University of Economics, Prague
Faculty of Informatics and Statistics
Department of Information Technologies

Master's degree program: Applied Informatics
Major: Information Systems and Technology

An Analysis of Potential Applications of Machine Learning in HTTP Load Balancing

DIPLOMA THESIS

Student : Bc. Jan Sýkora
Supervisor : Ing. Rudolf Pecinovský, CSc.
Opponent : Mgr. Zbyněk Šlajchrt, Ph.D.

2017

Declaration:

I declare that I have worked on this thesis independently using only the sources listed in the bibliography. All resources, sources and literature, which I have used in preparing or I drew on them, I quote in the thesis properly with stating the full reference to the source.

Prague, April 21, 2017 . . . .

Bc. Jan Sykora

Acknowledgment

I wish to express my sincere gratitude to Ing. Rudolf Pecinovský, CSc. for expert supervising of my diploma thesis and for granting me the opportunity to compose second thesis under his supervision. Also, I would like to thank my sister for her help with editing and proof-reading.

Abstract

Both machine learning and HTTP load balancing are well known and widely researched concepts and methods. My diploma thesis addresses possible applications of machine learning to HTTP load balancing. The main objective is to find a method to achieve better utilization of load balancing. This objective can reduce monetary costs and provide better stability of a load balanced system.

In the first part, machine learning workflow and methods are described in order to analyze whether such methods could be applied to load balancing systems. After that, the current state of HTTP load balancing methods and strategies is outlined. Finally, a load balancing method using machine learning is designed and tested. The method is based on the least loaded approach using predicted values to balance HTTP traffic, the machine learning models were selected by using a grid search to find the most accurate models. These methods were tested and performed well in comparison to other methods.

The tests were conducted with over a hundred machine learning models, not all models were accurate or had short enough learning times. Lacking those factors deemed them unsuitable for later tests. The models were compared based on measured utilization and performance metrics for regression based machine learning models.

The designed method could be applied to real world systems, however, it would require defining a domain specific metric. The applications should also employ a grid search in order to find the most accurate machine learning model.

Keywords

machine learning, load balancing, HTTP protocol

Abstrakt

Strojové učení i vyvažování zátěže jsou známá a již dobře prozkoumaná témata, z toho důvodu se tato diplomová práce se zaměřuje na možnosti aplikace strojového učení na vyvažování a distribuování HTTP protokolu. Hlavním cílem mé práce je nalézt metody pro zvýšení utilizace systémů distribuujících zátěž, dosáhnutím stanoveného cíle lze snížit monetární náklady a zvýšit stabilitu daného systému.

V první části se práce zaměřuje na metody a postupy v rámci strojového učení v kontextu případné aplikace na vyvažování. Následně jsou popsány současné přístupy k vyvažování zátěže v HTTP protokolu. V poslední části je navržena metoda vyvažování zátěže využívající modelu vytvořeného pomocí strojového učení. Metoda využívá „nejméně zatížený“ principu s predikovanými hodnotami. Modely byly nalezené pomocí prohledávání hrubou silou s cílem nalezení nejkvalitnějších modelů. Nalezené modely byly otestovány a dosahovaly dobrých výsledků

Testovalo se více než sto různých modelů a ne všechny dosahovaly dobrých výsledků a nebo měly příliš dlouhé doby učení. Tyto faktory je diskvalifikovaly z případného použití v dalších testech. Modely byly porovnány na základě změřené utilizace a metrik pro regresní modely strojového učení.

Navrženou metodu je možné aplikovat na reálný systém, ale vyžadovalo by to definovat doménově specifickou metriku. Aplikace by vyžadovala opakovat prohledávání hrubou silou za účelem nalezení nejlepšího modelu a navržení systému na základě vybraného modelu.

Klíčová slova

strojové učení, vyvažování zátěže, HTTP protokol

Contents

List of Figures

List of Tables

List of Code Samples

1 Introduction

According to Gartner's Hype Cycle (Gartner, 2016) from 2016, machine learning is at its peak and should be adopted as one of the mainstream technologies by most companies and enterprises within two to five years. The machine learning field has matured over the past few years and can provide first-rate quality tools and libraries. The vast majority of companies and enterprises focus on its possible applications and are looking for its use cases in process automation. A very notable use case is scoring model automation for loan systems, where banks and financial institutions lack the manpower to handle all the loan requests. In such cases, machine learning can be used to scale this process, increase accuracy and even decrease the default probability.

In almost every such case one invariably faces a hard to solve problem and lacks adequate means to deal with it. Such problems usually have a few things in common. One of them is that they handle large amounts of data and the other known issues are solving or output needs to be scaled in large. Last but not least is the requirement for consistent outputs. All these aspects make such tasks not feasible for humans to complete.

From most obvious use cases, machine learning now finds ways to solve more complex, general or abstract topics. The incentive for utilizing machine learning is quite clear: monetization, be it in the form of creating value or cutting down on costs. With this consideration in mind, machine learning can be applied to almost everything.

1.1 Problem and importance

Although machine learning is a well-known and widely researched area, many areas still remain poorly understood or misinterpreted. Generally, any research on machine learning can go two ways, the first one is more theoretical (focused on machine learning as such) and the second one is exploratory, trying to find practical ways to efficiently apply machine learning to a certain area. In this thesis, I have chosen to analyze whether machine learning can be applied to HTTP load balancing. There certainly are prior studies on the topic of applying machine learning to workload balancing, however they all focused on a specific domain or problem and none of them analyzed HTTP load balancing specifically. Queuing theory and load scheduling are well researched areas, but load balancing in computer science is too complex and it can be quite difficult to formalize the systems into mathematical models (queuing theory relies on mathematical models).

The reasons to research this area are quite simple, over the past decades load balancing has become quite a prominent topic in computer science and is recognized as a key concept of modern software engineering and system design. Some benefits of load balancing are hard to measure, but most of them can be quantified with metrics and compared to others. So it is quite straightforward to recognize which load balancing method is better in comparison. It was against this background that the idea of applying machine learning to load balancing was born, to create a load balancing method with better results (metric-wise).

The new load balancing method can achieve a better load distribution within a system and cut down on system maintenance costs. Better distribution can decrease the probability of a load peak on concrete service units. In the worst case scenario an uneven load can disrupt system stability and cause major incidents, which can be associated with a significant negative impact on business. The importance of load balancing can be further highlighted by the fact that load balancing is one of the key components of site reliability engineering.

HTTP load balancing contends with four main factors: small subsets of traffic/requests, varying traffic costs, machine diversity and unpredictable performance factors (Beyer, et al., 2016, pp. 241-243). The first one is straightforward - load balancing struggles to perform well when handling small traffic or a low number of requests. In practice, it is extremely rare for every request to carry the same request cost and even a small difference among requests can cause different request costs. Many data centers have different hardware for service units and using virtualization allows the unit to share hardware with other units. Every system using load balancing has to deal with unpredictable factors, such as sharing resources with other processes, restarting or terminating HTTP requests. Decreasing the influence of above stated factors could lead to substantial benefits for HTTP load balancing; therefore any methods that can achieve this are desirable.

1.2 Objective

The overall goal of this thesis is to analyze whether machine learning can be applied to HTTP load balancing. For the sake of a more systematic approach, four partial objectives were identified within the general goal. The first goal is to analyze the current state of machine learning and its methods and whether the tools it can provide at this point are able to sufficiently serve the purpose of applying them to HTTP load balancing. The second goal is to provide insight into the present condition of load balancing strategies/algorithms. The third objective is to design and implement a load balancing strategy supported by machine learning. The fourth goal is to test the designed strategy and write a manual for it. The general goal could also be characterized as aiming to minimize deviations from average balanced utilization of service units. The objectives are summarized in points below:

- Analyze machine learning in terms of ability to apply it to HTTP load balancing
- Analyze load balancing strategies
- Design and implement load balancing strategy via machine learning
- Test the designed strategy and write manual for it.

1.3 Approach

As stated in previous sections this thesis focuses primarily on machine learning and HTTP load balancing. Both terms cover fairly broad areas and exploring them fully would exceed the scope of this thesis. Therefore in this and the next chapter, both terms and approaches are defined and limited to be more specific and relevant for the purpose of this thesis.

All four objectives are addressed in this thesis. First two are more theoretical therefore the approach to them is theoretical as well. They provide meaningful insights into HTTP load balancing and machine learning. The research relevant to both of these objectives is conducted systematically and separately. When analyzing machine learning, emphasis is placed on workflow, methods and scaling (learning and predicting). It should offer sufficient knowledge to understand key concepts of machine learning, enabling readers to apply this newly gained information to real-world problems. The load balancing part covers performance metrics, utilization and methods. Readers should be able to refer to this thesis, apply the concepts within a selected system and also conduct comparisons when choosing the fitting method.

For the last two objectives five basic machine learning models were selected (ordinary least-squared regression, lasso regression, ridge regression, stochastic gradient descent and regression tree) and those were trained on a dataset containing HTTP traffic. Same dataset was used to measure performance of models and compared. The experiments had two rounds.

The first round consisted of a grid search for best performing model parameters; models were trained and tested on a subset of the main dataset. The second consisted of selecting the best performing models and comparing them against round robin and least loaded methods, with utilization as the key metric.

1.4 Constraints

In order to fulfill the objectives of this thesis, several constraints are posed on the topic. The two main reasons are as follows: the first one is to narrow down the area of research, because both main topics are quite broad and the second one is to make those redeemable. As with many problems this topic is very domain specific and it would be rather difficult to find solutions that fit all the cases of HTTP load balancing.

By default load balancing is described as distributing workload across several units. In computer science and networking, load balancing can be separated into two groups, layer 4 (network) and layer 7 (application). The layers are defined in the OSI Reference Model (Table 1). This research focuses only on layer 7 methods and methods such as DNS load balancing are excluded and only load balancing on application layer is researched.

Most of the systems communicating via HTTP protocol are open systems and only some of them are closed (using HTTP only for internal communication). I decided to make the assumption that HTTP traffic is a finite and closed set, which makes it easier to reason about it and to create machine learning models

Table 1 OSI Reference Model (Reference (Bourke, 2001, pp. 13-15))

Reference	**Layer**	**Function**
7	Application	Deals with application programs using the network
6	Presentation	Defines the way of data presentation to applications
5	Session	Manages sessions and connections between hosts
4	Transport	Handles transportation protocol and error handling
3	Network	Manages network connections and related issues
2	Data Link	Offers data delivery between physical connections
1	Physical	Defines the physical networks

For designing and testing machine learning backed load balancing, I limited the research to five basic algorithms (least-squared regression, lasso regression, ridge regression, stochastic gradient descent and regression tree). These five algorithms are a representative sample of accessible ones; the algorithms were limited to ones used for supervised learning.

When testing the machine learning backed load balancing, the utilization of services units was used as a metric. In order to compare them standard deviation was used. All tests use at least three service units to conduct test, in the real world no system should rely on two or less units. When having two units, losing one means 50% decrease in performance with only one unit left to handle all the traffic.

The constraints of this diploma thesis are summarized below:

- Key metric of load balancing is utilization of service units and utilization is domain specific metric.
- Load balancing is limited to HTTP traffic and requests, which can be described as a closed set.
- HTTP load balancing is limited to Layer 7 of OSI Reference Model and to server side.
- Only models for supervised learning are used.
- Selected algorithms for tested machine learning models are least-squared regression, lasso regression, ridge regression, stochastic gradient descent and regression tree.
- When evaluating performance of tested machine learning models utilization is used with least loaded load balancing method.
- For tests minimal amount of service units is 3 units.

2 Literature Review

The subject of this chapter is to present a literature review on the topic of machine learning applied to HTTP load balancing. The review was performed on 25th of March 2017 and surrounding days. While researching relevant sources, no publicly available literature was found on this specific topic. Therefore the review scope was extended to include machine learning and load balancing separately. This approach is more suitable because it provides systematic reviews of both areas. Load balancing in general is not covered by many sources; it is more common to cover a specific problem, algorithm or approach. On the other hand machine learning is a topic widely covered in literature and many sources focus on this general topic and also on more specific ones.

2.1 Load Balancing

The book named Server Load Balancing by Tony Bourke (Bourke, 2001) explains most common concepts and practices in load balancing. The book is divided into three main parts. The first part covers concepts and theory; it introduces the reader to the idea of load balancing, its terminology, the anatomy of a load balancer and also covers main metrics. The second part is about system design of a load balancing solution and should afford readers the basic skills to design and craft one by themselves. The third and last part is focused on specific solutions, their setup and configurations. The book covers load balancing without domain specific details and focuses more on general network load balancing. Only a subset of topics is applicable to HTTP load balancing.

The second book is by several authors, Peter Membrey, Eelco Plugge and David Hows. Its title is Practical Load Balancing with the subtitle Ride the Performance Tiger (Little, et al., 2012). This book includes a more domain specific approach to load balancing. Again the book is divided into three sections. The first one once again introduces basic concepts of load balancing and also caching. The second part is brimming with hands-on examples of fundamental solutions. The last part explains more advanced topics, such as high availability, clustering and IPv6. Throughout the book the utmost importance of a domain specific approach is stressed when dealing with load balancing.

The diploma thesis by Michal Nidl, Methodology of optimal usage of load balancing in data center environment (Nidl, 2016) incorporates information from the two books mentioned above and extends them by providing methodology for designing optimal load balancing within the data center based on a case study done by the author of the thesis.

There is also a study conducted by Petra Berenbrik, Martin Hoefer and Thomas Hoefer. These three authors researched the area of selfish agents in network load balancing. The study is called Distributed Selfish Load Balancing on Networks (Berenbrik, et al.,2014). It designs the protocol for balancing network load with selfish agents and also covers convergence to Nash equilibrium when applying to the problem.

2.2 Machine Learning

Machine Learning, The Art and Science of Algorithms that Make Sense of Data by Peter Flach (Flach, 2012) is a notable and highly recommended book on machine learning. Several educational institutions use it as a textbook in their classes on machine learning. It provides a comprehensive introduction into machine learning and data science. It covers all the main areas and gives detailed examples followed by the theory supporting them. The book does not contain any implementation specific details and serves as an entry point to the world of machine learning.

Machine Learning in Action by Peter Harrington (Harrington, 2012) is a clearly written book for developers, that takes you straight into techniques and examples. It focuses on the day-to-day usage of machine learning in the real world. It investigates five main areas: classification, forecasting, recommendations, summarization and simplification. All of the examples are written in Python and beside NumPy library do not require any other external dependency. The book shows basic implementations of main algorithms such as Apriori or Adaboost.

Real-World Machine Learning by Henrik Brink, Joseph W. Richards and Mark Fetherolf (Brink, et al., 2016) is a book that shows concepts and techniques of machine learning on real world problems and focuses more on a domain specific approach. The book covers the full life cycle of machine learning, from data acquisition to result application. After reading this book readers should be able to build, deploy and maintain their own machine learning systems. Throughout the book the authors repeatedly points out one should not attempt to reinvent the wheel when it comes to machine learning.

Large Scale Machine Learning with Python by Bastiaan Sjardin, Luca Massaron and Alberto Boschetti (Sjardin, et al., 2016) is a book following the rise of big data and the increasing demand for computational and algorithmic efficiency. It covers a wide range of topics from learning and prediction scalability to high predictive accuracy. It contains examples in Python with dozens of different libraries such as Pandas, H20, NumPy or SciPy. In the last part it provides an introduction into MapReduce framework used for machine learning such as Apache Hadoop and Apache Spark.

Learning Real-time Processing with Spark Streaming by Sumit Gupta (Gupta, 2015) is a book presenting key concepts and basics of Spark streaming and by introducing examples allows readers to learn how to build, deploy and maintain Spark streaming application. It covers the architecture of Spark and related technology such as Apache Kafka, Apache Yarn and Apache Mesos. It brings the concept of real-time processing into machine learning.

Closest to the topic of this diploma thesis comes the thesis by Prajer Richard, titled On possible approaches to detecting robotic activity of botnets (Prajer, 2016). It covers the machine learning approach of discovering botnets in DNS, HTTP and IRC communication. The thesis focuses on discovering botnets postmortem, after the fact.

3 Methods

This chapter fulfills two objectives, analyzing machine learning and HTTP load balancing. It contains an introduction to key concepts of both areas, which are required in order to be later applied in experiment. The both methods are covered separately.

3.1 Machine learning

The approach to machine learning bears great resemblance to, for example, linear programming. Typically, linear programming workflow consist of five phases: economical modeling, mathematical modeling, optimization, interpreting results and applying results (Lagová, 2014, p. 9). Most of the methods are iterative and results are fed back into modeling phases in order to increase model quality. What happens in economical modeling is that the real world system is portrayed and simplified in order to capture it as a whole. Then in the next step - mathematical modeling - the economical model is simply formalized so that is can be understood by anyone or by a computer.

When you look at machine learning, the approaches are analogous. The crucial distinction lies in that machine learning deals with complex systems, which cannot be simplified without losing their value. It is beyond human ability to produce a representation of a system with such a large number of details, processes, relationships and elements. Most of the problems that utilize machine learning can be solved by people, but they are usually not fast enough or not equipped to handle such large volumes.

The machine learning model consists of four basic components, which are commonly shared by all of them, namely: target, model or signal, explanatory features and noise or error (Brink, et al., 2016, pp. 53-55). The model can be formalized as shown in Figure 1. The target Y is an estimated or predicted variable, unknown for future data. The model or signal f is an unknown function, which is used for estimating or predicting the target. The explanatory features x are a set of features, that describe input data and error or noise ε are values of the target variable, which cannot be described by explanatory features.

$$Y = f(x) + \varepsilon$$

Figure 1 The formalized concept of machine learning problem

According to the formalized model, machine learning can be simplified to efficiently finding function f_{est}, which is optimal (Brink, et al., 2016, p. 55). Assuming the function is found, it can be used for two purposes, prediction or inference. In case of prediction the relationship between prediction Y_{pred}, function f_{est} and the new data x_{new} can be formalized as well (Figure 2).

$$Y_{pred} = f_{est}(x_{new})$$

Figure 2 The formalized concept of prediction process in machine learning

The prediction can be in the form of a variable, probability or class/cluster belonging. This can be quite useful in practical applications. If it adopts the form of inference, it can be used to better comprehend the system, model or features. Inference plays a big role in unsupervised learning and feature engineering, both of these terms are addressed in the next parts of this chapter.

3.1.1 Workflow

Machine learning workflow can be split into two main phases: modeling and predicting (Figure 3), with the first phase also containing a learning element. Both phases are iterative. The whole workflow encompasses all lifecycle parts, from data to deployment.

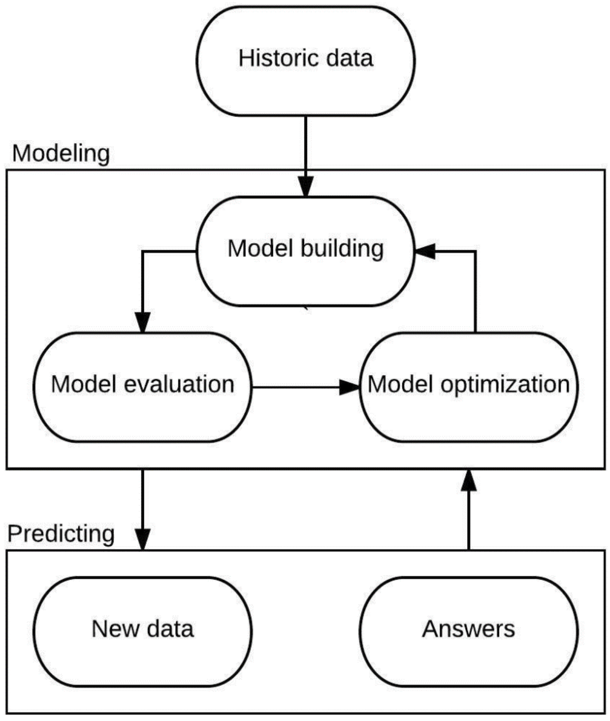

Figure 3 The basic workflow diagram in machine learning (Reference (Brink, et al., 2016, p. 17) edited)

The modeling phase consists of three processes: model building, model evaluation and model optimization. Model building is comprised of data preparation, transformation and learning a model from data, at the end of which this process produces a working model, which can be used for predicting. Following the building phase, the model is required to go through a rigorous evaluation for performance and accuracy. Based on these outcomes the model can be properly optimized and rebuild. Subsequently the cycle can be repeated numerous times. When model's accuracy and performance are assessed as sufficient and satisfactory it can continue to the predicting phase. Predicting uses new data to create answers (predictions), later on new data should be added to historical data. Feeding new data into historical ones facilitates building an even better model by returning to the modeling phase.

Historic data

Historic data are inputs into any machine learning algorithm and are necessary for a model to learn any insights into the problem. In the real world, there are many data sets available that can be used for learning. Collecting the data, which carries a hidden value inside, might, however, prove to be difficult. Since data structure frequently varies over time or merely a broken monitoring device can skew data and produce incorrect models, it is essential to be cognizant of the fact that machine learning workflow starts with data collection.

3.1.2 Model building

Model building is a process where a fully working machine learning model is built and in theory could be used for predicting without any iterations. Model building basically consists of three steps, algorithm selection, data preparation and learning. In practice, the second step is used twice, once before algorithm selection and once after. This happens due to differences among various algorithms, which might cause them to require different forms of inputs.

Algorithm selection

There is a wide range of algorithms with considerable variety among them available to us which can lead to serious difficulty when it comes to choosing the right one. Algorithm selection should ideally be based on several aspects. First and foremost one needs to consider what kind of prediction it is, whether clustering, variable or probability. The overwhelming majority of algorithms can only be used for one kind, unless some modification is made. The second aspect to acknowledge is whether a linear or nonlinear algorithm should be used. Despite their greater accuracy, nonlinear algorithms are significantly more demanding in terms of time and memory requirements. With both kinds there is the issue of fitting, whether underfitting or overfitting. In practice, an algorithm can underfit and not capture the nonlinear relationship between explanatory features and the target. On the other hand, the algorithm can also overfit and capture the nonlinear relationships only existing within the training data set. Examples of underfitting and overfitting can be seen in the Figure 4. The left side shows an example case of underfitting, where the model fails to recognize the nonlinear relationship. In the middle, there is a case of a well-trained model and on the right there is a clear case of overfitting. The algorithm used in Figure 4 is linear regression.

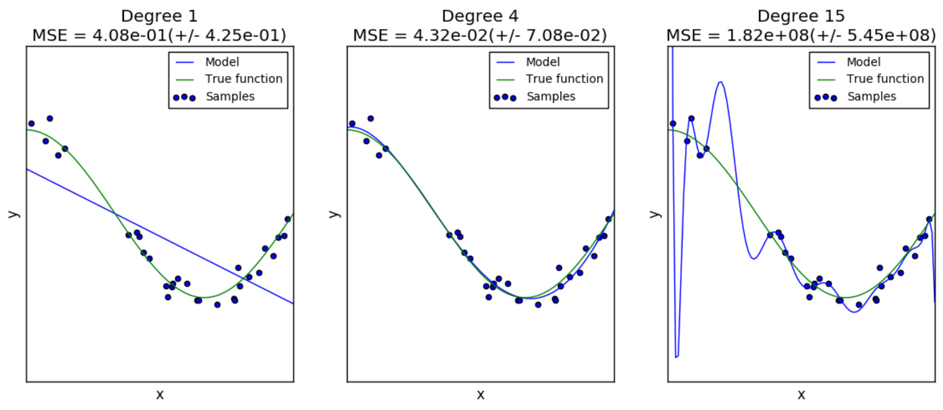

Figure 4 The graphs showing problems of underfitting and overfitting (Reference (Scikit-learn, 2016))

The third aspect is parametrization, algorithms fall into two groups: parametric and nonparametric. Algorithms in the first group, parametric, make assumptions and therefore are usually much simpler and easier to interpret. Those in the second group, nonparametric, do not make assumptions and their complexity is based on data and parameters, in consequence they are considerably more difficult to interpret. The complexity can be controlled by fine-tuning parameters. An example of a parametric algorithm is linear regression and an example of a nonparametric one is Support Vector Machine with kernel. The general overview of the most notable algorithms can be found in Table 2. The limits of this thesis make it unfeasible to delve deeper into this matter.

Table 2 The list of most well-known machine learning algorithms (Reference (Brink, et al., 2016, pp. 232-235) edited)

Name	Type	Linearity
Linear regression	Regression	Linear
Logistic regression	Classification	Linear
Support vector machine	Classification/regression	Linear
Support vector machine with kernel	Classification/regression	Nonlinear
K-nearest neighbors	Classification/regression	Nonlinear
Decision trees	Classification/regression	Nonlinear
Random forest	Classification/regression	Nonlinear
Boosting	Classification/regression	Nonlinear
Naive Bayes	Classification	Nonlinear
Neural networks	Classification/regression	Nonlinear
Vowpal Wabbit	Classification/regression	Nonlinear
XGBoost	Classification/regression	Nonlinear

Ordinary least-squares regression

This algorithm was introduced by Carl Friedrich Gauss in the late eighteenth century (Flach, 2012, p. 196) and the main idea is to learn a function estimator in a regression problem. Assuming there is $n\ x\ m$ (dimensions) matrix A, vector b and vector y with residuals ε (the differences between the actual and estimated values), the regression problem can be formalized as shown in Figure 5. As utility function sum of residuals is used with the objective to minimize it. Generally speaking, this is a basic linear regression.

$$y = A * b + \varepsilon$$

$$\sum_{i=1}^{n} \varepsilon_i \rightarrow min$$

Figure 5 The least-squares method

The Ordinary least-squared method extends the previous one with the assumption that actual values contain random noise. Without the noises or residuals the equation can be rewritten in such a form, so that coefficients of vector b can be calculated easily (Figure 6).

$$b = (A^T * A)^{-1} * A^T * y$$

Figure 6 Transformed equation of modified least-squared method

This representation (Figure 6) of the regression problem assumes that the matrix is “skinny”, meaning there are much more rows than columns ($m > n$). As far as machine learning use cases are concerned, this should always be true because there are usually more records - rows than describing parameters - columns. The “wide” matrix ($n > m$) can be an issue because the determinant is required to compute an inverse matrix and a “wide” matrix tends to have linearly independent rows and therefore the determinant is equal or close to zero. This problem has many workarounds, one of which is to use a pseudo-inverse matrix (Penrose, 1955).

Linear regression normally works with quantitative variables, both continuous and discrete. But by default cannot process quantitative variables without any transformations or pre-processing.

Ridge regression

When dealing with a “wide” matrix, there are several options available. One of them is a method called Ridge regression. In the previous chapter, I have introduced the transformed equation of the least-squared method (Figure 6). Ridge regression expands this equation with $\lambda * I$ by adding it to $A^T * A$. This forces it to be nonsingular, which as a result makes it possible to compute an inverse matrix to it. The λ is a scalar defined by a user or a program and is used as a ceiling for sums of values in vector b, the adjusted equation is in Figure 7.

$$b = (A^T * A + \lambda * I)^{-1} * A^T * y$$

$$\sum_{i=1}^{n} b_i^2 \leq \lambda$$

Figure 7 Equation of ridge regression method

The value of λ is found based on a grid search, with prediction error as a criterion (minimizing). In practice the grid search starts with a high number and slowly reduces it (shrinkage). This method can introduce an unpredictable bias into the dataset (Harrington, 2012, pp. 167-170). Ridge regression facilitates regularization and stabilization of the regression.

Lasso regression

Another available alternative is a method called Lasso regression. This method also expands on the least-squared method just as Ridge regression. It changes the new constraint (Figure 7) to the equation on Figure 8. The constraint makes the regression even more dependent on value in scalar λ. Because of this Lasso regression favours sparse solutions, with a lower number of used features.

$$\sum_{i=1}^{n} |b_i| \leq \lambda$$

Figure 8 Lasso regression method constraint

In practice, the lasso regression requires a quadratic solver and such a computation can become severely time-consuming. In many cases this method is replaced with a forward stagewise regression, which brings similar results (Harrington, 2012, p. 164).

Regression tree

Tree based models are very popular in machine learning because they are comprehensive, easy to learn and interpret. They can be applied to a wide range of problems. These models can be used for basically any type of problem: classification, ranking, probability estimation, regression and clustering. Probably the best-known example is Microsoft Kinect.

The model employs the basic concept of divide and conquer (Flach, 2012, p. 132). As a key concept it uses a feature tree, consisting of nodes and relations, just as every other tree or graph. Every node is either a split or a label. Label nodes are leaves of tree and represent target values. Split nodes control the flow of recalling target values based on one or more features and can have two or more children nodes. There are several algorithms for building feature trees (CART, ID3, C4, C4.5, C5).

Feature trees are normally binary trees but in certain situations the split nodes can have more children. When a split is done on a feature, it is usually removed from the feature variable. Every algorithm has a method for picking a feature to make a split on, the easiest method is to use information entropy.

Stochastic gradient descent

Stochastic gradient descent is a more advanced algorithm that has many niche usages and can be very useful when handling a large amount of data. This algorithm does not use linear regression to predict the target, and therefore computational complexity is not an issue; it is in order $O(n * p)$. The algorithm is popular with streaming processing and it is frequently used in Apache Spark and Apache Kafka based systems.

It has cost function J and the goal is to minimize its result, which leads to finding coefficients of function f. The main problem with cost function is that it uses a gradient to find a minimum and has no means to distinguish a local minimum from a global minimum, consequently learning can finish with suboptimal results. Such a cost function is formalized in Figure 9.

$$J(b) = \frac{1}{2n}\sum_{i=1}^{n}(A * b - y_i)^2$$

Figure 9 Gradient cost function

Data preparation

Data preparation is among the key concepts of model building, the data needs to be transformed into a format that can be used to fit the model. Formats across the algorithms have similar traits and only some of them need special transformations. Most of the algorithms require a matrix as input, this means that every explanatory feature needs to be described with a number. Features can be split into two groups: quantitative and categorical, and it is necessary to transform all categorical features to quantitative. This can be done by employing indicator variables. Given variable x, that has its values from set X, we can transform each value of set X to binary variables x_i, where i is from zero to size of set X minus one. This implies that set X is closed (Table 3).

Table 3 The formalization of transformation of categorical variables

Categorical variable	**Indicator variables**
$x \in X$	$x_i = \{0,1\}$, where $i = 0, \dots, (n-1)$ $n \dots$ number of values of set X

This transformation is immensely useful and well used throughout machine learning, but it can, in fact, cause issues when scaling. That is due to the increasing number of features (columns in matrix) when used. Solving this issue requires either a specialized algorithm or advanced feature engineering.

Another common concern is data with missing values and there are several methods to manage it. The most common one starts by posing the question whether missing data has meaning. If the answer is yes and the feature is categorical, missing values are replaced with a new category (‘missing’ or ‘other’), if the feature is quantitative the value is replaced with minus one or minus large number (substituting minus infinity). The decision whether to use minus one or something else, it depends on the values being normalized or not. If missing values do not have any meaning, the value can be replaced with mean average, preceding number or a row is removed from the data.

The last typical issue is data normalization. Since historical data come with different units or metrics and those can introduce a bias into the model, most models require data normalization. Most commonly normalized values have values between zero and one or minus one and one. The algorithm for basic normalization is in Code sample 1.

Code sample 1 Normalization in pseudo code

```
data = [...]
factor_min = -1
factor_max = 1
data_min = min(data)
data_max = max(data)
factor = (factor_max - factor_min) / (data_max - data_min)
for d in data
      normalized_d = factor_min + (d - data_min) * factor
```

Learning

The concept of learning by a machine can occasionally be misunderstood. For the purpose of this thesis, I choose the definition by Tom Mitchell (Mitchell, 1997, p. 2), “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.” For my usage of machine learning, the definition can be simplified to “A computer program can have insights into past events and generalize on them when coping with new events.”

In 2014, a competition was held for finding the best algorithm for the recognition of dogs and cats (Kaggle, 2014). The best models have over 98% accuracy when deciding whether there was a dog or a cat, the top one close to 99%. The majority of algorithms were taught by showing them labeled examples of cats and dogs and based on that they were able to make a guess. Such a concept of learning is also used with children, most children learn the difference between cats and dogs by examples, not by being told that most cats have pointier ears and smaller tongues. The concepts of supervised and unsupervised learning will be addressed in later sections of this chapter

In terms of formalizing what is learning within the machine learning concept, it is finding an estimated function f_{est} in Figure 2. Most algorithms/models are using regressors or trees as an estimated function, generally the model is fitted with historical data, therefore gaining insights into the area of the problem. The processing of learning can be very slow and needs scaling for real-world application.

3.1.3 Model evaluation

Model evaluation is a process of measuring model’s quality, accuracy and performance. In most cases, a model can be evaluated several times before being put into production. The iterative evaluation increases model’s quality and accuracy and also sets expectations for results in production. Applying the “no free lunch” theorem (Wolpert, et al., 1997) that states that when an algorithm performs well on a problem, it must certainly mean that performance on another problems must degrade. This may also apply between historical and new data, therefore it is really important to evaluate the model. In some cases it is impossible evaluate performance of models, it is either impossible to compute, too expensive to do so, or even if done it is too late to have a model that is significant for production use.

There is a general approach for model evaluation, called cross validation. The main idea is to use historical data for both learning and predicting and capture metrics to measure quality and accuracy of the model. The idea is simple but there are two main methods for this. I will introduce them in next two subchapters and then I will focus on metrics that can be used for evaluation.

Holdout validation

In the holdout method the historical data is split into two groups (Figure 10). Usually the second group consists of 30-70% of historical data. This method is very easy and should be used first when model accuracy seems too optimistic. The process involves four steps: data splitting, learning, predicting and comparing with real targets. The amount of data in the second group should be chosen randomly, but in most cases it should be around 30-40%, but it depends on the total amount of data.

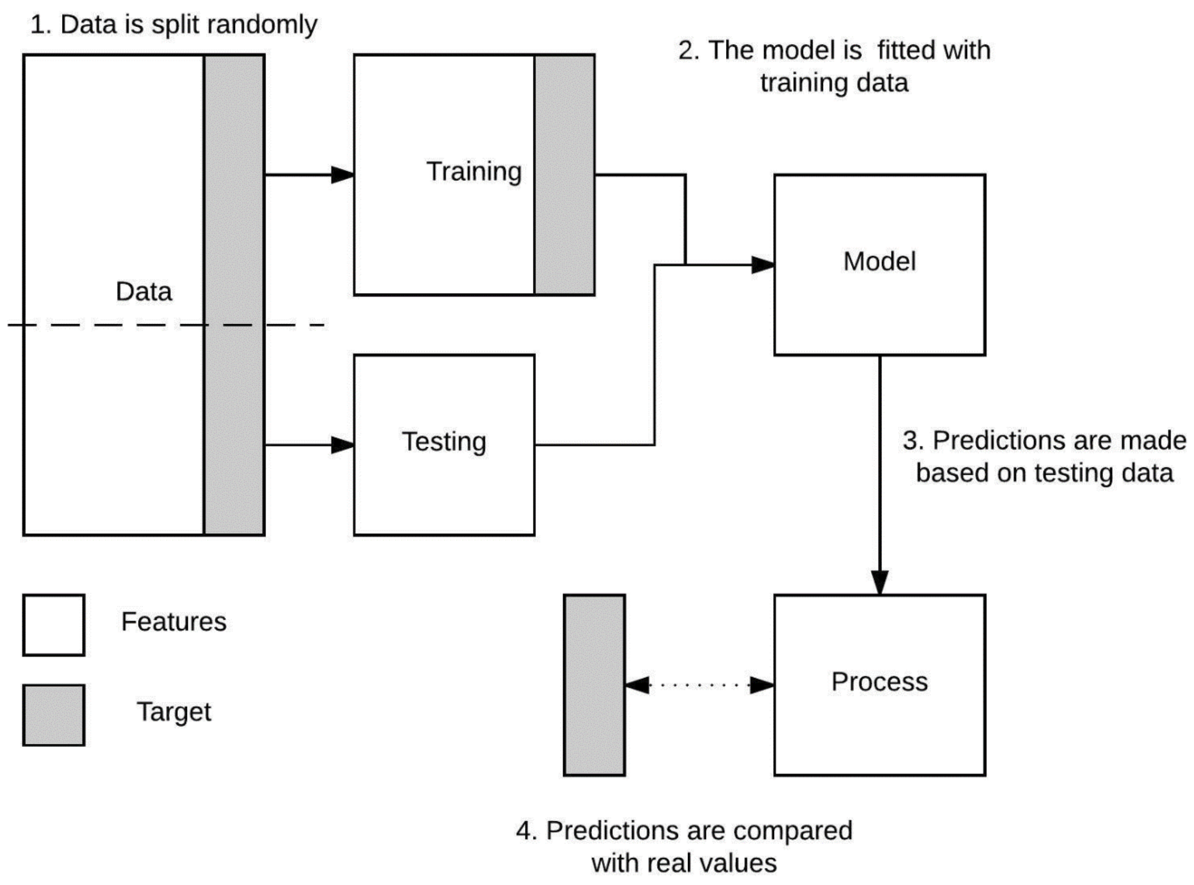
1. Data is split randomly
Data
Training
Testing
2. The model is fitted with training data
Model
3. Predictions are made based on testing data
Process
4. Predictions are compared with real values
Features
Target

Figure 10 The holdout validation method (Reference (Brink, et al., 2016, p. 83) edited)

K-fold cross validation

The K-fold method is a further extension of the holdout method, but is a fair bit more complex and much more computationally demanding. The advantage is that this method can be quite easily scaled by distributing to several computers. This validation can be parametrized by K, the amount of disjoint subsets that this validation uses. As in the holdout method, it incorporates; learning, predicting and comparing with real targets (Figure 11). But this happens K-times and is much more accurate. Therefore changing the K variable makes it possible to adjust accuracy of validation and computational demands. The K-fold method tests accuracy K-times, based on K models (each model is fitted with training data with excluded fold K), then accuracy of each model is either averaged or another statistical approach is used (percentile, median).

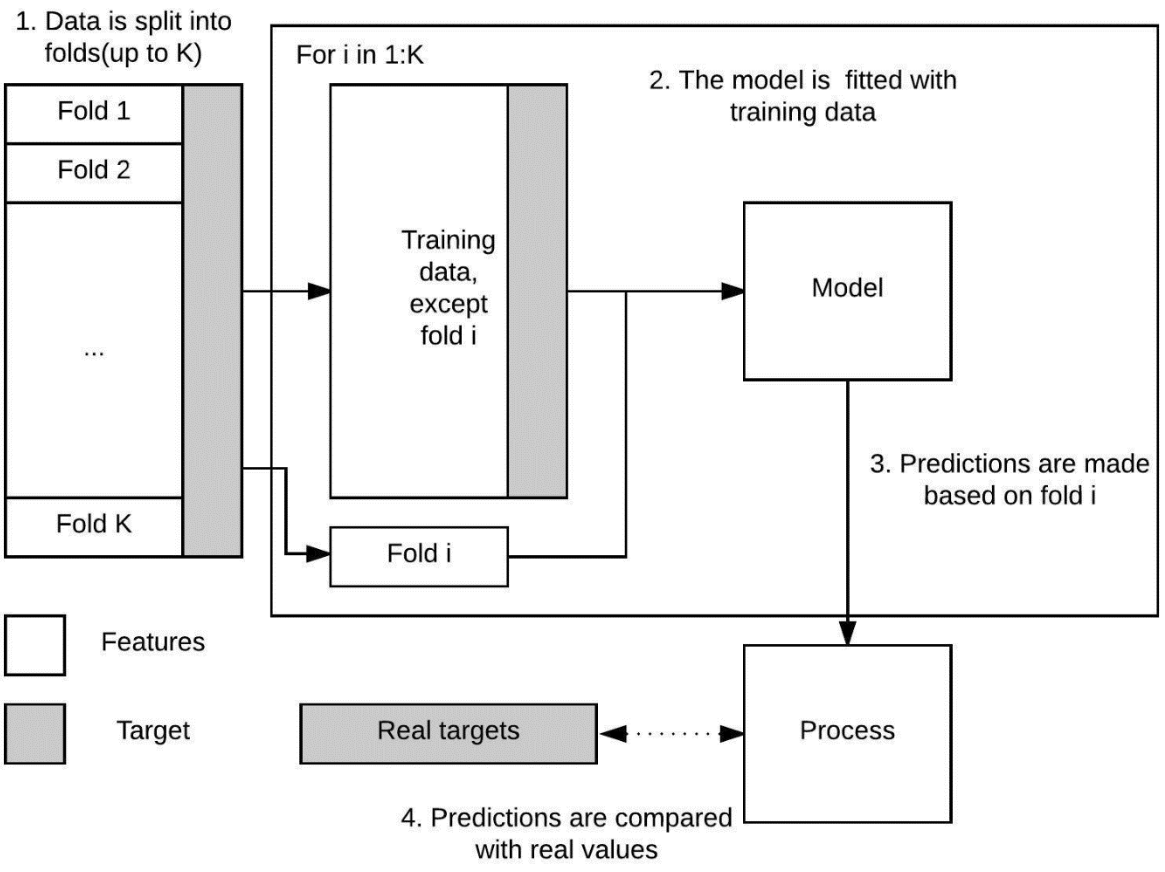
1. Data is split into folds(up to K)
Fold 1
Fold 2
...
Fold K
For i in 1:K
2. The model is fitted with training data
Training data, except fold i
Model
3. Predictions are made based on fold i
Fold i
Features
Target
Real targets
Process
4. Predictions are compared with real values

Figure 11The K-fold validation method (Reference (Brink, et al., 2016, p. 85) edited)

Confusion matrix

All machine learning models need a metric for measuring and comparing its performance or accuracy. To simplify it, only binary classification is taken into account, because otherwise metrics become excessively complex and it can prove difficult to interpret them. When comparing the real target and the predicted target (Figure 12), there are four basic scenarios which can occur, true prediction and true real target, false prediction and true real target, false prediction and true real target and false prediction and false real target. Each of these scenarios is also described by rate at which this scenario occurs. These scenarios are formalized by the confusion matrix.

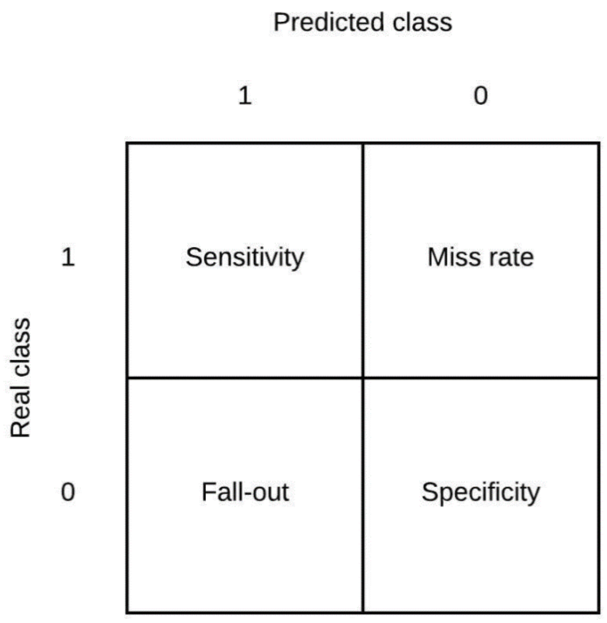

Figure 12 The confusion matrix

The rates in the confusion matrix are also called probability vectors or class probabilities and can be used to calculate receiver operating characteristics (ROCs), which is a true metric of model accuracy and can be compared between the models by counting area under the ROC curve (Figure 13). When following ROC curve comparisons, the sensitivity rate improves, but also sacrifices several correctly classified instances (specificity).

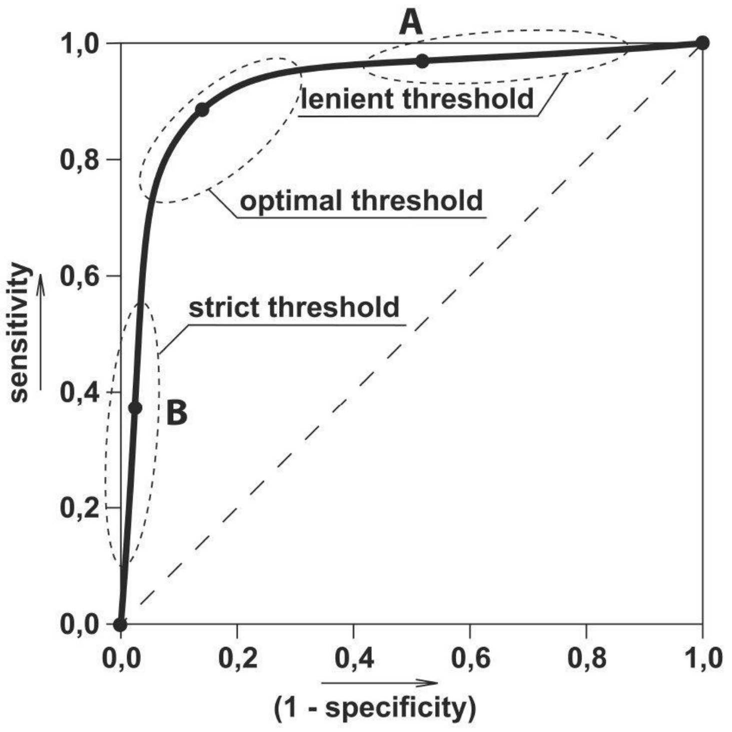

Figure 13 The area under ROC curve (Reference (Vránová, 2009))

Regression performance metrics

Compared to classification, regression prediction is not necessarily right or wrong, instead its accuracy is measured as the distance from the correct/real number. The distance can be called error or deviation and when calculated, the absolute value of differentiation between the real number and the predicted number is used. There are two main metrics for regression based machine learning. The first one is root-mean-square error (RMSE) and the second one is R-squared value.

The values of RMSE are in the same units as prediction, therefore need to be normalized when conducting comparisons with some other models. However on the other hand, they are much easier to interpret. The equation for computing RMSE is in Figure 14.

$$RMSE = \frac{1}{\sqrt{n}} * \sqrt{\sum_{i=1}^{n} [y_i - f(x_i)]^2}$$

Figure 14 RMSE equation

The R-squared value represents the portion of target value that can be explained by features. Byproduct of this metric is an indication of how much error or noise the historical data contains. The equation for calculating R-squared is in Figure 15.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - f(x_i))^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Figure 15 R-squared equation

There are a few more advanced metrics, that can be used, such as Akaike information criterion and Bayesian information criterion. When evaluating a model, it can prove quite useful to graphically display residuals for better understanding of the model and data.

3.1.4 Model optimization

Model optimization follows the model building and evaluating processes, when the model is ready and the performance metrics of this model are available. Based on those, there are several techniques to either increase accuracy, speed up the learning or the predicting. There might even be some cases which can lead up to collecting more data, if possible. The new collecting might be focused on getting new features or collecting more precise data. As stated many times before, machine learning is an iterative process and it is never a bad idea to come back to previous steps or processes.

When optimizing a model it is essential to gain a better understanding of the data and the model itself. The best way to achieve this is probably by using some graphical tools to visualize relationships, distributions and statistics on selected features and targets. When visualizing relationship between features and targets, there are four basic visual tools available for this purpose. These are mosaic plots, box plots, density plot and scatter plots. Rules when to use each one of them are depicted in Table 4.

Table 4 Rules for visualizing relations

Target \ Feature	**Categorical**	**Numerical**
Categorical	Mosaic plot	Box plot
Numerical	Density plot	Scatter plot

Feature engineering

Feature engineering is a set advanced technique for further transformations of features to attain better model accuracy. I am not going to cover all the numerous areas of feature engineering not only because of their large amount but also because they are in many cases domain specific (and require an expert on this subject matter to do them). One of the basic ones is transforming time features or series into a set of features, which represent days of the week, days of the month and similar features. These features can be useful for example, for event recommendations.

Other advanced techniques are associated with natural text processing, such as document indexing, bag of words and word to vector algorithms. I am not focusing on this area of expertise in this thesis, but there is a large community and many resources for this.

New features can also be acquired by combining other features together or using root or square values of a single feature to either increase or decrease the power of relational to target value. During this phase some values can even be removed if a flaw or a bias is discovered. A relevant example of such a bias is payment type (cash or credit) in datasets from New York taxis. Taxi drivers would not enter their tip when the payment was done in cash, consequently all trips paid by cash were biased when predicting the probability of the driver receiving a tip (Brink, et al., 2016, pp. 129-136).

Applying more advanced transformations can lead to undesirable side effects on performance and scalability. Implementing these transformations requires deeper understanding and more in-depth knowledge of computer science, which is something you cannot expect from ordinary users, who frequently have to implement these transformations themselves. A particular concern is not to parallelize the transformation and allocate too many object. Users repeatedly find themselves needing to concat or merge n-dimensional arrays and this operations is extremely costly because in the end the operational system has to allocate a new byte array in memory. This issue is however solved in Hadoop environment by HDFS, where the second array is simply appended to the first on the file system.

Algorithm parametrizing

Algorithm parametrizing is among the most expensive techniques for optimizing models and can only be used for models that can be parameterized. The main idea is to execute a brute force grid search to find parameter values, which in turn produces a model with the highest accuracy. In most cases, the relationships between parameters and models are not linear and some heuristics might need to be applied. Nevertheless, this technique still provides very a good pay off when employed. If combined with K-fold cross validations, the computational requirements grow even bigger, but as stated above the benefits are enormous.

3.1.5 Predicting

Predicting embodies two elements: new data and answers (predictions). New data is a set of explanatory features without target values, which are produced by real-world(production) systems and their ultimate purpose is to predict answers. Likely issues predicting could potentially struggle with are twofold: high volumes of new data and predicting velocity. The answers are recalled target values. Either with a predicted value or maybe a later collected real one, the new data can subsequently be incorporated into the historical data. Obviously this can be rather useful for online machine learning. Within this area, the term "real-time" is used fairly often, this is however not feasible in the real world, therefore the term "near real-time" should be adopted.

3.1.6 Supervised and unsupervised learning

Supervised learning refers to a situation where historical data contains a target value or a variable, this data is also usually called labelled data. Most of the techniques and methods concentrate on supervised learning. Accordingly, if these are to be employed for unsupervised learning, modifications are necessary. The most common cases for supervised learning are regression and classification. In many situations, it is normally much easier to build and interpret machine learning models in supervised learning because the set of values the target value can have, as well as the unit, are known in advance.

As outlined in the previous paragraph, unsupervised learning involves historical data which does not contain a target value or a variable. It can be used for a smaller subset of problems, such as classification, missing features or discovering rules. Finding real-world applications might also be highly challenging, but in the world of machine learning unsupervised learning plays a big role in understanding the model and the data that are used in supervised learning. As previously noted, using one or the other is not an omnipotent solution to fix all problems. The best results and outcomes can be achieved by combining both

3.1.7 Scaling machine learning

Scaling machine learning bears the same set of limitations as any other area in computer science. It varies from I/O limits, memory size or computational time. Traditional approaches to scaling include vertical and horizontal scaling, where vertical scaling needs to be supported by a machine learning model/algorithm or a system around it. Most machine learning algorithms have $O(n^2)$ or $O(n^3)$ complexity and therefore cannot be scaled linearly. This introduces the need to find other ways to scale and a whole new area of research and problems. Some of those problems are listed in table 3. Most approaches to these problems can have the potential side effect of decreased accuracy.

Table 5 The most common problems with scaling machine learning workflows

Problem
Training data set is too large to fit the model
Training data set is so large that cannot fit into memory
Data rates are too fast for the machine learning model
Feature-engineering or predictions are too slow
Data sizes (batches) are too large to be processed with the model

In order to correctly apprehend the nature of this problem, it might be useful to mention the various types of data machine learning deals with - it can be categorized into four general types: tall data, wide data, tall and wide data and sparse data. Tall data can be represented by a large amount of rows or records. Wide data on the other hand is described with a large number of features. Tall and wide data combines both and sparse data have rows without entries in features (when transformed into matrix, translated to zeros). All of these types introduce their own problems into machine learning. In the next two subchapters I will discuss scaling learning and predicting parts of machine learning, respectively.

Scaling learning

The main problem within scaling learning is data volume, as in the data being too large to fit in a computer’s memory. The machine learning workflow for scaling learning can be seen in Figure 16, scaling learning also implies that prediction needs to be scaled, so feedback time can be shortened. Learning scaling can be solved by two main approaches, first is to decrease data’s memory footprint and second is to distribute learning over instances (cluster computing).

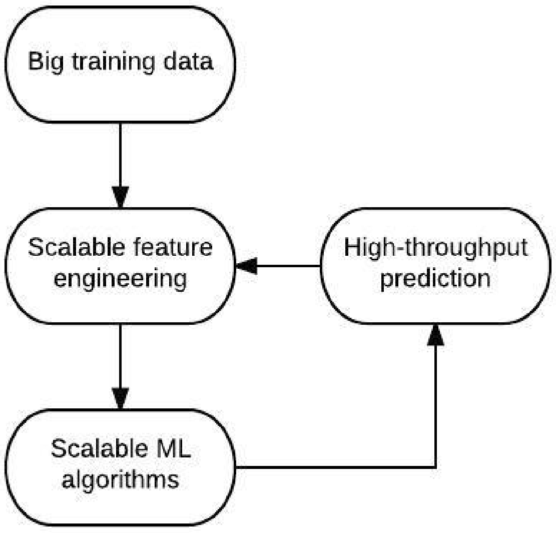

Figure 16 The scaled learning machine learning workflow (Reference (Brink, et al., 2016, p. 203) edited)

Lowering memory footprint can be attained via several methods, these are feature selection, out-of-core learning, subsampling and better machine learning algorithm selection. The feature selection comes from feature engineering part of machine learning, this is especially important for wide data (with many feature variables). The features need to be selected more rigorously to decrease the dataset size as much as possible. Forward and backward selection is a popular method for feature selection. This selection generally uses grid search by adding or removing features and completing a full cycle of machine learning, in most situations forward selection makes more sense because it requires less memory and computational time. A potential disadvantage of this kind of selection could be that leaving out a feature could prevent the discovery of certain relationships within the data, which would lead to a decrease in model accuracy.

Subsampling data is an option usually used as a last resort, when all else fails. Subsampling is just plain random selection of a subset of the main data set. It solves both problems of learning scaling (volume, velocity). But in most case it also introduces biases that are challenging to foresee, when truly selecting random subset every machine learning cycle or iteration, the biases vary greatly and the general machine learning model cannot be adjusted.

When selecting the same subset for every cycle, the model suffers a huge decrease in accuracy. Multiple subsample technique is called bagging (bootstrap aggregating). The main data set is subsampled N-times and N subsets are created, machine learning model is applied to each subset and then a majority vote is taken for predictions. Combination of models is also known as model ensembles. For example, random forest algorithm uses bagging or boosting technique and is very similar to this.

A further option is out-of-core learning, where the decision not to leave out any data or features is made, and learning is done in a stream-like manner (one instance at the time). Used algorithm needs to support this option and performance varies. This “online” learning can introduce a bias within the model by using specific order of data. Therefore shuffling techniques can be used to lower the possibility of bias, but it cannot be eliminated entirely. The shuffling techniques are limited by the same problems as machine learning algorithm, data does not fit into memory and in most cases needs to be split and shuffled separately. Using out-of-core learning essentially means solving one problem and introducing another - slow learning speed. Most algorithms are stochastic and use batch gradient descent.

The algorithm selection is crucial in scaling, for example, in case of sparse data, if the algorithm can process zeros within the matrix (by for example compression or using sparse matrix), the memory footprint is significantly reduced. As in previous options, selecting a different algorithm usually introduces another problem or lowers accuracy, meaning there is no one-size-fits-all solution. To truly scale the learning part, the prediction parts needs to be scaled as well, because the modeling is iterative and needs predicting for its validation.

Using distribution for scaling learning is probably used most commonly when data engineers encounter scaling issues. The most popular option is Hadoop (Sjardin, et al., 2016, pp. 307-324), it consists of two layers (storage and computing). The storage layer, called HDFS (Hadoop Distributed File System), is responsible for storing and distributing data across the cluster. The computing layer, MapReduce, does all the computations in two kinds of tasks (Map and Reduce). Map task can be distributed over the cluster and computations are done on all nodes, where Reduce task is responsible for collection and reducing outputs from Map tasks. The main disadvantage of Hadoop is that, Reduce tasks are done by storing results on disk, not into memory. Hadoop can be substituted with Apache Spark, which solves this issue and uses memory for Reduce tasks.

Scaling predictions

Scaling predictions is not only about scaling the velocity and volume of predictions, it also involves scaling the system as such. It is highly probable, that there would thousands of concurrent clients wanting to do the prediction and the system should be able to handle them all. Scaling predictions is more about designing the system and its architecture.

To scale the volume of predictions, one can use a worker pattern to distribute the load and handle peaks. The system consists of three main components: queue, workers and storage (Figure 17). The queue handles incoming requests for predictions and places them into a queue. There can be more queues based on the type of requests and the queue can work in different modes (FIFO, LIFO, PIFO or even random order). Workers take requests (can work in both modes – push or pull) from the top of the queue and all of them have the machine learning model and recall the requested target value. After that the worker puts the recalled value into storage, the recalled value is persisted with a unique identifier and metadata. It is up to the system's designers to decide how many workers are necessary. Potential pitfalls associated with this solution are an overflowing queue and a long prediction time when the system is flooded.

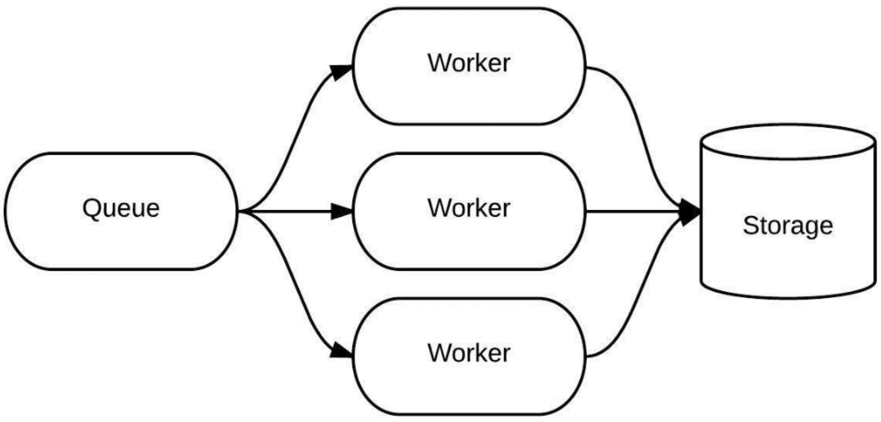

Figure 17 System design for scaling prediction volume

To scale prediction velocity, one must design the system properly and choose a machine learning algorithm to fit this need. A rather naive approach would be to handle prediction on multiple machines with the earliest result propagated back to request's origin. This approach is significantly inefficient and, simply put, a huge waste of resources. A more advanced approach is to design a cache and prediction dispatcher, which can either use results from cache if present or delegate the requests further. As with any other caching, result invalidation is a legitimate concern and is up to the system designer to deal with it. Additionally, to achieve a more complex and higher performing solution, one can pick a scalable model and design the system around it, an example would be using random forests. This solution consists of four components: producer, consumer, workers and storage (Figure 18). Producer handles incoming requests for prediction and dispatches it to workers (each worker is tasked with a subset of predictions). When any subprediction is finished, it is pushed to consumer, which aggregates the result. The consumer is responsible for aggregating results, storing results and subresults in storage and for responding with recalled value. The consumer can at any time decide to answer a request. The consumer can do so if there is a result in storage, over half of subpredictions are finished (inaccurate result) or a request is timing out. A system like this very complex and needs careful fine-tuning in order to work accurately. Proper cancellation of subpredictions, for example, is one of the struggles this kind of a system faces and if solved could save a substantial amount of resources. The solution can, however, prove to be quite challenging.

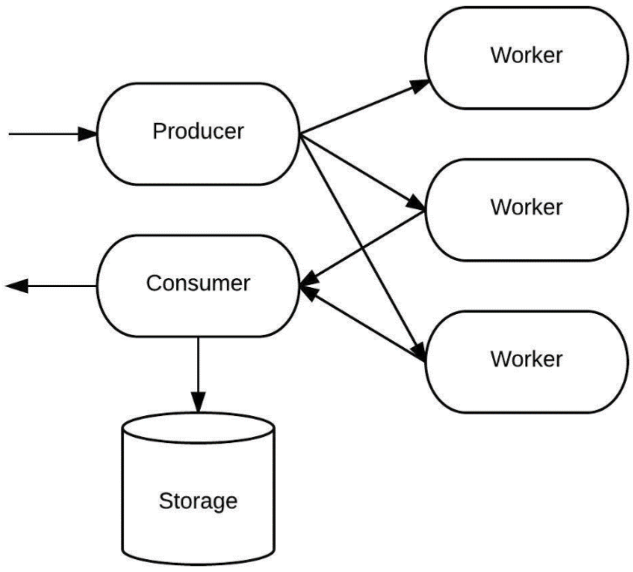

Figure 18 Consumer/producer system design (Reference (Brink, et al., 2016, p. 209) edited)

3.2 Load Balancing Strategies

Load balancing strategies can be described as rules for load distribution. The rules can be in a hierarchic structure (tree-like) and be applied from root rule to leaf rules in respective order. The simplest strategy is random distribution, but is rare for it to be applied in computer science (due to its performance). In the following chapters metrics, strategies and implementations are discussed.

For the objectives of this thesis, the system using load balancing can be simplified just into two components: load balancer and service units (backend servers), as portrayed in Figure 19. The load balancer handles incoming HTTP traffic and distributes the load across the service units according to strategy (via proxying). The service units handle proxied traffic and respond with a HTTP response. The load balancer proxies the responses back to its origins, but can take extra actions if required.

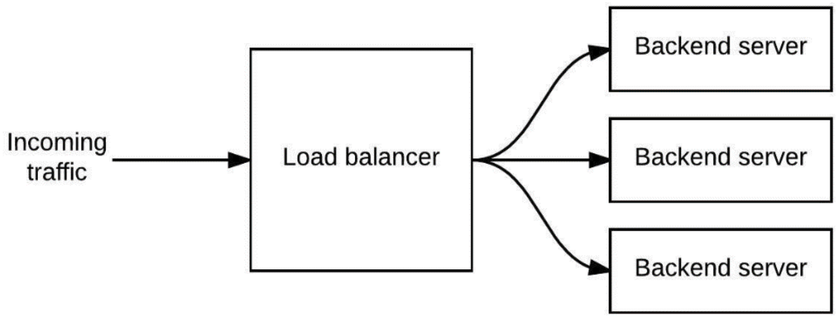

Figure 19 System using load balancing

3.2.1 Performance metrics

There are three main performance metrics when it comes to HTTP load balancing. But all three of them provide little to no insight into backend services. Having said that, these metrics are still important and are an industry standard when measuring load balancer's performance or any other server's. The first one is connections per second, it is a number of connections to the load balancer in any given second. This metric is directly connected to load balancer's ability to open and close HTTP/HTTPS connections. The second metric is total concurrent requests, measuring number of active connections in a given second. It shows how well can the load balancer handle heavy and long lasting traffic. The last metric is pure throughput, measuring the amount of traffic that the internal system can handle per second. Only the last one can reflect the state of backend services, but does not give any insight into the type of traffic. When profiling a load balancer, these metrics have an importance order based on traffic type (Table 6)

Table 6 Importance of metrics based on traffic type (Reference Bourke, 2001, p. 35)

Traffic type	**1st**	**2nd**	**3rd**
Web store	Total concurrent connections	Connections per second	Throughput
FTTP/Streaming	Throughput	Total concurrent connections	Connections per second
HTTP	Connections per second	Throughput	Total concurrent connections

These metrics can be used as a starting point when evaluating the performance of any load balancer. But as previously mentioned load balancing is a very domain specific issue and more specific metrics needs to be created when needed, reflecting the system and traffic profiles.

Utilization

The idea of having a performance metric that is based on the performance or the state of backends services, despite sounding fairly simple, is not that simple to implement in the real world as coming up with a such a metric or metrics is complicated. It seems rather useful to use queueing theory concepts to formalize those metrics

For the sake of simplicity, I will use a queueing model for M/M/1 system (Kořenář, 2010, pp. 148-153), exponential distribution income rate, exponential distribution service rate and one service unit. The basic model also expects queue using first in first out concept. There are several equations that formalize relations within this model (Figure 20).

$$\rho = \frac{\lambda}{\mu}$$

$$p_o = 1 - \rho$$

$$p_n = (1 - \rho) * \rho^n, where\ n \geq 0$$

Figure 20 The relations in M/M/1 model

The scalar λ describes the income rate into the system, the scalar μ describes the service rate, the scalar ρ is a stability variable (system is not stable when over 1) and the scalar p_n is a probability that there are n requests in the system.

With service unit limiting number of requests to be serviced at single moment to one, it is easy to use these equations to calculate probability when there is at least one request within the system ($1 - p_0$). This can be used as a utilization metric, with very simple reasoning behind it.

The queueing theory can provide the scientific and mathematical background for more advanced metrics, but it becomes difficult to apply when creating a domain specific utilization metric. For most cases it is enough to be a subject matter expert in order to be able to create a utilization metric.

3.2.2 Strategies

In load balancing, several terms are used to describe the decision making of which unit will be selected to service the requests. These terms are strategy, algorithm and method. While from a practical standpoint these terms are more or less interchangeable, when it comes to formalizing, algorithm is the one used most often. The strategies can be split into two groups: static and dynamic, where a static strategy is a set at load balancer and a dynamic strategy can be changed or extended at runtime. Most strategies also have a weighted variant, sometimes using ratios to express these weights.

Round robin

The round robin is probably the simplest and most used strategy world wide. Almost every load balancer implements this method and it is set as a default option. The load balancer is supplied with a list of servers (backends, units) and upon request advanced go through the list by one and forwards to server at current position. When the load balancer reaches the end, it loops back to the start of the list. Pseudo code is provided in Code sample 2

Code sample 2 The round-robin method in pseudo code

```
list= [...]
index = 0
requests = [...]
length = size(list) - 1
for request in requests:
    list[index].process(request)
    if (index < length):
        index++
    else:
        index =0
```

The round robin method provides a good starting point. However, its disadvantages, such as its lack of the ability to reflect the state of service units and the profile of current traffic make it less desirable. An additional weakness of this method is that with an increasing number of service units, it has higher deviations in units utilization.

IP hash table

The IP hash table (Nginx) is also popular and has some niche usages. It allows to bond service unit with IP (in many cases one client). In practice, it takes request's originating IP address and hashes it, after that it assigns it to one service unit and whenever the request from the specific IP address it is forwarded to the assigned service unit. The pseudo code is in Code sample 3.

Code sample 3 The IP hash table method in pseudo code

```
list= [...]
requests = [...]
ipHashTable = {...}
length = size(list) - 1
for request in requests:
    selected =ipHashTable[request.originatingIp]
    if (selected = None):
        selected =list[random(0,length)]
        ipHashTable[request.originatingIp] = selected
        selected.process(request)
     else:
        selected.process(request)
```

This is especially useful when the backend server has to share a state and encounters either latency or synchronization issues. This allows one IP to be bound with one backend server and from the outside it appears to be in a consistent state (large distributed system has to deal with CAP theorem). In practice, this method falls short in several areas. It neglects the fact that in the real world client's requests can originate from several IP addresses and can be continuously changing over time. It also fails to reflect the client profiles, one client can generate more traffic than others. This method is usually employed when handling file transfers, for example, when dealing with protocols such as WebDav, SFTP, FTPS and FTP. A principle similar to IP hash table could be applied when having domain specific affinity with backend units.

Least connection

The least connection method is also implemented in many load balancers. It provides the simplest reflection of backend server's states. By using several counters it keeps track of the number of open connections to backend servers. The pseudo code is in Code sample 4, in this instance, the algorithm is more difficult to formalize, since it relies on the current state of the load balancer. The problem with this method is that the number of connections does not usually translate into load metric, meaning load balancers normally reuse connections, and it does not reflect traffic profile.

Code sample 4 The least connection method in pseudo code

```
list= [...]
connections = [...]
requests = [...]
for request in requests:
    selected = min(connections)
    selectected.increment()
    selected.process(request)
    selectected.decrement()
```

The load balancer is a concurrent environment and multiple requests are processed in parallel, therefore some methods needs to be synchronized on its state to work correctly. The synchronization brings overhead and might also be the reason not to prefer this method over other methods.

Fastest

Implements a similar concept as least connections algorithm, but it picks the fastest out of the possible service units. It is almost impossible to use all-time average response time, therefore average response time in the last fifteen minutes is used, for example. In practice, it would require a much shorter period in order to reflect the current state of service units. This requires the implementation of rolling windows to efficiently count them. The rolling windows can be implemented by hash wheels.

Least loaded

The least loaded method is a modified version of the fastest method. The key difference lies in that this method uses a user defined for load of backend servers. The state of backend server could be passed via HTTP headers or other components. CPU utilization or system load can be used to compute the metrics. This method is very domain specific and also requires rolling windows in order to work.

4 Results

To illustrate whether machine learning can be applied to HTTP load balancing I have conducted tests upon which I drew conclusions. These tests were conducted in a limited environment and on a specific data set, which could have introduced a bias. However, it is virtually impossible not introduce any bias, because HTTP load balancing is very domain specific and machine learning has a high propensity for bias.

4.1 Tests

As covered in previous chapters, load balancing strategies can measure its performance based on a selected metric. When selecting a metric to use, I looked into response time and payload size of HTTP requests, because it was probable that both of them would appear in any HTTP traffic log. Next section covers dataset selection and description. Both response time and payload size can be used to reflect on current state of backend servers. Moving forward, I have decided to use response payload size.

Created metric is a sum of response payload size (in bytes) handled by one backend server, this metric is easy to interpret and can be derived from. It is formalized as in Figure 21.

$$A = (a_{ij}), binary\ matrix\ whether\ request\ i\ was\ balanced\ to\ backend\ server\ j$$

$$\sum_{i=1}^{n} a_{ij} * payload_i = load_j, where\ j\ is\ index\ of\ backend\ server$$

Figure 21 Simple metric based on response payload size

This metric is in an absolute number and because it was used as a target variable in a machine learning model I have decided to use a derivative of this metric. The average load is calculated (100 % utilization) together with deviations from it for each server, based on those numbers a standard deviation can be calculated Figure 22. This final metric is used for comparison between models and load balancing strategies.

$$load_j^{dev} = \overline{load} - load_j, for\ each\ backend\ server (index\ j)$$

$$stdev = \sqrt{\frac{\sum_{i=1}^{n}(load_i^{dev} - \overline{load^{dev}})^2}{n-1}}$$

Figure 22 Derived metric based on standard deviation

A utilization value under one means that the backend server is underutilized and utilization value over one means that it is overutilized. For supporting purposes, I have also calculated

RMSE and R-squared when dealing with machine learning models to gain a better understanding of created models. Later on, I also added timers to measure how long it took the model to learn, to see whether the measured model is a viable option in real life.

During the tests, I used two different load balancing strategies: round robin and least loaded with payload size. To simulate a load balancer I created two processors to act as load balancers, since it would be too costly to perform tests on a full scale system with real load balancers. Implementation of round robin is straight forward (Code sample 5) and drew on the method described in Round robin section.

Code sample 5 Implementation of round robin processor in Python

```python
import logging as log


class SimpleRoundRobinProcessor:

    def __init__(self, node_count):
        self.node_count = int(node_count)
        self.current_node = 1
        self.predict_node_counters = [0] * int(node_count)
        self.real_node_counters = [0] * int(node_count)

    def process(self, predict, real):
        log.debug('action=process status=start')
        # select next node
        selected_node = self.current_node
        # get next node
        self.current_node += 1
        # back to first node
        if self.current_node > self.node_count:
            self.current_node = 1
        # index starts with 0
        index = selected_node - 1
        # add payload_size to counter
        self.predict_node_counters[index] += predict if predict > 0 else 0
        self.real_node_counters[index] += real if real > 0 else 0
        log.debug('action=process status=end')
```

Both processors did a simple check when summing the payload size, if the payload size was less than zero, then zero was used. It is because the machine learning models are unaware of the HTTP protocol and it would be necessary to implement the algorithms with this in mind. The second processor chose “the least loaded” and implementation was very similar to the first processor (Code sample 6).

Code sample 6 Implementation of least loaded processor in Python

```
import logging as log


class BestCounterProcessor:

    def __init__(self, node_count):
        self.node_count = int(node_count)
        self.predict_node_counters = [0] * int(node_count)
        self.real_node_counters = [0] * int(node_count)

    def process(self, predict, real):
        log.debug('action=process status=start')
        index = self.predict_node_counters.index(
                    min(self.predict_node_counters))
        # adding payload_size size to counter
        self.predict_node_counters[index] += predict if predict > 0 else 0
        self.real_node_counters[index] += real if real > 0 else 0
        log.debug('action=process status=end')
```

As stated above, the tests consisted of three stages. The first stage was about creating a baseline and benchmarks for later comparisons. The stage used both processors to measure the standard deviations in the nine following scenarios:

1. 3 backend servers, 1000 rows
2. 3 backend servers, 10000 rows
3. 3 backend servers, whole dataset
4. 5 backend servers, 1000 rows
5. 5 backend servers, 10000 rows
6. 5 backend servers, whole dataset
7. 10 backend servers, 1000 rows
8. 10 backend servers, 10000 rows
9. 10 backend servers, whole dataset

The same nine scenarios were also used in the third stage for final comparison. The second stage selected five different algorithms to create machine learning models and used them to predict a payload size for a scenario with three backend servers. The model was fitted with 10000 rows and K-fold validation method was used (with 10 folds). The selected algorithms were:

- Ordinary least-squares regression (Scikit-learn, 2016)
- Ridge regression (Scikit-learn, 2016)
- Lasso regression (Scikit-learn, 2016)
- Regression tree (Scikit-learn, 2016)
- Stochastic gradient descent (Scikit-learn, 2016)

All of the above stated algorithms can be parametrized, therefore a grid search was conducted to find the best performing models. A full grid search was not used, because many parameters are of float type (it would create too many combinations, which were not used). During the feature engineering phase of creating models, it was discovered that the URL path, could be transformed in such a way it could better reflect its hierarchic structure (with limited depth of tree structure to eight), so this transformation was also included into the grid search, as well as limiting models to URL and HTTP method as feature variables only. The parameters for combinations were as follows (per algorithm):

- Ordinary least-squares regression:
    - Normalize: True, False
    - Fit intercept: True, False
    - Parse URL: True, False
    - Selected features: All, URL + method
- Lasso regression
    - Normalize: True, False
    - Fit intercept: True, False
    - Positive: True, False
    - Selection: cyclic, rando,
    - Parse URL: True, False
    - Selected features: All, URL + method

- Ridge regression
    - Solver: lsqr, cholesky, sparse_cg, sag, svd
    - Normalize: True, False
    - Fit intercept: True, False
    - Parse URL: True, False
    - Selected features: All, URL + method
- Regression tree
    - Criterion: mae, mse
    - Splitter: random, best
    - Max depth: 10
    - Max features: auto, sqrt, log2
    - Parse URL: True, False
    - Selected features: All, URL + method
- Stochastic gradient descent:
    - Loss: huber, epsilon. squared_epsilon, squared_loss
    - Fit intercept: True, False
    - Parse URL: True, False
    - Selected features: All, URL + method

The last stage involved shortlisting the best performing models from stage two and executing tests on all nine scenarios from stage one. The reason for reducing the number of tests was that it would be computationally too expensive to run every scenario with every combination in grid search. The tests can be summarized into three stages:

1. Creating a baseline and benchmarks for round robin and least loaded.
2. Creating models on the subset and testing them.
3. Testing the shortlist of models on the whole dataset.

4.1.1 Dataset

In 2001, Doug Laney described big data with 3 V's - variety, velocity and volume (Laney, 2001). In terms of my problem, the data qualifies for two of three V's, velocity and volume. For general data analysis, traffic on load balancers is usually too fast to be analyzed in real-time or in "near real-time" (short feedback loop). Meaning more data is incoming than what is processed and analyzed. Load balancers deal with thousand plus units of traffic (requests) per second and in most case also run in clusters, which further increases the number of units. Qualifying for the third V is questionable, because most load balancers handle closed sets of traffic and handling an unusual or unknown unit is not common.

While searching for a data set I could use for my tests, I created a set of requirements. The data set should be of a reasonable size, meaning not too small, to prevent the results from becoming skewed (underfitted) and not too big, so that tests would complete in within a reasonable timeframe (less than a day). The data set should consist of HTTP traffic and ideally contain requests to response time. Another requirement was for it not to have any limitations in terms of altering or using it enabling me to use it freely without any constraints. There is no shortage of TCP/IP traffic dumps available on the internet, which are one layer below of what I need and would make my tests too complex. The best suiting traffic dumps of HTTP I found were on The Internet Traffic Archive.

After scouring through several of them, I have chosen the EPA-HTTP trace (The Internet Traffic Archive, 1995). To be thorough, I have contacted Dr. Laura Bottomley from North Carolina University, to ask her about the data set. She confirmed that the data set is open to use and there are no limitations placed on it for altering or usage. The data set contains 47 748 requests, which are described with host, date, HTTP method, path, protocol, HTTP status code and response byte size. Response byte size can be used for utilization calculations and to be the predicted variable for future requests. This leaves six properties to be used as descriptive ones (Table 7).

Table 7 Structure of dataset

Name	**Description**
host	DNS hostname or IP address
date	Time stamp in DD:HH:MM:SS format
HTTP method	HTTP request method
path	Request path with query parameters
protocol	Used protocol and its version
HTTP status code	HTTP response status code
response byte size	Integer number of response payload size

4.1.2 Dataset transformation

The selected data set comes in a text file in an inconvenient data format (Code sample 1). In order to make the data as convenient as possible, I have decided to parse them and import them into a relational database. The relational database enabled me to repeatedly read without the cost of transformation and is simple to work with.

Code sample 7 Dataset format

```
host [date] "http_method path protocol" http_status_code size
```

For the database I have selected MySQL (on AWS) and MariaDB (on personal computer), due to having hands-on experience with both of them and also them both being interchangeable without any significant cost. For importing and parsing data, I have written my own script in Python. The script handles the file line by line and splits them into features, after that those features are templated into SQL insert statements. The script (Code sample 8) can handle missing data or features. When the dataset is parsed it can be imported effortlessly into the database.

Code sample 8 Transformation script in Python

```python
#!/usr/bin/env python
import sys

def is_http_method(param):
    return param == u'POST' or param == u'GET' or param == u'HEAD' or \
        param == u'OPTION' or param == u'DELETE' or param == u'PUT'

args = sys.argv
if (args and len(args) > 1):
    limit = int(args[1])
else:
    limit = None

with open('raw_data', 'r') as f:
    with open("data.sql", "a") as data_file:
        for i, line in enumerate(f):
            values = line.split()
            if (len(values) > 7):
                continue
            values[1] = values[1].strip("[]")
            values[2] = values[2].strip('"')
            if (not is_http_method(values[2])):
                values.insert(2, 'NULL')
            values[3] = values[3].strip('"')
            values[4] = values[4].strip('"')
            if (values[4] != u'HTTP/1.0'):
                values.insert(4, 'NULL')
            values[6] = values[6] if values[6] != u'-' else 0
            table = 'data'
            insert = "INSERT INTO %s (source, time_stamp, method, url, " \
                "protocol, status, payload_size) " \
                "VALUES (\'%s\', \'%s\', \'%s\', " \
                "\'%s\', \'%s\', \'%s\', %s); \n" % (table, values[0], \
                values[1], values[2], values[3], values[4], values[5], \
                values[6])
            data_file.write(insert)
            if (limit and limit < i):
                break
```

When loading the dataset into memory a few more transformation had to be done. Those were transformation of categorical variables to indicator variables and possibly parsing URL to hierarchic structure. These transformations were implemented in Code sample 9 and Code sample 10.

Code sample 9 Transformation of categorical values in Python

```
import pandas as pd
import transformation.url as url


def tranform(dataframe, omit, parse_url=False, limit_url=None):
    categorical_headers = list(dataframe.select_dtypes(
        include=['category', 'object']).columns)

    _remove_if_contains(categorical_headers, omit)

    result = dataframe.copy()

    for x in categorical_headers:
        dummies = _transform(x, result)
        # there is no inplace concat, in the end numpy concatenate will get
        # called and new array will be allocated
        result = pd.concat([result, dummies], axis=1)
        result.__delitem__(x)

    new_headers = list(result.columns)
    _remove_if_contains(new_headers, omit)
    if (parse_url and parse_url == u'true'):
        result = url.parse(result, limit_url)
        return result, list(result.columns)
    else:
        return result, new_headers


def _remove_if_contains(from_list, remove):
    for o in remove:
        if o in from_list:
            from_list.remove(o)


def _transform(column, result):
    dummies = pd.get_dummies(result[column])
    dummies.rename(columns=lambda x: "%s/%s" % (column, x), inplace=True)
    return dummies
```

Code sample 10 Transformation of URL to hierarchic structure in Python

```
import logging as log
import numpy as np
import pandas as pd
from itertools import chain
```

```
def parse(dataframe, limit=None):
    log.info("action=url_parse status=start")
    values = _remove_query_params(dataframe['url'].values.tolist())
    values = list(chain.from_iterable([_split(val, limit) for val in values]))
    values = _deduplicate(values)

    new_df = dataframe.copy()
    count = 0
    log.info("action=url_parse values_size=%s" % len(values))
    for value in values:
        if (count % 100 == 0):
            log.info("action=url_parse count=%s" % count)
        new_column = np.zeros(shape=(len(dataframe.index), 1))
        idx = 0
        for url in new_df['url'].values:
            if (url.startswith(value)):
                new_column[idx, 0] = 1
            idx = idx + 1
        new_df = pd.concat([new_df, pd.DataFrame(
            data=new_column, columns=['url-' + value])], axis=1)
        count = count + 1

    new_df.__delitem__('url')
    log.info("action=url_parse status=end")
    return new_df


def _split(url, limit=None):
    splits = url.split("/")
    splits = ["/".join(splits[:i + 2]) for i, split in enumerate(splits)]
    if (limit):
        splits = splits[:limit]
    return splits


def _remove_query_params(row):
    _row = row
    if "?" in _row[-1]:
        _row[-1] = _row[-1].split('?', 1)[0]
    return _row


def _deduplicate(seq):
    checked = []
    for e in seq:
        if e not in checked:
            checked.append(e)
    return checked
```

4.1.3 Environment and execution

The tests were conducted in three different environments, the first one was my personal computer, which was used to develop and test scripts. The other two were instances on Amazon Web Services and these were used to conduct the tests. The sizing of these two instances was crucial because machine learning can be very demanding for resources (computational power and memory). Therefore I have used c4.8xlarge and r4.8xlarge EC2 instances, the first one had 36 vCPUs and 60 GB memory and the second one had 32 vCPUs and 244 GB memory. The reason for this sizing was not to hit memory deficiency problems and to complete computations within a reasonable timeframe.

As the operational system, Linux distribution from Amazon was used - AMI. The basic distribution does not contain all the tools required to do any machine learning. Therefore I have snapshotted an image with all the required tools (plus data) and it can be provided upon request (reason not to include it was that it is around 8 GB in size). The environment can be set up manually by following these steps:

1. Install Fortran, C and C++ compilers (`yum install gcc-gfortran gcc gcc-c++)`
2. Install MySQL server, command line client and other related libraries (`yum isntall mysql mysql-server mysql-devel mysql-lib`)
3. Install BLAS prerequistes (`yum install automake autoconf libtool*)`
4. Install BLAS (`yum install lapack-devel blas-devel atlas-sse3-devel`)
5. Install Python 3.5 and virtualenv (`yum install python35 python35-virtualenv python35-pip`)
6. Create and source new environment (`virtualenv-3.5 ml; source ml/bin/activate`)
7. Install Cython via Pip (`pip install cython`)
8. Install required dependencies in source code folder (`pip install -r requirements.txt`)

Note: Installation via Pip may cause compilation of several libraries into C and C++ and it can make it very slow (verbose mode recommended). Other issue is that Pip may not respect dependencies between libraries in `requirements.txt` (install separately if necessary).

For the creation of machine learning models and other things several Python libraries were used. Despite attempting to implement models by myself at first, I could not compete with professional models. All of the libraries were configured to use BLAS if needed. The following libraries were used:

- NumPy (3.5.0) (Numpy, 2017)
- SciPy (1.3.9) (Scipy, 2017)
- Pandas (0.19.2) (Pandas, 2016)
- Scikit-learn (0.18.1) (Scikit-learn, 2016)

- Statsmodels (0.6.1) (Statsmodels, 2012)
- Mysqlclient (1.3.9) (Python Software Foundation, 2017)
- Configparser (3.5.0) (Python Software Foundation, 2017)
- Pytest (3.0.7) (Python Software Foundation, 2017)

For the purposes of test execution, there were several scripts and wrappers that started the execution. The main wrapper was written in Python and was named `limython.py`. It has several mandatory options. Help for this script is in Code sample 11.

Code sample 11 Help for limython.py script

```
usage: limython.py [-h] -p PROCESSOR -n NODE_COUNT -l LIMIT [-k K_FOLDS] -m
                   MODEL [-a ARGUMENTS] [-u URL] [-ul URL_LIMIT] [-f FEATURES]
                   -r RUN

Python wrapper to simulate load balancing system and to run tests.

optional arguments:
  -h, --help            show this help message and exit
  -p PROCESSOR, --processor PROCESSOR
                        Request processor(simple-round-robin,best-counter)
  -n NODE_COUNT, --node-count NODE_COUNT
                        Node count for processor(1,2,3,...)
  -l LIMIT, --limit LIMIT
                        Data limit(to avoid running out of memory)
  -k K_FOLDS, --k-folds K_FOLDS
                        Number of folds for cross validation(higher better,
                        but computation more expensive)
  -m MODEL, --model MODEL
                        ML model(ols, lasso, ridge, sgd, tree)
  -a ARGUMENTS, --arguments ARGUMENTS
                        ML model arguments - kwargs separated with comma
  -u URL, --url URL     Flag whether url should be parsed into tree like
                        indicators
  -ul URL_LIMIT, --url-limit URL_LIMIT
                        Limit depth of tree hierarchy of dummy variable parsed
                        from urls
  -f FEATURES, --features FEATURES
                        List of features - comma separated
  -r RUN, --run RUN     'true' or 'false' whether to run full with processor
```

There were two other scripts for the first and last stages of tests (`baseline.py` and `final.py`), the first scripts created benchmarks and the second script ran tests for all test scenarios. The last stage was automated with `final_cut.sh` script.

For stage number two, there were five BASH scripts to help parallelize executions and they ran the grid search and wrote output into the log files. After that log files were parsed to csv files with `result_to_csv.py`. The list of BASH script used is as follows:

- Ordinary least-squares: `ols.sh`
- Lasso: `lasso.sh`
- Ridge: `ridges.sh`
- Tree: `tree.sh`
- Stochastic gradient descent: `sgd.sh`

More detailed help and tips are available at the GitHub repository with all of the sources code and data (GitHub) or in Manual for scripts and tests

4.2 Reports

This chapter splits into three subchapters where each stage has its own reports. In most cases, the reports are modified in order to fit into the thesis, because their size can be overwhelming. Reports are based on log files collected from tests and parsed via `result_to_csv.py` script into csv files. Full logs and csv files can be found in the attachments to this thesis.

4.2.1 Baseline and benchmark stage

For this stage, there are two main reports: standard deviation of utilization for round robin report and standard deviation of utilization for least loaded (with known payload sizes in advance) report. Both reports are based on ETA-HTTP dataset with default ordering.

Standard deviation of utilization for round robin report

Benchmarks for standard deviations for round robin are in Table 8.

Table 8 The standard deviation of utilization for round robin report

Standard deviation of utilization	**3 backend servers**	**5 backend servers**	**10 backend servers**
1000 rows	0,635652908	0,647969823	0,913589388
10000 rows	0,199151598	0,202802295	0,272264000
All data	0,064889348	0,142088391	0,137149010

Standard deviation of utilization for least loaded report

Benchmarks for standard deviations for least loaded are in Table 9.

Table 9 The standard deviation of utilization for least loaded report

Standard deviation of utilization	**3 backend servers**	**5 backend servers**	**10 backend servers**
1000 rows	0,376755727	0,518987958	0,752120807
10000 rows	0,008293512	0,027780778	0,088761172
All data	0,004581321	0,007026258	0,016521689

4.2.2 Subset stage

With five different algorithms and several parameters, the number of combinations can grow exponentially. Within these tests there were 181 combinations identified and with 10 folds per each run, the report could not fit into the thesis. Four metrics were measured: standard deviation of utilization, learning times, RMSE, R-squared. Each metric has a report with top 10 values by 95th percentile (across the folds).

Standard deviation of utilization

The combinations (algorithms and parameters) for best performing models by comparing standard deviation of utilization are listed below and the results are in Table 10:

1. Ridge (solver=sag, normalize=True, fit_intercept=True, parse_url=true)
2. SGD (loss=huber, fit_intercept=1, parse_url=true)
3. Ridge (solver=sag, normalize=False, fit_intercept=True, parse_url=true, features=method,url)
4. Ridge (solver=lsqr, normalize=True, fit_intercept=False, parse_url=true, features=method,url)
5. Ridge (solver=lsqr, normalize=False, fit_intercept=True, parse_url=true, features=method,url)
6. Ridge (solver=lsqr, normalize=False, fit_intercept=False, parse_url=true, features=method,url)
7. Ridge (solver=lsqr, normalize=True, fit_intercept=True, parse_url=true, features=method,url)
8. SGD (loss=huber, fit_intercept=0, parse_url=true)
9. Lasso (normalize=True, fit_intercept=True, positive=False, selection=random, parse_url=true, features=method,url)
10. Lasso (normalize=False, fit_intercept=False, positive=False, selection=random, parse_url=true, features=method,url)

Table 10 Top 10 for standard deviations of utilization by 95th percentile

Combination	Min(Best)	Max(Worst)	Average	95th percentile
1.	0.1223997271	0.3859545203	0.2461609237	0.3626272154
2.	0.0061885561	0.3655940797	0.1303263835	0.3631977438
3.	0.1588243565	0.3828991316	0.2495062115	0.3639495889
4.	0.0281614523	0.3992078593	0.2414770141	0.3794169234
5.	0.0281614523	0.3992078593	0.2414770141	0.3794169234
6.	0.0281614523	0.3992078593	0.2414770141	0.3794169234
7.	0.0281614523	0.3992078593	0.2414770141	0.3794169234
8.	0.0029763972	0.386146587	0.1658991777	0.382955825
9.	0.000742486	0.3886434329	0.1086933993	0.3832811104
10.	0.000742486	0.3886930882	0.1090949306	0.3833084208

Learning time

The combinations (algorithms and parameters) for best performing models by comparing learning time in seconds (can be skewed by running tests in parallel) are listed below and the results are in Table 11:

1. Tree (criterion=mse, splitter=random, max_depth=10, max_features=log2, parse_url=true, features=method,url)
2. Tree (criterion=mse, splitter=random, max_depth=10, max_features=sqrt, parse_url=true, features=method,url)
3. Tree (criterion=mse, splitter=best, max_depth=10, max_features=log2, parse_url=true, features=method,url)
4. Tree (criterion=mse, splitter=best, max_depth=10, max_features=sqrt, parse_url=true, features=method,url)
5. Tree (criterion=mse, splitter=random, max_depth=10, max_features=sqrt, parse_url=true)
6. Tree (criterion=mse, splitter=random, max_depth=10, max_features=log2, parse_url=true)
7. Tree (criterion=mse, splitter=best, max_depth=10, max_features=log2, parse_url=true)

8. Tree (criterion=mse, splitter=best, max_depth=10, max_features=sqrt, parse_url=true)
9. SGD (loss=huber, fit_intercept=1, parse_url=true, features=method,url)
10. SGD (loss=epsilon_insensitive, fit_intercept=1, parse_url=true, features=method,url)

Table 11 Top 10 for learning time of utilization by 95th percentile

Combination	Min(Best)	Max(Worst)	Average	95th percentile
1.	0.2169258595	92.31888175	0.2536585331	0.2619518519
2.	1.663407803	2.009582043	2.179697895	2.567668927
3.	1.666857243	1.666857243	2.151520324	2.60237726
4.	2.330256224	2.337308407	2.813428211	3.234726763
5.	4.367028713	4.860736847	4.804517817	5.066563129
6.	3.983752251	3.983752251	4.676143646	5.070691526
7.	4.460699797	4.794395447	4.810140109	5.093491721
8.	4.61593008	4.61593008	5.026923466	5.579508209
9.	4.548009396	6.443040848	5.767314744	6.426808226
10.	5.613236189	7.925549746	6.723724794	7.738092411

RMSE

The combinations (algorithms and parameters) for best performing models by comparing RMSE are listed below and the results are in Table 12:

1. Lasso (normalize=False, fit_intercept=False, positive=True, selection=cyclic, , parse_url=true, features=method,url)
2. Lasso (normalize=False, fit_intercept=True, positive=True, selection=cyclic, parse_url=true)
3. Lasso (normalize=True, fit_intercept=True, positive=False, selection=cyclic, parse_url=true, features=method,url)
4. Lasso (normalize=False, fit_intercept=False, positive=False, selection=cyclic, parse_url=true, features=method,url)
5. Lasso (normalize=True, fit_intercept=True, positive=True, selection=cyclic, parse_url=true, features=method,url)

6. Lasso (normalize=False, fit_intercept=False, positive=False, selection=cyclic, parse_url=true)
7. Lasso (normalize=True, fit_intercept=False, positive=True, selection=cyclic, parse_url=true)
8. Lasso (normalize=True, fit_intercept=True, positive=False, selection=cyclic, parse_url=true)
9. Lasso (normalize=True, fit_intercept=False, positive=False, selection=cyclic, parse_url=true, features=method,url)
10. Lasso (normalize=False, fit_intercept=True, positive=True, selection=cyclic, parse_url=true, features=method,url)

Table 12 Top 10 for RMSE by 95th percentile

Combination	Min(Best)	Max(Worst)	Average	95th percentile
1.	24,74199074	172,1968685	87,29209859	161,0514487
2.	24,74199074	172,1968685	87,29209859	161,0514487
3.	24,74199074	172,1968685	87,29209859	161,0514487
4.	24,74199074	172,1968685	87,29209859	161,0514487
5.	24,74199074	172,1968685	87,29209859	161,0514487
6.	24,74199074	172,1968685	87,29209859	161,0514487
7.	24,74199074	172,1968685	87,29209859	161,0514487
8.	24,74199074	172,1968685	87,29209859	161,0514487
9.	24,74199074	172,1968685	87,29209859	161,0514487
10.	24,74199074	172,1968685	87,29209859	161,0514487

R-squared

The combinations (algorithms and parameters) for best performing models by comparing R-squared are listed below and the results are in Table 13Table 12:

1. Lasso (normalize=False, fit_intercept=False, positive=True, selection=cyclic, parse_url=true, features=method,url)
2. Lasso (normalize=False, fit_intercept=True, positive=True, selection=cyclic, parse_url=true,)
3. Lasso (normalize=True, fit_intercept=True, positive=False, selection=cyclic, parse_url=true, features=method,url)

4. Lasso (normalize=False, fit_intercept=False, positive=False, selection=cyclic, parse_url=true, features=method,url)
5. Lasso (normalize=True, fit_intercept=True, positive=True, selection=cyclic, parse_url=true, features=method,url)
6. Lasso (normalize=False, fit_intercept=False, positive=False, selection=cyclic, parse_url=true)
7. Lasso (normalize=True, fit_intercept=False, positive=True, selection=cyclic, parse_url=true)
8. Lasso (normalize=True, fit_intercept=True, positive=False, selection=cyclic, parse_url=true)
9. Lasso (normalize=True, fit_intercept=False, positive=False, selection=cyclic, parse_url=true, features=method,url)
10. Lasso (normalize=False, fit_intercept=True, positive=True, selection=cyclic, parse_url=true, features=method,url)

Table 13 Top 10 for R-squared by 95th percentile

Combination	**Max(Best)**	**Min(Worst)**	**Average**	**95th percentile**
1.	0,999997318	0,999996405	0,999997044	0,999997303
2.	0,999997318	0,999996405	0,999997044	0,999997303
3.	0,999997318	0,999996405	0,999997044	0,999997303
4.	0,999997318	0,999996405	0,999997044	0,999997303
5.	0,999997318	0,999996405	0,999997044	0,999997303
6.	0,999997318	0,999996405	0,999997044	0,999997303
7.	0,999997318	0,999996405	0,999997044	0,999997303
8.	0,999997318	0,999996405	0,999997044	0,999997303
9.	0,999997318	0,999996405	0,999997044	0,999997303
10.	0,999997318	0,999996405	0,999997044	0,999997303

4.2.3 Whole dataset stage

For the last stage three combinations were selected to be executed with all nine scenarios, the decision-making process on how they were chosen is in the commentary section of this chapter. Selected combinations were:

- Lasso (normalize=False, fit_intercept=True, positive=False, selection=random, parse_url=true, features=method,url),
- Tree (a=criterion=mse, splitter=best, max_depth=10, max_features=auto, parse_url=true)
- SGD (loss=huber, fit_intercept=1, parse_url=true)

Standard deviation of utilization for least loaded using Lasso Regression report

Benchmarks for standard deviations for least loaded using Lasso Regression are in Table 14.

Table 14 The standard deviation of utilization for least loaded using Lasso Regression report

Standard deviation of utilization	**3 backend servers**	**5 backend servers**	**10 backend servers**
1000 rows	0,376742593	0,519168391	0,752015872
10000 rows	0,008480679	0,027616199	0,089010914
All data	0,004170859	0,007040613	0,016503609

Standard deviation of utilization for least loaded using Tree Regression report

Benchmarks for standard deviations for least loaded using Tree Regression are in Table 15.

Table 15 The standard deviation of utilization for least loaded using Tree Regression report

Standard deviation of utilization	**3 backend servers**	**5 backend servers**	**10 backend servers**
1000 rows	0,376655095	0,519103516	0,752192189
10000 rows	0,008366067	0,027714698	0,088838256
All data	0,004554344	0,007050729	0,016519213

Standard deviation of utilization for least loaded using Stochastic Gradient Descent report

Benchmarks for standard deviations for least loaded using Stochastic Gradient Descent are in Table 16.

Table 16 The standard deviation of utilization for least loaded using SGD report

Standard deviation of utilization	**3 backend servers**	**5 backend servers**	**10 backend servers**
1000 rows	0,367945380	0,350297538	0,752221865
10000 rows	0,149160222	0,168338797	0,087832561
All data	0,003977699	0,015026236	0,039032588

4.3 Comments

In this section I have addressed results and reports from the tests, with each stage addressed separately. I drew conclusions based on the results, but I chose not to formalize or generalize them because the tests were concluded on too small a sample (ETA-HTTP dataset).

4.3.1 Baseline and benchmark stage

As expected, both baselines did not perform well on a small subset of data (only 1000 rows), in some cases using a method with random distribution could have had better results. Other than that it has shown that an increasing number of backend servers degrades performance of the load balancing methods and shows their weaknesses.

Based on the results I have decided that any model that could produce a standard deviation of utilization below 0,1 is deemed successful. This goal was aimed at scenarios with the full dataset. But if a model can reach this with a subset of data, it is regarded as successful as well.

When comparing the round robin and least loaded (knows payload sizes in advance) methods, the second method was able to reach a much better utilization rate than the first one, showing that round robin is the simplest method and completely unaware of the domain or system.

4.3.2 Subset stage

The second stage consisted of a grid search of 181 combinations which yielded some very well performing machine learning models but also many that performed quite poorly. With most of the models learning times were acceptable, K fold validations were not exceedingly time-consuming. The only exception were the models using the Ordinary least-squared algorithm. Their average fitting time (learning) was 77 minutes per fold and such a number exceeds the acceptable limit for a possible full grid search. The full cycle of K-fold validation took about 24 hours for some of them. Surprisingly, in addition to the long fitting times, those models also performed quite poorly in comparison with other models. Overall, the worst performance was observed for the models using the Ordinary least-square algorithm.

The models utilizing the Ridge regression turned out to be average performers with no significant exceptions, either way - none of them prominently better or worse. Out of sixty different combinations, eight of them did not finish the K-fold validation. All of them ended with errors (`LinAlgError("SVD did not converge") LinAlgError: SVD did not converge`). Without any debugging tools and a deeper knowledge of Scikit and Numpy library, it was difficult to determine what the cause was. If I were to make an educated guess, I would say the problem originated in normalization or transformation with `NaN`[1] or infinite or either number overflowing.

The models based on the Lasso regression were the top performers overall and there were over 20 models with RMSE metric over 0,999. This shows that the model reflects real values very well and when applying the models in load balancing the results were stable. The grid search has shown that the Lasso regression performed extremely well and there were only minor differences.

The tree based models were also average performers, a possible explanation for which would be the depth constraint of such trees (maximal depth was 10). There were some models with exceptional performance proving that tree based models are very sensitive to grid search with selected dataset.

The models with the Stochastic gradient descent algorithm performed very similarly to the tree based ones. There were three accurate models (out of twenty-four combinations), showing that the models were sensitive to parameters and grid search.

Based on these results and after careful consideration, I selected three models for the last stage involving testing on the whole dataset. Only three models were chosen to shorten the computational time and to follow the concept of grid search, I opted for the most accurate models. The criterion was having average standard deviation of utilization below 0,2. Altogether forty-three models met this condition, but only one per each kind of algorithm was chosen. The selected models (combinations) were:

- Lasso (normalize=False, fit_intercept=True, positive=False, selection=random, parse_url=true, features=method,url),
- Tree (a=criterion=mse, splitter=best, max_depth=10, max_features=auto, parse_url=true)
- SGD (loss=huber, fit_intercept=1, parse_url=true)

[1] Not a number – numeric data type value representing an undefined or unrepresentable value

4.3.3 Whole dataset stage

The three selected models were tested against all nine scenarios defined in the Tests section. All of them have successfully completed the tests, the only issue was long parsing of URL for the whole dataset, despite limiting the depth to 8. This could be considered a flaw of the models rather than of the tests, the proper way would be to parse the URL only once and store the results for later use.

Examining the results in contrast with the first stage results, as expected the models could not provide sufficient insights with a small amount of data (1000 rows). All three models were accurate when fitted with more data and load balancing methods based on them were reaching same utilizations of backends as method with real values. In a few cases, the standard deviation of utilization was even below the one with real values, this was caused by random noise inside the dataset. For all three models average standard deviation of utilization (for scenarios with 10000 rows and whole dataset) was 0,04998. Which is below the planned 0,1 and therefore all three models are considered successful and could be applied to real systems.

5 Discussion

Both machine learning and HTTP load balancing are well-known topics with well established procedures and methods. As I have already stressed in the introductory chapter, the proper approach to machine learning should always begin with the problem. Current systems certainly incorporate HTTP load balancing methods already and while some produce more or less satisfactory results, many struggle to perform well. The main objective of this thesis was to analyze whether machine learning could be applied to HTTP load balancing. With certain constraints and limitations, it is certainly possible to do so. Within the next few paragraphs I discuss the “why” and “how” and review the partial objectives one by one.

The first partial objective was to analyze, whether the current state of machine learning can provide sufficient methods and tools in order to apply it to HTTP load balancing. In this thesis, key concepts and workflow were covered and in the subsequent chapters I shifted my focus to scaling techniques. Such scaling would be essential to the process of building a fully functioning and stable system that utilizes machine learning for HTTP load balancing. For potential applications, I would recommend building a domain specific system, which would require a deep knowledge of the system and subject matter expert. In such a system, my suggestion would be to build a model based system, where the algorithm/model is chosen first and the system built around it.

The second partial objective was to analyze the current state of HTTP load balancing methods and strategies. I have shown, that although there already are some wide spread methods available, such as round robin, there certainly are many better performing options. Similarly to machine learning, it is necessary to be domain specific, while building or using a better performing method or strategy. It is necessary to have a great understanding of the system, its dynamics and HTTP traffic, both internal and external.

The third partial objective was to design such a load balancing method based on machine learning. My approach was to use the least loaded method with metric, the values of which would be predicted by the machine learning model. It was necessary to find the right correct algorithm and model. This part merged also into the last objective (testing), because machine learning workflow is iterative. The method was described and well formalized so it could be compared to other methods and to the same method using different machine learning model.

The last partial objective was to conduct tests for the created method with machine learning models and to write a manual with steps how to reproduce the tests in the same environment and the same results as found in this thesis. The tests were done in three stages, where first stage created baselines for later comparisons, the second stage consisted of a grid search to find the most accurate machine learning models and the last stage was about testing load balancing methods with selected models. All three stages have provided results and are discussed in the Comments section. The steps and setup for tests were described and readers should be able to recreate the tests or continue with further testing.

While having acceptable test results, this method may be labeled as flawed. There are several flaws which were introduced by constraints, limits or by test design. The whole method should be taken more as an approach or a course, which should be taken further if decided. The main flaw could be that response payload size probably would not be used as such metric to measure utilization of system. In the real world commonly used metrics are, for example, CPU utilization, system load (Gregg, 2013, pp. 34-35) or system specific metric. The second flaw is that the modified least loaded method does not employ rolling windows as counters, which makes perfect sense for isolated systems such as the one simulated in the tests. The system is isolated from the concept of time and so it is possible to conduct the tests within a reasonable timeframe. Having said that, the tests could be improved to use leaky bucket or token bucket as a counter. It definitely needs further testing, with real systems and different scenarios. This kind of advanced testing was out of the scope of this thesis. It would require further research on how such system would behave under different types of load (peaks, high load, slowly increasing load) and whether the machine learning model used with method would misbehave in any sort of way. Such behavior would be a huge argument against the possible applications.

Nevertheless, it is safe to say that the HTTP protocol can be very well described with features. Hierarchic structure or URL plays a big role in this, but requires a well-designed API of the system. Only possible issue with the URL as a feature variable are changing parts of URL, such as object identifiers. HTTP method is also a feature with a very good predictive value. The predictive value of HTTP traffic is highly dependent on the system, therefore I cannot exclude the possibility that my results are based on data with a bias.

For real world applications, it is clearly necessary to design and build the system around the machine learning model in order for it to work and scale properly. Such system would be required to make predictions within few milliseconds or provide fallback values. It would be recommended to employ the best practices of software engineering to make such a system reliable and stable. Patterns such as circuit breaker or caching should be used within the system. The designed system could greatly benefit from online learning, in which any HTTP traffic would be added to historic data and machine learning model could be adjusted in real time. For online learning, tree based models are recommended.

If evaluating the designed method objectively, either by adding more criteria or by removing any constraints and limitations, the benefits of this method would be outweighed by the cons and costs of the application. It may require further research and exploration whether using machine learning based methods for load balancing is too much of a complex idea and whethere or not it would be counterproductive in any way. But on the other hand, I have shown that it would be possible to design ranking or scoring models of HTTP traffic based on utilization metrics. Rate limiting of traffic could benefit from this area of research.

5.1 Objective Summary

With the general objective fulfilled, it is appropriate to give a summary of the four partial objectives of this thesis:

- It was shown that machine learning possesses the ability and options to be applied to HTTP load balancing, if the right model is chosen and scaling is conducted properly.
- The HTTP load balancing strategies and methods were described, with example implementations and their usages.
- Load balancing method with the least loaded approach using predicted HTTP response sizes as key metric was described and implemented.
- System using designed load balancing method was created and tested with a real world dataset containing HTTP traffic, with mixed results.

Attachment A: Source codes

The first attachment consists of source codes for scripts simulating a load balancing system and conducting tests. The scripts are implemented in Python and BASH. The source codes are available in file a titled *sources_xsykj20_diploma_thesis_2017.zip*, which is copy of my GitHub repository (https://github.com/jansyk13/limython). There is a branch name finished, with a submitted version the thesis. The structure of source codes is as follows:

- **src/limython.py** – main Python wrapper for simulating the system and running tests
- **src/baseline.py** – a Python wrapper to measure baselines
- **src/final.py** – a Python wrapper for running tests in final stage
- **src/utils** – a folder with utility tools
- **src/learning** – a folder with machine learning models
- **src/transformation** – a folder with data transformation tools
- **src/utils** – a folder with utility tools
- **src/validation** – a folder with validation tools
- **results** – a folder with log files from tests
- **test-data** – a folder containing testing dataset, a Python script for parsing and SQL script with INSERT statements
- **results_to_csv.py** – a Python script for parsing log files into csv format
- **ols.sh** – a BASH script for running tests with Ordinary least-squared models
- **lasso.sh** – a BASH script for running tests with Lasso models
- **ridge.sh** – a BASH script for running tests with Ordinary Ridge models
- **tree.sh** – a BASH script for running tests with Tree models
- **sgd.sh** – a BASH script for running tests with Stochastic gradient descent models
- **final_cut.sh** – a BASH script for running tests for final stage

Attachment B: Results

One of the attachments is a XLSX file (*results_xsykj20_diploma_thesis_2017.xlxs*) with results from conducted tests; the file is systematically structured. This file is also available at my GitHub repository (https://github.com/jansyk13/limython). There are 5 five sheets, each containing different data:

- **LB stdev utilization** – contains data from first and last stages, with utilizations of backend servers
- **GRID SEARCH stdev utilization** – contains standard deviations of utilizations collected in grid search
- **GRID SEARCH timer** – contains learning times collected in grid search
- **GRID SEARCH RMSE** – contains RMSE values collected in grid search
- **GRID SEARCH R^2** – contains R-squared values collected in grid search

Attachment C: Manual for scripts and tests

Limython

The repository containing the source code for my diploma thesis at University of Economics, Prague. The topic is 'An Analysis of Potential Applications of Machine Learning in HTTP Load-balancing'. The repository contains Python and BASH wrappers for machine learning and simulating load balancing system. The full text of the thesis can be found in the repository as well.

Guide

Can be run as Python script with several mandatory parameters(`limython.py`):

```
python src/limython.py -p=best-counter -l=100 -n=3 -k=2 -r=false
```

To get baselines of data use `baseline.py` script:

```
python src/baseline.py -n=3 -p=best-counter -k=10 -r=true -l=1000
```

To run final test use `final.py` script:

```
python src/final.py -p=best-counter -l=50000 -m=sgd -a=loss=huber,fit_inter-
cept=1 -u=true -ul=8
```

Parsing log files to csv format

Run `result_to_csv.py` script with parameters `stdev|timer|rmse|rsquared|counter` and path `/results/` and then forward standard output to file.

Database

Database should be running on `localhost` with port `3306`. Current setup uses `root` as user and `password` as password, but it is easy to change by editing scripts. Database schema should be named `mlrl` and table named `data`.

Setup

- Install Fortran, C and C++ compilers (`yum install gcc-gfortran gcc gcc-c++`)
- Install MySQL server, command line client and other related libraries (`yum isntall mysql mysql-server mysql-devel mysql-lib`)
- Install BLAS prerequisites (`yum install automake autoconf libtool*`)
- Install BLAS (`yum install lapack-devel blas-devel atlas-sse3-devel`)
- Install Python 3.5 and virtualenv (`yum install python35 python35-virtualenv python35-pip`)
- Create and source new environment (`virtualenv-3.5 ml; source ml/bin/activate`)
- Install Cython via Pip (`pip install cython`)
- Install required dependencies in source code folder (pip install -r requirements.txt)

Note: Installation via Pip may cause compilation of several libraries into C and C++ and it can make it very slow(verbose mode recommended). Other issue is that Pip may not respect dependencies between libraries in requirements.txt(install separately).

Data

Using test data from `http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html`.

BLAS

- See configuration with NumPy `python -c "import numpy; numpy.show_config()"`
- See configuration with SciPy `python -c "import scipy; scipy.show_config()"`
- Make sure BLAS is set up correctly by setting to environment variables:
  - `export MKL_NUM_THREADS=32`
  - `export OPENBLAS_NUM_THREADS=32`
  - `export NUMEXPR_NUM_THREADS=32`

Help limython.py

```
usage: limython.py [-h] -p PROCESSOR -n NODE_COUNT -l LIMIT [-k K_FOLDS] -m
                   MODEL [-a ARGUMENTS] [-u URL] [-ul URL_LIMIT] [-f FEATURES]
                   -r RUN

Python wrapper to simulate load balancing system and to run tests.

optional arguments:
  -h, --help            show this help message and exit
  -p PROCESSOR, --processor PROCESSOR
                        Request processor(simple-round-robin,best-counter)
  -n NODE_COUNT, --node-count NODE_COUNT
                        Node count for processor(1,2,3,...)
  -l LIMIT, --limit LIMIT
                        Data limit(to avoid running out of memory)
  -k K_FOLDS, --k-folds K_FOLDS
                        Number of folds for cross validation(higher better,
                        but computation more expensive)
  -m MODEL, --model MODEL
                        ML model(ols, lasso, ridge, sgd, tree)
  -a ARGUMENTS, --arguments ARGUMENTS
                        ML model arguments - kwargs separated with comma
  -u URL, --url URL     Flag whether url should be parsed into tree like
                        indicators
  -ul URL_LIMIT, --url-limit URL_LIMIT
                        Limit depth of tree hierarchy of dummy variable parsed
                        from urls
  -f FEATURES, --features FEATURES
                        List of features - comma separated
  -r RUN, --run RUN     'true' or 'false' whether to run full with processor
```

Help baseline.py

```
usage: baseline.py [-h] -p PROCESSOR -n NODE_COUNT -l LIMIT [-k K_FOLDS] -r
                   RUN

Python wrapper to simulate load balancing system and to create baselines.

optional arguments:
  -h, --help            show this help message and exit
  -p PROCESSOR, --processor PROCESSOR
                        Request processor(simple-round-robin,best-counter)
  -n NODE_COUNT, --node-count NODE_COUNT
                        Node count for processor(1,2,3,...)
  -l LIMIT, --limit LIMIT
                        Data limit(to avoid running out of memory)
  -k K_FOLDS, --k-folds K_FOLDS
                        Number of folds for cross validation(higher better,
                        but computation more expensive)
  -r RUN, --run RUN     'true' or 'false' whether to run full with processor
```

Help final.py

```
usage: final.py [-h] -p PROCESSOR -l LIMIT -m MODEL [-a ARGUMENTS] [-u URL]
                [-ul URL_LIMIT] [-f FEATURES]
```

```
Python wrapper to simulate load balancing system and to run tests for final
stage.

optional arguments:
  -h, --help            show this help message and exit
  -p PROCESSOR, --processor PROCESSOR
                        Request processor(simple-round-robin,best-counter)
  -l LIMIT, --limit LIMIT
                        Data limit(to avoid running out of memory)
  -m MODEL, --model MODEL
                        ML model(ols, lasso, ridge, sgd, tree)
  -a ARGUMENTS, --arguments ARGUMENTS
                        ML model arguments - kwargs separated with comma
  -u URL, --url URL     Flag whether url should be parsed into tree like
                        indicators
  -ul URL_LIMIT, --url-limit URL_LIMIT
                        Limit depth of tree hierarchy of dummy variable parsed
                        from urls
  -f FEATURES, --features FEATURES
                        List of features - comma separated
```

Glossary

Term	Definition (Reference)
machine learning	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. (Mitchell, 1997, p. 2)
linear programming	The problem of maximizing or minimizing a linear function over a convex polyhedron specified by linear and non-negativity constraints. (Wolfram, 2017)
regression	A process of learning real-valued function from training examples labelled with true function values. (Flach, 2012, p. 14)
computational complexity	A branch of the theory of computation in theoretical computer science that focuses on classifying computational problems according to their inherent difficulty, and relating those classes to each other. (Wikipedia, 2005)
BLAS	Specification for a set of low-level routines for performing linear algebra operations. (Wikipedia, 2006)
target variable	The numerical or categorical (label) attribute of interest. This is the variable to be predicted for each new instance. (Brink, et al., 2016, p. 25)
feature variable	The variable describing relevant objects in a domain. (Flach, 2012. p. 13)
metric	A standard of measurement. (Merriam-Webster)
HTTP load balancing	A process and technology that distributes traffic among several servers. (Bourke, 2001, p. 3)
LIFO	Abbreviation for Last In First Out: a stack is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed. (Wikipedia, 2004)
FIFO	Abbreviation for First In First Out: a method for organizing and manipulating a data buffer, where the oldest (first) entry, or 'head' of the queue, is processed first. (Wikipedia, 2005)
PIFO	Abbreviation for Push In First Out: a priority queue data structure (Sivaramam, et al., 2016)
HTTP	The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. (The Internet Engineering Task Force, 1999)
SFTP	The SSH File Transfer Protocol or SFTP is a network protocol that provides file transfer and manipulation functionality over any reliable data stream(SSH). (Wise-ftp)
FTPS	FTPS is a protocol for secure file transfer over FTP. (The Internet Engineering Task Force, 2005)

Term	**Definition (Reference)**
FTP	File Transfer Protocol is a protocol for file transfers between HOSTs on the ARPANET. (The Internet Engineering Task Force, 1985)
WebDAV	Web Distributed Authoring and Versioning (WebDAV) is a protocol consisting of a set of methods, headers, and content-types ancillary to HTTP/1.1 for the management of resource properties, creation and management of resource collections, URL namespace manipulation, and resource locking (collision avoidance). (The Internet Engineering Task Force, 2007)
IRC	Abbreviation for Internet Relay Chat: a protocol for text based conferencing. (The Internet Engineering Task Force, 2000)
IP address	Abbreviation for Internet Protocol Address: a number that is given to each computer when it is connected to the internet. (Cambridge Dictionary)
DNS	Abbreviation for Domain Name System: a system associating IP addresses with domain names. (Bourke, 2001, p. 6)
OSI layer model	Framework for developing protocols and applications that could interact seamlessly. (Bourke, 2001, pp. 13-15)
URL	URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet. (Oracle, 2015)
CAP theorem	The theorem states that, in a distributed system, you can only have two out of the following three guarantees across a write/read pair: Consistency, Availability and Partition Tolerance – one of them must be sacrificed. (Greiner, 2014)
"no free lunch" theorem	The theorem states that there is no one model that works best for every problem. (Cai, 2014)
scaling	A process of increasing performance of a system, either by creating a larger system or spreading the load across numerous systems. (Gregg, 2013, p. 69)
AWS	Web service that provides anything as a service. (Amazon Web Services, 2017)
EC2	Web service that provides a secure, resizable compute capacity in the cloud. (Amazon Web Services, 2017)
HDFS	Abbreviation for Hadoop Distributed File System: a fault-tolerant distributed filesystem. (Sjardin, et al., 2016, pp. 308-309)
rolling window	A data structure which can be used to track counters or set of values over time. (Netflix, 2012)
leaky bucket	An algorithm that may be used to determine whether a certain sequence of discrete events conforms to the defined limits on their average and peak rates or frequencies. (Wikipedia, 2007)
token bucket	An algorithm that consists of a bucket with a maximum capacity of N tokens which refills at a rate R of tokens per second. (Thomas, 2007)

Term	**Definition (Reference)**
API	Abbreviation for Application programming interface: a set of subroutine definitions, protocols, and tools for building application software. (Wikipedia, 2003)
circuit breaker	A software engineering technique of wrapping a dangerous operation with a component that can circumvent calls when the system is not healthy. (Nygard, 2007, pp. 93-95)

Reference

AMAZON WEB SERVICES. 2017. Amazon EC2. [Online] [Cited: 4 17, 2007.] Available at: https://aws.amazon.com/ec2/.

BERENBRIK, Petra, HOEFER, Martin and SAUERWALD, Thomas. 2014. Distributed Selfish Load Balancing on Networks. *ACM Transactions on Algorithms (TALG).*, Vol. 11, 1.

BEYER, Betsy, JONES, Chris, PETOFF, Jennifer and MURPHY, Niall Richard. 2016. *Site Reliability Engineering : How Google Runs Production Systems.* Sebastopol, United States : O'Reilly Media, Inc, USA. ISBN13 9781491929124.

BOURKE, Tony. 2001. *Server Load Balancing.* Sebastopol, United States : O'Reilly Media, Inc, USA. ISBN13 9780596000509.

BRINK, Henrick, RICHARDS, Joseph W. and FETHEROLF, Mark. 2016. *Real-World Machine Learning.* New York, United States : Manning Publications. p. 210. ISBN13 9781617291920.

CAI, Eric. 2014. Machine Learning Lesson of the Day – The "No Free Lunch" Theorem. *The Chemical Statistician.* [Online] [Cited: 4 17, 2017.] Available at: https://chemicalstatistician.wordpress.com/2014/01/24/machine-learning-lesson-of-the-day-the-no-free-lunch-theorem/.

CAMBRIDGE DICTIONARY. Meaning of "IP address" in the English Dictionary. [Online] [Cited: 4 17, 2017.] Available at: http://dictionary.cambridge.org/dictionary/english/ip-address.

FLACH, Peter. 2012. *Machine Learning : The Art and Science of Algorithms That Make Sense of Data.* Cambridge, United Kingdom : Cambridge University Press. ISBN13 9781107422223.

GARTNER. 2016. Gartner's 2016 Hype Cycle for Emerging Technologies Identifies Three Key Trends That Organizations Must Track to Gain Competitive Advantage. *Gartner Newsroom.* [Online] 8 16, 2016. [Cited: 4 13, 2017.] Available at: http://www.gartner.com/newsroom/id/3412017.

GREGG, Brendan. 2013. *Systems Performance : Enterprise and the Cloud.* Upper Saddle River, United States : Pearson Education (US). ISBN13 9780133390094.

GREINER, Robert. 2014. CAP Theorem: Revisited. [Online] [Cited: 4 17, 2017.] Available at: http://robertgreiner.com/2014/08/cap-theorem-revisited/.

GUPTA, Sumit. 2015. *Learning Real-Time Processing with Spark Streaming.* Birmingham, United Kingdom : Packt Publishing Limited. ISBN13 9781783987665.

HARRINGTON, Peter. 2012. *Machine Learning in Action.* Shelter Island, N.Y., United Kingdom : Manning Publications. ISBN13 9781617290183.

KAGGLE. 2014. Dogs vs. Cats. [Online] [Cited: 2 12, 2017.] Available at: https://www.kaggle.com/c/dogs-vs-cats.

KOŘENÁŘ, Václav. 2010. *Stochastické modely.* Prague : Nakladatelství Oeconomica. ISBN 9788024516462.

LAGOVÁ, Milada. 2014. *Lineární modely.* Praha : Nakladatelství Oeconomica. ISBN 9788024520209.

LANEY, Doug. 2001. 3D Data Management: Controlling Data Volume, Velocity, and Variety. [Online] [Cited: 4 17, 2017.] Available at: https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.

LITTLE, James, PLUGGE, Eelco, MEMBREY, Peter and HOWS, David. 2012. *Practical Load Balancing: Ride the Performance Tiger.* Berkley, United States : aPress. ISBN13 9781430236801.

MITCHELL, Tom M. 1997**.** *Machine Learning.* s.l. : McGraw-Hill Science/Engineer. ISBN13 9780070428072.

MERRIAM-WEBSTER. Definition of metric. [Online] [Cited: 4 18, 2017.] Available at: https://www.merriam-webster.com/dictionary/metric.

NETFLIX. 2012. HystrixRollingNumber. [Online] [Cited: 4 19, 2017.] Available at: https://github.com/Netflix/Hystrix/blob/master/hystrix-core/src/main/java/com/netflix/hystrix/util/HystrixRollingNumber.java.

NGINX. Session persistence. *Using nginx as HTTP load balancer.* [Online] [Cited: 4 17, 2017.] Available at: http://nginx.org/en/docs/http/load_balancing.html#nginx_load_balancing_with_ip_hash.

NIDL, Michal. 2016. Methodology of optimal usage of load balancing in data center environment. Prague.

NUMPY. 2017. Fundamental package for scientific computing with Python. [Online] [Cited: 4 16, 2017.] Available at: http://www.numpy.org.

NYGARD, Michael T. 2007. *Release It! : Design and Deploy Production-ready Software.* Raleigh, United States : The Pragmatic Programmers. ISBN13 9780978739218.

ORACLE. 2015 What Is a URL? *The Java™ Tutorials.* [Online] [Cited: 4 17, 2017.] Available at: https://docs.oracle.com/javase/tutorial/networking/urls/definition.html.

PANDAS. 2016. Python Data Analysis Library. [Online] [Cited: 4 16, 2017.] Available at: http://pandas.pydata.org/.

PENROSE, Roger. 1955. A generalized inverse for matrices. *Mathematical Proceedings of the Cambridge Philosophical Society.* 1955, Vol. 51, 3, pp. 406-413.

PYTHON SOFTWARE FOUNDATIOM. 2017. Configuration file parser. [Online] [Cited: 16 4, 2017.] Available at: https://docs.python.org/3/library/configparser.html.

—. 2017. Pytest: simple powerful testing with Python. [Online] [Cited: 4 16, 2017.] Available at: https://pypi.python.org/pypi/pytest.

—. 2017. Python interface to MySQL. [Online] [Cited: 4 16, 2017.] Available at: https://pypi.python.org/pypi/mysqlclient.

PRAJER, Richard. 2016. On possible approaches to detecting robotic activity of botnets. Prague.

SCIKIT-LEARN. 2016. Underfitting vs. Overfitting. [Online] 2016. [Cited: 4 14, 2017.] Available at: http://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html.

—.2016. Decision Tree Regression. [Online] [Cited: 4 15, 2017.] Available at: http://scikit-learn.org/stable/auto_examples/tree/plot_tree_regression.html.

—. 2016. Scikit-learn Machine Learning in Python. [Online] [Cited: 4 16, 2017.] Available at: http://scikit-learn.org/stable/.

—. 2016. sklearn.linear_model.Lasso. [Online] [Cited: 4 15, 2017.] Available at: http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html.

—. 2016. sklearn.linear_model.LinearRegression. [Online] [Cited: 4 15, 2017.] Available at: http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html.

—. 2016. sklearn.linear_model.Ridge. [Online] [Cited: 4 15, 2017.] Available at: http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html.

—. 2016. sklearn.linear_model.SGDRegressor. [Online] [Cited: 4 15, 2017.] Available at: http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html.

SCIPY. 2017. Scientific Computing Tools for Python. [Online] [Cited: 4 16, 2017.] Available at: https://www.scipy.org/.

SIVARAMAN, Anirudh, SUBRAMANIAN, Suvinay, ALIZADEH, Mohammad, CHOLE, Sharad, CHUANG, Shang-Tse, AGRAWAL, Anurag, BALAKRISHNAN, Hari, EDSALL, Tom, KATTI, Sachin and MCKEOWN, Nick. 2016. Programmable Packet Scheduling at Line Rate. [Online] [Cited: 4 18, 2017.] Available at: http://web.mit.edu/pifo/.

SJARDIN, Bastiaan, MASSARON, Luca and BOSCHETTI, Alberto. 2016. *Large Scale Machine Learning with Python.* Birmingham, United Kingdom : Packt Publishing Limited.. ISBN13 9781785887215.

STATSMODELS. 2012. Statisitcs in Python. [Online] [Cited: 4 16, 2017.] Available at: http://statsmodels.sourceforge.net/.

THE INTERNET TRAFFIC ARCHIVE. 1995. EPA-HTTP. [Online] [Cited: 4 17, 20017]. Available at: http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html

THE INTERNET ENGINEERING TASK FORCE. 1985 File Transfer Protocol (FTP). [Online] [Cited: 4 17, 2017.] Available at: https://www.ietf.org/rfc/rfc959.txt.

—. 2007. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). [Online] [Cited: 4 17, 2017.] https://www.ietf.org/rfc/rfc4918.txt.

—. 1999. Hypertext Transfer Protocol -- HTTP/1.1. [Online] [Cited: 4 17, 2017.] Available at: https://www.ietf.org/rfc/rfc2616.txt.

—. 2000. Internet Relay Chat: Client Protocol. [Online] [Cited: 4 18, 2017.] Available at: https://tools.ietf.org/html/rfc2812.

—. 2005. Securing FTP with TLS. [Online] [Cited: 4 17, 2017.] Available at: https://tools.ietf.org/html/rfc4217.

THOMAS, Alec. 2007. Implementation of the token bucket algorithm (Python recipe) . [Online] [Cited: 4 19, 2017.] Available at: http://code.activestate.com/recipes/511490-implementation-of-the-token-bucket-algorithm/.

VRÁNOVÁ, Jana. 2009. ROC Analysis and The Use of Cost – Benefit Analysis For Determination of the Optimal Cut Point. *Prolekare.cz.* [Online] [Cited: 4 14, 2017.] Available at: http://www.prolekare.cz/en/journal-of-czech-physicians-article/roc-analysis-and-the-use-of-cost-benefit-analysis-for-determination-of-the-optimal-cut-point-5403?confirm_rules=1.

WIKIPEDIA. 2006. Basic Linear Algebra Subprograms. *Wikipedia, the free Encyclopedia.* [Online] [Cited: 4 16, 2017.] Available at: https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms.

—. 2003. Application programming interface. *Wikipedia, The Free Encyclopedia.* [Online] [Cited: 4 18, 2017.] Available at: https://en.wikipedia.org/wiki/Application_programming_interface.

—. 2003. Computational complexity theory. *Wikipedia, The Free Encyclopedia.* [Online] [Cited: 4 18, 2017.] Available at: https://en.wikipedia.org/wiki/Computational_complexity_theory#Defining_complexity_classes.

—. 2005. FIFO (computing and electronics). *Wikipedia, The Free Encyclopedia.* [Online] [Cited: 4 18, 2017.] Available at: https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics).

—. 2007. Leaky bucket. *Wikipedia, The Free Encyclopedia.* [Online] [Cited: 4 19, 2017.] Available at: https://en.wikipedia.org/wiki/Leaky_bucket.

—. 2004. Stack (abstract data type). *Wikipedia, The Free Encyclopedia.* [Online] [Cited: 4 18, 2017.] Available at: https://en.wikipedia.org/wiki/Stack_(abstract_data_type).

WISE-FTP. What is SFTP? [Online] [Cited: 4 17, 2017.] Available at: https://www.wise-ftp.com/documentation/webhelp/html/what_is_sftp_en.htm.

WOLFRAM. 2017. Linear Programming. *Wolfram Math World.* [Online] [Cited: 4 18, 2017.] Available at: http://mathworld.wolfram.com/LinearProgramming.html.

WOLPERT, David H. and MACREADY, Wiliam G. 1997. No Free Lunch Theorems for Optimization. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION,*. 1997, Vol. 1, 1.