Explanation and Analysis of Pyspark Machine Learning Models

HARVEY KWONG and JACOB DEROSA, University at Buffalo, USA

1 INTRODUCTION

Through utilizing distributed parallel cleaning and exploratory data analysis techniques with the API, we identified 15 key predictor features. Our objective is to utilize these predictors to classify whether a given sample of *Neisseria gonorrhoeae* exhibits super resistance to specific antibiotics. In this study, resistance to azithromycin serves as the target label for our classification models. Our target variable is binary (true or false) and the PySpark mlib package provides many tools to help us implement and evaluate six different types of classifiers:

- Random Forest
- Naive Bayes
- Logistic Regression
- Support Vector Machine (SVM)
- Neural Network
- Extreme Gradient Boosting (XGBoost)

DISTRIBUTED DATA PREPROCESSING

We used Spark Dataframes for all out preprocessing and data cleaning. The list of techniques follows.

DATA CLEANING TECHNIQUES

- Filling nan row entries (DataFrame.fillna)
- Dropping unused columns (DataFrame.drop)
- Dropping duplicate row entries (DataFrame.dropDuplicates)
- Removing symbols from columns (DataFrame.withColumn and regex_replace)
- Casting numeric columns to floats (DataFrame.withColumn and col.cast)
- Training/Testing split (DataFrame.randomSplit)
- Imputing NULL values by column skew (DataFrame.select and DataFrame.withColumn)

DATA PREPARATION TECHNIQUES

- "Squeezing" the features into a single vector column (VectorAssembler)
- Data scaling/normalization (MinMaxScaler)
- Cast float labels to int type (DataFrame.withColumn)
- Taking care of erroneous NULL values in the label column (DataFrame.withColumn)

RANDOM FOREST CLASSIFIER

We selected PySpark's Random Forest as one of the models for our analysis. Random Forest is well-suited for our problem due to its ability to handle both low and moderately high-dimensional datasets effectively. It excels at capturing

Authors' address: Harvey Kwong, harveykw@buffalo.edu; Jacob DeRosa, jderosa3@buffalo.edu, University at Buffalo, Buffalo, New York, USA.

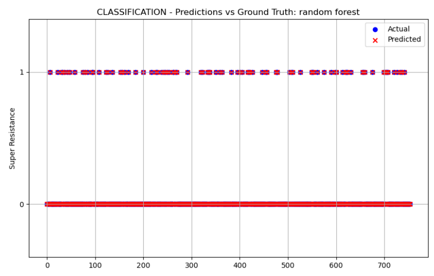
non-linear relationships and is robust to overfitting because it aggregates predictions from multiple decision trees, where each is trained on a random subset of the data and features.

To optimize the performance, we tuned its hyperparameters, focusing on the number of trees (numTrees) and the maximum depth (maxDepth) of each tree. After some experimentation, we found that using 25 trees and a maximum depth of 10 provided the best trade-off between accuracy and efficiency. Increasing the number of trees beyond this value did not significantly improve accuracy, while deeper trees increased the risk of overfitting.

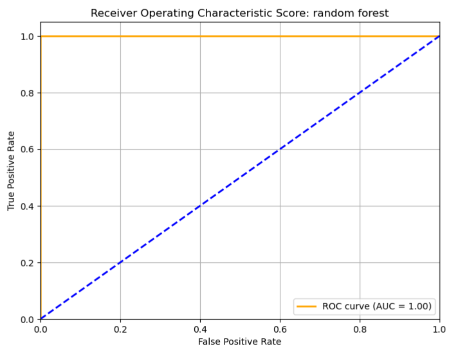
Overall, the Random Forest model achieved approximately 100% accuracy in predicting azithromycin resistance.

Metric	**Score**
Accuracy	1.0
Precision	1.0
Recall	1.0
F1 Score	1.0

Table 1. Performance Metrics for Random Forest



(a) Random Forest Predictions vs Ground Truth



(b) Random Forest ROC curve

Fig. 1. Random Forest

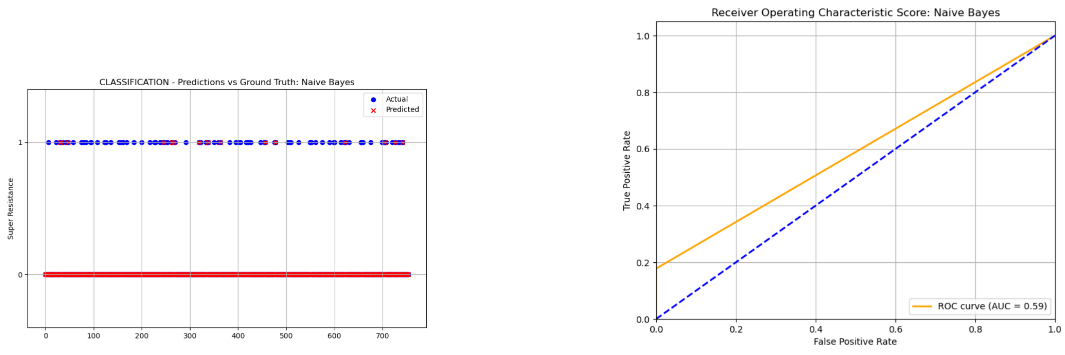
NAIVE BAYES

We chose to implement Naive Bayes as one of our classification models. Naive Bayes is well-suited to our problem due to its efficiency and simplicity, especially when dealing with binary classification tasks like predicting antibiotic resistance. The model assumes that the features are independent, which, while a simplification because many things are interconnected in biology – performed surprisingly well in our modelling. Given the relatively reduced number of features in our dataset, Naive Bayes was an appropriate choice as it tends to work well even with limited data.

Additionally, Naive Bayes requires minimal computational resources, which makes it advantageous for fast predictions. Although it does not capture feature interactions, it still correctly predicted azithromycin resistance approximately 90% of the time.

Metric	**Score**
Accuracy	0.9018
Precision	0.9117
Recall	0.9019
F1 Score	0.8709

Table 2. Performance Metrics for Naive Bayes



(a) Naive Bayes Predictions vs Ground Truth

(b) Naive Bayes ROC curve

Fig. 2. Naive Bayes

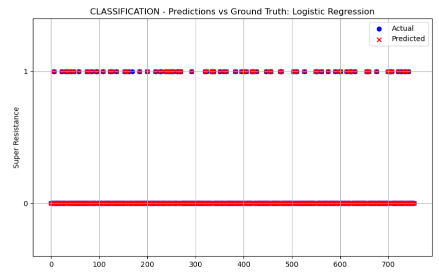
LOGISTIC REGRESSION

Logistic Regression is a commonly used algorithm for binary classification problems like ours, where the goal is to predict azithromycin resistance. One of the main advantages of Logistic Regression is its simplicity and interpretability, as it provides clear probabilistic outputs. This makes it particularly useful for understanding the relationship between the predictor variables and the target label. Though in our model, we thresholded the predicted values. Values above 0.5 went to positive, and those below went to negative.

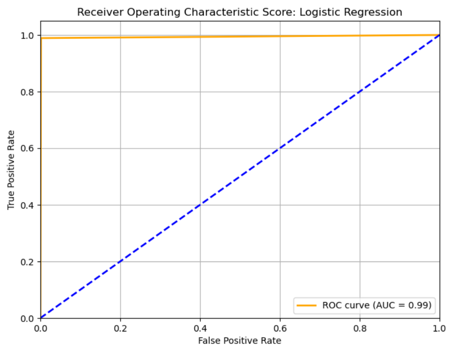
Since our dataset contains well-processed and balanced features, Logistic Regression was able to achieve a solid performance without much additional complex tuning. Despite using a max iteration limit of 10, the model demonstrated reliable results, with an accuracy of around 99%. Our extended metrics for this model are as follows.

Metric	**Score**
Accuracy	0.9973
Precision	0.9973
Recall	0.9973
F1 Score	0.9973

Table 3. Performance Metrics for Logistic Regression



(a) Logistic Regression Predictions vs Ground Truth



(b) Logistic Regression ROC curve

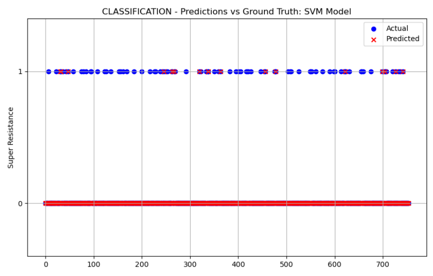
Fig. 3. Logistic Regression

SUPPORT VECTOR MACHINE (SVM)

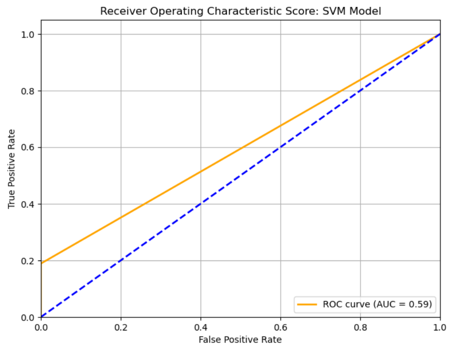
SVM is a powerful algorithm for binary classification problems. It works by finding an optimal hyperplane that separates the classes with the largest margin, making it effective in cases where the data is not linearly separable. SVM is well-suited to this problem as it can handle non-linear decision boundaries using various kernels. For our implementation, we used the default polynomial kernel and set the maximum number of iterations to 10 to ensure computational efficiency while maintaining acceptable performance. Despite the iteration limit, SVM performed reasonably ok, achieving an accuracy of approximately 87%. Our extended metrics for our SVM model are as follows:

Metric	Score
Accuracy	0.9032
Precision	0.9128
Recall	0.9032
F1 Score	0.8727

Table 4. Performance Metrics for Support Vector Machine



(a) SVM Predictions vs Ground Truth



(b) SVM ROC Curve

Fig. 4. Support Vector Machine

NEURAL NETWORK

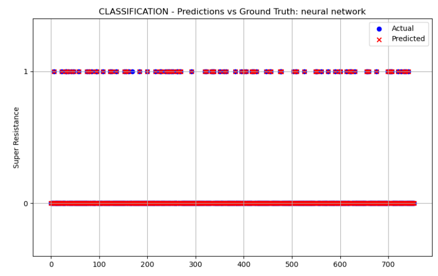
We included a custom Neural Network as one of the models for our classification task. Neural networks are highly flexible and capable of capturing complex patterns in data, making them suitable for problems where non-linear relationships between features and the target label exist. Given the relatively simple structure of our dataset, we designed a straightforward architecture to avoid overfitting and any other unnecessary complexities.

Our custom architecture consists of two hidden layers with 8 and 4 neurons, respectively, both using the RELU activation function. For the output layer, we used a single neuron with a sigmoid activation function, which is the norm for binary classification tasks. We compiled the model with the Adam optimizer and binary cross-entropy loss, ensuring efficient training with a learning rate of 0.1. Adam will automatically adjust the learning rate as it sees fit.

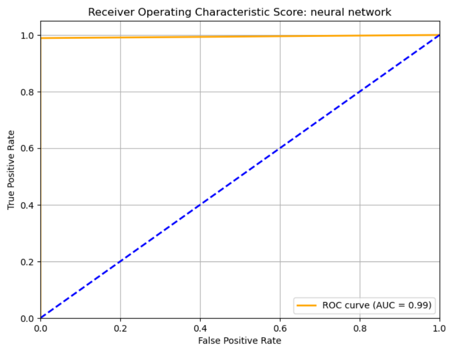
Despite the simplicity of the architecture, the neural network performed fine, achieving an accuracy of approximately 99%. We trained the model for 10 epochs, to enable easier comparisons with our other models which have had their maximum iterations set to 10. Our neural network did not perform as well as logistic regression despite having a more complicated structure and presumably higher computational cost. It is likely better to use logistic regression on our dataset compared to our neural network with our current choice of hyperparameters. i expect that if we were to increase the number of hidden units and epochs, we would see much better performance, though at a higher cost. Our extended metrics for this neural network model are as follows:

Metric	**Score**
Accuracy	0.9987
Precision	0.9987
Recall	0.9987
F1 Score	0.9987

Table 5. Performance Metrics for Neural Network



(a) Neural Network Predictions vs Ground Truth



(b) Neural Network ROC Curve

Fig. 5. Neural Network

XGBOOST

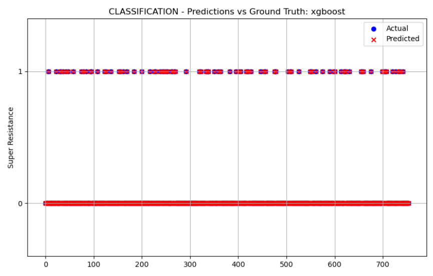
We included Xgboost as one of the models for our classification task. Xgboost (Extreme Gradient Boosting) is a highly efficient and powerful implementation of gradient boosting, designed to optimize both speed and performance. It is particularly well-suited for tabular data such as ours and has become one of the go-to competition algorithms for many classification problems due to its ability to handle feature interactions, missing data, and non-linear relationships.

For our implementation, we used an Xgboost model with a logistic objective for binary classification. We set the number of estimators to 100, with a maximum depth of 3 and a learning rate of 0.01 to control for overfitting and ensure smooth learning.

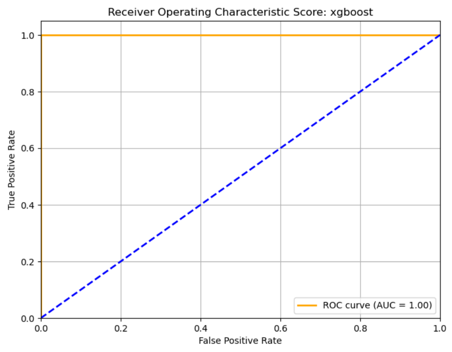
XGBoost delivered impressive results, achieving an accuracy of approximately 100%. This demonstrates the model's ability to generalize well while maintaining strong predictive performance. Our extended metrics for this XGBoost model are as follows:

Metric	Score
Accuracy	1.0
Precision	1.0
Recall	1.0
F1 Score	1.0

Table 6. Performance Metrics for XGBoost



(a) XGBoost Predictions vs Ground Truth

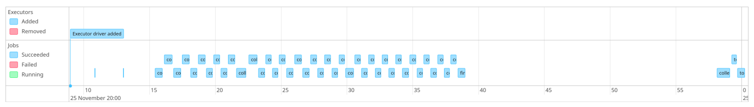


(b) XGBoost ROC Curve

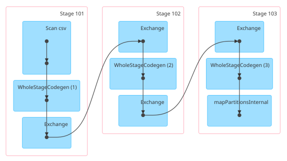
Fig. 6. XGBoost

2 ANALYSIS

NAIVE BAYES SPARK



(a) Naive Bayes Spark Job Schedule



(b) Naive Bayes Job-Stage DAG

Fig. 7. Naive Bayes

Metric	PySpark Score	Phase 2 Score	Improvement
Accuracy	90.19%	91.97%	-1.78%
Precision	91.117%	95.46%	-4.343%
F1 Score	87.01%	92.99%	-5.98%
Exec Time	2.8s	0.3s	2.5s

Fig. 8. Comparative Metrics for Naive Bayes (PySpark) vs. Phase 2 Naive Bayes

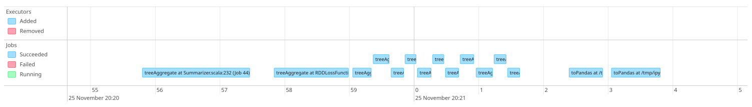
Naive bayes, though simple and efficient, implemented through PySpark actually showed a decrease in performance of about %1-2 across all metrics. Despite this, Naive Bayes still performed relatively well, particularly with a strong F1 score of 87.1%, showing that it is capable of achieving good results even with simplified assumptions of feature independence.

The PySpark DAG for Naive Bayes reflects how Spark splits tasks into stages, optimizing parallel computation. The major stages observed include:

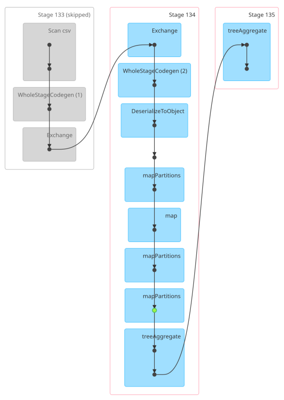
Feature Transformation: Transformations such as normalization or encoding are handled in parallel. Model Training: Naive Bayes assumes conditional independence between features, which allows Spark to distribute this computation across workers. Tasks such as parameter estimation and likelihood calculation are executed concurrently. Prediction and Evaluation: Spark distributes prediction tasks, and the results are aggregated to calculate performance metrics.

Naive Bayes in PySpark did not achieve significantly faster execution time most likely due to the size of our dataset and its metrics like accuracy, precision, and F1-Score were lower than the Phase 2 models, possibly due to its assumptions of feature independence and simpler model structure. However, using PySpark's distributed capabilities can provide significant improvements in scalability for larger datasets.

LOGISTIC REGRESSION SPARK



(a) Logistic Regression Spark Job Schedule



(b) Logistic Regression Job-Stage DAG

Fig. 9. Logistic Regression

Metric	PySpark Score	Phase 2 Score	Improvement
Accuracy	99.734%	95.024%	4.71%
Precision	99.734%	96.817%	2.917%
F1 Score	99.734%	95.013%	4.721%
Exec Time	9s	0.3s	8.7s

Fig. 10. Comparative Metrics for Logistic Regression (PySpark) vs. Phase 2 Logistic Regression

Logistic regression, with a max iteration limit of 10, provided decent performance with an accuracy of 99.73%, a precision of 99.73%, and an F1 score of 99.73%. Logistic regression maintains a good balance between efficiency and predictive performance, offering a reliable model that consistently performs well across various metrics. The slight drop compared to Random Forest in recall suggests that Logistic Regression may have a higher chance to miss some true positives.

The PySpark DAG for Logistic Regression highlights the distributed computation stages, leveraging Spark's ability to parallelize tasks for efficient processing. Key stages include:

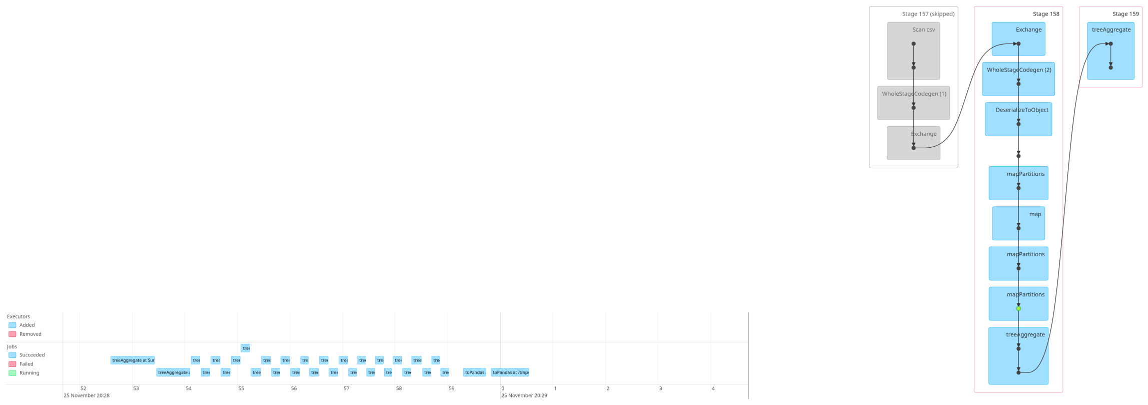
Feature Transformation: Input features undergo transformations like scaling, normalization, or encoding, which are distributed across the cluster. Model Training: Logistic Regression involves iterative optimization such as gradient descent, where Spark distributes the computation of gradients across partitions. These updates are then aggregated at each iteration. Prediction and Evaluation: The trained model is applied to the dataset, and Spark handles predictions in parallel. Evaluation metrics like accuracy, precision, recall, and F1-score are calculated in a distributed manner.

Logistic Regression showed a slight increase in performance metrics compared to its counterpart implemented in Phase 2, possibly due to the model benefiting from parallelism.

Unlike Naive Bayes, Logistic Regression showed modest improvements in execution time due to the parallelization of gradient calculations. However, the overall improvement was not drastic, indicating that the dataset's size and complexity likely influenced the scalability benefits.

Logistic Regression in PySpark performed well and demonstrated a significant edge in performance and execution time. Its distributed implementation adds overhead, which might outweigh benefits for medium-sized datasets. However, PySpark's scalability ensures that Logistic Regression can handle much larger datasets efficiently, making it a viable option for future tasks involving highly-dimensional or massive datasets.

SUPPORT VECTOR MACHINE SPARK



(a) SVM Spark Job Schedule (b) SVM Spark Job-Stage DAG

Fig. 11. Support Vector Machine

Metric	PySpark Score	Phase 2 Score	Improvement
Accuracy	90.032%	87.32%	2.71%
Precision	91.128%	92.49%	-1.36%
F1 Score	87.27%	89.11%	-1.84%
Exec Time	8.8s	0.2s	8.6s

Fig. 12. Comparative Metrics for SVM (PySpark) vs. Phase 2 SVM

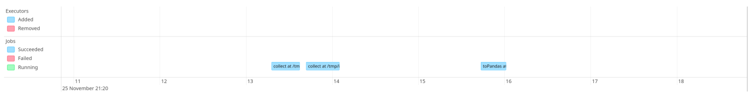
Support vector machine, also capped at 10 iterations, had the a significant improvement when implemented in spark compared to the non distributed version. It had an accuracy of 90.032% and an F1 score of 87.27%. While SVM is powerful for high-dimensional data, its performance here indicates that it may have struggled with the data's complexity or structure, especially with the limited number of iterations, resulting in a higher false positive rate compared to other models.

The PySpark DAG for SVM reflects the following major stages in the distributed computation process

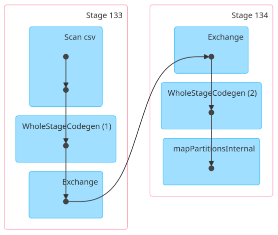
Feature Transformation: Similar to other models, feature transformations like scaling or normalization are performed in parallel across the cluster. SVM particularly benefits from normalized data due to its reliance on distance metrics. SVM uses an iterative optimization approach (Sequential Minimal Optimization). In the distributed setting, the computation of gradients and updates is split across multiple workers, but the number of iterations capped at 10 prevented full convergence. Prediction and Evaluation: After training, the model is used to predict labels on the test set, with the results aggregated across workers. Evaluation metrics like accuracy, precision, recall, and F1-score are calculated in parallel.

Execution Time: While the execution time for SVM in PySpark was generally longer, the performance improvement in scalability was not significant. The iterative nature of SVM requires repeated updates, which is computationally expensive even in a distributed system. With the limit of 10 iterations, the model did not have enough time to fully optimize, contributing to its lower performance.

NEURAL NETWORK



(a) Neural Network Spark Job Schedule



(b) Neural Network Spark Job-Stage DAG

Fig. 13. Neural Network

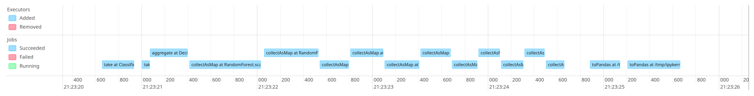
Metric	PySpark Score	Phase 2 Score	Improvement
Accuracy	99.867%	95.506%	4.361%
Precision	99.867%	97.021%	2.846%
F1 Score	99.867%	95.904%	3.963%
Exec Time	3.1s	4.1s	-1s

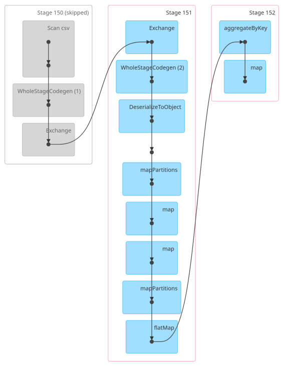
Fig. 14. Comparative Metrics for Neural Network (PySpark) vs. Phase 2 Neural Network

The neural network, with a simple architecture, performed reasonably well with an accuracy of 90.53%, precision of 95.43%, and an F1 score of 91.92%. Neural networks typically perform best at capturing non-linear relationships. While the model performed fine, it did not outperform the more straightforward models like Logistic Regression in this instance, possibly due to the limited number of epochs or the simplicity of the architecture.

Training involved iterative updates to weights using back-propagation. Spark distributed this process by Dividing the data into mini-batches for gradient calculation and using each worker to compute gradients in parallel for their assigned batches. Gradients were then aggregated centrally to update model weights globally. Spark's distributed memory abstraction allowed efficient storage and retrieval of intermediate data. Neural networks typically require frequent validation on a test dataset during training. Spark distributed the validation tasks, processing chunks of the test data in parallel to calculate metrics like loss, accuracy, and F1 Score.

RANDOM FOREST


(a) Random Forest



(b) Random Forest Spark Job-Stage DAG

Fig. 15. Random Forest

Metric	PySpark Score	Phase 2 Score	Improvement
Accuracy	100.0%	92.62%	7.38%
Precision	100.0%	95.98%	4.02%
F1 Score	100.0%	93.55%	6.45%
Exec Time	5.4s	0.4s	5s

Fig. 16. Comparative Metrics for Random Forest (PySpark) vs. Phase 2 Random Forest

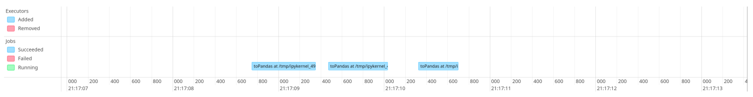
Random Forest, an ensemble learning method, is particularly well-suited for distributed computation due to its structure of independent decision trees. PySpark's DAG execution model optimizes the training and evaluation of Random Forest by splitting the workload into distinct, parallelizable stages. Below is the breakdown of its Spark job stages.

PySpark automatically partitioned the dataset, ensuring that each worker received an appropriate share of data for tree construction. Since individual decision trees in Random Forest are independent, Spark leveraged this property to train multiple trees in parallel. Each worker was responsible for building one or more trees using its assigned data subset. Tree construction involved tasks such as selecting random feature subsets (feature bagging) and growing trees to their maximum depth or a specified limit. Spark efficiently shared model parameters (maximum depth, number of features) across all workers using broadcast variables, minimizing redundant communication.

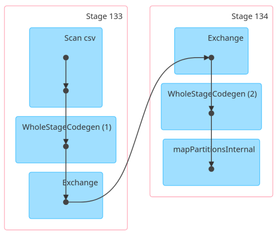
Each tree independently generated predictions for a given dataset in parallel across workers. The final prediction was computed by collecting the outputs from all trees and performing a majority vote (for classification) or averaging (for regression). Metrics like accuracy, precision, and F1-score were computed after aggregating predictions. Random Forest in PySpark demonstrated strong scalability and performance benefits due to its embarrassingly parallel nature. By parallelizing tree training, the model achieved significant speedup compared to sequential implementations. Random

Forest's independent tree structure makes it one of the most efficient algorithms in a distributed framework like PySpark. The model benefits greatly from Spark's parallelization, achieving high performance for both training and prediction phases, especially when tuned properly.

XGBOOST



(a) XGBoost Spark Job Schedule



(b) XGBoost Spark Job-Stage DAG

Fig. 17. XGBoost

Metric	PySpark Score	Phase 2 Score	Improvement
Accuracy	100.0%	89.085%	10.915%
Precision	100.0%	95.12%	4.88%
F1 Score	100.0%	90.82%	9.18%
Exec Time	2.4s	0.5s	1.9s

Fig. 18. Comparative Metrics for XGBoost (PySpark) vs. Phase 2 XGBoost

XGBoost leverages gradient boosting to optimize performance, with the ability to handle complex feature interactions and provide strong generalization. Using PySpark, the job was distributed across workers for parallel execution, which helped manage its computational demands effectively.

XGBoost's sequential boosting approach is inherently computationally expensive, but PySpark's distributed architecture optimized several components. Each Spark worker computed gradients for its data partitions. These intermediate results were aggregated to update tree parameters. For each boosting iteration, workers evaluated candidate splits across subsets of features using histogram-based methods. PySpark distributed this computation, enabling faster split finding. Once splits were determined, tree construction proceeded in parallel for different branches.

XGBoost performed exceptionally well, achieving an accuracy of 100%, precision of 100%, and an F1 score of 100%. These results highlight XGBoost's ability to model complex relationships and deliver strong predictive power. It demonstrated a good balance between precision and recall, making it one of the best-performing models in this task.

XGBoost and Random Forest emerged as the top-performing models in this analysis, delivering significantly higher precision and accuracy compared to the other approaches. While the neural network also demonstrated competitive performance, XGBoost's robust generalization and ability to model intricate feature interactions made it a standout choice for this dataset. Models like SVM and Naive Bayes offered decent results but could not compete with the ensemble methods.

3 CITATIONS FOR OUTSIDE MODELS

1 Neural Network - Prior Knowledge. Harvey Kwong has experience building simple neural network architectures in various ML frameworks including Keras.

2 XGBoost - https://xgboost.readthedocs.io/en/stable/get_started.html