Efficient and Effective Accelerated Hierarchical Higher-Order Logistic Regression for Large Data Quantities[*]

Nayyar A. Zaidi, Francois Petitjean, Geoffrey I. Webb [†]

Abstract

Machine learning researchers are facing a data deluge – quantities of training data have been increasing at a rapid rate. However, most of machine learning algorithms were proposed in the context of learning from relatively smaller quantities of data. We argue that a big data classifier should have superior feature engineering capability, minimal tuning parameters and should be able to learn decision boundaries in fewer passes through the data. In this paper, we have proposed an (computationally) efficient yet (classification-wise) effective family of learning algorithms that fulfils these properties. The proposed family of learning algorithms is based on recently proposed accelerated higher-order logistic regression algorithm: ALR^n. The contributions of this work are three-fold. First, we have added the functionality of out-of-core learning in ALR^n, resulting in a limited pass learning algorithm. Second, superior feature engineering capabilities are built and third, a far more efficient (memory-wise) implementation has been proposed. We demonstrate the competitiveness of our proposed algorithm by comparing its performance not only with state-of-the-art classifier in out-of-core learning such as Selective KDB but also with state-of-the-art in in-core learning such as Random Forest.

Keywords — Higher-order Logistic Regression, Feature Engineering, Tuple/Feature Selection, SGD, Adaptive Step-Size

1 Introduction

The continuous growth in the amount of training data brings new challenges to machine learning researchers and practitioners. [6, 11, 10]. Most machine learning algorithms were developed in the context of smaller quantities of data. On these small quantities, simple linear classifiers were to be preferred over non-linear classifiers (which, of course, resulted in complex decision boundaries), as variance contributed most to the error. A well-known widely used example of a linear classifier is Logistic Regression (LR), which optimizes the following objective function:

$$\mathrm{NLL}(\boldsymbol{\beta}) = \sum_{i=1}^{N} \left(\boldsymbol{\beta}_y^T \mathbf{x}^{(i)} - \log \sum_{c}^{C} \exp(-\boldsymbol{\beta}_c^T \mathbf{x}^{(i)}) \right) + \lambda ||\boldsymbol{\beta}||^2,$$

where NLL stands for Negative Log-Likelihood. It can be seen that the resulting model will be highly biased if the data is not linearly separable. Recently, it has been shown in [16] that one can reduce the bias of LR by taking into account higher-order feature interactions. The resulting ALR^n (Accelerated higher-order Logistic Regression) model optimizes the following objective function:

$$\mathrm{NLL}(\boldsymbol{\beta}) = \sum_{i=1}^{N} \left(\boldsymbol{\beta}_y^T f_n(\mathbf{x}^{(i)}) - \log \sum_{c}^{C} \exp(-\boldsymbol{\beta}_c^T f_n(\mathbf{x}^{(i)})) \right) + \lambda ||\boldsymbol{\beta}||^2,$$

where the function $f_n(\mathbf{x}) = \mathrm{vec}(\mathrm{lowDiag}(f_{n-1}(\mathbf{x})\mathbf{x}^T))$ and $f_{n=1}(\mathbf{x}) = \mathbf{x}$. Here, the function *lowDiag* returns the lower-diagonal terms of the matrix and the function *vec* vectorizes them. Note, the model is still linear in $\boldsymbol{\beta}$, but now, it is incorporating all n-order interactions. It can be seen that this greatly increases the number of parameters to be learned (size of the vector $\boldsymbol{\beta}$), and, therefore, an efficient implementation is warranted. Nonetheless, increasing the value of n in ALR^n leads to lower-biased models [1].

Let us first describe the main symbols and notations used in this work. We will use N to denote the number of data points, $|\mathcal{A}|$ is the number of attributes, C is the number of classes and n denotes the order in higher-order LR model. The symbols in bold face are vectors. The parameter to be optimized is: $\boldsymbol{\beta}_y$ – since, we are optimizing the softmax objective function, each class y has its own associated parameter vector, hence the subscript y. We will make a distinction between the terms *tuple*, *feature* and *parameter*. A *tuple* denotes a set of attributes, e.g. $\mathcal{T}_{12,5} = \{\mathrm{att}_{12}, \mathrm{att}_5\}$ is a *tuple* of order 2 consisting of attributes 12 and 5. Note the elements of the *tuples* are always in decreasing

[*]This work was supported by the Australian Research Council under awards DP140100087 and DE170100037. This material is based upon work supported by the Air Force Office of Scientific Research, Asian Office of Aerospace Research and Development (AOARD) under award number FA2386-17-1-4033.

[†]Faculty of Information Technology, Monash University, Clayton, VIC 3800, Australia.

[1]E.g., ALR^n with $n = 2$ is a quadratic model, that can not only capture quadratic interactions but also linear interactions. On the other hand, a linear model can not take into account quadratic terms. Hence ALR^2 will be lower biased.

order. This ensures the uniqueness of the *tuples*. A *feature*, on the other hand, is the instantiation of the attribute values in the *tuple*, e.g, $\mathcal{F}_{\mathbf{12}:1,\mathbf{5}:3} = \{\text{att}_{12} = 1, \text{att}_5 = 3\}$ is a *feature* of order 2 with attribute 12 taking value 1 and attribute 5 taking value 3. We use the term *parameter* for an element in any vector that is to be optimized, e.g, as described before, $\boldsymbol{\beta}$ is the *parameter* vector. Each element of parameter vector is associated with a *feature* and a class-value. For example $\beta_{\mathbf{12}:1,\mathbf{5}:3,\mathbf{C}:2}$ corresponds to: *feature*-class pair of $\{\text{att}_{12} = 1, \text{att}_5 = 3\}$ and $\{\text{att}_{\text{class}} = 2\}$ or just $\{\text{att}_{12} = 1, \text{att}_5 = 3, \text{att}_{\text{class}} = 2\}$.

For larger quantities of data, an algorithm should inherently be *low-biased* [1]. In other words, a model should be expressive enough to model most of the interactions that exist in the data – this capability is now widely known as the *Feature Engineering* capability of a model. We argue that the better the feature engineering capability, the better its performance on big datasets. Second, it is also desirable for the model to learn in few passes through the data. This is motivated from the fact that data in large quantities cannot be fully loaded into the memory and, therefore, out-of-core processing of the data is mandatory. A final property is being able to learn with minimal tuning parameters. For larger quantities of data, of course, a model should be able to learn with minimum intervention. More importantly, the model's performance should not be critically dependent on a large number of hyper-parameters.

Let us now analyze ALR^n in lights of above discussion to determine its suitability for learning with large quantities of data. It can be seen, that in this regard, there are at least two issues with ALR^n:

- First, ALR^n leads to a massive model. There is no *tuple* or *feature* selection and for moderate size datasets, training a higher-order LR will not be feasible. There is a need for an efficient *feature* engineering capability. This is one of the reasons that in the original work [16], the maximum value of n was set to 3.

- Second, ALR^n, in its original form is in-core – batch optimization-based methods such as Quasi-Newton, TRON, etc. which require loading the data into the memory and quite memory inefficient as just one iteration of the optimization leads to many function evaluations [15]. There is a need to learn in minimal passes through the data.

The proposed family of algorithms in this paper is motivated to address these shortcomings of ALR^n. We propose two new learning algorithms: sALR^n and hsALR^n. Here are the salient features of our proposed algorithms:

- The proposed algorithms processes data out-of-core. They are limited to a maximum of 10 passes through the data. The discriminative parameters are optimized with an efficient Stochastic Gradient-Descent (SGD) based method, where the step-size is adaptive and the initial step-size is tuned automatically.

- The algorithms are based on an efficient implementation for storing parameters. The resulting data-structure not only helps in managing memory requirements but also results in scaling to higher values of n.

- sALR^n does *tuple* selection to reduce the model size. It is based on a top-down approach, i.e, all *tuples* at level n are evaluated and selected only if they pass some selection test.
- hsALR^n hierarchically build *tuples* and *features*. This a bottom-up approach – lower-level *tuples* and *features* are evaluated first and selected only if they pass some test. The selected *tuples* and *features* are later used to build higher-order *tuples* and *features*.

We believe that sALR^n and hsALR^n are excellent representative examples of *Feature Engineering*.

The rest of this paper is organized as follows: we start by discussing related work in Section 2. We present our proposed algorithms in Section 3 and 4. The experimental results are given in Section 5. We conclude in Section 6 with pointers to future works.

2 Related Work

An efficient algorithm that fulfils the three properties of large-scale machine learning i.e, feature engineering, out-of-core data processing and minimal tuning parameters is proposed in [9]. The resulting SKDB (Selective K-Dependence Bayesian Network Classifier) is a three pass learning algorithm, that is built on a two pass K-Dependence Bayesian Classifier (KDB) algorithm. A KDB classifier factorizes the joint distribution as: $\mathrm{P}(Y, X) = \mathrm{P}(Y) \prod_{i=1}^{|\mathcal{A}|} \mathrm{P}(X_i|\mathrm{Pa}(X_i))$ – where the function $\mathrm{Pa}(X_i)$ returns the K parents of attribute X_i. In its first pass, it computes the mutual information of each attribute with the class: $\mathrm{MI}(X_i|Y)$ and also the conditional mutual information of the pair of attributes with the class: $\mathrm{CMI}(X_i, X_j|Y)$. These two statistics are used to learn the structure of the network, i.e., the function $\mathrm{Pa}(X_.)$.

Selective KDB adds a third pass to the standard KDB learning algorithm and use an elegant leave-one-out cross validation algorithm to select the best value of K and the best number of attributes (note, the algorithm relies on exploiting the symmetry of the KDB model). It has been shown that SKDB leads to comparable performance to in-core state-of-the art classifier – Random Forest [2].

3 sALR^n

Let us discuss our first proposed algorithm – Selective (higher-order) Accelerated Logistic Regression: sALR^n. As discussed, the algorithm is based on ALR^n – like ALR^n, it is based on learning both generative ($\boldsymbol{\theta}$) and discriminative ($\boldsymbol{\beta}$) parameters of the model. The generative parameters are to be used as a pre-conditioner for the discriminative parameters which should lead to faster convergence [13]. However, unlike ALR^n, it is far more computationally efficient due to its selection of *tuples*. Furthermore, unlike ALR^n, the algorithm is only an $I + 3$ pass learning algorithm, where I defaults to five – in total, three initial passes for *tuple* selection and *tuple* evaluation, and then learning the generative parameters, and five passes for discriminative learning of the parameters.

The algorithm starts by allocating two data structures: $\mathcal{DICT}$ and $\mathcal{BM}$. $\mathcal{DICT}$ is an index dictionary of size $\mathcal{T}$, where $\mathcal{T}$ is the total number of *tuples*. Each element of $\mathcal{DICT}$ assigns an index to a *tuple*. The goal is that, based on this index, and given the cardinality of each attribute in the *tuple*, one can uniquely assign a number to all possible *parameters* [3]. $\mathcal{BM}$ is an array of Booleans [4], which specifies if a certain *parameter* is present in the training data or not. It can be seen that regardless of whether a *parameter* is needed or not, a bit associated with it must be present in $\mathcal{BM}$. So the size of $\mathcal{BM}$ can be calculated in advance.

An outline of the algorithm is given in Algorithm 1. The first pass of the sALR^n is explanatory. If a certain *feature* is encountered, bits associated with its *parameters* are set. The goal is to allocate memory only for the *features* that are present in the data. This drastically reduces the size of model. Secondly, sALR^n, in this pass, extracts some data (denoted by $\mathcal{D}^{\mathcal{CV}}$) to be used in the cross-validation step. By default, five percent of the data is extracted and is stored in the memory. Note, this step will result in creating a new data set file on the disk with $\mathcal{D}^{\mathcal{CV}}$ removed. A stratified reservoir sampling is done to extract $\mathcal{D}^{\mathcal{CV}}$.

Once the $\mathcal{BM}$ is created and $\mathcal{D}^{CV}$ is removed from $\mathcal{D}$, the cardinality of the $\mathcal{BM}$: $|\mathcal{BM}|$ specifies the length of the *parameter* vector. Note, that now, any access to the *parameter* has to be through the data structures $\mathcal{DICT}$ and $\mathcal{BM}$. For example, to access the *parameter* associated with the *feature* $\{\text{att}_5 = 2, \text{att}_3 = 100, \text{att}_2 = 1\}$ and class $\{\text{att}_{\text{class}} = 5\}$, first an index, say i, is retrieved from $\mathcal{DICT}$, which is then used to query $\mathcal{BM}$. If $\mathcal{BM}(i)$ returns false, the *feature* is not present and is ignored, otherwise, the new index is calculated as:

$$\mathcal{BM}(i) = \sum_{j=0}^{i-1} \mathbf{1}_{\mathcal{BM}(j)==1}, \tag{3.1}$$

where $\mathbf{1}_{x=y}$ is a function that returns one if $x = y$, otherwise, zero. Note that the operation of Equation 3.1 is highly optimized in all programming languages. The *parameter* vector $\boldsymbol{\theta}$ is allocated in memory of the size of $|\mathcal{BM}|$ – let us denote the size of the *parameter* vector as $|\mathcal{P}|$.

The second pass of sALR^n involves calculating the counts of the *features* and populating the $\boldsymbol{\theta}$ vector. After the pass, the counts are converted into probabilities, M-estimate (Maximum likelihood estimates (MLE) with Dirichlet priors on the parameter) is used for computing probabilities. sALR^n uses the probabilities of the form: $\text{P}(\mathcal{F}|y)$, which is computed as:

$$\text{P}(\mathcal{F}|Y = y) = (\theta_{\mathcal{F},y} + m/|\mathcal{F}|)/(C + m).$$

Note, $\text{P}(\mathcal{F}|Y = y)$ denotes the probability of the *feature* $\mathcal{F}$ given the class y. Also, m is fixed to 0.1 and $|\mathcal{F}|$ is the cardinality of the *feature* $\mathcal{F}$ and is equal to the sum of the product of the cardinality of attributes in the *feature*.

Once the probabilities are computed, *tuple* selection begins. To do that, *tuples* are evaluated based on their mutual information (MI) score:

$$\text{MI}(\mathcal{T}|Y) = \sum_{y=1}^{C} \left(\sum_{x'=1}^{|\mathcal{T}_1|} \dots \sum_{x''=1}^{|\mathcal{T}_n|} \right) \text{P}(\mathcal{F}, y) \log \frac{\text{P}(\mathcal{F}, y)}{\text{P}(\mathcal{F})\text{P}(y)},$$

where $\mathcal{T}_i$ denotes the i-the attribute of *tuple* $\mathcal{T}$ and $|\mathcal{T}|$ its cardinality. Note, that for each possible instantiation of attribute-values of *tuple* $\mathcal{T}$, there is a *feature* $\mathcal{F}$. Hence we have left the indices for *feature* $\mathcal{F}$ in terms $\text{P}(\mathcal{F}|y)$ and $\text{P}(\mathcal{F})$ for simplicity of notation.

Once the *tuples* are evaluated based on their MI score, top t *tuples* are selected and others are ignored. This operation only requires modifying the $\mathcal{BM}$ data

[2]Note, another relevant algorithm is an ensemble model – Averaged-N-Dependence Estimator (AnDE), which is based on a single pass learning algorithm. Also, feature selection (known as SPODE selection) capability has been shown to work well for AnDE as well [14]. We have not included AnDE results in this work as the model in practice has similar performance to KDB.

[3]One could store an index for every *parameter* but only at the cost of huge memory consumption. Typical method rely on hashing the string-names of the attributes which results in approximate solutions due to possible collisions [8].

[4]Most programming languages have a BitSet data structure which returns either true or false at a particular index.

Algorithm 1 Selective Accelerated Higher-order LR

```
1: procedure sALR$^n$($\mathcal{D}, \mathcal{DICT}, \mathcal{BM}, \mathcal{TS}, t$)
2:
3:     [$\mathcal{BM}, \mathcal{D}^{\mathcal{CV}}$] ← explorationPass($\mathcal{D}$) # Pass 1
4:     $\boldsymbol{\theta}$ ← allocateMemory($\mathcal{BM}$),
5:     $\boldsymbol{\theta}$ ← getCountsFromData($\mathcal{D}, \mathcal{BM}$) # Pass 2
6:     $\mathcal{BM}^*$ ← doTupleSelection($\mathcal{D}^{\mathcal{CV}}, \mathcal{BM}, \mathcal{TS}, t$)
7:
8:     $\boldsymbol{\theta}^*$ ← allocateMemory($\mathcal{BM}^*$),
9:     $\boldsymbol{\theta}^*$ ← getCountsFromData($\mathcal{D}, \mathcal{BM}^*$) # Pass 3
10:    [$\boldsymbol{\beta}, G$] ← allocateMemory($\mathcal{BM}^*$),
11:
12:    # Algorithm 2
13:    $\boldsymbol{\beta}$ = ALR$^n_{\text{SGD}}$($\mathcal{D}, \mathcal{D}^{CV}, \boldsymbol{\theta}^*, \boldsymbol{\beta}, G$)
14:
15:    Return $\boldsymbol{\beta}, \theta^*$
16: end procedure
```

Algorithm 2 ALR^n Discriminative Passes

```
1: procedure ALR$^n_{\text{SGD}}$($\mathcal{D}, \mathcal{D}^{CV}, \boldsymbol{\theta}^*, \boldsymbol{\beta}, G$)
2:     # Algorithm 4
3:     $\eta_0$ = determineIntialStepSize($\mathcal{D}^{CV}$)
4:
5:     for iter = 1, . . . , 5 do
6:         # Pass $iter + 3$
7:         for i = 0, 1, . . . , N do
8:             for j = 0, 1, . . . , $|\mathcal{P}|$ do
9:                 $\eta_j = \frac{\eta_0}{\sqrt{G_j * G_j}}$, $g_j = \frac{\partial \text{NLL}(\mathbf{x}^{(i)};\boldsymbol{\beta})}{\partial \beta_j}$
10:                $\beta_j = \beta_j + \eta_j g_j$, $G_j = G_j + g_j$
11:            end for
12:        end for
13:    end for
14:    Return $\boldsymbol{\beta}$
15: end procedure
```

structure, i.e., the *tuples* which are ignored, their respective values are set to false. Once the $\mathcal{BM}$ is modified, *parameters* vector $\boldsymbol{\theta}$ is re-allocated. After re-allocation, a third pass is made for re-calculating the counts and computing the probabilities [5].

Once the generative learning is finished, the discriminative learning begins. For that two more *parameter* vectors of the same size as $\boldsymbol{\theta}$ (i.e., $|\mathcal{P}|$) are allocated: $\boldsymbol{\beta}$ and G. sALR^n relies on only limited passes of Stochastic Gradient-Descent (SGD). It is absolutely necessary for SGD to pick a good step-size. Typically, this value is tuned through cross-validation. sALR^n relies on adaptive step size method of AdaGrad [3]. In essence, it accumulates gradients in each direction and scale the step size by this sum. It, however, keeps the question of initial step size (η_0) open. sALR^n learns the value of η_0 by doing cross-validation line-search on $\mathcal{D}^{\mathcal{CV}}$. The details of *iterative line-search algorithm* can be found in Algorithm 4 and 5 in Appendix A.

The outline of the discriminative pass learning algorithm is given in Algorithm 2. The index *iter* is for the SGD iterations, the index i loops over the data points and the index j loops over the elements of the *parameter* vector. The partial derivative for data point $\mathbf{x}^{(i)}$ of the *parameter* j is denoted as g_j, and is computed as:

$$\frac{\partial \text{NLL}(\mathbf{x}^{(i)}, \boldsymbol{\beta})}{\beta_j} = (\mathbf{1}_y - \text{P}(y|\mathbf{x}))\mathbf{1}_{\mathcal{F}} \log \text{P}(\mathcal{F}|y),$$

where $\mathbf{1}_y$ is 1 if class label of $\mathbf{x}^{(i)}$ is same as that of *feature* indexed by j. Similarly, $\mathbf{1}_{\mathcal{F}}$ is 1 only if non-class attributes of $\mathbf{x}^{(i)}$ is same as that of *feature* indexed by j. Once the gradients are computed, they are used to update the corresponding *parameter* β_j and accumulated in G_j for adapting the step-size. The procedure terminates after five iterations over the data.

[5]Note, that this pass is optional, because one can always create a new *parameter* vector $\tilde{\boldsymbol{\theta}}$, and copy contents from old vector $\boldsymbol{\theta}$ to the new one $\tilde{\boldsymbol{\theta}}$.

3.1 Alternative Tuple Selection An extremely useful property of mutual information (MI) for selecting *tuples* is its computational efficiency. One can compute the necessary statistics and then evaluate *tuples* by just one quick pass over the data. It, however, is not the only potential measure for selecting the *tuples*. There are several alternatives, e.g., thresholding on the counts instead of the MI, hashing, etc. [2]. A far better alternative is to use the discriminative pass algorithm of sALR^n for *tuple* selection. Since looping over the entire data set will be expensive, one can easily train over much smaller dataset, i.e., $\mathcal{D}^{\mathcal{CV}}$, that resides in memory. Note $\mathcal{D}^{\mathcal{CV}}$ will be sliced into training, testing and validation set. This can be done once for all the *tuples*. In essence, we compute the score as:

$$\text{CVscore}(\mathcal{T}) = \text{ALR}^n_{\text{SGD}}(\mathcal{D}^{\mathcal{CV}}{}_{\neg\mathcal{T}}, \boldsymbol{\theta}^*, \boldsymbol{\beta}, G),$$

where $\mathcal{D}^{\mathcal{CV}}{}_{\neg\mathcal{T}}$ is the dataset $\mathcal{D}^{\mathcal{CV}}$ but with *tuple* $\mathcal{T}$ removed. The tuples are evaluated on the basis of their CVscore($\mathcal{F}$) score. Higher the score, more relevant the tuple is. Similar to MI, some t top tuples can be selected. The Algorithm 1 takes $\mathcal{TS}$ as an input parameter which actually specifies the *tuple* selection criterion. Also t is the threshold parameter.

3.2 Limitations of sALR^n It can be seen that sALR^n successfully alleviates the main shortcomings of the standard ALR^n method. It is out-of-core, does *tuple* selection and relies on a more efficient implementation. There, however, is a problem. The algorithm still relies on enumerating all the possible *tuples* (and associated *features* and *parameters*) to determine their relevance in the second stage. For many datasets, this is not a problem, the *tuple* selection passes are generally computationally inexpensive, and results in greatly speeding-up the discriminative learning by reducing the number of *parameters* to be optimized. However, for datasets with many attributes, it can be troublesome because the size of vector $\boldsymbol{\theta}$ vector will just be too big to fit in the memory. Therefore, sALR^n is limited to smaller values of n. How can we scale *tuple* selection to higher values of n? Let us now discuss a variant of sALR^n that hierarchically builds the *tuples* instead of enumerating them all.

4 Hierarchical Implementation – hsALR^n

Instead of enumerating all *tuples*, and then doing *tuple* selection, Hierarchical Selective (higher-order) Accelerated Logistic Regression – hsALR^n builds *tuples* hierarchically. Unlike sALR^n, hsALR^n starts by building n index-dictionaries and BitSet data structures – one for each level, since the *tuples* of order n will be build hierarchically in a bottom-up manner from lower orders. We will denote dictionary and Bitset at level i as $\mathcal{DICT}^i$ and $\mathcal{BM}^i$ respectively. The algorithm starts by building $\mathcal{DICT}^1$ and $\mathcal{BM}^1$ (similar to sALR^1).

An outline of the algorithm is given in Algorithm 3. Again, the first pass is similar to sALR^1, $\mathcal{BM}^1$ is updated and $\mathcal{D}^{\mathcal{CV}}$ is extracted. The memory is allocated for $\boldsymbol{\theta}$ and a second pass is made through the data to extract the count, followed by the *tuple* selection. Note, up till this stage, the behaviour is exactly similar to sALR^1. Next, it deviates. Now $i = 2$, and the second iteration of the loop begins. It initializes $\mathcal{BM}^2$. But, the intitialization are based on $\mathcal{BM}^1$ – i.e., the *tuples* which are not selected in *tuple* selection stage of $i = 1$ are ignored. In other words, the $\mathcal{BM}^2$ only constitutes of bits related to top t tuples. The similar process is repeated for $i = 3$ and so on. This way, at level n, the tuples are built hierarchically from 1 to n. A simple illustration of *tuple* building process is shown in Figure 1.

Once the *tuples* are built (engineered), memory is allocated for $\boldsymbol{\theta}$ based on $\mathcal{BM}^n$, and a final pass is made through the data to populate $\boldsymbol{\theta}$. Next, the algorithm follows the similar path as that of sALR^n, i.e., discriminative training.

4.1 ***Feature* Engineering** As an alternative to building *tuples* hierarchically, one can build *features* instead. This is motivated from the fact that the cardinality of *tuples* can vary significantly in some datasets. E.g., some attributes might have values in order of 10^6, while others in the order of 10^2. Of course, the resulting size of the model at $n > 1$ will not decrease significantly as long as one of these high-cardinality attribute is present in the *tuples*. This limitation can be addressed by selecting and building *features*, instead of *tuples*. A downside of this is that since there are many more *features* than *tuples*, the *feature* selection and engineering might take much longer than working with *tuples*.

Algorithm 3 Hierarchical Selective ALR^n

```
 1: procedure sHALR$^n(\mathcal{D}, \mathcal{TS}, t)$
 2:
 3:     Intialize $\mathcal{BM}^1$, Intialize $\mathcal{DICT}^1$
 4:     $[\mathcal{BM}^1, \mathcal{D}^{\mathcal{CV}}] \leftarrow$ explorationPass($\mathcal{D}$) # Pass 1
 5:     i = 1;
 6:     while i <= n do
 7:         if i > 1 then
 8:             Intialize $\mathcal{BM}^i$ from $\mathcal{BM}^{i-1}$
 9:         end if
10:         $\boldsymbol{\theta} \leftarrow$ allocateMemory($\mathcal{BM}^i$)
11:         $\boldsymbol{\theta} \leftarrow$getCountsFromData($\mathcal{D}, \mathcal{BM}^i$) # Pass i
12:         $\mathcal{BM}^i \leftarrow$ doTupleSelection($\mathcal{D}^{\mathcal{CV}}, \mathcal{BM}^i, \mathcal{TS}, t$)
13:         i $\leftarrow$ i + 1
14:     end while
15:
16:     $\boldsymbol{\theta}^* \leftarrow$ allocateMemory($\mathcal{BM}^n$),
17:     $\boldsymbol{\theta}^* \leftarrow$getCountsFromData($\mathcal{D}, \mathcal{BM}^n$) #Pass i + 1
18:     $[\boldsymbol{\beta}, G] \leftarrow$ allocateMemory($\mathcal{BM}^n$),
19:
20:     # Algorithm 2
21:     $\boldsymbol{\beta}$ = sALR$^n_{\text{SGD}}(\mathcal{D}, \mathcal{D}^{CV}, \boldsymbol{\theta}^*, \boldsymbol{\beta}, G)$
22:
23:     Return $\boldsymbol{\beta}, \boldsymbol{\theta}^*$
24: end procedure
```

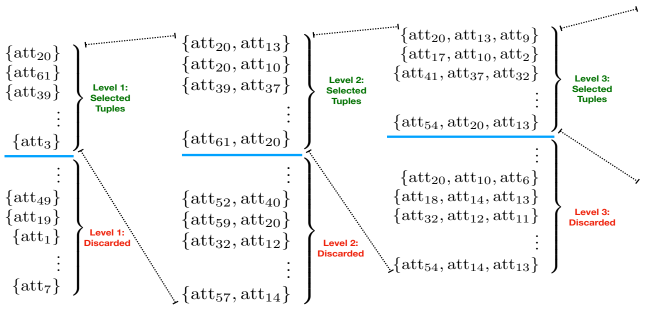

Figure 1: A schematic illustration of hsALR^n feature engineering process.

	#Instances	#Attributes	#Features	#Classes
Covtype	581012	13	333	7
SUSY	2461308	18	462	2
HIGGS	11000000	18	527	2
Activity	3850505	54	64574	19
Avazu	40428967	32	10266948	2

Table 1: Details of datasets.

5 Experimental Results

Let us compare the performance of our proposed algorithms on four datasets (`Covtype, SUSY, HIGGS, Activity`) from UCI repository [5] and one dataset (`Avazu`) from Kaggle repository [7]. The datasets were discretized before training using MDL discretiztaion [4] [6]. The details of the datasets can be found in Table 1. Note, since the labels for `Avazu` test files were not available, an 80 : 20 split of the training file has been done to create new training and test files. All experiments are computed on a standard computer with 64GB of RAM.

5.1 sALR^n vs s(K)DB Let us start by comparing the performance of sALR^n (with no *tuple* selection). A comparison of the 0-1 Loss performance of sALR^n with s(K)DB, naive Bayes and Random Forest with 100 trees is shown in Figure 2. The results are based on two rounds of two-folds cross-validation. Let us first establish the correspondence between sALR^n and s(K)DB in terms of the *order* of interactions they model. Note, that sALR^2 has *tuples* of length 2, i.e., it models interactions of the form: $\mathrm{P}(x_1, x_2|y)$. On the other hand, s(K=1)DB, models the interactions of similar order, that is: $\mathrm{P}(x_1|y, x_2)$. Therefore, if we have a function $\mathcal{Q}$ that returns the order of interactions, the following holds $\mathcal{Q}(\text{sALR}^n) = \mathcal{Q}(\text{s(n-1)DB})$.

It can be seen that in Figure 2, the results for s(K)DB and sALR^n are stacked together for matching order. We also have included AnJE results for comparison. Note, AnJE is the generative equivalent of ALR^n. The naive Bayes (NB) and Random Forest (RF) results are plotted as a horizontal lines for comparison. Note that A(n=1)JE $\equiv$ s(K=0)DB $\equiv$ NB. It can be seen from the results, that sALR^n is an effective limited pass learning algorithm that results in better performance than both s(K)DB and AnJE. On `SUSY` and `HIGGS` datasets, the results are even better than RF. It can be seen that, on `Covtype`, the performance of sALR^4 gets very close to RF, however, increasing the value of n to 5 results in out-of-memory (OOM) error. We will shown in Section 5.2 that even with $n = 4$ and with *tuple* selection, sALR^4 can lead to performance better than that of RF.

5.1.1 How to determine n? What is the best value of n for sALR^n? Note, s(K)DB has a sophisticated algorithm to determine the best value of K. Since, sALR^n is based on both generative and discriminative *parameters*, it can also rely on its (computationally efficient) generative counter-part AnJE to choose the best value of n. One can start from $n = 1$ and increment the value of n until the performance of AnJE is better on $\mathcal{D}^{\mathcal{CV}}$. Exploration of sophisticated algorithms to determine the best value of n has been left as a future work.

5.2 *Tuple* Selection for sALR^n A comparison of the 0-1 Loss performance of sALR^2, sALR^3 and sALR^4 by varying the threshold of *tuple* selection on `covtype` dataset is shown in Figure 3. *Tuple* selection is done with both mutual information (MI) and CVScore of Section 3.1. For sake of comparison, we also include sALR^n performance with no *tuple* selection as horizontal dotted line. Let us first analyze *tuple* selection with sALR^2 (red lines). It can be seen that at $n = 2$, MI is a better *tuple* selection method than CVScore. Also, at $t = 0.6$, it results in better performance than sALR^2. However, *tuple* selection with sALR^3 (yellow lines) and sALR^4 (blue lines) reveals that CVScore is a better criterion than MI, even though both criteria leads to better performance than sALR^3.

It can be seen that sALR^4 with CVScore based *tuple* selection can also result in better performance than RF (horizontal green line), an extremely encouraging result. A similar comparison is done for `SUSY` and `HIGGS` datasets in Figure 4.

5.2.1 How to determine t? Clearly, the optimal value of t is data specific. Since the main motivation behind *tuple* selection is to reduce the size of the model, it depends on available computational resources. However, it is clear from Figures 3 and 4 that *tuple* selection drastically improves the performance of sALR^n, and, therefore, it should not be avoided. Since MI and CVScore are fairly easy to compute, we recommend determining the best value of t using cross-validation.

5.3 Evaluating hsALR^n Let us evaluate hsALR^n in this section. We will compare the 0-1 Loss performance, Training Time and no. of *parameters* of ALR^n, sALR^n and hsALR^n models.

Let us compare the no. of *parameters* optimized

[6]Note, MDL discretization requires loading the data in memory. Since some of the datasets, e.g, `Activity` and `Avazu` were too big to be loaded into memory, the discretization was done attribute by attribute, i.e, a single attribute was only loaded into memory, discretized and the output was written back to the file. A similar procedure was repeated for all the attributes.

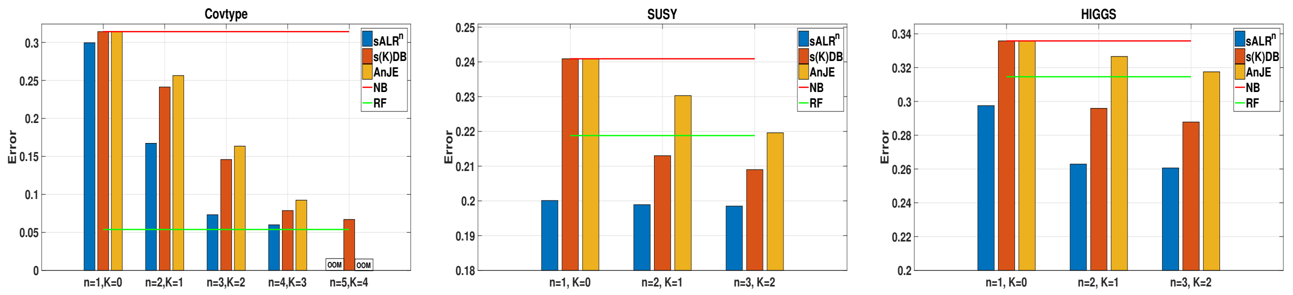

Figure 2: A comparison of the 0-1 Loss of sALR^n, s(K)DB and A(n)JE classifiers on `covtype`, `SUSY`, `HIGGS` datasets.

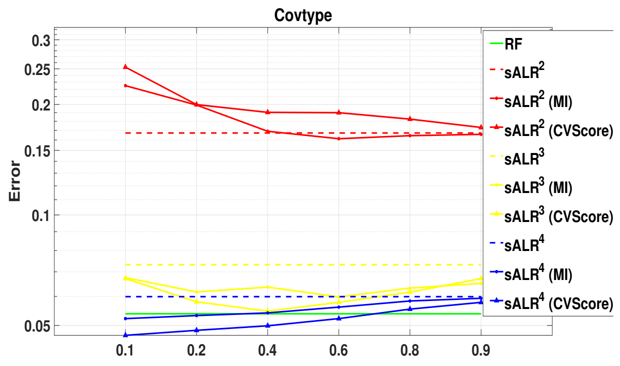

Figure 3: A comparison of 0-1 Loss of sALR^2 (MI), sALR^2 (CVScore), sALR^3 (MI), sALR^3 (CVScore), sALR^4 (MI) and sALR^4 (CVScore) by varying the *tuple* selection threshold. RF, sALR^2, sALR^3 and sALR^4 are also plotted as horizontal line for comparison.

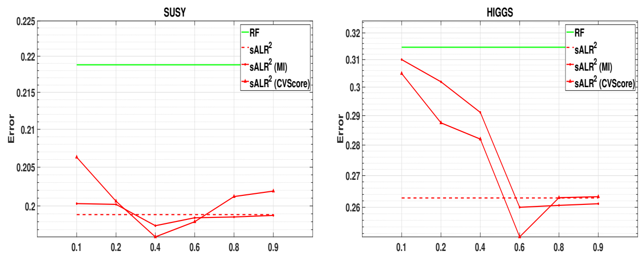

Figure 4: A comparison of 0-1 Loss of sALR^2 (MI), sALR^2 (CVScore) by varying the *tuple* selection threshold. RF, sALR^2 are also plotted as horizontal line for comparison.

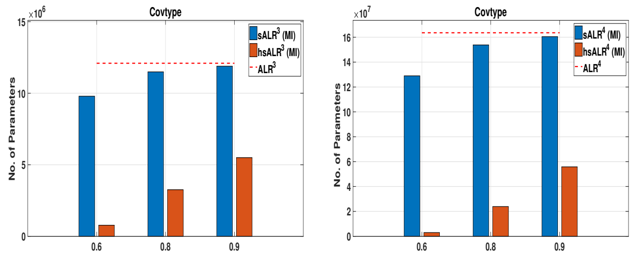

Figure 5: A comparison of the no. of *parameters* of ALR^n, sALR^n and hsALR^n with different thresholds of *tuple* selection. Left: $n = 3$, Right: $n = 4$.

by these three models in Figure 5 on `Covtype` dataset (using MI based *tuple* selection with a threshold of $0.6, 0.8$ and 0.9). It can be seen that both hsALR^3 and hsALR^4 results in greatly reducing the number of *parameters*. It is important to remember that hsALR^n is a bottom-up approach, i.e., it starts by building *parameters* from an empty set. On the other hand, sALR^n is top-down, i.e., it starts with the full ALR^n model and reduces its size.

A comparison of the training time of the three models is given in Figure 6. It can be seen that, hsALR^n results in greatly reducing the training time of sALR^n and ALR^n. Of course this is due to a reduced number of training *parameters*. Let us now see the effect of these reduced no. of *parameters* and faster training time on the 0-1 Loss results. The 0-1 Loss comparison is shown in Figure 7. Well, it can be seen that, except for $t = 0.6$ for $n = 3$, there is not a huge difference in the performance of sALR^n and hsALR^n. With greatly reduced number of *parameters*, faster training time, this is very encouraging result.

A similar comparison in terms of the no. of *parameters*, training time and the error is done for `SUSY` and `HIGGS` datasets in Table 2. Note, due to space constraints, only results are shown for threshold $t = 0.9$ with MI.

5.4 *Tuple* vs. *Feature* Selection for hsALR^n

So far, our evaluation has been constrained to *tuple* selection. Let us now focus on extremely big datasets of Table 1 – `Activity` and `Avazu` [7]. Simple ALR^2 on these datasets will lead to a model too big to be processed by standard computers. In fact, s(K=1)DB

[7]Out of around 30 attributes, there are two attributes in this dataset with over 10^6 values. This property makes it suitable for *feature* selection rather than *tuple* selection.

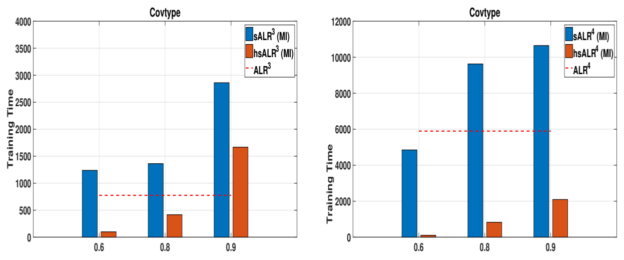

Figure 6: A comparison of the training time of ALR^n, sALR^n and hsALR^n with different thresholds of *tuple* selection. Left: $n = 3$, Right: $n = 4$.

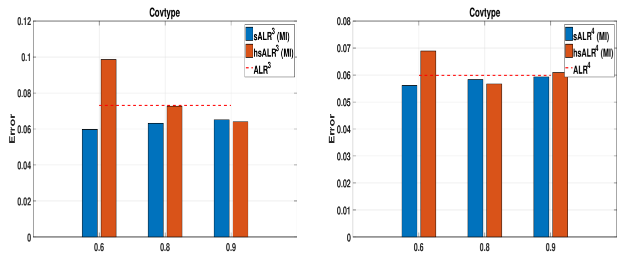

Figure 7: A comparison of 0-1 Loss performance of ALR^n, sALR^n and hsALR^n with different thresholds of *tuple* selection. Left: $n = 3$, Right: $n = 4$.

and RF (even with 10) trees resulted in out-of-memory (OOM) error on our experimentation hardware. We compare the performance of different models on these two datasets in Table 3. The results are obtained in a train-test scheme. An 80 : 20 split of the available data is used to create training and test files. Two versions of hsALR^n are reported. hsALR^n is the standard one with *tuple* selection and hsALR^n(∗) is the one with *feature* selection.

It can be seen that on the *Activity* dataset, the best result is obtained by sALR^2 with *tuple* selection, resulting in an error of 0.0011 ($t = 0.2$ with MI). Both hsALR^2 and hsALR^2(∗), results in reducing the size of the model, but results in slightly worst performance than sALR^2.

For the `Avazu` dataset, only results are reported for NB, ALR^1, and hsALR^2(∗) ($t = 0.1$ with MI). Other techniques could not be used because of the presence of very high cardinality features in this dataset. It can be seen that hsALR^2(∗) leads to the best result of 0.1219.

6 Conclusion and Future Works

In this paper, we proposed two simple yet effective algorithms for learning from extremely large data quantities. The algorithms are motivated from the need of a better feature engineering capability, out-of-core data processing and minimal tuning parameters. Our proposed algorithm sALR^n is based on standard accelerated higher-order logistic regression (ALR^n). It adds the functionality of *tuple* and *feature* selection and out-of-core limited pass learning. The second proposed algorithm hsALR^n hierarchically builds *tuples* and *features*. We show that the resulting algorithms lead to better (classification error) performance than the current state-of-the-art in out-of-core data processing and also competitive with state-of-the-art in-core classifier Random Forest. There are many exciting new directions from this work:

- The results reported in this work are not regularized. The reason for not regularizing is because, it opens the question of determining the value of the regularization parameter (λ). Of course, the extent of regularization depends on the value of n, the number of iterations and many other factors. Integrating a mechanism for choosing the value of λ has been left as a future work. Technique proposed in this work can be used [12].

- There is also need for automatically selecting the best value of threshold – t, at least for relatively smaller datasets.

- Performance of the model is also tied to the number of SGD iterations (T). Setting T to five is rather arbitrary, but has been done to minimize the training time. Clearly, there is a need to automatically tune these parameters to the constraints of a spe-

	ALR^2	**sALR^2**	**hsALR^2**	**ALR^2**	**sALR^2**	**hsALR^2**
	SUSY			HIGGS		
#Parameters	272k	256k	**108k**	258k	243k	**121k**
Training Time	**460**	1287	919	2316	3219	**2200**
Error	0.1989	**0.1985**	0.1988	0.2629	0.2620	**0.2600**

Table 2: A comparison of no. of *parameters*, training time and 0-1 Loss performance of ALR^2, sALR^2 and hsALR^2 with $t = 2$ of *tuple* selection.

	NB	**ALR^1**	**ALR^2**	**sALR^2**	**hsALR^2**	**hsALR^2(∗)**
	Activity					
#Parameters	1.2M	1.2M	250M	**136M**	111M	101M
Training Time	52	1596	3121	2771	**2300**	6511
Error	0.092	0.0115	0.0012	**0.0011**	0.0030	0.0045
	Avazu					
#Parameters	**20M**	20M	OOM	OOM	OOM	280M
Training Time	**189**	2321	OOM	OOM	OOM	10199
Error	0.3004	0.1732	OOM	OOM	OOM	**0.1219**

Table 3: A comparison of the no. of *parameters*, training time and 0-1 Loss performance of NB, ALR^1, sALR^2, hsALR^2, hsALR^2(∗) and FM^2.

cific application.

- Lastly, the three models proposed in this work: sALR^n, hsALR^n and hsALR^n(*), will result in different size models as the threshold parameter t varies. Clearly there is a need to systematically evaluate the performance of each by varying t, and making sure that the size of the model is the same.
- Applicability of proposed algorithms to smaller quantities of data needs to be explored as well.

A Iterative Line Search

The Algorithm 4 determines the best η_0 based on a simple line-search procedure. The algorithm starts with a large interval 10^6 and 10^{-6} and evaluates the performance of the algorithm with different values within the interval. At each iteration, it finds the best interval and applies the algorithm recursively to find an optimal value of the step size. The process continues until the difference in the performance is less than user-specified value – ϵ, which is fixed to 10^{-2}. An outline of the evaluateLineSearch() function is given in Algorithm 5. The EvaluateFunction() does a 90 : 10 split of $\mathcal{D}^{\mathcal{CV}}$. It learns a classifier (using the learning procedure of Algorithm 2) on the 90% of the data with step size of $10^{\alpha[i]}$ and test on the remaining 10%. The RMSE performance is returned.

Algorithm 4 Determine-Initial-Step-Size

```
1: procedure DETERMINEINTIALSTEPSIZE($\mathcal{D}^{\mathcal{CV}}$)
2:     $\epsilon = 0.01, High = 6, Low = -6, Scale = 10$
3:     while ($|P_A - P_B| > \epsilon$) do
4:         # Algorithm 5
5:         $P_A, P_B, high, low$ = evaluateLineSearch($\mathcal{D}^{\mathcal{CV}}, High, Low, Scale$)
6:         $High \leftarrow high$, $Low \leftarrow low$
7:     end while
8:     Return $\eta_0 = (10^{High} + 10^{Low})/2$
9: end procedure
```

Algorithm 5 Evaluate-Line-Search

```
1: procedure EVALUATELINESEARCH($\mathcal{D}^{\mathcal{CV}}, High, Low, Scale$)
2:     Intitialize $\alpha$, $P$ be a vector size ($Scale + 1$)
3:     $\rho = \frac{High - Low}{Scale}$
4:     $\alpha[0] \leftarrow Low$, $\alpha[Scale] \leftarrow High$
5:
6:     for $i = 1 \rightarrow$ size($\alpha$) do
7:         $\alpha[i] \leftarrow \alpha[i-1] + \rho$
8:         $P$[i] $\leftarrow$ EvaluateFunction($\mathcal{D}^V$, $10^{\alpha[i]}$)
9:     end for
10:    $j \leftarrow$ smallestInVector($P$)
11:
12:    Return $P[j-1], P[j+1], \alpha[j-1], \alpha[j+1]$
13: end procedure
```

References

[1] Brain, D., Webb, G.I.: The need for low bias algorithms in classification learning from small data sets. In: PKDD, pp. 62–73 (2002)

[2] Chapelle, O., Manavoglu, E., Rosales, R.: Simple and scalable response prediction for display advertising. ACM Transactions on Intelligent Systems and Technology **5**(4) (2015)

[3] Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. **12**, 2121–2159 (2011)

[4] Fayyad, U.M., Irani, K.B.: On the handling of continuous-valued attributes in decision tree generation. Machine Learning **8**(1), 87–102 (1992)

[5] Frank, A., Asuncion, A.: UCI machine learning repository (2010). URL `http://archive.ics.uci.edu/ml`

[6] Ganz, J., Reinsel, D.: The Digital Universe Study (2012)

[7] Juan, Y., Zhuang, Y., Chin, W., Lin, C.: Field-aware factorization machines for CTR prediction. In: ACM RecSys (2016)

[8] K, W., A, D., J, L., A, S., J, A.: Feature hashing for large scale multitask learning. In: ICML (2009)

[9] Martinez A. Chen, S., Webb, G.I., Zaidi, N.A.: Scalable learning of Bayesian network classifiers. Journal of Machine Learning Research **17**, 1–35 (2016)

[10] Provost, F.: A survey of methods for scaling up inductive algorithms. Data Mining and Knowledge Discovery **3** (1999)

[11] Qiu, J., Wu, Q., Ding, G., Xu, Y., Feng, S.: A survey of machine learning for big data processing. EURASIP Journal on Advances in Signal Processing **67** (2016)

[12] Rendle, S.: Learning recommender systems with adaptive regularization. In: ACM International Conference on Web Search and Data Mining, pp. 133–142 (2012)

[13] Zaidi, N.A., Carman, M.J., Cerquides, J., Webb, G.I.: Naive-Bayes inspired effective pre-conditioners for speeding-up logistic regression. In: IEEE International Conference on Data Mining, pp. 1097–1102 (2014)

[14] Zaidi, N.A., Webb, G.I.: Fast and efficient single pass Bayesian learning. In: Advances in Knowledge Discovery and Data Mining, pp. 149–160 (2012)

[15] Zaidi, N.A., Webb, G.I.: A fast trust-region Newton method for softmax logistic regression. In: SDM2017: SIAM International Conference on Data Mining, pp. 705–713 (2017)

[16] Zaidi, N.A., Webb, G.I., Carman, M.J., Petitjean, F., Cerquides, J.: ALR^n: Accelerating higher-order logistic regression. Machine Learning **104**, 151–194 (2016)