2024 IEEE International Conference on Big Data (BigData) | 979-8-3503-6248-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/BigData62323.2024.10825934

Automating Data Science Pipelines with Tensor Completion

Shaan Pakala
Dept. of CSE
UC Riverside
spaka002@ucr.edu

Bryce Graw
Dept. of CIS
San Diego Mesa Community College
bgraw@student.sdccd.edu

Dawon Ahn
Dept. of CSE
UC Riverside
dahn017@ucr.edu

Tam Dinh
Dept. of CS
Cal Poly Pomona
tamdinh@cpp.edu

Mehnaz Tabassum Mahin
Dept. of CSE
UC Riverside
mehnaztabassum.mahin@email.ucr.edu

Vassilis Tsotras
Dept. of CSE
UC Riverside
tsotras@cs.ucr.edu

Jia Chen
Dept. of ECE
UC Riverside
jiac@ucr.edu

Evangelos E. Papalexakis
Dept. of CSE
UC Riverside
epapalex@cs.ucr.edu

***Abstract*—Hyperparameter optimization is an essential component in many data science pipelines and typically entails exhaustive time and resource-consuming computations in order to explore the combinatorial search space. Similar to this problem, other key operations in data science pipelines exhibit the exact same properties. Important examples are: neural architecture search, where the goal is to identify the best design choices for a neural network, and query cardinality estimation, where given different predicate values for a SQL query the goal is to estimate the size of the output. In this paper, we abstract away those essential components of data science pipelines and we model them as instances of tensor completion, where each variable of the search space corresponds to one mode of the tensor. Now the goal is to identify all missing entries of the tensor, corresponding to all combinations of variable values, starting from a very small sample of observed entries. In order to do so, we first conduct a thorough experimental evaluation of existing state-of-the-art tensor completion techniques. We also introduce domain-inspired adaptations (such as smoothness across the discretized variable space) and an ensemble technique which is able to achieve state-of-the-art performance. We extensively evaluate existing and proposed methods in a number of generated datasets corresponding to (a) hyperparameter optimization for non-neural network models, (b) neural architecture search, and (c) variants of query cardinality estimation. By doing this, we demonstrate the effectiveness of tensor completion as a tool for automating data science pipelines. Furthermore, we release our generated datasets and code in order to provide benchmarks for future work on this topic.**

1. Introduction

In many different data science pipelines, it is necessary to automate the pipeline's design, such as choosing the optimal hyperparameters for a machine learning model, searching for optimal architecture for a deep neural network, and predicting the (distinct) output cardinality of all predicate values for SQL queries. Unfortunately, this usually requires exhaustive computation of all candidate design combinations, resulting in expensive computing time and resources, exponential with respect to the number of search components (e.g. hyperparameters or SQL query predicates).

Finding the optimal hyperparameter configuration of a machine learning model is commonly known as hyperparameter optimization. Standard approaches include grid search and random search [1]. Grid search (such as GridSearchCV [2]) exhaustively trains and evaluates a machine learning model's performance using a grid of hyperparameter combinations. The grid represents the hyperparameters on each axis, and the different axis values are the predefined values that hyperparameter takes. These predefined hyperparameter combinations are all trained and evaluated on a downstream task [3]. As the number of hyperparameters grow, the number of combinations of values they take grow exponentially, making an exhaustive search very difficult. An exhaustive search becomes impractical with even more intricate deep learning model architectures as well. Another simple approach, random search, evaluates a model on the hyperparameters that are drawn independently from a predefined distribution, such as uniform distribution. However, these algorithms often do not converge to the optimal configuration [1].

In the domain of hyperparameter optimization, there have been efforts to address the computational challenges of the standard methods posed by manual programming and testing. Bayesian optimization [4] uses probabilistic models such as random forests, Gaussian processes, and gradient boosting to decide which data sample to be fed for the evaluation in each iteration [5]–[7]. Stochastic population-based optimization methods, evolution strategies, iteratively find hyperparameter configurations with high fitness values, i.e., the (inverted) generalization error [8]–[10]. Other advanced frameworks include Hyperband [11], iterated racing [12], and Gradient-based optimization methods [13], [14]. The aforementioned methods have been shown promising results in searching for good hyperparameter configurations,

979-8-3503-6248-0/24/$31.00 ©2024 IEEE

but none of them are designed to explicitly take advantage of the underlying low-rank structure. In practice, there is often a small subset of the whole set of hyperparameters that influence the downstream task performance, and several values of the same hyperparameter, e.g., learning rate, are likely to yield similar or the same performance [15], [16]. There exists work on approaching hyperparameter optimization problem as a tensor completion task which assumes the tensor being low rank [17]–[19].

However, the existing methods are limited to suggesting a single optimal hyperparameter configuration, depending on strong underlying statistical assumptions, or being constrained to low-rank structure. For data science pipeline design, one may be interested in multiple optimal hyperparameter configurations or knowing the outcome of every configuration, or if one cannot make any statistical or structural assumptions. In databases, users tend to care about the cardinality of every query. To this end, we offer a generic model for automating data science pipelines through tensor completion. We model the outputs of the data science pipeline design (e.g., cardinality of SQL query searches and machine learning model evaluation scores) as the entries of a tensor, and each search variable (e.g., hyperparameter or a query predicate value) as one mode of the tensor. Given a very small fraction of observed entries, our goal is to estimate all missing entries of the tensor.

In this paper, we thoroughly explore a broad spectrum of the existing state-of-the-art (SOTA) tensor completion techniques including CPD, TuckER [20], CoSTCo [21], and NeAT [22], and apply them to automating data science pipelines. Leveraging the advances of CPD, we also propose a new tensor completion method, namely CPD-S, by enforcing the smoothness constraints on the latent CPD components, since any two similar values of the same search variable most likely lead to similar design output. Furthermore, to fully take advantage of the individual SOTA models, we propose ensemble methods. Specifically, we consider: 1) straight-forward schemes such as taking the mean or median of several predictions corresponding to the same tensor entry from the same model (CPD-S or CoSTCo) with different pre-defined tensor ranks, and 2) more advanced framework, i.e., learning the contributions of the results from multiple CPD-S or CoSTCo through a neural network such as multi-layer perceptron and aggregating the results nonlinearly to form a single prediction result for every design configuration.

In this work, we make several distinct contributions in sparse tensor completion for surrogate modeling as follows:

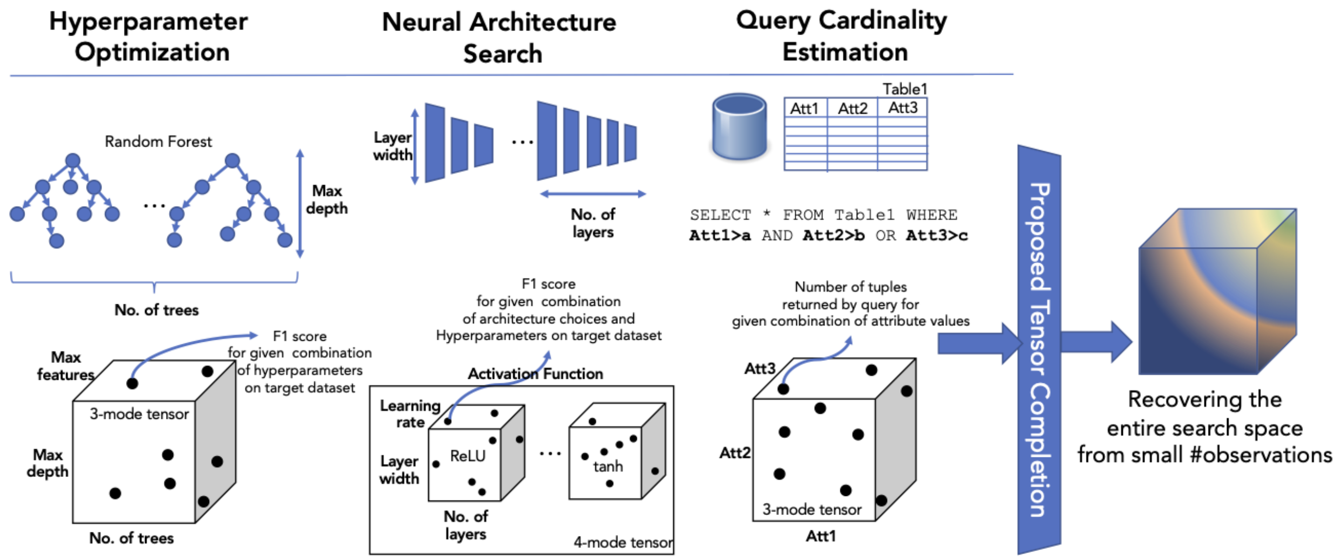
- **Broad Data Science Applications**: we take a broader view and demonstrate how this can generalize to a variety of data science problems in machine learning and database management.
- **Investigating Tensor Completion Variants**: we explore the performance of different tensor completion variants and identify pros and cons.
- **Leveraging and Exploring Tensor Modeling**: we explore the expressive power of tensor modeling in exploring ways to further improve surrogate modeling.
- **Proposed Methods**: Motivated by the problem structure, we propose applying a smoothness constraint to factor matrices in CPD tensor completion to improve performance for this application. Furthermore, we propose an ensemble tensor completion to aggregate the results of several tensor completion methods, for improved performance and reliability.
- **Public Code and Benchmark Datasets**: In order to promote reproducibility and follow-up research, we make our implementation and the benchmark datasets created for (i) hyperparameter optimization, (ii) neural architecture search, and (iii) query cardinality estimation publicly available at https://github.com/shaanpakala/STC_AutoML.

2. Preliminaries and Problem Formulation

2.1. Preliminaries

2.1.1. Data Science Pipelines. Below are the three bottleneck tasks within a data science pipeline that we focus on.
Hyperparameter Optimization: We are finding the best combination of hyperparameters [1] for a non-deep learning machine learning model (e.g. K-Nearest Neighbors, Decision Trees). This involves adjusting the values of various hyperparameters of the machine learning model, then training the model and evaluating its performance.
Neural Architecture Search: This is a similar task to Hyperparameter Optimization, except with Neural Networks (e.g. Dense Neural Networks, Convolutional Neural Networks) [23]–[25]. Now there are different hyperparameters, such as layer size or number of layers. The goal here is to find the Neural Network architecture that gives the best performance, by training and evaluating different architectures.
Query Cardinality Estimation: We want to estimate the cardinality of the output for complex database queries [26], [27]. In this application, we use tensor completion to extract the complex relationships between the predicates of a query, in order to infer the entire output cardinality. In this case, the hyperparameters would be the attribute in the predicates, and the hyperparameter values would be the range for that predicate's attribute.
Query Distinct Cardinality Estimation: We want to estimate the cardinality of the distinct values of a given attribute in the output for database queries. Here the focus is to infer the cardinality of the distinct values of the output, rather than just the entire output. This is almost identical to Query Cardinality estimation, except the entries in the tensor represent cardinality of distinct values (of a specificied attribute).

2.1.2. Surrogate Modeling. Surrogate modeling in machine learning is used to help guide the optimal hyperparameter search without exhaustively train and evaluating all combinations [18]. In our case, this will be done by estimating the performance (according to an evaluation metric, F1 Score) for all configurations of machine learning models. This will also be applied to infer the cardinalities of various database queries, with their own design configurations.



Hyperparameter Optimization
Neural Architecture Search
Query Cardinality Estimation
Random Forest
Max depth
No. of trees
Layer width
No. of layers
Table1
Att1
Att2
Att3
SELECT * FROM Table1 WHERE Att1>a AND Att2>b OR Att3>c
F1 score for given combination of hyperparameters on target dataset
F1 score for given combination of architecture choices and Hyperparameters on target dataset
Number of tuples returned by query for given combination of attribute values
Max features
3-mode tensor
Activation Function
Learning rate
ReLU
tanh
4-mode tensor
Proposed Tensor Completion
Recovering the entire search space from small #observations

Figure 1: In this work we unify a number of combinatorial and highly computationally intense data science tasks, such as hyperparameter optimization, neural architecture search, and query cardinality estimation, under the umbrella of tensor completion. We conduct a thorough and extensive study of existing tensor completion methods and propose a novel method for accurately recovering the entire search space in those data science tasks from a small number of observations, towards automating data science pipelines.

2.1.3. Tensors. Tensors are multidimensional arrays. In other words, a vector is a 1-dimensional tensor, and matrix is a 2-dimensional tensor. We will be looking at tensors of 3 or more dimensions [28], [29]. The tensors for our application will also be Sparse Tensors, which are tensors with many missing values. We use boldface italicized letters (e.g. $\mathcal{X}$) to denote dense tensors (tensors with no missing values). For sparse tensors, we will use the same notation, with a subscript "S". For example, $\mathcal{X}_S$ would be a sparse tensor corresponding to dense tensor $\mathcal{X}$.

2.1.4. Tensor Decomposition. Tensor decomposition is the process of expressing a tensor using smaller factors. For example, a common method of tensor decomposition is the Canonical Polyadic Decomposition (CPD) [28], [29]. CPD expresses a tensor as a sum of rank-one tensors. A third-order tensor $\mathcal{X} \in \mathbb{R}^{I \mathrm{x} J \mathrm{x} K}$ would be expressed as: $\mathcal{X} \approx \sum_{r=1}^{R}(\mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r)$, where $\circ$ denotes outer product, $\mathbf{a}_r \in \mathbb{R}^I, \mathbf{b}_r \in \mathbb{R}^J$, and $\mathbf{c}_r \in \mathbb{R}^K$. We represent a decomposition of tensor $\mathcal{X}$ as $\mathcal{X}_D$.

2.1.5. Tensor Completion. Tensor Completion is the process of filling in the missing values of a sparse tensor. There are several forms of Tensor Completion methods, such as:
Classical Tensor Methods: CPD [30] & TuckER [20],
Tensor Network Methods: Tensor Train [31], and
Neural Tensor Methods: CoSTCo [21] & NeAT [22].

Classical Tensor Methods heavily rely upon the tensor decomposition in order to infer back the missing values. CPD, for example, uses a series of matrix multiplication and addition in order to generate the dense tensor again.

Tensor Networks [32] have been witnessing a steady rise in popularity, as they provide a flexible and modular framework for expressing otherwise complex tensor models using elementary tensor operations such as multilinear maps (which, in a nutshell, describe how a tensor mode is projected from its original space to a new space) in network form. One of the most prominent Tensor Network models is the so-called Tensor Train [31], which we use here as a representative of Tensor Network methods.

Neural Tensor Methods use an additional neural network to infer back the missing values of the tensor. CoSTCo [21], for example, uses a Convolutional Neural Network (CNN) to extract information from very sparse tensors to accurately infer back the missing values.

2.1.6. Tensor Completion Training. Training our tensor completion algorithms begin with randomly initializing a decomposition of the dense tensor $\mathcal{X}$. Using this random decomposition $\mathcal{X}_D$, we can generate an estimation of the dense tensor $\mathcal{X}$ according to the corresponding method's algorithm. For example, CPD uses matrix products and sums to reconstruct the full tensor, while CoSTCo [21] uses CNNs. After building the estimation of the dense tensor $\mathcal{X}$, we calculate the loss using the Mean Squared Error (MSE):

$$Loss = \text{Mean}((\mathcal{X}_S - R(\mathcal{X}_D))^2 \cdot M_{\mathcal{X}}) \tag{1}$$

where $R(\mathcal{X}_{\mathcal{D}})$ represents the full reconstruction of tensor $\mathcal{X}$ using only the randomly initialized factors $\mathcal{X}_D$. $M_{\mathcal{X}}$ is a boolean/mask tensor, whose values are 1 if that entry in $\mathcal{X}_S$ is observed, and 0 if it is unobserved. This is because we only keep the reconstruction for the observed indices of the sparse tensor to calculate our loss. We multiply the loss by the mask, to convert the loss for the missing entries to 0, and the non-missing entries remain the same.

Now we can optimize the randomly initialized decomposition $\mathcal{X}_D$ in terms of MSE using Adam [33] with backpropagation. We are essentially iteratively getting closer to a full reconstruction of $\mathcal{X}$ such that the entries corresponding to the sparse tensor values are similar to their observed values.

2.2. Problem Definition

The general problem we are solving is to be able to efficiently and accurately approximate data science pipelines' outputs across a large combinatorial space of configurations.

2.2.1. Tensor Completion for Hyperparameter Tuning. Hyperparameter tuning can be modeled as a tensor, where each axis of the tensor represents a certain hyperparameter we are optimizing. Each of the values of that axis will represent the different values the hyperparameter can take. Each cell will now represent a specific combination of hyperparameter values, which will be filled in using some evaluation metric of the Machine Learning model using that hyperparameter combination. For our purposes, we will be using F1-Score for classification tasks.

This figure demonstrates how we will use only a portion of selected hyperparameter combinations, to infer the entire space of combinations of hyperparameters, using tensor completion. Above is an example of TuckER tensor completion, using Tucker tensor decomposition [20] Each of H_1, H_2, and H_3 represents a different hyperparameter that we are tuning, corresponding to that axis of the tensor.

In the figure, the tensor represents hyperparameter combinations for k-Nearest Neighbors (k-NN). Here, one axis represents k, with different values k could take, and another axis represents p (in Lp norm distance), with different values p could take. The cells represent a combination of hyperparameters that k-NN could take, using an evaluation metric. We will be using F1-Score for the evaluation metric.

2.2.2. Tensor Completion for Neural Architecture Search. Neural Architecture Search will be modeled similarly, where each axis will now represent a different kind of hyperparameter. Some examples are number of layers, layer size, and different activation functions. Each cell will also represent a combination of hyperparameter values, where the value of that cell will also represent some evaluation metric of the Deep Learning model using that hyperparameter combination. We will also be using F1-Score here.

2.2.3. Tensor Completion for Query Cardinality Estimation. Query Cardinality Estimation can be modeled as a tensor where each axis represents a certain predicate of a query. Each value of that axis represents a different value that the predicate could take. Now each cell will represent the cardinality of the output that is produced using this combination of predicate values.

2.2.4. Tensor Completion for Query Distinct Cardinality Estimation. Very similar to Query Cardinality Estimation, except now we are estimating the cardinality of the distinct values of an attribute in the output of a database query.

3. Proposed Method

In this work, we investigate the behavior of several types of Classical, Tensor Network, and Neural Tensor Completion models for a variety of tasks. We also propose applying a smoothness constraint [34] on all modes for CPD tensor completion. In addition to these individual tensor completion methods, we propose an ensemble tensor completion model, to aggregate the results of several individual ones.

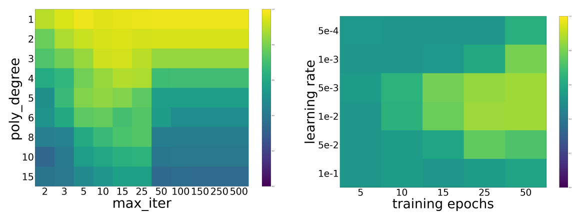
3.1. Dataset Generation

Dataset generation consists of exhaustively computing the outcomes of all combinations (of a prespecified range) of hyperparameters, neural network layers, and queries. From here, we can remove values from this complete tensor to produce our sparse tensor, simulating computing only a fraction of the combinations. Then we can observe how well we are able to infer them again. This will allow us to evaluate the performance of our tensor completion methods for these applications.
Non-Deep Learning We use scikit-learn [2] to exhaustively train and evaluate non-deep learning model hyperparameter combinations.
Neural Architecture Search We use PyTorch [36] to create dense neural networks, to also exhaustively run combinations of hyperparameters for our neural architecture search tensor.
Query Cardinality Estimation We run queries with combinations of predicate values and generate the cardinalities of the outputs for our Query Cardinality tensors and cardinalities of distinct output values for a given attribute.

A full detailed list of hyperparameters & components used for each tensor we generated can be found in Table 1.

3.2. CPD-S: Smooth CP Decomposition

The indices of the hyperparameter tensors are sequentially ordered and the corresponding performance values change smoothly as shown in Figure 2. Based on this observation, resulting factor matrices will exhibit smoothness property, where each row of factors is similar to its adjacent rows. To enforce this smoothness property, we use a kernel smoothing regularization [34] applied to all factor matrices.



(a) SVM Tensor (b) Neural Network Tensor

Figure 2: Indicative tensor slices where the nature of the hyperparameters involved results in smoothness across those dimensions motivating our proposed smoothness constrained CPD-S method.

Table 1: Hyperparameters & Ranges

Tensor File	Tensor Size	H1	H2	H3	H4	H5
Non-Deep Learning						
KNN_car_evaluation_828	3x7x2x9x11	scaler[None,minmax,standard]	PCA[1,2,3,4,5,6,None]	weights['distance','uniform']	p[1-10]	n_neighbors[1-75]
DT_Dermatology_828	8x8x8x9	max_depth[1-15,None]	max_features[1-15,None]	min_samples_leaf[1-100]	-	-
RF_Dermatology_828	8x8x8x9	min_samples_leaf[1-75]	max_features[1-15,None]	max_depth[1-15,None]	n_estimators[1-100]	-
SVM_Biodeg_905	3x2x7x9x11	scaler[None,minmax,standard]	SMOTE[False,True]	C[0.125-8]	degree[1-15]	max_iter[2-500]
SVM_Dermatology_905	3x2x7x9x11	scaler[None,minmax,standard]	SMOTE[False,True]	C[0.125-8]	degree[1-15]	max_iter[2-500]
SVM_Alzheimers_905	3x2x7x9x11	scaler[None,minmax,standard]	SMOTE[False,True]	C[0.125-8]	degree[1-15]	max_iter[2-500]
DT_Spambase_829	3x4x8x7x9	scaler[None,minmax,standard]	min_impurity_decrease[0-0.1]	min_samples_leaf[1-75]	max_features[1-10,None]	max_depth[1-15,None]
RF_Spambase_829	3x4x8x7x9	scaler[None,minmax,standard]	min_impurity_decrease[0-0.1]	min_samples_leaf[1-75]	max_features[1-10,None]	max_depth[1-15,None]
ET_Spambase_829	3x4x8x7x9	scaler[None,minmax,standard]	min_impurity_decrease[0-0.1]	min_samples_leaf[1-75]	max_features[1-10,None]	max_depth[1-15,None]
GB_Spambase_829	3x4x8x7x9	scaler[None,minmax,standard]	min_impurity_decrease[0-0.1]	min_samples_leaf[1-75]	max_features[1-10,None]	max_depth[1-15,None]
Neural Architecture Search						
FCNN_Dermatology_829	3x6x3x6x5	scaler[None,minmax,standard]	num_epochs[3-25]	batch_size[16-256]	num_layers[1-10]	hidden_size[32-5000]
FCNN_Alzheimers_902	3x6x3x6x4	scaler[None,minmax,standard]	num_epochs[2-25]	batch_size[128-2048]	num_layers[1-15]	hidden_size[32-512]
FCNN_car_evaluation_903	3x6x3x6x4	scaler[None,minmax,standard]	num_epochs[2-25]	batch_size[128-2048]	num_layers[1-15]	hidden_size[32-512]
FCNN_Dermatology_903	3x6x3x6x4	scaler[None,minmax,standard]	num_epochs[2-25]	batch_size[128-2048]	num_layers[1-15]	hidden_size[32-512]
FCNN_Particle_ID_903_02	3x6x3x6x4	scaler[None,minmax,standard]	num_epochs[2-25]	batch_size[128-2048]	num_layers[1-15]	hidden_size[32-512]
FCNN_Spambase_902	3x6x3x6x4	scaler[None,minmax,standard]	num_epochs[2-25]	batch_size[128-2048]	num_layers[1-15]	hidden_size[32-512]
FCNN_Spambase_905_50	3x6x5x5x5	activation[relu,sigmoid,tanh]	lr[0.0005-0.1]	num_epochs[5-50]	hidden_size[10-250]	num_layers[1-10]
FCNN_Biodeg_905	3x6x5x5x5	activation[relu,sigmoid,tanh]	lr[0.0005-0.1]	num_epochs[5-50]	hidden_size[10-250]	num_layers[1-10]
Query Cardinality						
AND_AND_801	10x10x10	surname_pcode['B','V','Q','G','L']	name_pcode_nf['G','I','M','O','K']	person_id[1e5-1e6]	-	-
AND_OR_801	10x10x10	surname_pcode['B','V','Q','G','L']	name_pcode_nf['G','I','M','O','K']	person_id[1e5-1e6]	-	-
OR_AND_801	10x10x10	surname_pcode['B','V','Q','G','L']	name_pcode_nf['G','I','M','O','K']	person_id[1e5-1e6]	-	-
OR_OR_801	10x10x10	surname_pcode['B','V','Q','G','L']	name_pcode_nf['G','I','M','O','K']	person_id[1e5-1e6]	-	-
Query Distinct Cardinality						
AND_AND_distinct_817	10x10x10	surname_pcode['B','V','Q','G','L']	name_pcode_nf['G','I','M','O','K']	person_id[1e5-1e6]	-	-
AND_OR_distinct_817	10x10x10	surname_pcode['B','V','Q','G','L']	name_pcode_nf['G','I','M','O','K']	person_id[1e5-1e6]	-	-
OR_AND_distinct_817	10x10x10	surname_pcode['B','V','Q','G','L']	name_pcode_nf['G','I','M','O','K']	person_id[1e5-1e6]	-	-
OR_OR_distinct_817	10x10x10	surname_pcode['B','V','Q','G','L']	name_pcode_nf['G','I','M','O','K']	person_id[1e5-1e6]	-	-

This table displays each tensor we generated and the hyperparameters or components used, along with its range of values. The Query Tensors (Query Cardinality & Distinct Cardinality) are generated on the IMDB dataset [35], and each cell represents the attribute along with the values the attribute will be compared against. For example, surname_pcode ['B','V','Q','G','L'] means the predicates are surname_pcode >= 'B', surname_pcode <= 'B', surname_pcode >= 'V', etc.

Given an N-order tensor $\mathbf{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ with observed entries Ω, and a window size S, we find factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K}$, $1 \leq n \leq N$ that minimizes

$$L = \sum_{\alpha \in \Omega} \left(x_\alpha - \sum_{k=1}^{K} \prod_{n=1}^{N} a_{i_n k}^{(n)} \right)^2 + \lambda \sum_{n=1}^{N} \sum_{i_n=1}^{I_N} \|\mathbf{a}_{i_n}^{(n)} - \tilde{\mathbf{a}}_{i_n}^{(n)}\|_2^2, \tag{2}$$

where

$$\tilde{\mathbf{a}}_{i_n}^{(n)} = \sum_{i_s \in \mathcal{N}(i_n, S)} w(i_n, i_s) \mathbf{a}_{i_s}^{(n)} \tag{3}$$

and $\mathcal{N}(i_n, S)$ indicates adjacent indices i_s of i_n in a window of size S. λ is a regularization constant to adjust the effect of the smoothing. The $\sum_{i_n=1}^{I_n} \|\mathbf{a}_{i_n}^{(n)} - \tilde{\mathbf{a}}_{i_n}^{(n)}\|_2^2$ term in Equation (2) introduces the smoothness into a factor by regularizing the i_nth row of the factor to the smoothed vector from the neighboring rows. The weight $w(i_n, i_s)$ denotes the weight to give to the i_sth row of the factor matrix. We use the Gaussian kernel to give the weight to a row closer to the i_tth row should be given a higher weight. Given a target row index i_n, an adjacent row index i_s, and a window size S, Gaussian kernel weight is defined as follows:

$$w(i_n, i_s) = \frac{\mathcal{K}(i_n, i_s)}{\sum_{i_{s'} \in \mathcal{N}(i_n, S)} \mathcal{K}(i_n, i_{s'})} \tag{4}$$

where $\mathcal{K}$ is defined by

$$\mathcal{K}(i_n, i_s) = \exp\left(-\frac{(i_n - i_s)^2}{2\sigma^2} \right)$$

Note that σ affects the degree of smoothing; a higher value of σ imposes more smoothing.

3.3. Proposed Ensemble Method

Utilizing the results of multiple tensor completion methods will allow for improved accuracy and consistency. The same intuition behind ensemble machine learning models (e.g. random forest) comes into play here: Using the results of several "weak learners" we generate a single prediction.

3.3.1. Simple Aggregation Functions. First we use some simple functions to aggregate the results of several models, such as mean, median, max and min. For this method, we fully train each individual model on the sparse tensor, and each individual model makes its own prediction. After this, for each prediction, the median, for instance, is taken from all the individual's prediction. For predicting the element at index x_i, the ensemble model's prediction, $\mathrm{P}_{\text{Ensemble}}(x_i)$, can be written as: $\mathrm{P}_{\text{Ensemble}}(x_i) = \text{median}\{\mathrm{P}_1(x_i), \mathrm{P}_2(x_i), ...\mathrm{P}_n(x_i)\}$, where $\mathrm{P}_j(x_i)$ is the jth individual model's prediction.

3.3.2. Learned Aggregation Functions. In addition to aggregating the individual model's results using a fixed function, we can learn an aggregation function. We experiment with training a Multi-Layer Perceptron (MLP) or Convolutional Neural Network (CNN) to aggregate the results of the individual tensor completion models. Here, each model's predictions would be the features, and each index that is being predicted would be the samples. This would be trained on the entire sparse tensor, besides the validation set. The individual models can be further trained while training the aggregation function, or their training can be stopped before.

3.3.3. Multiple Ranks for Tensor Completion. Introducing an ensemble tensor completion algorithm also allows us to perform different rank tensor completions on a sparse tensor. For example, we can use CPD tensor completion models that use rank 5, 10, and 15 decompositions, and aggregate all their results. The intuition behind this is that we will not know the true rank of the sparse tensor we are dealing with in the real world. For this reason, aggregating the results of multiple methods with different ranks, could offer us reliability regardless of the true rank of the tensor.

3.3.4. Splitting Data. We can also split up the sparse tensor slightly differently for each of the individual tensors in the ensemble model. For instance, if we have multiple models, we can train each individual model using only 90% of the entire sparse tensor. If we make sure each of the 10% unused is different for each model, each model would have slightly different data to train on. The intuition here is for the tensor completion models to extract different information about the data, by using different pieces of the sparse tensor.

3.3.5. Notation. We will refer to an ensemble instance as "TenSemble-model$_{aggregation}$." For example, an ensemble of CPD models, with median aggregation, will be referred to as TenSemble-CPD$_{median}$.

4. Experimental Evaluation

We use the full tensors that we generated as our ground truth. In a real world scenario, we obviously do not have access to the missing values, so there is no way to tell how well we are inferring the values. Since we generated the full tensors in this work, we will be experimenting with various types of sparse tensors (e.g. different downstream tasks, different levels of sparsity) in order to examine how well we are inferring the missing values.

4.1. Benchmarking Tensor Completion Methods

As a first step, we test and compare a number of existing tensor completion methods, in order to confirm whether our goal is feasible to begin with. To do this, we fix 5% observed values in each of our tensors (randomly sampled), and compute the error when recovering the 95% of missing values. We can observe in Figure 3 the performance of sparse tensor completion for our applications using a variety of tensor completion algorithms and a wide range of tasks and datasets. We compare these algorithms with a Naive method, which is just randomly sampling from the given sparse tensor (i.e. filling in the 95% of missing values by randomly sampling the 5% of observed values).

4.2. Ensemble Tensor Completion Performance

We want to explore the effect of aggregating the results of multiple tensor completion models together, in comparison with the individual tensor completion models. In these experiments we compare several tensor completion models with an ensemble, which consists of multiple tensor completion models of the same type. For example, Table 2 displays 3 individual CPD models, of rank 1, 3, and 5, compared with different ways to aggregate these 3 models into an ensemble. We compare the individual models with their various ensembles on our four tasks: Non-Deep Learning (NDL), Neural Architecture Search (NAS), Query Cardinality (QC), and Query Distinct Cardinalty (QDC).

Table 2: CPD Individual vs Ensembles (5% observed values)

	NDL	NAS	QC	QDC
Rank 1	.062 ± .00	.154 ± .01	.190 ± .06	.149 ± .05
Rank 3	.071 ± .00	.180 ± .02	.207 ± .03	.187 ± .02
Rank 5	.071 ± .01	.158 ± .01	.214 ± .03	.180 ± .02
TenSemble-CPD_$_*$				
* = Median	.060 ± .00	.151 ± .01	.170 ± .03	.149 ± .01
* = Mean	**.058** ± **.00**	.150 ± .00	**.167** ± **.03**	.145 ± .01
* = MLP	**.058** ± **.01**	**.127** ± **.02**	.170 ± .03	**.141** ± **.02**

Table 3: CPD-S Individual vs Ensembles (5% observed values)

	NDL	NAS	QC	QDC
Rank 1	.061 ± .00	.190 ± .08	.184 ± .10	**.123** ± **.02**
Rank 3	.048 ± .00	.177 ± .04	.164 ± .01	.176 ± .03
Rank 5	.047 ± .00	.144 ± .01	.151 ± .01	.180 ± .02
TenSemble-CPD-S_$_*$				
* = Median	.045 ± .00	.155 ± .03	.141 ± .01	.152 ± .02
* = Mean	.045 ± .00	.154 ± .03	.148 ± .02	.148 ± .02
* = MLP	**.043** ± **.00**	**.117** ± **.01**	**.134** ± **.00**	.156 ± .03

Table 4: CoSTCo Individual vs Ensembles (5% observed values)

	NDL	NAS	QC	QDC
Rank 10	.078 ± .01	.124 ± .01	.117 ± .02	.109 ± .02
Rank 20	.067 ± .00	.116 ± .01	.120 ± .03	.094 ± .01
Rank 32	.087 ± .02	.116 ± .00	**.099** ± **.02**	.093 ± .02
TenSemble-CoSTCo_$_*$				
* = Median	.070 ± .01	.113 ± .01	.106 ± .02	.089 ± .01
* = Mean	.069 ± .01	.114 ± .01	.107 ± .02	.089 ± .01
* = MLP	**.059** ± **.00**	**.105** ± **.01**	.107 ± .02	**.087** ± **.01**

Each cell is average ± standard deviation of MAE over 5 iterations.

Across these three models, CPD, CPD-S, and CoSTCo [21], we can see that aggregating the tensor completion methods in an ensemble usually improves upon the performance of the individual models. Of the aggregation functions, the MLP aggregation function seems it usually

Figure 3: Sparse Tensor Completion MAE using 5% observed values. Each cell represents average MAE over 5 iterations.

	NDL			NAS			QC			QDC		
	1	2	3	1	2	3	1	2	3	1	2	3
Naive	0.224	0.164	0.229	0.116	0.352	0.150	0.198	0.352	0.248	0.288	0.225	0.174
CPD	0.069	0.116	**0.018**	0.069	0.157	0.105	0.103	0.178	0.171	0.115	0.207	0.176
CPD-S	**0.048**	0.067	0.019	0.064	0.126	0.095	0.127	0.131	0.145	0.121	0.160	0.130
TuckER	0.190	0.085	0.151	0.126	0.316	0.120	0.288	0.262	0.258	0.276	0.315	0.269
Tensor Train	0.091	0.117	0.036	0.066	0.152	0.113	0.094	0.162	0.173	0.123	0.221	0.176
NeAT	0.159	0.119	0.155	0.108	0.245	0.120	0.150	0.222	0.198	0.185	0.179	0.146
CoSTCo	0.117	**0.061**	0.082	**0.062**	**0.099**	**0.090**	**0.080**	**0.077**	**0.100**	**0.086**	**0.110**	**0.099**

Tensors used in table:
NDL Tensors: (1) DT_Dermatology_828, (2) KNN_Alzheimers_902, (3) RF_Spambase_829
DL Tensors: (1) FCNN_Particle_ID_903_02, (2) FCNN_Dermatology_829, (3) FCNN_car_evaluation_903
QC Tensors: (1) AND_AND_801, (2) AND_OR_801, (3) OR_OR_801
QDC Tensors: (1) AND_AND_distinct_817, (2) OR_AND_distinct_817, (3) OR_OR_distinct_817

outperforms the others (mean & median), but it is unclear how consistent this result is.

4.3. Computational Efficiency

Even though our proposed CPD-S and ensemble methods match but not necessarily outperform CoSTCo, CPD-S and CPD-based ensembles are more lightweight in terms of parameters to be learned and can potentially be more efficient as a result. Tables 5 and 6 show runtimes for the top-performing models. For the ensemble model we have divided the runtime of serially executed base models over the number of base models to simulate the ideal parallel case. Indicatively, CPD-S is either on par or faster than CoSTCo while requiring much fewer parameters and TenSemble-CPD-S was $1.9\times - 78\times$ faster than CoSTCo [1] assuming parallel execution, since each base model in the ensemble works on a sparser tensor than an individual model. As we are primarily focused on feasibility in this work, these results are highly encouraging and we defer further scalability investigation to future work.

% observed	CPD-S	CoSTCo	TenSemble-CPD-S$_{MLP}$
1%	0.635 ± 0.29	0.447 ± 0.07	0.737 ± 0.20
2.5%	1.835 ± 1.48	1.113 ± 0.12	2.346 ± 1.06
5%	2.836 ± 1.33	6.143 ± 3.10	2.612 ± 0.25
10%	1.133 ± 0.10	116.460 ± 4.79	1.479 ± 0.11

Table 5: Runtime in seconds for each model (training & inference) on KNN_car_evaluation_828. Average ± STD over 5 iterations.

% observed	CPD-S	CoSTCo	TenSemble-CPD-S$_{MLP}$
1%	0.173 ± 0.06	0.193 ± 0.04	0.289 ± 0.04
2.5%	0.646 ± 0.54	0.366 ± 0.06	0.538 ± 0.21
5%	0.541 ± 0.29	1.213 ± 0.31	0.625 ± 0.12
10%	0.844 ± 0.20	8.997 ± 2.63	1.387 ± 0.87

Table 6: Runtime in seconds for each model (training & inference) on FCNN_Spambase_905_50. Average ± STD over 5 iterations.

1. For 10% observations and higher CoSTCo often fails to converge necessitating a restart, which results in significantly higher runtime

4.4. Data Efficiency Analysis

The rationale behind using completion to recover the entire space is to avoid evaluating a combinatorial number of potentially very expensive experiments. We are investigating in identifying the minimum number of observed entries (i.e., exact evaluations of parameter combinations) necessary in order to perform high-quality completion. In other words, we want to know how sparse our tensors can be.

Figure 4 displays the Mean Absolute Error (MAE) of tensor completion on the Y-axis, and the percent of observed entries in the sparse tensor on the X-axis (grouped by tensor completion model). The graphs show the performance of various tensor completion methods with respect to the sparsity of the tensor. One distinct pattern to be seen is the difficulty of sparse tensor completion with relation to the rank of the tensor. Query Cardinality and Non-Deep Learning tensors seem the easiest to complete, due to the low rank of the tensors. The Neural Architecture Search tensors seem harder to complete due to their high rank.

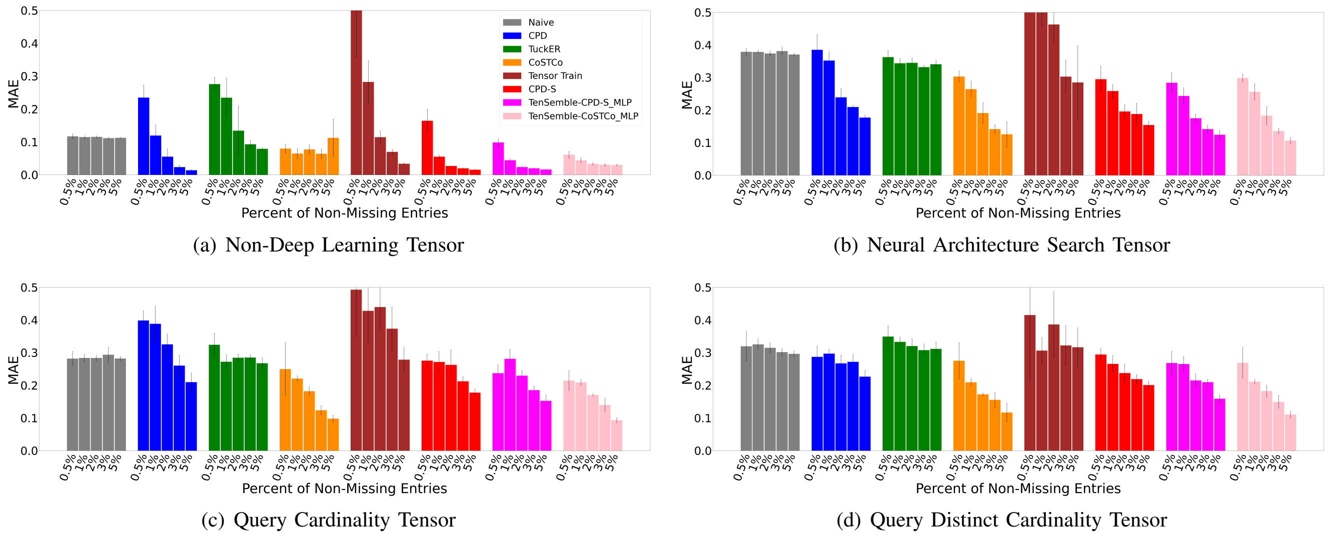
For many tasks, CoSTCo [21] has the least error with more missing values; however, our CPD-S Ensemble model gives very similar performance on most of the tensors. Interestingly though, this is not the case for the Query Cardinality Tensors. Across a variety of Non-Deep Learning and Neural Architecture Search tensors, though, we see that the CPD-S Ensemble gives very similar performance of CoSTCo, without needing the Convolutional Neural Network .

4.5. CPD-S Smoothness Sensitivity

When adding a smoothness constraint to our CPD tensor completion, we can choose to enforce this constraint more strictly or more loosely. This is done by increasing or decreasing the smoothness constraint's coefficient term, lambda, in our loss function. Note that when lambda = 0, the loss function is just the base loss function (MSE).

We observe in Figure 5 that a positive lambda value almost always helps with tensor completion in our application. In some cases, a smaller lambda value is more effective, but this does not seem to be true for all cases.

Figure 4: Error vs. levels of sparsity for various models across all four tasks. CoSTCo [21] & its ensemble consistently perform the best with little observed entries. Closer to 5% observed entries, the rest of the models seem to catch up. Our proposed CPD-S & TenSemble-CPD-S$_{MLP}$, however, has similar performance to CoSTCo [21], without using as many parameters as a CNN.



(a) Non-Deep Learning Tensor (b) Neural Architecture Search Tensor

(c) Query Cardinality Tensor (d) Query Distinct Cardinality Tensor

Figure 5: CPD-S Error with different lambda coefficient values, compared with regular CPD & Naive methods. These graphs display that a positive lambda value (enforcing smoothness constraint) almost always decreases the error in the scope of our application.

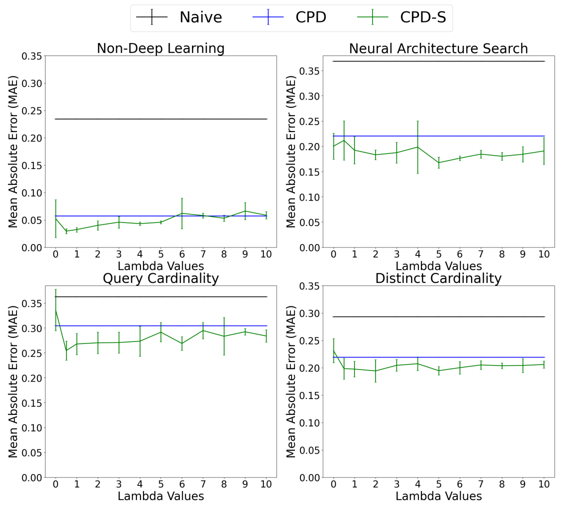
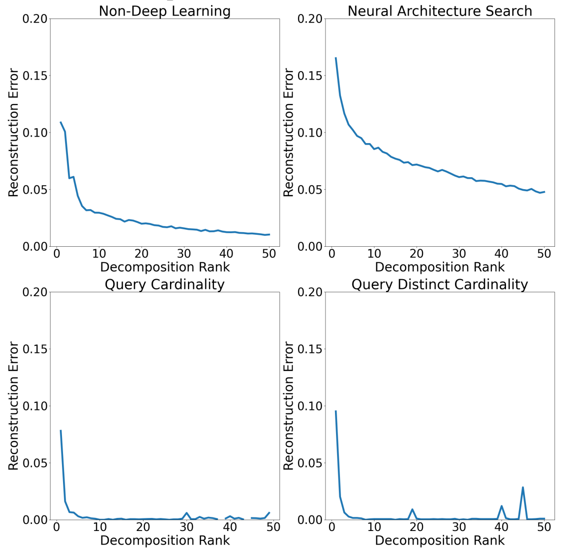


Figure 6: These graphs display the normalized (0, 1) error when decomposing and reconstructing a dense tensor of each task, with respect to the decomposition rank.



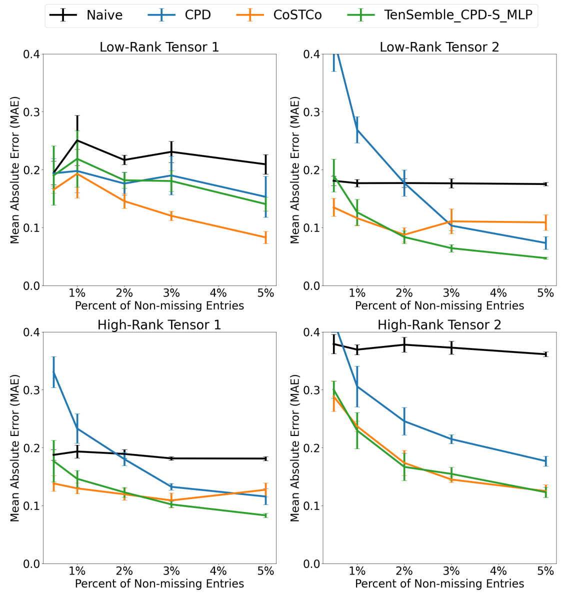
4.6. Investigating the Latent Structure of the Data

In our tensor completion algorithms, we assume there is structure in the tensors that we are completion, otherwise we could not complete them. However, it is unclear whether or not we can assume a strictly low-rank structure. In the real world, we will not be able to find out what the exact rank of our sparse tensor is, because to the all of the missing values. It is for this reason that it is important to conduct some analysis with respect to the rank of the tensors we are completing, since we do have the completed dense tensors.

In Figure 6, we perform TensorLy's PARAFAC decomposition [37] on our dense tensors. From this decomposition, we reconstruct the entire dense tensor back again, and record this error. When doing this decomposition and reconstruction across many different ranks, we observe what ranks are needed to accurately reconstruct the tensor. This gives us an estimation on the tensor's rank. For example, the pictured Query Cardinality tensor only needs a low-rank decomposition to accurately estimate the values of the entire dense tensor. On the other hand, the Neural Architecture Search tensor requires a much higher rank decomposition to be able to estimate the values accurately.

Generally we observe this trend where Query tensors tend to be very low rank, Non-Deep Learning tensors are slightly higher rank, and Neural Architecture Search tensors are significantly higher rank. This observation comes in contrast to previous work [17]–[19], that makes a strong low-rank assumption, which appears to not always be the case in the diverse set of scenarios we explore here, and further justifies the good performance of methods like CoSTCo, which do not make strong low-rank assumptions.

Figure 7: Completion error for low and high rank tensors. The top right is for a query cardinality tensor, top left graph for a SVM tensor, and the bottom two for Neural Architecture Search tensors.



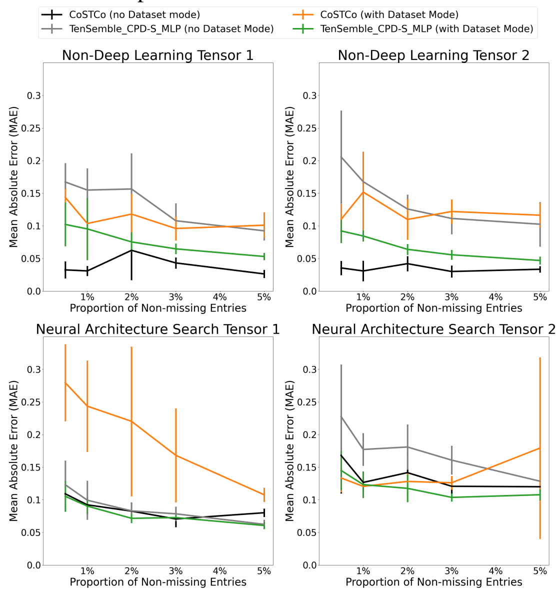
Previous work on Tensor Completion for AutoML assumes a low-rank structure of the tensor [17]–[19]; however, we experimentally verify the viability of tensor completion even for higher-rank tensors. Not only does this allow us to extend our work into Neural Architecture Search (which tends to produce higher rank tensors), but it also gives us robustness when we do not know the exact rank of our sparse tensor. Figure 7 shows results in low and high rank tensors.

Finally, we would like to note that low-rankness of a given tensor, as measured here, does not necessarily imply that the task is trivial. This is because our rank assessment is done using the full tensors, while in reality we would have to estimate that low-rank space from a small amount of observations, which is challenging.

4.7. Cross-dataset Completion

Taking advantage of the flexibility and extendibility of tensor modeling, we ask the question: can we introduce a “dataset” mode and conduct joint completion across different downstream datasets? Essentially, can we transfer results from one dataset to another using our formulation? Figure 8 shows the effects of introducing a dataset mode. Interestingly enough, CoSTCo [21] actually seems to suffer from introducing the dataset mode, likely due to overfitting to the other datasets’ tensors. On the other hand, we observe that our TenSemble-CPD-S$_{MLP}$ model can benefit significantly from the introduction of a dataset mode.

Figure 8: Introducing a dataset mode, for non-deep learning and neural architecture search tensors. The dataset mode is to simulate where we might already have the results of previous tasks. Note the other slices corresponding to the other datasets have 15% observed entries for this experiment.



5. Related Work: Tensor Methods for AutoML

There exists immediately related work that addresses the exact issue of using Sparse Tensor Completion for AutoML [17]–[19]. In contrast to those works, we relax the strict low-rank assumptions. This is to our benefit as we do come across a variety of Neural Architecture Search tensors that are quite high rank. This also adds to the robustness of this application since it may be difficult to tell the exact rank of the tensor when we only have a small fraction of observed entries. Finally, we go beyond hyperparameter optimization, as we take a broader approach and use tensor completion methods for a variety of data science tasks.

6. Conclusions

We make the following high-level contributions:

- We cast a number of diverse data science tasks, such as hyperparameter optimization, neural architecture search, and query cardinality estimation, which currently act as computationally intensive bottlenecks in building data science pipelines under the unifying umbrella of tensor completion.
- We extensively compare state-of-the-art tensor completion methods and confirm feasibility of our proposition.
- Inspired by the underlying structure that emerges in the data and by the observed behavior of existing completion methods, we propose a framework of novel tensor completion methods that are able to achieve state-of-the-art performance in our tasks.
- We publicly release our code and benchmark datasets, empowering further research in this direction.

Acknowledgements

We would like to thank Kuntal Pal for initial discussions. Research was supported by the National Science Foundation under REU Site grant no. OAC 2244480, CAREER grant no. IIS 2046086 and CREST Center for Multidisciplinary Research Excellence in Cyber-Physical Infrastructure Systems (MECIS) grant no. 2112650.

References

[1] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in neural information processing systems*, vol. 24, 2011.

[2] 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*, vol. 12, pp. 2825–2830, 2011.

[3] R. Bellman and R. Kalaba, "On adaptive control processes," *IRE Transactions on Automatic Control*, vol. 4, no. 2, pp. 1–9, 1959.

[4] J. Wu, X.-Y. Chen, H. Zhang, L.-D. Xiong, H. Lei, and S.-H. Deng, "Hyperparameter optimization for machine learning models based on bayesian optimization," *Journal of Electronic Science and Technology*, vol. 17, no. 1, pp. 26–40, 2019.

[5] J. van Hoof and J. Vanschoren, "Hyperboost: Hyperparameter optimization by gradient boosting surrogate models," *arXiv preprint arXiv:2101.02289*, 2021.

[6] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*. Springer, 2011, pp. 507–523.

[7] R. S. Olson, W. L. Cava, Z. Mustahsan, A. Varik, and J. H. Moore, "Data-driven advice for applying machine learning to bioinformatics problems," in *Pacific symposium on biocomputing 2018: Proceedings of the pacific symposium*. World Scientific, 2018, pp. 192–203.

[8] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge-based systems*, vol. 212, p. 106622, 2021.

[9] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies–a comprehensive introduction," *Natural computing*, vol. 1, pp. 3–52, 2002.

[10] C. Coello, "Evolutionary algorithms for solving multi-objective problems," 2007.

[11] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, vol. 18, no. 185, pp. 1–52, 2018.

[12] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, "F-race and iterated f-race: An overview," *Experimental methods for the analysis of optimization algorithms*, pp. 311–336, 2010.

[13] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of machine learning research*, vol. 18, no. 153, pp. 1–43, 2018.

[14] J. Lorraine, P. Vicol, and D. Duvenaud, "Optimizing millions of hyperparameters by implicit differentiation," in *International conference on artificial intelligence and statistics*. PMLR, 2020, pp. 1540–1552.

[15] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.

[16] B. Bischl, M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix *et al.*, "Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 13, no. 2, p. e1484, 2023.

[17] L. Deng and M. Xiao, "A new automatic hyperparameter recommendation approach under low-rank tensor completion e framework," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 4, pp. 4038–4050, 2022.

[18] C. Yang, J. Fan, Z. Wu, and M. Udell, "Automl pipeline selection: Efficiently navigating the combinatorial space," in *proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 2020, pp. 1446–1456.

[19] A. Rebello, K. Konstantinidis, Y. L. Xu, and D. Mandic, "Efficient hyperparameter optimization through tensor completion."

[20] I. Balažević, C. Allen, and T. M. Hospedales, "Tucker: Tensor factorization for knowledge graph completion," in *Empirical Methods in Natural Language Processing*, 2019.

[21] H. Liu, Y. Li, M. Tsang, and Y. Liu, "Costco: A neural tensor completion model for sparse tensors," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 324–334.

[22] D. Ahn, U. S. Saini, E. E. Papalexakis, and A. Payani, "Neural additive tensor decomposition for sparse tensors," in *33rd ACM International Conference on Information and Knowledge Management*. ACM, 2024.

[23] B. Zoph, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[24] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 1946–1956.

[25] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training," in *International conference on machine learning*. PMLR, 2021, pp. 7588–7598.

[26] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner, "Cardinality estimation with local deep learning models," in *Proceedings of the second international workshop on exploiting artificial intelligence techniques for data management*, 2019, pp. 1–8.

[27] T. Malik, R. C. Burns, and N. V. Chawla, "A black-box approach to query cardinality estimation." in *CIDR*, 2007, pp. 56–67.

[28] T. Kolda and B. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, 2009.

[29] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Signal Processing Magazine*.

[30] R. Harshman, "Foundations of the parafac procedure: Models and conditions for an" explanatory" multimodal factor analysis," 1970.

[31] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[32] R. Sengupta, S. Adhikary, I. Oseledets, and J. Biamonte, "Tensor networks in machine learning," *European Mathematical Society Magazine*, no. 126, pp. 4–12, 2022.

[33] D. P. Kingma, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[34] D. Ahn, J.-G. Jang, and U. Kang, "Time-aware tensor decomposition for sparse tensors," in *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2021, pp. 1–2.

[35] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, 2015.

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[37] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, "Tensorly: Tensor learning in python," *Journal of Machine Learning Research*, vol. 20, no. 26, pp. 1–6, 2019.