Replication and Extension of Dynamic Logistic Ensembles for Enhanced Binary Classification

Kris Seekford Amit Singh

Abstract

This study replicates the Dynamic Logistic Ensembles model proposed by Khan and Li (2024) for binary classification, focusing on its interpretability and performance on complex datasets. We reproduced the model's performance on the Wine dataset, achieving comparable accuracy, AUC, recall, and precision for one- and two-layer ensembles, but observed significant performance drops for three- and four-layer ensembles. We hypothesize issues in recursive probability calculations or gradient updates for deeper layers. Additionally, we extended the model to three datasets with natural internal groupings (PIMA Indian Diabetes, Cover Type, and Titanic), comparing the results against Random Forest and Gradient Boosting. Despite consistent interpretability, the model underperformed baselines, suggesting areas for refinement.

1 Introduction

Logistic regression models are valued for its simplicity and interpretability in binary classification, particularly for straight-forward datasets. However, it struggles with complex datasets that contain internal groupings, where methods such as boosting, bagging, and deep learning offer superior accuracy at the cost of interpretability. This trade-off is critical in fields like medicine, where understanding predictions is essential. Khan and Li [Khan and Li, 2024] introduced Dynamic Logistic Ensembles with Recursive Probability and Automatic Subset Splitting to address these challenges, combining the logistic regression's interpretability with improved classification accuracy for complex data sets via a tree-based ensemble.

Our project aimed to: 1. Replicate the model on the Wine dataset used by Khan and Li. 2. Evaluate our models performance compared to the authors'. 3. Extend the model to data sets with natural internal groupings (PIMA Indian Diabetes, Cover Type, Titanic) and compare their results against Random Forest and Gradient Boosting.

Our replication achieved similar results to the authors' for one- and two-layer ensembles but saw significant performance drops for three- and four-layer ensembles, suggesting implementation issues in recursive probability or gradient updates. The extension revealed that, despite interpretability, the model underperformed baselines, highlighting areas for improvement within our code.

2 Background: Logistic Regression

Logistic regression predicts class labels by mapping inputs to probabilities between zero and one using the sigmoid function:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad z = w^T x + b, \tag{1}$$

where w are weights, b is the bias, and x is the input data. The model optimizes weights to minimize a cost function (e.g., log-loss) via gradient-based training, iteratively updating weights to reduce

Preprint. Under review.

prediction errors. Once trained, optimized weights predict class labels for new data, with accuracy evaluated against true labels.

While logistic regressions are interpretable and effective for simple datasets, they often struggle with complex data that contain internal groupings. Methods like boosting, bagging, and deep learning improve accuracy but sacrifice interpretability, posing challenges in domains requiring transparency. Khan and Li's model aims to balance these factors.

3 Overview of the Dynamic Logistic Ensembles Model

Khan and Li's model constructs a tree of logistic regressions. At the root, a logistic regression determines whether to activate the left or right child node. This structure extends across an N-layer ensemble, where probabilities at the final layer are recursively aggregated to produce class predictions. This approach captures internal data groupings while maintaining interpretability, as each node is a simple logistic regression. The model thus addresses both the accuracy and interpretability challenges faced in binary classification.

4 Methodology

We implemented the model in Python, following Khan and Li's equations step-by-step to replicate their approach. We outline the authors' methodology and how we implemented it in Python below.

4.1 Authors' Methodology and Our Coding Approach

The model's theoretical framework is robust, but translating it into code was challenging. We adhered to the equations provided in Khan and Li [2024]:

1. **Data Preparation**: We used the Wine dataset from the UC Irvine Machine Learning Repository, as provided by the authors. It included features (e.g., acidity, sweetness) and binary class labels indicating wine quality. The authors added Gaussian noise to simulate internal groupings. The goal of the model for this dataset was to predict the quality of the wine (one or zero) using the input features it received for each observation. For example, each row in the dataset represents a different brand of wine, thus will have different levels of acidity, sweetness, etc. So, for each brand of wine in the dataset, the model will take these input characteristics and predict whether the wine is of "quality" or not. It is important to note that the authors provided the data set after they cleaned it by using LabelEncoder to make the class labels go from one and two, to zero and one, making the data compatible with their model. Additionally, the authors used principle component analysis to reduce the number of dimensions, effectively reducing the complexity of the data. That being said, the only data cleaning we implemented for our model was scaling the data using StandardScaler, splitting the data set into two data sets, one for the class labels and one for the input features, and splitting these two datasets into a training and testing dataset by keeping 80 percent of the observations for training and 20 percent for testing. The initial dataset had 12 features and 3198 observations. By breaking the dataset into the class labels and input features datasets, we then had one dataset with one row and 3198 observations (the y data set) and the other had 11 features and 3198 observations (the X data set). Lastly, by splitting both of these data sets into a test and training dataset, we ended up with 2558 observations for training and 640 for testing. This gave us the final datasets we used for model training and testing.
2. **Function Creation to Mimic the Authors' Model**: We used the exact process that was outlined through the equations in their paper. We started the code by uploading the data that the authors used and then scaled it as they did. Next, we created the logic of their model through the creation of definition functions for each part of their logic. To start, we created a function that would initialize weights to a small number for each feature in the input data set. Next, we created a function that took in the weights, bias, and input data, which then outputted the basic logistic regression result with the sigmoid activation function. (Equation (2) in the paper) Next, we created the path probability function that creates the probability that a certain node will be activated and is later used for calculating the class

label. (Equation (24) in the paper) After that, we created a function that calculates recursive probability. This function calculates the recursive probability based on which layer the node is in. (Equation (9) and (10) from the paper) After that, we created the function that calculates the probability for each node. (Defined as the term pj in the paper) Next, we created a function that updates the gradient for nodes that are not leaf or immediate parents of leaf nodes. (Process is described using equations (28), (29), (30), (31), (32), and (31) in the paper. After that we created a function that calculates the gradient depending on what layer the current node is on. (Equation (25) for leaf nodes, (27) for immediate parents of leaf nodes), and gradient update function for all other nodes.) We then created the model optimization function that outputs the weights and bias that minimizes the cost function. We then created the model predict function which runs the algorithm for a certain number of iterations, updating the weights and biases after each one. Finally, our last function is the final prediction function which uses the optimized weights and biases to make the predictions of what the class label should be.

3. **Function Definitions**:
   - **Weight Initialization**: Initialized weights to small random values for each feature.
   - **Basic Logistic Regression**: Implemented a sigmoid-based function (Eq. 2) to compute the logistic regression output given weights, bias, and input data.
   - **Path Probability**: Calculated the probability of node activation (Eq. 24) for class-label determination.
   - **Recursive Probability**: Computes recursive probabilities based on node layer (Eqs. 9, 10).
   - **Node Probability**: Calculated per-node probabilities as defined in the paper.
   - **Gradient Updates**: Iteratively updated the gradient calculation for non-leaf nodes.
   - **Gradient Calculations**: Calculated the gradients for non-leaf nodes (Eqs. 28–32) and leaf/parent nodes (Eqs. 25, 27).
   - **Model Optimization**: Optimized weights and bias to minimize the cost function using the above functions.
   - **Model Predictions**: Kept count of iterations and updated weights and bias based on the model optimization function's results.
   - **Final Prediction**: Aggregated probabilities across layers to predict class labels.

4.2 Model Execution

To run this model, we started by defining the number of layers, number of features, number of iterations, and the learning rate. For our implementation we started with one layer, 4500 iterations, and a learning rate of 0.00001. We then called the weight initialization function which started the weights and bias at zero. Then to train the model and get the optimized weights and bias, we called the model predict function, which takes in the input data, the true class labels, the initial weights and bias, the learning rate, and the number of iterations. This function then outputs the optimized weights and bias to be used in your final predictions. To get to this point, the algorithm works as the following. By calling the model predict function, the process starts by setting up the total number of iterations you desire and running the algorithm that many times. Each iteration starts by calling the model optimize function with the input data, class labels, weights, and bias. This will ultimately lead to the gradient weights, bias and the cost for the iteration, which is then used to update the weights and bias. To do that, first the model optimization function will call the recursive probability which, in turn, calls the logistic regression for the node and then calculates the initial probability. After this, the optimization function calculates the cost function using the probability and then calls the function to calculate the gradient. In the gradient calculation function, a loop is made to iterate over each node of the tree. The gradient is then calculated for each node, depending on what layer the node is in the tree. Despite which layer the node is in, they will all calculate the path probability of the node, the node probability, the gradient, and then subtract the mean of the input features multiplied by the gradient from the weights. Then, these weights and bias are outputted to model optimization function, which outputs them to the model predict function. The model predict function then updates the weights and bias by subtracting the current ones by the product of the gradient weights (or bias) and the learning rate. This process is then repeated 4500 times, once for each iteration, and then outputs the final weights and bias to be used for the final predictions. We then call the recursive probability function with our input data, 1 as the class label, the final weights and bias, and the number of layers. This then calculates the probability that the class label is one for each observation in the dataset. We then call the predict function which turns each of these probabilities either into class one or zero. Finally, we evaluate the model using accuracy, precision, recall, AUC, and create an ROC curve. This process is then repeated for a two-layered ensemble, a three-layered ensemble, and a four-layered ensemble. The general algorithm operates as follows:

1. **Initialization**: Set up iterations, number of layers, learning rate, initial weights and bias, and begin training.
2. **Optimization**: For each iteration, the optimize function:
   - Calls the recursive probability function using the logistic regression output.
   - Computes the cost function and gradients for all nodes, incorporating path probability, node probability, and recursive updates.
   - Updates the weights and biases by subtracting the product of gradients and the learning rate.
3. **Repetition**: Repeats a total of 4500 times for convergence.
4. **Prediction**: Computes final recursive probabilities with optimized weights and bias, predicting class labels via a threshold.
5. **Evaluation**: Evaluates performance using accuracy, precision, recall, AUC, and ROC curves for one- to four-layer ensembles.

5 Results

5.1 Replication on Wine Dataset

Table 1 compares our results with Khan and Li's for the Wine dataset.

Table 1: Performance metrics for Wine dataset ensembles.

Layers	Model	Train Acc.	Test Acc.	AUC	Recall	Precision
1	Ours	0.7463	0.7406	0.8015	0.6656	0.7786
	Khan and Li	0.7435	0.7375	0.8019	0.6688	0.7709
2	Ours	0.7447	0.7453	0.8017	0.6688	0.7852
	Khan and Li	0.7576	0.7547	0.8257	0.6972	0.7837
3	Ours	0.5837	0.5781	0.6287	0.7508	0.5548
	Khan and Li	0.7869	0.7641	0.8435	0.7224	0.7842
4	Ours	0.4980	0.4813	0.5222	0.4385	0.4744
	Khan and Li	0.8202	0.7531	0.8320	0.7476	0.7524

Our one- and two-layer ensembles closely matched the authors' results but, three- and four-layer ensembles showed significant performance drops, suggesting issues in how we coded the recursive probability or the gradient update implementations.

6 Discussion

The discrepancy between our three- and four-layer ensemble performances and our one- and two-layer ensemble performances indicates potential errors in our implementation of recursive probability calculations or gradient updates, which grow more complex as additional layers are added on. Debugging these functions by outputting intermediate accuracy metrics for deeper layers could identify the issue. Our replication confirms the model's interpretability and performance effectiveness for shallow ensembles but highlights implementation challenges for deeper ones. Since, the author's reported much higher accuracy measures for their model, we can assume there is a problem with the implementation of the model through our code.

7 Conclusion

Our replication of Khan and Li's Dynamic Logistic Ensembles model was partially successful. The one- and two-layer ensembles aligned with the authors' results, but the three- and four-layer ensembles showed significant inaccuracies. Future work should refine the implementations of the recursive calculation function and/or the gradient update function to ensure more accurate reproduction across all layers.

8 Extension to New Datasets (Extra Credit)

To extend the model, we applied it to three datasets with natural internal groupings from the University of California Irvine's Machine Learning Repository. The data sets we chose were the PIMA Indian Diabetes data set, the Cover Type data set, and Titanic data set. Each of these data sets are well known among the data science community and have been used for numerous studies in the past. We evaluated one- to six-layer ensembles on each data set and compared the performance against random forest and gradient boosting. Khan and Li hypothesized that deeper ensembles would better capture internal groupings, improving performance while maintaining interpretability. To test this theory, we used the code for their model given in their GitHub repository and applied it to the data sets discussed above. The results are shown in Table 2.

Table 2: Performance on extended datasets.

Dataset	Layers	Accuracy	Random Forest	Gradient Boosting
PIMA Diabetes	1	63.6	76.0	68.2
	2	60.4		
	3	64.3		
	4	42.9		
	5	59.7		
	6	50.6		
Cover Type	1	54.3	80.9	78.1
	2	50.3		
	3	51.1		
	4	52.5		
	5	44.2		
	6	47.8		
Titanic	1	55.9	79.7	79.7
	2	50.3		
	3	67.8		
	4	52.4		
	5	51.0		
	6	51.7		

8.1 PIMA Indian Diabetes

This dataset includes health metrics and diabetes labels of PIMA Indians, with possible groupings (e.g., sex, blood sugar levels). The goal of the model was to predict whether an individual has diabetes or not based on the health metrics it received. The performance metrics (accuracy, AUC, recall, precision) for ensembles of one to six layers were consistently lower than Random Forest (76.0%) and Gradient Boosting (68.2%), suggesting the model did not fully capture the internal groups.

8.2 Cover Type

This dataset describes land characteristics and pine tree cover, with groupings in soil or rainfall patterns. The goal of the model was to predict whether the land consisted of the "Pine" cover type based on input features such as rain fall and soil contents. The model accuracy (44. 2% –54. 3%)

underperformed against Random Forest (80. 9%) and Gradient Boosting (78. 1%), again suggesting the model failed to capture the internal grouping of the data.

8.3 Titanic

This data set includes passenger information and survival outcomes the famous Titanic boat crash, with groupings (e.g., cabin class, gender).The goal of the model was to predict whether an individual survived the crash based on the individual's demographics. The model's best accuracy (67. 8% in three layers) was below the Random Forest (79. 7%) and Gradient Boosting (79.7%), again showing that the model may be failing to capture the internal grouping structure of the data.

8.4 Discussion of Extension

While the model maintained interpretability by using a logistic regression at each node, Random Forest and Gradient Boosting significantly outperformed it across all data sets. The hypothesis that deeper ensembles would better capture internal groupings was not supported, as performance did not improve with additional layers. Additionally, our results leave open the debate that some use cases may favor interpretability over accuracy and when such cases are appropriate. Future research should test data sets with stronger internal groupings to validate the theoretical advantages of the model and ensure the model was correctly implemented in the code we used.

8.5 Contributions

Kris Seekford:

- Analyzing the Author's Paper for Model Creation
- Creation of Code for Replication of the Model
- Troubleshooting the Code to Ensure it Worked Properly
- Finding Additional Data Sets for Extra Credit
- Applying the Author's Code to the Extra Credit Data Sets
- Evaluating the Results and Comparison to Other Models
- Recording the Results into Excel and Making Visualizations of the Results
- Recording the Results and Placing Them in Tables
- Creating the Report and Revising It
- Putting the Report in LaTEX Code and Formatting to NeurIPS Standards
- Creating 2/3 of Slides for Presentations
- Presenting the Model and How It Works
- Addressing Questions and Concerns During the Presentation

Amit Singh

- Formatting the Graphs to Make them Visually Pleasing
- Wrote a Few Sentences for Each Extra Credit Data Set
- Creating 1/3 of Slides for Presentations
- Presenting the Initial Slides for the Setup of the Model

References

M.Z. Khan and D. Li. Dynamic logistic ensembles with recursive probability and automatic subset splitting for enhanced binary classification. *arXiv preprint*, 2024.