in order to achieve a greater goal are known as **Multi-Agent Systems**. According to Jennings and Wooldridge (1998) [21], agents are useful for introducing modularity to a complex system. To develop our work, we use agents for precisely this reason. By designing a hierarchy of agents with similar architectures, we can program them to delegate tasks to others in order to solve a complex problem with a simple architecture. Multi-agent systems have been used in a similar fashion in areas such as industrial process control, air traffic control, commerce information gathering and filtering, business intelligence, health care, entertainment, among others.

2.2 Machine Learning

Machine Learning is a sub-area of Artificial Intelligence (AI) that comprises algorithms that learn computational models from data. These models then tell the computer how to perform tasks without any need of explicit programming. According to Mitchell (1997) [28]:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

A good analogy to ease the understanding of Machine Learning (ML) is this: suppose a black box process A that takes a number x as input and generates another number y as output. Since this is a black box process, we do not know what logic process A uses to make the number transformation. Normally, if solving this kind of problem by hand, we would take note of the various inputs we send to this process, along with the outputs, and try to figure out a pattern linking x's to y's. What we can do with ML is similar to that idea: we can feed our observations of inputs and outputs to an ML algorithm and let it find the pattern. Once trained, this algorithm then imitates the behaviour of process A, even though we have not programmed it specifically to do so. Now, if we keep feeding the algorithm with new experiences (observations from process A), then its performance should improve with respect to some error measure E.

Problems that ML algorithms can solve include regression, classification and clustering. In regression, the algorithm learns a model from a set of feature vectors (or predictors) $\mathbf{x}$ and their respective targets y by approximating a function that maps predictors to targets. After learning, we can use these models on unseen data to make predictions. Classification is similar to regression in that it maps feature vectors to targets, but the targets and outputs in this case are discrete labels (*e.g.* "Yes" and "No", "Spam" and "Not Spam", or any other finite set of classes), whereas in regression they are real numbers. Regression and Classification algorithms fall under the category of *Supervised Learning algorithms*, because we train them by passing the output we expect them to learn. Clustering, on the other hand, pertains to the category of *Unsupervised Learning algorithms*. The idea of clustering (and unsupervised learning, therefore) is the opposite of classification: we do not know how to classify the data, so we use clustering algorithms to find similarities among data points and group them into *clusters*. In what follows, we review three supervised and one unsupervised learning algorithms that we use in this dissertation.

2.2.1 Unsupervised Learning with k-Means

Unsupervised learning groups data points with similar characteristics, translating many distinct values into a discrete set of **clusters** that represent the data points they contain. Here we discuss *k-Means* [26], a general-purpose clustering algorithm that is simple to implement, but effective enough for experimentation.

The k-Means algorithm uses centroids to represent groups of assigned data points. Each group, or cluster, has a centroid, and the number of clusters depends on parameter k, which we must provide to the algorithm. There is no fixed optimal value for this parameter, so one must experiment with different values to decide which one is best for a given problem. One way to evaluate different values for k is to compute the Sum of Squared Errors (SSE) for the learned clusters. The SSE

is the sum of the distances between each datapoint and the centroid of the cluster to which the data point is assigned. As we experiment with different k values, we take note of the SSE indexes and compute the difference between the SSEs of two consecutive experiments. As the number of clusters grow, the resulting SSE decreases, but once the decrease from one setting to the next becomes insignificant[1], we can stop the search. *e.g.* suppose that the SSE decrease from $k = 4$ to $k = 5$ is of 50%, but from $k = 5$ to $k = 6$ it is of only 1%. This method is also known as the **elbow method**.

The algorithm is iterative, and the first step in an iteration is choosing initial centroids. Since we have no prior information about any clusters, we start by picking k random centroids. The second step in an iteration is assigning a cluster to each datapoint based on the distance between the datapoint and the cluster's centroid. In this way, the algorithm assigns each datapoint to the cluster with the closest centroid. Now the process goes back to choosing new centroids, but this time the choice is not random. For each cluster, we compute the mean of the data points contained in it, and this value becomes the new centroid. These iterations carry on until no new cluster assignments occur. By the end of this process, we have (1) a configuration of clusters we can use for classifying new entries, and (2) a set of classified data points from which we can extract statistics to use with other algorithms.

2.2.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) [18], or simply Neural Networks (NNs), are a computational model inspired by the human brain and the way it processes information by propagating signals through a network of neurones. These models are commonly used in Machine Learning for their nonlinearity and generalisation potential, *i.e.* deriving answers from examples that were not present during the training phase of the network. There are a few ways to set up a Neural Network. They can contain a single neurone, a configuration that is known as a *Perceptron*, or contain multiple neurones organised in layers, a configuration that receives the fitting name of *MLP*. The difference between them is that the Perceptron model is only capable of solving problems that are linearly separable, while MLP can solve multi-class classification and even **regression** problems. In this dissertation, we discuss the latter.

Multi-Layer Perceptron

In a Multi-Layer Perceptron [18], we organise neurones and their weights in fully-connected layers in a way that the information entering the network can be propagated through the network. The idea is that, as one layer finishes processing its input, it relays its output for the next layer to process, until the information reaches the last layer. In other words, each neurone transforms the input based on its weight value and propagates the transformed information forward, eventually

[1] We should define the criteria for "insignificance" on a case-by-case basis.

transforming the input into a response. Equation 2.1 shows the total input value of an arbitrary neurone in the network, which is obtained by multiplying the weights of the connections from the previous layer by their respective outputs and summing the results.

$$x_j = \sum_i y_i w_{ji} \tag{2.1}$$

There are three types of layers in any one MLP: the input layer, the hidden layer, and the output layer. To aid in the explanation that follows, we refer to Figure 2.2. The input layer contains the neurones that perceive the information from which to learn, which means there is a neurone for each feature in the dataset. Hidden layers stand between the input layer and the output layer, and help smooth the weights of the network during learning. There can be many hidden layers, depending on the complexity of the model we are trying to learn. The advantage of using hidden layers is that they allow the model to learn the relevancy of features through the transformation of weights performed at each neurone. For each hidden layer there is an extra *bias* node that introduces the constant value 1 to all neurones in the layer. This helps the model to generalise better to cases in which all inputs are equal to 0. Finally, there's the output layer. The neurones in the last hidden layer propagate their output to the neurones in the output layer, and the neurones in the latter layer compute a final output value.

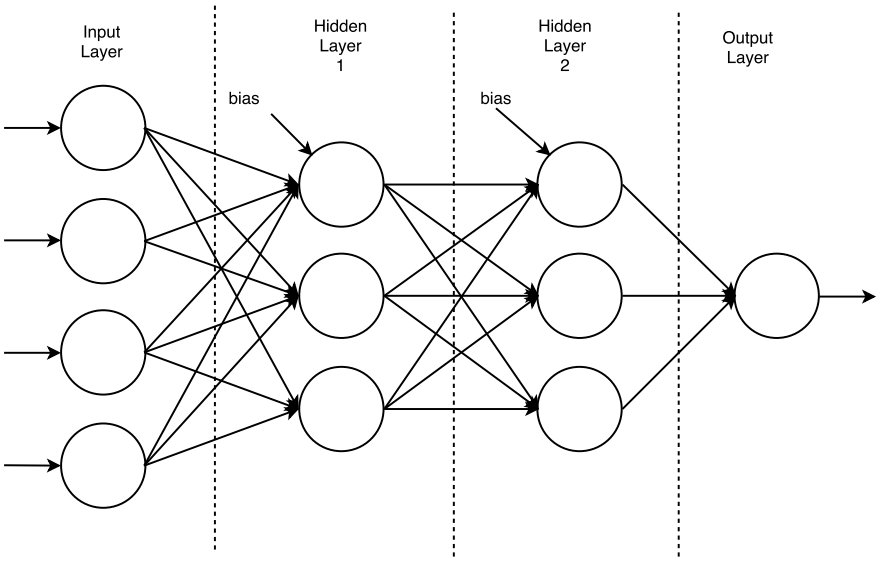

Figure 2.2 – An example of an MLP with two hidden layers. Each hidden layer has three neurones. The arrows leading from one neurone to the next represent the weights of the network.

Each neurone in the network is a processing unit which takes the input and transforms it into the output according to some **activation function**. The most basic activation function is the step function, which transforms the input data into binary output. This function uses a value θ to control the output: if the input is less than θ, the output is 0, if it is greater or equal to θ, then the output is 1. More sophisticated activation functions include the Rectified Linear Units (ReLU) type, in which the output value is defined by the function $f(x) = \max(0, x)$. In other words, a ReLU

activation function transforms negative input values to 0, therefore signifying that the neurone is not activated. If the input is greater than zero, then the neurone propagates it through the network without modification. Figures 2.3 and 2.4 show the plots for these two functions.

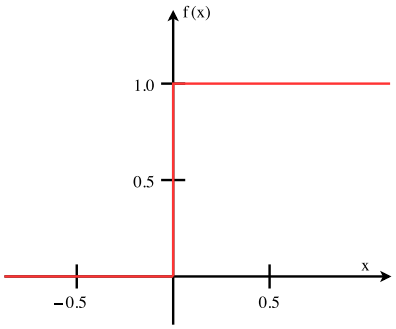

Figure 2.3 – A step function with threshold 0. Negative numbers deactivate the neurone, while positive numbers activate it.

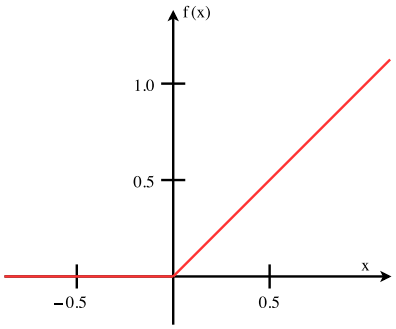

Figure 2.4 – The rectifier function, which transforms negative values to 0, leaving all positive values as they are.

Backpropagation

We have seen so far that Multi-Layer Perceptrons are organised in layers, where each layer is fully connected to the next. Each connection between neurones has a weight that is multiplied by the input, and each neurone has an activation function to transform the resulting value into an output. In this way, the network propagates the input forward, layer by layer, until it reaches the output layer. The ideal number of layers in the network can only be found by experimenting with different structures, and the activation functions depend on the problem we are trying to solve. What remains for us to define is the optimal set of weights for the network, which require a more sophisticated approach: the **backpropagation** [33, 18] algorithm.

In **backpropagation**, we take advantage of the network's structure to make the training data flow in two directions: *forward* and *backward*. Forward propagation is what we use to obtain predictions: the network feeds the output layer using the input signals and the weights of the network. To run the first iteration of forward propagation, we initialise the network's weights with random values between -1 and 1. At each iteration, we feed the algorithm with a set of examples $(\mathbf{x_i}, y_i), \ldots, (\mathbf{x_0}, y_0)$, and then compute the error between the network's output and the expected output value. The error is then sent back through the network in the opposite direction, with the objective of adjusting the weights of the network according to some optimisation function. We can do this by computing a delta for each weight according to Equations 2.2, where φ' is the derivative of the activation function on the input i of layer l, which is then multiplied by the sum of the weights multiplied by the deltas of the next layer.

$$\delta_i^l = \varphi'(\mathbf{x}^l i) \sum_{j=1}^{N^{l+1}} w_{ij}^l \delta_j^{l+1} \tag{2.2}$$

In the case where the next layer is the output layer, then the delta is a loss function applied to the network's output and the target. After computing the deltas, we obtain the gradient of the weights by multiplying the deltas by the activation function on the input, according to Equation 2.3.

$$g_{ij}^{l} = \frac{\partial e}{\partial w_{ij}^{l}} = \varphi(\mathbf{x}_{i}^{l})\delta_{j}^{l+1} \tag{2.3}$$

To update the weight, we multiply the gradient by a value α, which controls the learning speed of the algorithm (therefore aptly called the *learning rate*), and subtract it from the current weight value, as shown in Equation 2.4 After the weights are updated, we go back to the forward propagation step. There are two commonly used stopping criteria for this iterative process: an iteration limit, or an error threshold.

$$w_{ij}^{l} = w_{ij}^{l} - \alpha g_{ij}^{l} \tag{2.4}$$

2.2.3 Support Vector Regression

Support Vector Regression [14, 36] is a machine learning algorithm that is adapted from the Support Vector Machine (SVM) algorithm, which is used to solve classification problems. The SVM model, in Equation 2.5, uses a function φ on the input $\mathbf{x}$ to translate the input vector into a high-dimensional representation of its features, which is then multiplied by the normal vector of the hyperplane $\mathbf{w}$, and summed to a bias value b. To adapt SVM into Support Vector Regression (SVR), we must: (1) define a loss function, (2) establish the learning objective, and (3) choose an appropriate kernel function. In what follows, we discuss each of these steps.

$$y(\mathbf{x}) = \mathbf{w}^{T}\varphi(\mathbf{x}) + b \tag{2.5}$$

The loss function used by the SVR algorithm is the ϵ-*insensitive* function [37] of Equation 2.6, which considers any error within the range of $(t_n - \epsilon, t_n + \epsilon)$ (called the "ϵ-tube") not to be an error. In this equation, $y(\mathbf{x})$ is the model's prediction and t is the target.

$$E_{\epsilon}(y(\mathbf{x}) - t) = \begin{cases} 0, & \text{if } |y(\mathbf{x}) - t| < \epsilon \\ |y(\mathbf{x}) - t| - \epsilon, & \text{otherwise} \end{cases} \tag{2.6}$$

The graph in Figure 2.5 shows an example of a regression curve with its ϵ-tube. Absolute errors smaller than ϵ do not count as an error, but as a hit. Likewise, to scale the errors, we subtract ϵ from absolute errors larger than ϵ, which leads to the definition of slack variables ξ and $\hat{\xi}$ representing the areas above and below the grey area, respectively.

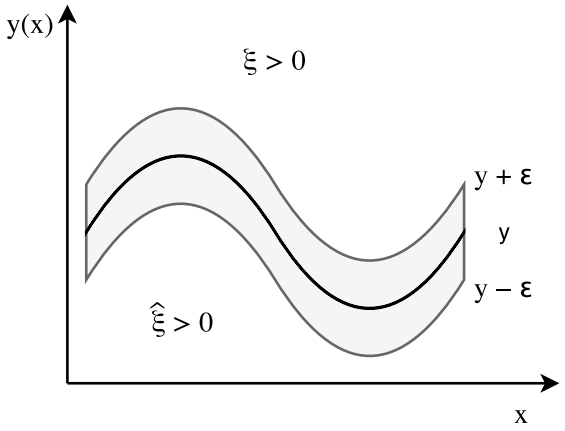

Figure 2.5 – A graph illustrating a 2-D Support Vector regression curve. The area in grey shows the range around the target in which errors are ignored.

To learn an SVR model, we must minimise the objective function:

$$C\sum_{n=1}^{N}\{E_\epsilon(y(x_n)-t_n)\}+\frac{1}{2}||\mathbf{w}||^2$$

By introducing slack variables ξ and $\hat{\xi}$, learning becomes a constraint optimisation problem, with the objective function becoming:

$$C\sum_{n=1}^{N}\{(\xi_n+\hat{\xi}_n)\}+\frac{1}{2}||\mathbf{w}||^2$$

and the constraints being the ones shown in Equations 2.7 to 2.10.

$$\xi_n \geq 0 \tag{2.7}$$

$$\hat{\xi}_n \geq 0 \tag{2.8}$$

$$t_n \leq y(\mathbf{x}_n)+e+\xi_n \tag{2.9}$$

$$t_n \geq y(\mathbf{x}_n)-e-\hat{\xi}_n \tag{2.10}$$

There are two regularisation terms this equation: C and $\frac{1}{2}||w||^2$. The C parameter controls the degree at which errors affect the model, while $\frac{1}{2}||w||^2$ controls the complexity of the model. These two parameters together control the tradeoff between a complex and rigid model, and a more flexible model albeit with more errors.

To solve the minimisation problem, Smola and Schölkopf (2004) [36] and Bishop (2006) [1] use Lagrangian multipliers and functions to transform the problem into the maximisation of the Lagrangian Equation 2.11, where a, $\hat{a}$, μ, and $\hat{\mu}$ are Lagrangian multipliers constrained by $0 \leq a_n \leq$

C, $0 \leq \hat{a}_n \leq C$, $0 \leq \mu_n \leq C$, and $0 \leq \hat{\mu}_n \leq C$.

$$\begin{aligned}\tilde{L}(a, \hat{a}) = -\frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}(a_n - \hat{a}_n)(a_m - \hat{a}_m)K(\mathbf{x}_n, \mathbf{x}_m) \\ -\epsilon\sum_{n=1}^{N}(a_n + \hat{a}_n)\sum n = 1(a_n - \hat{a}_n)t_n\end{aligned} \tag{2.11}$$

During this transformation, the authors also derive the following equivalences:

$$\mathbf{w} = \sum_{n=1}^{N}(a_n - \hat{a}_n)\varphi(\mathbf{x}_n) \tag{2.12}$$

$$a_n + \mu_n = C \tag{2.13}$$

$$\hat{a}_n + \hat{\mu_n} = C \tag{2.14}$$

From which we then use Equation 2.12 to substitute $\mathbf{w}$ in the SVM Equation 2.5, resulting in:

$$y(\mathbf{x}) = \sum_{n=1}^{N}(a_n - \hat{a}_n)K(\mathbf{x}, \mathbf{x}_n) + b$$

which is the equation that represents the SVR model, and therefore the one we use to make predictions. $K(\mathbf{x}, \mathbf{x}_n)$ is a kernel function that computes the distances between two input vectors $\mathbf{x}$ and $\mathbf{x}'$ in the high-dimensional space they are translated to with their respective φ functions. Finally, the bias value b is obtained by

$$b = t_n - \epsilon - \sum_{m=1}^{N}(a_m - \hat{a}_m)K(\mathbf{x}_n, \mathbf{x}_m)$$

The kernel function is the very advantage of Support Vector algorithms, as it allows the algorithm manipulate the data in a high-dimensional space without needing to perform costly operations that are peculiar to these spaces. *i.e.* if we were to translate feature vectors into high-dimensional space, we would need to compute the coordinates for each sample in order to obtain the distances between them, whereas with kernel functions we can use dot product operations between pairs of feature vectors for the same effect, but with a lower computational cost. In our work, we use the Radial Basis Function (RBF) for its synergy with Gaussian distributions. This kernel function has the form of Equation 2.15, where $||\mathbf{x} - \mathbf{x}'||^2$ is the squared Euclidean distance between a pair of feature vectors $\mathbf{x}$ and $\mathbf{x}'$, and σ is a free parameter. In SVR implementations such as *libsvm* [7], the denominator is substituted by a parameter γ, which equals to $\frac{1}{2\sigma^2}$, resulting in the kernel function shown in Equation 2.16.

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\sigma^2}\right) \tag{2.15}$$

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\gamma||\mathbf{x} - \mathbf{x}'||^2\right) \tag{2.16}$$

2.2.4 Tree and Gradient Tree Boosting

We use Gradient Tree Boosting [16, 17] to solve the regression problem, particularly the *eXtreme Gradient Boosting* (*XGB*) implementation developed by Chen and Guestrin (2016) [10]. The idea of Gradient Boosting is to combine several functions to build a model that fits the dataset used for training. In XGB, we use decision trees to learn such functions. Decision trees, as the name implies, are useful for separating a dataset into different categories, or *leaves*, according to some decision criteria related to the dataset's distinct column values. Since we are working with a regression problem instead of classification, the leaves of the decision tree are not categories, but continuous valued scores. The value predicted by the model, then, is the sum of the scores pertaining to the leaves that were activated according to the predictor variables.

Consider a dataset $\mathcal{D}$, with predictor variables $\mathbf{X}$ and targets $\mathbf{y}$. The Tree Boosting prediction model is a function ϕ that maps a set of features $\mathbf{x_i}$ to a prediction $\hat{y}_i$. In turn, the ϕ function is a sum of the weights w of the predictor trees q given the input $\mathbf{x_i}$, as shown in Equations 2.17 and 2.18. The idea is that each tree has its own structure q and number T of leaves. According to the input $\mathbf{x}_i$, each tree returns a weight value w, and the algorithm sums all returned weights to compose a prediction.

$$\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i) \tag{2.17}$$

$$f_k(\mathbf{x}) = w_{q(x)} \tag{2.18}$$

The learning objective of Gradient Tree Boosting is the minimisation of the error between predicted and actual values. In its most basic form, this minimisation takes the form of Equation 2.19, where l is the loss function that represents the prediction error, and $\Omega(f_k)$ is a function that regularises the learning task to control overfitting. To conform to the *gradient* part of the algorithm, we modify the learning objective by removing the constant loss term and substituting it for the first and second order gradients of the loss function multiplied by the prediction function $f_t(\mathbf{x}_i)$ at iteration step t on the input $\mathbf{x}_i$, as shown in Equation 2.20. In this version of the minimisation step, g_i is obtained by $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$, while h_i is obtained by $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$.

$$L(\phi) = \sum l(\hat{y}, y) + \Omega(f_k) \tag{2.19}$$

$$L^{(t)} = \sum_{i=1}^{n} [g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t) \tag{2.20}$$

The structure of the tree and the number of trees in the model depend on what the best splits in the dataset are. To find such splits, we use Equation 2.21 to score an arbitrary tree structure q, where T is the number of trees in the given structure, I is the set of instances within a single

tree, λ is a regularisation parameter and γ controls the point at which we need a new split in the tree to minimise the model's error. Since gradients g_i and h_i are given in terms of the loss function, we want these values to be as low as possible. Consequently, we look for the lowest scores as well when looking for new splits in the tree.

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T \tag{2.21}$$

2.3 Discussion

In this chapter, we have reviewed important concepts on software agents, as well as a selection of Machine Learning models for solving regression problems. By using software agents, we can develop autonomous and distributed solutions for problems that are too complex, or too time-consuming for a single program to solve. Machine Learning generalises models from a set of training data, and we can use these models to make predictions on new data. We combine the advantages of both the distributive power of software agents with the predictive power of Machine Learning models to construct the architecture in Chapter 4. In the next chapter, we provide a formal description of the domain and problem that our architecture solves.

3. DEMAND FORECASTING

In this chapter, we describe Supply and Demand domain in two parts. First, we give a formal description of domain elements and their interactions in Section 3.1. In Section 3.1.1, we define the more specific problem of demand forecasting, which is derived from the domain description.

3.1 Problem Specification

The problem we describe in this section concerns the distribution of supplies in a network of locations. We represent this problem as a tuple $\langle G, I \rangle$, where G is an undirected graph representing the network's topology and I is the set of items distributable among the network. The network topology is a graph $G = \langle N, E \rangle$, where N is the set of locations and E is the set of paths connecting locations. Each tuple $N_k = \langle n, S, D, P \rangle$ describes a location, where:

- n is an identifier for the location;
- S is a list of pairs $\langle i, q \rangle$ of available items at that location, where i is an item and q is the quantity associated to that item;
- D is a list of pairs $\langle i, q \rangle$ of items demanded at that location, where i is an item and q is the quantity associated to that item;
- P is a set of pairs $\langle i, c \rangle$, where $i \in I$ is an item that is producible at that location and c is its cost of production.

We represent the connections between these locations as tuples $E_k = \langle u, v, w \rangle$, where u is the origin of the connection, v is its destination and w is the distance between them.

We illustrate an example problem instance featuring 9 locations in Figure 3.1. A connection between two locations establishes the *supplies* relation between them, and we can use the logical predicate $supplies(locationA, locationB)$ to describe this. This relation tells us that we can move items from $locationA$ to $locationB$. A location may dispatch supplies to any location connected to it, and may demand supplies from the same set of locations. We represent a demand request as a tuple $\mathcal{D} = \langle n, \mathcal{I}, \mathcal{U} \rangle$, where n is the demanding location, $\mathcal{I}$ is the set of demanded items and $\mathcal{U}$ is the utility value of the delivery. To measure the utility value of a demand request we combine three factors: the cost for producing demanded items, the cost associated with delivery logistics, and the time elapsed until all items are supplied to the demanding location. The general idea is that utility represents the quality with which a location fulfils a request, so a high utility means a fast and high quality delivery, whereas a low utility means a slow and low quality delivery.

Distribution of items in the network is transitive, which means that if some location A requests some item x from location B, and location B lacks the ability to supply A with x, then B may request another location C for x. If C can supply x, then it supplies it to B, which then

supplies A with x. If C is also short on supplies of x, then C may request yet another location for x. There are two outcomes for this chain of events, either (1) the demand chain continues until some location is able to supply x, or (2) we discover a supply gap and conclude that we cannot supply item x at that moment. In case we discover a gap in supply, we must compensate for it by ordering the production of missing demanded items, so that we may supply them at a later moment. This, however, reduces the utility of the demand as producing an item only after some location has requested it increases the time needed for delivery.

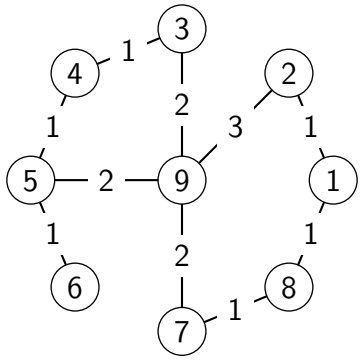

Figure 3.1 – A problem instance with 9 locations encoded as a graph.

One way to avoid supply gaps is for us to produce the item beforehand to guarantee its availability upon its requesting. Still, this is not a perfect solution because stocking items in a warehouse also incurs cost, and it is not guaranteed that the items we have produced will have an associated demand. With this in mind, we must account for a new factor in our calculations, which is the likeliness of an item's demand in the future. If some location is likely to demand an item, then it might be worth it to have it produced beforehand, increasing its future delivery utility. If it is unlikely that some location will demand a given item, then we may decide to skip pre-emptive production to avoid wasting unnecessary resources and have stale stock of said item. To obtain that factor, we must take into account that the domain evolves over time and study how that process takes place as a time series. In time series, the values assigned to each variable in the domain may change between time steps. In our problem, it means that if a location $n \in N$ with demand D_t^n receives a supply package S' at $t+1$, then $D_{t+1}^n \neq D_t^n$, meaning that its demand list is no longer the same at this posterior time. The same applies to the inventory S of the supplier, which diminishes after sending supplies to another location, therefore $S_t^n \neq S_{t+1}^n$. By making observations over how the system evolves during a certain time range, we can use that information as a basis to predict the future states of supply and demand lists. If we are able to predict demand lists and the likeliness of demand of each item in these lists, we can decide whether to produce items in advance and stock them, or to skip inventory and wait for actual demands. We refer to this problem as **supply and demand forecasting**.

We apply the task of prediction over the D list. For each location, we try to predict D_{t+k}^n for each n, where $t + k > t$ is a time step in the future and n is the identifier of a neighbouring location. By predicting D_{t+k}^n, we can direct production efforts to have some or all of the items ready by that time step, speeding up time of delivery and reducing costs incurred by slow delivery

times. From predicting D^n_{t+k} comes a decision problem: whether to produce items in advance or wait for actual demand to occur. Some factors might influence this decision, such as the trade-off between producing an item in advance and risking not actually distributing it, or waiting for an actual demand to arrive, but wasting extra time in the ad-hoc production of the item. We define the cost for supplying an item i from a location x to a location y as a function $f(i, x, y) = g(x, y) + h(i, x) + \varphi$, which sums the cost $g(x, y)$ of transporting item i from x to y; the cost of production given by the function $h(i, x)$; and a value φ that scales in proportion to the time taken between request and delivery. If we produce the item in advance, the cost of delivery changes to $f(i, x, y) = g(x, y) + h(i, x) + \psi$, which uses the same cost functions for transportation and production, and a value ψ representing the cost incurred from having the item stuck at that location. We remove variable φ from the equation, as we no longer assume any extra time between request and delivery. Hence, we can make the decision by selecting the option with the smallest value between φ and ψ:

$$\arg\min(\varphi^n_t, \psi^n_t)$$

We define the goal of prediction as the minimisation of the accumulated values of φ and ψ through supply cycles.

$$\min \sum_{n}^{t} \varphi_n + \psi_n$$

3.1.1 Demand Forecasting Formalisation

The Demand Forecasting (DF) problem is a subset of the Supply and Demand Forecasting (SDF) problem we have described in the previous section. Whereas in SDF we have that any location is able to make supply requests and deliveries, in DF we have that some locations are only able to make requests, while others are only able to make deliveries. We refer to locations that can only make requests as *customer regions*, and to locations that supply customer regions as *warehouses*. This subproblem is more suitable for customer supply chains, where end customers are the demanders of supplies, but never the distributors. Thus, we represent a customer location as a node $n \in N$ in which $\forall.t : S^n_t = \varnothing$, *i.e.*for all time steps, the set of supplies available at location n is empty. The same applies to the set P of producible items, where $\forall.t : P^n_t = \varnothing$. We represent warehouses as nodes of the type $n \in N$ in which $\forall.t : D^n_t = \varnothing$, *i.e.*for all time steps, the set of demands at location n is empty. We also introduce a new assumption on the list of producible items P, which is that $\forall.n : P_n = I$, *i.e.*all locations are able to produce all items in the I set. In this situation, we use the items's costs of production to abstract the transitivity of distribution between warehouses and supply gaps. Note that we could make the same assumption in the more general SDF problem, but we choose to make it in DF instead to focus our attention on the demand lists D of customer regions.

In DF, we use a graph to represent the network's topology, but instead of using an undirected graph, we switch to a mixed graph like the one we show in Figure 3.2. In this representation,

edges[1] indicate a mutual supply relation between locations, so that we can move supplies back and forth in the chain via that location. We use arcs to indicate the one-way supplying of a location, with the receiving end representing a customer region.

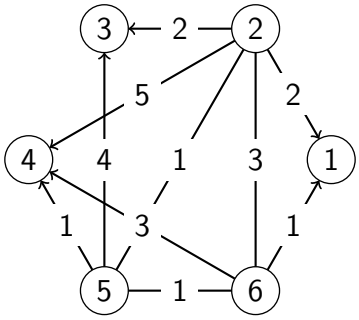

Figure 3.2 – A problem instance with 6 locations encoded as a graph. Of the 6 locations, 3 are distributing locations (warehouses), identified by the numbers 2, 5 and 6. The 3 remaining locations 1, 3 and 4 are customer regions.

Given these new assumptions, the task of prediction in the Demand Forecasting problem becomes to predict the values of D^n_{t+k}, where n is a customer region. We identify a customer region as any node $n \in N$ such that there are no edges or arcs in E which originate from n. If we can infer the list of items required by a given location n at some future time step $t + k$, then we can direct production efforts to have some or all items of that list produced by that time step, thus reducing the value of the φ factor in cost calculations.

[1] Edges are the same as undirected edges, while arcs are directed edges.

4. IMPROVING DEMAND FORECASTING

In this chapter, we describe a Multi-Agent System architecture that combines agents and machine learning to forecast the demand of products based on time and location information. We begin by an overview of the architecture of our forecasting system, then we move to the architecture specific to the agents operating in the system, and then move on to the description of the Multi-agent part of the system, in which agents interact to make predictions.

4.1 Forecasting Machine Architecture

We design the forecasting machine architecture illustrated in Figure 4.1, which includes a module for pre-processing the dataset, one for computing statistics, and another one for separating the dataset into subsets. An important assumption is that the problem involves multiple locations and multiple products to predict from, with the additional challenge of each location having a different demand pattern for each product. Therefore, there are two important requirements for this architecture to work: first, the dataset must contain both product and location information; second, the number of combinations of products and locations in the dataset must be greater than one.

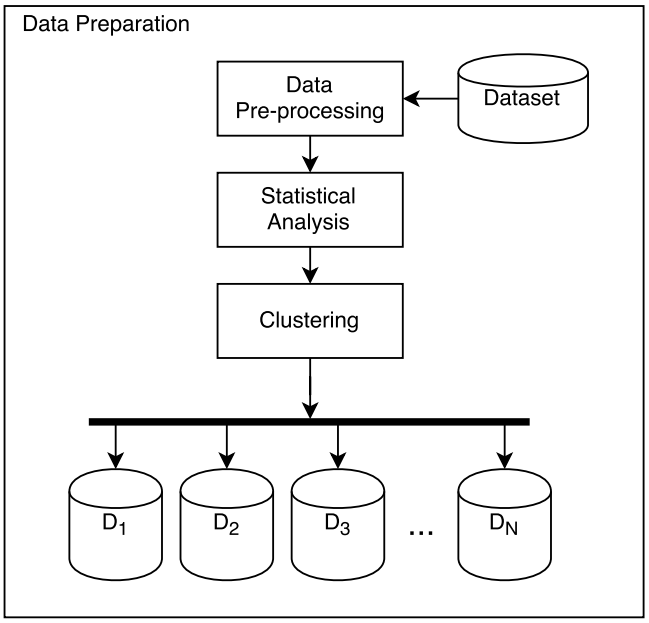

Figure 4.1 – The architecture enacted by the data preparation algorithm.

The first module is responsible for processing the source dataset, translating the data into a format that Machine Learning algorithms handle. *i.e.* if there are textual information in the original dataset, then Machine Learning algorithms cannot deal with it, so we must transform this into numbers. This module's function depends entirely on the dataset we are working with, and so there is no general algorithm to explain here. In our experimentation chapter, where we discuss dataset peculiarities, we go over the preprocessing details for each dataset. For now it suffices to say

that this module reads a dataset in a *human readable* form and translates it into *machine readable* form.

The second module is the statistics module, in which we run an algorithm that reads the dataset row-by-row and collects information with respect to the demand quantities of each item at each location. Our objective here is to count how many learning examples we have of each $\langle location, product \rangle$ pair, and to compute the mean and variance of the distribution of demands amongst these pairs. What these three pieces of information tell us is how stable our learning examples are. *e.g.* if there are few examples to learn from, and the demand quantities annotated in them have high variability, then they belong to a group in which accurate prediction is unlikely; if there is a large number of examples, and the quantities have low variability, then they belong to a group whose behaviour may be easier to predict.

The idea is to separate the data into groups with different levels of **predictability**. This becomes possible with the third module: Clustering. By running a clustering algorithm with the information collected in module two, we obtain groups of $\langle location, product \rangle$ pairs with similar demand patterns. Once again, we highlight the fact that the number of clusters depends on the dataset at hand. One could use methods like the **elbow method** (discussed in Section 2) to find an ideal number of clusters. In our case, we use the elbow method with a metric called Coefficient of Variation (CoV), which is the ratio between the standard deviation of a distribution and its mean (Equation 4.1). The lower the CoV, the more stable the clusters are. This metric is more appropriate than the standard Sum of Squared Errors (SSE) since our objective is not to minimise the clustering error, but to find the most stable clusters.

$$q(\mathbf{x}) = \frac{\sigma}{|\mu|} \tag{4.1}$$

The last step in this process is to effectively divide the dataset into parts. As we obtain the clusters from module three, we scan the processed dataset for the $\langle location, product \rangle$ pairs in each cluster to recreate smaller datasets containing information for those pairs only. With the subsets created, we can move on to introduce predictive agents in the system that access the datasets to make predictions.

4.2 Forecasting Agent Architecture

We use the agent architecture illustrated in Figure 4.2 to help us organise the handling of the dataset, parameter optimisation, training and evaluation of predictive models. The arrows in this diagram indicate a *feeds* relationship between processes, as in: $A \rightarrow B$ means Process A *feeds* Process B. Because we intend to coordinate several agents, it is important to define a clear set of tasks that they should perform, as well as the expected inputs and outputs of each task. There are four tasks that a forecasting agent must perform: Observation, Optimisation, Learning and Prediction.

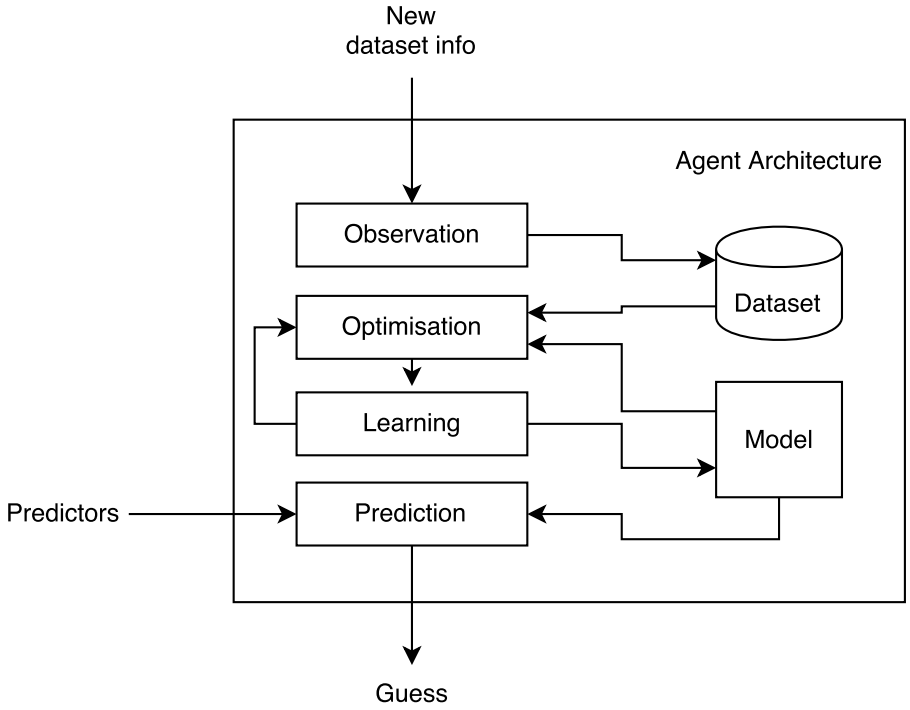

Figure 4.2 – The agent architecture we use for predictive agents.

In Observation, the agent maintains an internal representation of the dataset so that it can use for the other three tasks. Whenever the dataset is updated with more data, so must the agent update its records with the new data. Keeping the dataset updated is important to ensure that agents do not ignore relevant historical information to make their predictions.

In Optimisation, the agent uses the data it has stored to train its internal prediction model and find the parameters that optimise its predictive performance. In this step, the agent trains multiple instances of the same model, each with a different set of parameters, and elects the set of parameters that brings about the model with the lowest error for the current state of the dataset. For this reason, optimisation should follow updates on the dataset, and every time the agent finds a better parameter set, it should update its internal model to reflect its discovery. There are two factors we must watch out for in programming this task, however: First, the parameters that the agent experiments with must be established beforehand by the programmer, as each predictive model has its own relevant parameters. *i.e.* there is no general set of parameters that suits every predictive model available. The second factor is the parameter search space. Because the agent does an exhaustive search for the right parameters, the more options we give to it, the longer the optimisation process takes. Depending on the model we are training and the amount of data at hand, the task might just become impractical. So we should be careful when choosing parameter sets for the agent to try. In our implementation, whenever agents find the optimal set of parameters for a given dataset configuration, they save the parameters in a file in case they need to retrain the model on the same configuration.

Next comes Learning, where the agent uses learning algorithms to update its internal predictive model using the parameters obtained during the Optimisation step. Optimisation and

Learning run repeatedly until the agents elects the best parameter set. When the election is over, Learning trains a definitive model and stores it in memory for use with the Prediction module.

Finally, in the Prediction block, the agent uses the latest learned model, output from the Learning block, to make predictions. To feed this process, we must pass a set of predictors to the agent after it finishes its training. The output of the Prediction block is a vector of real numbers corresponding to the predicted demand for each set of predictors that were passed to it.

4.3 Combining Agent Effort

With the datasets created in the data preparation process (Figure 4.1) and the predictive agent architecture designed in the previous section, we can now build a Multi-agent architecture where multiple agents interact to make predictions. Our motivation for using more than one agent to make predictions is that it allows us to test different learning models for the same dataset at once. As such, we propose the architecture shown in Figure 4.3.

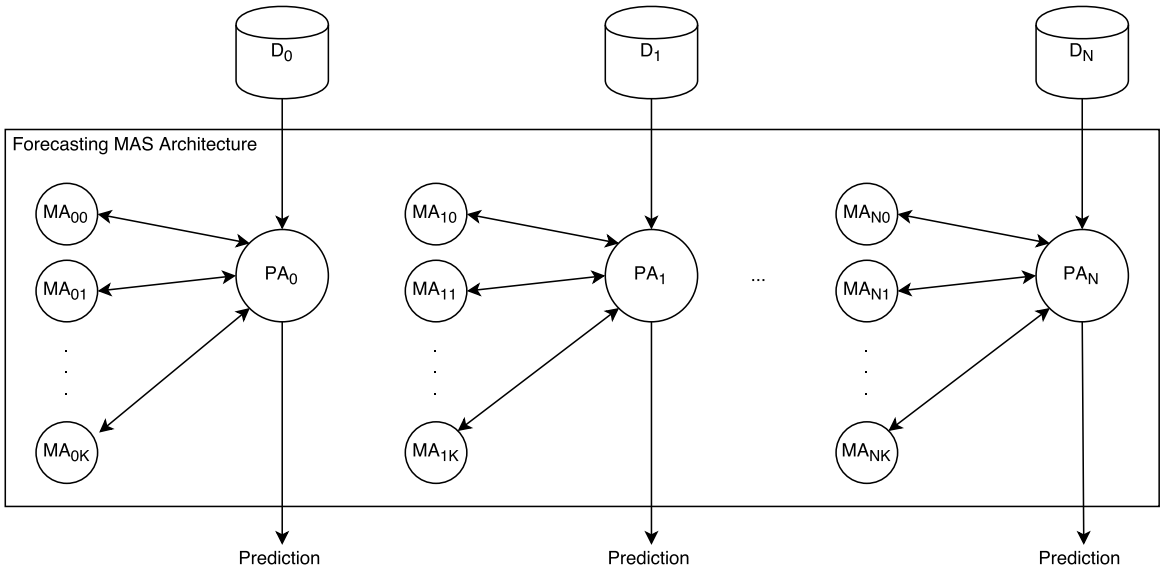

Figure 4.3 – The multi-agent forecasting architecture for N datasets and K models. PA stands for *Predictive Agent*, the one that outputs a prediction. MA stands for *Model Agent*, the one who trains a predictive model.

In this architecture, we have a group of agents set up for each of our data subsets. These groups obey a hierarchy of *master-and-servant*, in which the *master* delegates the learning task to its *servants*. Each *master* is assigned to a subset, and we request any predictions directly from these agents. In turn, *masters* will initialise their *servants*, which then learn a predictive model from the associated subset and stand by to respond to any prediction requests from their *masters*. The idea is that each *servant* should use a different prediction model than their peers. Whenever *servants* make a prediction, they pass that on to the *master* for it to decide on a final prediction.

There are two advantages to this approach. The first one is the parallel processing power of this architecture. If we were to use a single agent to train and evaluate all of the models, as well as make the final prediction for each of the subsets, then the time required to complete the whole process could easily become prohibitive, especially as the dataset grows with more historical data added to it. By using a multi-agent system, we can configure it so that agents can train and evaluate models independently, and for different subsets at the same time. The second advantage is the role of the *master* agent in this distribution scheme. Although *masters* delegate the task of learning, they must still decide on a final prediction. Thus, the existence of *master* agents allows us the liberty of choosing the best way to use the information that comes from *servants*. *e.g.* we can average the results that are output from all of them; we can make an ensemble of the results; or we can choose the one model with the best evaluation for each dataset.

The overall idea of this setup is to use a simple structure of agents to solve a problem that would be too costly or even impractical to solve with a single computer program. It is similar to a *divide and conquer* approach, we first divide the dataset into pieces so we can use a variety of workers to make predictions on each piece, as if they were isolated from the start. The expectation is that we improve prediction accuracy in the subproblems, and by averaging the results we obtain a better accuracy than that of a centralised approach. In the next chapter, we run experiments using this architecture and compare the results with that of centralised approaches.

5. EXPERIMENTS AND EVALUATION

In this chapter, we discuss two datasets that we use for experimentation, as well as the very experiments that we run on each to test our architecture. The first dataset is simpler than the second, and yields better performance when using mainstream demand forecasting methods. Our objective, therefore, is to use the first dataset as a reference point for the performance of our architecture.

5.1 Error Measures for Forecasting

To choose between different prediction models, we need to quantify their prediction capability by measuring how far from the truth they are. The simplest way to evaluate such models in demand forecasting problems is to compute the difference, or **error**, between predicted demand and actual demand values. There are different ways in which we can obtain an error measure from a prediction model, and each one has its own peculiarities. In what follows, we describe four error measures that we use in this dissertation to discuss the results of our experiments.

The first error measure is the Mean Absolute Error (MAE) and has a rather straightforward interpretation: it defines the range of values above and below the target in which predictions tend to fall when using the model. We obtain MAE by averaging the absolute differences between predicted and actual values:

$$E = \frac{1}{N}\sum_{n=0}^{N} |y(\mathbf{x_n}) - t_n|$$

Consider a vector $y(\mathbf{x}) = [0.5, 2, 1]$, a prediction output by a hypothetical model, and $t = [0.7, 1, 0.8]$, the target values. The MAE of this model is of approximately 0.46667. This means that, on average, the model misses the target by 0.46667 units, sometimes more, sometimes less. The advantage of using MAE is that it is easy to understand, the lower the value, the better the model is. However, this measure serves only to compare different models on the same dataset, as it depends directly on the scale of values within that dataset, *i.e.* a value of 3 in dataset A is not necessarily better than a value of 10 on another dataset B.

In the second error measure, the Root Mean Squared Error (RMSE), we substitute the absolute operator in the MAE equation for a square operation and extract the root from the resulting mean:

$$E = \sqrt{\frac{1}{N}\sum_{n=0}^{N} (y(\mathbf{x_n}) - t_n)^2}$$

Consider the same vectors of the previous example: $y(\mathbf{x}) = [0.5, 2, 1]$ and $t = [0.7, 1, 0.8]$. This time, after applying the RMSE formula, we obtain an error of 0.6. Although the total absolute error remains the same (1.4), it is higher than the MAE because of the magnitude of the difference between the second element of the two vectors. Because of the square operation, the RMSE weights heavier

on bigger differences, so it is a more robust measure than MAE. For the same reason, however, it is not as easy to interpret. Like with MAE, RMSE is a scale-dependent measure and can be used to compare models applied to the same dataset only.

The third error we use is the Mean Absolute Percentage Error (MAPE). Unlike with the previous measures, this one is scale-independent, meaning that it can be used to compare models that run on different datasets. The advantage of MAPE is that it is easy to interpret, as it represents the distance between predictions and targets as a percentage, so the lower the index, the better the model is. We obtain this measure by applying the formula:

$$E = \frac{1}{N}\sum_{n=0}^{N}\frac{|t_n - y(\mathbf{x_n})|}{t_n} \times 100$$

Following our sequence of examples, the MAPE for vectors $y(\mathbf{x}) = [0.5, 2, 1]$ and $t = [0.7, 1, 0.8]$ is of approximately 51.19048%, meaning that our model errs by that much, on average. Suppose we test the same hypothetical model on a different dataset, yielding the prediction vector $z(\mathbf{x}) = [10.4, 12.3, 6.6]$. Consider, now, that the targets for this dataset are $t' = [10, 12.2, 7]$. The MAPE for this new dataset is of approximately 3.51132%. Since it is a percentage, we can safely conclude that the model works better in the second dataset than it does in the first.

The fourth and final error we discuss here is the Mean Absolute Scaled Error (MASE). Like MAPE, this is a scale-independent error, and it is especially useful for comparing models on different datasets [19]. To obtain this measure, we compute the ratio between the MAE of a candidate model (the one we wish to evaluate) and the MAE of a reference model. In demand forecasting, the reference model we use is a naïve model, which is the most basic form of forecasting available. This measure is given by the formula:

$$E = \frac{MAE_{candidate}}{MAE_{reference}}$$

A result of less than one indicates that the candidate model surpasses the reference model, whereas a result greater than one indicates that the candidate model is worse. Suppose we have the vectors $y(\mathbf{x}) = [0.5, 2, 1]$, $t = [0.7, 1, 0.8]$, and a naïve vector $n(\mathbf{x}) = [2, 0.7, 1]$. The MAE between the naïve model and the target is 0.6. We know that the MAE between $y(\mathbf{x})$ and t is 0.46667, so the MASE between these models is approximately 0.77778, meaning that our hypothetical model is better than the naïve one. The advantage of using MASE is in the requirement of a reference model, which offers a base for comparison and lets us conclude whether the model is useful or not.

5.2 Bike Sharing Demand

The first dataset we discuss is the Bike Sharing Demand dataset [15], which is an open dataset that was used in a Machine Learning competition[1]. The Bike dataset contains 10.886 instances, and a total of 12 features, where 4 of these features are time-related (year, month, day and hour). The remaining features are related to weather information, such as season, temperature and weather condition. We choose this dataset to serve as grounds for comparison due to its contrast between its quality and simplicity. Quality with respect to its rich contextual information regarding time and weather, which are acceptable predictors of bicycle demand, and simplicity in that it is straightforward in its semantics: each sample corresponds to a single point in time, and all demand corresponds to a single item: bicycles. If not for the weather information included in the dataset, this problem could be easily modelled as a Time-series prediction problem, and solved with standard statistical methods.

5.2.1 Experimentation

Given that this dataset contains only one item and no location information, we do not use our architecture in this experiment. Instead, we use the same ML models we would have used directly on the dataset. As we can see in Table 5.1, the XGB model is the one with the best performance. In fact, it is the only model to surpass the naïve model, with an improvement of 0.3065 in the MASE index. Meaning that it is more advantageous to use XGB than to naïvely forecast bike demand. Compared to the other methods, however, naïve forecasting is more advisable.

We include the RMSLE (Root Mean Squared Log Error) in these results, because it is the error measure used to score participant entries in the competition. Our RMSLE of 0.33201 is slightly lower than the first place in the public leaderboard of the competition, who won with an index of 0.33757. [2] With these results, we have a base for comparing the performance of this model with that of other datasets.

5.3 Computer Retailer Demand

In this section we describe the peculiarities of the dataset we were provided by a computer retailer regarding their sales history. We detail the information in the dataset and how we adapt

[1] https://www.kaggle.com/c/bike-sharing-demand

[2] We should clarify, however, that our RMSLE is only a reference, because it was not tested in the same manner as in the competition. In the leaderboards, the score is obtained from the participant's predictions for a testing period, in which the competitors do not have access to the answers. Instead, we divided the training dataset and selected 80% of its instances for training, and the remaining 20% for testing, so that we could calculate our own error measures to compare with our other models.

Table 5.1 – Forecasting errors for the Bike Sharing Demand dataset.

	Bike Demand				
	MAE	RMSE	RMSLE	MASE	MAPE
Naïve	59.91615	92.53395	0.65422	1	60.01745%
XGB	**40.73085**	**63.76418**	**0.33201**	**0.6798**	**24.75928%**
NN	156.99589	225.50814	1.08953	2.62025	96.81162%
SVR	168.88809	237.17341	1.19912	2.81874	206.97317%

it to conform to machine learning standards. The description contained here corresponds to the preprocessing module that we use in our implementation of the architecture.

5.3.1 Raw Demand Dataset

There are 5 columns in the raw dataset: *Year*, *Week*, *Item*, *Location* and *Demand*. Demand is given in terms of weeks, so the maximum amount of samples for a single year is of 52 for a given item-location pair. This is a limitation for training, considering the short lifespan of products in the computer equipments market. Yearly values are limited to 2015, 2016 and 2017[3], and within 2017 there are approximately five months worth of data[4], amounting to 126 weeks worth of data for each $\langle location, product \rangle$ combination. Note that 126 weeks is an *upper bound* on the number of training examples, in some cases the actual number of training examples is even less. With respect to Item and Location data we have a variety of 32 items and 50 locations, and 516 combinations available (not every locations orders every item). For each combination, there is an average of 55.379845 training examples, which is less than half of the upper bound of 126 examples per combination.

As far as *Demand* values are concerned, we can look at the scatter plot of Figure 5.1. The x axis in this graph works like a timeline, going from the first to the last week in the dataset. Demand is concentrated on the lower range of the y axis, with a few scattered outliers on the upper range. These outliers represent peaks in demand, standing out from the normal demand distribution, and therefore a possible challenge for learning.

5.3.2 Processing the dataset

The raw dataset provides limited data for forecasting. For each $\langle \text{location}, \text{product} \rangle$ combination there are at most 126 learning examples, and there is little variety of predictor variables to choose from. To augment the dataset, we run an algorithm that adds a new feature called "Previous Demand" by looking for the last observed demand of a given item-location pair. Our objective is to use the naïve forecast as a training component to see if we can improve the resulting accuracy

[3]The information comes in fiscal years, so the first fiscal week of fiscal year 2015 corresponds to the first week of October 2014, while the last week of fiscal year 2017 corresponds to the last week of September 2016.

[4]Converting to Gregorian dates, this would correspond to February 2016.

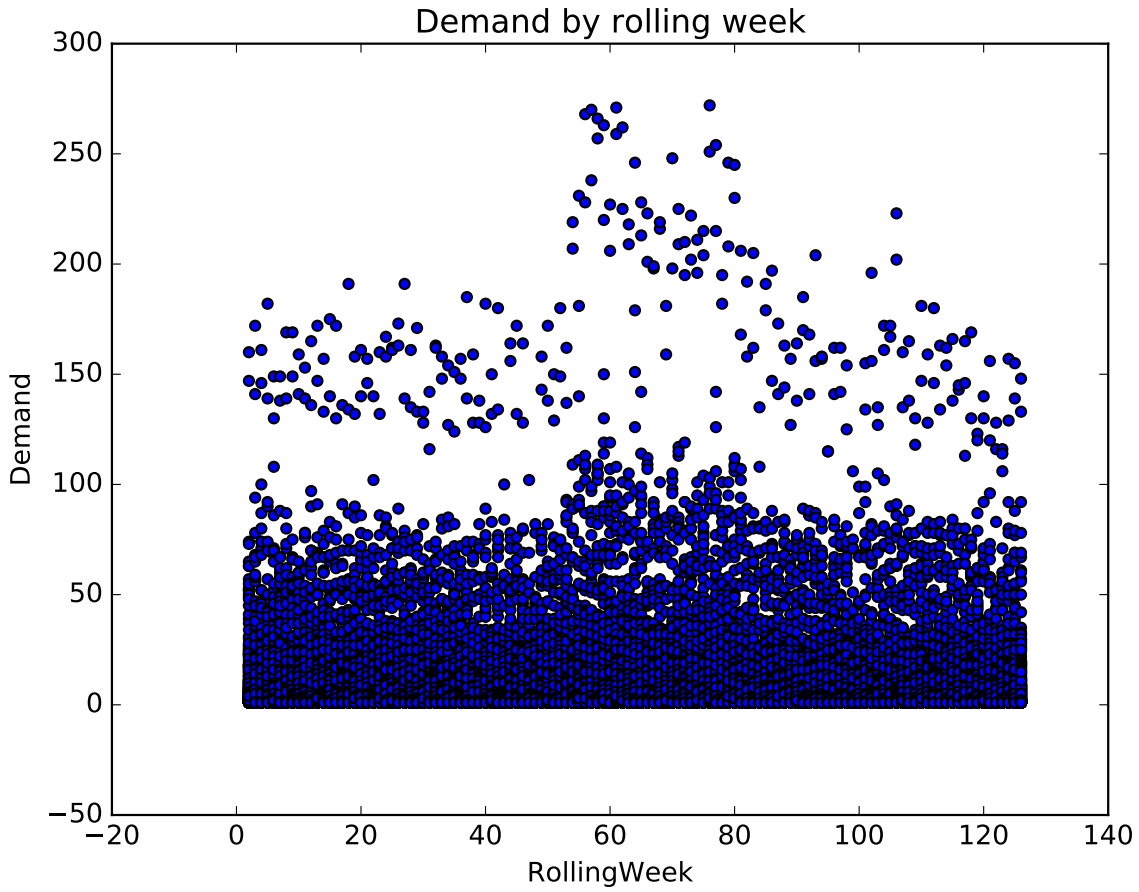

Figure 5.1 – Demand distribution per rolling week.

of the prediction model by offering it a hint. If the algorithm finds no previous demand, it then removes the row from the dataset altogether, eventually eliminating all 516 rows with missing data (approximately 1.602% of the dataset).

Because *Item* and *Location* are categorical features, and therefore represented as strings, they cannot be processed by the algorithms as they are. To solve this problem, we encode these features into a machine-friendly format by adding a new column to the dataset for each possible value of item and location. Then, we assign a value of 1 to the columns that correspond to the category of the training sample, and 0 to all other columns. Suppose that a dataset contains two locations and two items: we add four new columns to the dataset, two for each location, and two more for each item. If a training example corresponds to item 1 being order at location 2, then the columns *Location2* and *Item1* will have a value of one, while columns *Location1* and *Item2* will have a value of zero. This way, we can represent the different combinations of items and locations using only zeroes and ones.

In the form that they come, the *Year* and *Week* features allow the model to ponder the target demand according to the time period. However, this form of representation is problematic, as the integer values of *Week* are supposed to be circular (after week 52, the year increases, but the week value returns to 1). So using these values as they come can mislead the prediction model

to consider weeks 1 and 52 to be 51 units apart, when they are actually not. To work around this, we remove both *Year* and *Week* features to substitute them for a new feature called *Rolling Week*, in which we use numeric values starting at 1 to represent weeks continuously over time. This way, after week 52 ends, instead of going back to week 1, we proceed to week 53, which is the first week of the second year of training data, followed by week 54 and so on.

The final transformation that we make on the dataset is to smooth the variance of demand values for training by using natural logarithms instead of raw values, which gives us the scatter plot of Figure 5.2. As Lütkepohl and Xu (2012) [24] show, when the log transformation stabilises the data, this transformation can improve forecasting accuracy. In the raw dataset, the Variance to Mean Ratio $\frac{\sigma^2}{\mu}$ of demand values is of 37.71181. This ratio falls down to 1.0068 after applying log transformations, meaning that the variance has scaled down more than the mean after the transformation, and we can conclude that the data has become more stable.

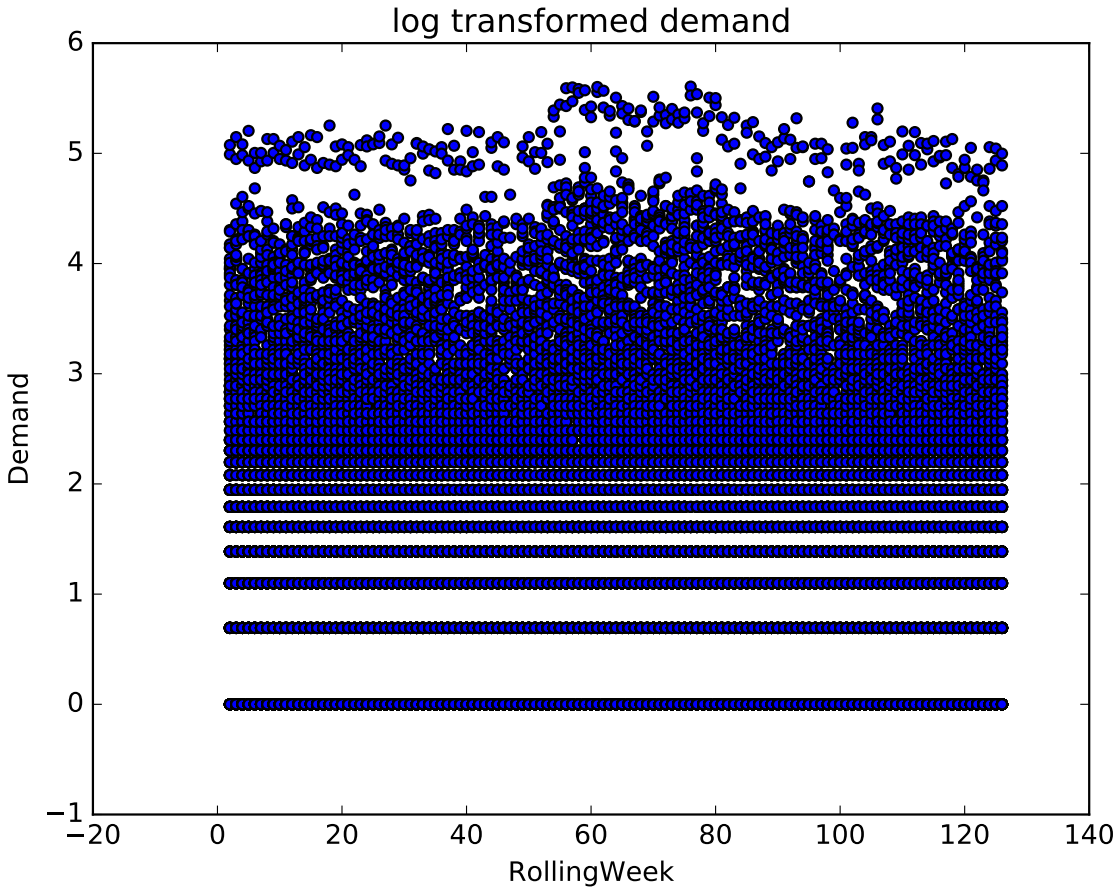

Figure 5.2 – Log transformation of demand distribution per rolling week.

5.3.3 Experimentation

Predicting with the entire dataset

For the first experiment we use four agents to make predictions on the whole dataset as a single cluster. The four agents are the Naïve agent, XGB, MLP and SVR. There are two versions of the dataset that we experiment on: the Base dataset, which contains the original information of the dataset (after preprocessing), and the Augmented dataset, which contains the additional *Previous Week* feature. Our objectives here are to see how the additional column changes the performance of each algorithm, and to take note of the performances of each agent when we use a single cluster for all the data. The approach is similar to what we did with the Bike Sharing Demand dataset: running prediction algorithms over a single dataset. This will give us the base against which to compare the performance of our architecture in further experiments.

As we can see in Table 5.2, forecasting demand using the entire dataset is not a good idea, no matter what model we use. The XGB agent returns with an MAE of 4.33662, which is slightly higher than that of the Naïve model, and the same is true for the RMSE index. XGB's MAPE index, however, is lower than that of the Naïve agent, which may seem strange at first, but has a plausible explanation. The MAPE measure is asymmetrical [27], so when the target is lower than the prediction, the percentage error we obtain is bigger than if the target was the highest value, even if their absolute differences are the same. What this shows to us is that the Naïve model tends to over-estimate demand, whereas XGB underestimates it.

Table 5.2 – Forecasting errors for the Computer Equipment Retailer Demand dataset.

	Base				Augmented			
	MAE	RMSE	MASE	MAPE	MAE	RMSE	MASE	MAPE
Naïve	3.99419	6.62672	1	65.38169%	3.99419	6.62672	1	65.38169%
XGB	4.33662	8.32248	1.08573	55.5338%	4.07705	7.38249	1.02075	**57.95101%**
MLP	6.65449	12.41026	1.66604	149.592%	3.9821	6.43928	0.99697	77.20404%
SVR	10.2786	19.64989	2.57339	148.63564%	9.48528	16.63145	2.37477	213.85592%

The situation remains unchanged in the Augmented dataset: the Naïve agent is still the best option based on both its performance and simplicity. There is, however, a small improvement. Thanks to the additional *Previous Week* feature, the overall performance of the models had a slight increase, with the exception of the MAPE indices of both XGB and SVM agents. Once again, we may attribute this increase in MAPE to the measure's symmetry problem. The bottomline of this experiment is that the Naïve model is still the best choice, as even though XGB's MASE in both cases might indicate that its performance is not that different from Naïve's, the latter is still a simpler model to implement, and therefore more preferable.

After this experiment, we can see how it is no use to apply the same forecasting procedure that we did on the Bike Sharing Demand dataset on the Computer Retailer Demand dataset. Even

though the nature of the problem is the same, the same models have very distinct performances on the two datasets.

Predicting with the Forecasting Machine Architecture

In this next experiment we validate our Forecasting Machine Architecture. We divide the dataset into twelve subsets using the clustering technique described in Chapter 4. For each resulting subset, we assign a set of five agents, four of which are model agents, and one predictive agent. The predictive agent, aptly called the Best Method Agent (BMA), is the one who decides on the best prediction to output, based on the predictions made by the four model agents: Naïve, XGB, MLP and SVR. This strategy selects the model which best conforms to each of the data subsets, thus allowing us to optimise prediction accuracy.

First, we use the Base dataset as the source dataset for clustering. Like we did with the previous experiment, we would like to take note of the change in performance caused by the addition of the *Previous Week* feature. One of the results that call our attention in Table 5.3 is that the Average Mean Absolute Error is higher in the distributed case than it is in the unified case. At first, one might think that we have lost accuracy by dividing the dataset. There are two factors, however, that defuse this premature conclusion. First, if we look at the Mean Absolute Percentage Error, we can see that it is far lower than in the unified case. In the majority of clusters the MAPE sits around 18% and reaches a minimum of 11.2924971% in cluster 1 and a maximum of 37.45979724 in cluster 9. Note that even the highest MAPE value is still lower than in the unified case. The second factor explains the increase in the overall MAE. One of the very effects of clustering with *k-means* is that each resulting cluster has a different mean based on its members. Because of this, the MAE varies more between the clusters in the distributed case, bringing the average MAE index up, as opposed to the unified case, in which higher values are diluted in a majority of lower values.

What is interesting about this experiment is that we can see that for every cluster there is a Machine Learning method that is superior to Naïve forecasting, unlike what happened in the previous experiment. The average percentage of improvement stays around 30% in the MASE index, with a maximum improvement of 30.12843268% in cluster 0 and a minimum improvement of 19.39069811% in cluster 9. The agent with the greatest promise is XGB, with it being selected for nine of the twelve datasets, and next comes SVR, which is selected for the remaining three datasets.

Cluster 9 is an interesting case study, since it is the cluster with the worst performance. This happens because the clustering process does not eliminate the bad disposition of data in the original dataset. Rather, it mitigates this shortcoming. As we increase the number of clusters, we can separate the good training examples from the bad, but that means that at least one of the resulting clusters will perform worse than the others. This is a worthy trade-off, however, since the worst results we have are still better than the best result with the original dataset intact.

In our last experiment, we use the Augmented dataset as the source dataset for clustering. This time, however, we can notice in Table 5.4 that the results are slightly worse than in the Base case. The average MASE increases to 0.7674136183, which is a loss of approximately $4,468861515\%$

Table 5.3 – Prediction performance for each cluster when using the Best Method Agent without the *Previous Week* feature.

Cluster	MAPE	MAE	RMSE	Method	Naive MAE	MASE
0	18.2819226	0.5106976375	1.296563988	SVR	0.7309090909	0.6987156732
1	11.2924971	17.44785623	22.44837643	XGB	24.352	0.7164855548
2	18.55464256	9.010531034	11.4760615	XGB	11.484	0.7846160775
3	17.93265018	0.5152856541	1.317001772	SVR	0.7571428571	0.6805659583
4	18.2819226	0.5106976375	1.296563988	SVR	0.7309090909	0.6987156732
5	18.55464256	9.010531034	11.4760615	XGB	11.484	0.7846160775
6	17.10429058	11.93369444	15.16547205	XGB	17.056	0.699677207
7	24.16778801	7.075095684	9.216233562	XGB	9.0904	0.7783041103
8	17.10429058	11.93369444	15.16547205	XGB	17.056	0.699677207
9	37.45979724	6.219988025	8.326307244	XGB	7.716216216	0.8060930189
10	17.10429058	11.93369444	15.16547205	XGB	17.056	0.699677207
11	23.55519412	6.865198742	8.933492844	XGB	8.940363636	0.7678880883
Average	19.94949406	7.747247084	10.10692325		10.53782841	0.7345859878
Stdev	6.39712584	5.312690586	6.541307447		7.511827533	0.04534172924
CoV	0.320666069	0.6857520521	0.6472105591		0.7128439791	0.06172419566

in comparison to the Base case. What is interesting to note in these results is that, although there is a small increase in MAPE, its Coefficient of Variance indicates that it is more stable than before, while the remaining indices are more unstable and have gotten worse. The bottomline here is to stick with the simpler models, so the Base case wins this round.

Table 5.4 – Prediction performance for each cluster when using the Best Method Agent using the *Previous Week* feature.

Cluster	MAPE	MAE	RMSE	Method	Naive MAE	MASE
0	20.26735117	0.5193655162	1.270819679	SVR	0.7309090909	0.7105747112
1	13.33959618	21.22086435	27.64503223	XGB	24.352	0.8714218277
2	16.88746544	9.326387827	11.85380109	XGB	11.484	0.8121201521
3	19.40792729	0.5200639515	1.292023856	SVR	0.7571428571	0.686876917
4	20.26735117	0.5193655162	1.270819679	SVR	0.7309090909	0.7105747112
5	16.88746544	9.326387827	11.85380109	XGB	11.484	0.8121201521
6	17.76975287	12.55720192	16.12775902	XGB	17.056	0.736233696
7	25.66378421	7.361717976	9.617135191	XGB	9.0904	0.8098343281
8	17.76975287	12.55720192	16.12775902	XGB	17.056	0.736233696
9	32.92923765	6.069968025	8.114688376	XGB	7.716216216	0.7866508474
10	17.76975287	12.55720192	16.12775902	XGB	17.056	0.736233696
11	25.24361473	7.15308378	9.340353802	XGB	8.940363636	0.8000886844
Average	20.35025432	8.307400877	10.88681267		10.53782841	0.7674136183
Stdev	5.260653502	6.121490536	7.709460043		7.511827533	0.05552751091
CoV	0.2585055409	0.7368719322	0.7081466611		0.7128439791	0.07235669213

6. RELATED WORK

Lu (2014) [22] also studies the problem of forecasting computer related product sales. In his research, he seeks to find the best set of predictor variables to improve the forecasting model. First, he creates around 30 candidate predictors by using the dataset to generate new information, *e.g.* the demand in the previous year, the average of the past weeks, the variance of the past weeks, amongst others. He then uses the Multivariable Adaptive Regression Splines (MARS) tool to select the features that most benefit the prediction model's accuracy. Eventually, he uses these selected variables as the predictors for a Support Vector Regression (SVR) model to make the final predictions. From his work, we can see that the set of ideal predictors may vary according to the product and its sales history. With respect to their result, they are able to achieve MAPE indices ranging from $15,22\%$ to $24,22\%$. Their research does not consider, however, the locations from where demand originates and limits the quantity of products to five, making it more manageable to create separate models for each product.

Choi *et al.* (2014) [11] investigate sales forecasting in the fast fashion industry. Although their target industry is different from ours, it actually shares some of the characteristics of the associated dataset, such as the short span of products in the market. They do not, however, consider demand locations. In their work, they use a combination of forecasting models to predict sales data with limited data and time. They employ Extreme Learning Machines (ELM), which is a type of Neural Network used for regression, and Grey Systems Theory for forecasting time series. The authors separate the general forecasting task into two: forecasting the trend of the time series using Grey Models, and forecasting the residuals using Extended Extreme Learning Machines. Their insight is that the trend series has a different behaviour than that of the residual series, and so using distinct predictive models for each might improve the overall prediction accuracy by first improving the accuracy of these separate models. Still, with their approach, they are able to achieve MAPE indices ranging from $43,2\%$ to $50,7\%$.

Carbonneau *et al.* [5] use Machine Learning methods to solve the demand forecasting problem. Their contribution is a more analytical one, in which they compare more traditional types of forecasting, such as Naive Forecasting, Moving Average and Multiple Linear Regression, with more advanced ones. The advanced techniques they use are Artificial Neural Networks (ANNs), Recurrent Neural Networks (RNNs) and Support Vector Machines (SVMs). Overall, their results show that the advanced techniques have no significant improvement over multiple linear regression, with MASE indices of $0,77506$ for Neural Networks, $0,62447$ for Recurrent Neural Networks, $0,62855$ for SVR and $0,6565$ for Multiple Linear Regression. Still, they conclude that RNNs and SVMs provide better results than standard ANNs. In our work, we provide new evidence that ANNs perform similarly to Linear Regression, but in some cases we obtain a better result with SVR models.

7. CONCLUSION

In this dissertation, we have discussed the use of a modular architecture that combines agents and predictive models to solve the demand forecasting problem considering time, product and location information. The challenges of forecasting demand in this setting are (i) that demanding locations have different demand patterns for each available product, and (ii) as the possible combinations of locations and products grow, it becomes impractical to use a single model to predict all of them. In our particular application of this architecture we face yet a third challenge, which is the limited amount of information available.

Our main contribution is an architecture to aid in inventory management decisions regarding future demand. This architecture scans a source dataset for combinations of products and locations with similar demand patterns, and then breaks this dataset into smaller datasets. We can then use a group of agents per dataset to make predictions distributively and more accurately. Each of the agents can use different prediction models, and we can organise agents to communicate their results and find the best model for each of the data subsets. Our secondary contribution is the formal description of the demand forecasting problem considering these location and product constraints, as well as the costs involved in keeping items in stock *versus* the gain of dispatching products on time.

We have experimented with this architecture on a dataset with many locations and products, and a varying amount of training examples per combination of location and product. To complete these experiments we used a variety of Machine Learning algorithms as the core predictive mechanism of agents, including Gradient Tree Boosting [16, 17, 10], Multi-Layer Perceptron [18, 33] and Support Vector Regression [37, 14, 36, 1]. To compare our results, we use a classical forecasting technique of Naïve forecasting, in which we use the current observation as the prediction for the next time step. Through experimentation, we have observed that using Machine Learning methods directly on the computer retailer dataset is ineffective, as the patterns contained in the dataset are too varied for any of the attempted methods to generalise and replicate. On the other hand, running these methods on the Bike Demand dataset [15] yields more useful results, as this dataset has a higher ratio of training examples per feature, and it does not contain any information regarding location, thus being fundamentally simpler. We then validate our architecture by applying it on the Computer Retailer dataset and dividing the data into datasets with similar patterns. The results show that the use of our architecture provides better predictions than was possible with the original dataset.

There are, of course, limitations to our approach. Because of the clustering process, which isolates the existing combinations of locations and products, our architecture cannot predict demand for products that are not in the dataset. This is a disadvantage considering that the computer market moves fast, and companies frequently add new products to the market. The clustering process, although it does help improving predictions, has a caveat. Depending on the dataset, more specifically on the number of training examples and the quality of these examples, there will be at

least one cluster in which accurate prediction will be a hard goal to achieve. As we have reviewed, however, it is a caveat that eventually pays off.

Due to the multi-disciplinary nature of the work done in this dissertation, there are a few avenues for future work. From the agent and machine learning perspectives, we can improve agents by programming them to experiment not only with model parameters, but with different feature sets, since each dataset might benefit from a different set of features. Under the same perspective, we can combine agents with different predictive models to make an ensemble, or program an agent to train a new model using the predictions from its servants as the predictor variables for the target values. From a multi-agent perspective, we can improve the exchange of information between agents so that they consider not only the predictions that come from their models, but information from other sources as well, like other agents or even from the external world. Information such as special dates (inspired by Catal *et al.* [6]) or news from the sector for which we are forecasting demand (thus opening a path to Natural Language Processing research). We can improve the multi-agent system by adding multi-agent strategies such as elections (agents vote on the best prediction) or auctions (agents with higher accuracy can bid higher for them to be selected), to make the selection of the most appropriate prediction. From the perspective of demand forecasting, we can shift the focus of predicting products to predicting product categories instead. Since categories of products do not change as often as products do, shifting would allow us to overcome the limitation of not being able to predict unreleased products.

BIBLIOGRAPHY

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[2] Michael Bratman. *Intention, Plans, and Practical Reason*. Center for the Study of Language and Information, 1987.

[3] Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[4] Jose Caceres-Cruz, Pol Arias, Daniel Guimarans, Daniel Riera, and Angel A Juan. Rich Vehicle Routing Problem: Survey. *ACM Comput. Surv.*, 47(2):32:1—-32:28, dec 2014.

[5] Real Carbonneau, Kevin Laframboise, and Rustam Vahidov. Application of machine learning techniques for supply chain demand forecasting. *European Journal of Operational Research*, 184(3):1140 – 1154, 2008.

[6] Cagatay Catal, Ayse Fenerci, Burcak Ozdemir, and Onur Gulmez. Improvement of Demand Forecasting Models with Special Days. *Procedia Computer Science*, 59:262 – 267, 2015.

[7] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27, May 2011.

[8] Stephan Chang and Felipe Meneguzzi. Forecasting demand with limited information using Gradient Tree Boosting. In 30th International Florida Artificial Intelligence Research Society Conference (to appear), 2017.

[9] I-Fei Chen and Chi-Jie Lu. Sales forecasting by combining clustering and machine-learning techniques for computer retailing. *Neural Computing and Applications*, pages 1–15, 2016.

[10] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining*, KDD 2016, pages 785–794, New York, NY, USA, 2016. ACM.

[11] Tsan-Ming Choi, Chi-Leung Hui, Na Liu, Sau-Fun Ng, and Yong Yu. Fast fashion sales forecasting with limited data and time. *Decision Support Systems*, 59:84 – 92, 2014.

[12] G. B. Dantzing and J. H. Ramser. The Truck Dispatching Problem. *Management Science*, 6(1):80–91, 1959.

[13] Mehdi Dastani. 2APL: A Practical Agent Programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, June 2008.

[14] Harris Drucker, Christopher J. C. Burges, Linda Kaufman, Alex J. Smola, and Vladimir Vapnik. Support vector regression machines. In M. I. Jordan and T. Petsche, editors, *Advances in Neural Information Processing Systems 9*, pages 155–161. MIT Press, 1997.

[15] Hadi Fanaee-T and Joao Gama. Event labeling combining ensemble detectors and background knowledge. *Progress in Artificial Intelligence*, 2(2):113–127, 2014.

[16] Jerome H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics*, 29:1189–1232, 2001.

[17] Jerome H. Friedman. Stochastic Gradient Boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, February 2002.

[18] S.S. Haykin. *Neural Networks and Learning Machines*. Number v. 10 in Neural networks and learning machines. Prentice Hall, 2009.

[19] Rob J. Hyndman and Anne B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679 – 688, 2006.

[20] Nicholas R. Jennings and Michael J. Wooldridge. *Agent Technology: Foundations, Applications and Markets*. Springer-Verlag Berlin Heidelberg, Secaucus, NJ, USA, 1998.

[21] Nicholas R. Jennings and Michael J. Wooldridge. *Agent Technology: Foundations, Applications and Markets*, chapter Applications of Intelligent Agents, pages 3–28. Springer-Verlag Berlin Heidelberg, Secaucus, NJ, USA, 1998.

[22] Chi-Jie Lu. Sales forecasting of computer products based on variable selection scheme and support vector regression. *Neurocomputing*, 128:491 – 499, 2014.

[23] Chi-Jie Lu and Ling-Jing Kao. A clustering-based sales forecasting scheme by using extreme learning machine and ensembling linkage methods with applications to computer server. *Engineering Applications of Artificial Intelligence*, 55:231 – 238, 2016.

[24] Helmut Lütkepohl and Fang Xu. The role of the log transformation in forecasting economic variables. *Empirical Economics*, 42(3):619–638, 2012.

[25] Yungao Ma, Nengmin Wang, Ada Che, Yufei Huang, and Jinpeng Xu. The bullwhip effect on product orders and inventory: a perspective of demand forecasting techniques. *International Journal of Production Research*, 51(1):281–302, 2013.

[26] J B MacQueen. Some Methods for classification and Analysis of Multivariate Observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967.

[27] Spyros Makridakis. Accuracy measures: theoretical and practical concerns. *International Journal of Forecasting*, 9(4):527 – 529, 1993.

[28] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[29] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI Reasoning Engine. In Rafael H. Bordini, Mehdi Dastani, JÃ¼rgen Dix, and Amal El Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 149–174. Springer, 2005.

[30] Patrícia Ramos, Nicolau Santos, and Rui Rebelo. Performance of state space and ARIMA models for consumer retail sales forecasting. *Robotics and Computer-Integrated Manufacturing*, 34:151 – 163, 2015.

[31] Anand S. Rao. Agentspeak(l): BDI agents speak out in a logical computable language. In Walter Van de Velde and John W. Perram, editors, *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55, Eindhoven, The Netherlands, 1996. Springer.

[32] Anand S. Rao and Michael P. Georgeff. Modeling rational agents with a bdi-architecture. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, chapter Modeling Rational Agents with a BDI-architecture, pages 317–328. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[33] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 10 1986.

[34] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Pearson Education, 3rd edition, 2010.

[35] Yoav Shoham. Agent-oriented Programming. *Artificial Intelligence*, 60(1):51–92, March 1993.

[36] Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004.

[37] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[38] Sanford Weisberg. *Applied linear regression*. Wiley series in probability and mathematical statistics. Wiley, New York [u.a.], 2. ed edition, 1985.

[39] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.