Improving Stochastic Gradient Descent Initializing with Data Summarization

Robin Varghese[1] and Carlos Ordonez[1]

Department of Computer Science
University of Houston, Houston TX 77204, USA

Abstract. Linear Regression (LR) is the prototypical statistical model, which can be applied on a wide range of predictive problems. Ordinary Least Squares (OLS), is the standard technique for estimating the parameters of the LR model. However, such computation can be slow and resource-hungry for large data sets with high dimensionality, due to heavy matrix operations. More importantly, OLS may be impractical for large data sets as the entire data set is required to be loaded into main memory, often exceeding RAM capacity. These limitations emphasize the need for optimization techniques to compute LR. Two state of the art algorithms used to compute LR are: Stochastic Gradient Descent (SGD) and Data Summarization (DS), combined with a matrix factorization. A few decades ago DS was the main technique to accelerate data mining computations, followed by SGD. Nowadays, SGD has become the workhorse behind most ML algorithms and deep neural networks. Merging both techniques, we propose to initialize SGD with a solution computed via DS on the initial batch of points, leaving SGD computation on the remaining points "as is". An experimental evaluation with several data sets shows our improved SGD algorithm reaches higher quality solutions (lower MSE error, higher R^2) and it converges faster (less iterations, reduced data usage, less computation time). We believe our simple SGD change can benefit many more ML models beyond LR.

1 Introduction

Linear Regression is the basis for linear models. LR is highly regarded for its simplicity and ease of implementation, which makes it accessible to users with varying levels of technical expertise. This includes researchers, analysts, and practitioners in various fields, such as finance, healthcare, social sciences, and engineering. The intuitive nature of LR also makes it useful for educators and students seeking to understand fundamental concepts in statistics and data analysis. The need to accelerate the computation of LR arises because it can be computationally expensive, particularly when dealing with large data sets. This is because LR involves solving a system of equations to find the optimal values of the coefficients, that minimize the sum of the squared errors. Furthermore as the size of the data set increases, the number of calculations required to solve these equations also increases, leading to long computation times and slow performance.

Gradient descent (GD) has been making significant strides in the field of machine learning (ML) also for its simplicity, but more importantly its ability to be applied to a wide variety of problems [15]. It can be said to be the definitive optimization algorithm in ML. Logistic regression builds on the concepts of LR and is a step towards creating non-linear models. Neural networks, used in both statistical models, have their quality and speed affected by GD's initialization. Many aspects and properties of GD for LR have been extensively researched including, optimal learning rates, learning rate schedulers, cost functions and data shuffling [4, 10, 14]. Furthermore from [16], we see how initialization can have significant impact on model performance. Compared to the traditional base case initializations of 0 and 1, we can see from [3] how random initialization can result in better model performance. In this paper we propose using data summarization as weight initialization. Additionally there are many variations of GD, most commonly Stochastic Gradient Descent (SGD) and mini-batch SGD. GD is computed on the entire data set, SGD on a single point, and mini-batch SGD on a batch of data. Mini-batch SGD is often favored because it is a balance between GD and SGD. This paper implements mini-batch SGD, but is referred to throughout the paper as simply SGD.

Gamma (Γ) is but one form of DS and can be used to integrate sufficient statistics (SS) of an input data set into a single matrix. Then, the computation of many ML models and tasks such as Naive Bayes, Principal Component Analysis (PCA), Linear Discriminant Analysis and Linear Regression can be accelerated [11]. The authors of [1], compute Γ in Python and have shown to be as fast as popular Python ML libraries including scikit-learn (sklearn) [13].

Our contributions and experimental findings show that, using data summarization for initialization in gradient descent can result in:

1. Faster convergence with less data
2. Lower model error
3. Higher quality (R^2)

The paper is organized as follows: Section 2 presents background information, key concepts, definitions, and notations that are mentioned throughout the paper. Section 3 details the system and theoretical aspects of the underlying related algorithms and our contribution. Section 4 provides an overview of the experimental setup, implementation details, results, and analysis. In Section 5, we explore related works in gradient descent optimizations and initialization techniques. Finally, in Section 6 we give concluding remarks and discuss the potential for future research.

2 Definitions

2.1 Input data set

Each input data set is defined as X. Here, X is a $d \times n$ matrix, equivalent to a set of n column vectors corresponding to each observation in the data set, having d

dimensions or attributes. In the case of LR, augment X with a row vector of n ones along the X_0 dimension producing the $(d+1) \times n$ matrix $\mathbf{X}$. $\mathbf{Y}$ is a n row vector corresponding to the output for each n observation.

2.2 LR Model

Given a $(d+1) \times n$ input matrix $\mathbf{X}$, and a $(d \times 1)$ column vector $\hat{\beta}$ of coefficients, compute $\hat{\mathbf{Y}}$. $\hat{\mathbf{Y}}$ is a n row vector, corresponding to the predicted outputs of each n observation, produced by the LR model.

$$\hat{\mathbf{Y}} = \hat{\beta}^T \mathbf{X} + \epsilon \tag{1}$$

To achieve a close approximation of $\mathbf{Y}$ the true predicted outputs, fit the LR model Eq. 1. The model can be fit using many methods most commonly, Ordinary Least Squares (OLS). This method finds the $\hat{\beta}$ that minimizes the Residual Sum of Squares (RSS), resulting in $\hat{\mathbf{Y}} \approx \mathbf{Y}$ [5].

$$RSS = (\mathbf{Y} - \hat{\beta}^T \mathbf{X})(\mathbf{Y} - \hat{\beta}^T \mathbf{X})^T \tag{2}$$

To minimize RSS, differentiate w.r.t. $\hat{\beta}$ and set equal to 0, deriving the unique solution:

$$\hat{\beta} = (\mathbf{X}\mathbf{X}^T)^{-1} \mathbf{X}\mathbf{Y}^T \tag{3}$$

Given the trained coefficients $\hat{\beta}$ from fitting the model in Eq. 3, the model can be utilized to make predictions on new or future observations.

3 System and Algorithms

3.1 Gamma Summarization (Γ)

For computing gamma (Γ), augment $\mathbf{X}$ with $\mathbf{Y}$. In general, this $(d+2) \times n$ matrix is defined as $\mathbf{Z}$, but in practice is $(d+2) \times c$, where c is equivalent to chunk size. It should be emphasized that c is considered a hyper-parameter and can vary to achieve optimal performances. However for our experiments c is held constant to give a fair comparison between the algorithms, Γ and SGD. The terms $\mathbf{X}\mathbf{X}^T$ and $\mathbf{X}\mathbf{Y}^T$ in Eq. 3 and other sufficient statistics (SS), are contained in the single matrix Γ. Therefore the computation of Γ becomes a two phase algorithm as follows, Phase 1: Obtain SS stored in Γ; Phase 2: Solve $\hat{\beta}$ exploiting SS.

$$\mathbf{Z} = \begin{bmatrix} 1 & 1 & ... & 1 \\ x_{11} & x_{12} & ... & x_{1c} \\ x_{d1} & x_{d2} & ... & x_{dc} \\ y_{11} & y_{12} & ... & y_{1c} \end{bmatrix} \tag{4}$$

Phase 1 Begin by reading one chunk of the input data set of size $d \times c$. Augment this chunk with a row vector of c ones and $\mathbf{Y}$ (dependant variable) also a c row vector. This will produce $\mathbf{Z}$, a $(d+2) \times c$ dense matrix. Compute $\mathbf{Z}\mathbf{Z}^T$ to produce the partial gamma Γ_i. It should be noted Γ is computed on the entire data set, but because only the first chunk is used, we utilize partial gamma Γ_i to initialize SGD.

$$\Gamma_i = \mathbf{Z}\mathbf{Z}^T = \begin{bmatrix} n & L^T & \mathbf{1}^T \cdot Y^T \\ L & Q & XY^T \\ Y \cdot \mathbf{1} & YX^T & YY^T \end{bmatrix} \tag{5}$$

The sufficient statistics (SS) L, n and Q are defined as follows: $n = |X|$, $L = \sum_{i=1}^{n} x_i$, and $Q = XX^T = \sum_{i=1}^{n} x_i \cdot x_i^T$, n is total number of points in the data set, L is the linear sum of x_i and Q is the sum of vector outer products of x_i. It should be noted that Phase 1 takes majority of the computation time, between the two phases.

Phase 2 Begin Phase 2 by exploiting the sufficient statistics integrated into the single matrix Γ_i to further compute a ML model or task. In this case, the task is to compute the regression coefficients $\hat{\beta}$ also seen in Eq. 3. The sufficient statistic Q is exploited by substituting it into the OLS analytical solution (Eq. 3) as shown in Eq. 6. It should be emphasized that the initial chunk is no longer needed and is summarized within the significantly smaller matrix Γ_i. This allows $\hat{\beta}$ to be solved in main memory in $O(d^3)$.

$$\hat{\beta} = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{Y}^T = Q^{-1}(\mathbf{X}\mathbf{Y}^T) \tag{6}$$

3.2 Mini-Batch SGD

The closed-form solution of LR as shown in Eq. 3, is computationally expensive. An alternative would be to implement gradient descent, where MSE is iteratively optimized until reaching a local minimum Eq. 7,8. In mini-batch SGD, Mean Squared Error (MSE) is typically used over RSS because MSE averages the squared errors, providing normalization that makes training more scalable. As the number of data points increases, there are more residuals to square and sum, leading to large RSS values regardless if the errors are relatively small. This means that even if a model is making relatively good predictions, the RSS value can still be large simply due to the large number of data points. Mathematically, MSE is equivalent to: MSE = RSS / c

$$f = MSE = \frac{1}{c}(\mathbf{Y} - \hat{\beta}^T\mathbf{X})(\mathbf{Y} - \hat{\beta}^T\mathbf{X})^T \tag{7}$$

After each chunk, obtain the gradient ∇f by taking the partial derivative w.r.t $\hat{\beta}$ of Eq. 7, resulting in Eq 8.

$$\nabla f = \frac{1}{c}\sum_{i=1}^{c} -2x_i(y_i - (\hat{\beta}x_i + \hat{\beta}_0)) \tag{8}$$

$$\hat{\beta} = \hat{\beta} - \alpha \nabla f \tag{9}$$

By utilizing the gradient for the processed chunk and scaled by the learning rate α, update $\hat{\beta}$ with its new value. After each update, $\hat{\beta}$ iteratively minimizes the models loss until reaching convergence.

3.3 Mini-Batch SGD Initialization Using Gamma

Our contribution improves SGD by incorporating Γ as initialization. Although this may seem like a small and simple alteration, it has yielded significant and impactful results. Depending on the initialization method, the number of updates required to reach convergence can vary greatly. Therefore, we propose computing Γ on a chunk and using the computed $\hat{\beta}$ to initialize Mini-Batch SGD. In general, the algorithm will take an input data set X of d features, n observations, and the corresponding outputs for each observation $\mathbf{Y}$. As output, the trained $\hat{\beta}$'s, R^2, and MSE are returned. The outline of our proposed approach, as seen in Figure 1, is as follows:

1. Read one chunk of data
2. Compute Γ_i (Phase 1)
3. Exploit SS to compute $\hat{\beta}$ (Phase 2)
4. Initialize Mini-Batch SGD weights using the previously computed $\hat{\beta}$ from step 3
5. Begin model training using the initial and remaining chunks

It is important to note that Γ is computed only on the first chunk. Then Mini-Batch SGD begins training as standard, continuously and sequentially reading in chunks until convergence. Additionally, Γ can be computed sequentially and also more favorably, in parallel [11]. This highlights another aspect where Γ can help aid in Mini-Batch SGD.

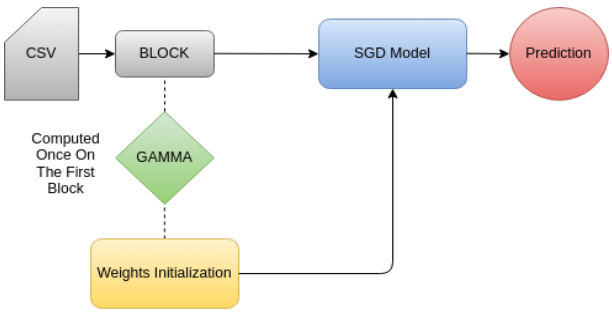


Fig. 1: Gamma Initialization System Design

4 Experiments

4.1 Experimental Setup

The YearPredictionMSD data set was obtained from the publicly available, UCI machine learning repository [7]. The creators of the YearPredictionMSD data set [2], created it to serve as a benchmark data set for regression tasks in ML, particularly for the problem of predicting the release year of a song, given a set of musical features. The data set was released as part of the Million Song Dataset project, which aimed to provide a comprehensive data set for music analysis and recommendation systems. The data set consists of $n = 515,345$ songs and $d = 90$ features. Each song is represented by a 90-dimensional feature vector, which includes information such as tempo, timbre, and loudness. The goal of the regression task is to predict the year in which the song was released, given the 90-dimensional feature vector. Additionally as stated in the original data set, the following train/test split should be respected to avoid the 'producer effect' (making sure no song from a given artist ends up in both the train and test set):

1. train: first 463,715 examples
2. test: last 51,630 examples

The California Housing data set can be obtained directly from sklearn (`sklearn.datasets.fetch_california_housing`). The data set is also a widely used LR benchmark in machine learning. It contains information on the median house prices, the number of households, the median income, and other factors in various neighborhoods across California. It was collected from the 1990 U.S. Census, and consists of $n = 20,640$ census block groups and $d = 10$ features [12]. A census block group is the smallest geographical unit the Census Bureau uses to collect data. The target variable for the model is the median house value (in units of 100,000 dollars). Since no train/test split is given by the authors, we use sklearns `sklearn.model_selection.train_test_split` method:

1. train: 70%
2. test: 30%

Table 1: Data sets

Data set	**Description**	n	d
YearPredictionMSD	Song Year Prediction	515,345	90
California Housing	Median House Value	20640	9

Using k-fold cross-validation may not be necessary for comparing different initialization methods of SGD because cross-validation is typically used to assess how well a model performs on new, unseen data. However, when testing initialization methods, the main goal is to compare their effects on the optimization process of SGD, rather than evaluating the model's performance on unseen data.

The system used for the experiments is a Pentium(R) Quadcore CPU running at 1.60 GHZ, 8 GB RAM, 1TB storage and with Linux Ubuntu as the operating system. Pre-processing and standardizing the data is important in gradient descent to prevent under/overflows occurring resulting in inaccurate results. Standardization is a common pre-processing step used in machine learning to transform data so that it has a mean of 0 and a standard deviation of 1. We utilize sklearns StandardScaler() on the data set to achieve this. To read chunks of the .csv, we use the Pandas data frame library [9]. Pandas is a powerful tool for data analysis and manipulation in Python. It provides a fast and efficient way to handle data in a variety of formats, including CSV, Excel, SQL databases, and more.

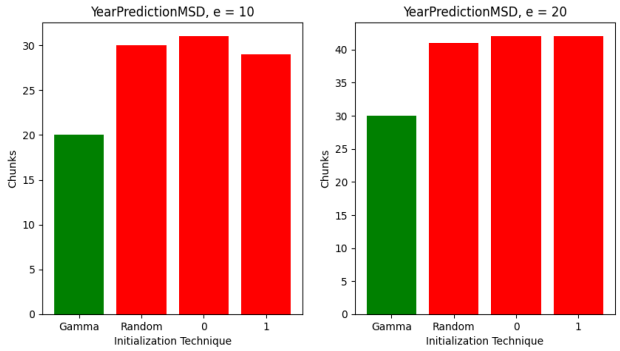
We use sklearn's SGDRegressor as an honest SGD implementation. Sklearn's SGD, internally has many optimizations and automatic hyper-parameter tuning. Most importantly, the initial learning rate, learning rate scheduler, and early stopping. For fair comparisons between initialization methods, we keep all hyper-parameters constant between trials and default to what sklearn provides. This includes using the default learning rate and default inverse scaling scheduler, in addition to early stopping. Only the initialization method is changed before training begins. Early stopping is a regularization technique commonly used in SGD to prevent overfitting and improving generalization. Overfitting occurs when the model learns the noise in the training data and fails to generalize well to new, unseen data. In our experiments, when the model does not achieve a lower error (MSE) after e steps, we stop training. We tested using 5, 10, and 20 steps however we only include 10 and 20 in this paper as there were little to no change in accuracy. This highlights one of the main benefits of SGD over OLS. That is, being able to approximately reach the optimal accuracy without requiring training on the entire data set. Once the weights are computed by Γ, we are easily able to initialize the sklearn model using the `reg.coef_` attribute. For 0 and 1 initialization, we set the `reg.coef_` attribute respectively. For random initialization, we simply initialize the sklearn SGD object and begin the training loop. In the case of Γ initialization, we use the first initial chunk to compute Γ, and then set `reg.coef_` using the produced β's. Additionally, we used 1% of the total data set as chunk size for both the initial Gamma chunk and SGD chunks. In the case of the YearPredictionMSD data set, chunk size was 5153, 1% of 515,345. For the California Housing data set, chunk size was 144 1% of 14,448. Due to the randomness factor in SGD, the results are taken as the average over 100 trials in addition to the experiment being ran multiple times with identical averages each time.

4.2 Experimental Results

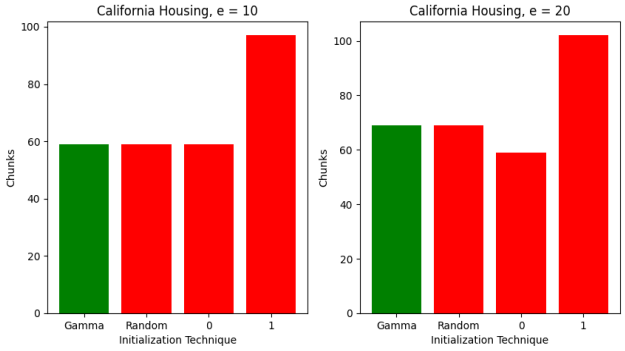
The YearPredictionMSD data set is a challenging regression problem, as the relationship between the musical features and the release year is complex and non-linear. This can be verified by the low R^2 value (0.39) produced by computing OLS on the entire data set. The coefficient of determination (R^2), is a commonly used metric in LR because it measures the proportion of variance in

Table 2: R^2 Results For Initialization (YearPredictionMSD: $R^2 = 0.39$, California Housing: $R^2 = 0.59$), e = early stopping steps

Data set	e	Gamma SGD	Random SGD	0 SGD	1 SGD
YearPredictionMSD	10	**0.26**	0.23	0.23	0.23
California Housing	10	**0.60**	0.58	0.58	0.56
YearPredictionMSD	20	**0.27**	0.24	0.23	0.23
California Housing	20	**0.60**	0.58	0.58	0.56



(a) YearPredictionMSD

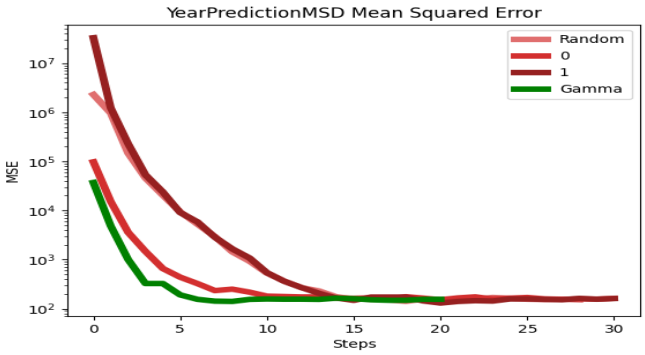


(b) California Housing

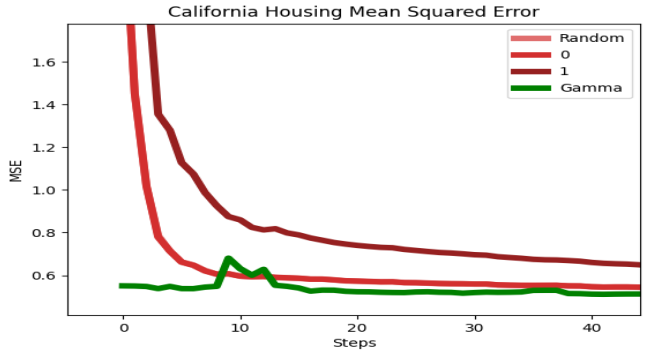
Fig. 2: Number of chunks used in training

Table 3: Speed Comparisons (Time in seconds)

Data set	OLS	Gamma SGD	Random SGD	0 SGD	1 SGD
YearPredictionMSD	26.517	**4.354**	5.583	5.694	5.516
California Housing	**0.942**	1.152	1.174	1.111	1.258



(a) YearPredictionMSD



(b) California Housing

Fig. 3: Model MSE during training

the dependent variable explained by the independent variables included in the model. An R^2 score of 1.0 indicates that the model perfectly fits the data. On the contrary, the California Housing data set has a relatively higher R^2 value (.59), indicating much more linearity in comparison.

Discussion Our analysis of the performance of different models on the YearPredictionMSD data set, as shown in Table 2 and Figure 2 (a), demonstrates that Gamma outperforms other models in terms of R^2 value despite using significantly less data. Whereas randomization the second best performing model used 30 chunks, Gamma used only 20 showing a 1.5x (30/20) increase in training efficiency. This can be attributed to Gamma's better ability to handle non-linear and noisy data. As previously mentioned, the YearPredictionMSD data set is challenging due to its low optimal (OLS) R^2, which makes it difficult for linear models to learn effectively. Therefore, the relative improvements in R^2 and data usage achieved by Gamma is noteworthy.

Conversely our results on the significantly more linear data set California Housing, across the models are identical. While Gamma does achieve a higher R^2, from Figure 2 (b) it is clear there is no longer a major improvement in training efficiency. This highlights that while Gamma provides advantages for non-linear data sets, it also does not perform worse than other methods on linear data sets.

We also verify SGD's efficiency over OLS in Table 3. On the YearPrediction-MSD data set, we can see OLS taking a substantial 26.5 seconds to compute. Using Gamma initialization, SGD is able to converge in 4.3 seconds, a speed up of approximately 6x (26.5/4.3). In order to identify potential findings, trends, or avoiding local minima and to ensure that each initialization technique received a fair evaluation, we conducted initial experiments with a maximum of 5 steps for early stopping. However, the resulting accuracy's were below acceptable levels with all models reaching below R^2=0.18. We then increased the maximum number of steps to 10, which led to a significant improvement in accuracy, approaching optimality. However, our experimental findings show that increasing the number of steps further to 20 did not result in any noticeable performance improvements, as the performance of the initialization techniques remained similar. We also did not include the plot for each models MSE when e=20 steps for this reason, as the trend is same. From Figure 3, the intuition behind Gamma initialization becomes clear. SGD is able to begin its descent from a smaller peak and eventually reach convergence.

5 Related Work

In this section, we overview previous work in SGD optimizations and weight initialization.

SGD Optimizations: Although gradient descent algorithms are widely used, it is still often viewed as a black-box optimizer. As a result, research in gradient descent has produced many improvements in the algorithm.

Optimal hyper-parameter tuning is vital during training and can lead to significant improvements [16]. Specifically, the authors show the importance of momentum and initialization in performance. Momentum is a fundamental method in learning rate scheduling. Intuitively, momentum will accelerate the progress towards the steepest descent thereby, accelerating convergence in comparison to a constant learning rate. Additionally ADAGRAD and ADADELTA are first and second order methods respectively that improve upon momentum and are commonly used [8] [17]. ADAGRAD has been shown to be well-suited to sparse data sets as the learning rate adapts towards the frequency of parameter updates. Its extension ADADELTA, addresses the problem of ADAGRAD's aggressive decreasing of learning rate.

In addition to hyper-parameter tuning, research has also produced many variants in the overall gradient descent algorithm. Depending on the training environment, modifications to the algorithm such as batch size, density, or sparsity can be leveraged to improve performance [15].

Initialization: Weight initialization is a vital step before training of a neural network begins. During training, the weights are repeatedly updated until the models loss or error converges to a minimum value. Therefore, weight initialization directly affects the convergence or training time of a model [6]. For its simplicity and versatility, randomization is a well known technique that can be found all across machine learning [3].

6 Conclusions

SGD is a building block for complex machine learning algorithms such as neural networks. Improving initialization of SGD can lead to faster and more accurate predictions. In this work, we propose a method for using DS to accelerate the computation of SGD. Our results show that this method can lead to faster convergence with less data, higher R^2 values, and lower error (MSE) in LR models. We provide experimental results on the YearPredictionMSD data set, a challenging regression problem with a complex and non-linear relationship between the musical features and the release year. Our analysis demonstrates that Γ initialization outperforms other models in terms of R^2 value, particularly for non-linear and noisy data. Additionally, our results on the more linear California Housing data set show that Γ initialization is equally effective as other methods. Our proposed change can be easily integrated into common Python SGD implementations. Furthermore, we provide an example in sklearn.

In future work, we plan to improve the quality of sparse SGD for LR implementations, Logistic Regression, and explore Γ as a hyper-parameter. Sparse SGD offers memory and computational efficiency by focusing only on non-zero features, leading to faster computation times and reduced memory consumption compared to dense SGD. In complement to LR, logistic regression is often viewed as the first step towards a non-linear model. This may broaden the possibilities for our research to delve into the extensive realm of classification applications. In this study, we propose the use of DS on a single data chunk to initialize SGD and subsequently continue with the standard model training. However, further research might investigate how the number of chunks or the amount of data used during the initialization stage with DS compares to the number of chunks used during the SGD model training.

References

1. Al-Amin, S.T., Ordonez, C.: Incremental and accurate computation of machine learning models with smart data summarization. Journal on Intelligent Information Systems (JIIS) **59**, 149–172 (2022). https://doi.org/https://doi.org/10.1007/s10844-021-00690-5
2. Bertin-Mahieux, T., Ellis, D.P.W., Whitman, B., Lamere, P.: The Million Song Dataset. https://labrosa.ee.columbia.edu/millionsong/ (2011)
3. Chen, Y., Chi, Y., Fan, J., Ma, C.: Gradient descent with random initialization: Fast global convergence for nonconvex phase retrieval. Mathematical Programming **176**, 5–37 (2019)

4. Hu, T., Wu, Q., Zhou, D.X.: Convergence of gradient descent for minimum error entropy principle in linear regression. IEEE Transactions on Signal Processing **64**(24), 6571–6579 (2016)
5. James, G., Witten, D., Hastie, T., Tibshirani, R.: An introduction to statistical learning, vol. 112. Springer (2013)
6. Kumar, S.K.: On weight initialization in deep neural networks. arXiv preprint arXiv:1704.08863 (2017)
7. Lichman, M.: UCI machine learning repository (2013), http://archive.ics.uci.edu/ml
8. Lydia, A., Francis, S.: Adagrad—an optimizer for stochastic gradient descent. Int. J. Inf. Comput. Sci **6**(5), 566–568 (2019)
9. Wes McKinney: Data Structures for Statistical Computing in Python. In: Stéfan van der Walt, Jarrod Millman (eds.) Proceedings of the 9th Python in Science Conference. pp. 56 – 61 (2010). https://doi.org/10.25080/Majora-92bf1922-00a
10. Meng, Q., Chen, W., Wang, Y., Ma, Z.M., Liu, T.Y.: Convergence analysis of distributed stochastic gradient descent with shuffling. Neurocomputing **337**, 46–57 (2019). https://doi.org/https://doi.org/10.1016/j.neucom.2019.01.037, https://www.sciencedirect.com/science/article/pii/S0925231219300578
11. Ordonez, C., Zhang, Y., Cabrera, W.: The Gamma matrix to summarize dense and sparse data sets for big data analytics. IEEE Transactions on Knowledge and Data Engineering (TKDE) **28**(7), 1906–1918 (2016)
12. Pace, R.K., Barry, R.: Sparse spatial autoregressions. Statistics & Probability Letters **33**(3), 291–297 (1997)
13. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
14. Picheny, V., Dutordoir, V., Artemev, A., Durrande, N.: Automatic tuning of stochastic gradient descent with bayesian optimisation. In: Hutter, F., Kersting, K., Lijffijt, J., Valera, I. (eds.) Machine Learning and Knowledge Discovery in Databases. pp. 431–446. Springer International Publishing, Cham (2021)
15. Ruder, S.: An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747 (2016)
16. Sutskever, I., Martens, J., Dahl, G., Hinton, G.: On the importance of initialization and momentum in deep learning. In: International conference on machine learning. pp. 1139–1147. PMLR (2013)
17. Zeiler, M.D.: Adadelta: an adaptive learning rate method. arXiv preprint arXiv:1212.5701 (2012)