Classification in Big Data Environments

Clemens Damke

Intelligent Systems and Machine Learning Group (ISG)
Heinz Nixdorf Institute
Paderborn University
Warburger Straße 100
33098 Paderborn

Abstract. The training of classifiers on big datasets is an important problem in many domains. This paper gives an overview of optimization methods that aim to speed up the training process. We describe two approaches to speed up hyperparameter optimization by learning probabilistic models of error functions. We then present methods to adapt bootstrapping, gradient descent, logistic regression and SVMs to big datasets.

Keywords: Big Data · Automated Machine Learning

1 Introduction

Over the last few years Big Data processing has become increasingly important in many domains. This increase in the data volume also poses new challenges for machine learning applications. The training time of learners is usually polynomially dependent on the size of the training dataset $\mathcal{D}_{train}$, i. e. $\mathcal{O}(|\mathcal{D}_{train}|^{\alpha}), \alpha \geq 1$. Since training has to be repeated for every iteration of cross-validation and hyperparameter search, always using the entire dataset quickly becomes infeasible. This paper gives an overview of approaches to tackle this problem with a focus on the domain of classification problems.

The process of finding a hypothesis can in general be split into three phases:

1. **Model selection:** At first a model has to be selected that determines the class of hypothesis spaces out of which the final hypothesis will be selected. The choice of model is often implicitly encoded in a learner L. Automating this step is non-trivial. In practice the model is typically selected by experts with domain specific knowledge about the problem at hand.
2. **Hyperparameter search:** Optimizing a vector λ in the hyperparameter space Λ_L of the learner L representing a hypothesis space $\mathcal{H}_\lambda$. A naïve approach to do this is to systematically try configurations using a grid search or a random search over Λ_L. To evaluate the quality of a given λ, L is usually trained on a training dataset $\mathcal{D}_{train}$ using λ. This yields a hypothesis $\hat{h}_\lambda \in \mathcal{H}_\lambda$ that is evaluated using a validation dataset $\mathcal{D}_{valid}$. The goal of hyperparameter optimization is to minimize the loss $l(\lambda)$ of $\hat{h}_\lambda$ on $\mathcal{D}_{valid}$, i. e. to find an approximation $\hat{\lambda}$ of $\lambda^* := \arg\min_\lambda l(\lambda)$.
3. **Training or parameter search:** Let w be a vector in the parameter space $W_{\mathcal{H}_\lambda}$, describing a hypothesis $h_{\lambda,w} \in \mathcal{H}_\lambda$ given a hyperparameter configuration λ. The

goal of parameter search is to find an approximation $\hat{h}_\lambda$ of the hypothesis $h^*_\lambda := \arg\min_{h_{\lambda,w}} \ell(\mathcal{D}_{train}|h_{\lambda,w})$, with $\ell(\mathcal{D}_{train}|h_{\lambda,w})$ being the empirical loss of $h_{\lambda,w}$ on a given training dataset $\mathcal{D}_{train}$ according to some loss function ℓ. Depending on the learner L, various kinds of optimization methods are used to find this minimum, e. g. Bayesian optimization, quadratic programming or, if $\nabla_w e(\mathcal{D}_{train}|h_{\lambda,w})$ is computable, gradient descent. The quality l of $\hat{h}_\lambda$ is measured by the loss on a validation or test dataset, i. e. $l(\lambda) := \ell(\mathcal{D}_{valid}|\hat{h}_\lambda)$.

This paper is structured according to the last two phases, i. e. we will assume that the learner L is given. Section 2 describes ways to speed up the hyperparameter search. Section 3 then describes how to improve the training methods of existing learners. Most of the techniques described in this paper improve upon independent components of the hypothesis finding process allowing them to be combined.

2 Hyperparameter optimization

As described in the introduction, the goal of hyperparameter optimization is to find a global minimum of l. Since l is generally unknown, analytical methods or gradient descent cannot usually be applied. The only way the get information about l is to evaluate it, which is costly. There are multiple ways to reduce the total cost of those evaluations:

1. **Number T of evaluations of l:** During optimization multiple hyperparameter configurations $\lambda_1, \dots, \lambda_T$ will be evaluated using l. T is usually fixed when using a grid search or a random search. After evaluating T configurations, the best one is chosen. Those naïve approaches assume that $l(\lambda)$ is independent of $l(\lambda')$ for all pairs $\lambda \neq \lambda'$. We will see that this strong assumption of independence is not necessarily true which in turn allows reducing T.

2. **Training dataset size S:** The performance of a given configuration $l(\lambda)$ is computed by training the learner on $\mathcal{D}_{train}$ which is expensive for big datasets. By training on S instead of $|\mathcal{D}_{train}|$ datapoints the evaluation can be sped up.

3. **Number of training iterations E:** Depending on the learner, training often is an iterative process, e. g. gradient descent. To speed up hyperparameter optimization training could be terminated before convergence.

2.1 FABOLAS

The first approach we will discuss is called Fabolas (Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets) [6]. It can be applied to any learner L and is based upon two main ideas:

1. The validation loss l is modeled as a *Gaussian process* (GP) f based on the assumption that two configurations λ and λ' will perform similar if they are similar according to some kernel $k(\lambda, \lambda')$. The Gaussian process f is used as a surrogate to estimate the expected value and variance of l given λ. Using *Bayesian optimization* l will be probed at promising positions to iteratively improve f. Hyperparameter

configurations that are expected to perform worse than the current optimum will not be probed. This effectively reduces T.

2. The training dataset size S is modeled as an additional hyperparameter of f giving the optimizer an additional degree of freedom. This allows extrapolating the value of l when trained on the complete dataset while only probing smaller subsets which effectively reduces S.

We will now describe how those two ideas can be applied.

Gaussian processes A Gaussian process is a family of *random variables* (RVs) $(X_\theta)_{\theta \in \Theta}$, s. t. every finite subset of them follows a multivariate normal distribution. More intuitively it can be understood as a probability distribution over functions $f : \Theta \to \mathbb{R}$ where $X_\theta \mathrel{\widehat{=}} f(\theta)$. Prior knowledge about the likelihood of each f is described by a prior mean function $\mu_0(\theta) = \mathbb{E}[f(\theta)]$ and a positive-definite kernel $k(\theta, \theta') = \mathrm{Cov}(f(\theta), f(\theta'))$. The covariance kernel models how informative it is to know $f(\theta)$ to determine $f(\theta')$.

Let $\mathcal{D}_n = \{(\boldsymbol{\theta}_i, \boldsymbol{y}_i)\}_{i=1}^n$ denote a set of observations. Those observations can be used to update the means and variances of the RVs via GP regression. This collapses the space of possible functions f to those functions that align with $\mathcal{D}_n$ (see fig. 2):

$$
\begin{aligned}
\boldsymbol{m} &:= (\mu_0(\boldsymbol{\theta}_1), \ldots, \mu_0(\boldsymbol{\theta}_n))^T \\
\boldsymbol{k}(\theta) &:= (k(\boldsymbol{\theta}_1, \theta), \ldots, k(\boldsymbol{\theta}_n, \theta))^T \\
\boldsymbol{K} &\in \mathbb{R}^{n \times n}, \boldsymbol{K}_{ij} := k(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) \\
\mathbb{E}[f(\theta) \,|\, \mathcal{D}_n] &:= \mu_n(\theta) = m_0(\theta) + \boldsymbol{k}(\theta)^T \boldsymbol{K}^{-1}(\boldsymbol{y} - \boldsymbol{m}) \qquad (1) \\
\mathrm{Cov}(f(\theta), f(\theta') \,|\, \mathcal{D}_n) &:= k(\theta, \theta') - \boldsymbol{k}(\theta)^T \boldsymbol{K}^{-1} \boldsymbol{k}(\theta') \qquad (2)
\end{aligned}
$$

Fabolas works by modeling the loss function l as a Gaussian process $f \sim \mathcal{GP}(m, k)$ with parameter set $\Theta := \Lambda \times [0, 1]$ where $\mu_0(\lambda, s) = \mathbb{E}[f(\lambda, s)] = \mathbb{E}[l(\lambda) \,|\, \text{training size } s]$. To model the covariances between different combinations of hyperparameters and training set sizes, the following product kernel is used:

$$
k((\lambda, s), (\lambda', s')) := k_{\text{MATÉRN5}}(d_M(\lambda, \lambda')) \cdot k_{\text{lin}}(s, s') \qquad (3)
$$

Here $k_{\text{MATÉRN5}}$ denotes the stationary Matérn kernel ($\nu = 5/2$) with d_M being the Mahalanobis distance between the two compared hyperparameter configurations. k_{lin} essentially is a simple linear kernel modeling the assumption that l monotonically decreases when s is increased. We will only give an intuition for this choice of kernel and refer to Klein et al. [6] for the details.

The Mahalanobis distance d_M is used instead of the Euclidean distance because the hyperparameters in a configuration typically use very different scales and are in some cases also correlated. Figure 1 gives an intuition for this.

Based on the Mahalanobis distance between two configurations λ, λ' the MATÉRN5 kernel is used to compute a covariance. The class of Matérn kernels interpolates between the Gaussian (SQ-EXP) and the exponential (MATÉRN1) kernel (see fig. 2). Because the exponential kernel drops off quickly, configurations quickly become uncorrelated which causes noisy samples. The Gaussian kernel drops off less quickly

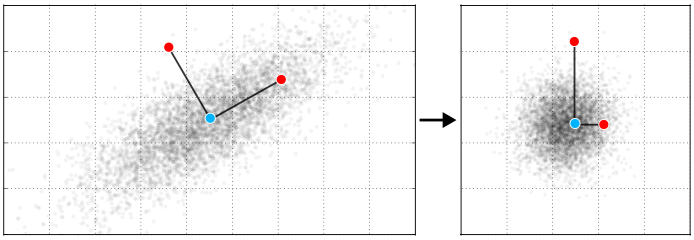


Fig. 1: Intuition for the Mahalanobis distance. Using the Euclidean distance the red points would be equally far away from the blue one. The Mahalanobis distance fixes this by first normalizing the hyperparameters and removing correlations.

causing smoother samples. Fabolas uses Matérn5 as it empirically fits the smoothness of typical loss functions l quite well.

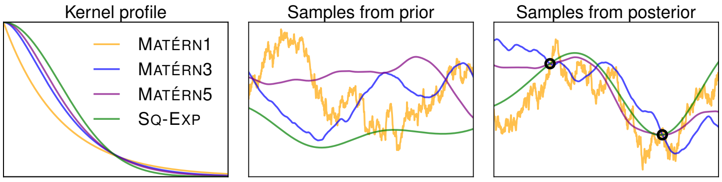


Fig. 2: Comparison between different covariance kernels. The middle shows randomly sampled functions f using different kernels. The right shows random samples after two f values were observed and incorporated into the model via GP regression.

Bayesian optimization To find $\arg\min_\lambda l(\lambda)$ the bias and variance of f has to be reduced by probing l at promising positions. This is called Bayesian optimization. The estimated minimum after n probes is described by $\arg\min_\lambda \mu_n(\lambda, s = 1)$, i. e. the configuration with the smallest predicted error on the full test dataset. To reduce the number of probes required until this minimum converges, an *acquisition function* is used. Its role is to trade-off exploration vs. exploitation of l by describing the expected utility of probing (λ_{n+1}, s_{n+1}) given a set of previous probes $\mathcal{D}_n$. Fabolas uses an aquisition function that rates configurations by their *information gain* per computation time:

$$a_F(\lambda, s) := \frac{1}{c(\lambda, s)} \mathbb{E}_y \left[p(y \mid \lambda, s, \mathcal{D}_n) \ \cdot \mathrm{KL}_{\hat{\lambda}}(p_{\min}(\hat{\lambda} \mid \mathcal{D}_n \cup \{(\lambda, s, y)\}) \,||\, u(\hat{\lambda})) \right] \quad (4)$$

$$p_{\min}(\lambda \mid \mathcal{D}) := p(\lambda \in \arg\min_{\lambda'} f(\lambda', s = 1) \mid \mathcal{D})$$

It measures the expected amount of available information about the optimal configuration if a given configuration were probed, i. e. the Kullback-Leibler divergence between the density $p_{\min}$ of λ being optimal after a probe and the uniform density u. This information gain of a probe is compared to its expected associated computation time c. c is estimated using a separate Gaussian process that is maintained alongside f. Fabolas considers the cost of a probe because it tries to minimize the total optimization time not the total number of probes.

Since it is infeasible to compute a_F numerically, its maximum is estimated using *Markov-Chain Monte Carlo* (MCMC). The estimated most promising configuration will be probed. The resulting loss value and runtime are then used to update the loss model f and cost model c via GP regression.

Evaluation Fabolas was evaluated in *support vector machine* (SVM) and *convolutional neural network* (CNN) optimization tasks on the MNIST and CIFAR-10 dataset respectively. Figure 3 compares Fabolas to the following other hyperparameter optimization approaches:

- **Random Search:** Simple random hyperparameter search. Each configuration is evaluated on the full dataset.
- **Entropy Search & Expected Improvement:** Bayesian optimization methods that always evaluate on the full dataset. Expected Improvement uses an aquisition function that simply probes at the current expected optimum. Entropy Search uses an aquisition function similar to the one used by Fabolas but without the cost model.
- **MTBO-N (Multi-Task Bayesian Optimization):** Like Fabolas but restricts probes to two sizes $s \in \{{}^1\!/\!{}_N, 1\}$, i. e. either a small subsample or the entire dataset is used. Multiple values for N were evaluated: 4, 32 and 512.

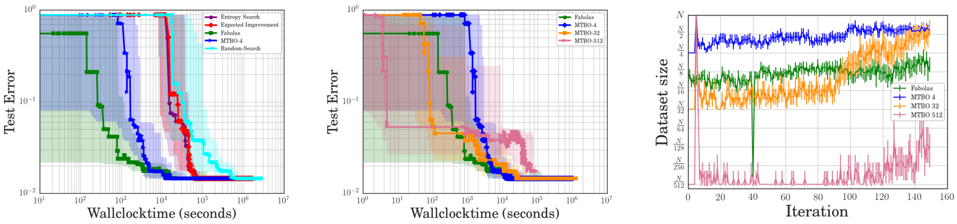


Fig. 3: SVM optimization on the MNIST dataset. (Left) Comparison of the test performance over time of different optimzers. (Middle) Comparison of Fabolas with different MTBO subsample sizes. (Right) Comparison of the subsample sizes s that MTBO and Fabolas choose for their probes. The average over 10 runs is depicted.

All Bayesian optimization approaches are at least one order of magnitude faster than random search. By allowing two probing sizes, MTBO is one additional order of magnitude faster. Depending on the choice of N MTBO sometimes improves faster than Fabolas initially. Once Fabolas starts improving it does however find a good configuration about one order of magnitude faster than MTBO. The optimal configuration is found at roughly the same time by both Fabolas and MTBO. Overall Fabolas finds a good configuration between 100 and 1000 times faster than random search does. Similar results are obtained when optimizing CNNs on CIFAR-10.

2.2 Learning Curve Extrapolation

The second approach for speeding up hyperparameter optimization focuses on reducing the number of training iterations E. It can in principle be applied to any iterative learner and can be integrated into any hyperparameter optimizer. The idea is to monitor the learning curve of a learner during training with a hyperparameter configuration λ. If it is unlikely that a good accuracy will be reached with λ, training will be terminated before convergence.

The method was first described by Domhan et al. [4] in the context of hyperparameter optimization for *deep neural networks* (DNNs) that are trained using *stochastic gradient descent* (SGD).

Extrapolation Method Let $y_{1:n}$ denote the observed learning curve of SGD after n iterations, i e. the sequence of training accuracies $y_i \in [0, 1]$. Normally SGD iterations would be run for each hyperparameter configuration λ until convergence or until a maximum number of iterations E has been reached. The learning curve extrapolation optimization works by predicting y_E every p iterations:

```
 1: $\hat{y} \leftarrow -\infty$
 2: **for** $\lambda \leftarrow$ next hyperparameter configuration to evaluate **do**
 3:     $n \leftarrow 0$
 4:     **repeat**
 5:         Run $p$ SGD iterations using $\lambda$ with resulting accuracies $y_{(n+1):(n+p)}$.
 6:         $n \leftarrow n + p$
 7:         Estimate $P(y_E < \hat{y} \,|\, y_{1:n})$.
 8:     **until** SGD converged $\vee\ n \geq E \vee P(y_E < \hat{y} \,|\, y_{1:n}) > \delta$
 9:     **if** $y_n > \hat{y}$ **then** $\hat{y} \leftarrow y_n$ **end if**
10: **end for**
```

The prediction step (line 7) uses a probabilistic model. Similar to Fabolas, a distribution over candidate functions is fit to the observations $y_{1:n}$. Unlike Fabolas however, which uses a flexible non-parametric GP model, we use prior knowledge about the shape of learning curves to restrict the model to parameterized, increasing, saturating functions. More specifically, the learning curve $y_{1:n}$ is modeled as a linear combination f_{comb} of a family of given functions.

$$f_{comb}(t \,|\, \xi) := \sum_{k=1}^{K} w_k f_k(t \,|\, \theta_k),\ \xi = (w_1, \ldots, w_k, \theta_1, \ldots, \theta_k, \sigma^2) \tag{5}$$

$$y_t \sim \mathcal{N}(f_{comb}(t \,|\, \xi), \sigma^2) \tag{6}$$

Domhan et al. [4] use $K = 11$ types of functions $\{f_1, \ldots, f_K\}$ that are each parameterized by $\{\theta_1, \ldots, \theta_K\}$. The assumption is that every function type captures certain aspects of learning curves. By allowing linear combinations a more powerful model can be obtained. Figure 4 illustrates this idea. To estimate the probability $P(y_E < \hat{y} \,|\, y_{1:n})$

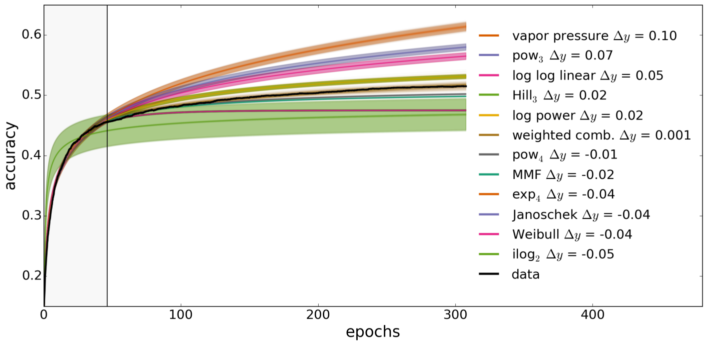


Fig. 4: Comparison of an observed learning curve (black) with the 11 types of learning curve models and a linear combination of them. Each type is parameterized to fit the first 50 observations $y_{1:50}$. As can be seen in the legend on the left, the linear combination has the smallest deviation $\varDelta y$ from the observed data after 300 iterations.

MCMC is used to sample S learning curves $\{\xi_1, \ldots, \xi_S\}$ from the posterior

$$P(\xi \mid y_{1:n}) \propto P(y_{1:n} \mid \xi) P(\xi) \tag{7}$$

$$P(y_{1:n} \mid \xi) = \prod_{t=1}^{n} \mathcal{N}(y_t; f_{comb}(t \mid \xi), \sigma^2) \tag{8}$$

$$P(\xi) \propto \mathbb{1}[f_{comb}(1 \mid \xi) < f_{comb}(E \mid \xi) \wedge \forall k : w_k > 0] \tag{9}$$

The prior on ξ is used to model the fact that learning curves do not typically decrease over time. Given the learning curve samples, we can now estimate

$$\begin{aligned} P(y_E < \hat{y} \mid y_{1:n}) &= \int P(\xi \mid y_{1:n}) P(y_E < \hat{y} \mid \xi) \, \mathrm{d}\xi \\ &\approx \frac{1}{S} \sum_{s=1}^{S} \varPhi(\hat{y}; f_{comb}(E \mid \xi_s), \sigma^2) \end{aligned} \tag{10}$$

Evaluation The early termination method we just described was evaluated on the CIFAR-10, CIFAR-100 and MNIST dataset. Figure 5 shows the behavior of early termination and the obtained speedup on CIFAR-10. As expected, configurations with learning curves that tend to approach low accuracies are terminated early. Configurations with high accuracies are evaluated until convergence. This approach consistently speeds up the hyperparameter optimization by a factor of two across the tested datasets while reaching the same quality.

3 Optimizing Training

Now follows an overview of approaches to speed up the training process. We will discuss four approaches:

1. A general purpose method that combines subsampling with bootstrapping.
2. An iterative method to select the optimal subsample size during gradient descent.

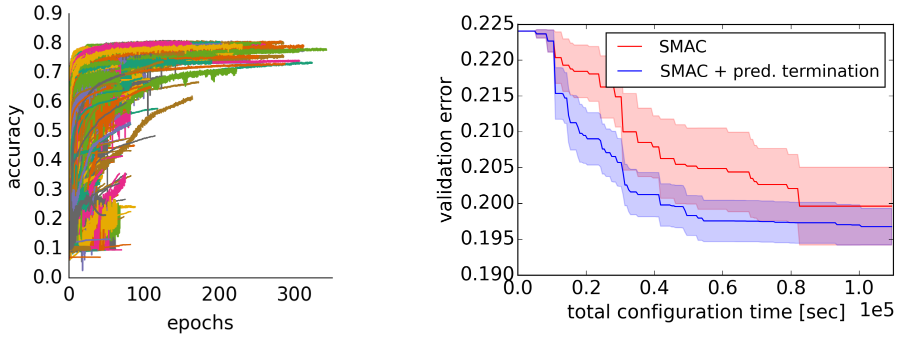


Fig. 5: Evaluation on the CIFAR-10 dataset. The left graph shows the learning curves of all hyperparameter configurations that were evaluated. The right graph shows the average validation error over time.

3. Improving the quality of subsampling for logistic regression by weighing the samples.
4. Speeding up the training of SVMs via k-means clustering.

3.1 Bag of Little Bootstraps

The first approach we will discuss is called *Bag of Little Bootstraps* (BLB) [7]. It combines subsampling with bootstrapping and is particularly well suited for parallelized implementations.

In the context of Big Data training typically cannot be performed on the entire dataset. A naïve way to solve this problem is to simply train on a random b out of n subsample of the data $\mathcal{D}_{train} = \{X_1, \ldots, X_n\}$. This approach is highly sensitive to noise in the training dataset, especially if $b \ll n$. To overcome this problem bootstrapping can be used. The regular n out of n bootstrapping technique for variance reduction is not suitable for big datasets because it uses 63% of the training data on average. However the *b out of n bootstrapping* (BOFN) approach can in principle be applied. It uses s samples $\{\check{X}^{(i)} = (\check{X}_1^{(i)}, \ldots, \check{X}_b^{(i)}) \,|\, 1 \leq i \leq s\}$ of b datapoints each. Since this approach independently learns s hypotheses h_i on small datasets $\check{X}^{(i)}$, their parameterizations θ_i tend to have large confidence intervals. Because of that, the quality of the combined hypothesis is strongly dependent on b. BLB reduces this dependence.

Intuition BLB is a simple extension of BOFN that is consistently more robust regarding the choice b across datasets. The basic idea is to add another sampling step. BLB uses each subsample $\check{X}^{(i)}$ as a seed for n out of b sampling. This yields bigger resamples $\{X^{*(i,k)} = (X_1^{*(i,k)}, \ldots, X_n^{*(i,k)}) \,|\, 1 \leq i \leq s, 1 \leq k \leq r\}$ that each contain at most b different elements. Training is then run on the resamples X^* instead of the small seed samples $\check{X}$. The learned hypothesis parameterizations are finally combined to a single hypothesis parameterization θ^* via a model specific combination function, e. g. by simply taking the average. Figure 6 illustrates those steps.

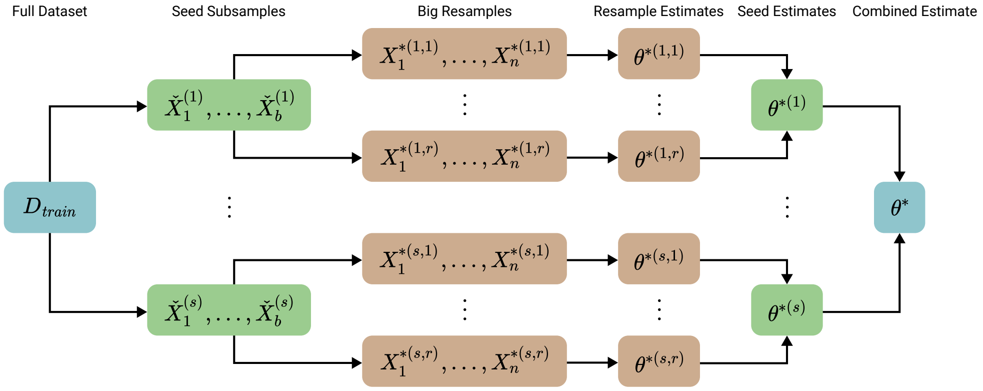

Full Dataset
Seed Subsamples
Big Resamples
Resample Estimates
Seed Estimates
Combined Estimate
D_{train}
$\check{X}_1^{(1)}, \dots, \check{X}_b^{(1)}$
$\check{X}_1^{(s)}, \dots, \check{X}_b^{(s)}$
$X_1^{*(1,1)}, \dots, X_n^{*(1,1)}$
$X_1^{*(1,r)}, \dots, X_n^{*(1,r)}$
$X_1^{*(s,1)}, \dots, X_n^{*(s,1)}$
$X_1^{*(s,r)}, \dots, X_n^{*(s,r)}$
$\theta^{*(1,1)}$
$\theta^{*(1,r)}$
$\theta^{*(s,1)}$
$\theta^{*(s,r)}$
$\theta^{*(1)}$
$\theta^{*(s)}$
θ^{*}

Fig. 6: Overview of the steps of BLB.

Even though BLB trains classifiers on resamples of size n its time and space complexity effectively still depends on b, not n. This is because each resample X^* only contains at most b different elements which means that it can be efficiently represented by a list of b multiplicity counts $(c_1, \dots, c_b) \in \mathbb{N}^b$, i. e. $\mathit{space} = \mathcal{O}(b \log n)$. Training on such a dataset is equivalent to training on a dataset of size b with weights $w_i = \frac{c_i}{n}$. Since most commonly used classifiers support weighted samples, BLB is widely applicable.

Evaluation To show the advantages of BLB for classification it was evaluated with logistic regression. Figure 7a shows that BLB converges on a solution much faster than the regular n out of n bootstrapping (BOOT) with comparable results. It also shows that BLB is less sensitive to the choice of b than BOFN. BLB reached good results with $b \geq n^{0.6}$ whereas BOFN required at least $b \geq n^{0.7}$.

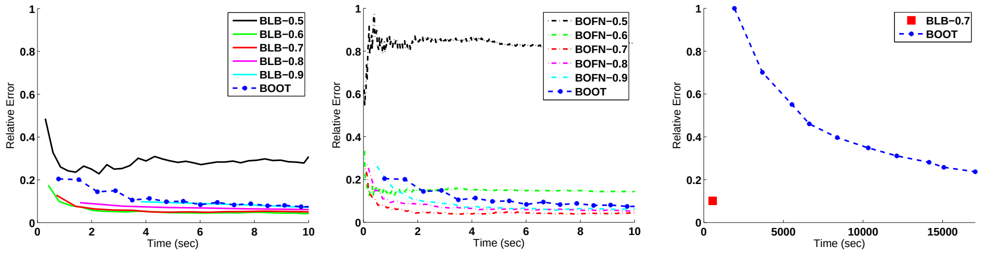
While BLB already outperforms BOOT without parallelism, as training is overproportionally faster on small samples, its scalability becomes more apparent when parallelized. Figure 7b shows that BLB significantly outperforms BOOT on a Spark cluster with 10 workers. This is due to the fact that each BLB sample can be held in memory by its corresponding worker node. The much larger BOOT samples, on the other hand, require disk reads for large datasets.

3.2 Subsample Size Selection for Gradient Descent

Next we will discuss an optimization technique for *stochastic gradient descent* (SGD). The size of the subsample $\mathcal{S}$ that is considered in a single gradient descent step heavily influences the optimizer's behavior:

- In the stochastic approximation regime small samples, typically $|\mathcal{S}| = 1$, are used. This causes fast but noisy steps.
- In the batch regime large samples are used, typically $|\mathcal{S}| = N$ with $N := |\mathcal{D}_{train}|$. Steps are expensive to compute but more reliable.

Both extremes are usually not suitable for Big Data applications. Very small samples cannot be parallelized well, making them a bad fit for the compute clusters that are



(a) Single-threaded results on a subset of the data. $b = n^\gamma$ for multiple values of $\gamma \in [0.5, 1]$ and $r = 100$ is used. s is not fixed and grows over time.

(b) Parallelized results on the entire dataset. $b = n^{0.7}$, $s = 5$ and $r = 50$ is used for BLB. For BOOT s grows over time.

Fig. 7: Comparison of BLB and BOFN with BOOT using logistic regression. Shows the error on the moving average of all sample hypotheses that are computed within a given time.

typically available nowadays. The gradients for very big samples however are often too slow to compute. $|\mathcal{S}|$ should ideally lie somewhere in between.

Size Selection Method Byrd et al. [3] describe an iterative algorithm that dynamically increases the size of $\mathcal{S}$ as long as this promises to significantly reduce the gradient noise. Let $\mathcal{S} \subseteq \{1, \dots, N\}$ describe a random subsample of $\mathcal{D}_{train} = \{(x_i, y_i) \,|\, 1 \leq i \leq N\}$. SGD will take a step in the descent direction $d = -\nabla J_\mathcal{S}(w)$ where $J_\mathcal{S}(w) := \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \ell(h_w(x_i), y_i)$ is the differentiable average loss on $\mathcal{S}$ given the current configuration w. Let $J(w)$ be the average loss on the entire dataset $\mathcal{D}_{train}$. $J(w)$ is the objective function we want to minimize. Our goal is to trade off $|\mathcal{S}|$ s. t. it is as small as possible and $\nabla J_\mathcal{S}(w)$ still *tends to agree* with the objective gradient $\nabla J(w)$, or more formally

$$\|\nabla J_\mathcal{S}(w) - \nabla J(w)\|_2 \leq \theta \|\nabla J_\mathcal{S}(w)\|_2, \theta \in [0, 1) \tag{11}$$

A value of $\theta = 0$ means that $\nabla J_\mathcal{S}(w)$ always has to be equal to $\nabla J(w)$, whereas $\theta = 1$ would allow steps that directy oppose $\nabla J(w)$. Since it is infeasible to compute $\nabla J(w)$, condition (11) can however not be checked directly. We will instead resort to an estimate and check whether the condition is satisfied in expectation:

$$\underbrace{\mathbb{E}_\mathcal{S}[\|\nabla J_\mathcal{S}(w) - \nabla J(w)\|_2^2]}_{=\|\mathrm{Var}_\mathcal{S}(\nabla J_\mathcal{S}(w))\|_1} \leq \theta^2 \|\nabla J_\mathcal{S}(w)\|_2^2 \tag{12}$$

Computing $\mathrm{Var}_\mathcal{S}(\nabla J_\mathcal{S}(w))$ directly is also infeasible because it would require considering all samples of a certain size. Given a sample $\mathcal{S}$, the variance of all samples of that size can instead be approximated by

$$\|\mathrm{Var}_\mathcal{S}(\nabla J_\mathcal{S}(w))\|_1 \approx \frac{1}{|\mathcal{S}|(|\mathcal{S}| - 1)} \sum_{i \in \mathcal{S}} \|\nabla \ell(h_w(x_i), y_i) - \nabla J_\mathcal{S}(w)\|_2^2 \tag{13}$$

This approximation assumes that $|\mathcal{S}| \ll N$. Using (13) we can now estimate (12) which in turn estimates (11). If we estimate that (12) is not satisfied, i. e. that the sample gradient is likely to deviate significantly from the objective gradient, a larger sample $\widehat{\mathcal{S}}$ has to be used. In principle we could simply increase the sample size by a constant amount repeatedly and recheck (12) but this is slow if $|\mathcal{S}|$ is far off from satisfying the condition. Instead we will adaptively choose $|\widehat{\mathcal{S}}|$ s. t. it is expected to satisfy (12):

$$|\widehat{\mathcal{S}}| = \frac{|\mathcal{S}|\,\|\mathrm{Var}_{\mathcal{S}}(\nabla J_{\mathcal{S}}(w))\|_1}{\theta^2\|\nabla J_{\mathcal{S}(w)}\|_2^2} \tag{14}$$

Please refer to Byrd et al. [3, chapter 3] for a more detailed explanation of (13) and (14). To incorporate the ideas described above into SGD (12) has to be checked after each gradient descent step. If the check fails, the size of the following samples has to be increased according to (14). Good values for the initial sample size and for θ have to be found via hyperparameter optimization.

The idea outlined above can similarly also be applied to other gradient based optimization methods like the curvature-aware *Newton Conjugate Gradient* (NCG) method. It not only uses $\nabla J_{\mathcal{S}}(w)$ but also information from the Hessian $\nabla^2 J_{\mathcal{S}}(w)$ to compute the direction d of the next step. We refer to Byrd et al. [3, chapter 5] for the details.

Evaluation Subsample size selection was evaluated on a multi-class logistic regression problem using NCG for optimization. At first we look at the accuracy of the estimation of (11) via (13) and (12). On average $\|\mathrm{Var}_{\mathcal{S}}(\nabla J_{\mathcal{S}}(w))\|_1$ deviates about 4% from $\|\nabla J_{\mathcal{S}}(w) - \nabla J(w)\|_2$ on the evaluation dataset if $|\mathcal{S}| \ll N$.

Figure 8 shows that this accuracy is sufficient. Dynamic subsample size selection reaches the same quality as the batch method (fixed $|\mathcal{S}| = N$) while using significantly fewer datapoints. This in turn makes it significantly faster. The speed of convergence however does depend on the choice of θ. If θ is too small (see $\theta = 0.1$), $|\mathcal{S}|$ is increased quickly which slows down the optimization. If θ is too big (see $\theta = 0.75$), $\nabla J_{\mathcal{S}}(w)$ is allowed to deviate significantly from $\nabla J(w)$ which causes more erratic gradient steps.

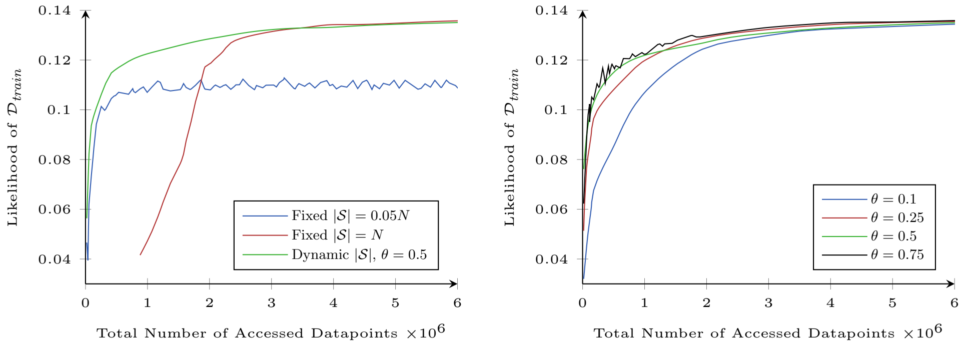


Fig. 8: Results on a multi-class logistic regression task using NCG. (Left) Comparison of dynamic subsample size selection with fixed sample sizes. (Right) Comparison of different values for θ.

3.3 Subsampling for Logistic Regression

Subsampling usually increases the *mean squared error* (MSE) of the resulting hypothesis compared to one that is trained on the full dataset $\mathcal{D}_{train}$. Let $\mathcal{S} := \{(x_i^*, y_i^*)\}_{i=1}^r$ be a random subsample of $\mathcal{D}_{train}$ that is drawn with or without replacement according to the probabilities $\{\pi_i\}_{i=1}^N$ where $N = |\mathcal{D}_{train}|$ and $\sum_{i=1}^N \pi_i = 1$. Usually $\mathcal{S}$ is drawn from a uniform distribution, i. e. each datapoint x_i is drawn with probability $\pi_i = N^{-1}$. Then a *maximum likelihood estimate* (MLE) $\beta_\mathcal{S} = (\beta_\mathcal{S}^{(1)}, \dots, \beta_\mathcal{S}^{(d)})$ is calculated as an estimate of the objective parameter vector $\beta_{\mathcal{D}_{train}}$ that maximizes the likelihood of the entire dataset. This strategy is often not optimal since some datapoints might have a smaller influence on $\beta_{\mathcal{D}_{train}}$ than others. The core idea now is to choose the probabilities π_i s. t. more informative datapoints are more likely to be sampled.

Case Control A simple idea to adjust the probabilities is to use *Case-Control subsampling* (CC) in which a roughly equal amount of positive and negative samples is drawn. Let $\mathcal{D}_{train}^+ := \{(x, y) \in \mathcal{D}_{train} \mid y = 1\}$ and $\mathcal{D}_{train}^- := \{(x, y) \in \mathcal{D}_{train} \mid y = 0\}$. CC samples would be chosen without replacement with probabilities

$$\pi_i = \begin{cases} |\mathcal{D}_{train}^+|^{-1} & \text{if } y_i = 1 \\ |\mathcal{D}_{train}^-|^{-1} & \text{if } y_i = 0 \end{cases} \tag{15}$$

This approach however is problematic because it introduces a bias towards unambiguous samples (see fig. 9).

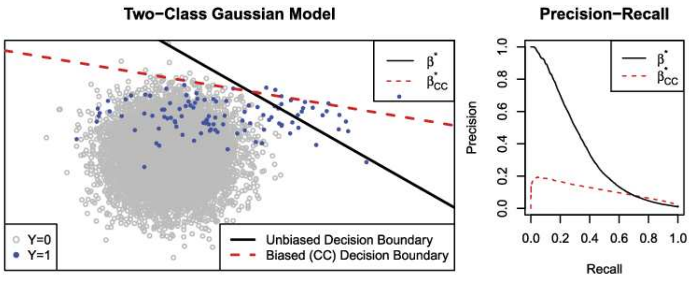


Fig. 9: Illustration of the bias introduced by CC.

Local Case Control Fithian and Hastie [5] proposed *Local Case-Control subsampling* (LCC) to remove the bias from CC. LCC determines the sampling probabilities π_i via a pilot estimate β_0. The estimate β_0 is the MLE of a small pilot sample $\mathcal{S}_0$ that is drawn with uniform sample probabilities. Datapoints are now weighted by the error of the pilot estimator on them:

$$\pi_i = \frac{|y_i - p(x_i \mid \beta_0)|}{\sum_{j=1}^N |y_j - p(x_j \mid \beta_0)|} \text{ with } p(x \mid \beta) = \frac{1}{1 + \exp(\beta^T x)} \tag{16}$$

Then a sample $\mathcal{S}_{\text{LCC}}$ is drawn using those probabilities, typically with replacement since this is computationally less expensive. This produces an estimate $\beta_{\mathcal{S}_{\text{LCC}}}$ that is consistent with $\beta_{\mathcal{D}_{train}}$, i. e. $\|\beta_{\mathcal{S}_{\mathrm{LCC}}} - \beta_{\mathcal{D}_{train}}\|_2 \to 0$ as $r \to \infty$. Additionally LCC prioritizes datapoints that are close to the decision boundary estimated by the pilot. This tends to reduce the variance of the estimate $\beta_{\mathcal{S}_{\mathrm{LCC}}}$, especially if $\mathcal{D}_{train}$ contains an imbalanced amount of positive and negative samples.

OSMAC While LCC tends to reduce the estimate's variance, it does not necessarily minimize it. The ***O****ptimal* ***S****ubsampling* ***M****otivated by the* ***A****-Optimality* ***C****riterion* (OSMAC) [9] method improves upon LCC by minimizing the expected variance. Like LCC it also uses a pilot estimate β_0 but the sampling probabilities π_i are then calculated differently.

Let $V := \mathrm{Cov}(\beta_{\mathcal{S}} - \beta_{\mathcal{D}_{train}})$ be the covariance matrix of the difference between the sample estimate and the complete dataset estimate. Given $\mathbb{E}[\beta_{\mathcal{S}} - \beta_{\mathcal{D}_{train}}] = \mathbf{0}$, V can be interpreted as a measure of the expected error introduced by subsampling. Using the A-optimality criterion of optimal design, OSMAC sets the sampling probabilities π_i so that $\mathrm{tr}(V)$ is minimized in expectation. More intuitively this minimizes the sum of the MSEs on the regression coefficients $\beta_{\mathcal{S}}^{(k)}$, i. e. $\sum_{k=1}^{d} \mathbb{E}[(\beta_{\mathcal{S}}^{(k)} - \beta_{\mathcal{D}_{train}}^{(k)})^2]$.

It turns out that finding the minimizing probabilities π_i of $\mathrm{tr}(V)$ is computationally expensive. However the optimal values for π_i can be approximated using

$$\pi_i = \frac{|y_i - p(x_i \mid \beta_0)| \cdot \|x_i\|_2}{\sum_{j=1}^{N} |y_j - p(x_j \mid \beta_0)| \cdot \|x_j\|_2} \tag{17}$$

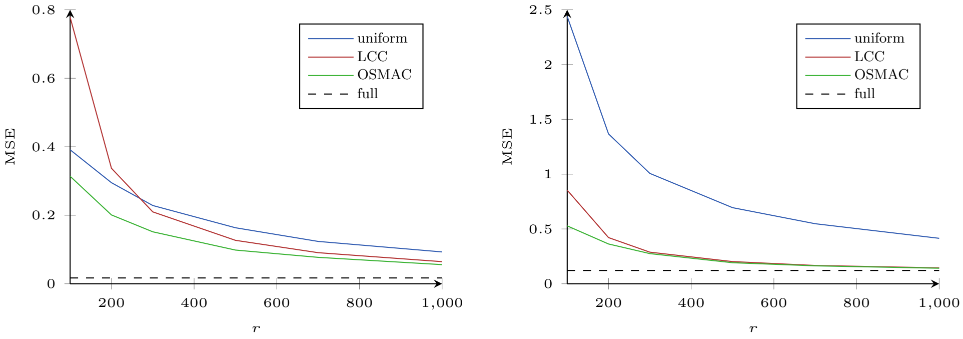
The only difference to LCC here is the added $\|x_i\|_2$ factor. The intuition behind this is that samples with large norms tend to be further away from the decision boundary[1]. An incorrectly classified sample that is far from the decision boundary is more surprising than an incorrectly classified sample close to it. Since the sigmoidal p function saturates quickly this fact is often not captured by LCC sampling.

Evaluation We will now compare uniform, LCC and OSMAC sampling on the two datasets. The datapoints x_i are randomly sampled from different distributions. The corresponding classes $y_i \in \{0, 1\}$ are then assigned using a fixed coefficient vector β. These two datasets are used:

- **mzNormal:** Uses a multivariate normal distribution $\mathcal{N}(0, \Sigma)$ with mean 0 and $\Sigma_{ij} = 0.5^{\delta_{i \neq j}}$. Contains a roughly equal amount of positive and negative samples.
- **nzNormal:** Uses a multivariate normal distribution $\mathcal{N}(1.5, \Sigma)$ with mean 1.5. About 95% of the samples are positive.

Figure 10 shows the MSEs $\|\beta_{\mathcal{S}} - \beta\|_2^2$ for different subsample sizes r. OSMAC consistently gives the closest approximation of β, confirming its theoretical A-optimality.

[1] This intuition is not entirely correct since it does not consider the offset and rotation of the decision boundary. Those aspects are ignored because (17) only approximates the A-optimal probabilities. For details please refer to Wang et al. [9, chapter 3.2].



(a) mzNormal

(b) nzNormal

Fig. 10: MSEs on β for different subsample sizes r.

3.4 WKM-SVM

To speed up the training of SVMs Almeida et al. [1] proposed a simple method that reduces the dataset size via k-means clustering. It can be described as a simple three-step procedure:

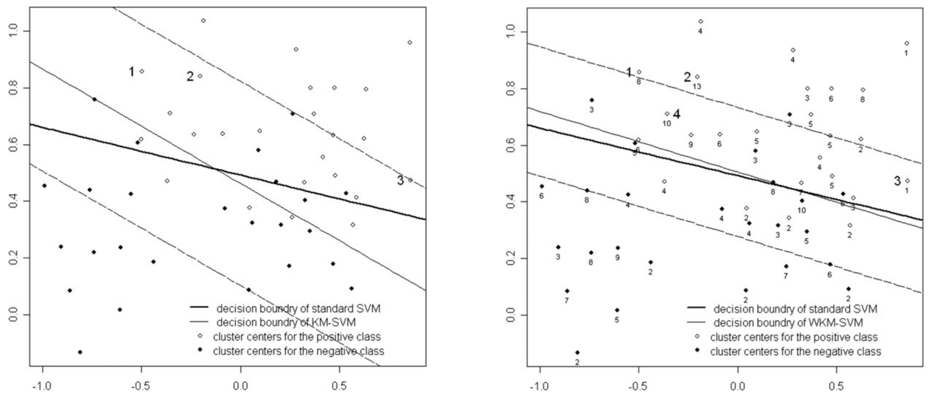
1. Group the training samples $\mathcal{D}_{train}$ into k clusters $C_1, \dots, C_k$ with centers $c_1, \dots, c_k$ where k should be determined via hyperparameter optimization.
2. Check for each cluster C_i whether all associated datapoints belong to the same class, i. e. $\exists\, z \in \{+1, -1\} : \forall (x, y) \in C_i : y = z$. If yes, all datapoints in C_i are removed from $\mathcal{D}_{train}$ and replaced by c_i. If not, they are kept in the dataset. The intuition behind this is that clusters with points from multiple classes might be near the decision boundary so they are kept to serve as potential support vectors.
3. Finally standard SVM training is performed on the reduced training dataset.

This approach performs similarly to SVM training on the full dataset. However the size of the reduced dataset is very unpredictable. Large homogeneous clusters with only a few noisy outliers of a different class will not be used to reduce the dataset size. Because of that the effective speedup and memory requirements may vary greatly depending on the dataset.

Lee et al. [8] proposed KM-SVM, an alternative approach that solves this problem by performing clustering on the datapoints of each class separately. This method has more predictable time and memory requirements but it also tends to modify the structure of the dataset. Figure 11a shows that this can cause KM-SVM to deviate significantly from the full dataset SVM.

WKM-SVM [2] improves upon KM-SVM by weighting each cluster center c_i by the amount of datapoints $|C_i|$ it represents. This solves the problem that small clusters of outliers have the same influence on the decision boundary as big clusters of more representative datapoints in KM-SVM. Figure 11b shows how this improves the quality of the decision boundary.

Evaluation We will now compare KM-SVM and WKM-SVM using different compression rates $R \in \{1, 3, 5, 10\}$ that describe the number of clusters $k = \frac{|\mathcal{D}_{train}|}{R}$. A



(a) KM-SVM **(b)** WKM-SVM

Fig. 11: Comparison of the decision boundaries of KM-SVM and WKM-SVM.

compression rate of $R = 1$ corresponds to a SVM that is trained on the entire dataset. For $\mathcal{D}_{train}$ the `PimaIndiansDiabetes2` dataset is used. Figure 12 shows that WKM-SVM consistently performs better than KM-SVM with a roughly identical training time.

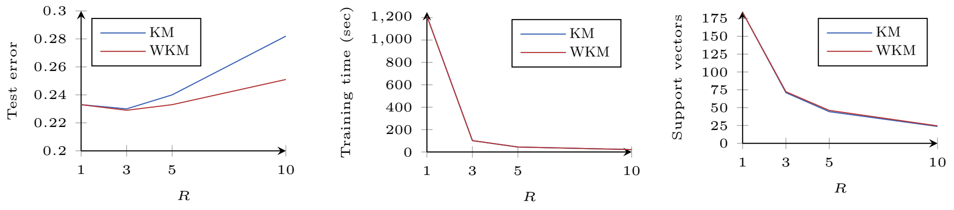


Fig. 12: Comparison of KM-SVM and WKM-SVM.

4 Conclusion

TODO

References

[1] M. Barros de Almeida, A. de Padua Braga, and J. P. Braga. “SVM-KM: speeding SVMs learning with a priori cluster selection and k-means”. In: *Proc. Vol.1. Sixth Brazilian Symp. Neural Networks*. Nov. 2000, pp. 162–167 (cit. on p. 14).

[2] Sungwan Bang and Myoungshic Jhun. “Weighted Support Vector Machine Usingk-Means Clustering”. In: *Communications in Statistics - Simulation and Computation* 43.10 (2014), pp. 2307–2324 (cit. on p. 14).

[3] Richard H. Byrd, Gillian M. Chin, Jorge Nocedal, and Yuchen Wu. “Sample size selection in optimization methods for machine learning”. In: *Mathematical Programming* 134.1 (2012), pp. 127–155 (cit. on pp. 10, 11).

[4] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. “Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves”. In: *Proceedings of the 24th International Conference on Artificial Intelligence.* IJCAI’15. Buenos Aires, Argentina: AAAI Press, 2015, pp. 3460–3468 (cit. on p. 6).

[5] William Fithian and Trevor Hastie. “Local case-control sampling: Efficient subsampling in imbalanced data sets”. In: *Annals of Statistics 2014, Vol. 42, No. 5, 1693-1724* (June 16, 2013). arXiv: `1306.3706v2 [stat.CO]` (cit. on p. 12).

[6] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. “Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets”. In: (May 23, 2016). arXiv: `1605.07079v2 [cs.LG]` (cit. on pp. 2, 3).

[7] Ariel Kleiner, Ameet Talwalkar, Purnamrita Sarkar, and Michael I. Jordan. “A Scalable Bootstrap for Massive Data”. In: (Dec. 21, 2011). arXiv: `1112.5016v2 [stat.ME]` (cit. on p. 8).

[8] S. J. Lee, C. Park, M. Jhun, and J-Y. Ko. “Support vector machine using K-means clustering”. English. In: *JOURNAL OF THE KOREAN STATISTICAL SOCIETY* 36.1 (2007), 175–182 (cit. on p. 14).

[9] HaiYing Wang, Rong Zhu, and Ping Ma. “Optimal Subsampling for Large Sample Logistic Regression”. In: (Feb. 3, 2017). arXiv: `1702.01166v2 [stat.CO]` (cit. on p. 13).