

“Deep” Learning for Missing Value Imputation in Tables with Non-Numerical Data

Felix Biessmann*, David Salinas*, Sebastian Schelter, Philipp Schmidt, Dustin Lange

Amazon Research

{biessman,dsalina,sseb.phschmid,langed}@amazon.de

ABSTRACT

The success of applications that process data critically depends on the quality of the ingested data. Completeness of a data source is essential in many cases. Yet, most missing value imputation approaches suffer from severe limitations. They are almost exclusively restricted to numerical data, and they either offer only simple imputation methods or are difficult to scale and maintain in production. Here we present a robust and scalable approach to imputation that extends to tables with non-numerical values, including unstructured text data in diverse languages. Experiments on public data sets as well as data sets sampled from a large product catalog in different languages (English and Japanese) demonstrate that the proposed approach is both scalable and yields more accurate imputations than previous approaches. Training on data sets with several million rows is a matter of minutes on a single machine. With a median imputation F1 score of 0.93 across a broad selection of data sets our approach achieves on average a 23-fold improvement compared to mode imputation. While our system allows users to apply state-of-the-art deep learning models if needed, we find that often simple linear n -gram models perform on par with deep learning methods at a much lower operational cost. The proposed method learns all parameters of the entire imputation pipeline automatically in an end-to-end fashion, rendering it attractive as a generic plugin both for engineers in charge of data pipelines where data completeness is relevant, as well as for practitioners without expertise in machine learning who need to impute missing values in tables with non-numerical data.

ACM Reference Format:

Felix Biessmann, David Salinas, Sebastian Schelter, Philipp Schmidt, Dustin Lange. 2018. “Deep” Learning for Missing Value Imputation in Tables with Non-Numerical Data. In *The 27th ACM Int’l Conf. on Information and Knowledge Management (CIKM’18)*, Oct. 22–26, 2018, Torino, Italy. ACM, NY, NY, USA, 9 pages. <https://doi.org/10.1145/3269206.3272005>

1 INTRODUCTION

The success of many applications that ingest data critically depends on the quality of the data processed by those applications [26]. A

*these authors contributed equally.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM '18, October 22–26, 2018, Torino, Italy

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6014-2/18/10...\$15.00

<https://doi.org/10.1145/3269206.3272005>

central data quality problem is missing data. For instance in retail scenarios with a large product catalog, a product with an empty value for a product attribute is difficult to search for and is less likely to be included in product recommendations.

Many methods for missing data imputation were proposed in various application contexts: simple approaches such as mean or mode imputation as implemented in most APIs for data wrangling and Machine Learning (ML) pipelines (see footnote 1 for details with respect to the pandas and Spark libraries), matrix completion for recommendation systems [16] or supervised learning approaches for social science applications [28, 30]. Most of these imputation approaches, however, are either limited to small data sets or focus on imputation of numerical data from other numerical data. But in many real-world scenarios the data types are mixed and contain text. This kind of data is not easily amenable to imputation with existing methods or software packages, as discussed in more detail in Section 2. In these cases the gap between a data source, containing unstructured text data or categorical data, and a data sink, often requiring complete data, needs to be bridged by custom code to extract numerical features, feed the numerical values into an imputation method and transform imputed numerical values back into their non-numerical representation. Such custom code can be difficult to maintain and imposes technical debt on the engineering team in charge of a data pipeline [31].

Here we propose an imputation approach for tables with attributes containing non-numerical data, including unstructured text and categorical data. To reduce the amount of custom feature extraction glue code for making non-numerical data amenable to standard imputation methods, we designed a system that allows its users to combine and automatically select feature extractors for categorical and sequential non-numerical data, leveraging state of the art deep learning methods and efficient optimization tools. Our work extends existing imputation methods with respect to three aspects. First in contrast to existing simple and scalable imputation approaches such as mode imputation, the system achieves on average a 23-fold increase in imputation quality as measured by the F1-score. Second in contrast to more sophisticated approaches, such as *k-nearest-neighbor* based methods and other established approaches [10, 30], the proposed approach scales to large data sets as demonstrated in experiments on tables with millions of rows, which is up to four orders of magnitude more data than considered in aforementioned imputation studies. And third in contrast to other scalable data cleaning approaches, such as *HoloClean* [27] or *NADEEF* [9], the proposed approach can be easily automated and does not require human input. The motivation for using machine learning and a scalable, automatable implementation is to enable data engineers in charge of data pipelines to ensure completeness as well as correctness of a data source. Beyond this application

scenario, our system’s simple API (presented in Section 4) is also beneficial for other use cases: users without any machine learning experience or coding skills who want to run simple classification experiments on non-numerical data can leverage the system as long as they can export their data from an Excel sheet and specify the target column.

In order to demonstrate the scalability and performance of our approach, we present experiments on samples of a large product catalog and on public datasets extracted from Wikipedia. In the experiments on the product catalog, we impute missing product attribute values for a variety of product types and attributes. The sizes of the data sets sampled from the product catalog are between several 1,000 rows and several million rows, which is between one to four orders of magnitude larger than data sets in previous work on missing value imputation on numerical data [10]. We evaluate the imputations on product data with very different languages (English and Japanese), and find that our system is able to deal with different languages without any language-specific preprocessing, such as tokenization. In the Wikipedia experiments, we impute missing infobox attributes from the article abstracts for a number of infobox properties.

A sketch of the proposed approach is shown in Figure 1. We will use the running example of imputing missing color attributes for products in a retail catalog. Our system operates on a table with non-numerical data, where the column to be imputed is the color attribute and the columns considered as input data for the imputation are other product attribute columns such as product description. The proposed approach then trains a machine learning model for each to be imputed column that learns to predict the observed values of the to be imputed column from the remaining columns (or a subset thereof). Each input column of the system is fed into a featurizer component that processes sequential data (such as unstructured text) or categorical data. In the case of the color attribute, the to be imputed column is modeled as a categorical variable that is predicted from the concatenation of all featurizer outputs.

The reason we propose this imputation model is that in an extensive literature review we found that the topic of imputation for non-numerical data beyond rule-based systems was not covered very well. There exists a lot of work on imputation [30], and on modeling non-numerical data, but to the best of our knowledge there is no work on end-to-end systems that learn how to extract features and impute missing values on non-numerical data at scale. This paper aims at filling this gap by providing the following contributions:

- **Scalable deep learning for imputation.** We present an imputation approach that is based on state of the art deep learning models (Section 3).
- **High precision imputations.** In extensive experiments on public and private real-world datasets, we compare our imputation approach against standard imputation baselines and observe up to 100-fold improvements of imputation quality (Section 6).
- **Language-agnostic text feature extraction.** Our approach operates on the character level and can impute with high precision and recall independent of the language present in a data source (Section 5, Section 6).

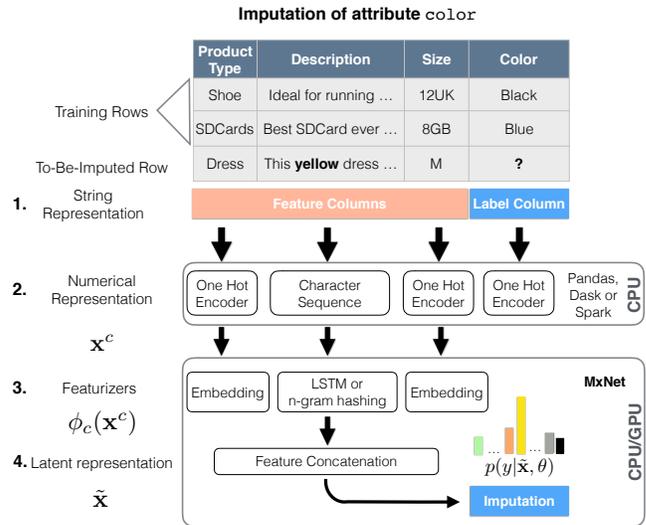


Figure 1: Imputation example on non-numerical data with deep learning; symbols explained in Section 3.

- **End-to-end optimization of imputation model.** Our system learns numerical feature representations automatically and is readily applicable as a plugin in data pipelines that require completeness for data sources (Section 3).

2 RELATED WORK

Missing data is a common problem in statistics and has become more important with the increasing availability of data and the popularity of data science. Methods for dealing with missing data can be divided into the following categories [21]:

- (1) Remove cases with incomplete data
- (2) Add dedicated missing value symbol
- (3) Impute missing values

Approach (1) is also known as complete-case analysis and is the simplest approach to implement – yet it has the decisive disadvantage of excluding a large part of the data. Rows of a table are often not complete, especially when dealing with heterogeneous data sources. Discarding an entire row of a table if just one column has a missing value would often discard a substantial part of the data.

Approach (2) is also simple to implement as it essentially only introduces a placeholder symbol for missing data. The resulting data is consumed by downstream ML models as if there were no missing values. This approach can be considered the de-facto standard in many machine learning pipelines and often achieves competitive results, as the missingness of data can also convey information. If the application is really only focused on the final output of a model and the goal is to make a pipeline survive missing data cases, then this approach is sensible.

Finally, approach (3) replaces missing values with substitute values (also known as imputation). In many application scenarios including the one considered in this work, users are interested in imputing these missing values. For instance, when browsing for a product, a customer might refine a search by specific queries (say

for size or color filters). Such functionalities require the missing values of the respective attributes to be imputed. The simplest imputation methods fill in the same value for all missing data cases for a given column of a table, similar to approach (2) outlined above. Examples are *mean imputation* (for continuous numerical data) or *mode imputation* (for non-numerical data), where the mean or mode of the respective attribute distribution is imputed. These imputation methods are implemented in most libraries and APIs that offer imputation methods¹. To the best of our knowledge, few software packages for data wrangling/pipelining go beyond this limited functionality and integrate with those well established tools. While these approaches suffice for replacing missing data in order to not make a data pipeline fail, they are not useful for actual imputation of missing data since their precision and recall levels are rather low as demonstrated by our experiments in Section 6.

For more sophisticated approaches to imputation, there is a substantial body of literature using both supervised as well as unsupervised methods [30]. One example from supervised learning is *Hot Deck Imputation*, which refers to the idea of using similar entries of a table as source for finding substitute values [2]. Replacement values can be found using *k*-nearest neighbors [3]. Another approach leveraging supervised learning techniques is *Multivariate imputation by chained equations* (MICE) [28]. The idea is to learn a supervised model on all but one column of a table and use that model to impute the missing values. This approach has been adopted in many other studies, see [10].

Another line of research focuses on unsupervised methods for imputation, such as matrix factorization [16, 22, 33]. Building on research for recommender systems, matrix factorization techniques were improved with respect to stability, speed and accuracy of imputations. However, not all use cases naturally lend themselves to a matrix completion model. For instance, when dealing with tables containing multiple columns with free text data, it is not obvious how to apply these methods. Text data needs to be vectorized before it can be cast into a matrix or tensor completion problem, which often discards valuable information, such as the order of tokens in a text. Another drawback of these methods is that they solve a more difficult problem than the one we are actually interested in: in many cases, we are merely interested in the imputation of a single cell, not the entire row of a table; learning a model that only tries to impute one column can be much faster and cheaper than learning a model for the entire table. The most important drawback of matrix factorization methods for the application scenario we consider is however that it is not straightforward to obtain imputations for new rows that were not present in the training data table. This is because matrix factorization methods approximate each matrix entry as an inner product of some latent vectors. These latent vectors are learned during training on the available training data. Hence, for rows that were not amongst the training data, there is no latent representation in the matrix factorization model. Computing such a latent vector for new rows of a table can be too costly at prediction time. One use case we are investigating in this work is that of product attribute imputation in a large product catalog. In such a scenario, an important requirement is to be able to ingest new data. New data should be as complete as possible, so we would want to

impute missing values already when new products enter the catalog. Ideally, we want to leverage information about products that are already in the catalog. While this use case – imputation for new data – can be tackled with matrix factorization type methods and there are a number of solutions in the *cold start recommender systems* literature, it is much simpler to implement with the approach presented here: as we do not learn a latent representation for each row index of a table and use only the content of observed values in the input columns for the imputations, our approach naturally lends itself to ingesting new data from rows that were not in the set of training data.

Another line of missing value imputation research in the database community is using rule based systems, such as *NADEEF* [9], which efficiently applies user-specified rules for the detection and repairing of quality constraint violations. Such rule based systems can achieve high precision for imputation, but this often requires a domain expert in the loop to generate and maintain the set of rules to apply. In our approach, we leverage machine learning to allow for automatic high precision imputations. Yet another line of research is the direction of active learning. Cleaning real world data sets requires ground truth data, which is most easily obtained from human annotators. If there is a human in the loop, active learning methods allow us to select and prioritize human effort [34]. *ActiveClean* uses active learning for prioritization and learns and updates a convex loss model at the same time [17]. *HoloClean* generates a probabilistic model over a dataset that combines integrity constraints and external data sources to generate data repair suggestions [27]. While such research is important if humans are in the loop, we focus on approaches that require as little human intervention as possible. The solution presented in this work is however easily extendable to batch based active learning scenarios, which we consider a future research direction. Finally, a recent line of work similar to our approach is [11], where the authors follow the idea of MICE [28], but propose to leverage deep learning to impute numerical values jointly (several columns at a time). Similar to the previously mentioned approaches, this work only considers numerical values on small data sets. In addition, the evaluation metric in this study is very different from ours, as the authors evaluate the error on a downstream classification or regression task, which renders comparisons with our results difficult.

To summarize, to the best of our knowledge, there are few machine learning based solutions focusing on scalable missing value imputation in tables with non-numerical data. A lot of the research in the field of imputation originates from the social sciences [28] or the life sciences [30], targeting tables with only dozens or hundreds of rows. Next to the scalability issues, most of the existing approaches assume that the data comes in matrix form and each column is numeric. In contrast to these approaches, our work focuses on imputation for large data with non-numerical data types, with an emphasis on extensibility to more heterogeneous data types.

3 IMPUTATION MODEL

In this section, we describe our proposed model for imputation. The overall goal of the model is to obtain a probability estimate of the likelihood of all potential values of an attribute or column, given an imputation model and information extracted from other columns.

¹ `DataFrame.fillna` in pandas/Python and `ml.feature.Imputer` in Spark/Scala

For illustration, we work with the example use case presented in Figure 1: given a product catalog where some product attributes, say color of a product, are missing for some products, we want to model the likelihood of all colors that could possibly be imputed. So in the example in Figure 1, we would like to estimate the likelihood for a product to have the color yellow given the information of all other columns for this row/product as well as the trained imputation model:

$$p(\text{color}=\text{yellow} \mid \text{other columns, imputation model}) \quad (1)$$

As the product description for this particular product contained the word 'yellow', we would expect the likelihood for this color value to be high. Generally, we would always predict the likelihood for all of the possible values that an attribute can take and then take the value that has the highest likelihood as the imputation for this value. In practice it can be useful to tune the model for each potential value to only make a prediction if the model achieves a certain precision or recall required by an application. All parameters and their optimization are explained in the following sections, but the high level overview over the approach can be subdivided into four separate stages also indicated in Figure 1:

- (1) **String representation:** In this stage, we separate the columns into input/feature and to-be-imputed/target columns. Data is still in their textual representation. All rows that have an observed value are considered for *training data* (and validation or testing). Rows with missing values are considered for imputation.
- (2) **Numerical representation:** In order to train machine learning methods for imputation, we need to first create a numerical representation of input and target columns. Depending on the type of data, we either model columns as categorical variables or sequential variables (such as free text fields).
- (3) **Feature representation:** The quality of predictions of machine learning models depends critically on the feature representation used. We build on a large body of work on embeddings for categorical and sequential data and use learnable feature representations.
- (4) **Imputation (Equation 3):** We finally compute the likelihood of all potential values from the concatenation of all extracted features (Equation 3).

In the following, we explain all of these stages in detail. Throughout the section, we use the index $c \in \{0, 1, 2, \dots, C\}$ to refer to input or feature columns/attributes, either as superscript for vectors (indicated by boldface font) or subscript for functions. We omit row indices to keep notation simple. When we mention input data/features or target variables, we always refer to a single row without a row index.

Numerical encoding. In order to make the data amenable to machine learning models, the first step in the model is to transform the string data of each column c for each row into a numerical representation \mathbf{x}^c . We use different encoders for different non-numerical data types and distinguish between *categorical* and *sequential* data. For categorical data, the numerical representation $x^c \in \{1, 2, \dots, M_c\}$ is the index of the value in the histogram of size M_c computed on column c ; note that we include an additional

missing symbol, hence there are $M_c + 1$ values that x^c can take. For notational simplicity, these scalar variables will be denoted as vector \mathbf{x}^c in the following. We chose to base the indexing on histograms in order to retain the information on the symbol frequency and in order to be able to discard too infrequent symbols more easily. For sequential data, the numerical representation $\mathbf{x}^c \in \{0, 1, 2, \dots, A_c\}^{S_c}$ is a vector of length S_c , where S_c denotes the length of the sequence or string in column c and A_c denotes the size of the set of all characters observed in column c . Also here we include an additional missing symbol that increases the number of possible symbols to $A_c + 1$. The data types are determined using heuristics. In the data sets used in the experiments, the data types of the columns are easy to separate into free text fields (product description, bullet points, item name) and categorical variables (e.g. color, brand, size, . . .). If the data types are not known upfront, heuristics based on the distribution of values in a column can be used for type detection.

Feature extraction. Machine learning models often produce inaccurate predictions if the feature representation is not optimized – and on the other hand, very simple machine learning models can perform surprisingly well when the appropriate features are extracted from the data prior to model training and prediction. Here we employ state-of-the-art methods from deep learning as well as simple, but highly performant established feature extractors to derive useful features from the numerical representation of the data. Once the non-numerical data is encoded into their respective numerical representation, a column-specific feature extraction mapping $\phi_c(\mathbf{x}^c) \in \mathbb{R}^{D_c}$ is computed, where D_c denotes the dimensionality for a latent variable associated with column c . We consider three different types of featurizers $\phi_c(\cdot)$:

- Categorical variables:
 - One-hot encoded embeddings
- Sequential variables:
 - Hashed character n-grams
 - Long short-term memory neural networks

For one-hot encoded categorical data we define a featurizer as an embedding layer (as in word embeddings [24] or matrix factorization [16]) that is fed into a single fully connected layer. The hyperparameter for this featurizer is used to set both the embedding dimensionality as well as the number of hidden units of the output layer. For columns c with sequential string data, we consider two different possibilities for $\phi_c(\mathbf{x}^c)$: an n-gram representation or a character-based embedding using a long short-term memory (LSTM) recurrent neural network [13]. For the character n-gram representation, $\phi_c(\mathbf{x}^c)$ is a hashing function that maps each n-gram, with $n \in \{1, \dots, 5\}$, in the character sequence \mathbf{x}^c to a D_c dimensional vector; here D_c denotes here the number of hash buckets. In the LSTM case, we featurize \mathbf{x}^c by iterating an LSTM through the sequence of characters of \mathbf{x}^c that are each represented as a continuous vector via a character embedding. The sequence of characters \mathbf{x}^c is then mapped to a sequence of states $\mathbf{h}^{(c,1)}, \dots, \mathbf{h}^{(c,S_c)}$; we take the last state $\mathbf{h}^{(c,S_c)}$, mapped through a fully connected layer as the featurization of \mathbf{x}^c . The hyperparameters of each LSTM featurizer include the number of layers, the number of hidden units of the LSTM cell, the dimension of the character embedding c , and

the number of hidden units of the final fully connected output layer of the LSTM featurizer. Note that the hashing featurizer is a stateless component that does not require any training, whereas the other two types of feature maps contain parameters that are learned using backpropagation in an end-to-end fashion along with all other model parameters. Finally, all feature vectors $\phi_c(\mathbf{x}^c)$ are concatenated into one feature vector

$$\tilde{\mathbf{x}} = [\phi_1(\mathbf{x}^1), \phi_2(\mathbf{x}^2), \dots, \phi_C(\mathbf{x}^C)] \in \mathbb{R}^D \quad (2)$$

where $D = \sum D_c$ is the sum over all latent dimensions D_c . As is common in the machine learning literature, we refer to the numerical representation of the values in the to-be-imputed target column as $y \in \{1, 2, \dots, D_y\}$.

Imputation model. After extracting the features $\tilde{\mathbf{x}}$ of input columns and the observed values y of the to be imputed column we cast the imputation problem as a supervised learning problem by learning to predict the label distribution of y from $\tilde{\mathbf{x}}$. Our imputation approach models $p(y|\tilde{\mathbf{x}}, \boldsymbol{\theta})$, the D_y -dimensional probability vector over all possible values in the to be imputed column conditioned on some learned model parameters $\boldsymbol{\theta}$ and an input vector $\tilde{\mathbf{x}}$ (containing information from other columns) with a standard logistic regression type output layer

$$p(y|\tilde{\mathbf{x}}, \boldsymbol{\theta}) = \text{softmax}[\mathbf{W}\tilde{\mathbf{x}} + \mathbf{b}] \quad (3)$$

where the learned parameters $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{z}, \mathbf{b})$ include the learned parameters of the output layer (\mathbf{W}, \mathbf{b}) and \mathbf{z} , comprising all parameters of the learned column featurizers ϕ_c . Finally, $\text{softmax}(\mathbf{q})$ denotes the element-wise softmax function $\frac{\exp q_j}{\sum_j \exp q_j}$ where q_j is the j -th element of a vector \mathbf{q} . The parameters $\boldsymbol{\theta}$ are learned by minimizing the cross-entropy loss between the predicted distribution and the observed labels y by computing

$$\boldsymbol{\theta} = \arg \min_{\boldsymbol{\theta}} \sum_1^N -\log(p(y|\tilde{\mathbf{x}}, \boldsymbol{\theta}))^\top \text{onehot}(y) \quad (4)$$

where \log denotes element-wise logarithm and the sum runs over N rows for which a value was observed in the target column corresponding to y . We use $\text{onehot}(y) \in \{0, 1\}^{D_y}$ to denote a one-hot encoding of the label y , which is a vector of zeros and a single one in the entry k corresponding to the class index encoded by y . We apply standard backpropagation and stochastic gradient descent (SGD) [6] in order to optimize all parameters, including those of the featurization, in an end-to-end fashion. Training the model with SGD is very memory efficient, as it requires us to only store one mini-batch of data at a time in memory, which typically consists of a few hundred rows of a table. The approach thus easily scales to tables with millions of rows.

4 IMPLEMENTATION AND API

Building a real world machine learning application such as an end-to-end imputation system poses not only algorithmic challenges, but also requires careful thinking about the system design. The goal of our work is to free users of our system from the need of feature engineering. We use machine learning not only to learn the imputation model, but also to learn an optimal feature extraction.

An important advantage of a system that automatically tunes its parameters is that we can keep its interface simple and enable practitioners without an ML background to use it. The API we designed allows to impute missing values by just passing a table as a pandas DataFrame to the imputation model and specifying the to be imputed column and input columns, as shown in the Python code in Listing 1. All (hyper-)parameters are derived from the data and learned automatically. For data type detection, we use heuristics; for the differentiable loss functions of the entire imputation model, we use backpropagation and stochastic gradient descent; and for hyperparameter optimization on non-differentiable loss functions (as for instance the model architecture parameters such as number of hidden units of an LSTM), we apply grid search (alternatives are random search or bayesian global optimization techniques).

```
# load training and test tables
table = pandas.read_csv('products.csv')
missing = table[table['color'].isnull()]

# instantiate and train imputer
model = Imputer(
    data_columns=['description', 'product_type', 'brand'],
    label_columns=['color'])
    .fit(table)

# impute missing values
imputed = model.transform(missing)
```

Listing 1: Example of Python imputation API.

We perform all encoding steps using custom Python code and standard libraries; for representing the table data we apply pandas, for the hashing vectorizer on character n-grams, we leverage the HashingVectorizer of scikit-learn [25]. We implement the modeling steps performed after the numerical encoding of the data in Apache MXNet[7]. The featurization (except for the character n-gram representation, which is passed to the network as a sparse vector), is set up using the Symbolic Python API. We employ the standard GPU-optimized version of MXNet for the LSTM-based featurizations.

5 EXPERIMENTS

We ran experiments on a large sample of a product catalog and on public Wikipedia datasets. For both datasets, we impute a variety of different attributes. As the number of valid attribute values present in a given to be imputed column of the training data has a strong impact on the task difficulty, we applied filters to make sure that the results are comparable across product types and attributes. We included only attribute values that were observed at least 100 times (for a product type) or at least once (in the smaller Wikipedia data set) and considered only the 100 most frequent attribute values.

Product attributes. We used samples of a large product catalog in different languages to demonstrate the ability of our system to reliably impute data in non-numerical tables independent of the language of the text in a table. In our experiments, we trained models for imputing a set of attributes for product types listed in Table 2 for English and Japanese product data. Note that these two languages have very different alphabets and usually require language-specific preprocessing. An example of such a language-specific step would be tokenization, which can be difficult for some languages, including Japanese. We did not apply any language-specific

preprocessing in our experiments and used the same imputation models and parameter sets for both languages. For each product type, we extracted all products that matched the language and the product type. As input columns, we used both columns containing unstructured text data (title, product description, bullet points) as well as columns containing categorical variables (e.g., brand, manufacturer, size, display technology). The cardinality of the character set for sequential data was set to 100 (for English) and to 1000 (for Japanese); 1000 characters covered most Japanese letters and some Chinese symbols. The number of rows in the tables sampled for these experiments was between 10,000 and 5,000,000.

Wikipedia. In addition to the product catalog data sample, we extracted attributes found in the infoboxes of Wikipedia articles. The data is publicly available as part of the DBpedia project. For our experiments, we have used the 2016-10 version². DBpedia provides the extracted Wikipedia graph in the turtle format, where each row consists of triplets to describe subject, predicate, object. We have mapped the textual (long abstracts) descriptions of subjects to their corresponding infobox objects for birth_place, genre and location. These predicates are most commonly found in the infoboxes of Wikipedia articles. In many cases, each subject may be related to several genre objects, e.g., by relating a band to multiple genres. In order to transform the DBpedia data into a multi-class dataset, where each training instance has exactly one label associated to it, we have excluded all of the Wikipedia articles with multiple objects per subject. The number of rows in the resulting tables were 129,729 for location, 333,106 for birthplace and 170,500 for genre.

Experimental settings. After extracting the string data tables, we selected featurizers for each column depending on the data type, as described in Section 3. In our experiments, we used one LSTM per free text column in the case of LSTM featurizers; in the case of the n-gram featurizer we concatenated and hashed the texts of all columns into one feature vector using the same hashing function. The LSTM hyperparameters were kept the same for the featurizers of all columns. For all sequential features, we applied a sequence length of 300 based on a heuristic using the length histograms of representative data. Both types of sequential featurizers were combined with the categorical embedding featurizers for all categorical columns in the data set, excluding the to be imputed column.

We ran grid search for hyperparameter optimization. Next to the model hyperparameters described in Section 3 we also optimized an L_2 norm regularizer using weight decay. An overview of the hyperparameters optimized can be found in Table 1. For all experiments, we split the available data into a 80%, 10%, 10% split for training, validation and test data, respectively. All metrics reported are computed on test data which was not used for training or validation. All experiments were run on a single GPU instance (1 GPU with 12GB VRAM, 4 vCPUs with 60GB RAM)³. Training was performed with a batch size of 128 and Adam SGD [15] for a maximum of 50 epochs and early stopping if the loss does not improve for 3 consecutive epochs. The two baseline approaches were performed in a Spark

shell running Scala/Spark on a single host (36 vCPU, 60 GB RAM). The reason the experiments were performed on different hardware is that for the Spark experiments we did not leverage GPUs.

Hyperparameter	Range	Best value(s)
LSTM layers	[2,4]	2
LSTM hidden units	[10, 150]	{100, 120}
Dimensionality of LSTM output	[10, 150]	100
Dimensionality of LSTM character embedding	[10, 100]	50
Dimensionality of hashing vectorizer output	[2 ¹⁰ , 2 ²⁰]	{2 ¹⁰ , 2 ¹⁵ , 2 ¹⁸ }
Dimensionality of embeddings for categorical variables	[10, 50]	10
SGD learning rate	[10 ⁻⁵ , 10 ⁻¹]	{0.001, 0.008}
Weight Decay/ L_2 Regularization	[0, 10 ⁻²]	{0.0001, 0}

Table 1: Ranges and optimal (for a given model/data set) hyperparameters for model selection.

Baseline methods. For comparison, we added two baseline methods. The first baseline is a simple *mode* imputation, which always predicts the most frequent value of a column. The second baseline is a rule-based *string matching* approach that predicts the label that had most string matches in the input columns, similar to rule-based imputation engines, such as the approach presented in [9].

6 RESULTS

We performed extensive evaluations on Wikipedia data sets and on product attribute data for several product types and attributes sampled from a large product catalog. Methods are compared with respect to imputation quality, as measured by F1 scores weighted by class frequency, as well as with respect to their operational cost.

Product attribute results. Results for the imputation tasks for a number of product types and various product attributes are listed in Table 2. Our proposed approach reaches a median F1 score of 92.8% when using LSTM-based featurizers and a median F1 score of 93% for a linear model with an n-gram featurizer. Both clearly outperform the baselines mode imputation (median F1 4.1%) and string matching of the label to the free form text columns (median F1 30.1%). We argue that in the case we are considering, mode imputation can be considered the de-facto standard for imputation. For one it is implemented in popular libraries for data pipelines (footnote 1), hence it is the most accessible option for data engineers working in these frameworks. Second, while there are a number of open source packages for imputation in Python (e.g. MIDAS, fancyimpute) and R (MICE), none of those packages address the use case we are considering; all of those existing packages work on matrices containing only numeric data. In contrast we are considering unstructured data like text as additional input. This use case is not accounted for in existing packages, to the best of our knowledge. Compared to mode imputation, we see an up to 100-fold improvement in imputation F1 score (median 23-fold improvement) with our proposed approach. The string matching method gives an F1 score close to the best performing models only in rare cases where the attribute value is usually included in the article name, such as for the brand of shoes.

²<http://wiki.dbpedia.org/downloads-2016-10>

³A single virtual CPU or vCPU on the AWS EC2 cloud service is a single hyperthread and approximately equivalent to half a physical CPU.

Dataset	Attribute	Mode	String matching	LSTM	N-gram
dress, EN	brand	0.4%	80.2%	99.9%	99.8%
	manufacturer	1.0%	22.6%	99.0%	99.6%
	size	3.7%	0.1%	77.4%	74.4%
monitor, EN	brand	12.4%	41.6%	93.5%	88.4%
	display	27.6%	12.2%	90.0%	90.2%
	manufacturer	13.8%	30.6%	91.2%	86.9%
notebook, EN	brand	3.8%	47.9%	98.7%	97.8%
	cpu	80.0%	85.1%	95.6%	96.7%
	manufacturer	4.0%	33.4%	92.8%	93.0%
shoes, EN	brand	0.5%	91.4%	99.8%	99.9%
	manufacturer	0.5%	77.9%	97.1%	98.3%
	size	1.2%	0.0%	54.8%	45.3%
	toe style	12.1%	21.7%	89.1%	92.3%
shoes, JP	brand	2.6%	19.2%	98.4%	99.6%
	color	16.8%	48.1%	78.0%	82.5%
	size	51.1%	1.7%	66.6%	66.1%
	style	57.6%	12.6%	87.0%	94.0%
Median		4.1%	30.1%	92.8%	93.0%

Table 2: F1 scores on held-out data for imputation task *product attributes* for mode imputation, string matching, LSTM and character n-gram featurizers. For each attribute, between 10,000 and 5,000,000 products were sampled. Independent of the featurizers used, LSTMs or n-grams, our imputation approach outperforms both baselines in terms of F1 score, achieving on average a 23-fold increase (compared to mode imputation) and a 3-fold increase (compared to string matching).

Wikipedia results. Also when imputing infobox properties from Wikipedia abstracts, we see that both machine learning based imputation methods, LSTMs and character n-gram models, significantly outperform the baseline approaches Table 3. On average we observe an almost 100-fold improvement in F1 score when comparing a simple n-gram model to mode imputation. It is worth noting that despite the strong improvements with the proposed machine learning imputation approach, the F1 scores of the imputations are only reaching up to 72% for the Wikipedia data. After inspecting the data we attribute this to label noise. For instance the confusion matrix of true and imputed values shows that in the imputation task for birth place, when the true value is 'us', the most frequent imputed values are 'us', 'california' and 'florida' or when the true value is 'wales', the most frequent imputed values are 'wales', 'uk' and 'england'. So there are many 'misclassifications' that are due to ambiguity related to the political taxonomy of locations in the training data, which we cannot expect the imputation model to correct for.

N-Gram models vs. LSTM. In many experiments, we achieve high scores with both the deep learning LSTM model and n-gram methods. The linear n-gram model often achieves competitive results: only in six out of 20 cases, the LSTM clearly performed better than the linear model. One reason for this could be that most of the tasks are too simple for an LSTM to achieve much higher performance. We assume that the advantage of the LSTM will become clearer with more difficult imputation problems. However our results are in line with some recent work that finds simple n-gram models to

Dataset	Attribute	Mode	String matching	LSTM	N-gram
Wikipedia, English	birth place	0.3%	16.3%	54.1%	60.2%
	genre	1.5%	6.4%	43.2%	72.4%
	location	0.7%	7.5%	41.8%	60.0%
Median		0.7%	7.5%	43.2%	60.2%

Table 3: F1 scores on held-out data for imputation task *Wikipedia*. See Table 2 for a description of columns.

compare favourably with more sophisticated neural network architectures [12, 14]. The considerably faster training for the linear model is an important factor for production settings. We therefore compare the operational cost and training speed in the following section.

Operational cost comparison. One application scenario of the proposed method is automatic imputations in data pipelines to ensure completeness of data sources. In this setting, operational cost imposed by the memory footprint of a model and the training time can be an important factor. We compare the models used in our experiments with respect to these factors. The size and training speed of the models depends on the model selection process; we measured model size in MB and sample throughput in samples per seconds during training for the models with the highest validation score. The model size in MB for n-gram models is 0.4/13.1/104.9 MB (5th/50th/95th percentile) and the model size for LSTM based imputation models is 18.6/37.9/45.7 MB. Depending on the best performing hyperparameter setting for a given data set, there are some n-gram models that are much smaller or much larger than the average LSTM models, but the median model size of n-gram models is about three times smaller than that of LSTM models. Sample throughput during training was one to two orders of magnitude larger for imputation models using only character n-gram features (1,079/11,648/77,488 samples per second) compared to deep learning based LSTM featurizers (107/290/994 samples per second). Assuming a sample throughput of 11,000 samples per second, one training pass through a table with 1,000,000 rows takes less than 90 seconds. For a data set of this size, typically less than 10 passes through the data are needed for training to converge.

7 LESSONS LEARNED

When we set out to create a system for imputation on non-numerical data, we faced the question of choosing appropriate algorithms and execution platforms. Many of our initial decisions turned out to be suboptimal in one way or another. In this section, we describe some of the learnings we made along the way.

Choice of imputation method and feature extractors. The first challenge was to decide which imputation method to use and which featurization method should precede the imputation method. As highlighted in Section 2, there seems to be a gap between the methods used in practice by data engineers and machine learning practitioners (simple approaches such as mode imputation) and the mathematically more sophisticated matrix factorization approaches, which are less often encountered in practice. There are

several reasons why we decided not to follow the research on matrix factorization for imputation and opted for the approach presented in this work. For one, the current approach is much simpler to implement, to extend, and to adapt to new scenarios and data types. For example, for image data, we can add an off-the-shelf pretrained neural network [18] and fine tune it along with all other parameters of the imputation model. Secondly, the approach presented here is much cheaper to train. Matrix factorization models obtain an advantage over other methods by modeling latent relationships between all observed aspects of the data. In our approach, however, we only learn to impute one column at a time, which can be more efficient than modeling the entire table.

Another challenge we faced was the question of how to model non-numerical data. We ran extensive studies on different types of feature extractors, including linguistic knowledge, word embeddings, and also other types of sequential feature extractors, such as convolutional neural networks [20]. The main conclusion from those experiments are the same as we draw from the experiments presented in this work: in practice many popular deep learning approaches did not outperform rather simple feature extractors. The systematic comparison in this study demonstrates that sparse linear models with hashed n-gram character features achieve state of the art results on some tasks when compared directly to deep learning methods, similar to the findings in [14]. Such models are much faster during training and prediction, work well on CPUs, and require less memory. Yet, we emphasize that this could be related to the data sets we tested on. We hypothesize that in more complicated settings, LSTMs are more likely to produce better results than linear n-gram models.

System-specific challenges. As of today, there are no off-the-shelf solutions available for complex end-to-end machine learning deployments, and many data management related questions from the ML space are only beginning to raise the attention of the database community [19, 26, 31]. In practice, a wide variety of systems is applied for large-scale ML, with different advantages and drawbacks. These systems range from general-purpose distributed dataflow systems such as Apache Spark [35], which support complex preprocessing operations, but are difficult to program for ML practitioners with a background in statistics or mathematics, to specialized deep learning engines such as Apache MXNet [8] or Google's Tensorflow [1], which provide mathematical operators optimized for different hardware platforms but lack support for relational operations.

We started with an imputation approach built on a distributed dataflow system, in particular the *SparkML* [23] API. We designed an API on top of DataFrames, which allowed us to quickly build and try different featurizer and imputation model combinations. Spark turned out to be very helpful for large-scale data wrangling and preprocessing workloads consisting of relational operations mixed with UDFs, but is in our experience very difficult to use for complex machine learning tasks [5]. The complexity of the data (matrices and vectors) and the operations to apply forced us to base our implementations on the low-level RDD-API, rather than the SparkSQL API, which would provide automatic query optimization. Programs on the RDD level represent hardcoded physical execution plans that are usually tailored to run robustly in production setups,

and therefore naturally result in huge overheads when run on smaller data. Additionally, difficult choices about the materialization of intermediate results are entirely left to the user [29].

Next, we leveraged a recently developed deep learning system [8] for the imputation problem, which allow us to quickly design models and optimize them efficiently (even for non-neural network models). This is due to dedicated mathematical operators, support for automatic differentiation, and out-of-the-box efficient model training with standard optimization algorithms on a variety of hardware platforms. A major obstacle in leveraging deep learning frameworks is the integration with Spark-based preprocessing pipelines. Deep learning toolboxes are typically used through their Python bindings, and while Python offers a great ecosystem for data analysis, we mostly aim to run preprocessing and feature extraction workloads on the type safe and stable Spark/JVM platform. In order to keep the best of both worlds, we used a hybrid system that extracted features in Spark and communicated with MXNet via a custom disk-based serialization format. In practice, this system turned out to be difficult to use, as debugging required us often to dig through several layers of stack traces [31]. We had to set up two runtimes and tune their configurations (which can be especially difficult for Spark's memory settings). Furthermore, experimentation was not simple, as the feature extraction step was rather involved and the required materialization of preprocessed features made it tedious to quickly try out different features or data sets. Finally, it is challenging to efficiently schedule the resulting workloads of such hybrid systems, as the results from the Spark-based preprocessing jobs, executed on clusters of commodity machines, need to be transferred to specialized GPU instances for the training of deep learning models.

8 CONCLUSION

We have presented an approach to missing value imputation for tables containing non-numerical attributes using deep learning. The goal was to bridge the gap between existing imputation methods that are primarily targeted at imputation of numerical data and application scenarios where data comes in non-numerical tables. Our approach defines a simple imputation API over tables with non-numerical attributes that expects only the names of to-be-imputed columns and the names of the columns used for imputation. Automatic hyperparameter optimization is used to determine the optimal combination of featurizer modules. If the data and its schema are not known, heuristics can be used to use a custom architecture for featurizing non-numerical content. The presented system allows researchers and data engineers to plug in an imputation component into a data pipeline to ensure completeness of a data source using state-of-the-art machine learning models. In extensive experiments on product data from a sample of a large product catalog as well as a number of data sets obtained from Wikipedia, we have shown that the approach efficiently and reliably imputes missing product attributes in tables with millions of rows. Model training time for a table of about one million rows and up to ten input columns is usually between a few minutes for a simple model configuration, such as a sparse linear character n-gram model, and around an hour for the most complex models, on a single GPU instance. Experiments on product data in English and Japanese demonstrate that

our character based approach is language-agnostic and can be used for imputation of tables that contain very different languages.

In our experiments, we found that while deep learning methods perform very well, a simple character n-gram feature extraction often achieves competitive results. This could be due to the fact that the task was too easy. On the compute instances on which we ran experiments, simple linear character n-gram models achieved throughputs of several ten thousand samples per second during training, whereas models with the LSTM-based featurizers usually only could process several hundreds of rows of a table during learning. Depending on the task and the best hyperparameter configuration, LSTM based models can be smaller in size compared to the n-gram models. On average however n-gram models are not only faster to train but also smaller in size. This finding confirms other work that highlights the potential of relatively simple n-gram models especially when compared to more expensive to train neural network architectures [12, 14] We note that while the current setting was restricted to imputation of categorical values, it can be extended straightforwardly by adding a standard numerical regression loss function. Another extension is to consider imputing several columns at the same time, potentially of different types. This can be easily adapted by summing the column specific losses. We did run experiments with such multi-task loss functions but we found that single output models perform best when evaluated on single columns only. Finally, we highlight that many existing imputation approaches are based on matrix factorization and learn a latent representations associated with each row index of a table. This makes it more difficult to impute values for new rows of a table, a use case that is relevant when new rows need to be appended to a table. Our approach was designed to allow for simple and efficient insertions of new rows while preserving the ability to compute a latent numerical representation for each row which could be used for other purposes, such as information retrieval, recommendation systems, or nearest neighbor search in large tables containing non-numerical data [4].

The code used in this study is available as open source package <https://github.com/awslabs/datawig>.

REFERENCES

- [1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.
- [2] R. R. Andridge and R. J. Little. A review of hot deck imputation for survey non-response. *International statistical review*, 78(1):40–64, 2010.
- [3] G. Batista and M. C. Monard. An analysis of four missing data treatment methods for supervised learning. *Applied Artificial Intelligence*, 17(5-6):519–533, 2003.
- [4] R. Bordawekar and O. Shmueli. Using word embedding to enable semantic queries in relational databases. In *Workshop on Data Management for End-to-End Machine Learning at Sigmod*, page 5, 2017.
- [5] J.-H. Böse, V. Flunkert, J. Gasthaus, T. Januschowski, D. Lange, D. Salinas, S. Schelter, M. Seeger, and Y. Wang. Probabilistic demand forecasting at scale. *PVLDB*, 10(12):1694–1705, 2017.
- [6] L. Bottou. On-line learning in neural networks. chapter On-line Learning and Stochastic Approximations, pages 9–42. Cambridge University Press, New York, NY, USA, 1998.
- [7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.
- [8] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *Machine Learning Systems workshop at NIPS*, 2015.
- [9] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 541–552. ACM, 2013.
- [10] P. J. García-Laencina, J.-L. Sancho-Gómez, and A. R. Figueiras-Vidal. Pattern classification with missing data: a review. *Neural Computing and Applications*, 19(2):263–282, 2010.
- [11] L. Gondara and K. Wang. Multiple imputation using deep denoising autoencoders. *CoRR*, abs/1705.02737, 2017.
- [12] E. Grave, T. Mikolov, A. Joulin, and P. Bojanowski. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017, Valencia, Spain, April 3-7, 2017, Volume 2: Short Papers*, pages 427–431, 2017.
- [13] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.
- [14] A. Joulin, E. Grave, P. Bojanowski, M. Nickel, and T. Mikolov. Fast Linear Model for Knowledge Graph Embeddings. *arXiv:1710.10881v1*, 2017.
- [15] D. Kingma and J. Ba. Adam: A method for stochastic optimization. Technical report, preprint arXiv:1412.6980, 2014.
- [16] Y. Koren, R. M. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.
- [17] S. Krishnan, M. J. Franklin, K. Goldberg, J. Wang, and E. Wu. Activeclean: An interactive data cleaning framework for modern machine learning. In *SIGMOD'16*, pages 2117–2120, 2016.
- [18] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [19] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Record*, 2015.
- [20] Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 336(10):1995, 1995.
- [21] R. J. A. Little and D. B. Rubin. *Statistical Analysis with Missing Data*. John Wiley & Sons, Inc., New York, NY, USA, 1986.
- [22] R. Mazumder, T. Hastie, and R. Tibshirani. Spectral regularization algorithms for learning large incomplete matrices. *Journal of Machine Learning Research*, 11:2287–2322, 2010.
- [23] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(1):1235–1241, 2016.
- [24] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.
- [25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [26] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data management challenges in production machine learning. In *SIGMOD'17*, pages 1723–1726, 2017.
- [27] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.
- [28] D. Rubin. Multiple imputation for nonresponse in surveys. *Bioinformatics*, 17(6):520–525, 1987.
- [29] S. Schelter, A. Palumbo, S. Quinn, S. Marthi, and A. Musselman. Samsara: Declarative Machine Learning on Distributed Dataflow Systems. In *Machine Learning Systems Workshop at NIPS'16*.
- [30] G. M. Schmitt P, Mandel J. A comparison of six methods for missing data imputation. *J Biom Biostat*, 6(224), 2015.
- [31] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, and D. Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, 2015.
- [32] A. P. Singh and G. J. Gordon. A unified view of matrix factorization models. In *ECML/PKDD*, pages 358–373, 2008.
- [33] O. G. Troyanskaya, M. N. Cantor, G. Sherlock, P. O. Brown, T. Hastie, R. Tibshirani, D. Botstein, and R. B. Altman. Missing value estimation methods for DNA microarrays. *Bioinformatics*, 17(6):520–525, 2001.
- [34] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, Feb. 2011.
- [35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.