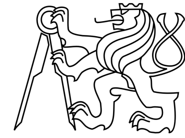
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER’S THESIS

Title:	Scalability of Predictive Modeling Algorithms
Student:	Bc. Tomáš Frýda
Supervisor:	Ing. Pavel Kordík, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

Explore predictive modeling algorithms and techniques for optimization of their parameters. Design a methodology allowing us to evaluate generalization abilities of algorithms with anytime properties. Extend the H2O open source framework with training algorithms from FAKE GAME project and compare the accuracy with deep learning and decision forest models. Explain the effect of meta-optimization techniques.

References

http://www.h2o.ai/
http://fakegame.sourceforge.net/doku.php

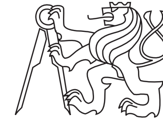
L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 20, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of theoretical computer science



Master's thesis

Scalability of Predictive Modeling Algorithms

Bc. Tomáš Frýda

Supervisor: Ing. Pavel Kordík, Ph.D.

9th January 2017

Acknowledgements

I would like to express my gratitude and appreciation to my supervisor Ing. Pavel Kordík, Ph.D., and to my parents for their support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 9th January 2017 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2017 Tomáš Frýda. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Frýda, Tomáš. *Scalability of Predictive Modeling Algorithms.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Tato práce má dva hlavní cíle — (1) paralelizovat FAKE GAME integrací do open source frameworku H2O, zaměřeného na strojové učení, a (2) hodnocení anytime vlastností algoritmů strojového učení a vlivu optimalizace hyper-parametrů na tyto algoritmy. Tyto cíle jsem realizoval integrací FAKE GAME do H2O. Za účelem vyhodnocení anytime vlastností jsem implementoval nový nástroj nazvaný Benchmarker.

Vyhodnocení anytime vlastností ukázalo, že pro některé problémy modely z FAKE GAME překonají modely z H2O, jak v přesnosti, tak i ve výkonu. Na druhou stranu vyhodnocení vlivu optimalizace hyper-parametrů ukázalo poměrně malý úspěch při optimalizaci algoritmů strojového učení z H2O.

Domnívám se, že zanedbatelné zvýšení výkonnosti, a pro některé z optimalizovaných modelů dokonce i nižší výkon než u výchozí konfigurace, je způsobeno automatickým laděním některých hyper-parametrů, které se provádí ve výchozím nastavení H2O.

Klíčová slova Anytime učení, FAKE GAME, H2O, Optimalizace hyper-parametrů, Strojové učení

Abstract

This thesis has two main goals — (1) parallelize FAKE GAME by integration into, an open source machine learning framework, H2O, and (2) evaluation of anytime properties of machine learning algorithms and influence of hyper-parameter optimization on them. To meet these objectives, I have integrated FAKE GAME into H2O and, in order to evaluate anytime properties, I have implemented, a new tool, called Benchmarker.

The evaluation of anytime properties shows that for some problems FAKE GAME models outperform state-of-the-art models from H2O, in both, accuracy and performance. Moreover, the evaluation of hyper-parameter optimization show little success, when optimizing H2O machine learning algorithms.

I hypothesise that the negligible performance improvement, and for some optimized models even lower performance than with default configuration, is caused by hyper-parameter automatic tuning, which is done by default in H2O for some hyper-parameters.

Keywords Anytime learning, FAKE GAME, H2O, Hyper-parameter Optimization, Machine Learning

Contents

Introduction **1**
Related Work 2

1 Theoretical Background **5**
1.1 FAKE GAME 5
1.2 Meta-learning Templates 8
1.3 H2O 11
1.4 H2O Ensemble 22
1.5 Hyper-Parameter Optimization 23

2 Analysis and design **35**
2.1 Integration of FAKE GAME into H2O 35
2.2 Benchmarker 38

3 Methodology **45**
3.1 Evaluating Anytime Properties 45
3.2 Evaluating Hyper-parameter Optimization 46
3.3 Datasets 46

4 Results **49**
4.1 Anytime Learning 49
4.2 Hyper-parameter optimization 59

5 Discussion **63**
5.1 Airline Dataset and Sigmoid-based Ensembles 63
5.2 Hyper-parameter Optimization 64

Conclusion **67**

Bibliography **69**

A Acronyms **77**

B Parameter Ranges for Hyper-parameter Optimization **79**

C Contents of enclosed DVD **83**

List of Figures

1.1 An example of hierarchical combination of algorithms 9
1.2 Template execution 10
1.3 Nested ensembles 10
1.4 Ilustration of a single neuron and multi-layer feed-forward artificial neural network 13
1.5 Grid and random search 24
1.6 Visualization of acquisition functions 29

2.1 A scheme of FAKE GAME model parallelization 36

4.1 Comparison of several machine learning algorithms in H2O — Higgs 50
4.2 Comparison of machine learning algorithms in H2O — Higgs . . . 51
4.3 Predicting IsDepDelayed on Airline dataset: comparison of algorithms in H2O trained on subsamples of increasing size. 52
4.4 Predicting IsDepDelayed on Airline dataset with DepTime: comparison of algorithms in H2O trained on subsamples of increasing size. 53
4.5 Comparison of machine learning algorithms in H2O — Airline (Predicting IsDepDelayed with DepTime) 54
4.6 Predicting IsArrDelayed on Airline dataset: comparison of algorithms in H2O trained on subsamples of increasing size. 55
4.7 Comparison of machine learning algorithms in H2O — Airline (Predicting IsArrDelayed with DepTime) 56
4.8 Predicting IsArrDelayed on Airline dataset: comparison of algorithms in H2O trained on subsamples of increasing size. 57
4.9 Comparison of machine learning algorithms in H2O — Airline (Predicting IsArrDelayed without DepTime) 58
4.10 Hyper-parameter optimization — Higgs dataset 59
4.11 Hyper-parameter optimization — Higgs dataset — time 60
4.12 Hyper-parameter optimization — Airline dataset 61

4.13 Hyper-parameter optimization — Airline dataset — time 62

5.1 Decision boundaries of models trained on Airline dataset 65

List of Tables

1.1 Activation functions usable in H2O 13
1.2 Loss Functions 14

B.1 Deep Learning 79
B.2 Distributed Random Forest 80
B.3 Gradient Boosting Machine 81
B.4 Generalized Linear Model 81
B.5 ClassifierBagging{$m\times$ ClassifierBoosting{$n\times$ DecisionTree}} . . . 82
B.6 CascadeGenProb{$m\times$ Boosting{$n\times$ ClassifierModel{<outputs> $\times$ SigmoidNorm}}} 82

Introduction

Artificial intelligence (AI) has become omnipresent in our daily lives. This thesis focuses on one subfield of AI called machine learning — specifically supervised learning. Supervised learning is a task of creating a predictive model based upon set of observations and corresponding responses. The predictive model should be able to generalize from the given data.

Model selection is one of the important tasks in machine learning. Automation of model selection is one of the things Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution (FAKE GAME) aims to solve. However, Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution (FAKE GAME) is not ready for the Big Data era, as it was created to deal with smaller amounts of data. The first task of this thesis was to enable running FAKE GAME on bigger data. This is accomplished by using a parallelization technique called MapReduce [1]. In order to make the usage easier I have integrated FAKE GAME in, an open source framework, H2O. This enables it to use preprocessing and parallel data load as well as many other features already implemented in H2O. One additional benefit is that it can be used easily from R, Python, Java, and it can also be used through H2O's RESTful API from different languages.

Another topic of this thesis is an evaluation of anytime properties. There isn't one canonical definition of anytime learning. It is often studied in context of reinforcement learning, however, in this thesis, I study anytime properties of supervised machine learning algorithms implemented in H2O. Unfortunately, at the time of writing this thesis, H2O was incapable of true anytime learning, to evaluate it I emulate it. This is described in Section 3.1.

Last but not least, I have explored techniques for hyper-parameter optimization. Parameters of machine learning algorithms are quite important, many of them have direct impact on the plasticity of generated predictive models [2]. Unfortunately, machine learning algorithms are often evaluated with their default parameter settings only. There are many ways of optimizing parameters of machine learning algorithms. The most common is a grid

search and recently a random search. However, recent studies showed the potential of Bayesian methods [3] outperforming both random search [4] and grid search [5].

Related Work

Generalization of the task I was given, i.e., evaluation of anytime properties and hyper-parameter optimization is meta-learning. Meta-learning is a field of study that aims to automate model selection by learning how to select a model from previous experiments.

Several major research projects have targeted meta-learning [6]. *ESPRIT Statlog* [7] compared the performance of numerous classification algorithms on several real-world datasets. Ranking of algorithms can be obtained by fast pairwise comparisons [8] just on the most useful cross-validation tests [9]. Another project was *METALA* [10], an agent-based distributed data mining system, supported by meta-learning. Again, the goal was to select from among available data mining algorithms the one producing models with the best generalization performance for the given data.

Individual models, ensembles and combination of ensembles in time series forecasting can be selected adaptively [11] by meta-learning. In [12] so called lazy meta-learning is applied to create customized ensembles on demand.

Another meta-learning approach is to use a *knowledge base* containing information that might help to improve the performance of learning. One possible approach [13] is to describe the dataset by a vector of meta-features and based on this vector, select the best performing algorithm for the nearest dataset from the knowledge base.

The *CASH* approach [14] combines selection and hyper-parameter optimization [3] of classification algorithms. Similarly to my experiments, CASH approach uses Bayesian optimization, but it adds the selection of a model to it.

Implementation part of my thesis consist of parallelization of FAKE GAME and creation of benchmarking environment. There are several options usable for Big Data. The most popular open source options are Hadoop, Storm, Spark, H2O, and Flink. Hadoop, Spark and H2O are build primarily for batch processing, whereas Storm and Flink are primarily used for stream processing. However, Spark can emulate stream processing by using mini-batches, and similarly Flink is capable of batch processing. My task was to parallelize FAKE GAME and since FAKE GAME's model training is computationally expensive I have chosen a batch processing as a way to go.

Hadoop, Spark, and H2O, are based on MapReduce [1]. Hadoop and Spark are general frameworks that are widely used with Big Data, not just for model training, but for ETL and other data intensive jobs. On the other hand, H2O is a framework that focuses on machine learning and as such it provides more

support for creating new models. Using H2O results in a more user-friendly experience without much effort, e.g., automatically generated binding for R, Python, and Java, etc. This was the main reason for choosing H2O.

Chapter **1**

Theoretical Background

1.1 FAKE GAME

Disclaimer: This whole section is strongly influenced by [2] of which I am a co-author.

Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution (FAKE GAME) is a system introduced in [15]. FAKE GAME is a Java based machine learning package that is focused on supervised algorithms. It can be used to train simple models, ensembles of the models, and evolve ensembles by the use of genetic programming.

FAKE GAME builds ensembles from fast weak learners [16]. Many of FAKE GAME's base models resemble neurons with different activation function. FAKE GAME can build both classification and regression ensembles. In this thesis, I focus on classification tasks only, however regression models can also be present in classification ensembles.

The classification task itself can be decomposed into subproblems by separation of single classes from the others. These binary problems can be solved by regression models — by estimating continuous class probabilities. Hence, FAKE GAME uses regression models and meta-models. The classifier consisting of regression models is referred to as **ClassifierModel**.

1.1.1 Base Algorithms

Training regression models is fast and straightforward. FAKE GAME uses several activation functions, namely, **Sigmoid, SigmoidNorm, Sine, Polynomial, Gaussian, Exponential** and **Linear**.

To train coefficients of linear or polynomial models, the General Least Squares method [17] is applied. For models non-linear in their coefficients, an iterative optimization process is needed.

As an example, training of Gaussian models is shown below. The output of the model y_j for jth instance with target variable d_j and input vector $\vec{x}_j$

can be computed as

$$y_j = (1 + a_{n+1}) e^{\overbrace{-\frac{\sum_{i=1}^{n} (x_{ij} - a_i)^2}{(1 + a_{n+2})^2}}^{\rho_j}} + a_0 \tag{1.1}$$

where coefficients $\vec{a}$ should be optimized to fit a training data and reduce the error $E = \sum_{j=1}^{m} (y_j - d_j)^2$ of the model.

The gradient of error can be computed as $\nabla \vec{E} = \left(\frac{\partial \vec{E}}{\partial a_0}, \frac{\partial \vec{E}}{\partial a_1}, \cdots, \frac{\partial \vec{E}}{\partial a_{n+2}} \right)$, where $\frac{\partial \vec{E}}{\partial a_i} = \sum_{j=1}^{m} \frac{\partial \vec{E}}{\partial y_j} \frac{\partial y_j}{\partial a_i}$ and $\frac{\partial \vec{E}}{\partial y_j} = 2 \sum_{j=1}^{m} (y_j - d_j)$. The last partial derivative $\frac{\partial y_j}{\partial a_i}$ has to be computed for each coefficient a_i.

In case of the Gaussian model from Eq. 1.1, the easiest partial derivative to compute is the one in the direction of the a_0 coefficient $\frac{\partial \vec{E}}{\partial a_0} = 2 \sum_{j=1}^{m} (y_j - d_j)$. For the coefficient a_{2n+1}, the equation becomes a bit more complicated as $\frac{\partial \vec{E}}{\partial a_{n+1}} = 2 \sum_{j=1}^{m} (y_j - d_j) e^{\rho_j}$. Remaining coefficients are in the exponential part of the transfer function. They can be computed as follows $\frac{\partial \vec{E}}{\partial a_i} = \sum_{j=1}^{m} \frac{\partial \vec{E}}{\partial y_j} \frac{\partial y_j}{\partial \rho_j} \frac{\partial \rho_j}{\partial a_i}$. Then the remaining components of the gradient are

$$\frac{\partial \vec{E}}{\partial a_{n+2}} = 2 \sum_{j=1}^{m} \left[(y_j - d_j) (1 + a_{n+1}) e^{\rho_j} 2 \frac{\sum_{i=1}^{n} (a_i x_{ij})^2}{(1 + a_{n+2})^3} \right], \tag{1.2}$$

$$\frac{\partial \vec{E}}{\partial a_i} = 2 \sum_{j=1}^{m} \left[(y_j - d_j) (1 + a_{n+1}) e^{\rho_j} 2 \frac{a_i^2 x_{ij}^2}{(1 + a_{n+2})^2} \right]. \tag{1.3}$$

Accordingly, analytical gradients of error on training data were derived for other models such as Sigmoid, Sine and Exponential models.

Gradients are then supplied together with errors to the quasi-Newton optimization method [18, 19] during the training to speed up the convergence. More details can be found in [20] and the source code is also available [21].

The **LocalPolynomial** base model as well as **Neural Network (NN), Support Vector Machine (SVM), Naive Bayes classifier (NB), Decision Tree (DT), K-Nearest Neighbor (KNN)** were adopted from the Rapidminer environment [22].

In following section, I present mathematical formulas for most of the base models:

Exponential

$$y = a_{m+2} + a_{m+1} \exp \left(a_{m+3} \left(\sum_{i=0}^{m-1} a_i x_i + a_m \right) \right)$$

Gauss

$$y = a_m + (1 + a_{m+1}) \exp\left(\frac{-\sum_{i=0}^{m} (x_i - a_i)^2}{(1 + a_{m+2})^2} \right)$$

GaussNorm

$$y = \exp\left(\frac{-\sum_{i=0}^{m} (x_i - a_i)^2}{(1 + a_m)^2} \right)$$

GaussMulti

$$y = \exp\left(\frac{-\sum_{i=0}^{m} (x_i - a_i)^2}{2a_{m+i}^2} \right)$$

Linear

$$y = \sum_{i=0}^{m-1} a_i x_i + a_m$$

Polynomial

$$y = \sum_{k=0}^{m-1} a_k \prod_{i=0}^{p} x_i^{e_{ki}} + a_m$$

Sigmoid

$$y = a_{m+2} + \frac{a_{m+1}}{1 + \exp\left(-\left(\sum_{i=0}^{m-1} a_i x_i + a_m \right) \right)}$$

SigmoidNorm

$$y = \frac{1}{1 + \exp\left(\sum_{i=0}^{m-1} a_i x_i + a_m \right)}$$

Sin

$$y = a_{3m} + \sum_{i=0}^{m-1} a_{2m+i} \sin(a_i x_i + a_{m+i})$$

SinNorm

$$y = \sin\left(a_m + \sum_{i=0}^{m-1} a_i x_i \right)$$

Since derivation of gradients of squared errors is straight-forward, I won't demonstrate any other than the one for Gaussian model shown in previous section.

1.1.2 Ensembling algorithms

The performance of models can often be further increased by combining or ensembling [23, 24, 25, 26, 27, 28] base algorithms, particularly in cases where base algorithms produce models with insufficient plasticity or models overfitted to training data [29].

A detailed description of the large variety of ensemble algorithms can be found in [30]. Brief description of the ensembling algorithms that are used in my thesis is given below.

Bagging [31] is the simplest one. Bagging stands for boostrap aggregating. It takes samples of instances randomly with repetition and trains each base model on a corresponding sample. Prediction is done by a simple average or voting (in case of classification) of predictions of all base models. Bagging is one of the easiest ensembling techniques to parallelize.

Boosting [26] specializes models on instances incorrectly handled by previous models and combines them with weighted average. Boosting is an iterative process, which make it hard to parallelize. Further description of a particular type of boosting called gradient boosting is given in Section 1.3.1.4.

Stacking [25] uses a meta model, which is learned from the outputs of all base models, to combine them.

Another ensemble utilizing meta models is the **Cascade Generalization** [32], where every model except the first one uses a dataset extended by the output of all preceding models.

Delegating [33] and **Cascading** [34, 35] both use a similar principle: they operate with certainty of model output. The latter model is specialized not only in instances that are classified incorrectly by previous models, but also in instances that are classified correctly, but previous models are not certain in terms of their output. Cascading only modifies the probability of selecting given instances for the learning set of the next model.

Arbitrating [36] uses a meta-model called referee for each model. The purpose of this meta-model is to predict the probability of correct output.

In this thesis, I use selected ensemble methods implemented within the FAKE GAME project [21].

1.2 Meta-learning Templates

The meta-learning template [6] is a prescription how to build hierarchical supervised models. In the most complex case, it can be a collection of ensembling algorithms, modeling and classification algorithms combined in a hierarchical manner, where base algorithms are leaf nodes connected by ensembling nodes. Models or classifiers deeper in the hierarchy can be more specialized in a particular subset of data samples or attributes. This scheme decomposes the prediction problem into subproblems and combines the final solution (model)

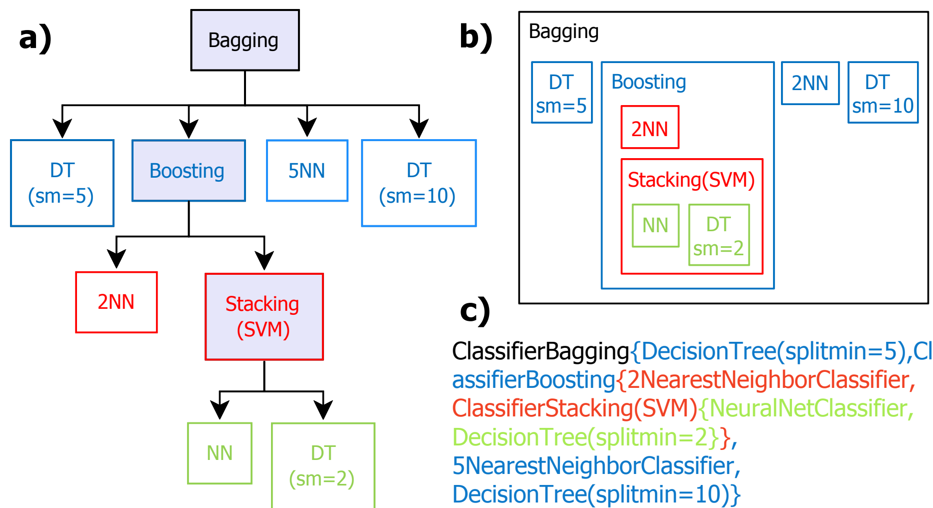


Figure 1.1: An example of hierarchical combination of algorithms. Using this meta-learning template, a classifier can be produced (see Figure 1.2). The template can be represented by **a)** a tree, **b)** embedded boxes or **c)** by text [2].

from subsolutions. The procedure of problem decomposition depends on ensembling methods. Typically, it distributes data to member models and when all outputs are available, they are combined to the ensemble output. Ensemble algorithms act as inner nodes, whereas base algorithms act as leaves in the tree representing the hierarchy.

Note that meta-learning templates are not data mining models, but algorithms. Models are produced when templates are executed.

Figure 1.1 shows an example of a meta-learning template. When executed, the full training dataset is passed to a top level bagging that generates 4 bootstrap training datasets for members of the ensemble. The second bootstrap training dataset is used to train a KNN classifier by boosting and samples where this classifier demonstrates high error are more likely to be used in the training set for the second member model of the boosting: the stacking of NN and DT classifiers. Bottom level NN and DT are evaluated on training data and upon their responses a SVM meta-model is trained. The stacking is evaluated and a weight is assigned to its output in boosting. The output of boosting is averaged with the other three top level base models and the whole classifier is finished (see the left-hand tree in Figure 1.2).

The resulting classifier is depicted in Figure 1.2. The tree in the center shows how the input attributes are presented to the model. The propagation of input vector is straightforward in this example, but some ensembles (e.g., Cascading) involve evaluation of member models (their outputs are added to input vectors of subsequent models). The right-hand tree shows how outputs of base models are blended to produce the final output.

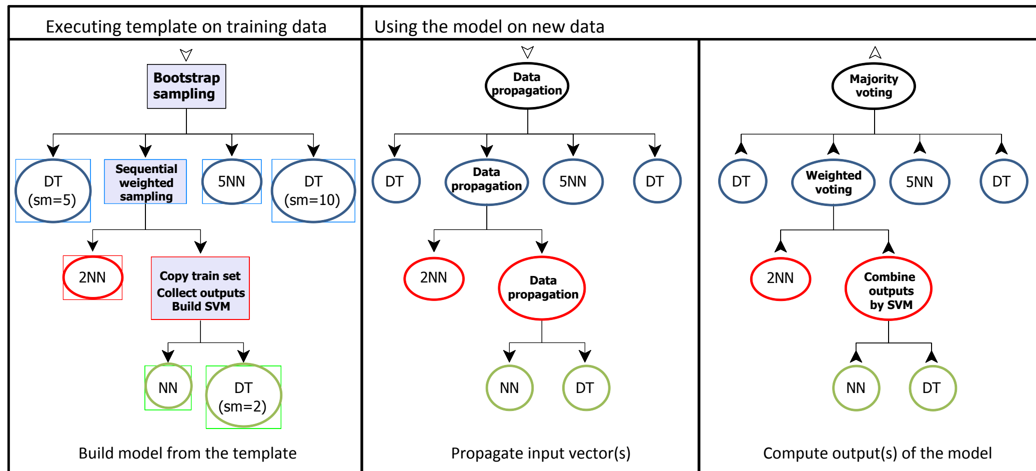


Figure 1.2: An ensemble classifier can be produced by the hierarchical combination of algorithms depicted in Figure 1.1. Executing the template will distribute data to leaf base models according to procedures specified by ensembling algorithms. Base models and ensembles are constructed until the root ensemble (base model) is finished. Using the model involves propagating and presenting an input vector to leaf models and combining their outputs by ensembling procedures. [2]

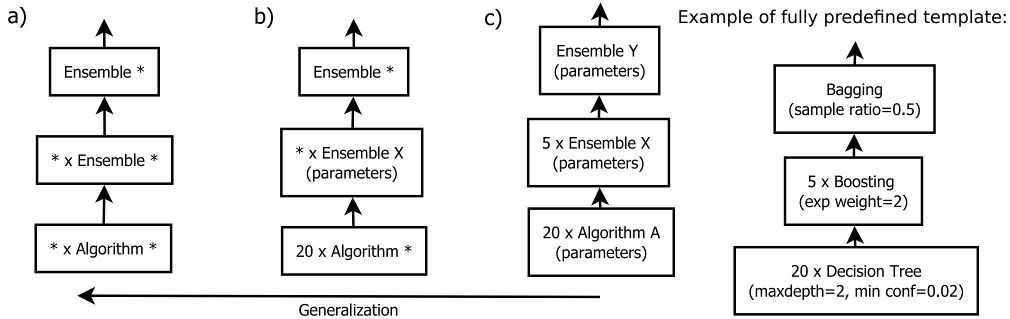


Figure 1.3: Nested ensembles can be represented by a template. Using wildcards, specific (or predefined) template can be generalized to represent set of templates. [2]

Whereas data mining workflows are directed acyclic graphs, meta-learning templates are hierarchical structures. Fully predefined templates are algorithm configurations containing parameters of both ensembles and base algorithms. Templates can be generalized using wildcards (see Figure 1.3) to represent a subspace of the search space of topologies and parametrizations of hierarchical ensembles.

Similarly to the Holland's schema theorem [37], FAKE GAME can define fitness of a template as average/maximum fitness of individual algorithms rep-

resented by this particular template. Wildcards here are used just as placeholders for random decisions on type of ensembles or base algorithms and their parameters. On the contrary, in rooted tree schema theory [38] wildcards represent sub-trees.

1.2.1 Discovering templates

The meta-learning template can be designed manually using expert knowledge (for example, bagging boosted decision trees showed good results on several problems) so it is likely to perform well on a new dataset. This is, however, not guaranteed.

FAKE GAME optimizes templates on data sub-samples using a genetic programming [39]. In this way, FAKE GAME can search the space of possible architectures of hierarchical ensembles and optimize their parameters simultaneously.

1.3 H2O

H2O is parallel machine learning package written mainly in Java. It provides bindings via its representational state transfer application programming interface (RESTful API) [40] to Java, Python, and R as well as web interface.

H2O uses in memory MapReduce [1] paradigm to distribute work. One of its advantages is clever distribution of data chunks. When H2O imports a file, it does so in distributed fashion, i.e., each node tries to load it in parallel. When the data file is parsed, it gets split in to several data chunks. H2O tries to have more data chunks than CPU cores. Since it uses MapReduce paradigm, it is often useful to be able to look in different chunk in map phase than the one map worker was given. H2O assumes that such look-ups are more likely to data chunks that are near to the given chunk. And for this reason H2O tries to keep those chunks in the same node.

Another benefit of using H2O is simplicity of deployment. H2O is capable of exporting trained models as Java class (POJO). It also makes easy to use all of its models from various languages. Officially supported are R, Python, and Java. Although it is possible to use it from other languages using its RESTful API.

H2O contains several machine learning algorithms, namely:

1. Supervised
    a) Deep Learning
    b) Distributed Random Forest
    c) Generalized Linear Model
    d) Gradient Boosting Machines

e) Naïve Bayes

2. Unsupervised

a) Deep Learning (Auto Encoder)
b) GLRM
c) K-Means
d) PCA

1.3.1 Supervised algorithms in H2O

Since I have used only supervised algorithms for this thesis, I will describe only them.

1.3.1.1 Deep Learning

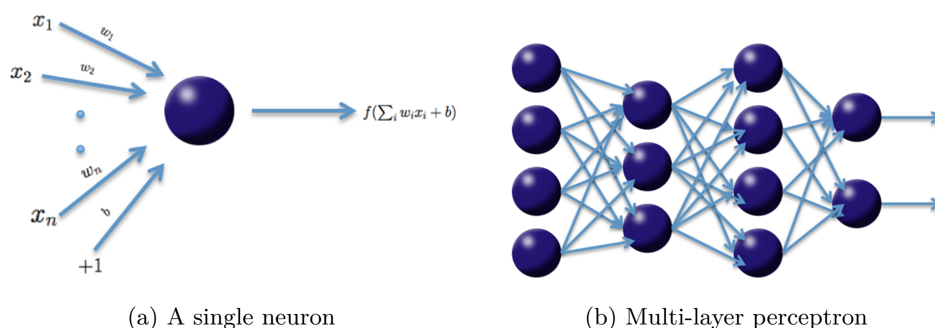
Deep Learning is based on a multi-layer feed-forward artificial neural network (also known as multi-layer perceptron (MLP)). In H2O, Deep Learning is trained with stochastic gradient descent using back-propagation. The artificial neural network can contain a large number of hidden layers consisting of neurons with tanh, rectified linear function and maxout activation functions. However, since H2O Deep Learning is implemented in Java and it does not support training on GPUs, the usual number of hidden layers is 2 to 5. For this reason, H2O Deep Water project started, which aims to integrate TensorFlow, Caffe, and MXNet, all of which can use GPUs to train the neural network yielding much higher performance.

H2O's Deep Learning has advanced features such as adaptive learning rate, rate annealing, momentum training, dropout, L1 or L2 regularization.

Each compute node trains a copy of the global model parameters using its local data with multi-threading (asynchronously), and contributes periodically to the global model via model averaging across the network [41].

The basic unit in the multi-layer perceptron (MLP) is the neuron (shown in Figure 1.4a), a biologically inspired model of the human neuron. In the MLP, the weighted combination $\alpha = \sum_{i=1} w_i x_i + b$ of input signals is aggregated, and then an output signal $f(\alpha)$ transmitted by the connected neuron. The function f represents the nonlinear activation function used throughout the network, and the bias b accounts for the neuron's activation threshold.

MLP consist of multiple layers of interconnected neuron units starting with an input layer followed by multiple layers of non-linearity and ending with a linear regression or classification layer to match the output space [41]. To make easier optimization of threshold of every single neuron bias units are included in each non-output layer of the network and threshold is then computed using weights just as every other input into each neuron. The learning phase of MLP



(a) A single neuron (b) Multi-layer perceptron

Figure 1.4: Ilustration of a single neuron and multi-layer feed-forward artificial neural network [41]

consists of adapting these weights to minimize the error on labeled training data.

Prediction of MLP is done by multiplying each input with corresponding weight and then each neuron sums its multiplicands and then activation function is applied on this sum. This is visualized in Figure 1.4a. The result is then propagated to the next layer (see Figure 1.4b). This is repeated until the output layer is reached.

Universal approximation theorem states that MLP with at least one hidden layer with finite number of neurons with activation function that's non-constant, bounded, and monotonically-increasing continuous function is capable of creating arbitrarily accurate approximation to a function [42].

MLP in H2O uses three activation functions — tanh, rectified linear activation function and maxout (see Table 1.1). The tanh function is rescaled and shifted logistic function that is symmetric around 0. The rectified linear function has demonstrated high performance on image recognition tasks, it is a more biologically accurate [43] and it is computationally cheaper. Maxout is a generalization of the rectified linear activation function where each neuron picks the larger output of k separate channels each with its own weights and bias values. The current implementation supports only $k = 2$ [41]. The rectified linear activation function is the special case of maxout where one channel always outputs 0.

Function	Formula	Range
Tanh	$f(\alpha) = \frac{e^{\alpha}-e^{-\alpha}}{e^{\alpha}+e^{-\alpha}}$	$f(\alpha) \in [-1, 1]$
Rectified Linear	$f(\alpha) = \max(0, \alpha)$	$f(\alpha) \in \mathbb{R}_+$
Maxout	$f(\alpha_1, \alpha_2) = \max(\alpha_1, \alpha_2)$	$f(\boldsymbol{\alpha}) \in \mathbb{R}$

Table 1.1: Activation functions usable in H2O

The distribution function of the response variable can be specified as one of the following: AUTO, Bernoulli, Multinomial, Poisson, Gamma, Tweedie,

Laplace, Huber or Gaussian. Each distribution has a primary association with a particular loss function. Some distributions can be used with a non-default loss function. Bernoulli and Multinomial are primarily associated with Cross Entropy (also known as Log-Loss). Gaussian is primarily associated with Mean Squared Error, Laplace with Absolute loss and Huber with Huber loss. Poisson, Gamma and Tweedie distributions requires loss to be set to AUTO [41].

Loss Function	Formula	Typical use
MSE	$L(W,B\|j) = \frac{1}{2}\|\|t^{(j)} - o^{(j)}\|\|_2^2$	Regression
Absolute	$L(W,B\|j) = \|\|t^{(j)} - o^{(j)}\|\|_1$	Regression
Huber	$L(W,B\|j) = \begin{cases} \frac{1}{2}\|\|t^{(j)} - o^{(j)}\|\|_2^2 & \text{for } \|\|t^{(j)} - o^{(j)}\|\|_1 \leq 1 \\ \|\|t^{(j)} - o^{(j)}\|\|_1 - \frac{1}{2} & \text{otherwise.} \end{cases}$	Regression
Cross Entropy	$L(W,B\|j) = -\sum_{y \in O} \left(\ln(o_y^{(j)}) t_y^{(j)} + \ln(1 - o_y^{(j)})(1 - _y^{(j)}) \right)$	Classification

Table 1.2: Loss Functions

H2O trains MLP using stochastic gradient descend, namely, with *HOGWILD!*, which is lock-free version of stochastic gradient descend [44]. Intuition of training MLP using *HOGWILD!* can be given as follows [41]:

1. Initialize global model parameters W, B
2. Distribute training data T across nodes (can be disjoint or replicated)
3. Iterate until convergence criterion reached:
   a) For nodes n with training subset T_n, do in parallel:
      i. Obtain copy of the global model parameters W_n, B_n
      ii. Select active subset $T_{na} \subset T_n$ (user-given number of samples per iteration)
      iii. Partition T_{na} into T_{nac} by cores n_c
      iv. For cores n_c on node n, do in parallel:
         A. Get training example $i \in T_{nac}$
         B. Update all weights $w_{jk} \in W_n$, biases $b_{jk} \in B_n$

$$w_{jk} = w_{jk} - \alpha \frac{\partial L(W,B|j)}{\partial w_{jk}}$$
$$b_{jk} = b_{jk} - \alpha \frac{\partial L(W,B|j)}{\partial b_{jk}}$$

   b) Set $W, B = \text{avg}_n W_n, \text{avg}_n B_n$
   c) Optionally score the model on (potentially sampled) train/validation scoring sets

HOGWILD! updates asynchronously both weights W and weights for the bias units B.

H2O Deep Learning (DL) framework supports L1 (Lasso) and L2 (Ridge) regularizations as well as dropout. To support L1 and L2 regularizations loss function is modified as follows:

$$L'(W, B|j) = L(W, B|j) + \lambda_1 R_1(W, B|j) + \lambda_2 R_2(W, B|j)$$

$R_1(W, B|j)$ represents the sum of all ℓ_1 norms of the weights and biases in the network. $R_2(W, B|j)$ represents the sum of squares of the weights and biases in the network. The constants λ_1 and λ_2 are typically very small, e.g., 10^{-5}. Dropout can be accomplished by using activation function with dropout in its name such as `TanhWithDropout`, `RectifierWithDropout` or `MaxoutWithDropout`.

H2O DL has several more advanced optimizations such as Momentum Training, Rate Annealing, and Adaptive Learning. Full description of those optimizations are beyond the scope of this thesis, so I will just briefly mention them. More detailed description with references can be found in [41].

Momentum Training modifies back-propagation by allowing prior iterations to influence the current iteration in particular a velocity vector. Momentum Training can aid in avoiding local minima, however, too much momentum can lead to instabilities. The Nesterov accelerated gradient method is recommended improvement when using momentum updates [41].

Rate Annealing gradually reduces learning rate α in order to lower the chance of oscillations around local optimum.

H2O DL implements adaptive learning rate algorithm ADADELTA [45] that automatically combines the benefits of learning rate annealing and momentum training to avoid slow convergence. Adaptive learning in general produces better results than constant learning rate which is the reason for adaptive learning being enabled by default in H2O.

Another use of DL is auto-encoder, however, it is used for unsupervised learning so I won't describe it in this thesis.

1.3.1.2 Distributed Random Forest

Distributed Random Forest (DRF) can be used for classification and regression tool. DRF creates a forest of classification or regression trees rather than a single classification or regression tree. It is similar to bagging (see Section 1.1.2), however, it is build upon de-correlated trees [46].

More trees will reduce the variance. Both classification and regression take the average prediction over all of their trees to make a final prediction, whether predicting for a class or numeric value. For a categorical response column, DRF maps factors (e.g., 'dog', 'cat', 'mouse') in lexicographic order to a name lookup array with integer indices (e.g., 'cat' $\rightarrow$ 0, 'dog' $\rightarrow$ 1, 'mouse' $\rightarrow$ 2) [47].

Creation of random forest can be described as follows [46]:

1. **For** $b = 1 \to B$
   a) Draw a bootstrap sample of size N from the training data
   b) Grow a random-forest tree T_b to the bootstrapped data by recursively repeating following steps until minimum node size is reached:
      i. select m variables at random from the p variables
      ii. pick the best split-point among the m variables
      iii. split the node into two daughter nodes
2. **Return** ensemble of trees $\{T_i | i \in \{1, \dots, B\}\}$

Apart of standard random forest algorithm (described in [48, 46]), H2O supports extremely randomized trees [49] via `histogram_type="Random"` parameter settings [47].

In extremely randomized trees (also known as Extra-Trees), instead of looking for the best split, thresholds (for the split) are drawn at random for each candidate feature and the best is picked as the splitting rule.

An empirical validation of Extra-Trees conducted by a bias/variance analysis of the Extra-Trees algorithm has shown that Extra-Trees work by decreasing variance, while at the same time increasing bias. Once properly adjusted, the variance almost vanishes, while bias only slightly increases with respect to standard trees [49].

DRF builds its trees layer by layer. This is the same way that is used in H2O's Gradient Boosting Machines. It starts by having all data in one node then optimal split point is determined. In the next step, rows are rearranged according to the split-point in the parent node. Then on each node a local histogram is created. In a subsequent step, local histograms are merged into one histogram, which is then used for finding a next split-point. This process of creating layers goes until stopping criteria are met, e.g., exceeding maximal depth, having less than a particular number of data points in the leaf, etc.

Two most influencing parameters, both performance and computational complexity, are tree depth and a number of trees in the forest. The former defaults to 20 opposed to Gradient Boosting Machines (GBM) which uses only 5 by default. The latter is set to 50 in default setting [47].

1.3.1.3 Generalized Linear Model

Generalized Linear Model (GLM) are an extension of traditional linear models. By traditional linear model, I mean linear regression with normally distributed error.

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon} \qquad \boldsymbol{\varepsilon} \sim \mathcal{N}(0, \sigma^2)$$

In this model, β are the parameters to be estimated. Under assumption of identically and independently normally distributed error, this can be solved analytically using ordinary least squares (OLS). It can be easily shown that

maximum likelihood estimate (MLE) yields the same estimate as minimizing mean squared error (MSE). This estimate has the following form:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

However, this assumes that the response y is normally distributed $y \sim \mathcal{N}(\mathbf{X}\boldsymbol{\beta}, \sigma^2)$. Since this can be too restrictive, GLM relaxes those assumptions by allowing the variance to vary as a function of the mean, non-normal errors and a non-linear relation between the response and covariates [50]. The response distribution is assumed to belong to the exponential family. Exponential family includes Gaussian, Poisson, binomial, multinomial and gamma [50]. The components of a GLM are:

- the random component f for the dependent variable y — $f(y; \theta, \phi)$ has a probability distribution from the exponential family parametrized by θ and ϕ.
- the systematic component $\boldsymbol{\eta} = \mathbf{X}\boldsymbol{\beta}$
- the link function g: $\mathbb{E}(y) = \mu = g^{-1}(\eta)$ which relates the expected value of the response μ to the linear component η.

In other words, it removes the restriction on the distribution of error and allows non-homogeneity of the variance with respect to the mean vector and also it relaxes the constrains on the additivity of covariates and it allows the response to belong to a restricted range of values depending on the chosen transformation g [50]. One well known special case is logistic regression, a model that's commonly used for binary classification in medical applications.

In general, estimate of $\hat{\beta}$ can be obtained using MLE.

$$\hat{\boldsymbol{\beta}} = \arg\max_{\beta}(\text{Log-Likelihood of the model}(\beta))$$

Linear regression with any other distribution of error implemented in H2O can't be solved analytically [50].

GLM usually uses some kind of regularization. Common options are L1, L2 and elastic net regularization penalty that combines both L1 and L2 penalty. Regularizations are applied to β without the intercept β_0.

$$\hat{\boldsymbol{\beta}} = \arg\max_{\beta}(\text{Log-Likelihood of the model}(\beta_0, \beta) - \text{Regularization Penalty}(\beta))$$

Elastic net regularization penalty is the weighted sum of the ℓ_1 and ℓ_2 norms of the coefficients vector.

$$\text{Regularization Penalty}(\beta) = \lambda \left(\alpha ||\beta||_1 + \frac{1}{2}(1-\alpha)||\beta||_2^2 \right)$$

Regularization leads to sparsity and shrinks the coefficients. These regularizations yields model with less variance in the predictions and the sparsity makes easier interpretation of the model.

The elastic net parameter $\alpha \in [0,1]$ controls the penalty distribution between L1 (least absolute shrinkage and selection operator (lasso)) and L2 (ridge). When $\alpha = 0$ the L1 penalty is not used and when $\alpha = 1$ the L2 penalty is not used. The second argument of the elastic net λ influences the strength of the regularization.

Similar to [51], H2O can compute the full regularization path starting from the null-model (evaluated at the smallest λ penalty for which all coefficients are set to zero) down to a minimally penalized model [50]. To improve efficiency of this search, H2O employs the strong rules described in [52]. In addition, cross-validation can be used for the full regularization path to determine optimal λ.

As mentioned earlier, GLM with Gaussian error has analytical solution. This also applies with L2 regularization. However, with L1 regularization there is no analytical solution, therefore iterative method like iteratively reweighted least squares (IRLSM), limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm (L-BFGS) or gradient descend is used.

Amongst other measures, H2O's GLM returns the logarithm of the ratio of likelihoods, called deviance, and Akaike information criterion (AIC). AIC is based on Kullback-Leibler divergence of the model likelihood [53]. It's worth mentioning that AIC should be used for model comparison only. If all models are performing bad AIC won't indicate it. However, it is useful as it takes into account possible overfitting of the model [50]. In H2O, AIC is available only for GLM.

Linear regression with Gaussian family uses identity as its link function g and normal distribution as the error distribution. The model has the following form:

$$\hat{y} = x^T\beta + \beta_0$$

The model is fitted by minimizing the least squares yielding a model equivalent to MLE and the regularization part:

$$\operatorname*{arg\,min}_{\beta,\beta_0} \frac{1}{2N}\sum_{i=1}^{N}(x_i^T\beta + \beta_0 - y_i)^2 + \lambda\left(\alpha||\beta||_1 + \frac{1}{2}(1-\alpha)||\beta||_2^2\right)$$

Linear regression with binomial family is also known as logistic regression and it is used for binary classification. The canonical link for the binomial family is the logit function (also known as log odds). Its inverse is the logistic function, which projects any real number to interval $[0,1]$. It models a probability of an observation belonging to an output category given

data $\mathrm{P}(y=1|x)$ [50]. The model has the following form:

$$\hat{y} = \mathrm{P}(y=1|x) = \frac{\exp(x^T\beta + \beta_0)}{1+\exp(x^T\beta + \beta_0)}$$

This can be alternatively written as:

$$\log\left(\frac{\hat{y}}{1-\hat{y}}\right) = \log\left(\frac{\mathrm{P}(y=1|x)}{\mathrm{P}(y=0|x)}\right) = x^T\beta + \beta_0$$

The model is fitted by maximizing the regularized likelihood:

$$\arg\max_{\beta,\beta_0} \frac{1}{N}\sum_{i=1}^{N}\left(y_i(x_i^T\beta + \beta_0) - \log(1+e^{x_i^T\beta+\beta_0})\right) - \lambda\left(\alpha||\beta||_1 + \frac{1}{2}(1-\alpha)||\beta||_2^2\right)$$

Linear regression with multinomial family is generalization of the logistic regression used for multi-class response variables. The model represents the conditional probability of a given class given the data $\mathrm{P}(y=c|x)$. The model has the following form:

$$\hat{y}_c = \mathrm{P}(y=c|x) = \frac{\exp(x^T\beta_c + \beta_{c,0})}{\sum_{k=1}^{K}(\exp(x^T\beta_k + \beta_{k,0}))}$$

where β_c is a vector of coefficients for class `c` and $\hat{y}_c$ is probability of class `c`. The model is fitted by as described in following formula:

$$\arg\max_{\boldsymbol{\beta},\boldsymbol{\beta}_0} \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{k=1}^{K}(y_{i,k}(x_i^T\beta_k + \beta_{k,0})) - \log\left(\sum_{k=1}^{K}\exp(x_i^T\beta_k + \beta_{k,0})\right)\right) - \lambda\left(\alpha\sum_{j=1}^{P}||\beta_j||_1 + \frac{1-\alpha}{2}\sum_{j=1}^{P}||\beta_j||_2^2\right)$$

Linear regression with Poisson family also known as Poisson regression is typically used for datasets, where the response represents counts and the errors are assumed to have a Poisson distribution [50]. It can be applied to any data where the response is non-negative. The model has the following form:

$$\hat{y} = \exp(x^T\beta + \beta_0)$$

The model is fitted by maximizing the following formula:

$$\arg\max_{\beta,\beta_0} \frac{1}{N}\sum_{i=1}^{N}\left(y_i(xi^T\beta + \beta_0) - \exp(x_i^T\beta + \beta_0)\right) - \lambda\left(\alpha||\beta||_1 + \frac{1-\alpha}{2}||\beta||_2^2\right)$$

Linear regression with gamma family is useful for modeling a positive continuous response variable, where the conditional variance of the response grows with its mean, but the coefficient of variation of the response $\sigma^2(y_i)/\mathbb{E}y_i$ is constant. It is usually used with the inverse link $g(\mathbb{E}y) = (\mathbb{E}y)^{-1}$ or log link.

$$\hat{y} = \frac{1}{x^T\beta + \beta_0}$$

The model is fitted by maximizing the following expression:

$$\arg\max_{\beta,\beta_0} -\frac{1}{N}\sum_{i=1}^{N}\left(\frac{y_i}{x_i^T\beta + \beta_0} + \log(x_i^T\beta + \beta_0)\right) - \lambda\left(\alpha||\beta||_1 + \frac{1-\alpha}{2}||\beta||_2^2\right)$$

Linear regression with Tweedie family is especially useful for modeling positive continuous variables with exact zeros [50]. Tweedie distributions are a family of distributions which include gamma, normal, Poisson and their combination. The Tweedie distribution is parametrized by variance power p. The variance of the Tweedie distribution is proportional to the p-th power of the mean. It is defined for all p values except in the $(0, 1)$ interval. It has following special cases [50]:

- $p = 0$: normal
- $p = 1$: Poisson
- $p \in (1, 2)$: compound Poisson, non-negative with mass at zero
- $p = 2$: gamma
- $p = 3$: inverse-gamma

For more information about GLM refer to [54] and in context with H2O refer to [50].

1.3.1.4 Gradient Boosting Machines

A GBM is an ensemble of classification or regression trees.

Boosting, described in Section 1.1.2, is an iterative procedure that helps to improve the accuracy of trees [55]. Boosting uses weak learners and iteratively creates an ensemble by gradually adding weight to misclassified training samples.

The modified dataset is used for training a new tree. Subsequently, using an ensemble that includes the new tree, boosting adds more weight to currently misclassified training samples. This goes on until some stopping conditions are met.

Therefore, boosting can't be easily parallelized. H2O builds its trees using the same parallelized tree creation as DRF does.

Generic gradient tree-boosting can be summarized as follows [46]:

$f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$;
for $m = 1 \to M$ **do**
  for $i = 1 \to N$ **do**
    $r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}$
  end
  fit a regression tree to the targets r_{im} giving terminal regions R_{jm} for $j = 1, 2, \dots J_m$;
  for $j = 1 \to J_m$ **do**
    $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$
  end
  Update $f_m = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$
end
return $\hat{f}(x) = f_M(x)$

The first line initializes optimal constant model, which is just a single terminal node. On the second line, generalized residuals r are computed. Those residuals are components of gradient.

For classification, the for-cycle on the second line is repeated K-times, where K is the number of categories. The output for classification task consists of k different (coupled) tree expansions $f_{kM}(x) \quad k = 1 \dots K$ [46].

Two basic tuning parameters are the number of iterations M and sizes of each constituent trees J_m.

Implementation of GBM in H2O uses distributed trees. Each node creates, in parallel, a local histogram using only node-local data. Then histograms are merged into one and a split column is selected to make the decision. The rows are reassigned to nodes and the whole procedure is repeated [55].

Initially, all rows start on node 0. In the next step, rows are reorganized according to a split in the root node. Each consequent step creates a new level in a similar fashion — rows are grouped according to their position in the tree. A tree with depth n needs n MapReduce passes.

GBM creates histogram using predefined number of bins, which defaults to 1024. This number should be at least as large as the number of factors (nominal variables).

1.3.1.5 Naïve Bayes

Naïve Bayes (NB) is a classification algorithm that has assumption of independence between covariates. It is based on Bayes' theorem and it is useful in task with highly dimensional predictor space [46].

Naïve Bayes outputs a list of a-priori and conditional probabilities of each class of the response variable. The a-priori probability is the estimated probability of a particular class before observing any of the predictors. Each conditional probability corresponds to a particular predictor column. Naïve Bayes

assumes that each numerical predictor is sampled from a Gaussian distribution given the class of the response [56].

I will demonstrate NB for simplified binomial case presented on H2O's web site [56].

Naïve Bayes assumes independence of predictors hence the joint likelihood of the data can be expressed as:

$$\mathcal{L}(\phi(y), \phi_{i|y=1}, \phi_{i|y=0}) = \prod_{i=1}^{m} p(X^{(i)}, y^{(i)})$$

where $\phi_{i|y=0} = p(x_i = 1|y = 0)$ and $\phi_{i|y=1} = p(x_i = 1|y = 1)$, i.e., $\phi_{i|y=0}$ can be thought of as a fraction of the observed instances, where feature $x_i = 1$ the outcome is $y = 0$ similarly for $\phi_{i|y=1}$.

Maximal likelihood estimates of ϕ are:

$$\phi_{j|y=1} = \frac{\sum_i^m I(x_j^{(i)} = 1 \bigcap y^{(i)} = 1)}{\sum_{i=1}^m I(y^{(i)} = 1)}$$

$\phi_{j|y=0}$ is defined in similar fashion and $\phi(y) = \frac{I(y^{(i)}=1)}{m}$.

Prediction is then carried using Bayes' rule as follows:

$$p(y = 1|x) = \frac{\prod p(x_i|y = 1)p(y = 1)}{\prod p(x_i|y = 1)p(y = 1) + \prod p(x_i|y = 0)p(y = 0)}$$
$$p(y = 0|x) = \frac{\prod p(x_i|y = 0)p(y = 0)}{\prod p(x_i|y = 1)p(y = 1) + \prod p(x_i|y = 0)p(y = 0)}$$

and then the class with the highest probability is chosen as the prediction.

In a case, when the training dataset does not contain particular category or some particular category is rare, Laplace smoothing can be used. Otherwise, the prediction could have probability of 0. It is a form of regularization of NB.

Laplace smoothing is done by adding some fixed α to the numerator and αk to the denominator of both conditional probabilities ϕ, where k is number of categories. The most commonly used value of α is one. In this case, the term *add-one smoothing* is often used.

1.4 H2O Ensemble

H2O Ensemble is based upon the Super Learner algorithm, which is generalization of the stacking algorithm. The "Super Learner" name was introduced due to the theoretical oracle property and its consequences [57].

Super Learner ensemble is asymptotically equivalent to the oracle. Under most conditions, the Super Learner should be at least as good as the best individual base learner. Hence, if model performance is the primary objective, the Super Learner is a good candidate for a model to use.

Stacking (also known as stacked generalization) is a procedure for ensemble learning. Stacking uses two layers, in the first layer there are base learners and the second layer consists of a so-called metalearner, which uses output of base learners as its input. The output from base learners is also called *level-one* data and the original training data are called *level-zero* data. The Super Learner (SL) theory requires cross-validation to generate *level-one* data [57].

The process of constructing the *level-one* data involves generating an $n \times L$ matrix of k-fold cross-validated predicted values, where n is the number of rows and L is the number of base learners. The metalearner is used to find optimal combination of base learners [57].

At the time of writing this thesis, H2O Ensemble is implemented only in R, however, there are plans to port it to H2O core to make it easily usable from Python, Java (and JVM based languages) and R.

1.5 Hyper-Parameter Optimization

Hyper-parameter optimization can have a great impact on performance of machine learning algorithms. It often affects both time used for learning and its ability to predict (measured by accuracy, AUC, F-measure, etc.).

In the next few sections, I will describe several approaches to hyper-parameter optimization. To make things clearer, I will use the term *cost function* instead of accuracy, AUC, etc., of trained machine learning model, since the next sections are not dependent on selected performance measure.

1.5.1 Manual Hyper-Parameter Search

Manual hyper-parameter search is the simplest and also the most costly. It requires a human expert interaction in each step of the hyper-parameter tuning. Moreover, a good hyper-parameters are often data dependent, so manual hyper-parameter search can't be much effective. Due to its simplicity, manual hyper-parameter search is still widely used.

1.5.2 Grid Search

Grid search is systematic approach to finding good hyper-parameters. Human expert selects which hyper-parameters to optimize if not all. Then regular n-dimensional grid is created, where n is the number of selected hyper-parameters. The next step is to test every point of the grid cross-section. This makes it computationally intensive with increasing number of selected hyper-parameters, because the number of cross-sections increases exponentially as the number of selected hyper-parameters.

Another issue is that the cost function is usually very sensitive to changes in some subset of hyper-parameters, while for some other hyper-parameters it acts as rather insensitive. This can be seen in Figure 1.5.

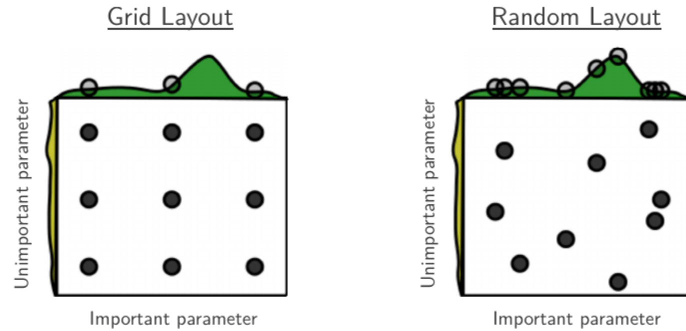


Figure 1.5: Grid and random search of nine trials for optimizing a function $f(x, y) = g(x) + h(y) \approx g(x)$ with low effective dimensionality. Above each square $g(x)$ is shown in green, and left of each square $h(y)$ is shown in yellow. With grid search, nine trials only test $g(x)$ in three distinct places. With random search, all nine trials explore distinct values of g. This failure of grid search is a rule rather than an exception in high dimensional hyper-parameter optimization [58].

The latter is one of the reasons why random search performs usually better [58].

1.5.3 Random Search

Random Search is another kind of systematical hyper-parameter optimization. It samples i.i.d. the hyper-parameter space; therefore, it can find even those optima that would be hidden for grid search in the same amount of time.

Grid search and random search are both uninformed, since they don't modify the set of hyper-parameters to be evaluated during the hyper-parameter optimization.

1.5.4 Bayesian Optimization

Bayesian optimization is another kind of optimization that selects the hyper-parameters to be evaluated according to previously seen hyper-parameters and corresponding values of cost function.

Sequential Model-based Bayesian Optimization (SMBO) is general approach to Bayesian optimization. This approach can be described as follows in Algorithm 1.

There are 3 important steps in SMBO. In Algorithm 1, those are denoted as `fitModel`, `selectConfiguration`, and `evaluate`.

In `fitModel`, SMBO uses some probabilistic model. An usual choice is Gaussian process (GP), but at least one noteworthy SMBO uses Random

```
Result: set of hyper-parameters and resulting cost function
configurations[0] ← sampleHyperParameters();
costs[0] ← evaluate(configurations[0]);
while ¬ stoppingCondition do
    model ← fitModel(configurations, costs);
    config ← selectConfiguration(model);
    cost ← evaluate(config);
    append(configurations, config);
    append(costs, cost);
end
```

Algorithm 1: Description of Sequential Model-based Bayesian Optimization. It is important to realize that the `model` is a model used for predicting new hyper-parameters not the model for which the hyper-parameters are optimized.

Forest (RF) as its probabilistic model. The latter SMBO is called Sequential Model-based Algorithm Configuration (SMAC) and will be described later.

Next step is `selectConfiguration`. In this step, SMBO select configuration using so-called acquisition function. There are three majorly used acquisition functions Probability of Improvement (POI), Expected Improvement (EI), Upper Bound Confidence (UBC). General form of these acquisition functions is given below.

$$\mathrm{POI}(x) = P(f(x) \geq f(x^{+}) + \xi) \tag{1.4}$$

$$\mathrm{EI}(x) = \mathbf{E}(\max\{0, f_{t+1}(x) - f(x^{+})\}) \tag{1.5}$$

$$\mathrm{UBC}(x) = \mu(x) + \kappa\sigma(x) \tag{1.6}$$

UBC is usually used with GP, so μ, σ are mean and standard deviation respectively. Acquisition functions are one of the reasons why this kind of optimization is called Bayesian. Bayesian statistics often uses whole distribution as opposed to frequentist approach, where a point estimate is more common. Acquisition functions are used to create a value from a distribution. In layman's terms, they often combine expected value of prediction with the uncertainty of the prediction, i.e., predicted variance. Then using some optimization algorithm, SMBO optimizes the acquisition function whose evaluation (and evaluation of the underlying probabilistic model) is much cheaper than the evaluation of the machine learning algorithm whose hyper-parameters are optimized.

The last step of SMBO is evaluation of the selected hyper-parameters and insertion of the hyper-parameter configuration and its *cost* to the respective sets used for the training of the probabilistic model.

This section described general outline of Sequential Model-based Bayesian Optimization. The next section will describe basic SMBO with Gaussian processes.

1.5.4.1 Sequential Model-based Bayesian Optimization with Gaussian Processes

With GP, it becomes clearer why SMBO is called Bayesian. There are two common approaches in statistics frequentist and Bayesian. The main difference lies within the concept of probability. The former approach takes probability as number of positive outcomes divided by total number of occurrences, whereas in Bayesian approach probability is more like belief updated with data. Since in the latter approach I can use assumptions about problem and formulate it as a prior distribution, Bayesian approach often uses whole distributions opposed to frequentist approach, where point estimates are used more commonly.

In following sections, I will show how SMBO uses whole distribution (using acquisition functions) as opposed to point estimate such as mean or median. Since it can be most easily described using GP, I will use GP to introduce the basic idea of SMBO.

Gaussian Process is an infinite dimensional generalization of multivariate Gaussian distribution. Gaussian process is random process, which can be parametrized by a mean function and (co)variance[1] function. Mean function can be any function, but the usual choice is constant 0. Covariance function has to be positive-definitive function. This class of functions is also known as kernel functions. Covariance functions influence mainly smoothness and periodicity of Gaussian process.

The most important information for the following section is that a sample from random process can be viewed as a function since it is defined on whole continuum. Due to the properties formulated in last two sentences, GP can be viewed as prior random distribution for functions with some properties that depend on selected covariance function [59].

Gaussian process is a random process, so it is possible to use Bayes' rule and perform so called Bayesian updating. Fortunately, there is an analytical solution, so no iterative method needs to be used.

There are two basic views on deriving GP — weight-space view and function-space view. I like the function-space view better, since it is easier to grasp.

A prior Gaussian process is updated by data resulting in a posterior Gaussian process.

The main idea used for using GP for regression, in the function-space view, stems from unique property of multivariate Gaussian distributions called

[1] naming differs in different sources. I will refer to this as covariance or kernel function.

consistency. In layman's terms, consistency can be described by the following statement: "Marginal distribution of a multivariate Gaussian distribution is also a Gaussian distribution".

This property can be demonstrated as follows. Let $\boldsymbol{X}$ is N-dimensional normal random variable.

$$\boldsymbol{X} \sim \mathcal{N}(\mu, \boldsymbol{V})$$

And B is N-dimensional matrix that consists of M-dimensional identity matrix where $M < N$. And the rest of matrix $\boldsymbol{B}$ are zeros.

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & \ddots & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \ddots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \end{pmatrix}$$

Then one M-dimensional marginal distribution of $\boldsymbol{X}$ can be written as $\boldsymbol{BX}$.

$$\boldsymbol{Y} = \boldsymbol{BX} \sim \mathcal{N}(\boldsymbol{B}\mu, \boldsymbol{BVB}^T)$$

Other marginal distribution can be found by permuting N elements on the diagonal of matrix $\boldsymbol{B}$.

Gaussian process regression uses this property to get rid of infinite dimensions and deal with only finitely dimensional Gaussian distribution. Specifically, with N+1 dimensional Gaussian distribution, where N is number of observations.

Even though, I did not use SMBO with GP they are useful for explanation and visualisation of acquisition functions.

For this reason, I will briefly demonstrate how to use GP for regression. Detailed explanation is out of the scope of this thesis and there are nice publications suitable for this purpose [59, 60],

Let's assume I have n observations and, for simplicity, I will use prior Gaussian process with mean constantly zero. To predict a point (x^*, y^*), I need to calculate $(n+1) \times (n+1)$-dimensional covariance matrix. To do this, I use covariance function k. I will use parts of this covariance matrix, so I denote $(n \times n)$-dimensional upper-left matrix as $\boldsymbol{K}$. The rest is assigned in following equation.

$$\boldsymbol{\Sigma} = \begin{pmatrix} k(x_1, x_1) & k(x_1, x_2) & \dots & k(x_1, x_n) & k(x_1, x^*) \\ k(x_2, x_1) & k(x_2, x_2) & \dots & k(x_2, x_n) & k(x_2, x^*) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ k(x_n, x_1) & k(x_n, x_2) & \dots & k(x_n, x_n) & k(x_n, x^*) \\ k(x^*, x_1) & k(x^*, x_2) & \dots & k(x^*, x_n) & k(x^*, x^*) \end{pmatrix} = \begin{pmatrix} \boldsymbol{K} & K^{*T} \\ K^* & K^{**} \end{pmatrix}$$

Then the marginal normal distribution is as follows.

$$\begin{pmatrix} \mathbf{y} \\ y_* \end{pmatrix} = \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} \boldsymbol{K} & K^{*T} \\ K^* & K^{**} \end{pmatrix}\right)$$

Using the marginal distribution, I can predict y^* using following formula.

$$y^*|\mathbf{y} \sim \mathcal{N}(K^*\boldsymbol{K}^{-1}\mathbf{y}, K^{**} - K^*\boldsymbol{K}^{-1}K^{*T})$$

The most important parameter for GP is covariance or kernel function. Typical choice is squared-exponential, which yields Gaussian process as prior for smooth functions, i.e., infinitely derivable. Another popular choice is Matérn kernel with $\nu = 5/2$ or $\nu = 3/2$. With given ν Gaussian process is $\lceil \nu - 1 \rceil$-times derivable[59]. It can be shown that as $\nu \to \infty$ Matérn kernel converges to squared-exponential kernel. Another kernel that's not widely used, but is worth mentioning is dot-product kernel. With this kernel function the mean of the posterior Gaussian process is equivalent to ridge regression.

Hutter et al.[3] have given a definition of mixed kernel, i.e., kernel defined for numerical and categorical parameters. They use weighted squared-exponential and weighted hamming distance combined together. It is noteworthy that this kernel wasn't used in any major Bayesian optimization method.

Formulas for mentioned kernels follows:

- squared-exponential

$$k(x_i, x_j) = \sigma^2 \exp\left(\frac{-||x_i - x_j||^2}{2l^2}\right) + \sigma_n^2 \delta(x_i, x_j)$$

- Matérn[2]

$$k(x_i, x_j) = \sigma^2 \frac{1}{\Gamma(\nu)2^{\nu-1}} \left(\gamma\sqrt{2\nu}\frac{||x_i - x_j||}{l})\right)^{\nu} K_\nu\left(\gamma\sqrt{2\nu}\frac{||x_i - x_j||}{l}\right)$$

- dot product

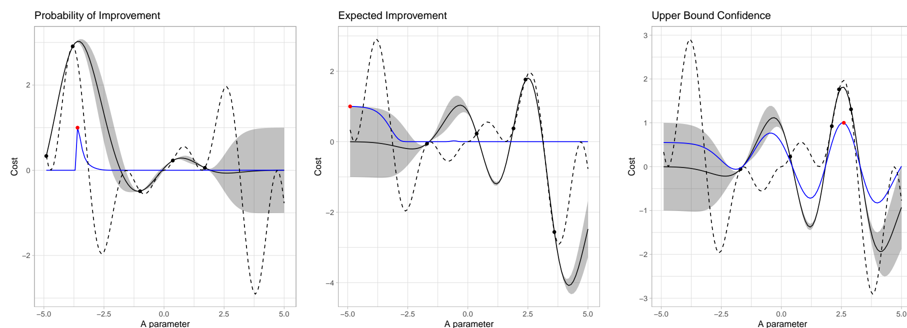
$$k(x_i, x_j) = \sigma_0^2 + x_i \cdot x_j$$

- mixed

$$k(x_i, x_j) = \exp\left[\sum_{l \in \mathcal{P}cont} -\lambda_l (x_i^{(l)} - x_j^{(l)})^2 + \sum_{l \in \mathcal{P}cat} -\lambda_l (1 - \delta(x_i^{(l)}, x_j^{(l)}))\right]$$

Some kernel functions contain parameters that can be optimized, e.g., by optimizing maximal likelihood of marginal distribution of the GP over the observations.

Usage of Gaussian process for regression requires $\mathcal{O}(n^3)$ computational steps. To overcome this as a potential bottleneck, an approximation called Projected Processes can be used [59, 3].



(a) Probability of Improvement (b) Expected Improvement (c) Upper Bound Confidence

Figure 1.6: Visualization of acquisition functions used for maximizing cost function. Black line with ribbon is posterior Gaussian process with $\pm 1\sigma$ region. Black dashed line is the cost function (ground truth). And the blue line represents respective acquisition function(normalized to -1 – +1).

Acquisition Functions for Gaussian Processes All three mentioned acquisition functions have closed form for GP, i.e., can be computed analytically [61].

Let x_{best} denote best observed hyper-parameters so far, μ mean of the posterior GP, σ standard deviation of the posterior GP, Φ cummulative distribution function (cdf) of standard normal distribution and ϕ probability density function (pdf) of standard normal distribution.

Then let $\gamma(x)$ be z-score normalized maximal observed value.

$$\gamma(x) = \frac{f(x_{\text{best}}) - \mu(x)}{\sigma(x)}$$

Then acquisition functions, defined in previous section, have the following form.

- Probability of Improvement

$$\text{POI}(x) = \Phi(\gamma(x))$$

- Expected Improvement

$$\text{EI}(x) = \sigma(x)(\gamma(x)\Phi(\gamma(x)) + \phi(\gamma(x))$$

- Upper Bound Confidence

$$\text{UBC}(x) = \mu(x) - \kappa\sigma(x)$$

[2] K_ν is the modified Bessel function of the second kind

Selection of acquisition function depends on the problem, however, the most commonly used is the Expected Improvement. Other acquisition functions are not very common. If one of the previously mentioned acquisition functions is not used it is very probable that some combination of them is used.

1.5.4.2 Tree-structured Parzen Estimator

Tree-structured Parzen Estimator (TPE) is another approach to Sequential Model-based Bayesian Optimization. Whereas GP are discriminative model TPE is generative, i.e., GP models $p(y|x)$, TPE models $p(x|y)$ and $p(y)$ [4].

TPE makes the following replacements — uniform $\rightarrow$ truncated Gaussian mixture, log-uniform $\rightarrow$ exponentiated truncated Gaussian mixture, categorical $\rightarrow$ re-weighted categorical.

TPE models $p(y)$ and $p(x|y)$. The latter is modelled by two densities separated by some threshold y^*.

$$p(x|y) = \begin{cases} l(x), & \text{if } y < y^* \\ g(x), & \text{if } y \geq y^* \end{cases}$$

y^* is usually chosen as γ-quantile[3] of observed costs [14]. Intuitively, this creates a probabilistic density estimator $l(\cdot)$ for hyper-parameters that appear to do 'well', and a different density estimator $g(\cdot)$ for hyper-parameters that appear 'poor' with respect to the threshold [14].

Bergstra et al. [4] showed that EI for TPE is proportional to $\left(\gamma + \frac{g(x)}{l(x)}(1-\gamma)\right)^{-1}$.

In other words, to maximize expected improvement select points x with high probability under $l(x)$ and low probability under $g(x)$. The tree-structured form of l and g makes it easy to draw many candidates according to l and evaluate them according to $\frac{g(x)}{l(x)}$. On each iteration, the algorithm returns the candidate x^* with the greatest EI.

1.5.4.3 Randomized Online Aggresive Racing

Randomized Online Aggresive Racing is unusual SMBO, since it uses uniform model as its model, or, in other words, Randomized Online Aggresive Racing (ROAR) is model-free. However, ROAR is important to mention, because it uses evaluation[4] procedure that is used in next SMBO.

ROAR was developed for optimization in stochastic settings. In this setting, it is unsure whether a configuration that performed well, will perform well in the future; therefore each hyper-parameter configuration is tested several times before it can be accepted as best configuration.

[3] default setting $\gamma = 0.15$ for minimization

[4] in the original paper it is called intensification procedure [3]

```
Data: χ hyper-parameters to evaluate, x_best best (incumbent) hyper-parameters,
      M model, R sequence of target algorithm runs, t_intensify time bound,
      Π instance set, ĉ cost metric
Result: x_best best (incumbent) hyper-parameters,
        R sequence of target algorithm runs
for i := 1, ..., length(χ) do
    x_new ← χ[i];
    if R contains less than maxR runs with configuration x_best then
        Π' ← {π' ∈ Π | R contains less than or equal number of runs
         using x_best and π' than using x_best and any other π'' ∈ Π};
        π ← instance sampled uniformly at random from Π';
        s ← seed, drawn uniformly at random;
        R ← ExecuteRun(R, x_best, π, s);
    end
    N ← 1;
    while True do
        S_missing ←(instance, seed) pairs for which x_best was run before,
         but not x_new;
        S_torun ←random subset of S_missing of size min(N, |S_missing|);
        foreach (π, s) ∈ S_torun do
            R ← ExecuteRun(R, x_new, π, s);
        end
        S_missing ← S_missing \ S_torun;
        Π_common ← instances for which we previously ran both x_best and x_new;
        if ĉ(x_new, Π_common) > ĉ(x_best, Π_common) then
            break;
        else if S_missing = ∅ then
            x_best ← x_new;
        else
            N ← 2N;
        end
    end
    if time spent in this call to this procedure exceeds t_intensify and i ≥ 2 then
        break;
    end
end
return (R, x_best)
```

Algorithm 2: Evaluation step in ROAR also known as intensification

ROAR can be defined as SMBO with uniform model and intensification procedure (Algorithm 2) as evaluation.

1.5.4.4 Sequential Model-based Algorithm Configuration

Sequential Model-based Algorithm Configuration comes with several enhancements, when compared to previous GP-based approach.
Main enhancements are:

1. evaluation scheme
2. use multiple observation to reduce the noise

3. user-defined cost metric and its transformation

4. sampling of the acquisition function

Evaluation scheme is the same as in ROAR. Which alone is outperforming random search, at least in stochastic settings, i.e., one configuration can be good for one evaluation and bad for another.

Another enhancement is ability to use multiple observations for single configuration. Classic SMBO with GP approach is also able to use multiple observations, however, it is not, in general, used. That is probably because there is not an obvious way, how to determine uncertainty of observation other than retesting the same configuration for GP. On the other hand, it is straightforward with random forests, where you can use variance of prediction of the configuration in question.

Next enhancement is important when you have decision problem such as Boolean Satisfiability Problem (SAT) — in such a case you generally don't care how good the solution is because in this settings it doesn't make much sense, however, you often care about how long did the computation took. It has been demonstrated that log-transformation applied to a temporal cost metric often performs better than just using untransformed cost metric [62, 63].

However, transformation of cost metric can complicate some situations [3]. Such as having multiple observations for one configuration, e.g., while using log-tranformed cost metric GP would use geometric mean instead of arithmetic mean [3]. Mainly for this reason, SMAC uses Random Forest as its probabilistic model as working with transformations becomes trivial — in the leaf nodes untransform the values, apply user-defined cost metric and transform the results.

The last enhancement worth mentioning, in this thesis, is the optimization of the acquisition function. The usual approach for SMBO was to take 10000 random samples of the acquisition function and pick the best [63, 64, 65]. Nonetheless, this is not very effective when considering highly dimensional configuration spaces. SMAC computes EI for every previously evaluated configuration. Then pick ten best according to EI and then initialize a local search at each of them. To seamlessly handle mixed categorical/numerical parameters SMAC uses randomized one-exchange neighborhood, i.e., including the set of all configuration that differ in the value of exactly one discrete parameter as well as four random neighbours for each numerical parameter [3]. More concretely, SMAC normalizes range of each numerical parameter to $[0, 1]$ and the random neighbor is sampled from Gaussian distribution with mean at the current observation and standard deviation 0.2. Since batch model prediction is cheaper than separate predictions (and with that the evaluation of EI) Sequential Model-based Algorithm Configuration uses best improvement search, evaluating EI for all neighbours at once [3]. The local search is stopped when none of the neighbours has larger EI.

Since computing EI is cheap SMAC takes additional 10,000 uniform samples over the parameter space and then sorts the 10,010 configurations[5] according to their EI [3]. Finally, SMAC interleaves those sorted configurations with another randomly sampled configurations in order to provide unbiased training data for future models [3].

1.5.5 Hyperband

Hyperband is a new hyper-parameter optimization algorithm that tries to perform better evaluation [66]. It does so by assuming that rank of performances on a subproblem will be similar to rank of performances on whole problem.

Hyperband defines budget $\mathcal{B}$ of computational resources. Each evaluation of a subproblem takes some of the computational resources, which can be time (e.g., number of iterations of gradient descend, wall time), dataset subsamples, or feature subsamples [66].

Hyperband extends *successive halving* algorithm proposed by [67, 68]. The main idea of *successive halving* is described in the following paragraph.

Uniformly allocate a budget to a set of hyper-parameter configurations, evaluate the performance of all configurations, throw out the worst half, and repeat until only one configuration remains. The algorithm allocates exponentially more resources to more promising configuration [66].

Hyperband requires two inputs — the maximum amount of resource that can be allocated to a single configuration ($\mathcal{R}$) and a value that controls the proportion of configurations discarded each round of successive halving (η).

Input: $\mathcal{R}$, η (default $\eta = 3$)
Initialization: $s_{max} = \lfloor \log_\eta(\mathcal{R}) \rfloor$, $\mathcal{B} = (s_{max} + 1)/\mathcal{R}$
for $s \in \{s_{max}, s_{max-1}, \ldots, 0\}$ **do**
    $n = \lceil \frac{\mathcal{B}\eta^s}{\mathcal{R}(s+1)} \rceil, \quad r = \mathcal{R}\eta^{-s}$;
    T =get-hyperparameter-configuration(n) ;
    `// SuccessiveHalving` {
    for $i \in \{0, \ldots, s\}$ **do**
        $n_i = \lfloor n\eta^{-i} \rfloor$;
        $r_i = r\eta^i$;
        $L = \{\text{run-then-return-validation-loss}(t, r_i) | t \in T\}$;
        $T = \text{top-k}(T, L, \lfloor n_i/\eta \rfloor)$;
    end
    `//` }
end
Result: Configuration with the smallest intermediate loss seen so far

Algorithm 3: General outline of Hyperband algorithm

[5] 10 are from the local search

- `get-hyperparameter-configuration`(n) returns n i.i.d. random samples of hyper-parameter space
- `run-then-return-validation-loss`(t, r_i) runs configuration t with allocated resources r_i and returns validation loss
- `top-k`($configs, losses, k$) returns top-k performing configurations

Since H2O did not support suspending and resuming training after some given time and various machine learning algorithms take various time, I did not evaluate this hyper-parameter optimization method. Another reason is that it would not seem fair to compare this hyper-parameter optimization method that highly specialized on problems that can be transformed to similar subproblems, i.e., training on subset of the data takes less time and generally yields a similar model.

Chapter 2

Analysis and design

H2O is build upon MapReduce paradigm introduced in [1]. This concept originates in functional programming languages, such as Lisp. Unlike pure old Map and Reduce, which operates on simple lists, MapReduce has the following type signatures:

$$\begin{aligned} \textbf{Map} &:: \text{Key}_a, \text{Value}_a \rightarrow \text{Key}_b, \text{Value}_b \\ \textbf{Reduce} &:: \text{Key}_b, \text{Value}_b \rightarrow \text{Value}_c \end{aligned}$$

In the **Map**-phase, a function with type signature written above is applied on each key-value pair independently returning the same number of key-value pairs. Then resulting key, value pairs are shuffled so that key-value pairs with the same key are close, ideally on the same node.

This explicit shuffling is in the original paper omitted, instead they use file-system as a key-value storage, so the shuffling is just reading a file, however, persisting data and reading it causes this approach to be slower than it can be. This is one of the reasons why is Spark faster than Hadoop. Both are implementations of the same MapReduce paradigm, but Spark uses keeps its data in memory (RAM) and so does H2O.

In the **Reduce**-phase, a function with type signature written above is applied on each key-value pair and it produces one aggregated value per key. Such aggregation can be sum, mean, or any other user-defined function.

All models in H2O are written using in memory MapReduce, however, there are usually more than one MapReduce passes. It also enables you to write only Map or Reduce phase.

2.1 Integration of FAKE GAME into H2O

As part of my thesis, I have integrated FAKE GAME into H2O as a model. FAKE GAME is therefore usable through H2O Flow (H2O's web interface),

R, Python, Java, and H2O's RESTful API.

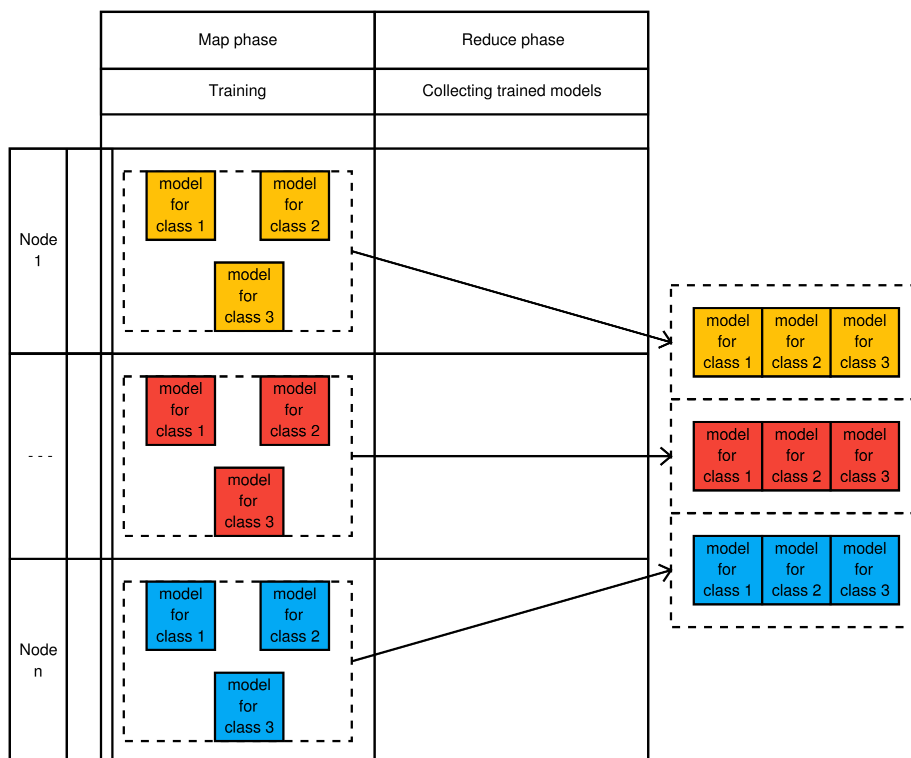


Figure 2.1: Training of FAKE GAME model in H2O uses distributed data. Each node trains its FAKE GAME models. Since classification in FAKE GAME is done by regression on each class, you can see three colored boxes one for each hypothetical class. Reduce phase consists of simple collection of trained models. Dashed box represents instance of FAKE GAME. During prediction, one class is predicted by each instance (one prediction for each color). Resulting class is determined by voting.

H2O FAKE GAME model, which I have implemented, takes a `model_config` parameter, which is a template specification for FAKE GAME. Each **Map** job trains a FAKE GAME model specified by the template given in the `model_config` parameter. It does so completely independently on the other **Map** jobs only with its local data. It is similar to bagging (bootstrap aggregating), however, it does not use bootstrapped data. The main reason for the parallelization of FAKE GAME is to be able to create models on bigger data in reasonable amount of time so bootstrap sampling would be just a computational overhead since it shouldn't make a significant difference [69].

In the **Map** job, the first step consists of converting H2O's local data in to FAKE GAME's internal data structure. H2O uses distributed key-value

storage. At the beginning, each **Map** job gets a chunk of data. It can be part of a single column (for each mapper different) or a block of subset of rows with all columns depending on its type signature. In the FAKE GAME model, I use the latter and transform it to a matrix-like data structure for use in FAKE GAME.

Next step in **Map** job consists of training FAKE GAME models. Each model uses the same template, however, the resulting models can be quite different. One of the models in FAKE GAME is a model that uses genetic programming to evolve ensembles on a given data, which often results in different models. Another important thing to mention is that the classification model in FAKE GAME creates one regression model for each class. This is shown in Figure 2.1.

In the **Reduce**-phase, the models are collected from different **Map** jobs.

Prediction is done by either voting or averaging depending on task. Voting is used for classification tasks and averaging is used for regression tasks.

Since templates are essential for usage of H2O's FAKE GAME model, I will briefly describe one template that was used to produce one of the best performing models on Airlines dataset. FAKE GAME uses XML files as its templates.

On line 7, number of classifiers used in the resulting model (from each **Map** job) is specified. Models are selected from the list beginning with line 9. On line 8, method of selection of resulting models is specified, however, in this case there is only one model to choose from, i.e., the `Boosting{5x ClassifierModel{<outputs>x SigmoidNorm}}` ensemble. On lines 10–38, you can see, how it is possible to embed one ensemble to another one. On line 17, `ClassifierModel` is used to wrap regression models (`SigmoidNorm`). This is done by creating as many regressions models as response categories (one for each class).

```
1  <configuration.classifiers.ensemble.ClassifierCascadeGenProbConfig>
2    <classRef>game.classifiers.ensemble.ClassifierCascadeGenProb</classRef>
3    <description>CascadeGenProb{5x Boosting{5x
4                 ClassifierModel{<outputs>x SigmoidNorm}}}</description>
5    <maxLearningVectors>-1</maxLearningVectors>
6    <maxInputsNumber>-1</maxInputsNumber>
7    <classifiersNumber>5</classifiersNumber>
8    <baseClassifiersDef>RANDOM</baseClassifiersDef>
9    <baseClassifiersCfgs>
10     <configuration.classifiers.ensemble.ClassifierBoostingConfig>
11       <classRef>game.classifiers.ensemble.ClassifierBoosting</classRef>
12       <maxLearningVectors>-1</maxLearningVectors>
13       <maxInputsNumber>-1</maxInputsNumber>
14       <classifiersNumber>5</classifiersNumber>
15       <baseClassifiersDef>RANDOM</baseClassifiersDef>
16       <baseClassifiersCfgs>
17         <configuration.classifiers.single.ClassifierModelConfig>
18           <classRef>game.classifiers.single.ClassifierModel</classRef>
```

```
19          <description>SigmoidNorm classifier</description>
20          <maxLearningVectors>-1</maxLearningVectors>
21          <maxInputsNumber>-1</maxInputsNumber>
22          <baseModelsDef>UNIFORM</baseModelsDef>
23          <baseModelCfgs>
24            <configuration.models.single.SigmoidNormModelConfig>
25              <classRef>game.models.single.SigmoidNormModel</classRef>
26              <maxLearningVectors>-1</maxLearningVectors>
27              <maxInputsNumber>-1</maxInputsNumber>
28              <trainerClassName>QuasiNewtonTrainer</trainerClassName>
29              <trainerCfg class="configuration.game.trainers.QuasiNewtonConfig">
30                <rec>10</rec>
31                <draw>10</draw>
32                <forceAnalyticHessian>false</forceAnalyticHessian>
33              </trainerCfg>
34              <validationPercent>30</validationPercent>
35              <validationEnabled>true</validationEnabled>
36            </configuration.models.single.SigmoidNormModelConfig>
37          </baseModelCfgs>
38        </configuration.classifiers.single.ClassifierModelConfig>
39      </baseClassifiersCfgs>
40    </configuration.classifiers.ensemble.ClassifierBoostingConfig>
41  </baseClassifiersCfgs>
42 </configuration.classifiers.ensemble.ClassifierCascadeGenProbConfig>
```

2.2 Benchmarker

Second part of my assignment was to evaluate anytime performance of models implemented in H2O and FAKE GAME and significance of hyper-parameter optimization.

In order to do that, I have created a simple benchmarking environment in Python. From now on, I will call it as the Benchmarker.

Benchmarker has the ability to benchmark any supervised machine learning algorithm in H2O. In addition, it can also run H2O Ensemble, which is, at the time of writing this thesis, implemented in R only.

Benchmarker was created in order to test anytime properties, however, at the time of implementation H2O did not support pausing and resuming the training, nor did it support setting a time constraint as a stopping criterion for training. The former might appear in future versions of H2O.

H2O is incapable of anytime learning, but it can be to some degree emulated by using variously sized subsets of a dataset and through that regulate the duration of training. More on that in chapter 3 describing methodology.

Benchmarker is capable of benchmarking anytime properties, meta-optimization which will be mentioned later, and producing plots. The most significant part of Benchmarker is the part used for benchmarking anytime properties. First of all, a dataset is randomly divided into 3 subsets. This is done by creating a vector of length same as the number of rows in the whole dataset. The

vector contains samples from uniform distribution between $[0, 1]$. Then the *training set* contains all rows that correspond to the values of the vector that are lower than `train_ratio`, which is defined in the experiment (either an instance of a Python class or YAML file). The *validation set* corresponds to rows that had their corresponding values in the vector between `train_ratio` and `validation_ratio`. The remaining rows are used for *test set*. To be clear, let's assume I have m rows of dataset D then training set, validation set and test set are defined as follows:

$$\begin{aligned} R &\in \mathcal{U}^m(0,1) \\ \text{training set} &= \{D_i | i \in \{1,\dots,m\} : R_i < \texttt{train_ratio}\} \\ \text{validation set} &= \{D_i | i \in \{1,\dots,m\} : R_i \geq \texttt{train_ratio} \wedge \\ &\qquad\qquad R_i < \texttt{validation_ratio}\} \\ \text{test set} &= \{D_i | i \in \{1,\dots,m\} : R_i \geq \texttt{validation_ratio}\} \end{aligned}$$

Since each supervised machine learning algorithm in H2O including the FAKE GAME model requires some parameters, I use the same format for multiple parameters as scikit [70, 71] does in both representations of the experiment. That's the reason, why on the lines 8, 15, 21 and 26, in the following example, I use square brackets around strings. String can be iterated through and the experiment without those brackets would create multiple parameter configurations, each with one letter from the string.

As mentioned before, an experiment can be defined either programmatically in Python or by defining a YAML file. Definition of an experiment in YAML follows:

```
1  !<Experiment>
2  name: HIGGS-100k
3  filename: ../data/HIGGS_100k.csv
4  models:
5    ens: !<SLModelConfig>
6        name: H2O Ensemble
7        params_grid:
8            family: [binomial]
9            learner: [[h2o.glm.wrapper, h2o.randomForest.1,  h2o.gbm.1, h2o.deeplearning.1]]
10           cvControl: [V: 10]
11   deeplearning: !<ModelConfig>
12       base_model: !!python/name:h2o.estimators.deeplearning.H2ODeepLearningEstimator ''
13       name: Deep Learning
14       params_grid:
15           distribution: [bernoulli]
16   gbm: !<ModelConfig>
17       base_model: !!python/name:h2o.estimators.gbm.H2OGradientBoostingEstimator ''
18       name: Gradient Boosting
19       params_grid:
20           ntrees: [50]
```

```
21             distribution: [bernoulli]
22     glm: !<ModelConfig>
23         base_model: !!python/name:h2o.estimators.glm.H2OGeneralizedLinearEstimator ''
24         name: Generalized Linear Model
25         params_grid:
26             family: [binomial]
27     drf: !<ModelConfig>
28         base_model: !!python/name:h2o.estimators.random_forest.H2ORandomForestEstimator ''
29         name: Distributed Random Forest
30         params_grid:
31             ntrees: [50]
32 subsets:
33   min_fraction_denom: 1
34   max_fraction_denom: 1000
35   number_of_samples: 20
36 train_ratio: 0.5
37 validation_ratio: 0.75
38 x: ["lepton pT", "lepton eta", "lepton phi", "missing energy magnitude",
39     "missing energy phi", "jet 1 pt", "jet 1 eta", "jet 1 phi", "jet 1 b-tag",
40     "jet 2 pt", "jet 2 eta", "jet 2 phi", "jet 2 b-tag", "jet 3 pt",
41     "jet 3 eta", "jet 3 phi", "jet 3 b-tag", "jet 4 pt", "jet 4 eta",
42     "jet 4 phi", "jet 4 b-tag", "m_jj", "m_jjj", "m_lv", "m_jlv", "m_bb",
43     "m_wbb", "m_wwbb"]
44 y: "class"
```

The `name` parameter is used only to differentiate different experiments, since multiple experiments can be saved to the same file. The `filename` parameter is a file name of the dataset.

The `models` contains a dictionary of models to be benchmarked. Key is used only internally, in YAML file it doesn't have any significant meaning, however, it can be used in Python to manipulate those models at run-time. In `models` dictionary, only model configurations are specified, currently only two classes are used for model configuration *ModelConfig* and *SLModelConfig* class. The former is used for all models implemented in H2O and the latter is used only for H2O Ensemble, which is based on Super Learner [72] R package, hence the name. Both *ModelConfig* and *SLModelConfig* classes require name attribute that is used to differentiate different models in results and for plotting. Next attribute used by both classes is the `params_grid` attribute, mentioned earlier. In addition, *ModelConfig* class requires `base_model` attribute that is a qualified name of a H2O model to use.

The next dictionary is the `subsets` dictionary. It is used to specify how many subsets are evaluated and the minimal and maximal denominator of subset fraction. The example above will create 20 differently sized subsets, the smallest has 1/1000th of training and validation set. The biggest contains whole training and validation sets as it training and validation sets respectively. And the remaining subsets are evenly (linearly) sized from 1/1000th to 1. It is important to note that test set has always the same size. This is to

ensure that the estimation of generalization error is with low variance, i.e., the smaller fractions could have bigger variance, if I would scale it too.

Parameters `train_ratio` and `validation_ratio` were explained earlier.

Parameters `x` and `y` are used to indicate which columns should be used for training. They can be either specified by the names of the columns or by their indices.

Since Python code used to run an experiment is more verbose, I will demonstrate only a fraction of the equivalent code to the YAML configuration shown above.

```
1   import h2o
2   import Benchmarker.experiment as e
3   import Benchmarker.model_config as m
4
5   h2o.init("127.0.0.1", 54321)
6
7   x = ["lepton pT", "lepton eta", "lepton phi", "missing energy magnitude",
8       "missing energy phi", "jet 1 pt", "jet 1 eta", "jet 1 phi", "jet 1 b-tag",
9       "jet 2 pt", "jet 2 eta", "jet 2 phi", "jet 2 b-tag", "jet 3 pt",
10      "jet 3 eta", "jet 3 phi", "jet 3 b-tag", "jet 4 pt", "jet 4 eta",
11      "jet 4 phi", "jet 4 b-tag", "m_jj", "m_jjj", "m_lv", "m_jlv", "m_bb",
12
13  exp = e.Experiment("HIGGS-100k",filename="../data/HIGGS_100k.csv",
14                     x=x, y='class', train_ratio=0.5, validation_ratio=0.75,
15                     subsets= {
16                          "min_fraction_denom":1,
17                          "max_fraction_denom": 1000,
18                          "number_of_samples": 20
19                      })
20
21  mod = m.ModelConfig("deeplearning",h2o.H2ODeepLearningEstimator,
22                      {"distribution":["bernoulli"]})
23  exp.add("deeplearning", mod)
24
25  mod2 = m.ModelConfig("gbm", h2o.H2OGradientBoostingEstimator,
26                       {"ntrees":[50],
27                        "distribution":["bernoulli"]})
28  exp.add("gbm",mod2)
29
30  ...
31
32  try: # Needed in case something goes wrong so h2o can clean it up
33      results = exp.execute()
34      results.to_csv("{}-data.csv".format(exp.name)) # save measured data in csv
35  except e:
36      print(e)
```

The biggest caveat in implementing Benchmarker was the need to benchmark H2O Ensemble, which is implemented in R only. Since all models use the same subsample, I needed to be able to manipulate R using Python. In

order to do this, I use *rpy2* package, which converts R data structures to Python's and vice versa. Most of the time working with *rpy2* is straight-forward, however, it doesn't enable you to use polymorphism with '$' operator, which complicated some things with H2O, since it uses RESTful API and parses resulting JSON on the fly. This made harder to introspect the data in R.

Benchmarker outputs data in three distinct ways. The first is CSV file, which is created after the whole benchmarking is completed. However, it can happen that the whole experiment won't finish, e.g., the computer runs out of memory etc. To alleviate the risk, Benchmarker creates two so-called journal files[6]. Each measurement is added to the journal files just after the measurement. One of the journal files is plain text file, to which rows are appended in Python representation. The second file is SQLite database file, which is then used for plotting.

During the development of Benchmarker, a new feature request arose. It should be able to benchmark meta-optimization (also known as hyper-parameter optimization) algorithms. Since in H2O isn't implemented any sophisticated meta-optimization apart of grid and random search, benchmarking meta-optimization isn't configurable through YAML files. The meta-optimization part of Benchmarker is implemented and configured in a single Python file. It uses the same structure of output files as the rest of Benchmarker. Benchmarker meta-optimization doesn't use multiple subsamples of data; this is due to a computational complexity of meta-optimization, e.g., one bigger subsamle took almost a week to optimize (300 steps).

Since the results are not easy to comprehend just by looking at the data, Benchmarker has basic plotting support. For each experiment I create a "plot set" file such as below:

```
1  !<PlotSet>
2  name: Higgs-subsets
3  plots:
4    - $accuracyTest_x_trainTime
5    - $accuracyTest_x_subset
6    - $accuracyTest_x_subset_bars
7    - $accuracyTest_x_trainTime_log
8    - $trainTime_x_subset
9    - $totalTime_x_subset
10 query: SELECT * FROM results ORDER BY model
```

In the "plot set", there can specified either some predefined plots prefixed with '$' or definitions of plots can be written inline. Each "plot set" file has to have a `query` and a `name` parameter. The `name` parameter is used for naming the output directory. The `query` parameter is SQLite query, which is used to select results which should be plotted.

[6]the name journal is borrowed from file-systems, such as ext3, since it has a similar purpose

Predefined plots are placed in `Benchmarker/plot_config/` and they are just YAML files. An example is given below.

```
!<Lines>
name: Accuracy test vs train time
group_by: [model, params, cluster_name, nthreads]
legend_labels: !!python/name:Benchmarker.utils.legend_label_fg ''
style: o-
log_x: true
title: Accuracy on Test Data Set vs Training Time
x_col: trainTime
xlab: Training time [log(s)]
y_col: accuracy_test
ylab: Accuracy
```

Inline definition of a plot is in the same format as are those predefined. The only difference is that they are indented accordingly.

Chapter **3**

Methodology

As mentioned in previous chapter, I have conducted 2 distinct types of experiments — evaluating anytime properties of machine learning algorithms and evaluating hyper-parameter optimization.

3.1 Evaluating Anytime Properties

Since H2O doesn't support anytime learning, I had to emulate it by subsampling datasets. In order to do that I have implemented Benchmarker, which is described in previous chapter. It automates the subsampling process and provides an easy way to create new experiment. Due to the overwhelming amount of data, I do not use cross-validation to estimate the generalization properties of the models, but I do use a big test datasets, which make a good estimate of generalization error and, since predicting is in H2O significantly faster than training, it takes a lot less time.

In previous chapter, I have shown, how does Benchmarker split the data. Now, I will describe it little more using an example, where `train_ratio`} is 0.4 and `validation_ratio` is 0.5 and the dataset has $1,000$ rows. Let's assume I have m rows of dataset D then training set, validation set and test set are defined as follows:

$$
\begin{aligned}
R &\in \mathcal{U}^{1000}(0,1) \\
\text{training set} &= \{D_i | i \in \{1,\ldots,1000\} : R_i < 0.4 \\
\text{validation set} &= \{D_i | i \in \{1,\ldots,1000\} : R_i \geq 0.4 \wedge R_i < 0.5\} \\
\text{test set} &= \{D_i | i \in \{1,\ldots,1000\} : R_i \geq 0.5\}
\end{aligned}
$$

It is reasonable to assume that the training set will have approximately 400 rows, the validation set ~ 100 rows and the test set ~ 500 rows. Now, let's say `number_of_samples` $= 10$, `min_fraction_denom` $= 1$ and `max_fraction_denom` $=$ 10, then Benchmarker will benchmark following subsets written as a triplet

(#training rows, #validation rows, #test rows)[7]: (40, 10, 500), (80, 20, 500), (120, 30, 500), (160, 40, 500), ..., (400, 100, 500).

The test set is the same throughout whole benchmarking process. The training subsets are subsets of the training set and each bigger subset will contain the rows in the smaller one. The same applies to validation set. This is mainly to eliminate variance in models trained on small amount of data, e.g., model trained on less data performing better than model trained on more data thanks to the luck of the former model on having better representing training set.

Proportions in previous example are used for all of my anytime experiments, however, number of rows is significantly higher.

3.2 Evaluating Hyper-parameter Optimization

Hyper-parameter optimization was evaluated on subsample with fixed number of rows. Since H2O supports just grid and random search, I implemented simple wrapper that uses PySMAC, which is a simple Python wrapper to SMAC [3], which is implemented in Java. PySMAC requires description of optimized parameters, which are given at appendix B. The same parameter ranges and enumerations are used by random search, which I also had to implement in Benchmarker in order to be able to gain all the results I needed, in other words, to have proper control over the optimization process.

I have benchmarked hyper-parameter optimization on several machine learning algorithms. Benchmarked hyper-parameter optimization methods were random search and SMAC. In order to have comparison, I ran those algorithms several times with default settings, which yielded good results with low variance.

3.3 Datasets

I've conducted all of my experiments on two datasets. The next two sections will describe both datasets.

Both of the datasets are used for binomial classification. This is mainly because H2O Ensemble, which at the time of writing this thesis, did not support multinomial classification. Due to computational requirements, I have randomly sampled $1,000,000$ rows from each dataset. And all of my experiments use such subset.

3.3.1 Higgs Dataset

Higgs dataset [73] is used for binomial classification. The original problem was to assess whether the measurements are from signal process, which produces

[7] the counts are approximate due to the method used for splitting the data

Higgs bosons, or a background process, which does not.

The data were produced using Monte Carlo simulations. The first 21 features are kinematic properties measured by particle detectors in the accelerator. The next 7 are functions of the 21 first features. These are high-level features derived by physicists to help discriminate between the two classes.

Original paper involving this dataset was a benchmark of Bayesian Decision Trees and 5-layer neural network.

3.3.2 Airline Dataset

This dataset contains data about delays of U.S. domestic flights operated by large air carriers.

Original dataset comes from the U.S. Department of Transportation's (DOT) Bureau of Transportation Statistics (BTS), however, I have used dataset, which comes with H2O [74]. It can be easily downloaded by using `./gradlew syncBigdataLaptop` in H2O's source folder, but beware, it will download several GB of data.

It contains 31 columns, though some column are highly correlated with the target column. After elimination of those columns, I have 13 features. Three of those are categorical and the rest is numerical.

It worth noting that different papers use different columns, so it can be troublesome to blindly compare the results with another paper that uses the same dataset.

In this thesis, I did test two different sets of attribute columns and two different response columns. As response columns, I used **IsDepDelayed** and **IsArrDelayed**, both of which are binomial. The difference in attribute columns used for training lies in containing **DepTime** column, which can be compared with **CRSDepTime** and, by using a simple comparison, I am able to get quite good prediction about whether the plane will be delayed either on departure (it isn't 100 %, but close to it) or on arrival. When I include **DepTime** column, I use 13 features, without it only 12.

It may seem like an unreasonable thing to do, include the **DepTime** column, but to my surprise a simple model from FAKE GAME beat state-of-the-art models from H2O.

Chapter **4**

Results

I have conducted experiments to get insight into the scalability of several machine learning algorithms from H2O as well as FAKE GAME's parallel training of FAKE GAME's templates.

I have chosen two public datasets — Airline Delays, which is available through H2O [74], and Higgs [73]. Those datasets are used for binomial classification of selected output attributes.

I have benchmarked FAKE GAME's parallelized templates to models available in H2O implemented using the MapReduce approach. Models originating from FAKE GAME are prefixed in results with **ENS**. **Generalized Linear Model** [75] is using logistic regression to deal with classification problems. **Naive Bayes** classifier assumes independence of input attributes and classifies based on conditional probabilities obtained from training data. **Deep learning** [41] is a feedforward neural network with various activation functions in neurons. **Distributed Random Forest** and **Gradient Boosted Machine** [55] are ensembles based on **CART** decision trees. **H2O Ensemble** is an ensemble classifier called Super Learner by [76].

In all my results, I will present only **test accuracy**, however, Benchmarker saves also training and validation accuracy. In addition to that, it saves several more performance measures including AUC, MSE, AIC (where appropriate), and more.

4.1 Anytime Learning

Following experiments use 1,000,000 randomly selected rows from each dataset. Then 50 % rows is randomly selected as test set and the rest is then sampled to subsets of growing size to examine scalability of algorithms. This sampled data are randomly split to training set (80 %) and validation set (20 %). For more details see chapter 3.

Some algorithms are much faster than the others, which is the reason, why I will show most of my plots with logarithm of time instead of time. Such plots

have x axis labelled with `[log(s)]`.

4.1.1 Higgs

At first, I have examined scalability of algorithms on the Higgs dataset. Figure 4.1 shows logarithm of learning time and performance of individual algorithms executed on subsets of growing size. The best performance was achieved by Deep Learning, which was also reasonably fast. Gradient Boosting is faster, but it does not have capacity to improve with bigger data subsets. Distributed Random Forest is also reasonably accurate and fast, but it is dominated by Deep Learning. Ensembles produced from templates are not very competitive on this dataset. Only complex hierarchical ensemble of decision trees is approaching the performance of Distributed Random Forest, but it is much slower.

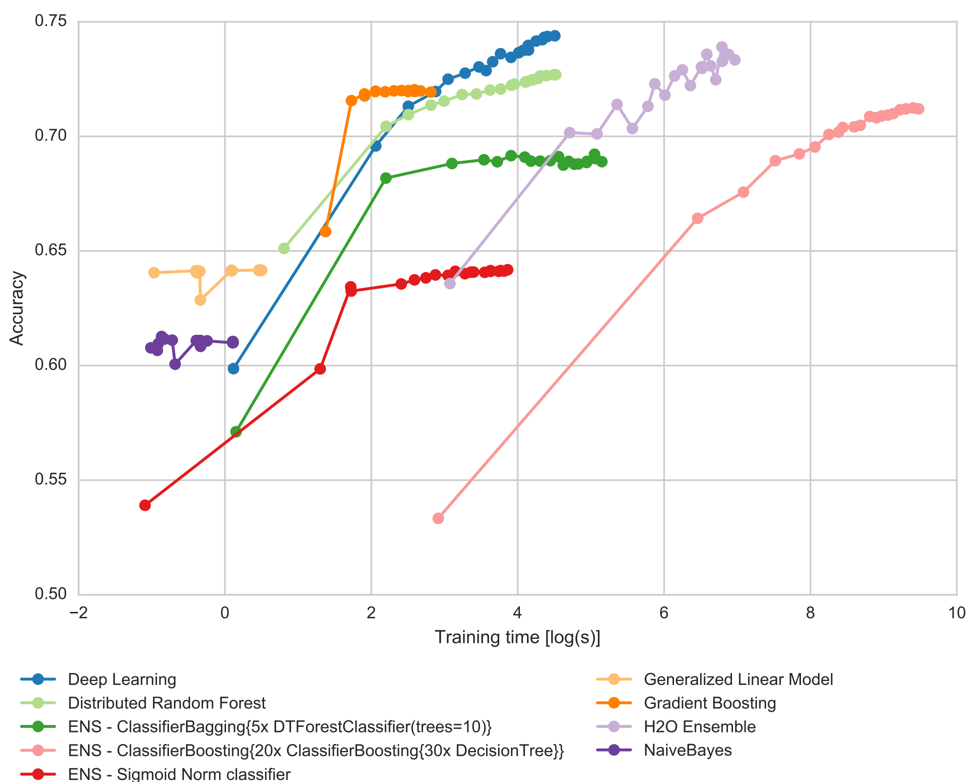


Figure 4.1: Comparison of several machine learning algorithms in H2O trained on samples with various sizes from Higgs dataset [73]

It can be seen that selected templates from FAKE GAME in H2O do not generally perform very well on Higgs dataset, when compared with H2O's original machine learning algorithms. To get more complete picture look at Figure 4.2, where all tested machine learning algorithms are shown.

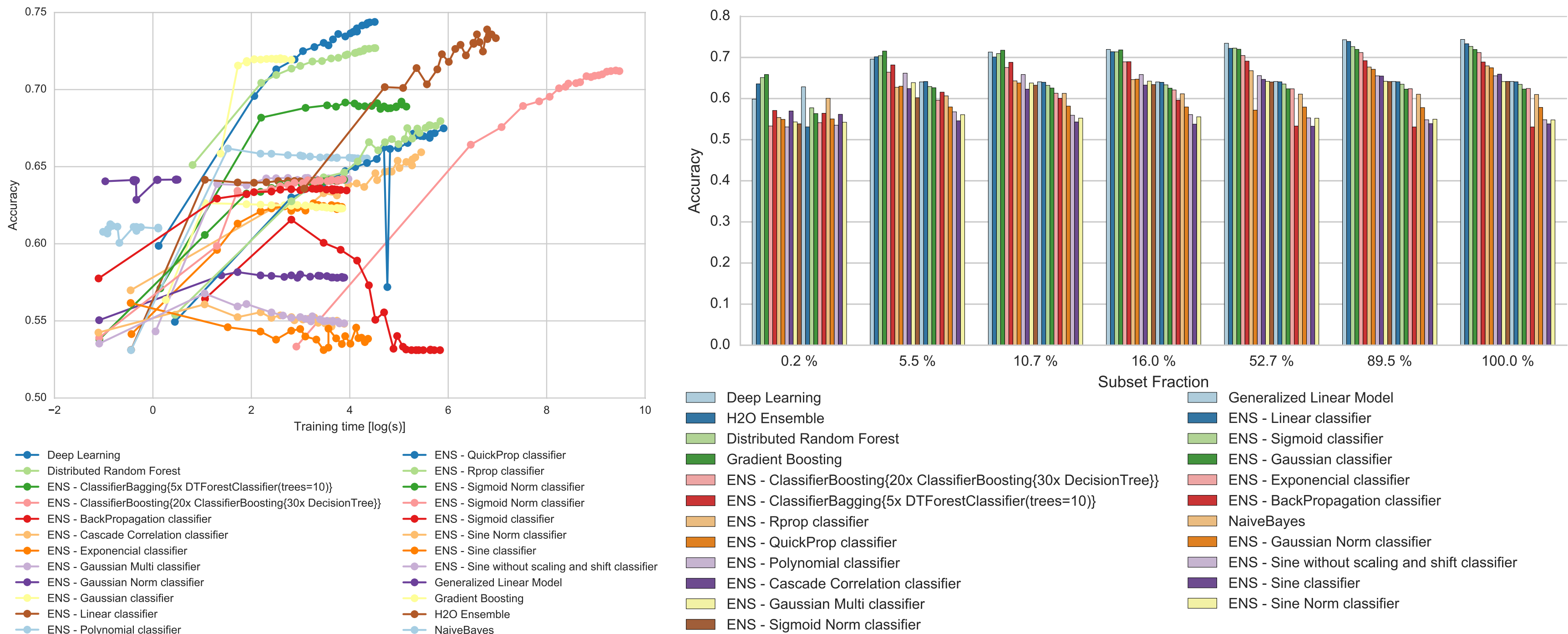


Figure 4.2: Comparison of several machine learning algorithms in H2O trained on samples with various sizes from Higgs [73] dataset. Since humans are able to discriminate only between a few colors, I have decided to show both lines plot and bar plot with a few easily recognizable colors. As a consequence those colors repeat for multiple models. *Bar plot* has the advantage that the *bars are in the same order as in the legend* making it easy to discriminate between various models. Using accuracy from the bar plot you can, with some effort, discriminate between different models even in the lines plot. Figure 4.1 is based upon subset of models of the same data.

4.1.2 Airline

As mentioned in chapter 3, there are 4 different scenarios which I evaluated. These scenarios differ in columns used for training and predicting. It is possible to predict either **IsDepDelayed** or **IsArrDelayed**. The difference in training columns is the inclusion or exclusion of **DepTime** column. However, scenarios without **DepTime** column resulted very similarly, for this reason I will present only one of them — the one predicting **IsArrDelayed**.

To summarize, in the next 3 sections, I will show results for predicting **IsDepDelayed** with **DepTime**, and predicting **IsArrDelayed** with and without **DepTime**.

4.1.2.1 Predicting IsDepDelayed with DepTime

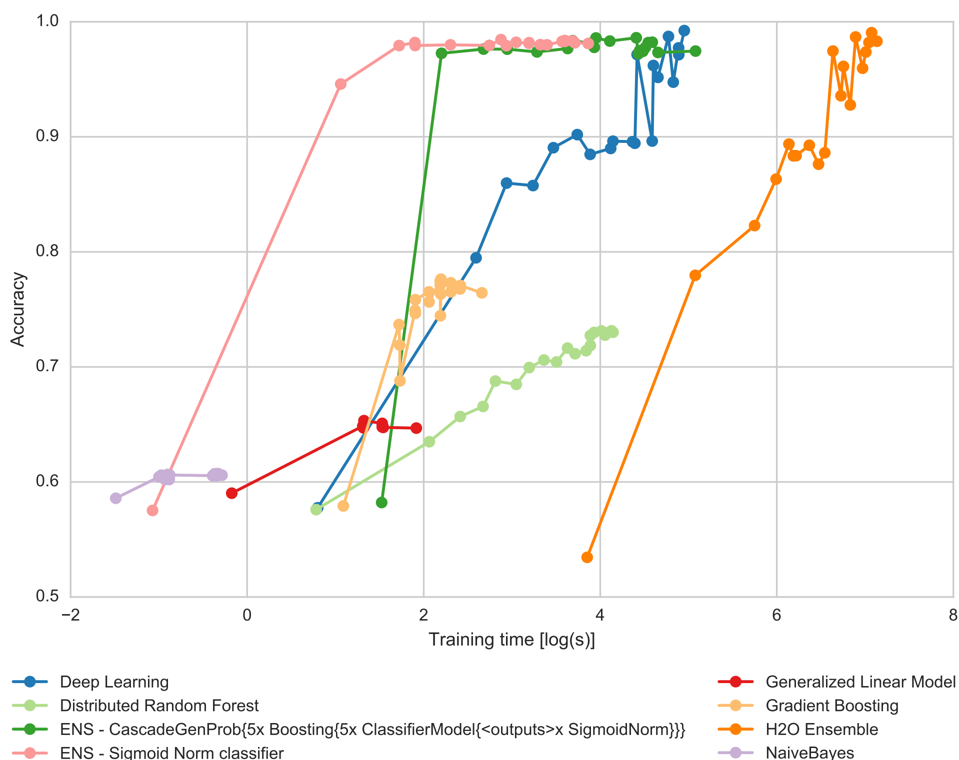


Figure 4.3: Predicting IsDepDelayed on Airline dataset: comparison of algorithms in H2O trained on subsamples of increasing size.

Looking at the Figure 4.3, where arrival delay is predicted on the Airlines dataset, results are completely different. FAKE GAME ensembles are both more accurate and faster. The difference is so big that I decided to analyze these results further (see Section 5.1).

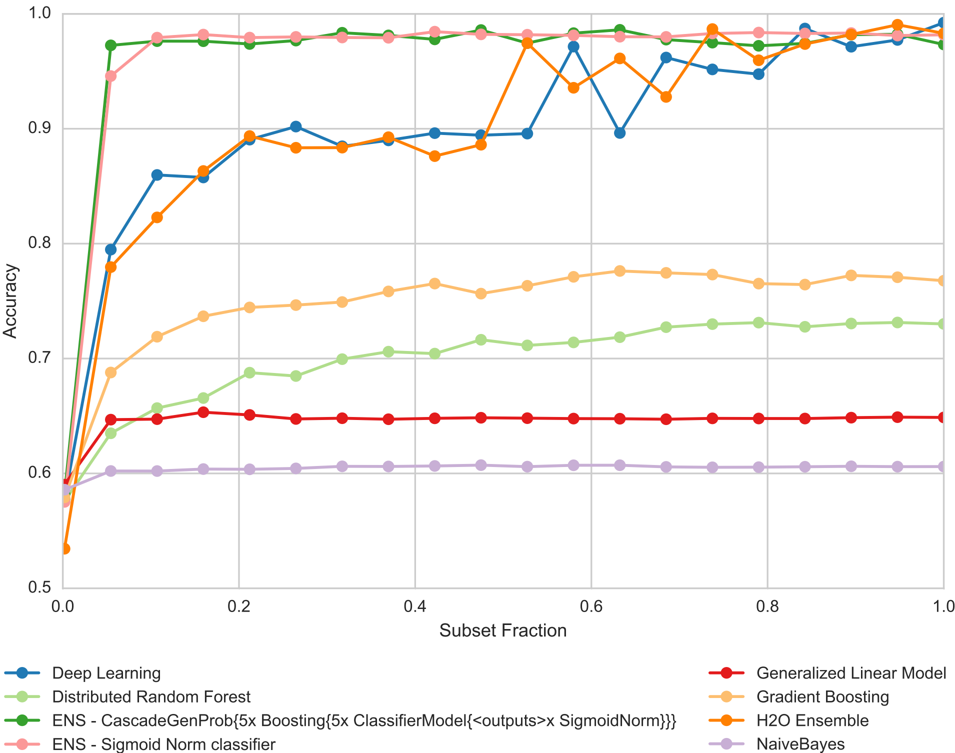


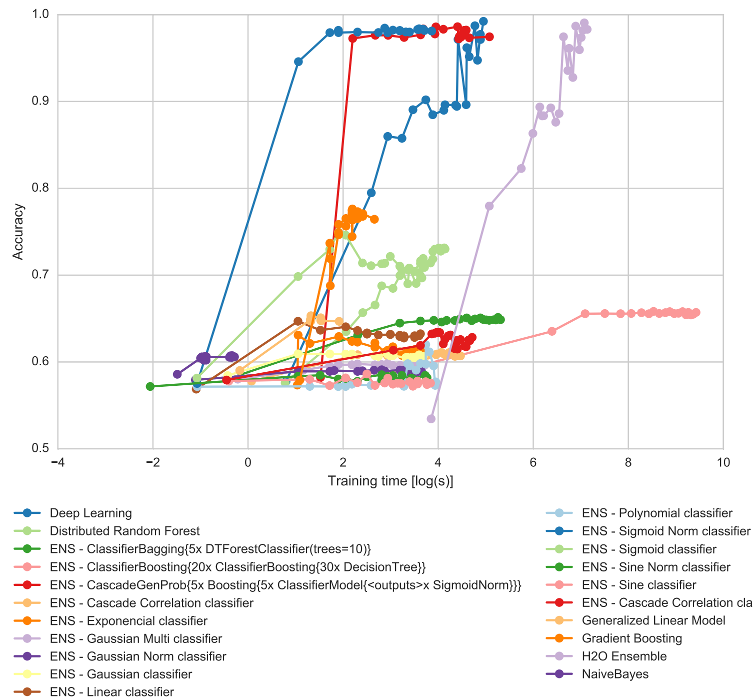
Figure 4.4: Predicting IsDepDelayed on Airline dataset with DepTime: comparison of algorithms in H2O trained on subsamples of increasing size.

The prediction problem in this scenario is quite trivial, because you can obtain the target[8] (is departure delayed?) by comparing **DepTime** and **CRSDepTime** attribute. Unsurprisingly, this was the most successful scenario of all 4.

It is quite surprising that most of the classifiers are mislead by other attributes and fail to discover this simple relationship. Figures 4.4, 4.3 show that FAKE GAME's simple ensembles based on Sigmoidal model are able to learn fast and solve the problem even on small subsets. H2O Ensemble and Deep Learning discovered the relationship on 250 thousand instances and their learning time was significantly higher.

As with Higgs dataset, I will also show plots with all tested algorithms (Figure 4.5). In Figure 4.5 it can be seen that on the full dataset, machine learning algorithms either produce a model that can find this simple decision line (comparison of **DepTime** and **CRSDepTime** attribute), which results in two clusters — high performing models and low performing models. Those two clusters are divided by a large gap in accuracy.

[8]you can get $\sim 99\,\%$ accuracy using the comparison



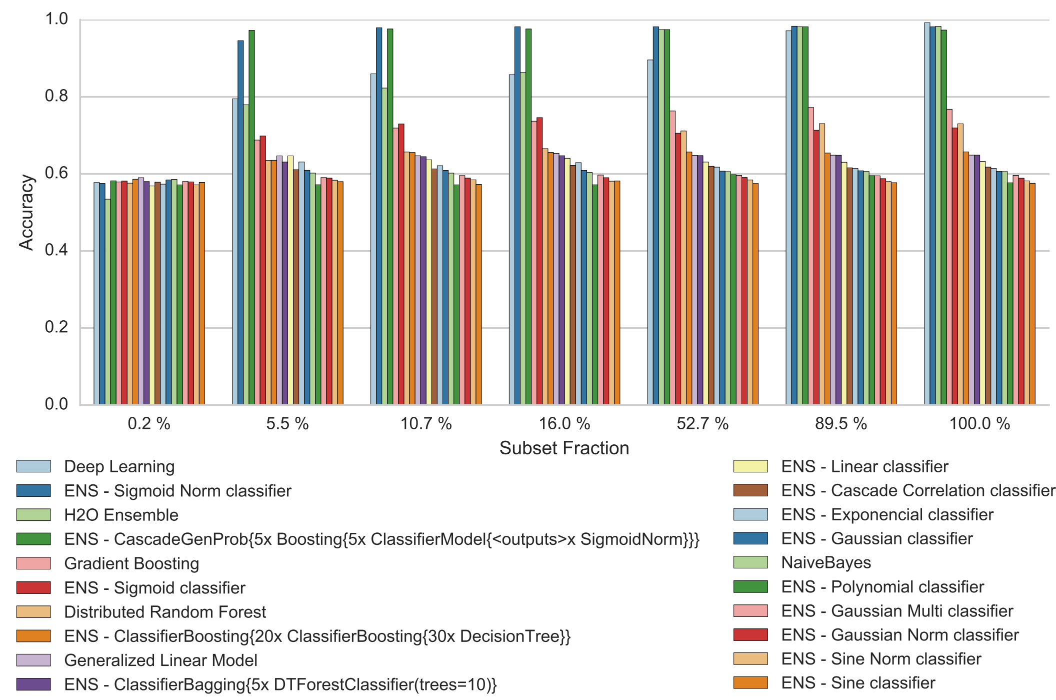


Figure 4.5: Comparison of several machine learning algorithms in H2O trained on samples with various sizes from Airline dataset (Predicting IsDepDelayed with DepTime). Since humans are able to discriminate only between a few colors, I have decided to show both lines plot and bar plot with a few easily recognizable colors. As a consequence those colors repeat for multiple models. *Bar plot* has the advantage that the *bars are in the same order as in the legend* making it easy to discriminate between various models. Using accuracy from the bar plot you can, with some effort, discriminate between different models even in the lines plot. Figure 4.3 is based upon subset of models of the same data.

4.1.2.2 Predicting IsArrDelayed with DepTime

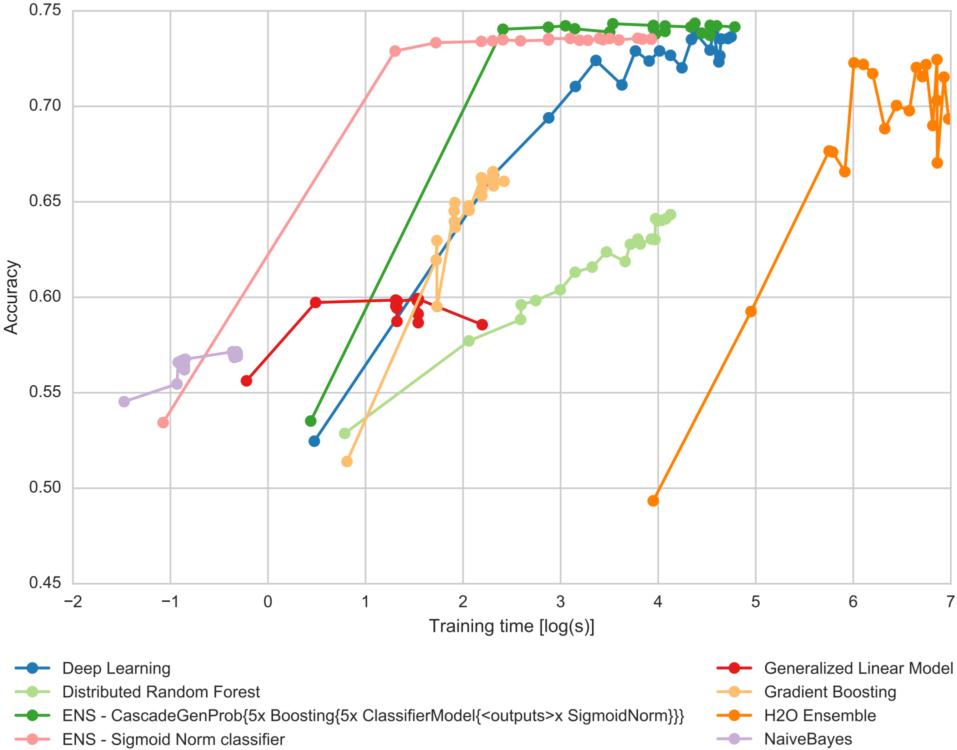
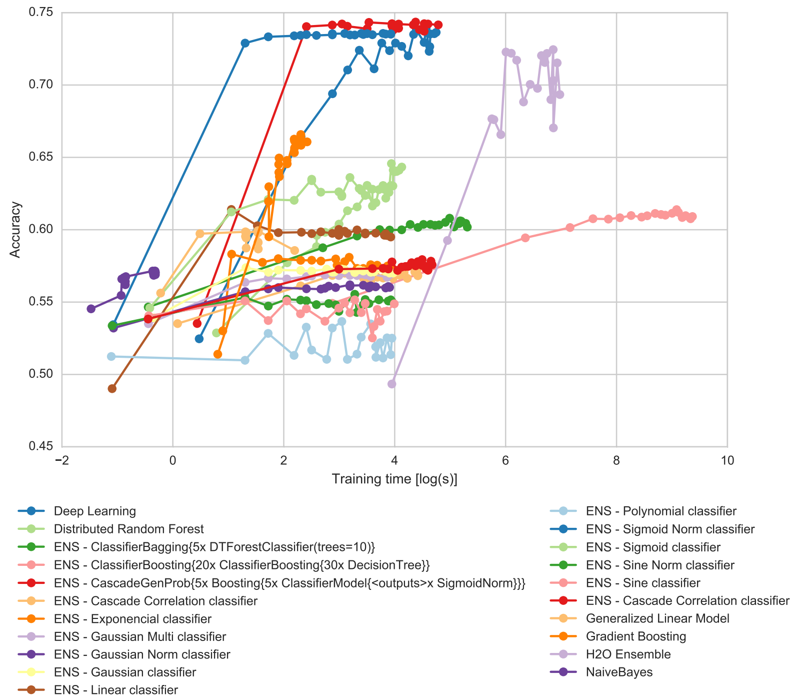


Figure 4.6: Predicting IsArrDelayed on Airline dataset: comparison of algorithms in H2O trained on subsamples of increasing size.

This scenario was significantly harder, which resulted in maximal accuracy around 75 %. Unlike previous scenario, no algorithm from H2O outperformed FAKE GAME's ensemble, though there was only a small gap in accuracies. Another observation is that differences across accuracies of models are not as significant as in the previous scenario.

On the other hand, similarly to the previous scenario, the two best FAKE GAME's models are significantly faster than any other model that yields similar accuracy. **Deep Learning** and **H2O Ensemble** are only two models from H2O that have accuracies greater than 70 %, however, **H2O Ensemble** performed rather inconsistently yielding a model on full dataset that performed lower than 70 %. And in all of the evaluated machine learning algorithms, it was the second slowest machine learning algorithm. The only slower machine learning algorithm is FAKE GAME's template **ClassifierBoosting{20× ClassifierBoosting{30× DecisionTree}}**.



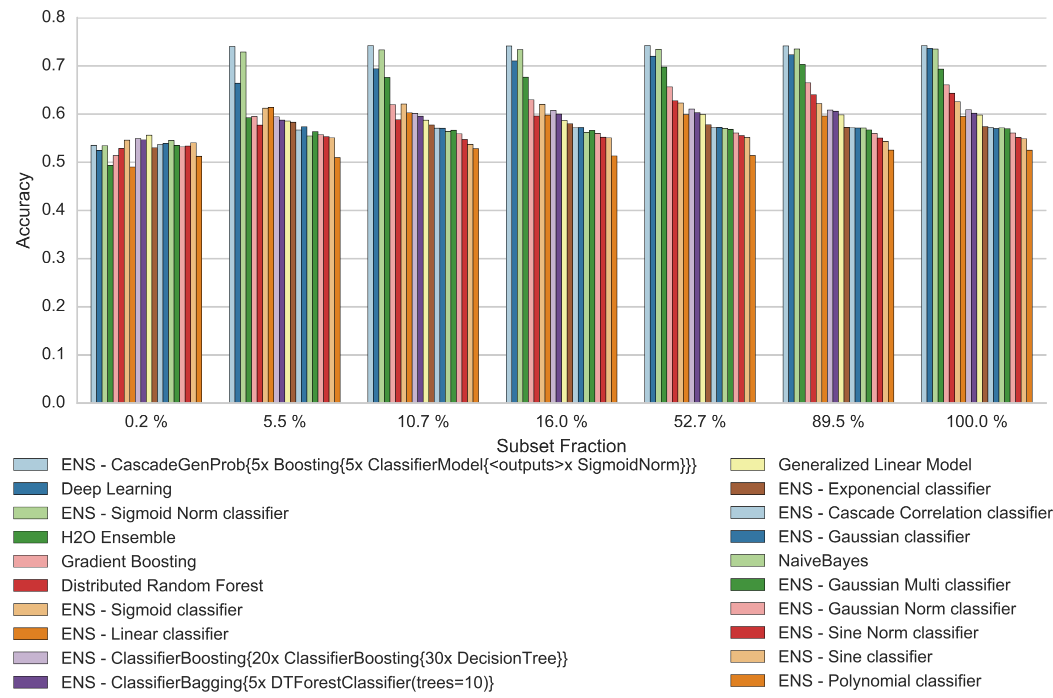


Figure 4.7: Comparison of several machine learning algorithms in H2O trained on samples with various sizes from Airline dataset (Predicting IsArrDelayed with DepTime). Since humans are able to discriminate only between a few colors, I have decided to show both lines plot and bar plot with a few easily recognizable colors. As a consequence those colors repeat for multiple models. *Bar plot* has the advantage that the *bars are in the same order as in the legend* making it easy to discriminate between various models. Using accuracy from the bar plot you can, with some effort, discriminate between different models even in the lines plot. Figure 4.6 is based upon subset of models of the same data.

4.1.3 Predicting IsArrDelayed without DepTime

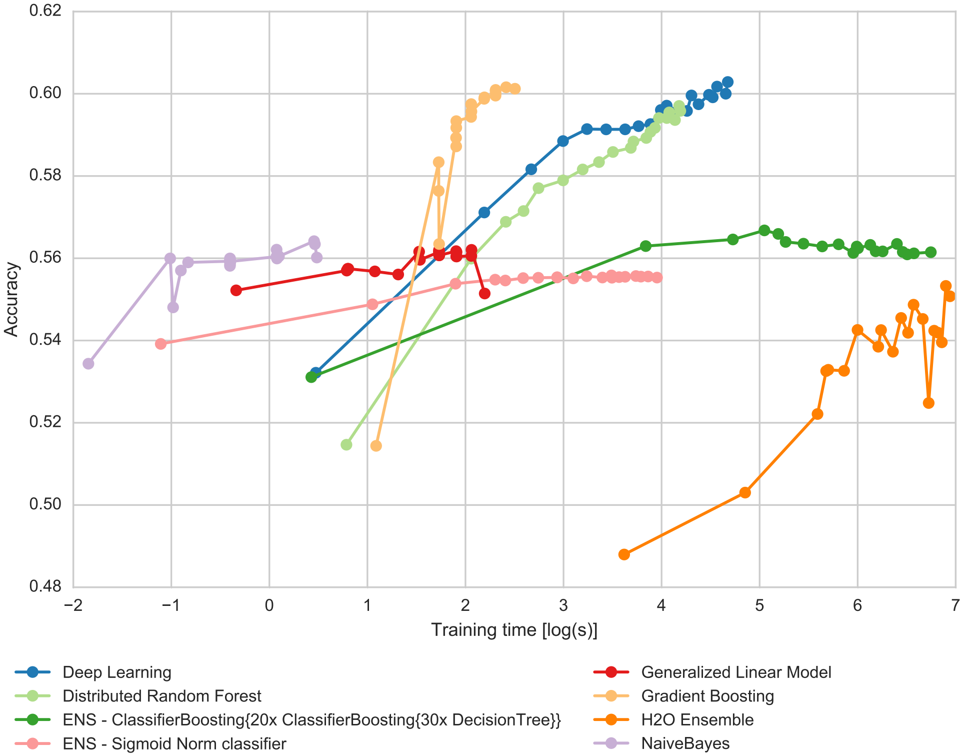


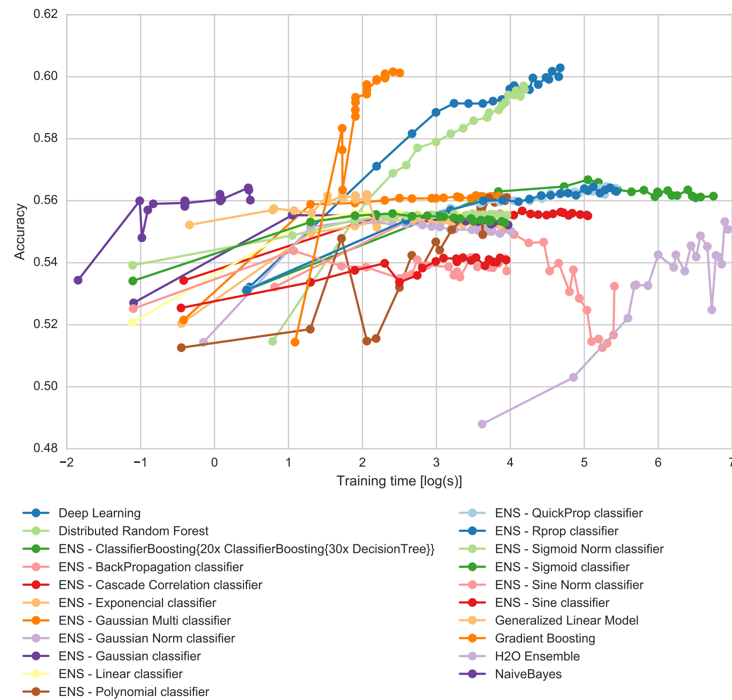
Figure 4.8: Predicting IsArrDelayed on Airline dataset: comparison of algorithms in H2O trained on subsamples of increasing size.

This scenario resulted in the worst performance, which is expected, given that the previous scenarios could compare **DepTime** and **CRSDepTime** attribute.

Surprisingly, **H2O Ensemble**, which is based on Super Learner, is among the worst performing machine learning algorithms. It is surprising, because it should asymptotically perform at least as good as the best model in it, which in this case is a Deep Learning model.

In this scenario, all FAKE GAME's models performed similarly badly. Difference between the bad performing models is negligible.

It can be seen on Figure 4.9 that models on the full dataset tend to create two clusters, similarly to previous scenarios. However, in this case, in the good performing cluster is not a single model from FAKE GAME. It contains only **Deep Learning**, **Gradient Boosting Machine**, and **Distributed Random Forest**. This could suggest that those algorithms found a similar decision boundary.



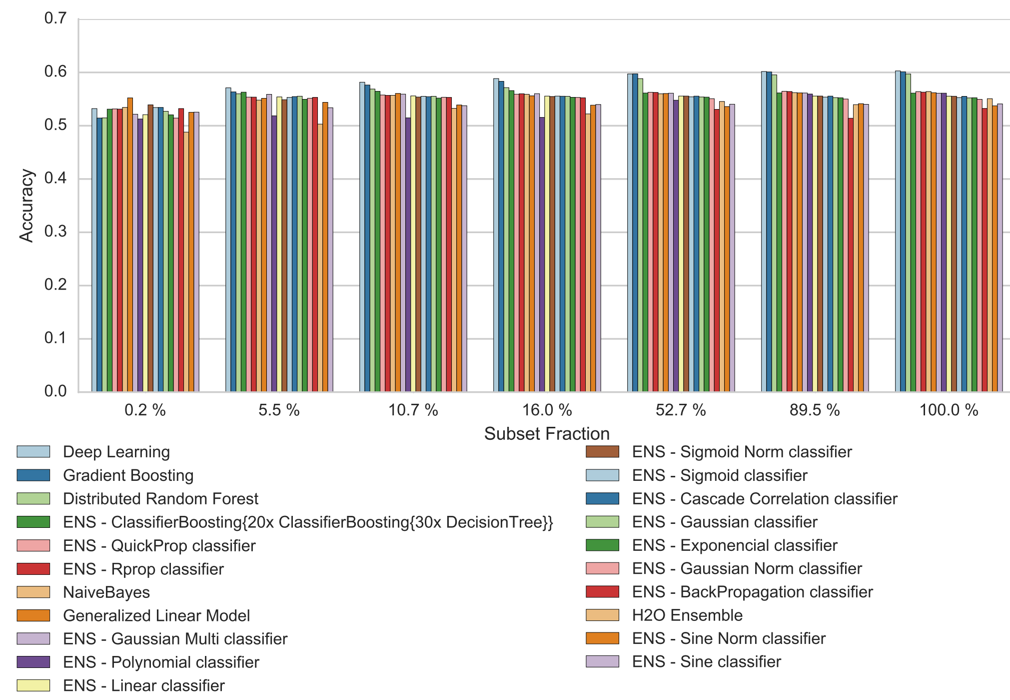


Figure 4.9: Comparison of several machine learning algorithms in H2O trained on samples with various sizes from Airline dataset (Predicting IsArrDelayed without DepTime). Since humans are able to discriminate only between a few colors, I have decided to show both lines plot and bar plot with a few easily recognizable colors. As a consequence those colors repeat for multiple models. *Bar plot* has the advantage that the *bars are in the same order as in the legend* making it easy to discriminate between various models. Using accuracy from the bar plot you can, with some effort, discriminate between different models even in the lines plot. Figure 4.8 is based upon subset of models of the same data.

4.2 Hyper-parameter optimization

Hyper-parameter optimization is another task, I have dealt with in this thesis. Unfortunately, hyper-parameter optimization is slow, due to the need of running machine learning algorithms with multiple configuration. This could be mitigated if I could evaluate machine learning algorithms sooner than they finish training, the same way as Hyperband (see section 1.5.5).

To speed up the hyper-parameter optimization, I have taken a smaller sample of data — 100k rows, to be precise. Then I used half of this sample as a training set, and 25 % and 25 % as a validation, and a test set, respectively. Those subsets were created by sampling without replacement.

Even with such modification, hyper-parameter optimization for one dataset took almost a week. And since some machine learning algorithms have hyper-parameter that influences memory usage, namely, Distributed Random Forest, Gradient Boosting Machines, it took a lot of time to figure out, what should be maximal acceptable values of several hyper-parameters, to be able to finish the hyper-parameter optimization, i.e., to not get killed by OOM killer. Those are the reasons, why I have optimized only two out of four scenarios — Higgs dataset and Airline dataset with DepTime predicting IsDepDelayed.

4.2.1 Higgs dataset

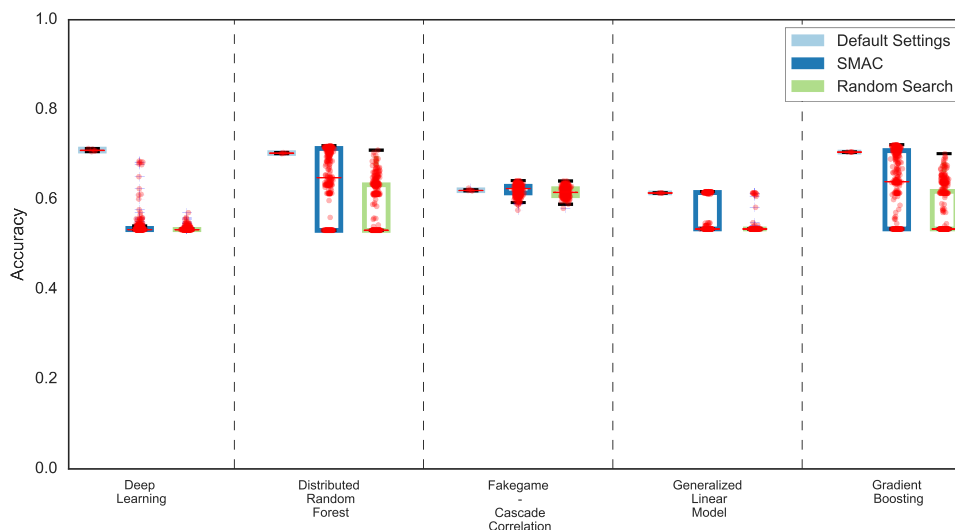


Figure 4.10: Hyper-parameter optimization of several machine learning algorithms using Random Search and SMAC on Higgs dataset. All configurations are plotted in order to gain some insight into hyper-parameter × test accuracy landscape.

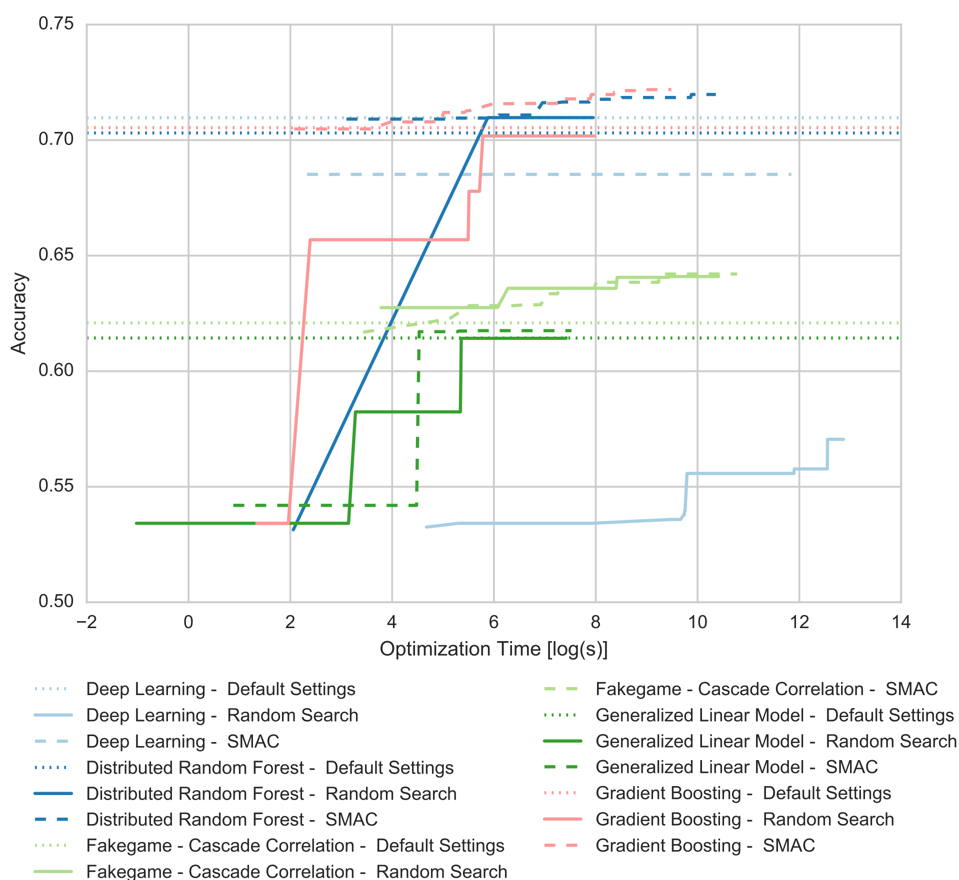


Figure 4.11: Hyper-parameter optimization of several machine learning algorithms using Random Search and SMAC on Higgs dataset. In this figure, I present best test accuracy for each time point.

4.2.2 Airline dataset

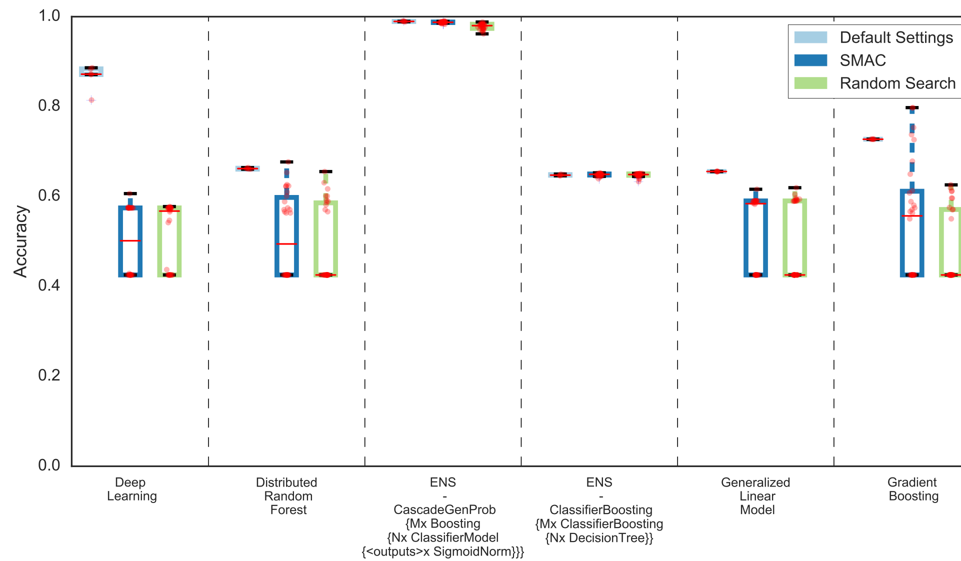


Figure 4.12: Hyper-parameter optimization of several machine learning algorithms using Random Search and SMAC on Airline dataset. All configurations are plotted in order to gain some insight into hyper-parameter × test accuracy landscape.

Both scenarios resulted very similarly. Hyper-parameter optimization didn't work better than default settings in some cases. In other cases, SMAC behaved better than random search, but the difference wasn't significant.

A hypothesis, why are default settings often better than hyper-parameter optimized versions is given in the following chapter.

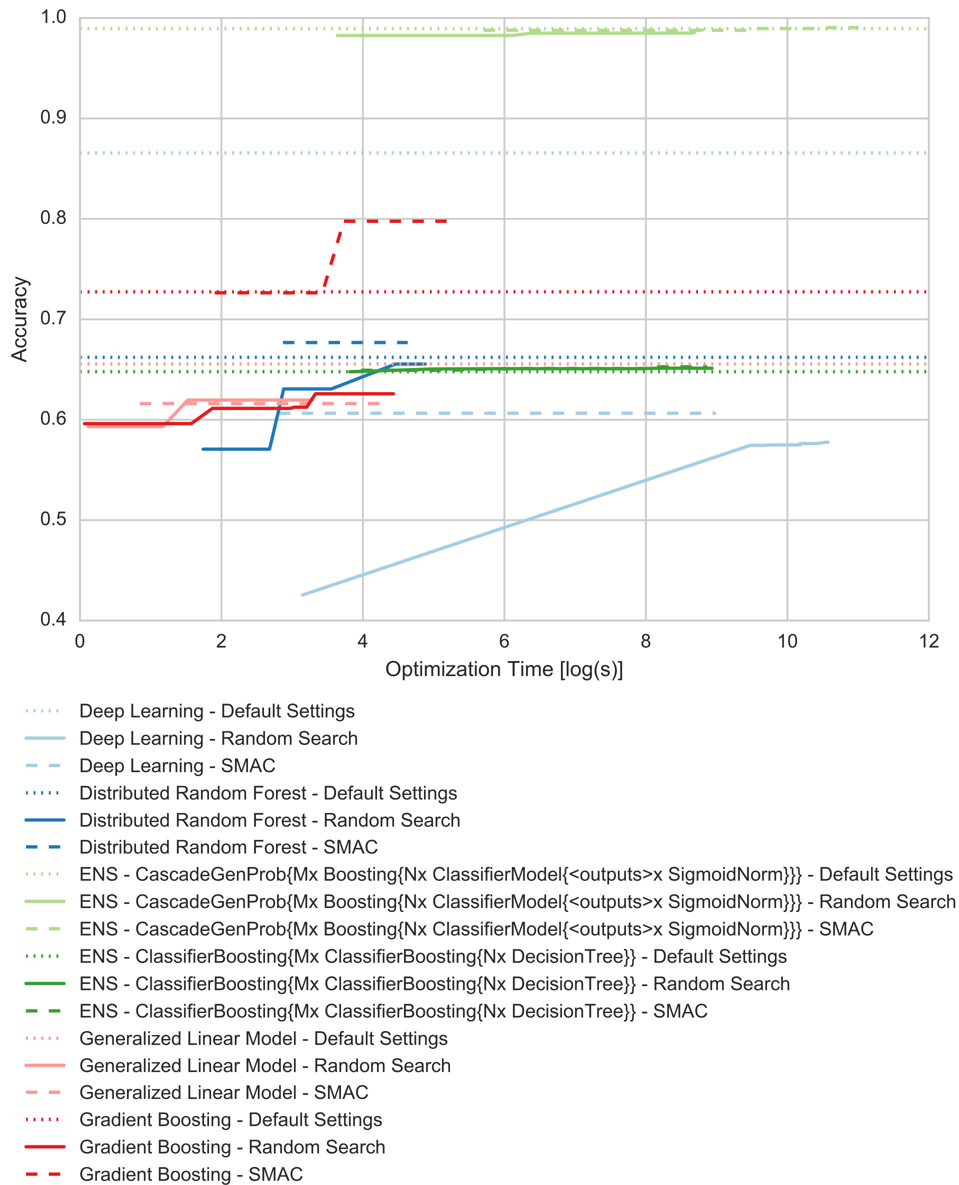


Figure 4.13: Hyper-parameter optimization of several machine learning algorithms using Random Search and SMAC on Airline dataset. In this figure, I present best test accuracy for each time point.

Chapter **5**

Discussion

In the previous chapter, I have presented results, some of which deserves a further discussion. Those are the unreasonable effectiveness of sigmoid-based ensembles and hyper-parameter optimization. In this chapter, I will discuss both of them.

5.1 Airline Dataset and Sigmoid-based Ensembles

SigmoidNorm model $y = \left(1 + \exp\left(\sum_{i=0}^{m-1} a_i x_i + a_m\right)\right)^{-1}$, which is in all successful ensembles in scenarios with included **DepTime** attribute, is too simple to overfit even on smaller data, which is one of the reasons, why it is successful straight from the start.

H2O FAKE GAME **SigmoidNorm** model is trained in parallel fashion, which is more closely described in Section 2.1. When the data are loaded, H2O distributes them on its nodes. Then the data are split into chunks on each node. Subsequently, a single FAKE GAME **SigmoidNorm** classifier is trained for each chunk, in parallel. Since it is a classifier, it creates two **SigmoidNorm** regression models, one for each class. Prediction is done by voting, i.e., each classifier predicts a class and the class, which was predicted most frequently, i.e., has the most votes, is the resulting one.

One of the questions I will try to answer in this section is, why does other machine learning algorithms perform so badly, given the simplicity of comparing two attributes, namely, **DepTime** and **CRSDepTime**.

Using a simple comparison, i.e., when **DepTime** $>$ **CRSDepTime** then predict **IsDepDelayed** to be `True`, I got accuracy 0.98706 on a randomly selected set of 1,000,000 observations. A slight modification of not using a strict inequality, i.e., **DepTime** $\geq$ **CRSDepTime** yielded prediction with accuracy 0.777506 on the same set.

This suggests that the decision boundary is close to be linear and that it can be very sensitive to minuscule shifts. The next step is to inspect decision

boundaries of some generally successful machine learning algorithms as well as **SigmoidNorm**. This is visualized in Figure 5.1.

Plots of class probabilities and decision boundaries helped me to reveal the reason of poor performance of decision tree based ensembles. Figure 5.1 shows that successful classifiers (ensemble of **SigmoidNorm** models, **Deep Learning**) were able to identify simple relation of two input attributes to departure delay prediction. The relationship (decision boundary) is hard for decision trees to model with their orthogonal decisions. It is also impossible to solve for **Naïve Bayes** classifier assuming independence of input attributes.

Evidently, FAKE GAME was able to discover very efficient template for this trivial problem. It can be also seen that unlike **Deep Learning** FAKE GAME's **SigmoidNorm** is much closer to the simple comparison.

Logically, it is obvious that when a plane is delayed one unit of time or ten, it is still delayed, however, looking at Figure 5.1, I can see that **Deep Learning** predicts the delay only in a small stripe. For this reason, I would say that **SigmoidNorm** generalizes better on this problem than **Deep Learning**. Although, there are some outliers, which can be learnt by **Deep Learning** and not by **SigmoidNorm**.

5.2 Hyper-parameter Optimization

Hyper-parameter optimization performed worse than I anticipated. I believe that the reason for this lies in the automatic tuning of hyper-parameters during the training, which is on by default in most of the algorithms in H2O. Since I optimized even those, automatically-tunable, parameters, I effectively discarded the information, which could be discovered during the training. The list of all parameters that were optimized and their ranges are available in appendix B.

Figure 4.12 shows that most of the H2O algorithms are very sensitive to improper parameter settings. **Deep Learning** has the most automatically-tunable parameters, which in turn corresponds to worse performance with the optimized parameters.

However, Figure 4.12 shows that SMAC consistently tests mostly better configurations than random search does. Similarly, negative impact was observed for **Generalized Linear Model**.

For **Gradient Boosting** and **Distributed Random Forest**, optimization discovered better performing configurations, however, the difference was not significant. One of the problems, with hyper-parameter optimization of ensembles of trees, is that with more trees the accuracy usually gets higher, which yields models that are slow to train and predict. This could be mitigated by using similar approach to TB-SPO [65], which takes into account, in addition to the model performance (accuracy), the duration of model evaluation.

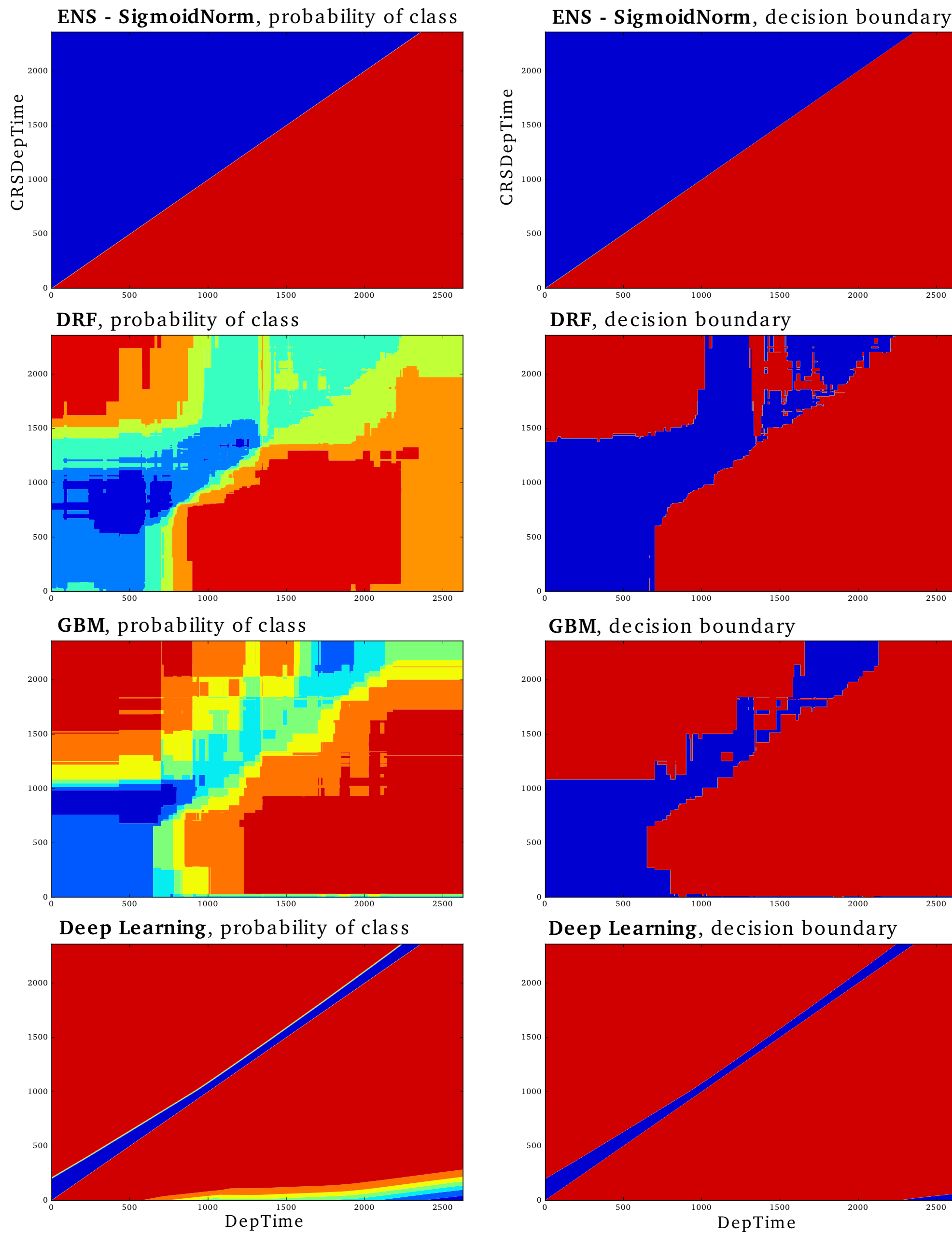


Figure 5.1: Decision boundaries of algorithms on problem of predicting aircraft departure delay. Simple ensemble of sigmoid classifiers was able to generalize the relationship well, whereas decision tree based ensembles overfitted the data. Deep Learning discovered the relationship only on large data samples.

I also tried to optimize number of models in FAKE GAME's hierarchical ensembles, but apparently it had almost no effect on accuracy, but it usually chosen the configuration with more base models than the default configuration, making it run slower.

Conclusion

In this thesis, I explored machine learning algorithms as well as hyper-parameter optimization of those algorithms. To evaluate generalization abilities of those machine learning algorithms, I have designed and implemented a tool called Benchmarker. Using this tool, I have evaluated anytime properties of supervised machine learning algorithms implemented in H2O, as well as, newly implemented H2O FAKE GAME model. In addition, I used parts of Benchmarker to create simple script to evaluate hyper-parameter optimization. This script is included in Benchmarker package.

I have compared all supervised machine learning algorithms from H2O, namely, **Deep Learning**, **Distributed Random Forest**, **Gradient Boosting Machines**, **Naïve Bayes**, and several templates from **FAKE GAME**.

In some cases, the newly implemented H2O **FAKE GAME** model was significantly faster and the test accuracy was, in cases with less data, much higher than the one from the state-of-the-art algorithms from H2O. In cases with full dataset, the test accuracies were comparable, however, the newly implemented H2O **FAKE GAME** model was significantly faster than any other from H2O with comparable test accuracy.

In addition, I evaluated a state-of-the-art hyper-parameter optimization method called SMAC and a commonly used one — random search. The results were in favor to SMAC, however, when compared with default settings, it pales in comparison. I think, this is caused by automatic tuning of parameters, which is enabled by default in H2O. And thus, using hyper-parameter optimization effectively discards any run-time information that could be used by automatic tuning. Although for **Deep Learning**, hyper-parameter optimization did not find any better configuration than default one, for tree based models, i.e., **Distributed Random Forest**, **Gradient Boosting Machines**, hyper-parameter optimization found better configurations, some of which didn't use more base learners than the default configuration.

I have submitted those results, i.e., evaluation of both anytime properties and hyper-parameter optimization, as a co-author, to the Machine Learning,

impacted journal of Springer Publishing [2].

Using more base learners, i.e., trees, was often not the best strategy. Often it seems that optimizing other hyper-parameters yields similarly good or even better test accuracies, which lead me to propose using a similar approach to TB-SPO [65], i.e., penalizing configurations by the time they took to evaluate. However, this is beyond the scope of this thesis.

Bibliography

[1] Dean, J.; Ghemawat, S. MapReduce: Simplied Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, 2004: pp. 137–149, ISSN 00010782, doi:10.1145/1327452.1327492, 10.1.1.163.5292.

[2] Kordík, P.; Černý, J.; et al. Discovering Predictive Ensembles for Transfer Learning and Meta-learning. *Machine Learning*, December 2016, submitted.

[3] Hutter, F.; Hoos, H. H.; et al. Sequential Model-Based Optimization for General Algorithm Configuration. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6683 LNCS, 2011: pp. 507–523, ISSN 03029743, doi:10.1007/978-3-642-25566-3_40.

[4] Bergstra, J.; Bardenet, R.; et al. Algorithms for Hyper-Parameter Optimization. *Advances in Neural Information Processing Systems (NIPS)*, 2011: pp. 2546–2554, ISSN 10495258, doi:2012arXiv1206.2944S, 1206.2944.

[5] Coope, I. D.; Price, C. J. On the convergence of grid-based methods for unconstrained optimization. *SIAM Journal on Optimization*, volume 11, no. 4, 2001: pp. 859–869.

[6] Kordík, P.; Černý, J. Self-organization of Supervised Models. In *Meta-Learning in Computational Intelligence*, *Studies in Computational Intelligence*, volume 358, edited by N. Jankowski; W. Duch; K. Gra¸bczewski, Springer Berlin, Heidelberg, 2011, ISBN 978-3-642-20979-6, pp. 179–223.

[7] Sutherland, A.; Henery, R.; et al. *StatLog: Comparison of Classification Algorithms on Large Real-World Problems.* Springer Berlin, 1993.

[8] Leite, R.; Brazdil, P. Active Testing Strategy to Predict the Best Classification Algorithm via Sampling and Metalearning. In *ECAI*, 2010, pp. 309–314.

[9] Leite, R.; Brazdil, P.; et al. Selecting classification algorithms with active testing. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, Springer, 2012, pp. 117–131.

[10] Botia, J. A.; Gomez-Skarmeta, A. F.; et al. METALA: A Meta-learning Architecture. In *Proceedings of the International Conference, Seventh Fuzzy Days on Computational Intelligence, Theory and Applications*, 2001.

[11] Lemke, C.; Gabrys, B. Meta-learning for time series forecasting and forecast combination. *Neurocomputing*, volume 73, no. 10, 2010: pp. 2006–2016.

[12] Bonissone, P. P. Lazy Meta-Learning: Creating Customized Model Ensembles on Demand. In *IEEE World Congress on Computational Intelligence*, Springer, 2012, pp. 1–23.

[13] Soares, C.; Brazdil, P. *A Comparative Study of Some Issues Concerning Algorithm Recommendation Using Ranking Methods*, volume 2527. Springer, 2002, pp. 80–89. Available from: `http://10.255.0.115/pub/2002/SB02a`

[14] Thornton, C.; Hutter, F.; et al. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013: pp. 847–855, doi: 10.1145/2487575.2487629, `arXiv:1208.3719v2`. Available from: `http://dl.acm.org/citation.cfm?id=2487629`

[15] Kordík, P. *Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution.* Dissertation thesis, Czech Technical University in Prague, FEE, Dep. of Comp. Sci. and Computers, FEE, CTU Prague, Czech Republic, September 2006.

[16] Duffy, N.; Helmbold, D. A geometric approach to leveraging weak learners. In *European Conference on Computational Learning Theory*, Springer, 1999, pp. 18–33.

[17] Marquardt, D. W. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics*, volume 11, no. 2, 1963: pp. 431–441.

[18] Kordik, P. *Hybrid Self-Organizing Modeling Systems*, *Studies in Computational Intelligence*, volume 211, chapter GAME - Hybrid Self-Organizing Modeling System based on GMDH. Springer-Verlag, Berlin, Heidelberg, 2009, p. 290 p.

[19] Bičík, V. *Continuous optimization algorithms*. Master's thesis, CTU in Prague, 2010.

[20] Kordík, P.; Koutník, J.; et al. 2010 Special Issue: Meta-learning approach to neural network optimization. *Neural Netw.*, volume 23, no. 4, May 2010: pp. 568–582, ISSN 0893-6080, doi:10.1016/j.neunet.2010.02.003. Available from: `http://dx.doi.org/10.1016/j.neunet.2010.02.003`

[21] The FAKE GAME environment for the automatic knowledge extraction. available online at: http://www.sourceforge.net/projects/fakegame, February 2011.

[22] Software. Rapid Miner, data mining (http://rapid-i.com/). Available from: `http://rapid-i.com/`

[23] Brazdil, P.; Giraud-Carrier, C.; et al. *Metalearning: Applications to Data Mining*. Cognitive Technologies, Springer, January 2009. Available from: `http://www.liaad.up.pt/pub/2009/BGSV09`

[24] Kuncheva, L. *Combining Pattern Classifiers: Methods and Algorithms*. John Wiley and Sons, New York, 2004.

[25] Wolpert, D. H. Stacked Generalization. *Neural Networks*, volume 5, 1992: pp. 241–259.

[26] Schapire, R. E. The Strength of Weak Learnability. *Mach. Learn.*, volume 5, no. 2, 1990: pp. 197–227, ISSN 0885-6125, doi:http://dx.doi.org/10.1023/A:1022648800760.

[27] Woods, K.; Kegelmeyer, W.; et al. Combination of Multiple Classifiers Using Local Accuracy Estimates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 19, 1997: pp. 405–410.

[28] Holeňa, M.; Linke, D.; et al. Boosted Neural Networks in Evolutionary Computation. In *Neural Information Processing. Lecture Notes in Computer Science 5864*, Springer Verlag, Berlin, 2009, pp. 131–140.

[29] Brown, G.; Wyatt, J.; et al. Managing Diversity in Regression Ensembles. *Journal of Machine Learning Research*, volume 6, 2006: pp. 1621–1650.

[30] Brazdil, P.; Giraud-Carrier, C.; et al. *Metalearning, Applications to Data Mining*. Cognitive Technologies, Springer Berlin Heidelberg, 2009, ISBN 978-3-540-73262-4.

[31] Breiman, L. Bagging predictors. *Mach. Learn.*, volume 24, no. 2, 1996: pp. 123–140, ISSN 0885-6125, doi:http://dx.doi.org/10.1023/A:1018054314350.

[32] Gama, J.; Brazdil, P. Cascade Generalization. *Mach. Learn.*, volume 41, no. 3, 2000: pp. 315–343, ISSN 0885-6125, doi:http://dx.doi.org/10.1023/A:1007652114878.

[33] Ferri, C.; Flach, P.; et al. Delegating classifiers. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, New York, NY, USA: ACM, 2004, ISBN 1-58113-828-5, p. 37, doi:http://doi.acm.org/10.1145/1015330.1015395.

[34] Alpaydin, E.; Kaynak, C. Cascading Classifiers. *Kybernetika*, volume 34, 1998: pp. 369–374.

[35] Kaynak, C.; Alpaydin, E. MultiStage Cascading of Multiple Classifiers: One Man's Noise is Another Man's Data. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, ISBN 1-55860-707-2, pp. 455–462.

[36] Ortega, J.; Koppel, M.; et al. Arbitrating among competing classifiers using learned referees. *Knowl. Inf. Syst.*, volume 3, no. 4, 2001: pp. 470–490, ISSN 0219-1377, doi:http://dx.doi.org/10.1007/PL00011679.

[37] Holland, J. H. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[38] Rosca, J. P. Analysis of complexity drift in genetic programming. *Genetic Programming*, 1997: pp. 286–294.

[39] Koza, J. R. Genetic Programming. *IEEE Intelligent Systems*, volume 14, no. 4, 2000: pp. 135–84. Available from: `http://www.ncbi.nlm.nih.gov/pubmed/20934511`

[40] Fielding, R. T. *Architectural styles and the design of network-based software architectures.* Dissertation thesis, University of California, Irvine, 2000.

[41] Arora, A.; Candel, A.; et al. Deep Learning with H2O. 2015.

[42] Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, volume 4, no. 2, 1991: pp. 251 – 257, ISSN 0893-6080, doi:http://dx.doi.org/10.1016/0893-6080(91)90009-T. Available from: `http://www.sciencedirect.com/science/article/pii/089360809190009T`

[43] Glorot, X.; Bordes, A.; et al. Deep sparse rectifier neural networks. *AISTATS '11: Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, volume 15, 2011: pp. 315–323, ISSN 15324435, doi:10.1.1.208.6449, 1502.03167.

[44] Recht, B.; Re, C.; et al. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, 2011, pp. 693–701.

[45] Zeiler, M. D. ADADELTA: An Adaptive Learning Rate Method. *arXiv*, 2012: p. 6, 1212.5701. Available from: http://arxiv.org/abs/1212.5701

[46] Hastie, T.; Tibshirani, R.; et al. The Elements of Statistical Learning. *Elements*, volume 1, 2009: pp. 337–387, ISSN 03436993, doi:10.1007/b94608, 1010.3003. Available from: http://www.springerlink.com/index/10.1007/b94608

[47] H2O.ai. H2O documentation. Available from: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/drf.html

[48] Breiman, L. Random Forests. *Machine Learning*, volume 45, no. 1, 2001: pp. 5–32, ISSN 1573-0565, doi:10.1023/A:1010933404324. Available from: http://dx.doi.org/10.1023/A:1010933404324

[49] Geurts, P.; Ernst, D.; et al. Extremely randomized trees. *Machine Learning*, volume 63, no. 1, 2006: pp. 3–42, ISSN 08856125, doi:10.1007/s10994-006-6226-1.

[50] Nykodym, T.; Kraljevic, T.; et al. Generalized Linear Modeling with H2O's R. 2016. Available from: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/booklets/GLMBooklet.pdf

[51] Friedman, J.; Hastie, T.; et al. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software*, volume 33, no. 1, 2010.

[52] Tibshirani, R.; Bien, J.; et al. Strong rules for discarding predictors in lasso-type problems. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, volume 74, no. 2, 2012: pp. 245–266.

[53] Akaike, H. A New Look at the Statistical Model Identification. *Automatic Control, IEEE Transactions on*, volume 19, no. 6, 1974: pp. 716–723, ISSN 15582523, doi:10.1109/TAC.1974.1100705, arXiv:1011.1669v3.

[54] McCullagh, P.; Nelder, J. A. Generalized Linear Models, Second Edition. 1989, doi:10.1007/978-1-4899-3242-6, arXiv:1011.1669v3.

[55] Click, C.; Malohlava, M.; et al. Gradient Boosting Machine with H2O. 2016.

[56] H2O.ai. H2O documentation. Available from: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/naive-bayes.html

[57] LeDell, E. Scalable Ensemble Learning and Computationally Efficient Variance Estimation. *Thesis*, volume 53, no. 9, 2013: pp. 1689–1699, ISSN 1098-6596, doi:10.1017/CBO9781107415324.004, arXiv:1011.1669v3.

[58] Bergstra, J.; Bengio, Y. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, volume 13, 2012: pp. 281–305, ISSN 1532-4435.

[59] Rasmussen, C. E.; Williams, C. K. I.; et al. Gaussian Processes for Machine Learning. *the MIT Press*, 2006.

[60] Ebden, M. Gaussian processes for regression: A quick introduction. *The Website of Robotics Research Group in Department on Engineering Science, University of Oxford*, 2008.

[61] Snoek, J.; Larochelle, H.; et al. Practical Bayesian Optimization of Machine Learning Algorithms. *Advances in Neural Information Processing Systems*, 2012: pp. 2951–2959, ISSN 10495258, doi:2012arXiv1206.2944S, 1206.2944. Available from: https://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf

[62] Xu, L.; Hutter, F.; et al. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, volume 32, 2008: pp. 565–606, ISSN 10769757, doi:http://dx.doi.org/10.1613/jair.2490.

[63] Hutter, F.; Hoos, H. H.; et al. An Experimental Investigation of Model-Based Parameter Optimisation: SPO and Beyond. *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, 2009: pp. 271–278, doi:10.1145/1569901.1569940. Available from: http://portal.acm.org/citation.cfm?id=1569901.1569940

[64] Bartz-Beielstein, T.; Lasarczyk, C.; et al. Sequential Parameter Optimization. *2005 IEEE Congress on Evolutionary Computation*, volume 1, 2005: pp. 773–780, doi:10.1109/CEC.2005.1554761. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1554761

[65] Hutter, F.; Hoos, H. H.; et al. Time-bounded sequential parameter optimization. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinform-*

atics), volume 6073 LNCS, 2010: pp. 281–298, ISSN 03029743, doi: 10.1007/978-3-642-13800-3_30.

[66] Li, L.; Jamieson, K.; et al. Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits. *arXiv preprint*, 2016, 1603.06560. Available from: http://arxiv.org/abs/1603.06560

[67] Jamieson, K.; Talwalkar, A. Non-stochastic Best Arm Identification and Hyperparameter Optimization. *Proceedings of AISTATS*, 2015, 1502.07943. Available from: http://arxiv.org/abs/1502.07943

[68] Karnin, Z.; Koren, T.; et al. Almost optimal exploration in multi-armed bandits. *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, 2013: pp. 1238–1246. Available from: http://jmlr.org/proceedings/papers/v28/karnin13.pdf

[69] Buja, A.; Stuetzle, W. Observations on bagging. *Statistica Sinica*, volume 16, 2006: pp. 323–351, ISSN 10170405.

[70] Pedregosa, F.; Varoquaux, G.; et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, volume 12, 2011: pp. 2825–2830.

[71] Buitinck, L.; Louppe, G.; et al. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[72] Van der Laan, M. J.; Polley, E. C.; et al. Super learner. *Statistical applications in genetics and molecular biology*, volume 6, no. 1, 2007.

[73] Baldi, P.; Sadowski, P.; et al. Searching for Exotic Particles in High-energy Physics with Deep Learning. *Nature Communications 5*, 2014.

[74] H2O.ai. *H2O: Scalable Machine Learning*. 2015. Available from: http://www.h2o.ai

[75] Hussami, N.; Kraljevic, T.; et al. Generalized Linear Modeling with H2O. 2015.

[76] LeDell, E. Scalable Super Learning. *Handbook of Big Data*, 2016: p. 339.

Appendix A

Acronyms

AIC Akaike information criterion.

cdf cummulative distribution function.

DL Deep Learning.

DRF Distributed Random Forest.

EI Expected Improvement.

FAKE GAME Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution.

GBM Gradient Boosting Machines.

GLM Generalized Linear Model.

GP Gaussian process.

IRLSM iteratively reweighted least squares.

L-BFGS limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm.

lasso least absolute shrinkage and selection operator.

MLE maximum likelihood estimate.

MLP multi-layer perceptron.

MSE mean squared error.

NB Naïve Bayes.

OLS ordinary least squares.

pdf probability density function.

POI Probability of Improvement.

RESTful API representational state transfer application programming interface.

RF Random Forest.

ROAR Randomized Online Aggresive Racing.

SAT Boolean Satisfiability Problem.

SL Super Learner.

SMAC Sequential Model-based Algorithm Configuration.

SMBO Sequential Model-based Bayesian Optimization.

TPE Tree-structured Parzen Estimator.

UBC Upper Bound Confidence.

Appendix B

Parameter Ranges for Hyper-parameter Optimization

Table B.1: Deep Learning

Name	Type	Possible Values	Default Value
hidden_l2	integer	[10, 1000]	200
rate_annealing	real	[1e-15, 0.2]	1e-06
l1	real	[0, 1]	0
quantile_alpha	real	[0, 1]	0.5
balance_classes	categorical	[True, False]	False
momentum_ramp	real	[100, 100000000]	1000000
loss	categorical	['Automatic', 'CrossEntropy', 'Quadratic', 'Huber', 'Absolute']	Automatic
input_dropout_ratio	real	[0, 0.5]	0
momentum_start	real	[0, 1.0]	0.0
max_after_balance_size	real	[0.1, 10]	5.0
epochs	real	[2, 100]	10
initial_weight_distribution	categorical	['UniformAdaptive', 'Uniform', 'Normal']	UniformAdaptive
rate	real	[1e-08, 0.1]	0.005
rate_decay	real	[0.8, 1.2]	1
nesterov_accelerated_gradient	categorical	[True, False]	True
sparse	categorical	[True, False]	False
hidden_l1	integer	[10, 1000]	200
stopping_rounds	integer	[0, 10]	5
adaptive_rate	categorical	[True]	True
stopping_metric	categorical	['AUTO', 'logloss', 'MSE', 'lift_top_group', 'r2', 'misclassification', 'mean_per_class_error']	AUTO
epsilon	real	[1e-15, 0.2]	1e-08
overwrite_with_best_model	categorical	[True, False]	True
stopping_tolerance	real	[0, 0.5]	0
activation	categorical	['Tanh', 'TanhWithDropout', 'Rectifier', 'RectifierWithDropout', 'Maxout', 'MaxoutWithDropout']	Rectifier

Table B.1 – continued from previous page

Name	Type	Possible Values	Default Value
elastic_averaging_moving_rate	real	[0.5, 1]	0.9
standardize	categorical	[True, False]	True
elastic_averaging_regularization	real	[1e-08, 0.1]	0.001
rho	real	[0.5, 1.0]	0.99
shuffle_training_data	categorical	[True, False]	False
use_all_factor_levels	categorical	[True, False]	True
tweedie_power	real	[1, 1.9999999999999]	1.5
mini_batch_size	integer	[1, 100]	1
l2	real	[0, 1]	0
momentum_stable	real	[0, 1]	0.0
initial_weight_scale	real	[0.1, 10]	1.0
fast_mode	categorical	[True, False]	True

Table B.2: Distributed Random Forest

Name	Type	Possible Values	Default Value
nbins_cats	integer	[4, 2048]	1024
nbins	integer	[2, 100]	20
stopping_rounds	integer	[0, 10]	0
balance_classes	categorical	[True, False]	False
nbins_top_level	integer	[200, 2048]	1024
max_depth	integer	[2, 40]	20
histogram_type	categorical	['AUTO', 'UniformAdaptive', 'Random', 'QuantilesGlobal', 'RoundRobin']	AUTO
min_rows	real	[1, 100]	1
max_after_balance_size	real	[0.1, 10]	5
r2_stopping	real	[0.95, 0.9999999999]	0.999999
col_sample_rate_change_per_level	real	[0, 2]	1
binomial_double_trees	categorical	[True, False]	False
sample_rate	real	[0, 1.0]	0.632000029087
stopping_tolerance	real	[1e-09, 0.1]	0.001
ntrees	integer	[10, 200]	50
col_sample_rate_per_tree	real	[0, 1]	1.0
stopping_metric	categorical	['AUTO', 'logloss', 'MSE', 'AUC', 'lift_top_group', 'r2', 'misclassification', 'mean_per_class_error']	AUTO
min_split_improvement	real	[1e-08, 0.1]	1e-05

Table B.3: Gradient Boosting Machine

Name	Type	Possible Values	Default Value
nbins	integer	[2, 200]	20
stopping_rounds	integer	[0, 10]	0
quantile_alpha	real	[0, 1]	0.5
balance_classes	categorical	[True, False]	False
histogram_type	categorical	['AUTO', 'UniformAdaptive', 'Random', 'QuantilesGlobal', 'RoundRobin']	AUTO
max_after_balance_size	real	[0.1, 10]	5
stopping_tolerance	real	[1e-09, 0.1]	0.001
ntrees	integer	[10, 500]	50
learn_rate_annealing	real	[0.95, 1.0]	1
stopping_metric	categorical	['AUTO', 'logloss', 'MSE', 'AUC', 'lift_top_group', 'r2', 'misclassification', 'mean_per_class_error']	AUTO
col_sample_rate	real	[1e-10, 1]	1
nbins_cats	integer	[4, 8192]	1024
nbins_top_level	integer	[200, 8192]	1024
max_depth	integer	[1, 50]	5
min_rows	real	[1, 100]	10
col_sample_rate_per_tree	real	[1e-10, 1]	1.0
r2_stopping	real	[0.95, 0.9999999999]	0.999999
col_sample_rate_change_per_level	real	[0, 2]	1
sample_rate	real	[1e-10, 1]	1
tweedie_power	real	[1, 1.999999999]	1.5
learn_rate	real	[1e-10, 1]	0.1
min_split_improvement	real	[1e-12, 0.1]	1e-05

Table B.4: Generalized Linear Model

Name	Type	Possible Values	Default Value
family	categorical	['binomial']	binomial
prior	real	[1e-10, 0.9999999999999]	-1
alpha	real	[0, 1]	0.5
lambda_search	categorical	[True, False]	False
lambda_min_ratio	real	[-1, 1]	-1
nlambdas	integer	[1, 10]	-1
beta_epsilon	real	[1e-10, 0.1]	0.0001
standardize	categorical	[True, False]	True
tweedie_link_power	real	[0, 5]	1
max_after_balance_size	real	[0.1, 10]	5
lambda_	real	[0, 10]	0
max_active_predictors	integer	[1, 100]	-1
non_negative	categorical	[True, False]	False
early_stopping	categorical	[True, False]	True

Table B.4 – continued from previous page

Name	Type	Possible Values	Default Value
solver	categorical	['AUTO', 'IRLSM', 'COORDINATE_DESCENT_NAIVE', 'COORDINATE_DESCENT']	AUTO
tweedie_variance_power	real	[0, 5]	0
balance_classes	categorical	[True, False]	False
max_iterations	integer	[1, 1000]	-1

Table B.5: ClassifierBagging{$m\times$ ClassifierBoosting{$n\times$ DecisionTree}}

Name	Type	Possible Values	Default Value
m	integer	[1, 80]	5
n	integer	[1, 80]	5

Table B.6: CascadeGenProb{$m\times$ Boosting{$n\times$ ClassifierModel{<outputs> $\times$ SigmoidNorm}}}

Name	Type	Possible Values	Default Value
m	integer	[1, 80]	5
n	integer	[1, 80]	5

Appendix C

Contents of enclosed DVD

```
/
├── data ........... Shuffled subsamples of the datasets used in the thesis
├── experiments ..................... Experiment files used in the thesis
├── readme.txt .................. The file with DVD contents description
├── src ................................. The directory of source codes
│   ├── h2o-benchmarker ......................... Benchmarker sources
│   ├── h2o-fakegame ..................... H2O with integrated fakegame
│   └── fakegame ....................... Modified FAKE GAME sources
├── text ............... The directory of LaTeX source codes of the thesis
├── thesis.pdf ......................... The thesis text in PDF format
└── thesis.ps ........................... The thesis text in PS format
```