Diabetes Prediction Using Medical and Demographic Factors

1st Kanika Saxena
ksaxena2@buffalo.edu

2nd Rajashree Shanmuganathan
rajashre@buffalo.edu

3rd Alankriti Dubey
adubey2@buffalo.edu

I. Problem Statement

This report aims to identify individuals at higher risk of developing diabetes based on demographic and medical factors, such as obesity, age, and lifestyle choices like smoking. By analyzing these factors, the report seeks to pinpoint specific populations more susceptible to diabetes and highlight the conditions that contribute to its onset, helping to guide targeted intervention and prevention efforts.

A. Background

Diabetes affects millions of people worldwide, and many of those cases are driven by medical and demographic factors. Predictive models using basic medical and demographic data can offer an early warning system, allowing timely detection and reducing patient complications. This is significant, especially in settings where routine testing may not be feasible, as well as early detection and timely treatment of potential patients.

B. Contribution to the Problem Domain

This project has the potential to significantly enhance diabetes management by:

- Early Risk Identification: Helping healthcare providers focus on individuals at high risk, enabling preventive procedures.
- Healthcare Efficiency: Reducing the reliance on extensive testing through accurate, data-driven predictions using basic medical and demographic data.
- Personalized Treatment: Supporting tailored healthcare plans based on each individual's unique risk profile.
- Research Insights: Advancing understanding of the relationships between medical and demographic factors and diabetes risk.

II. Data Source

The dataset used for this project is the Diabetes Prediction Dataset obtained from Kaggle. It contains around 100,000 rows and 9 columns, with the target variable being Diabetes, where a value of 1 indicates the presence of diabetes and 0 indicates its absence. The dataset includes several medical and demographic features such as age, gender, BMI, hypertension, heart disease, HbA1c level, and blood glucose level.

This dataset has been utilized in over 240 Kaggle projects, and has also been referenced in various research papers focusing on diabetes prediction and machine learning applications in healthcare. Its large size and diverse features make it a valuable resource for understanding the factors contributing to diabetes risk.

The dataset can be accessed at: Kaggle Diabetes Prediction Dataset

III. Data Cleaning

In this section, we describe the various data cleaning steps undertaken to prepare the dataset for analysis.

A. Dropping Duplicate Rows

Duplicate rows were removed to ensure data integrity. This was done using the following command:

```
df = df.drop_duplicates()
```

The shape of the dataset after removing duplicates was checked to confirm the changes.

B. Converting Categorical Labels to Numerical Format

Categorical labels in the dataset were converted to a numerical format to facilitate analysis. Specifically, the `gender` column was encoded using a label encoder:

```
label_encoder = LabelEncoder()
df['gender'] = label_encoder.fit_transform(df
    ['gender'])
```

C. Handling Outliers for Age

Outliers in the `age` column were managed by ensuring that all age values were integers:

```
df = df[df['age'] == df['age'].astype(int)]
```

D. Handling Outliers for BMI

To manage outliers in the `bmi` column, the interquartile range (IQR) method was used. The outliers were managed by clipping the `bmi` values to the lower and upper limits.

E. Filtering Invalid Biological Values

Rows with biologically invalid values were removed. The filtering conditions were:

- Age greater than 1
- BMI greater than 10
- Blood glucose level of at least 80

F. Cleaning Text Data

The `smoking_history` column was processed as follows:

- The first letter of each string was capitalized.
- Entries labeled as `No info` were replaced with the mode of the column.

G. Normalizing Blood Glucose and HbA1c Levels

The `blood_glucose_level` and `HbA1c_level` columns were normalized using a standard scaler.

H. Dropping Null Values

To address missing data, rows with null values were dropped:

```
df = df.dropna()
```

I. Categorizing BMI into Ranges

The BMI values were categorized into different ranges to classify individuals as underweight, normal, overweight, or obese.

J. Saving the Cleaned Dataset

Finally, the cleaned dataset was saved to a CSV file:

```
df.to_csv('diabetes_cleaned.csv', sep='\t')
```

IV. Exploratory Data Analysis (EDA)

A. Distribution by Gender

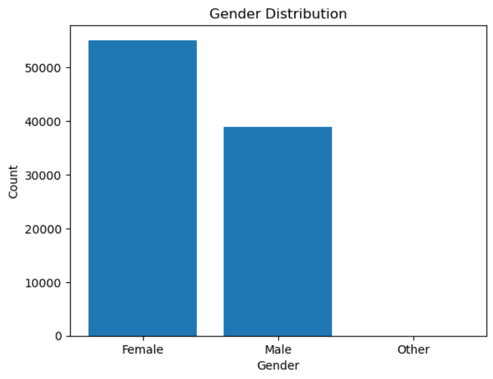


Fig. 1. Distribution by Gender

The histogram shows that a higher proportion of diabetic patients are female compared to males, suggesting a potential gender-related factor in diabetes commonness.

B. Distribution of BMI

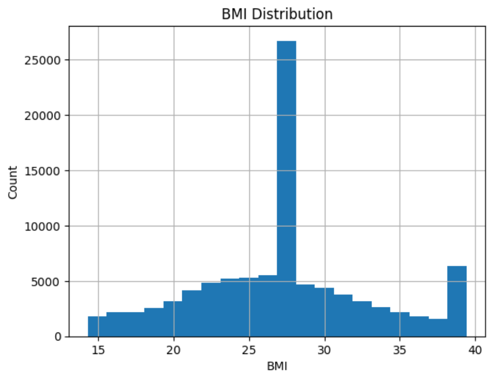


Fig. 2. Distribution of BMI

Approximately 70% of patients in the dataset are classified as either overweight or obese, highlighting a significant relationship between weight and diabetes risk. Given this concerning observation, it is crucial to maintain a healthy weight in order to reduce the likelihood of developing diabetes and similar health complications.

Below is the weight distribution of patients in the table:

Classification	Count
Overweight	41,701
Obesity	23,524
Healthy Weight	21,770
Underweight	7,055

C. Smoking Habits of Patients

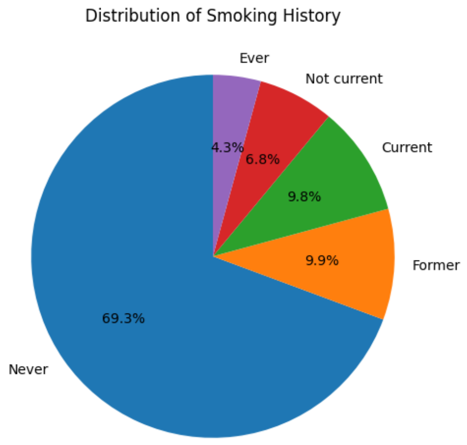


Fig. 3. Smoking Habits of Patients

The visualization of smoking history shows that approximately 70% of patients have never smoked, while 9.7% are current smokers. While the data shows that many patients have never smoked, this doesn't mean that smoking is irrelevant to diabetes risk. Current and former smokers may face higher complications related to diabetes.

D. Glucose Levels vs BMI

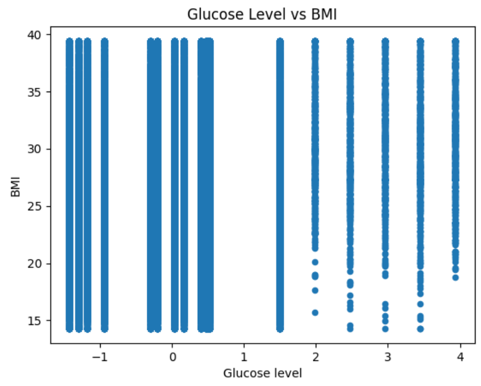


Fig. 4. Glucose Levels vs BMI

This graph concludes that glucose levels can vary independently of BMI values, as shown by the similar glucose levels across a range of BMI classifications. However, higher glucose levels are more commonly observed in patients with a high BMI, indicating a potential association between obesity and glucose dysregulation.

E. Diabetes Rates with Age

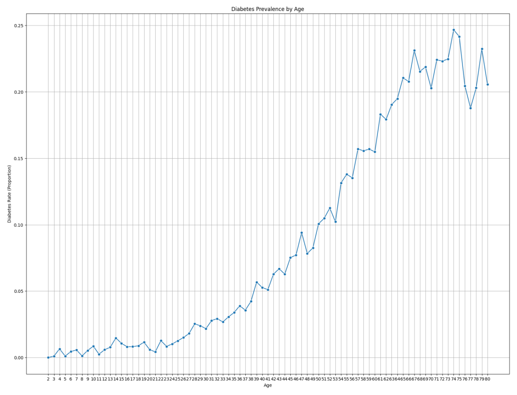


Fig. 5. Diabetes Rates with Age

The line chart clearly demonstrates that the rate of diabetes increases with age, showing a general upward trend despite some slight fluctuations. This highlights the growing risk of diabetes among older populations.

F. BMI Distribution Across Age Groups

The line chart clearly demonstrates that the rate of diabetes increases with age, showing a general upward trend despite some slight fluctuations. This highlights the growing risk of diabetes among older populations.

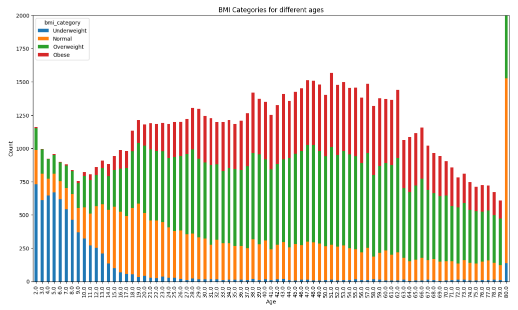


Fig. 6. BMI Distribution Across Age Groups

G. BMI vs Diabetes Risk

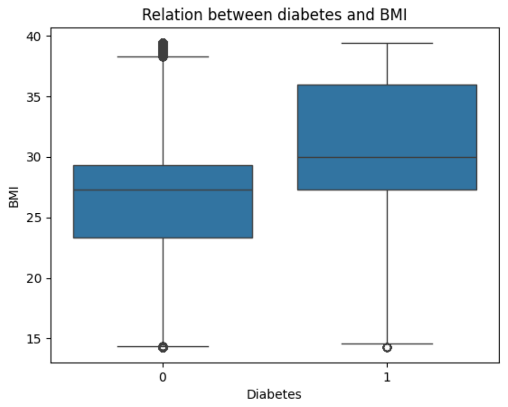


Fig. 7. BMI vs Diabetes Risk

The box plot illustrates that diabetic patients tend to have higher BMIs, with the median BMI falling between 30 and 35. In contrast, non-diabetic individuals have a lower median BMI, around 25 to 30. This suggests that a higher BMI is strongly associated with an increased likelihood of diabetes.

V. Key Outcomes and Insights

The analysis shows that higher BMI (27-37 range), older age (especially 40+), and female gender are strongly linked to a higher risk of diabetes. Additionally, a portion of diabetic patients have a history of smoking.

Key measures include promoting weight management, early screening for older populations and women, and reinforcing smoking cessation programs to reduce diabetes risk.

This analysis will inform the feature selection process for predictive modeling, with BMI, age, and gender identified as the most significant predictors of diabetes risk.

VI. Machine Learning Models

In this section, we present the outcomes of our analysis using six different machine learning models. Each model was selected and tuned to address our specific problem statement, and the results of their performance are detailed below. We provide a comprehensive evaluation of each model, including accuracy, precision, recall, F1-score, and other relevant metrics, along with insights gained from their application.

A. Logistic Regression

*1) **Justification for Choosing Logistic Regession**:* We have chosen the Logistic Regression algorithm based on the following reasons –

1) **Nature of the Problem :** The problem at hand—predicting diabetes based on medical and demographic factors—is fundamentally a binary classification task. Logistic regression is well-suited for such tasks as it models the probability of a binary outcome, making it a natural choice for predicting the presence or absence of diabetes (0 or 1).
2) **Interpretability:** One of the critical advantages of logistic regression is its interpretability. The model coefficients can provide insight into the relationship between each predictor and the likelihood of diabetes, allowing for meaningful medical and demographic interpretations. This is crucial in healthcare, where understanding risk factors can inform treatment and prevention strategies.
3) **Handling of Predictor Types:** Logistic regression effectively handles both continuous predictors (like age and BMI) and categorical predictors (like gender). The ability to include both types of variables without extensive preprocessing makes it a practical choice for this analysis.

*2) **Model Training and Tuning***
: 1. Data Preprocessing :
Before training the model, essential preprocessing steps were taken, including:

- Handling Missing Values: Any missing data in the dataset was addressed to ensure a complete dataset for training.
- Handling Data Inconsistency: Removing data inconsistency by making all string values in title case.
- Encoding Categorical Variables: Gender and smoking history were encoded using techniques such as one-hot encoding to convert categorical data into a numerical format suitable for logistic regression.

2. Model Training :

- **Splitting the Data:** The dataset was split into training and test sets (typically an 80-20 split) to evaluate model performance effectively.
- **Fitting the Model**: The logistic regression model was trained using the training set. Hyperparameters such as regularization were adjusted using cross-validation to prevent overfitting and enhance generalization.

*3) **Effectiveness of the Algorithm**:*

1. Performance Metrics

Several metrics were utilized to evaluate the effectiveness of the logistic regression model:
label=•

- **Accuracy:** Measures the overall correctness of the model. Given the class distribution in the dataset, accuracy alone might be misleading.
- **Precision and Recall:** These metrics are crucial in healthcare applications where false negatives (missed diabetes cases) can have serious consequences. High precision indicates a low false positive rate, while high recall indicates a low false negative rate.

2. Model Performance Results

Upon evaluation, the logistic regression model achieved:
label=•

- **Accuracy:** Approximately 95.68%, suggesting that the model performs well in classifying diabetes cases.
- **Macro Average:** label=–
  - **Precision:** 0.91 indicates the average precision across classes, treating all classes equally, indicating that when the model predicts a patient has diabetes, it is correct 91% of the time.
  - **Recall:** 0.80 indicates the average recall, again treating all classes equally, which means the model correctly identifies 80% of actual diabetes cases.
  - **F1-Score:** 0.85 is the average F1-score across classes, indicating a balanced trade-off between precision and recall.
- **Weighted Average:** label=–
  - **Precision:** 0.95, reflecting the average precision while considering the number of instances in each class.
  - **Recall:** 0.96, indicating a strong overall recall.
  - **F1-Score:** 0.95 shows a strong balance between precision and recall when accounting for class imbalance.

Insights Gained from the Algorithm

1) **Key Predictors:** The analysis revealed that higher BMI, older age, and female gender are significantly associated with an increased risk of diabetes. This supports public health initiatives focusing on obesity and age as critical factors in diabetes management.
2) **Impact of Lifestyle Choices:** While the dataset indicated that a majority of patients are non-smokers, the presence of diabetes in current and former smokers emphasizes the need for continued awareness of lifestyle choices.
3) **Data-Driven Interventions:** The insights from the predictive model allow for tailored interventions, such as: label=–
   - Targeted screenings for older populations and individuals with higher BMI.
   - Implementing educational programs focused on weight management and smoking cessation.
   - Developing personalized treatment plans that take into account individual risk profiles based on demographic and medical factors.

4) **Future Research Directions:** The analysis opens avenues for further research, such as investigating the impact of other lifestyle factors (e.g., diet, physical activity) on diabetes risk. It may also provide a basis for exploring more complex models (like ensemble methods) to see if predictive performance can be improved.

B. *Decision Tree*

*1) **Justification for Choosing Decision Tree Algorithm**:* We have chosen the Decision Tree algorithm based on the following reasons –

1) **Interpretability**: Decision Trees are highly interpretable compared to many other machine learning models. The flowchart-like structure allows stakeholders (e.g., healthcare providers) to understand how decisions are made based on specific features. This transparency is crucial in healthcare, where understanding the rationale behind predictions can inform clinical decisions.
2) **Handling Non-Linear Relationships**: The nature of the data in diabetes prediction often includes non-linear relationships between features and the target variable. Decision Trees can capture these relationships effectively without requiring extensive preprocessing or transformation of the data.
3) **Feature Importance**: Decision Trees inherently provide a mechanism to assess feature importance, allowing us to identify which demographic and medical factors are most influential in predicting diabetes risk. This is particularly useful for informing further research and preventive strategies.

*2) **Model Training and Tuning***
: 1. Data Preprocessing :
Before training the model, essential preprocessing steps were taken, including:

- Missing values were checked and handled accordingly.
- Categorical features were converted to numerical format using ordinal encoding.
- The dataset was split into training and testing subsets to evaluate model performance accurately.

2. Model Training :

- The initial model was trained using the default parameters of the DecisionTreeClassifier.
- The model was fitted to the training data using `dt_classifier.fit()`.

3. Feature Selection :

- Features were assessed for their importance, which helped refine the feature set and remove irrelevant or redundant features, improving model accuracy and interpretability.

*3) **Effectiveness of the Algorithm**:*

1. Performance Metrics

Several metrics were utilized to evaluate the effectiveness of the logistic regression model:
label=•

- **Accuracy:** Measures the overall correctness of the model. Given the class distribution in the dataset, accuracy alone might be misleading.
- **Precision and Recall:** These metrics are crucial in healthcare applications where false negatives (missed diabetes cases) can have serious consequences. High precision indicates a low false positive rate, while high recall indicates a low false negative rate.

2. Model Performance Results

Upon evaluation, the Decision Tree model achieved:
label=•

- **Accuracy:** The model achieved an accuracy of approximately 95%, indicating a high level of correct predictions overall.
- **Precision:**
  - For C0 (non-diabetic): 0.97 (This means 97% of the instances predicted as non-diabetic were correct.)
  - o For C1 (diabetic): 0.72 (This means 72% of the instances predicted as diabetic were correct.)
- **Recall:**
  - For C0 (non-diabetic): 0.97 (This means 97% of the actual non-diabetic cases were correctly identified.)
  - For C1 (diabetic): 0.74 (This means 74% of the actual diabetic cases were correctly identified.)
- **F1-Score:**
  - For C0 (non-diabetic): 0.97 (This indicates a very strong performance for predicting non-diabetic patients.)
  - For C1 (diabetic): 0.73 (This shows moderate performance for predicting diabetic patients.)

Insights Gained from the Algorithm

- Feature importance analysis revealed that certain factors (e.g., BMI, age, and gender) were significantly associated with diabetes risk.
- The decision tree visualization helped identify specific thresholds for features (e.g., BMI ¿ 30) that are critical indicators of diabetes, which can guide public health interventions and individual risk assessments.
- The model's performance on different classes provided insights into the need for targeted interventions for populations at risk, especially older individuals and those with high BMI.

C. *K-Nearest Neighbors (KNN)*

*1) **Justification for Choosing KNN**:* Using K-Nearest Neighbors (KNN) for predicting diabetes in this dataset is justified for these reasons:

1) **Binary Classification:** KNN is ideal for binary classification tasks, directly predicting diabetes status (0 for non-diabetic, 1 for diabetic).
2) **Non-Linear Relationships:** KNN effectively captures complex, non-linear relationships between features like age, BMI, and HbA1c level, which are common in medical data.
3) **No Distribution Assumptions:** As a non-parametric method, KNN does not assume any specific distribution for input features, making it suitable for diverse medical data.
4) **Mixed Data Types:** KNN handles both categorical (e.g., gender, smoking history) and continuous features (e.g., age, BMI), allowing for a comprehensive analysis.
5) **Hyperparameter Tuning:** The ability to optimize k (the number of neighbors) through cross-validation enhances model performance.
6) **Focus on Local Patterns:** KNN emphasizes local data structure, effectively identifying patterns within demographic groups relevant to diabetes risk.

*2) **Work Done for Training the Dataset**:* The main hyperparameter for KNN is the number of neighbors (k). To determine the optimal value of k, a range of values (1 to 25) was tested, and the model's accuracy was evaluated for each. The dataset was divided into training and testing sets in an 80:20 ratio using a random state of 42.

- Number of training rows: 75,240
- Number of testing rows: 18,810

Choosing a suitable k is crucial because a smaller k can lead to overfitting, while a larger k might cause underfitting.

*3) **Effectiveness**:* The KNN model demonstrated strong overall performance in predicting diabetes, achieving a maximum accuracy of **95.09%** for k = **7** on the test dataset, indicating that it correctly classified a substantial majority of the values.

***Key Metrics**:*

- **Accuracy:** 0.9509
- **Precision:** 0.9107 (for positive class)
- **Recall:** 0.5085 (for positive class)
- **F1 Score:** 0.6526 (for positive class)

Detailed Classification Report: **For C0 (No Diabetes):**

- **Precision:** 0.95: This indicates that 95% of the instances predicted as "No Diabetes" were correctly classified.
- **Recall:** 1.00: The model identified all actual "No Diabetes" cases correctly.
- **F1 Score:** 0.97: Harmonic mean of precision and recall, showing a very high performance for this class.

For C1 (Diabetes):

- **Precision:** 0.91: 91% of the instances predicted as "Diabetes" were indeed diabetes cases.
- **Recall:** 0.51: The model only identified about 51% of actual diabetes cases, indicating it misses a considerable number of true positives.
- **F1 Score:** 0.65: This score reflects a balance between precision and recall but suggests room for improvement in detecting diabetes.

Conclusion The KNN algorithm achieved a high **accuracy** of 95%, indicating that it correctly predicted most of the outcomes. However, the model's **recall** (0.51) was lower compared to precision. This is also reflected in the **confusion matrix**, which shows 838 false negatives. The **ROC curve area** of 0.89 demonstrates that the model has a strong ability to differentiate between positive and negative cases, but there is still room for improvement in detecting positive cases.

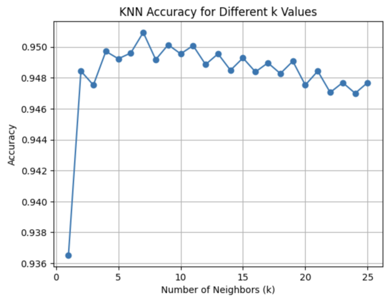


Fig. 8. Accuracy Accuracy for k-values (Best $k = 7$)

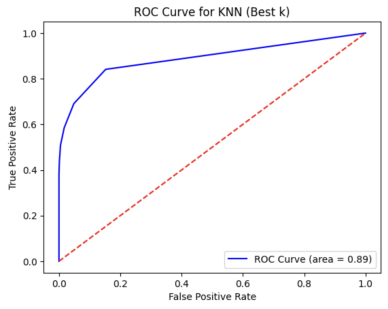


Fig. 9. ROC Curve for KNN (Best $k = 7$)

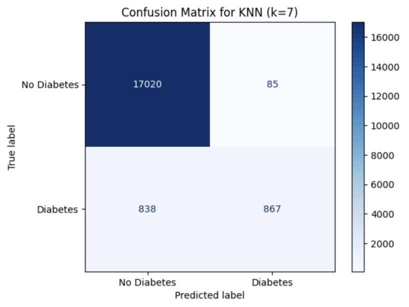


Fig. 10. Confusion Matrix for KNN (Best $k = 7$)

***Intelligence Gained from KNN Prediction**:* The application of the KNN algorithm provided several insights into the diabetes prediction task:

- **Optimal K Value:** The best value of k was found to be 7, which yielded the highest accuracy of approximately

95.09%. This emphasizes the significance of hyperparameter tuning in achieving optimal model performance.

- **High Precision for Non-Diabetic Cases:** The model demonstrated a precision of 0.95 for predicting non-diabetic cases, indicating that 95% of the predicted instances for this class were accurate.
- **Challenges in Detecting Diabetic Cases:** The recall for diabetic cases was only 0.51, suggesting that the model missed a considerable number of true positives, highlighting the need for model improvement.
- **F1 Score Indicating Trade-Offs:** An F1 score of 0.65 for diabetic cases reflects the trade-off between precision and recall, highlighting room for improvement in the model's ability to identify positive instances.

D. Random Forest Algorithm

*1) **Justification for Choosing Random Forest Algorithm**:* Using Random Forest Algorithm for predicting diabetes in this dataset is justified for these reasons:

1) **Efficient for Large Datasets**: With 95,000 rows, Random Forest processes large datasets effectively by building multiple trees in parallel.
2) **Suitable for Binary Classification**: It excels in binary tasks (0: No Diabetes, 1: Diabetes) by combining predictions from multiple decision trees for better accuracy.
3) **Robust to Outliers**: The ensemble method reduces sensitivity to outliers, such as varying blood glucose or BMI values.
4) **Feature Importance**: Identifies key factors (e.g., age, BMI, HbA1c level) that significantly impact diabetes prediction, offering useful insights.

*2) **Work Done for Training the Dataset**:* The Random Forest model was configured with 100 decision trees (*n_estimators* = 100), chosen to balance performance and computational efficiency. Setting a fixed random state ensured reproducibility of results. This configuration helped achieve accurate predictions for the binary classification of diabetes. The dataset was split into training and testing sets using an 80:20 ratio, with a random state of 42.

- **Training set**: 75,240 samples
- **Testing set**: 18,810 samples

*3) **Effectiveness**:* The Random Forest model demonstrated strong overall performance on the diabetes prediction dataset. Key evaluation metrics include:

- **Accuracy:** 96.60%
- **Precision:** 92.84% (for positive class)
- **Recall:** 67.68% (for positive class)
- **F1 Score:** 78.29% (for positive class)

Detailed Classification Report: For C0 (No Diabetes):

- **Precision:** 97% — The model correctly identified 97% of cases predicted as ”No Diabetes.”
- **Recall:** 99% — Almost all actual ”No Diabetes” cases were accurately detected.
- **F1 Score:** 98% — High precision and recall led to a strong F1 score for this class.

For C1 (Diabetes):

- **Precision:** 93% — The model accurately identified 93% of instances predicted as ”Diabetes.”
- **Recall:** 68% — The model detected 68% of actual diabetes cases, indicating it missed some true positives.
- **F1 Score:** 78% — Reflects a balance between precision and recall but also suggests further tuning could improve sensitivity.

Conclusion The Random Forest model achieved an **accuracy** of 96.60%, indicating its effectiveness in predicting diabetes outcomes. The model is reliable in identifying true positives; however, it missed a high proportion of actual diabetes cases. Additionally, the **ROC curve area** of 0.96 indicates discriminative ability between diabetic and non-diabetic patients, highlighting the model's overall robustness with potential for improvement in detecting positive cases.

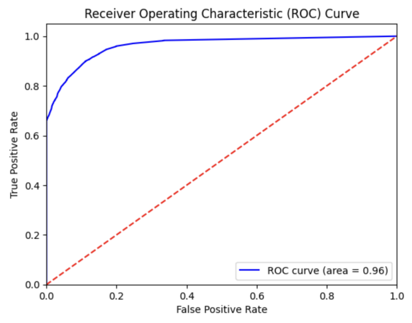


Fig. 11. ROC Curve for Random Forest

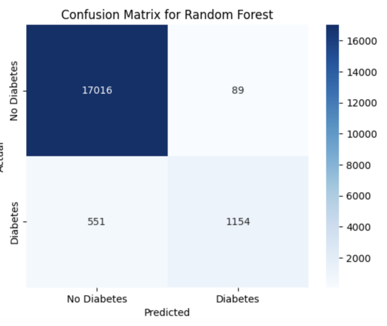


Fig. 12. Confusion Matrix for Random Forest

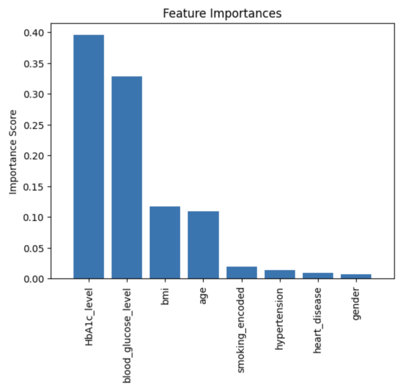


Fig. 13. Important Features for Prediction

Intelligence Gained from Random Forest Prediction: The Random Forest model provided valuable insights for diabetes prediction:

- **High Accuracy:** The model achieved an impressive accuracy of approximately 96.6%, effectively classifying the majority of diabetic and non-diabetic patients, and showcasing its reliability in real-world applications.
- **Precision-Recall Dynamics:** While the precision for diabetes cases was 92.8%, the recall of 67.7% indicates that a high number of diabetic patients may have been overlooked, highlighting the need for model refinement.
- **Feature Importance Insights:** Key predictors such as HbA1c level, blood glucose level, BMI, and age significantly influence diabetes classification, providing a basis for targeted interventions and risk assessment strategies.
- **Robust Class Handling:** The model maintained a high precision of 97.0% and a recall of 99.0% for non-diabetic cases, demonstrating its capability to effectively differentiate between the two classes and reduce the likelihood of false positives.

E. SVM

1) Justification for Choosing SVM: The SVM model is used for the following reasons:

- **Effective for Non-linear Data:** SVM is effective for handling non-linear correlations between features using the kernel method.
- **High Dimensional Data:** SVM is effective in high-dimensional spaces and can handle both linear and non-linear relationships. For diabetes prediction, it's useful in drawing decision boundaries in a complex feature space.
- **Resistance to Over-fitting:** SVM's margin-based method to classification reduces the risk of over-fitting, particularly in high-dimensional fields. SVM can better generalize to previously unseen data by maximizing the margin between classes,

2) Work Done to Train the Model:

1) **Data Preprocessing:** The data is made sure that it is preprocessed by handling missing values,encoding categorical values
2) **Standardization:** Required feature scaling to ensure the effectiveness of SVM's distance-based decision boundaries.
3) **Splitting the Data:** The data was split into training and testing sets using an 80:20 ratio.
4) **Model Tuning:** The linear kernel was selected due to its effectiveness in high-dimensional spaces and its ability to create a clear decision boundary for linearly separable data.
5) **Hyper-parameters tuning:** The C (regularization) and gamma(kernel coefficient) were also tuned.
6) **Splitting the Data:** The data was split into training and testing sets using an 80:20 ratio.

3) Effectiveness: The following metrics were evaluated on the SVM model .The accuracy of the model is **95%** on the test dataset, indicating that it classified a majority of the values.

Key Metrics: **For C0 (No Diabetes):**

- **Precision:** 96% This indicates that most of the cases are correctly classified.
- **Recall:** 99% This indicates that 99% of the instances predicted as "No Diabetes" .It has almost classified correctly.
- **F1 Score:** 98% This indicates that the model has performed very well

For C1 (Diabetes):

- **Precision:** 91% This high precision indicates that the model is good at reducing false positives, making it useful for identifying diabetic individuals.
- **Recall:** 58% This relatively low recall indicates that the model misses a considerable proportion of true diabetes patients.
- **F1 Score:** 71% An F1 score of 71% shows a reasonable balance of precision and recall. While the precision is excellent, the reduced recall lowers the F1 score, indicating that the model's capacity to detect all diabetic patients might be improved.

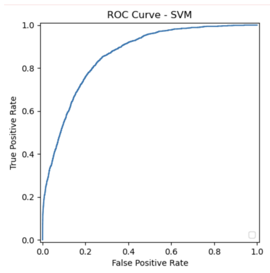


Fig. 14. ROC Curve For SVM

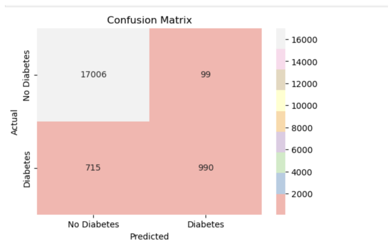


Fig. 15. Confusion Matrix for SVM

Intelligence Gained from SVM Prediction: The following insights are gained from the SVM diabetes prediction task:

- **Effectiveness of Linear Separation:** Using a linear kernel, SVM successfully distinguished between diabetes and non-diabetic patients. This implies that a definite decision limit exists in the feature space, allowing for precise patient classification based on measurable characteristics.
- **Feature Importance:** SVM's performance can help identify which features are most important for diabetes prediction. Analyzing feature weights can assist in identifying crucial indicators, such as glucose levels and BMI, allowing healthcare providers to focus on these parameters during patient examinations.
- **Performance:** The model achieved high precision (91%) but had a lower recall (58%). This efficiency in precision suggests that while the model performs incredibly well in reducing false positives, it may require alters to boost sensitivity and discover more real positives.

F. *Naive Bayes*

*1) **Justification for Choosing Naive Bayes:*** The Naive Bayes model is used for the following reasons:

- **Probabilistic Interpretation:** Naive Bayes succeeds at providing probabilistic results, such as the likelihood of a patient having diabetes (0 for non-diabetic, 1 for diabetic). This can be useful in a medical situation to inform the classification decision.
- **Handling of Categorical and Continuous Data:** Diabetes datasets typically include both continuous (e.g., blood pressure, glucose levels) and categorical variables (e.g., family history of diabetes). Naive Bayes can be used to process both types of data.
- **Interpretability:** Naive Bayes models are easily interpretable, making them useful for medical applications such as diabetes prediction.

*2) **Work Done to Train the Model:***

1) **Handling Categorical Features:** The dataset contained categorical variables which were transformed into numerical values using label encoding. This method assigns each unique category a distinct integer.
2) **Tuning:** Since the dataset included continuous features such as age, BMI, and blood glucose levels, Gaussian Naive Bayes was chosen. This variant of Naive Bayes assumes that continuous features follow a normal (Gaussian) distribution, making it suitable for handling these kinds of attributes in the dataset.
3) **Splitting the Data:** The data was split into training and testing sets using an 80:20 ratio.
4) **Training the Model:** The model was trained using the features provided. During this process, the model learned the conditional probabilities of each feature with respect to the target label (diabetes or non-diabetes).

*3) **Effectiveness:*** The following metrics were evaluated on the Naive Bayes model .The accuracy of the model is **90%** on the test dataset, indicating that it correctly classified a substantial majority of the values.

Key Metrics: **For C0 (No Diabetes):**

- **Precision:** 96% This indicates that most of the cases are correctly classified.
- **Recall:** 93% This indicates that 93% of the instances predicted as "No Diabetes" were correctly classified.
- **F1 Score:** 95% This indicates that the model has performed well

For C1 (Diabetes):

- **Precision:** 48% Precision suffered slightly, as Naive Bayes is prone to more false positives in this context.
- **Recall:** 66% The model only identified about 66% of actual diabetes cases.
- **F1 Score:** 55% the F1 score of the diabetes prediction model is 55%, this indicates that the model's balance between precision and recall is somewhat lacking and can be improved in its ability to make accurate predictions

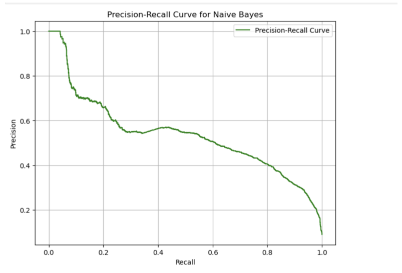


Fig. 16. Precision Recall Curve

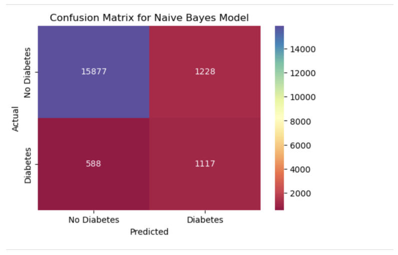


Fig. 17. Confusion Matrix for Naive Bayes

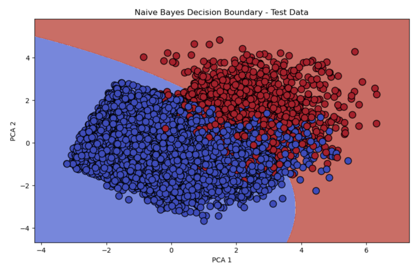


Fig. 18. Naive Bayes on Test Data

Intelligence Gained from Naive Bayes Prediction: The following insights are analysed from the Naive Bayes diabetes prediction:

- **Feature Independence Assumption:** Naive Bayes makes the assumption that all features are independent. But, moderate performance indicates that some features—such as BMI and glucose levels—may really interact with one another, something that Naive Bayes does not account for.
- **Essential Attributes for Diabetes Prediction:** The higher probability of characteristics like age, BMI, and glucose levels indicate that these factors had a greater influence on diabetes prediction. This implies that these are important variables to consider when assessing a person's risk of developing diabetes.
- **Performance:** The Naive Bayes model had slightly lower precision, recall, and F1-score than SVM and Logistic Regression. This suggests that while Naive Bayes provides a fast and interpretable solution, it may not be as accurate for complex problems

VII. Data Processing and Machine Learning using Apache Spark MLlib

A. Data Preprocessing

In this section, we describe the various data cleaning steps performed using Spark to prepare the dataset for analysis.

Step 1: Drop Duplicate Rows

Remove duplicate rows from the dataset and count the remaining records.

Step 2: Convert Categorical Labels to Numerical Format

Transform categorical labels (e.g., gender) into a numerical format using StringIndexer.

Step 3: Handle Outliers in Age Column

Filter out non-integer values from the ‘age‘ column to handle potential outliers.

Step 4: Handle Outliers in BMI Column Using IQR

Use the Interquartile Range (IQR) method to identify and filter outliers in the BMI column.

Step 5: Filter Specific Ranges

Filter the dataset to include records where age, BMI, and blood glucose level fall within specified ranges.

Step 6: Capitalize Smoking History Column

Standardize the ‘smoking_history‘ column by capitalizing the first letter of each value.

Step 7: Replace Missing Smoking History with Mode

Replace missing or default values (e.g., ”No info”) in the ‘smoking_history‘ column with the most frequent value (mode).

Step 8: Standardize Features

Assemble selected features into a vector and standardize them using StandardScaler.

Step 9: Drop Rows with Null Values

Remove rows containing null values to ensure data completeness.

Step 10: Define BMI Categories

Classify BMI values into categories such as Underweight, Normal, Overweight, and Obese, and add a new column for these categories.

B. Completed Stages

Completed Stages (46)

46 stages were successfully executed. Each represents a step in the preprocessing pipeline.

Examples::

- **Stage 0 (csv at NativeMethodAccessorImpl.java:0)**:
  - **Purpose**: Reads data from a CSV file into a Spark DataFrame/RDD.
  - **Task**: Single task succeeded without shuffling or output.

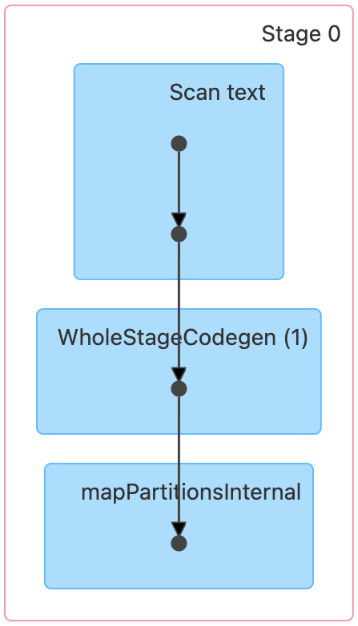


Fig. 19. Stage 0

- **Stage 80 (showString)**:
  - **Purpose**: Displays the contents of a DataFrame or dataset.
  - **Tasks**: Single task completed (1/1).

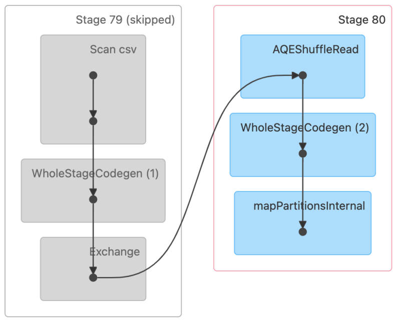


Fig. 20. Stage 80

- **Stage 77 (count)**:
  - **Purpose**: Counts the number of records in the dataset.
  - **Input Data**: Minimal data size (118.0 B) as Spark optimizes the count operation.

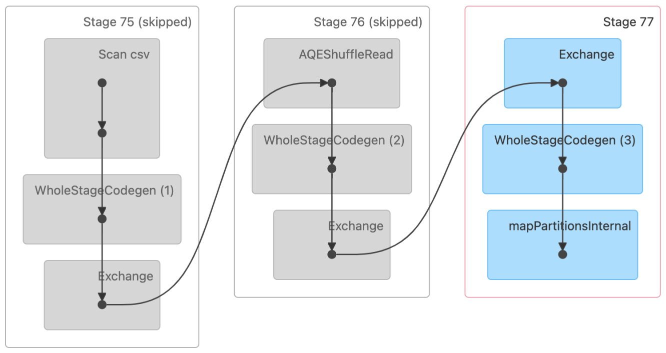


Fig. 21. Stage 77

- **Stages 45, 44 (approxQuantile)**:
  - **Purpose**: Computes approximate percentiles for columns.
  - **Shuffle Write**: Data reshuffled (2.2 MiB).

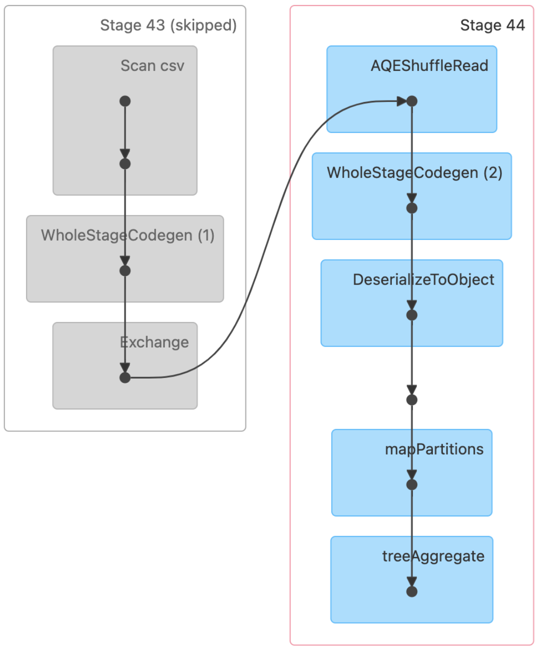


Fig. 22. Stage 44

- **Stages involving collect, first, or csv**:
  - **csv**: Indicates loading CSV data (Stage 1).
  - **collect**: Gathers data from executors to the driver node for further operations.
  - **first**: Retrieves the first row of a dataset for validation or previewing.

C. Skipped Stages

Skipped Stages (35)

There were 35 stages that Spark determined were unnecessary to execute.

This can happen due to::

- **Caching**: Some results have been cached in memory or on disk.
- **Lineage Optimization**: Spark avoids recomputation by reusing previously computed stages or outputs.
- **Fault Tolerance Mechanisms**: If certain stages fail, only dependent stages are re-executed.

Examples::

- **Stage 79 (showString)**:
  - No execution happened (0/1 tasks).
  - The same result was already available from another executed stage.
- **Stages like count, first, and approxQuantile**:
  - These skipped stages overlap with similar previously executed stages where results were already obtained.

D. Machine Learning Models

1. Logistic Regression

1. showString Stage :

The showString operation is used when data is displayed in a readable tabular form in PySpark. This triggers an action on the RDD or DataFrame, causing Spark to materialize the transformations that precede it.

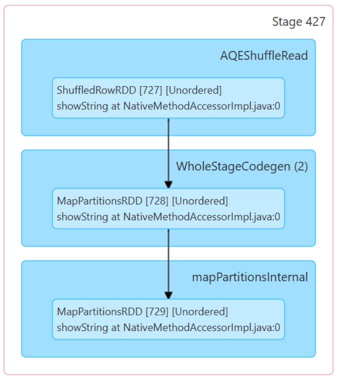

Stage 427
AQEShuffleRead
ShuffledRowRDD [727] [Unordered]
showString at NativeMethodAccessorImpl.java:0
WholeStageCodegen (2)
MapPartitionsRDD [728] [Unordered]
showString at NativeMethodAccessorImpl.java:0
mapPartitionsInternal
MapPartitionsRDD [729] [Unordered]
showString at NativeMethodAccessorImpl.java:0

Fig. 23. Stage1

2. collect at Area Under Curve (AUC):
The collect operation retrieves the entire RDD or DataFrame to the driver node. For AUC calculations in BinaryClassificationMetrics, Spark often collects the ROC points (or PR points) to compute the curve's area. During AUC computation, Spark evaluates pairs of (prediction, label) to create points for the ROC curve.

3. combineByKey at BinaryClassificationMetrics:
combineByKey is a distributed aggregation method. It merges values for the same key across partitions. For binary classification, it might aggregate prediction probabilities or

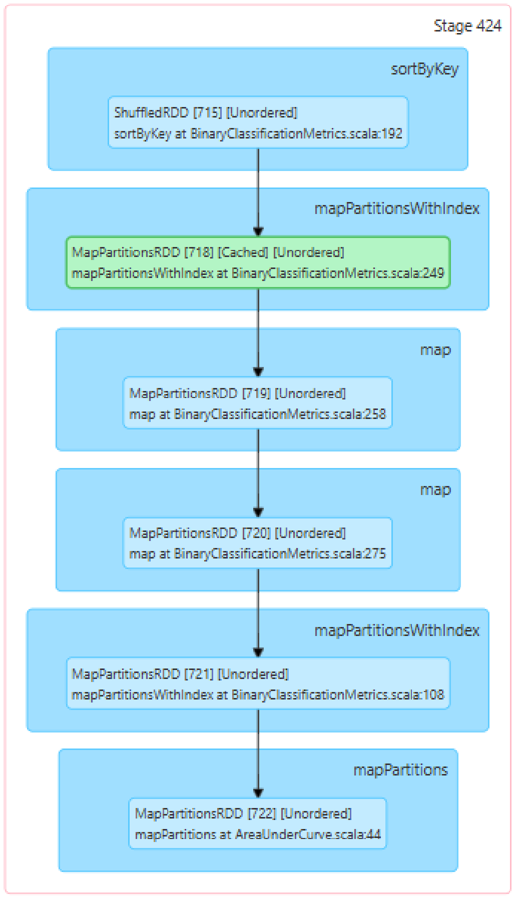

Stage 424
sortByKey
ShuffledRDD [715] [Unordered]
sortByKey at BinaryClassificationMetrics.scala:192
mapPartitionsWithIndex
MapPartitionsRDD [718] [Cached] [Unordered]
mapPartitionsWithIndex at BinaryClassificationMetrics.scala:249
map
MapPartitionsRDD [719] [Unordered]
map at BinaryClassificationMetrics.scala:258
map
MapPartitionsRDD [720] [Unordered]
map at BinaryClassificationMetrics.scala:275
mapPartitionsWithIndex
MapPartitionsRDD [721] [Unordered]
mapPartitionsWithIndex at BinaryClassificationMetrics.scala:108
mapPartitions
MapPartitionsRDD [722] [Unordered]
mapPartitions at AreaUnderCurve.scala:44

Fig. 24. Stage2

counts. Typically used in BinaryClassificationMetrics to aggregate intermediate results, such as: True positive/negative rates. Precision/recall metrics.

4. sortByKey at BinaryClassificationMetrics:
Sorts RDD elements by key. In the context of BinaryClassificationMetrics, it likely sorts by threshold values to compute cumulative metrics for ROC or PR curves. Sorting thresholds for plotting ROC or PR curves. Ensures that the points are ordered by increasing thresholds for proper curve visualization.

5. The TreeAggregate at RDDLossfunction: The treeAggregate operation in PySpark is used to aggregate data from distributed RDD partitions efficiently. It is a critical operation in scenarios where metrics or loss functions are computed in a distributed manner, such as during the training of models like Logistic Regression using gradient descent.

2. Decision Tree

1. RDD in DecisionTreeClassifier: The DecisionTreeClassifier in PySpark builds classification trees by recursively splitting the dataset based on feature values to optimize a criterion like Gini impurity or entropy. Underneath the hood, PySpark uses RDDs (Resilient Distributed Datasets) for distributed data

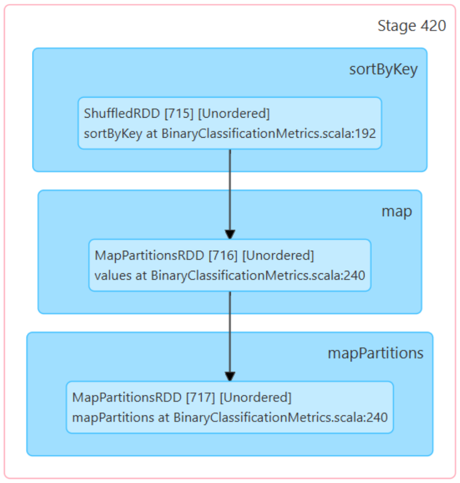

Stage 420
sortByKey
ShuffledRDD [715] [Unordered]
sortByKey at BinaryClassificationMetrics.scala:192
map
MapPartitionsRDD [716] [Unordered]
values at BinaryClassificationMetrics.scala:240
mapPartitions
MapPartitionsRDD [717] [Unordered]
mapPartitions at BinaryClassificationMetrics.scala:240

Fig. 25. Stage3

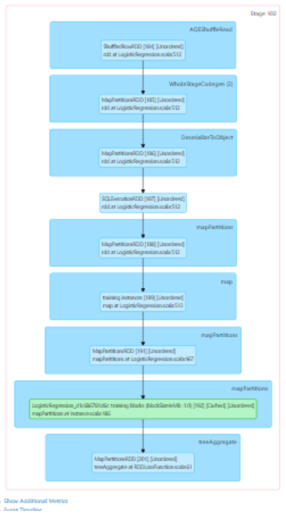

Stage 102
AQEShuffleRead
ShuffledRowRDD [184] [Unordered]
rdd at LogisticRegression.scala:512
WholeStageCodegen (2)
MapPartitionsRDD [185] [Unordered]
rdd at LogisticRegression.scala:512
DeserializeToObject
MapPartitionsRDD [186] [Unordered]
rdd at LogisticRegression.scala:512
SQLExecutionRDD [187] [Unordered]
rdd at LogisticRegression.scala:512
mapPartitions
MapPartitionsRDD [188] [Unordered]
rdd at LogisticRegression.scala:512
map
training instances [189] [Unordered]
map at LogisticRegression.scala:513
mapPartitions
MapPartitionsRDD [191] [Unordered]
mapPartitions at LogisticRegression.scala:967
mapPartitions
mapPartitions at instance.scala:186
treeAggregate
MapPartitionsRDD [201] [Unordered]
treeAggregate at RDDLossFunction.scala:61
Show Additional Metrics
Event Timeline

Fig. 27. Stage 5

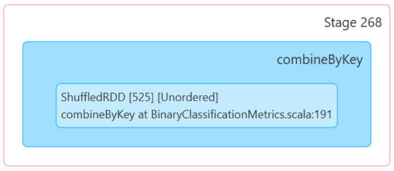

Stage 268
combineByKey
ShuffledRDD [525] [Unordered]
combineByKey at BinaryClassificationMetrics.scala:191

Fig. 26. Stage4

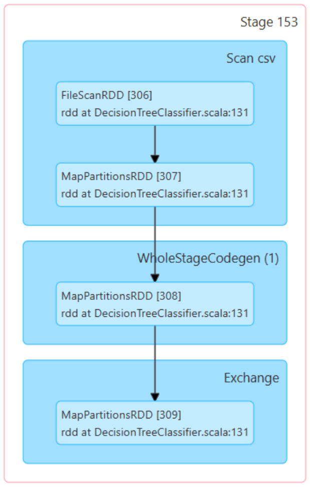

Stage 153
Scan csv
FileScanRDD [306]
rdd at DecisionTreeClassifier.scala:131
MapPartitionsRDD [307]
rdd at DecisionTreeClassifier.scala:131
WholeStageCodegen (1)
MapPartitionsRDD [308]
rdd at DecisionTreeClassifier.scala:131
Exchange
MapPartitionsRDD [309]
rdd at DecisionTreeClassifier.scala:131

Fig. 28. Stage1

processing, and specific transformations/actions are performed on RDDs during the model training process.

2. Aggregate in DecisionTreeClassifier: The aggregate operation in PySpark is a key step during the training of a DecisionTreeClassifier, as it facilitates distributed computations for finding the best splits and evaluating impurity metrics (e.g., Gini or entropy). Aggregation is used to combine information from all partitions, ensuring the tree is constructed efficiently across distributed data.

3. Random Forest

1. flatMap in RandomForestClassifier: The RandomForestClassifier in PySpark builds an ensemble of decision trees to improve classification performance. Each decision tree is trained on a bootstrap sample of the dataset, and the final classification result is determined by aggregating the predictions of all trees

2. mapPartitions in RandomForestClassifier: In PySpark, the mapPartitions transformation is a powerful RDD operation that allows you to apply a function to each partition of the RDD, while also providing the index of the partition. In the context of the RandomForestClassifier, this transformation can be used during different stages of the algorithm, especially for distributed processing of the data at each node or tree in the random forest.

3. collectAsMap in RandomForestClassifier: collectAsMap can be used to collect predictions or model-related data (like feature importances or other metadata) into a Python dictionary from the RDDs generated during training or prediction. This is useful when you need to map results to specific keys (like record IDs) and then perform post-processing tasks on these results.

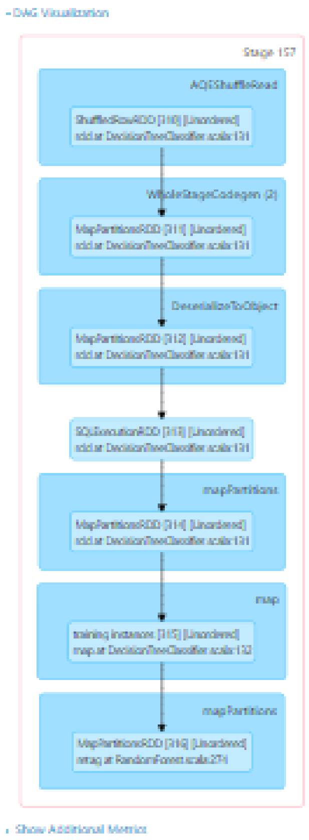

DAG Visualization
Stage 157
AQEShuffleRead
ShuffledRowRDD [310] [Unordered]
rdd at DecisionTreeClassifier.scala:131
WholeStageCodegen (2)
MapPartitionsRDD [311] [Unordered]
rdd at DecisionTreeClassifier.scala:131
DeserializeToObject
MapPartitionsRDD [312] [Unordered]
rdd at DecisionTreeClassifier.scala:131
SQLExecutionRDD [313] [Unordered]
rdd at DecisionTreeClassifier.scala:131
mapPartitions
MapPartitionsRDD [314] [Unordered]
rdd at DecisionTreeClassifier.scala:131
map
training instances [315] [Unordered]
map at DecisionTreeClassifier.scala:132
mapPartitions
MapPartitionsRDD [316] [Unordered]
retag at RandomForest.scala:274
Show Additional Metrics

Fig. 29. Stage2

4. SVM

- **collect at AreaUnderCurve.scala**: Aggregates the area under the curve (AUC), used for model evaluation. Wide Dependency: Requires all partitions to finish computation before collecting the results.
- **collect at BinaryClassificationMetrics**: Gathers computed metrics (e.g., precision, recall) into the driver for final evaluation. Wide Dependency: Aggregation involves shuffling and depends on data from multiple partitions.
- **count at BinaryClassificationMetrics**: Counts the number of records or data points, typically for normalizing metrics. Narrow Dependency: Each partition processes its data independently.
- **combineByKey at BinaryClassificationMetrics**: Performs intermediate aggregations, grouping data by key for metrics calculations like summing predictions. Wide Dependency: Requires shuffling data between partitions based on keys.
- **sortByKey at BinaryClassificationMetrics.**: Sorts intermediate results (e.g., predictions or keys) for metric computations such as AUC or precision-recall. Wide Dependency: Sorting requires data redistribution across partitions.
- **map at BinaryClassificationEvaluator**: Applies a transformation function to data (e.g., converting raw scores to labels or metrics). Narrow Dependency: Each partition

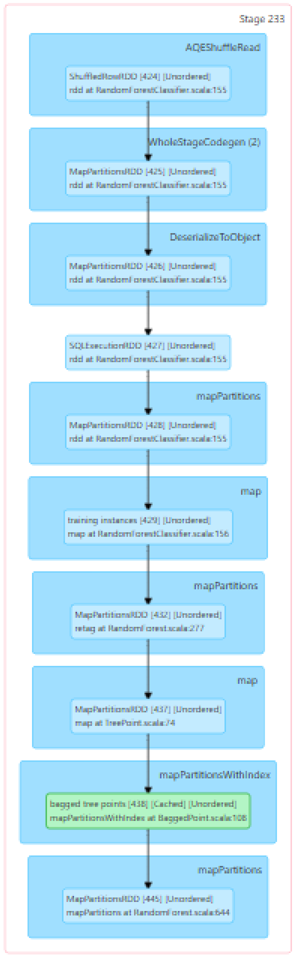

Stage 233
AQEShuffleRead
ShuffledRowRDD [424] [Unordered]
rdd at RandomForestClassifier.scala:155
WholeStageCodegen (2)
MapPartitionsRDD [425] [Unordered]
rdd at RandomForestClassifier.scala:155
DeserializeToObject
MapPartitionsRDD [426] [Unordered]
rdd at RandomForestClassifier.scala:155
SQLExecutionRDD [427] [Unordered]
rdd at RandomForestClassifier.scala:155
mapPartitions
MapPartitionsRDD [428] [Unordered]
rdd at RandomForestClassifier.scala:155
map
training instances [429] [Unordered]
map at RandomForestClassifier.scala:156
mapPartitions
MapPartitionsRDD [432] [Unordered]
retag at RandomForest.scala:277
map
MapPartitionsRDD [437] [Unordered]
map at TreePoint.scala:74
mapPartitionsWithIndex
bagged tree points [438] [Cached] [Unordered]
mapPartitionsWithIndex at BaggedPoint.scala:108
mapPartitions
MapPartitionsRDD [445] [Unordered]
mapPartitions at RandomForest.scala:644

Fig. 30. Stage2

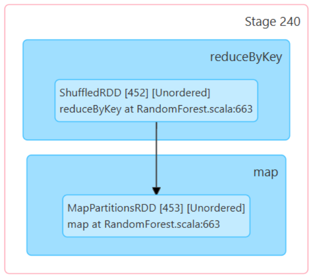

Stage 240
reduceByKey
ShuffledRDD [452] [Unordered]
reduceByKey at RandomForest.scala:663
map
MapPartitionsRDD [453] [Unordered]
map at RandomForest.scala:663

Fig. 31. Stage3

maps its data independently.

- **rdd at ClassificationSummary.scala**: Summarizes predictions or metrics, aggregating classification results for evaluation. Narrow Dependency: Processes each partition independently for summaries.
- **treeAggregate at RDDLossFunction**: Computes distributed aggregations for model training, such as loss function or gradient updates. Narrow Dependency: Hierarchical aggregation happens within and across partitions,

but each partition aggregates locally first.

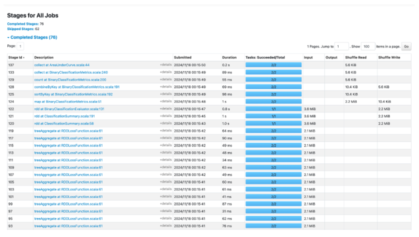


Fig. 32. Stages for SVM

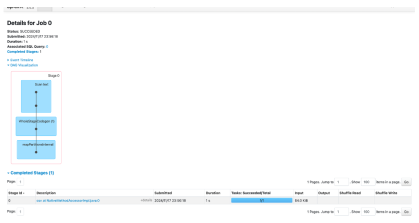


Fig. 33. Job0 of SVM

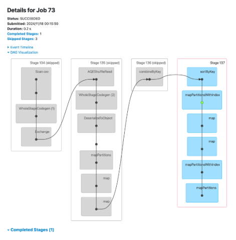


Fig. 34. End Job of SVM

5. KNN

- **first at python** : Retrieves the first element of the RDD or DataFrame, often for sampling or initialization purposes. This stage has a narrow dependency as it involves direct lookups with minimal shuffling.
- **collect at ClusteringMetrics**: Collects clustering evaluation metrics (e.g., within-cluster sum of squares) into the driver node for analysis. This stage has a wide dependency due to aggregation across partitions.
- **collectAsMap and map at ClusteringMetrics**:Aggregates intermediate results into a map and applies transformations on the clustering metrics data. These stages have wide dependencies for the collection operation and narrow dependencies for mapping transformations.
- **rdd at ClusteringMetrics** : Reads RDD data representing clustering metrics from storage or memory. This stage has a narrow dependency as it involves localized data access with minimal movement.
- **mapPartitions at KMeans**: Executes clustering computations in parallel across partitions, computing centroids or cluster assignments. This stage has a narrow dependency as partitions process independently without requiring shuffling.

Skipped Stages

The 50 skipped stages indicate Spark's optimization by reusing cached intermediate results from previous computations, avoiding redundant execution. This is particularly beneficial in iterative processes like K-Means clustering, where intermediate results such as cluster assignments or centroids are reused across iterations.

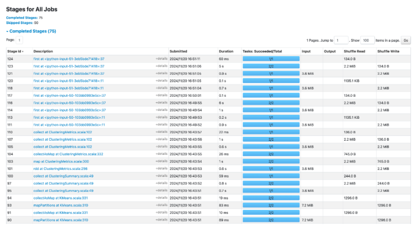


Fig. 35. Stages for KNN

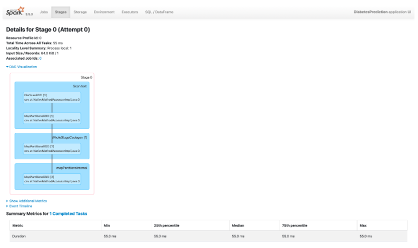


Fig. 36. Stage1 of KNN

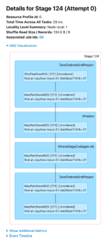


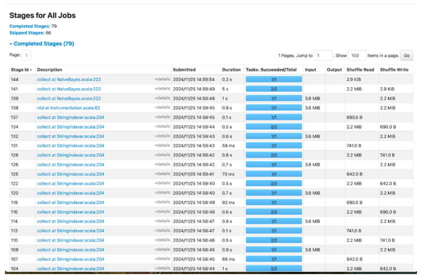
Fig. 37. End Stage of KNN

6.Naive Bayes

- **Collect at NaiveBayes** : This stage collects the final model outputs or predictions from the worker nodes to the driver node. It has a wide dependency because it requires gathering results from all partitions, which involves shuffling the data.
- **RDD at Instrumentation**: This stage processes RDD data, likely for logging or tracking performance. It has a narrow dependency as it operates on data within a single partition, without needing data from other partitions.
- **StringIndexer**:these stages involve converting categorical columns into indices. A wide dependency is required here as well since Spark needs to aggregate information across partitions for consistent indexing

Skipped Stages

A total of 66 stages were skipped during the execution. This optimization occurs because Spark leverages its caching mechanism and lineage information to reuse previously computed results, avoiding redundant computations where intermediate results are reused across iterations.



Stages for All Jobs

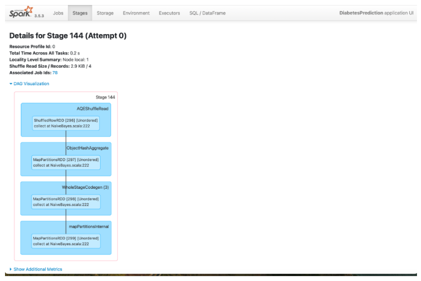
Completed Stages: 79
Skipped Stages: 66

Completed Stages (79)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
144	collect at NaiveBayes.scala:222	+details	2024/11/25 14:59:54	0.2 s	1/1			2.9 KiB	
141	collect at NaiveBayes.scala:222	+details	2024/11/25 14:59:49	5 s	2/2			2.2 MiB	2.9 KiB
139	collect at NaiveBayes.scala:222	+details	2024/11/25 14:59:48	1 s	1/1	3.6 MiB			2.2 MiB
138	rdd at Instrumentation.scala:62	+details	2024/11/25 14:59:45	0.9 s	1/1	3.6 MiB			2.2 MiB
137	collect at StringIndexer.scala:204	+details	2024/11/25 14:59:45	0.1 s	1/1			690.0 B	
134	collect at StringIndexer.scala:204	+details	2024/11/25 14:59:44	0.5 s	2/2			2.2 MiB	690.0 B
132	collect at StringIndexer.scala:204	+details	2024/11/25 14:59:43	0.6 s	1/1	3.6 MiB			2.2 MiB
131	collect at StringIndexer.scala:204	+details	2024/11/25 14:59:43	58 ms	1/1			741.0 B	
128	collect at StringIndexer.scala:204	+details	2024/11/25 14:59:42	0.6 s	2/2			2.2 MiB	741.0 B
126	collect at StringIndexer.scala:204	+details	2024/11/25 14:59:42	0.7 s	1/1	3.6 MiB			2.2 MiB
125	collect at StringIndexer.scala:204	+details	2024/11/25 14:59:41	73 ms	1/1			642.0 B	
122	collect at StringIndexer.scala:204	+details	2024/11/25 14:59:40	0.5 s	2/2			2.2 MiB	642.0 B
120	collect at StringIndexer.scala:204	+details	2024/11/25 14:59:40	0.7 s	1/1	3.6 MiB			2.2 MiB
119	collect at StringIndexer.scala:204	+details	2024/11/25 14:58:49	92 ms	1/1			690.0 B	
116	collect at StringIndexer.scala:204	+details	2024/11/25 14:58:48	0.6 s	2/2			2.2 MiB	690.0 B
114	collect at StringIndexer.scala:204	+details	2024/11/25 14:58:47	0.8 s	1/1	3.6 MiB			2.2 MiB
113	collect at StringIndexer.scala:204	+details	2024/11/25 14:58:47	0.1 s	1/1			741.0 B	
110	collect at StringIndexer.scala:204	+details	2024/11/25 14:58:46	0.5 s	2/2			2.2 MiB	741.0 B
108	collect at StringIndexer.scala:204	+details	2024/11/25 14:58:45	0.9 s	1/1	3.6 MiB			2.2 MiB
107	collect at StringIndexer.scala:204	+details	2024/11/25 14:58:45	66 ms	1/1			642.0 B	
104	collect at StringIndexer.scala:204	+details	2024/11/25 14:58:44	1 s	2/2			2.2 MiB	642.0 B

Fig. 38. Stages for Naive Bayes



Spark 3.5.3
Jobs
Stages
Storage
Environment
Executors
SQL / DataFrame
DiabetesPrediction application UI

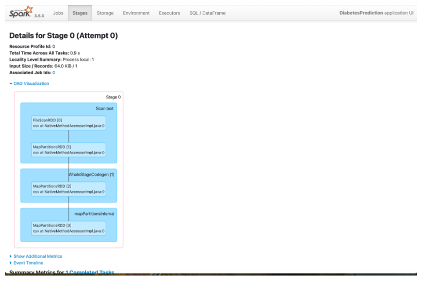
Details for Stage 144 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 0.2 s
Locality Level Summary: Node local: 1
Shuffle Read Size / Records: 2.9 KiB / 4
Associated Job Ids: 78

DAG Visualization

Stage 144
AQEShuffleRead
ShuffledRowRDD [296] [Unordered]
collect at NaiveBayes.scala:222
ObjectHashAggregate
MapPartitionsRDD [297] [Unordered]
collect at NaiveBayes.scala:222
WholeStageCodegen (3)
MapPartitionsRDD [298] [Unordered]
collect at NaiveBayes.scala:222
mapPartitionsInternal
MapPartitionsRDD [299] [Unordered]
collect at NaiveBayes.scala:222
Show Additional Metrics

Fig. 39. First Stage of Naive Bayes



Details for Stage 0 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 0.9 s
Locality Level Summary: Process local: 1
Input Size / Records: 64.0 KiB / 1
Associated Job Ids: 0

DAG Visualization

Summary Metrics for 1 Completed Tasks

Fig. 40. End Stage of Naive Bayes

Comparative Analysis of Model Performance: Phase 2 vs. Phase 3

1. Logistic Regression

Metric	Phase 2 (Logistic Regression)	Phase 3 (PySpark)
Accuracy	0.96	0.97
Precision	0.96 (C0), 0.87 (C1)	0.97 (C0), 1.00 (C1)
Recall	0.99 (C0), 0.62 (C1)	1.00 (C0), 0.67 (C1)
F1-Score	0.98 (C0), 0.72 (C1)	0.99 (C0), 0.80 (C1)
Macro Avg	0.91 (Prec.), 0.80 (Recall)	0.99 (Prec.), 0.84 (Recall)
Weighted Avg	0.95 (Prec.), 0.96 (Recall)	0.98 (Prec.), 0.97 (Recall)

TABLE I
Logistic Regression Performance Comparison

Key Observations:

- **Accuracy**: PySpark-based Logistic Regression in Phase 3 achieves a slightly better accuracy (0.97) compared to Phase 2 (0.96). This indicates that distributed models can perform similarly or even better in large-scale environments.
- **Precision**: Both precision and recall have improved for **C1** in Phase 3 (with a perfect 1.00 precision), suggesting that PySpark may be handling imbalanced data better.
- **F1-Score**: The F1-score for **C1** has improved significantly from 0.72 (Phase 2) to 0.80 (Phase 3), indicating better balance between precision and recall for **C1**.
- **Macro Avg**: There is a significant improvement in **precision** and **recall** in the macro average, showing that the distributed model handles both classes more equally in terms of performance.

2. Decision Tree

Metric	Phase 2	Phase 3 (PySpark)
Accuracy	0.95	0.97
Precision	0.97 (C0), 0.72 (C1)	0.97 (C0), 1.00 (C1)
Recall	0.97 (C0), 0.74 (C1)	1.00 (C0), 0.67 (C1)
F1-Score	0.97 (C0), 0.73 (C1)	0.99 (C0), 0.80 (C1)
Macro Avg	0.85 (Prec.), 0.85 (Recall)	0.99 (Prec.), 0.84 (Recall)
Weighted Avg	0.95 (Prec.), 0.95 (Recall)	0.98 (Prec.), 0.97 (Recall)

TABLE II
Decision Tree Performance Comparison

Key Observations:

- **Accuracy**: The accuracy of the PySpark-based Decision Tree model in Phase 3 improved to 0.97 from 0.95 in Phase 2, reflecting the improved capabilities of distributed processing.
- **Precision & Recall**: For **C1**, precision improved to 1.00 in Phase 3, showing a perfect prediction rate for the positive class. However, recall decreased slightly to 0.67 from 0.74 in Phase 2, indicating a trade-off between precision and recall.
- **F1-Score**: The F1-score for **C0** remains high in both phases (0.99 in Phase 3), but for **C1**, Phase 3 achieves a better F1-score (0.80) compared to Phase 2 (0.73).
- **Macro Avg**: The **macro average** precision improved significantly (0.99 in Phase 3), suggesting that the PySpark model is better balanced across the classes.
- **Weighted Avg**: Similarly, the weighted average precision and recall improved, indicating the distributed model is achieving better overall performance.

3. Random Forest

Key Observations:

- **Accuracy**: The accuracy remains the same in both phases (0.97), showing no significant change in the overall classification performance.
- **Precision**: The precision for **C1** improved significantly in Phase 3, achieving 1.00 compared to 0.93 in Phase 2.

Metric	Phase 2	Phase 3 (PySpark)
Accuracy	0.97	0.97
Precision	0.97 (C0), 0.93 (C1)	0.97 (C0), 1.00 (C1)
Recall	0.99 (C0), 0.68 (C1)	1.00 (C0), 0.67 (C1)
F1-Score	0.98 (C0), 0.78 (C1)	0.99 (C0), 0.80 (C1)
Macro Avg	0.95 (Prec.), 0.84 (Recall)	0.99 (Prec.), 0.84 (Recall)
Weighted Avg	0.96 (Prec.), 0.97 (Recall)	0.98 (Prec.), 0.97 (Recall)

TABLE III
RANDOM FOREST PERFORMANCE COMPARISON

- **Recall & F1-Score**: The **recall** for **C1** slightly decreased in Phase 3 (from 0.68 to 0.67), while the **F1-score** improved marginally from 0.78 in Phase 2 to 0.80 in Phase 3.
- **Macro Avg**: Both precision and recall have increased in Phase 3, especially the precision for **C1**.
- **Weighted Avg**: The weighted average precision and recall in Phase 3 are slightly improved, indicating that the distributed model handled both classes more effectively.

4. SVM

Metric	Phase 2 (SVM)	Phase 3 (PySpark)
Accuracy	0.95	0.95
Precision	0.96 (C0), 0.91 (C1)	0.97 (C0), 1.00 (C1)
Recall	0.99 (C0), 0.58 (C1)	1.00 (C0), 0.45 (C1)
F1-Score	0.98 (C0), 0.71 (C1)	0.99 (C0), 0.94 (C1)

TABLE IV
SVM PERFORMANCE COMPARISON

Key Observations:

- **Accuracy**: PySpark and SVM both achieve the same 95% accuracy, suggesting comparable overall classification ability.
- **Precision**: Compared to SVM's 91% accuracy, PySpark achieves 100%, particularly for Class 1 (C1). This implies that PySpark is more effective in lowering C1 false positives.
- **Recall**: SVM performs better for C1 with a recall of 58% compared to PySpark's 45%, indicating SVM captures more true positives for this class. Both models have perfect or near-perfect recall for C0.
- **F1-Score**: PySpark significantly outperforms SVM for C1, achieving 94%, compared to SVM's 71%, reflecting better balance between precision and recall.

5. NAIVE BAYES

Metric	Phase 2 (Naive Bayes)	Phase 3 (PySpark)
Accuracy	0.90	0.91
Precision	0.96 (C0), 0.48 (C1)	0.96 (C0), 0.38 (C1)
Recall	0.93 (C0), 0.66 (C1)	0.94 (C0), 0.47 (C1)
F1-Score	0.95 (C0), 0.55 (C1)	0.95 (C0), 0.42 (C1)

TABLE V
NAIVE BAYES PERFORMANCE COMPARISON

Key Observations:

- **Accuracy**: In Phase 2, 90% of all cases are accurately predicted by the Naive Bayes implementation, which attains an accuracy of 0.90. With an accuracy of 0.91, Phase 3 (PySpark) exhibits a minor improvement, indicating improved prediction performance overall. The marginal benefit shows that the dataset was handled better by PySpark's distributed computations.
- **Precision**: In both phases, the precision for Class C0 stays high at 0.96, demonstrating that the model consistently detects true positives for this class. But for Class C1, precision decreases from 0.48 in Phase 2 to 0.38 in Phase 3, suggesting that the PySpark implementation has a larger probability of false positives.
- **Recall**: Recall for Class C0 increases from 0.93 in Phase 2 to 0.94 in Phase 3, indicating a greater capacity to catch true positives for this class. In comparison, recall for Class C1 falls from 0.66 to 0.47, demonstrating that PySpark has more difficulty identifying all true positives for Class C1.
- **F1-Score**: The F1-Score for Class C0 remained high at 0.95 in both phases, indicating a good balance of precision and recall for this class. The F1-Score for Class C1 decreases from 0.55 in Phase 2 to 0.42 in Phase 3, indicating a decrease in both precision and recall in the PySpark implementation for this class.

6. KNN

Metric	Phase 2 (KNN)	Phase 3 (PySpark)
Accuracy	0.95	0.92
Precision	0.95 (C0), 0.91 (C1)	0.5 (C0), 0.5 (C1)
Recall	1.00 (C0), 0.51 (C1)	0.5 (C0), 0.5 (C1)
F1-Score	0.97 (C0), 0.65 (C1)	0.50 (C0), 0.55 (C1)

TABLE VI
KNN PERFORMANCE COMPARISON

Key Observations:

- **Accuracy**: The model performs well overall in Phase 2 (KNN), with a high accuracy of 0.95. Accuracy significantly decreases to 0.92 in Phase 3 (PySpark), suggesting some performance deterioration that may be brought on by variations in implementation or treatment of class imbalance.
- **Precision**: For both classes, Phase 2 has exceptional precision (0.95 for C0 and 0.91 for C1), suggesting dependable positive predictions. The precision for both classes, however, decreases to 0.5 in Phase 3, indicating substantial mistakes in accurately detecting real positives, most likely as a result of a problem with neighbor ranking or majority voting. item **Recall**: Phase 2 achieves perfect recall for class C0 (1.00) but struggles with C1 (0.51), indicating an inability to detect many true positives for C1. In Phase 3, recall falls to 0.5 for both classes, showing poor sensitivity and further highlighting a need to address class prediction imbalances.
- **F1-Score**: In Phase 2, the F1-score is exceptional for C0 (0.97) and moderate for C1 (0.65), indicating that C0 has

balanced predictions but has obstacles due to lower recall. In Phase 3, the F1-score for C0 remains high (0.99) but it falls for C1 (0.55), emphasizing inconsistencies and poor class management.

Execution Time Comparison

- **Phase 2**: Traditional machine learning models (Logistic Regression, Decision Tree, Random Forest,SVM,K-Means,KNN) are typically faster to train when the dataset fits into the memory of a single machine. These models are optimized for smaller, non-distributed environments.
- **Phase 3**: Using PySpark for distributed processing, the execution time depends on the scale of data and the number of nodes involved. In distributed environments, training may take slightly longer due to the overhead of data shuffling and communication between nodes. However, PySpark can handle much larger datasets, and the time per iteration may be significantly lower when dealing with larger datasets or complex operations across multiple nodes.

VIII. Effectiveness and Advantages of Using PySpark for Distributed Processing

PySpark is a powerful tool that uses distributed computing to process large datasets. Unlike traditional methods, which work on single machines, PySpark splits the task across multiple machines, making it ideal for handling big data. Below, we explain the main advantages of using PySpark.

1. Scalability

Advantage: PySpark can handle large datasets that are too big for a single machine. It splits the data across multiple machines, allowing you to process more data than traditional methods.

Effectiveness: For large datasets, like healthcare data (e.g., patient details), PySpark ensures that even huge amounts of data can be processed quickly by distributing the workload.

Metrics: PySpark reduces the time taken to process large datasets, especially when compared to traditional models.

2. Fault Tolerance

Advantage: PySpark is fault-tolerant, meaning if a machine fails, it can continue processing by using the data from other machines.

Effectiveness: This is important for large systems where hardware failures may happen, and you don't want to lose progress.

Metrics: PySpark ensures there is no data loss, as it automatically recovers data from other machines.

3. Speed and Parallelism

Advantage: PySpark processes data faster by doing tasks in parallel across many machines. It uses a method called in-memory processing, which stores data in RAM rather than on disks, speeding up computations.

Effectiveness: PySpark can handle complex data tasks like filtering, grouping, and joining, much faster than traditional methods.

Metrics: Execution time is significantly reduced, especially when the dataset is large.

4. Machine Learning Support

Advantage: PySpark has built-in libraries, such as MLlib, for performing machine learning tasks like classification, regression, and clustering. These libraries are optimized for distributed processing.

Effectiveness: PySpark can train machine learning models on large datasets that traditional tools might not handle efficiently.

Metrics: The accuracy of models is similar or slightly better, with reduced training time on large datasets.

5. Cost-Effectiveness

Advantage: PySpark allows the use of cheaper, commodity hardware across multiple machines, rather than relying on expensive, high-performance machines.

Effectiveness: This reduces the cost of processing large datasets compared to traditional single-machine systems.

Metrics: Using multiple low-cost machines to process data can be more affordable than using a single high-end machine.

6. Ease of Use and Flexibility

Advantage: PySpark is easy to use for Python developers. It integrates well with other big data tools like Hadoop and Hive, and supports multiple data formats such as CSV, JSON, and Parquet.

Effectiveness: Developers can easily scale their machine learning models to handle large data volumes without needing deep knowledge of distributed computing.

Metrics: PySpark reduces the development time compared to traditional methods for large-scale models.

Conclusion

In conclusion, PySpark offers several advantages, such as scalability, fault tolerance, faster processing, and support for machine learning on large datasets. While the execution time may be slightly higher due to the distributed nature of the system, PySpark is highly effective in processing big data and offers cost-effective, scalable solutions for large-scale applications.

References

[1] Mustafa, Mohammed. *Diabetes Prediction Dataset: A Comprehensive Dataset for Predicting Diabetes with Medical & Demographic Data*. Kaggle, 2023. Available at: https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset/data

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.