Optimizing Machine Learning Models for Early Diabetes Detection Using a Large-Scale CDC Dataset

Team: - Yulin Shao - Tong Liu - Yichen Zhao - Yana Xu

Abstract

This project focuses on developing and optimizing machine learning models for early diabetes detection using the CDC Diabetes Health Indicators dataset, which includes over 250,000 records. We explored and enhanced several models including K-Nearest Neighbors (KNN), Logistic Regression, Decision Tree, Random Forest, and XGBoost. To manage the dataset's size and complexity, we implemented optimization techniques such as dimensionality reduction, parallel computing, and algorithm-specific enhancements. Our optimized models demonstrated significant improvements in both runtime and accuracy, showcasing the effectiveness of these strategies. These results can help in early diabetes prediction, potentially improving patient outcomes and reducing healthcare costs. For more details and code, visit our GitHub repository: https://github.com/yanax726/625_Final.

Abstract

Introduction

Diabetes is a significant public health concern, affecting millions of individuals worldwide and leading to high healthcare costs. Early detection is critical for improving patient outcomes and reducing complications, which makes accurate and computationally efficient predictive models highly relevant. The goal of this project was to build and optimize machine learning models that can predict diabetes using demographic and health-related indicators.

We began with a smaller Heart Disease dataset (around 900 entries) to design our initial pipelines. However, the dataset's small size did not expose the computational challenges encountered in large-scale machine learning. To better reflect real-world conditions, we switched to the **CDC Diabetes Health Indicators dataset**, which contains over **250,000 records**. Handling such a large dataset required us to implement **parallel computing**, memory-efficient strategies, and algorithm-specific optimizations. This report describes the methods we used, the optimizations implemented, and their effects on training time and accuracy.

Data

We started with the CDC Diabetes Health Indicators dataset, which contains more than 250,000 records. However, we discovered that the data was quite imbalanced, with far more non-diabetic (label 0) cases than diabetic (label 1). Imbalanced datasets can make it tougher for machine learning models to learn minority-class patterns effectively.

To address this, we rebalanced by using a 70,000-row subset where the classes are roughly equal in size. We felt that training on a more balanced dataset could help models detect diabetic cases more accurately.

We then split this balanced subset into 80% training and 20% testing. All the models in our study used exactly the same training and test splits to keep the comparisons consistent. The training split was used for hyperparameter tuning, cross-validation, and model fitting, while the test split served as a final check for accuracy and runtime benchmarks across all methods. This approach ensured an apples-to-apples comparison among KNN, Logistic Regression, Decision Tree, Random Forest, and XGBoost.

Methods

K-Nearest Neighbors

K-Nearest Neighbors (KNN) is a straightforward algorithm often recommended for classification or regression, but it becomes slow on large datasets because it calculates distances from every test sample to the entire training set. To deal with our massive dataset, we made several key adjustments. First, we reduced dimensionality by running **PCA** on numeric variables and **MCA** on binary variables, cutting down from 20 features to fewer components. We then rewrote the distance function in **C++**, using `nth_element` to partially sort only the top k distances, instead of sorting them all. We also precomputed each training sample's sum of squares to skip that calculation for every query. Finally, we parallelized KNN by splitting the test data into chunks of about 1,000 rows each and applying `parLapply`. Altogether, these optimizations slashed computational time and kept the original KNN logic intact.

Logistic Regression

Logistic Regression was our interpretable baseline model, mapping features to diabetes risk with the logistic function:

$$P(Y=1|X) = \frac{1}{1+e^{-(\beta_0+\sum_{i=1}^{p}\beta_i X_i)}}$$

Because of our high-dimensional dataset, we added **elastic net** regularization:

$$\min_{\beta_0,\beta}\Big[-\sum_{i=1}^{n}\log\big(P(y_i \mid x_i)\big)+\lambda\big(\alpha\|\beta\|_1+\frac{1-\alpha}{2}\|\beta\|_2^2\big)\Big].$$

Cross-validation was also time-consuming, so we created **two versions** of logistic regression. The standard version processed cross-validation folds sequentially, while our parallel version used `doParallel` to distribute fold computations across CPU cores. Both versions kept identical model parameters ($\alpha = 0.5, \lambda \in [0.001, 0.1]$), but the parallel approach significantly reduced training time.

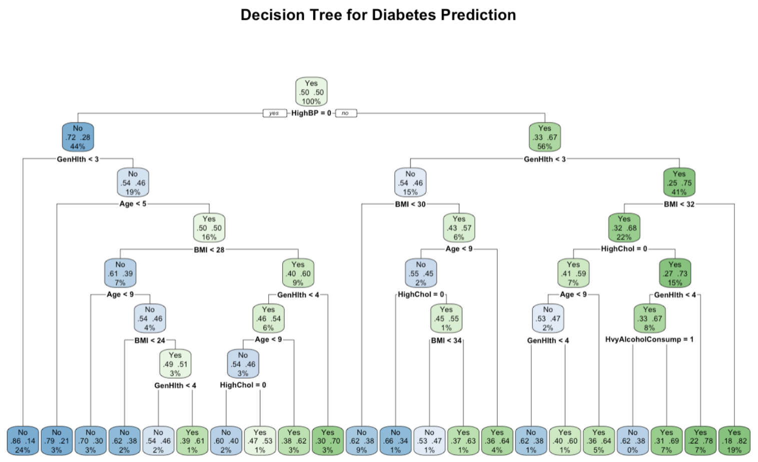
Decision Tree

A single **Decision Tree** (using `rpart`) served as a foundational model in our project. A decision tree splits the dataset into smaller subsets by repeatedly choosing the best feature and threshold to isolate more homogenous groups. Each node in the tree poses a simple question (e.g., "Is BMI > 30?"), and the data branches accordingly.

We tuned the **complexity parameter** (cp) in the range $[0.0005, 0.02]$, which controls how aggressively the tree prunes its branches. Smaller cp values allow deeper, more complex trees, while larger cp values prune more aggressively to avoid overfitting. To select the optimal cp, we performed a grid search combined with **5-fold cross-validation**, using **parallel backends** so each fold could be processed simultaneously. This parallelization sped up our training significantly, especially with tens of thousands of records.

Despite being a relatively simple method, decision trees are valued for their interpretability. Each split corresponds to a clear, human-readable rule. In a medical setting, following the path from the root to a leaf can help clinicians understand exactly why the model classifies a patient as diabetic or not, adding a layer of transparency that's often missing in more complex models.

Below is the decision tree model:



Random Forest

Random Forest is like having a whole team of decision trees vote on an answer. Each tree in the forest is built on a random subset of the data and uses a random subset of features at each split. This randomness helps the ensemble avoid the overfitting that often plagues a single decision tree.

To train the forest, we experimented with parameters like the number of trees and the number of features randomly chosen at each split. Since our dataset was quite large, the standard `randomForest` package struggled with runtime. We switched to **ranger**, which supports multi-threading and handles big data more efficiently. During training, each tree grows independently, which is where parallelization really helps.

After the forest is built, it makes predictions by combining ("voting") the outputs of all the individual trees. This voting tends to produce more stable and accurate results compared to just one tree. The main upside of Random Forest is that it requires minimal tuning yet typically achieves strong performance for many types of data, especially structured health indicators like ours.

XGBoost

XGBoost (eXtreme Gradient Boosting) extends traditional gradient boosting by applying a more regularized model framework to avoid overfitting while preserving strong predictive accuracy. In each iteration, XGBoost builds a new tree to correct the residual errors from the previous round. The algorithm's objective function is given by:

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k),$$

where the first term measures how well the model fits the data, and the second term imposes a penalty on model complexity to prevent overfitting. Specifically, the regularization term is:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2,$$

with T representing the number of leaves, w_j the leaf weights, γ controlling minimum loss reduction for splitting, and λ handling L2 regularization on those leaf weights.

Given our dataset's large size, we tested four implementations of XGBoost to handle memory and computational demands. First, we compared **dense** matrices (straightforward but memory-intensive) vs. **sparse** matrices (efficient for datasets with many zero entries, like one-hot categorical variables). Second, we explored **sequential** vs. **parallel** processing strategies, leading to four distinct configurations: **dense-sequential**, **dense-parallel**, **sparse-sequential**, and **sparse-parallel**. Across all variants, we kept consistent hyperparameters—maximum tree depth of 6, subsample ratio of 0.8, and column sampling ratio of 0.8—and performed grid search over learning rates $\{0.01, 0.1, 0.3\}$. We also applied **early stopping** at 10 rounds, up to a maximum of 100 rounds, to guard against overfitting.

The sparse matrix approach cut down on memory usage by ignoring zero elements, which was especially helpful for categorical variables converted into dummy variables. Parallel XGBoost utilized multi-threading to distribute the computational load, dramatically reducing training time. All versions were evaluated using **5-fold cross-validation**, with **AUC** serving as our main performance metric, since it handles potential class imbalance well in medical datasets. By combining memory-friendly data structures and parallel tree building with early stopping, XGBoost balanced computational efficiency and model quality without sacrificing accuracy.

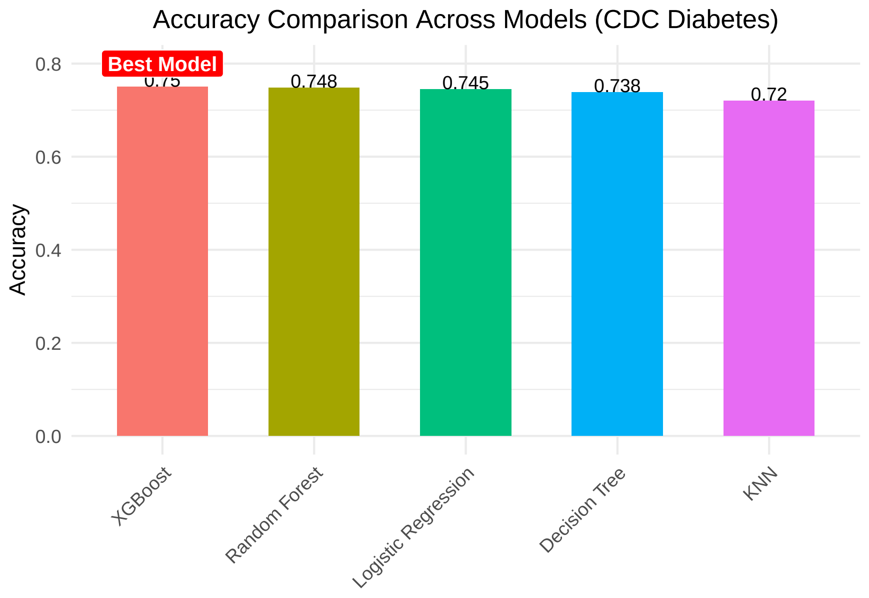
Results

Below is a table comparing runtime and accuracy for unoptimized and optimized models:

Table 1: Runtime and Accuracy Comparison for Unoptimized and Optimized Models

Model	**Unoptimized**		**Optimized**	
	Runtime	**Accuracy**	**Runtime**	**Accuracy**
Decision Tree	15.5s	73.8%	12.4s	73.8%
KNN	85s	72%	32s	72%
Logistic Regression	7.44s	74.5%	**3.25s**	74.5%
XGBoost	38.3s	75.0%	14.1s	**75.1%**
Random Forest	362s	74.8%	3.42s	74.81%

Below is a bar chart comparing the final **accuracy** of each model on our balanced **CDC Diabetes** test set:



From the chart and Table 1, XGBoost comes out on top with 75% accuracy (AUC around 0.82), closely followed by Random Forest at 74.8% (AUC ~0.82). Logistic Regression also performed well, achieving 74.5% accuracy, which is higher than we initially expected. This boost is thanks to parallelizing the cross-validation process, making it about 3x faster than the sequential version.

Our Decision Tree reached 73.8% accuracy and was about 20% faster due to the parallel grid search for the complexity parameter. This speedup made training the Decision Tree much more efficient without sacrificing too much accuracy. Finally, KNN achieved 72% accuracy after significant optimizations like dimensionality reduction, rewriting distance calculations in C++, and parallelizing predictions. These changes reduced KNN's runtime by approximately 62% compared to the naive approach.

Overall, while XGBoost and Random Forest led in both accuracy and efficiency, Logistic Regression proved to be a strong contender with its balanced performance and speed. The Decision Tree also showed competitive accuracy with improved training times, making it a viable option for interpretable models. KNN, despite being the least accurate, became much more feasible with the applied optimizations.

Conclusion

Our project highlights how crucial optimization techniques like dimensionality reduction, parallel computing, and specialized algorithm implementations are when working with large-scale datasets for early diabetes detection. Even simpler models like KNN and Logistic Regression can perform efficiently and effectively with the right tweaks.

XGBoost and Random Forest delivered the best balance of accuracy (~75%) and training speed, both achieving an AUC around 0.82. These ensemble methods proved to be both powerful and scalable, making them excellent choices for handling large, balanced datasets.

The Decision Tree was surprisingly competitive, reaching 73.8% accuracy while becoming ~20% faster through parallel grid search. Its ease of interpretation makes it particularly valuable in healthcare settings where understanding the decision-making process is essential.

While KNN had the lowest accuracy at 72%, the extensive optimizations we implemented—such as PCA/MCA for dimensionality reduction, C++ for distance calculations, and parallel processing—made it a practical option by reducing runtime by ~62%.

Going forward, we plan to delve deeper into benchmarking runtime and memory usage for each model and explore additional ensemble or sampling techniques to further enhance diabetes prediction. These efforts aim to improve model reliability and speed, ultimately contributing to better early detection and intervention strategies in real-world healthcare environments.

References

- Rios Burrows, N., Hora, I., Geiss, L. S., Gregg, E. W., & Albright, A. (2017). Incidence of End-Stage Renal Disease Attributed to Diabetes Among Persons with Diagnosed Diabetes—United States and Puerto Rico, 2000–2014. Morbidity and Mortality Weekly Report, 66(43), 1165–1170.
- Detrano, R., Jánosi, A., Steinbrunn, W., Pfisterer, M., Schmid, J., Sandhu, S., Guppy, K., Lee, S., & Froelicher, V. (1989). International application of a new probability algorithm for the diagnosis of coronary artery disease. American Journal of Cardiology.