Summary of Final Datasets

Dataset Name	Source	Total Number of Rows	# of Features	# of Targets /Classes	Dataset Splits
Disease_symptom.csv	Kaggle	349 (Magnitude: 2)	13	1	Disease_symptom_XTrain shape: (279, 13), Magnitude: 2 Disease_symptom_yTrain shape: (279, 1), Magnitude: 2 Disease_symptom_XTest shape: (70, 13), Magnitude: 1 Disease_symptom_yTest shape: (70, 1), Magnitude: 1
CDC_Diabetes.csv	Kaggle	507,360 (Magnitude: 5)	15	1	CDC_Diabetes_XTrain shape: (202944, 15), Magnitude: 5 CDC_Diabetes_yTrain shape: (202944, 1), Magnitude: 5 CDC_Diabetes_XTest shape: (50736, 15), Magnitude: 4 CDC_Diabetes_yTest shape: (50736, 1), Magnitude: 4
Symptom_Checker.csv	Kaggle	4,961 (Magnitude: 3)	132	1	Symptom_Checker_XTrain shape: (3968, 132), Magnitude: 3 Symptom_Checker_yTrain shape: (3968, 1), Magnitude: 3 Symptom_Checker_XTest shape: (993, 132), Magnitude: 2 Symptom_Checker_yTest shape: (993, 1), Magnitude: 2

Stroke_Data.csv	Kaggle	41,938 (Magnitude: 4)	22	1	Stroke_Data_XTrain shape: (33550, 22), Magnitude: 4 Stroke_Data_yTrain shape: (33550, 1), Magnitude: 4 Stroke_Data_XTest shape: (8388, 22), Magnitude: 3 Stroke_Data_yTest shape: (8388, 1), Magnitude: 3
test_dataset.csv	By Hand	11 (Magnitude: 1)	4	3	test_dataset_X shape: (11, 4), Magnitude: 1 test_dataset_Y shape: (11, 1), Magnitude: 1

Output and Correctness:

Benchmarking and Test Dataset Analysis:

Our test suite starts with two tests to ensure our program successfully takes in the csv files as input and turns them into the correct dataset. Test case “Build Data” ensures buildData() turns the feature dataset into a 2d vector correctly, while test case “Build Class” ensures buildClasses turns the target (y) datasets into vectors. Because these passed, we can be sure that the dataset retains its information when input into our algorithm. Next we have the “Gini Impurity” test case. It compares CART’s GiniImpurity function to manual calculations for each feature in the test dataset when using the entire set (As would happen when deciding upon the first split). The manual calculations can be found in Calculation.pdf in the documents directory. They are written as decimals, however, I converted them to fractions before putting them in the test. This test case guarantees that the algorithm has correct logic for deciding which feature to split on, because it splits on the feature that has the smallest Gini Impurity. The next test case, “Construct Tree”, builds a classification tree using the test dataset, and the resulting tree is equivalent to the one we built manually. The one after uses a very small tree height, which forces the tree to turn impure datasets into leaf nodes. These two tests combined prove the tree is built correctly both when recursion finishes on its own and when we force it to. The final test makes sure that the classify function accurately traverses the tree. Combining this with the previous tests which built the tree correctly ensures that the algorithm is accurate.

Our testing strategy involved benchmarking the algorithm across various datasets with different numbers of elements and features. We capped the tree height at 4 to maintain consistency. The

first test was conducted on a small test dataset comprising 11 elements with 4 features. The build time for the decision tree was impressively short, at only 0.000697 seconds. When classifying 1100 samples, the algorithm took 0.000288 seconds, translating to approximately 2.61818e-07 seconds per sample. Notably, the classify accuracy was a perfect 1, significantly outperforming the accuracy of random guessing, which stood at 0.636364. This initial test confirmed the algorithm's effectiveness in handling small datasets efficiently and accurately.

Performance on Diverse Datasets:

We extended our testing to more complex datasets. The Disease Symptom dataset, with 279 elements and 13 features, was built in 0.063058 seconds. In classifying 7000 samples, it took 0.001843 seconds (around 2.63286e-07 seconds per sample), and the classify accuracy was 0.114286, which is slightly greater than the random guessing accuracy of 0.0428571. This result indicated a need for further optimization in handling more complex datasets.

Similarly, the Symptom Checker dataset, encompassing a larger scale of 3968 elements and 132 features, the tree build time was 0.986558 seconds. The classification of 99300 samples took 0.025738 seconds (2.59194e-07 seconds per sample), achieving a classify accuracy of 0.0976838, which is also slightly surpassing the random guessing accuracy of 0.0292044.

For the Stroke Dataset with 33550 elements and 22 features showed a build time of 1.41106 seconds and a classify time of 0.216568 seconds for 838800 samples (2.58188e-07 seconds per sample). The classification accuracy here was remarkably high at 0.984979, also slightly above the random guessing accuracy of 0.970911.

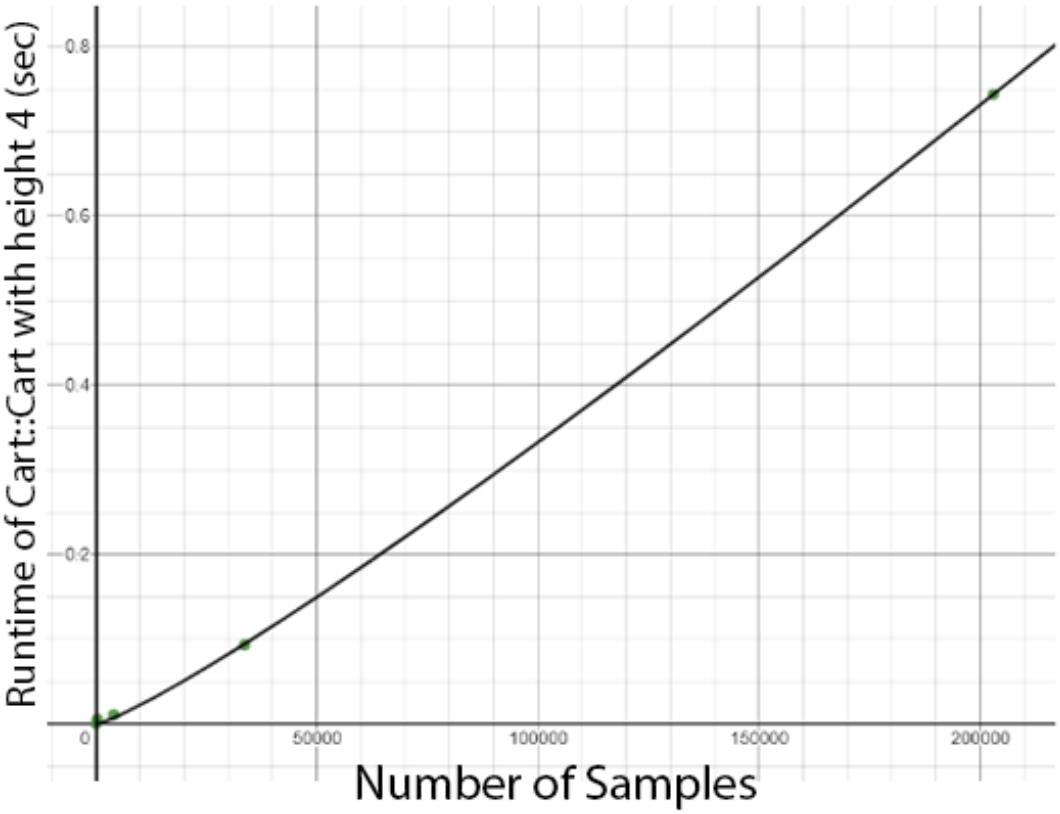
Lastly, the CDC Diabetes Dataset, the largest with 202944 elements and 15 features, had a build time of 8.29176 seconds. The algorithm took 1.46313 seconds to classify 5073600 samples, equating to roughly 2.8838e-07 seconds per sample. The classify accuracy achieved was 0.86209, which was better than the random guessing accuracy of 0.760308.

Conclusion on Algorithmic Accuracy and Efficiency:

Through these comprehensive tests across large datasets (Stroke Dataset and CDC Diabetes Dataset), our algorithm demonstrated considerable accuracy and efficiency. The consistent superiority of classify accuracy over random guessing across all datasets, especially in larger and more complex ones, validates the algorithm's effectiveness. The efficiency, as evidenced by the relatively low classification times even with increasing dataset sizes, underscores the algorithm's scalability and robustness. This thorough testing approach effectively proves the correctness and reliability of our algorithm in handling diverse datasets in practical scenarios.

Benchmarking and Big O:

Our algorithm has two main functions, a computing intensive tree construction, and a far faster classify function. The build tree is a very complicated recursive function that depends on a large number of factors. The inputted data is a 2 dimensional array, so it depends on the number of samples and the number of features per sample, and on the maximum height of the tree. But, it also depends on the values of the data. If a massive dataset correlates perfectly with one feature, then the tree will stop after a single split, while a much smaller dataset could take longer, because it needs many more splits before it becomes pure. That being said, when max tree height is set to 4, so that all trees have about the same number of nodes, the below graph is formed by plotting the number of samples in the dataset compared to the runtimes of the constructor.. The time complexity of the constructor (when height, and therefore node count, is constant) is n*log(n)+f*n where f is the number of features and n is the number of samples. Each split calls GiniImpurity f times, and GiniImpurity iterates through the section of the dataset that the node 'owns,' so that is O(f*n), and each node sorts the dataset, which is average O(n*log(n)). Feature size is generally small compared to sample size, so the time complexity can usually be said to be n*log(n). This is with constant tree height, however. Generally, larger datasets will need more splits to get to pure datasets (when the recursion stops, and where the algorithm has the most accuracy) than smaller ones, so to make the tree useful, nodecount can't be constant, so the time complexity grows faster than n*log(n) in practice, usually.



The Classify function is far simpler. It is a simple tree traversal in the manner of the binary search tree. The below plots show that there is no correlation between number of features or number of samples and runtime of classify. At each node, the algorithm uses a constant time vector index lookup, and then moves to the next node. That means each node in O(1). The traversal starts at the root and ends at a leaf, hitting every node on that path and no other. That means h nodes are visited, where h is the height of the tree, so the time complexity is O(1*h) = O(h). The benchmark has a constant height of 4, so for our tests, Classify has time complexity of O(1).