# ADABench - Towards an Industry Standard Benchmark for Advanced Analytics

Tilmann Rabl[1,2], Christoph Brücke[2], Philipp Härtling[2], Stella Stars[2], Rodrigo Escobar Palacios[3], Hamesh Patel[3], Satyam Srivastava[3], Christoph Boden[4], Jens Meiners[5], and Sebastian Schelter[6]

[1]HPI, University of Potsdam, [2]bankmark, [3]Intel, [4]Mercedes Benz.io, [5]EWE, [6]NYU
`tilmann.rabl@hpi.de, {christoph.bruecke, philipp.haertling, stella,stars}@bankmark.de, {rodrigo.d.escobar.palacios,hamesh.s.patel,satyam.srivastava}@intel.com, christoph.boden@mercedes-benz.io, j.meiners@ewe.de, sebastian.schelter@nyu.edu`

**Abstract.** The digital revolution, rapidly decreasing storage cost, and remarkable results achieved by state of the art machine learning (ML) methods are driving widespread adoption of ML approaches. While notable recent efforts to benchmark ML methods for canonical tasks exist, none of them address the challenges arising with the increasing pervasiveness of end-to-end ML deployments. The challenges involved in successfully applying ML methods in diverse enterprise settings extend far beyond efficient model training.
In this paper, we present our work in benchmarking advanced data analytics systems and lay the foundation towards an industry standard machine learning benchmark. Unlike previous approaches, we aim to cover the complete end-to-end ML pipeline for diverse, industry-relevant application domains rather than evaluating only training performance. To this end, we present reference implementations of complete ML pipelines including corresponding metrics and run rules, and evaluate them at different scales in terms of hardware, software, and problem size.

## 1 Introduction

Enterprises apply machine learning (ML) to automate increasing amounts of decision making. This trend, fueled by advancements in hardware and software, has spurred huge interest in both academia and industry [36]. Today, the market for ML technology and services is growing and this trend will even increase in the coming years [13]. This has lead to the development of a huge variety of ML tools and systems, which are providing ML functionality at different levels of abstraction and in different deployment infrastructures. Examples include recently popularized deep learning systems such as TensorFlow [2], PyTorch [25], and MXNet [10], which greatly benefit from specialized hardware such as GPUs; well established Python-based libraries such as scikit-learn [26]; and solutions on the Java Virtual Machine such as SparkML [23] and Mahout Samsara [33], which are based on general-purpose data flow systems and integrate well with common cloud infrastructures. Additionally, there are many specialized cloud ML services, which typically address only certain verticals such as machine translation, object detection, or forecasting.

This diversity makes it increasingly difficult for users to choose between the options for real-world use cases and has led to one-dimensional methods of comparison, which either measure the best accuracy achievable or the speed of the model training in isolation. While both metrics are important, they do not address the fundamental challenges of ML in practice. In practice, one has to not only train ML models, but execute end-to-end ML pipelines, which include multiple phases such as data integration, data cleaning, feature extraction, as well as model serving. Many of these phases have hyperparameters (e.g., the vector dimensionality if feature hashing is used), which need to be tuned analogously to model hyperparameters. All hyperparameters of an ML pipeline will impact both accuracy and speed of a model, and successfully deploying an ML system requires to trade-off the two. The artificial restriction on only measuring model training has led to a narrow focus on a few well researched data sets, such as Netflix [5], MNIST [20], or ImageNet [19]. While these data sets are highly relevant to the specific tasks at hand, they are not necessarily representative for end-to-end real-world ML pipelines “in the wild”, which often include data integration, data cleaning, and feature extraction as the most tedious tasks.

We discuss some of the neglected challenges in these tedious tasks in the following: The input data may dramatically vary in size across the pipeline, it could reside in relational tables that need to be joined, and numerical representations may be sparse or dense. The emphasis of workloads may vary between exploration and continuous improvement of model quality and model serving. Furthermore, the requirements encountered in industrial settings extend beyond model accuracy as solutions have to scale, be reasonably easy to maintain and adapt, and experimentation has to adhere to strict budgets. These notable differences between real-world requirements and offline evaluations on static, pre-processed data sets also led Netflix to discard the wining solutions the Netflix Prize contest, as *“the additional accuracy gains [...] measured did not seem to justify the engineering effort needed to bring them into a production environment.”* [3]. An industry standard benchmark for advanced analytics should reflect these trade-offs and requirements. It should cover complex end-to-end ML pipelines and include a broad range of use cases beyond well established data sets.

In this paper, we outline ADABench, an end-to-end ML benchmark addressing these challenges. Unlike previous benchmarks [1,5,19], we cover the complete ML lifecycle, from data preparation all the way to inference. The benchmark is built to cover real business requirements and includes different scale ranges, which are mapped to problem sizes typically found in industry. As a basis, we use an up-to-date market potential analysis [13], which we intend to convert into 16 use cases, out of which we present six in this paper. These use cases are designed to cover various dimensions of analytics in the retail business vertical. Our benchmark model draws from industry standards as specified from the Transaction Processing Performance Council[1] (TPC) and the Standard Performance Evaluation Corporation[2] (SPEC). Our core contributions are:

– We propose a novel end-to-end ML benchmark that covers different business scale factors, and industry relevant metrics (Section 2).
– We specify the first four out of 16 planned use cases that cover a wide range of industry-relevant ML applications (Section 2.1).
– We detail our reference implementations of the proposed use cases, and report first performance evaluations for different scale factors (Section 3).

In the following, we will give an overview of the implemented use cases, outline the benchmark specification, and present first results running our use cases along several dimensions of scalability, namely software (Python-environment vs. Spark-environment), hardware (single-node vs. scale-up cluster), and size (MBs vs. GBs).

## 2 Machine Learning Benchmark

ADABench covers the complete end-to-end pipeline of ML workloads as described by Polyzotis et al. [28], which contains several additional steps besides model training, such as data integration, data cleaning, feature extraction, and model serving. From a business perspective, it is relevant to consider that people conducting advanced data analytics usually spend most of their time in tasks like data preparation [15]. Given this widely agreed-on fact, an ML benchmark should also consider these phases. In general, each of the phases can be a one-time action, as in a single prediction; an iterative process, e.g., a daily forecast; or a continuous process, as in an online recommendation scenario. In the following, we describe the first four use cases in detail and give details about the core aspects of the benchmark specification.

### 2.1 Use Cases

To account for the diversity of real-world scenarios, the ADABench workload is comprised of multiple ML use cases, each covering a whole pipeline. Our use cases are derived from retail business applications and analysis tasks common in online businesses according to their market potential identified by McKinsey [13]. We select the first two cases out of the two highest impact business problem areas: customer service management and predictive maintenance. The second two use cases cover prediction and hyper-personalization in form of customer segmentation. The final benchmark aims to not only provide a specification and a data generator but also reference implementations for each use case. The current version includes two such reference implementations for each use case: (a) an implementation based on Apache Spark [40], and (b) an implementation based on

[1] `http:\www.tpc.org`
[2] `http:\www.spec.org`

Python libraries such as scikit-learn [26] and pandas [22]. We adapt the Parallel Data Generation Framework (PDGF) for data generation [31].

**Use Case 1: 'Trip-Type Classification'** The first use case (UC1) is a classification task that falls into the category of customer service management. The goal is to identify the type of shopping a customer wants to do, i.e., if a customer is on a large weekly grocery trip, wants to buy ingredients for a dinner, or is shopping for clothing. Based on the classification, the shopping experience of customers can be improved. UC1 is inspired by a Kaggle competition[3]. However, the data set and implementation are completely independent. The use case employs structured data from a transaction database as input. These transactions include the amount of items bought or returned, the department from which they originate, a line number, the day of the week, an order ID to distinguish different sessions, and a trip type. The task is to train a classifier to predict the trip type based on the transactional data. Our reference implementation is based on XGBoost[9] for model training, and pandas / scikit-learn and Apache Spark for general data handling and pre-processing. The pipeline first loads the data into memory and filters all samples that contain NULL values. Then the features are extracted. There are three types of features: (a) aggregates, i.e., sum and absolute sum of items bought or returned per order, (b) a one-hot encoding of the weekday of the order, and (c) a pivotization of the different departments occurring in the order, i.e., sum of items bought or returned per department per order. The basic task is to unstack the data, such that a single order is expressed as a single vector instead of a set of records. After pre-processing, XGBoost is trained using *softprob* as objective function. The key performance metric for training and serving is throughput while a threshold on the weighted F1-measure has to be met.

**Use Case 2: 'Predictive Maintenance'** The second use case (UC2) is a classification task in the category of predictive maintenance. It is inspired by an approach to predict failures published by Chigurupati et al. [11]. The goal is to predict hardware failures in advance based on sensor data measured in periodical time intervals. The key idea is to separate the log data into categories depending on the timely distance to failure. A proxy for this generic use case is to predict imminent hard-drive failures based on daily S.M.A.R.T.[4] measurements [35]. We provide semi-structured data from log files as input. As part of the preprocessing stage, this data needs to be parsed and filtered to be further used for training or serving. The main challenge is a highly imbalanced data set, i.e., non-failures are much more common than failures. The characteristics and distributions of the synthetic data set are inspired by hard drive statistics published by Backblaze[5]. The data set contains log entries with a time stamp, an ID (model and serial number), a series of S.M.A.R.T. attribute values, and a binary label indicating whether a complete failure occurred. Analogous to the first use case, the reference implementations for this use case are also based on pandas, scikit-learn, and

[3] `https://www.kaggle.com/c/walmart-recruiting-trip-type-classification`
[4] Self-Monitoring, Analysis and Reporting Technology
[5] `https://www.backblaze.com/b2/hard-drive-test-data.html`

Spark. The main tasks of the preprocessing stage are (a) labeling of the training samples and (b) up-sampling of the minority (failure) class. For that, we split the training samples into failures, log entries that occurred on the day or one day before the actual failure, and non-failures, log entries that occurred two or more days before a failure or log entries of disks that never failed anyway. Then the samples of the failure class are sampled with replacement until there are equal amounts of samples for each class, the up-sampled data set is used to train a support vector machine (SVM). Since imminent failures are predicted to enable pro-active measures (such as replacing faulty drives before they crash), the key quality metric here is a low false positive rate at a high true positive rate. We measure this using area under the receiver operating curve (AUC). The key performance metric is throughput again in this case.

**Use Case 3: 'Sales Prediction'** The third use case (UC3) is a regression task in the category of forecasting. The aim is to predict store and/or department sales from historical sales data, i.e., weekly sales numbers from previous years are used to find seasonal and regional patterns and forecast future sales. Based on these predictions stores can optimize their expenditure for sales promotion and personnel planning. UC3 is borrowing the idea from a Kaggle competition[6] but the data generation and implementation are independently built. The data consists of sales numbers aggregated by stores, departments, and weeks over several years. Every store has sales data for every department over every week of the years. Sales peaks are sampled from a normal distribution. Sales curves between departments follow a similar seasonal pattern. No other additional information, e.g., product group or amount of sale, is provided. Our data is generated and then exported into a CSV file. As in the other use cases our reference implementation uses pandas, scikit-learn, and Spark. In the first step of our pipeline, the sales numbers are loaded from that CSV. A preprocessing step is not necessary in this use case, since we already generate aggregated data. In the training phase, we build a seasonal model for each store and each department using a Holt-Winters exponential smoothing approach. This model is then used to predict forecasts for all stores and departments and exported into a CSV file. The key performance metric is processing time of each step, we measure the mean squared error as a quality metric.

**Use Case 4: 'Customer Segmentation'** The forth use case (UC4) is a clustering task. Clustering is an unsupervised machine learning technique. In customer segmentation, customers are divided into groups based on similar characteristics. By segmenting users, campaigns can be more efficient, because they can be directed to the correct target audiences. UC4 performs behavioral segmentation, as the problem to be solved is finding clusters based on aggregate features that group customers based on their spending and returning behavior. The input in this use case consists of order and return transactions. We use k-means clustering to calculate user clusters based on the data provided. The patterns in the data are used to identify and group similar observations. First, we load the data into memory and clean it (we remove unnecessary elements,

[6] `https://www.kaggle.com/c/walmart-recruiting-store-sales-forecasting`

e.g., duplicates, transactions not assigned to a customer, or cancel orders without counterpart). The features extracted after that are (a) return ratio, i.e., total value of the returns divided by the total value of the orders and (b) number of purchases per customer. The scaled feature vector and the number of clusters are input for the k-means algorithm used in training. The key performance metric for training and serving is throughput. The quality metric measures the mean of the distances of the predicted clusters to the clusters of the reference implementation.

**Use Case 5: 'Spam Detection'** The fifth use case (UC5) is a supervised classification task in the category of discovering new trends/anomalies. Spam detection means to find comments, reviews, or descriptions with spam content. The input in this use case consists of reviews or comments in a retail business. The problem to be solved is to identify those reviews that are spam. The analysis uses Naive Bayes. Naive Bayes methods are a set of supervised learning algorithms based on Bayes' theorem. The data loaded into memory for training is text labeled as spam or ham, for serving unlabeled text. After cleaning (e.g., removing duplicates), the pipeline performs prepocessing to convert the text into a numerical feature vector (like tokenization of the text, removing stopwords, building n-grams, and computing the term frequency–inverse document frequency). This feature vector is fed to the multinomial Naive Bayes algorithm. The output after serving is an array of label predictions. The performance metric is throughput, the time that is needed to run through all the steps of reading the data, data transformation, training and serving. To evaluate the quality of the classification, the quality metric is *F1 Score*, since it takes both precision and recall of the classifications into consideration.

**Use Case 6: 'Product Recommendation'** Use Case 6 (UC6) is a recommendation task in the category of personalization. The aim is to recommend items to a user based on their and other users previous similar ratings, i.e., find next-to-buy recommendations. Proof of concept has been done with the MovieLens dataset[7] that consists of user-item-rating triplets for various movies. Our data generation and implementation creates similar triplets. Each user is given a main user category for which the bulk of product ratings are generated. The rest is filled with ratings from other user categories. Ratings for products in their main category have a normal distribution around a high mean, ratings in the other categories have a normal distribution around a low mean. Our data is generated and exported into a CSV file. We use pandas, scikit-learn, and Spark for our reference implementation. The actual recommendation is done by SurPRISE[8] in Python and by the collaborative filtering module of Spark[9]. As the first step of our pipeline the pre-generated ratings are loaded. In a preprocessing step in Python we transform the pandas DataFrame in a dataset that SurPRISE can utilize. The Spark implementation does not need a preprocessing step. The training phase consists of a matrix factorization done with a singular value decomposition algorithm in python and an alternating least squares algorithm in

[7] `https://grouplens.org/datasets/movielens/`
[8] `http://surpriselib.com/`
[9] `https://spark.apache.org/docs/latest/ml-collaborative-filtering.html`

Spark. The resulting model is then used to predict the rating for every user-item pair and lastly to return the top 10 recommendations for every user. The key performance metric is throughput. As a quality metric we use mean-squared-error and mean-absolute-error.

For the remaining 10 use cases (which are out of the scope of this paper), we aim for diversity along several dimensions: the structure of the data (structured vs. unstructured), type of data (numerical, audio, visual, textual, etc.), business case (predictive maintenance, customer service management, etc.), and ML methods and algorithms (classification, regression, recommendation, clustering, deep neural networks, etc.).

## 2.2 Benchmark Details

**Data Model, Scale Factor, and Execution.** The ADABench data model is structurally related to the BigBench data model [17,4]. Analogous to BigBench, we design a data warehouse model, which is used as a source for many of the ML use cases, especially for analyses related to customer management, sales, and other transactions. Additionally, we add sources such as click logs, text, and sensor readings. We further extended the data set to capture a broader variety of data types and demand more preprocessing. Some of the extensions are taken from the ideas presented in the proposal for BigBench 2.0 [30], i.e., diverse data types (text, multi-media, graph) and different velocities of data (streaming vs. batch).

| **Scale** | **Workload** | **Problem Size (GB)** | **Total Cores Approx.** |
|---|---|---:|---:|
| Extra Small (XS) | Small data science projects | $10^0$ | 4-8 |
| Small (S) | Traditional analytical tasks | $10^1$ | 8-32 |
| Medium (M) | Lower tier big data tasks | $10^3$ | 32-320 |
| Large (L) | Big data tasks | $10^5$ | 320-3200 |
| Extra Large (XL) | Internet company tasks | $10^7$ | 3200-32000 |

Table 1: Data size and number of cores for each scale.

ADABench incorporates different data set sizes that are used in industry and academia. We identified five different classes that are reflected in five scale factors, XS - XL, which represent data problem sizes of 1 GB to 10 PB respectively, shown in Table 1. These classes are related to the size of the operations found in industry. The smallest class, XS, relates to a very small operation typically done by a single individual at a single location. S represents an operation driven by a small team at a single location with lightly distributed computing capabilities. At Scale Factor M the use of a small cluster of commodity hardware is necessary. To run an L scale, a rather large cluster is required and this kind of setup is typically found in global organizations handling large amounts of data and transactions. Finally, we define the XL boundary, which is a quite rare scale factor only seen at large Internet companies. Data in ADABench is synthetically generated using tools such as PDGF [32]. The data generation process provides three major functionalities. ❶ *High data generation velocity* that achieves the described scale factors XS-XL in a reasonable time frame. ❷ *Data generation that incorporates characteristics and dependencies concerning the respective scale factor.* As a result, a higher data volume contains not more of the same information but new, previously unseen characteristics. Hence the data becomes more complex with an increasing scale factor. For this, we propose an approach that is build on the assumption that rare occurrences become more common, in absolute terms, the more data is generated. For instance in case of UC1 there might be only three different classes at scale factor XS in the synthetic data set, whereas there might be 40 at scale factor M. This approach can be generalized for several supervised learning tasks, by ensuring that certain phenomena have a high probability while others have a low probability. One artifact of this approach is that the different classes become more imbalanced with increasing volume, which in turn also makes the data more realistic. ❸ *Introduction of errors into the data* to simulate a real-world production environment. These errors can be, e.g., missing values, outliers, and wrong data types. While the first functionality eases the utilization of the benchmark, the other two aim to close the gap between laboratory grade experiments and real-world application fuzziness. For *UC 1* and *UC 3* - *UC 6* functionality two is highly important because large scale data should contain previously unseen information. For *UC 2* the third functionality is of higher importance since the false positive rate is the critical measurement and must be robust to erroneous input data.

In order to cover various business setups and challenge machine learning deployments in different ways, we specify two general modes of execution: Sequential execution and parallel execution of the use cases. During sequential execution, every use case has to be run exclusively from start to end, while in the parallel execution mode, all use cases can be run in parallel. In both setups, we allow for a pipelined execution of individual steps within a use case. This is similar to the TPC's benchmark execution model (e.g., TPC-DS [27] or TPCx-BB [4]), with the distinction that we do allow for the parallel execution of different use cases rather than the parallel execution of streams of all queries.

**Metrics.** In ADABench, we define individual metrics for each use-case, which are aggregated (via geometric mean) into one global metric summarizing the entire benchmark. For each use case, we define individual metrics, which are a combination of measurements of the stages in a use-case. We use the geometric mean to weight the diverse use cases equally in the final metric and equally encourage optimization on all stages. Driven by the industry-oriented scenario we aim to cover with ADABench, we chose *throughput* in *samples per second* as the main performance measure while defining thresholds for prediction quality metrics such as accuracy or AUC and latency that have to be met. This is similar to recent proposals like DAWNBench, which propose a time-to-accuracy metric [14] and reflects industry requirements. Most use cases define a training and a serving stage, which we measure separately ($TP_t$ and $TP_s$)and summarize in the total throughput for the use case $TP_i$ (see Equation 1). The formula for the total throughput is shown in Equation 2.

$$TP_i = \sqrt{TP_{ti} * TP_{si}} \tag{1}$$

$$TP = \sqrt[n]{\Pi_{i=1}^{n} TP_i} \tag{2}$$

The final performance metric, $ADASps@SF$, is the throughput at a scale factor. This means that systems can only be compared at the same scale factor. Besides performance metrics, we also specify price performance and energy performance metrics. These are defined according to TPC standards [38,37]. This means, the price performance is defined as the total cost of the execution by a pay-as-you-go model divided by $ADASps@SF$. Similarly, the energy performance is defined as the total consumed energy per benchmark execution divided by $ADASps@SF$.

## 3 Evaluation

In this section, we present results of an experimental evaluation of the six use cases UC1 to UC6 introduced in Section 2.1. While providing valuable insights about system performance, the experiments also show how the benchmark operates with respect to metrics and measurements. We evaluate the reference implementations of both use cases with a focus on the first four scale factors discussed in Section 2.2: XS, S, M, and L. We use two different compute clusters with nodes of the following specs:

- **Small node:** 1 Intel Xeon CPU @ 2.10GHz, 10 threads (virtualized) and 64 GB RAM
- **Big node:** 2 Intel Xeon CPUs @ 2.30GHz, 72 threads and 768 GB RAM

As discussed in Section 2.1, we have two different reference implementations for each use case: one based on pandas, scikit-learn, named *Python* and one based on Apache Spark. The experiments are deployed on one node for *Python* and on 10 nodes for *Apache Spark* for both *small* and *big* node configurations. Figure 1, 2, 3, 4, 5, and 6 show the results of these experiments for different scale factors. For each use case and scale factor, we generate three synthetic data sets: one for training, one for serving, and one for scoring. We use the data sets for training and serving for the throughput tests and the data set for scoring to measure model quality. As discussed in Section 2.2, *throughput* in samples per second is the main performance measure evaluated in the experiments. The trained models have to adhere thresholds for model quality: For instance the F-Measure for UC1 was above 0.8 and the AUC for UC2 was higher than 0.95. The other use cases have similar quality thresholds.

We found that for the trip type classification (UC1), Spark had a higher throughput during training, starting at a relatively low scale factor of 1 (see Figure 1, which approximately corresponds to 100,000 orders with an average of 10 line items per order, that is 1,000,000 line items in total. In case of the implementation based on pandas, scikit-learn, and XGBoost for UC1 approximately half of the time is spent for pre-processing the data and the remaining half is

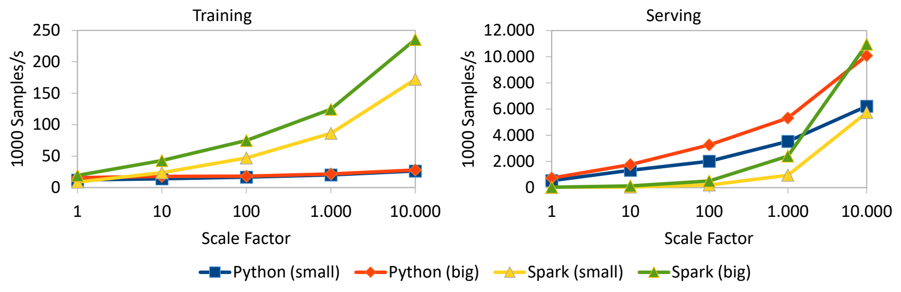

Fig. 1: Throughput measured for the training and serving stage at different scale factors. The Python-based implementation is shown in blue and red when run on a small or big node respectively. Analogous the Spark-based implementations are show in yellow and green.

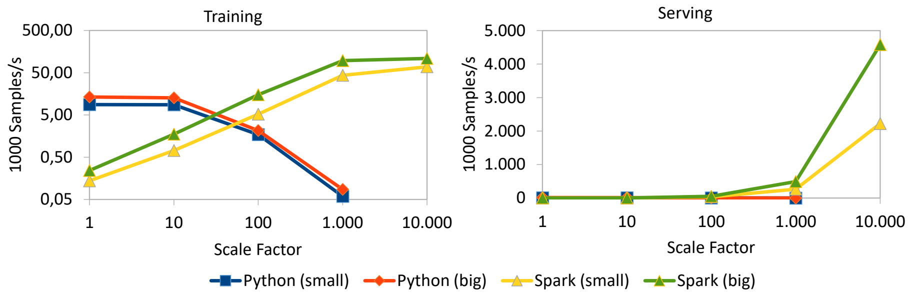

Fig. 2: Throughput measured for UC02 training and serving stage as in Figure 1.

spent for the actual model training. Since the pre-processing includes joins and pivotization, Spark is much better suited for this task and outperforms pandas using more efficient join strategies. Another reason why Spark performs better is XGBoost's parallelization capabilities. Using Spark and hence all 10 nodes of the cluster also enables XGBoost to leverage the increased computational power, i.e., distribute the training over all the nodes in the cluster. For the serving stage of UC1, we observe similar results, yet the tipping point is at a much higher scale factor. We conclude that serving an XGBoost model does not require much computational power; hence as long as the data fits into memory, the Python based implementation has higher throughput than Spark.

In contrast to UC1 where Spark outperforms a single node solution based on pandas and XGBoost, we see that for UC2 the data size has to be increased significantly before Spark outperforms the Python-based implementation (see Figure 2). We attribute this to the fact this use case is light on pre-processing. Only when the data becomes too big to fit into memory a Spark cluster performs better. For serving, we see the similar results as in UC1: a Spark cluster is only reasonable for high volume and/or velocity. One interesting finding is that the

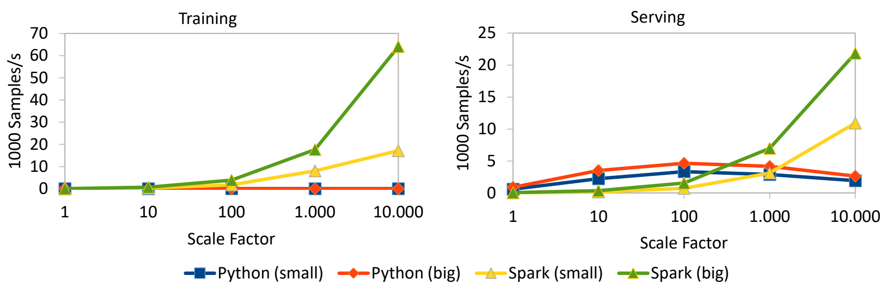

Fig. 3: Throughput measured for UC03 training and serving stage as in Figure 1.

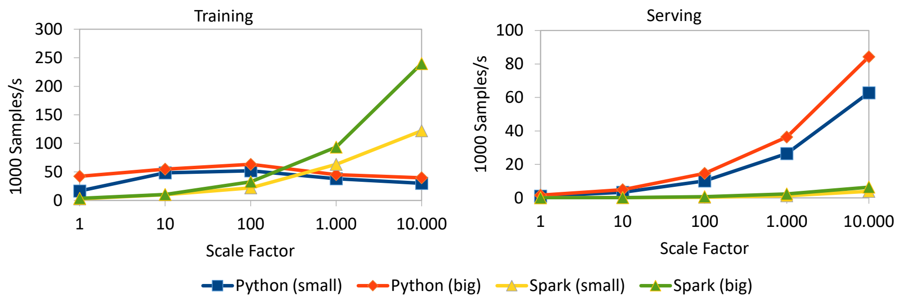

Fig. 4: Throughput measured for UC04 training and serving stage as in Figure 1.

serving throughput of the Python implementation for UC2 decreases drastically with an increasing model size, i.e., more training samples. This probably is based on the fact that there are more support vectors and it can most likely be compensated using a form of regularization.

The experiments for UC3 (see Figure 3) show a different characteristic. We can see that the Holt-Winters implementation for Spark is performing better for the training as well as for the serving stage. It can observed that with increasing scale factor the serving throughput decreases for the Python-based implementation. This is caused by memory pressure, which means that the more data is processed the lower is the throughput. For UC4, the clustering use-case, illustrated in Figure 4, we see similar effects as for UC2. As long as the data can be handled in memory on a single node the Python-based implementation is faster than the Spark implementation. But the limit, in terms of throughput, is reached rather fast and the Spark-based implementation is outperforming the Python-based implementation between scale factor 100 and 1000, depending on the available memory. Because the k-Means algorithm is highly parallelizable and the Spark environment has access to ten times as many cores, we see a drastic increase of the Spark implementation at bigger scale factors. Serving on the other hand does not require a lot of compute resource and both implementation have not reached their respective limits for scale factors up to 10000. In spam classification as in UC5

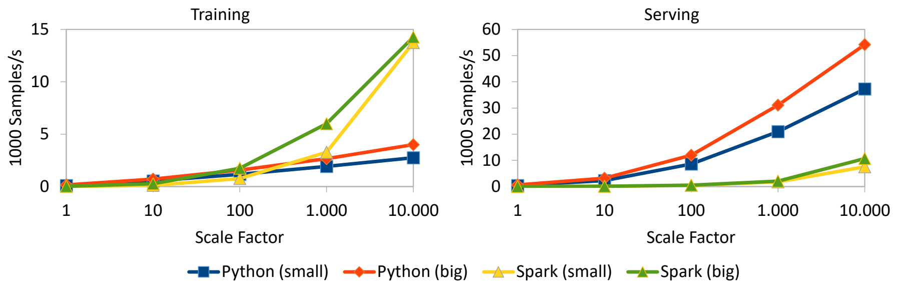

Fig. 5: Throughput measured for UC05 training and serving stage as in Figure 1.

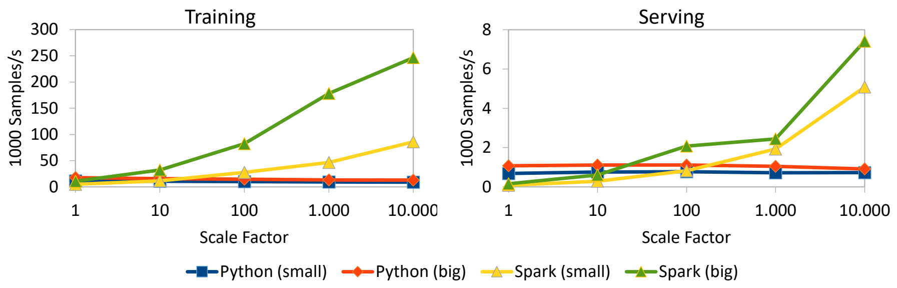

Fig. 6: Throughput measured for UC06 training and serving stage as in Figure 1.

(see Figure 5) neither implementation is reaching their throughput limit but for the training the Spark-based implementation has much better scaling capabilities than the Python-based one. The inverse is true for serving. The last use-case UC6 is a recommendation use-case (see Figure 6). It can be observed that with a higher scale factor and hence a bigger user-item-rating matrix performance is suffering for the Python-based implementation during training and constant during serving. Whereas the Spark-based implementation can make use of the additional computational power and has not reached its throughput limit up to scale factor $L$.

Since the main goal of this benchmark is to compare different ML systems for complete pipelines, we introduce a combined benchmark metric: the *ADABench Metric* – $ADASps@SF$ – as defined in Section 2.2. This metric is given by the geometric mean of all use case metrics, where each use case's metric is the geometric mean across the training and serving throughput. Using the geometric mean weights the use cases and their phases equally and, therefore, is more stable for diverse workloads. Figure 7 illustrates the ADABench Metric for our reference implementation and test systems at the different scale factors.

Given this metric, we see that a Spark-based implementation outperforms the Python-based implementation at high scale factors, while the Python-based

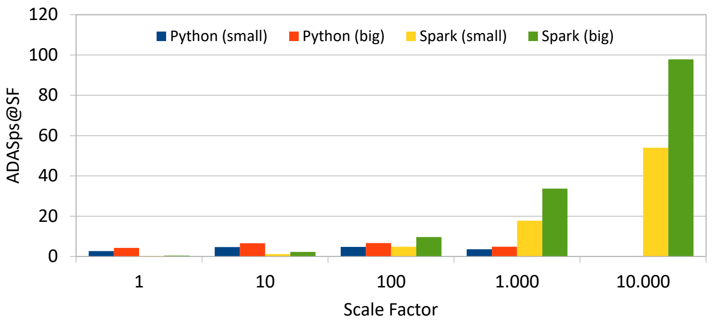

Fig. 7: ADABench Metric - Combined metric of all stages and use case per systems and scale factor (higher is better)

implementation clearly dominates at smaller scale factors, with scale factor 100 being the tipping point.

In summary, we can see that our initial set of use cases already captures a variety of performance characteristics. Some use cases rely heavily on pre-processing the data before training a model, while others employ algorithms that are computational more expensive. We can also see that our implementations in Python and Spark behave differently in scaling. In general, Python is more efficient for smaller scale factors, while Spark outperforms Python when data sets do not fit in memory. It has to be noted that in all experiments, Spark has 10 times more hardware resources than Python.

## 4 Related Work

In this section, we give an overview on benchmarking in the ML space in the past and previous approaches in benchmarking ML systems. We discuss popular benchmarks, their focus and metrics, and identify strengths and weaknesses.

In general, the current benchmarking landscape can be divided into three distinct types of benchmarks [18]: *Component-level benchmark*s test and compare only a single component of a system or application. They could, for example, run a specific operation of an application on different processors. Component benchmarks are interesting to the developers of the *component under study* (CUS) or to select the best instance of the CUS. We found that most of the current component-level benchmarks are hardware benchmarks in essence, which compare different hardware components in terms of training or inference performance. A typical component-level benchmark is *DeepBench* [24]. *System-level benchmark*s tests and compares complete ML systems. Typically, they cover set of different use cases in their workload and the metrics measure performance over a set of executions. *Algorithmic competitions* such as the Netflix Prize [5] or the ImageNet competition [19] compare different algorithms or models on a constrained problem. Recent work acknowledges that data preparation time is a major bottleneck for many ML applications and discusses a repository of related tasks to foster research in this area [34].

BigDataBench [16] is a benchmark with 13 representative real-world data sets and more than 40 workloads that can be characterized in five different domains: search engines, social networks, e-commerce, multimedia processing and bioinformatics. Moreover, the workloads have a varied data sources and cover diverse applications scenarios, including search engines, e-commerce, relational data queries, basic datastore operations (OLTP), and artificial intelligence (AI). It also uses two types of metrics: user-perceivable metrics, which are simple and easy to understand by non-expert users, and architectural metrics, which provide details targeted for architecture research. It can be categorized as a system-level benchmark. Unlike ADABench, BigDataBench does not include dedicated end-to-end scenarios.

AdBench [6] is a proposal system-level end-to-end benchmark with workloads that are representative of web and mobile advertising. The patterns exhibited by workloads are often seen in workloads from other sectors, such as financial services, retail, and healthcare. The benchmark combines workloads for ad-serving, streaming analytics, streaming ingestion, ML for ad targeting, and other big data analytics tasks. In addition, AdBench introduces a set of metrics to be measured for each stage of the data pipeline and scale factors of the benchmark. Unfortunately, there is no official implementation or complete specification of AdBench available.

*BigBench* [17,29,12] was developed can be seen as a basis for ADABench. Its strengths are that it is inspired by real-world use cases as well as its ease to use. A major drawback as far as a ML-specific benchmarking is concerned is its focus on traditional data analytics. BigBench only contains a small number of machine learning-related queries and features no metrics to compare different ML implementations.

DawnBench [14] as well as its industry-backed successor MLPerf [1] are benchmarking deep learning systems in terms of training time. The main contribution of DawnBench, and MLPerf consequently, is the novel time-to-accuracy metric. They are a hybrid between a system-level benchmark and an algorithmic competition. Both consists of a set of different and well-studied ML challenges, such as image classification, machine translation, sentiment analysis. There are many more benchmarks or benchmark attempts that are leveraging already existing competitions or challenges, e.g., convnet-benchmarks[10] or MLBench[21], but to the best of our knowledge, ADABench is the only end-to-end ML benchmark proposal.

Several studies have researched the performance characteristics of ML algorithms and workloads on specific systems. In contrast to these, we study end-to-end performance. However, we benefit from the insights of these studies and, for example, provide single- and multi-node implementations as proposed by Boden et al. [8,7].

[10] https://github.com/soumith/convnet-benchmarks

## 5 Conclusion

In this paper, we present our vision towards an industry standard benchmark for end-to-end machine learning. Our benchmark, ADABench, is comprised of complete machine learning use cases, which include data preparation and preprocessing, model training and tuning, as well as model serving. We use up-to-date market research to identify most relevant scenarios, and create novel machine learning pipelines for those. As an initial step, we have implemented four use cases in a Python (pandas, scikit-learn) and a Spark environment. Our evaluations show the different trade-offs in terms of performance based on data size, choice of environment, and complexity. We are currently implementing further use cases, also using deep neural networks. We will continue with the development of our benchmark with the goal of a wide coverage of machine learning technology and use cases along the dimensions identified by market research as most relevant, and aim to establish this proposal as a standard.

## References

1. MLPerf (2018), `https://mlperf.org/`, `https://mlperf.org/`
2. Abadi, M., Barham, P., Chen, J., et al.: Tensorflow: a system for large-scale machine learning. In: OSDI. pp. 265–283 (2016)
3. Amatriain, X.: Building Industrial-Scale Real-World Recommender Systems. In: RecSys. pp. 7–8 (2012)
4. Baru, C., Bhandarkar, M., Curino, C., et al.: Discussion of BigBench: A Proposed Industry Standard Performance Benchmark for Big Data. In: TPCTC. pp. 44–63 (2014)
5. Bennett, J., Lanning, S., et al.: The netflix prize. In: Proceedings of KDD Cup and Workshop. p. 35 (2007)
6. Bhandarkar, M.: AdBench: A Complete Data Pipeline Benchmark for Modern Data Pipelines. In: TPCTC. pp. 107–120 (2016)
7. Boden, C., Rabl, T., Markl, V.: Distributed machine learning-but at what cost. In: Machine Learning Systems Workshop at Conference on Neural Information Processing Systems (2017)
8. Boden, C., Spina, A., Rabl, T., Markl, V.: Benchmarking data flow systems for scalable machine learning. In: SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond. pp. 5:1–5:10. BeyondMR (2017)
9. Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: ACM SigKDD. pp. 785–794 (2016)
10. Chen, T., Li, M., Li, Y., et al.: Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
11. Chigurupati, A., Thibaux, R., Lassar, N.: Predicting hardware failure using machine learning. In: Reliability and Maintainability Symposium (RAMS). pp. 1–6 (2016)
12. Chowdhury, B., Rabl, T., Saadatpanah, P., Du, J., Jacobsen, H.A.: A BigBench Implementation in the Hadoop Ecosystem. In: WBDB. pp. 3–18 (2013)
13. Chui, M., Manyika, J., Miremadi, M., Henke, N., Chung, R., Nel, P., Malhotra, S.: Notes from the AI Frontier – Insights from Hundreds of Use Cases. Tech. rep., McKinsey Global Institute (2018)

14. Coleman, C., Kang, D., Narayanan, D., Nardi, L., Zhao, T., Zhang, J., Bailis, P., Olukotun, K., Ré, C., Zaharia, M.: Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. CoRR **abs/1806.01427** (2018)
15. Deng, D., Fernandez, R.C., Abedjan, Z., et al.: The Data Civilizer System. In: CIDR (2017)
16. Gao, W., Zhan, J., Wang, L., et al.: BigDataBench: {A} Dwarf-based Big Data and {AI} Benchmark Suite. CoRR **abs/1802.0** (2018)
17. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.A.: BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In: SIGMOD (2013)
18. Jain, R.: The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling. Wiley professional computing, Wiley (1991)
19. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. pp. 1097–1105 (2012)
20. LeCun, Y.: The mnist database of handwritten digits (1998), `http://yann.lecun.com/exdb/mnist/`
21. Liu, Y., Zhang, H., Zeng, L., Wu, W., Zhang, C.: Mlbench: Benchmarking machine learning services against human experts. PVLDB **11**(10), 1220–1232 (2018)
22. McKinney, W., et al.: Data structures for statistical computing in python. In: Proceedings of the 9th Python in Science Conference. pp. 51–56 (2010)
23. Meng, X., Bradley, J., Yavuz, B., et al.: Mllib: Machine learning in apache spark. The Journal of Machine Learning Research **17**(1), 1235–1241 (2016)
24. Narang, S.: DeepBench (2016), `https://svail.github.io/DeepBench/`
25. Paszke, A., Gross, S., Chintala, S., et al.: Automatic differentiation in pytorch. In: NIPS Autodiff Decision Workshop (2017)
26. Pedregosa, F., Varoquaux, G., Gramfort, A., et al.: Scikit-learn: Machine learning in python. Journal of machine learning research **12**, 2825–2830 (2011)
27. Poess, M., Rabl, T., Jacobsen, H.A.: Analysis of TPC-DS: the First Standard Benchmark for SQL-based Big Data Systems. In: Proceedings of the 2017 Symposium on Cloud Computing (2017)
28. Polyzotis, N., Roy, S., Whang, S.E., Zinkevich, M.: Data management challenges in production machine learning. In: SIGMOD. pp. 1723–1726 (2017)
29. Rabl, T., Frank, M., Danisch, M., Gowda, B., Jacobsen, H.A.: Towards a complete bigbench implementation. In: In Proceedings of the 2014 Big Big Data Benchmarking Workshop. pp. 3–11 (2015)
30. Rabl, T., Frank, M., Danisch, M., Jacobsen, H.A., Gowda, B.: The Vision of BigBench 2.0. In: Proceedings of the Fourth Workshop on Data Analytics in the Cloud. pp. 3:1–3:4. DanaC'15, ACM, New York, NY, USA (2015)
31. Rabl, T., Frank, M., Sergieh, H.M., Kosch, H.: A Data Generator for Cloud-Scale Benchmarking. In: TPCTC. pp. 41–56 (2010)
32. Rabl, T., Poess, M.: Parallel data generation for performance analysis of large, complex RDBMS. In: DBTest '11. p. 5 (2011)
33. Schelter, S., Palumbo, A., Quinn, S., Marthi, S., Musselman, A.: Samsara: Declarative machine learning on distributed dataflow systems. In: NIPS Workshop MLSystems (2016)
34. Shah, V., Kumar, A.: The ml data prep zoo: Towards semi-automatic data preparation for ml. In: DEEM. pp. 11:1–11:4 (2019)
35. Stevens, C.E.: Information TPC-Cechnology – ATA/ATAPI Command Set – 2 (ACS-2). Tech. rep., ANSI INCITS (2011)

36. Sun, C., Shrivastava, A., Singh, S., Gupta, A.: Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. In: ICCV (2017)
37. Transaction Processing Performance Council: TPC-Energy Specification (2012), version 1.5.0
38. Transaction Processing Performance Council: TPC Pricing Specification (2019), version 2.5.0
39. Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., Zhen, C., Lu, G., Zhan, K., Li, X., Qiu, B.: BigDataBench: a Big Data Benchmark Suite from Internet Services. In: HPCA (2014)
40. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: HotCloud. pp. 10–10 (2010)