

**BIG DATA ARCHITECTURE PROJECT**

**MEMBERS: NEEMA NDANU- 666457**

**MITCHELLE KANURI-666565**

**ROSE MASINDE- 666750**

**NATASHA KOKI- 665602**

**INSTALLATION AND CONFIGURATION GUIDE:**

**Introduction**

This document includes the way we achieve our goal, predictive analysis of the diabetes data, in the Hadoop ecosystem using tools like Apache Hadoop, Pig,Nifi Power BI and programming languages like Python.

**Apache NiFi (Master HDFS), Data Ingestion, and Preprocessing**

One of the first open-source data flow automation solutions is Apache NiFi, which was developed to streamline and accelerate data processing and transmission between various systems. This comprehensive handbook describes the painstaking procedure for setting up and configuring Apache NiFi, explaining how to make it a primary node on the Hadoop Distributed File System (HDFS). It also covers the steps involved in gathering data from many sources and the preparatory work required to analyse and apply the data later.

With its user-friendly interface and strong data processing capabilities, Apache NiFi presents itself as a suitable option for businesses looking to simplify their data integration and processing processes. Following the instructions in this guide will enable users to fully utilize NiFi and create reliable, custom data flow channels that address their unique needs.

**2. Installation of Apache NiFi:**

Implementing Apache NiFi: In addition to its many benefits, Apache NiFi is quite simple to install on a variety of operating systems, such as Windows, Linux, and macOS. In our case, we used a Linux OS. The steps are to download the NiFi binary package, extract the contents to a location of your choice, and start the NiFi service. Let us get into a detailed explanation of the process.

**2.1 Download NiFi:**

Follow these instructions to begin the process of getting Apache NiFi:

* Go here to open the official Apache NiFi webpage in your web browser.
* Choose the NiFi version that is appropriate for your operating system based on its architecture.
* To obtain the relevant NiFi binary package, click the provided download link. nifi.apache.org.

**2.2 Unzip NiFi:**

After downloading the NiFi binary package, follow these steps to begin the extraction process:

* Open a terminal window on your system if it runs Linux.
* Open the terminal and navigate to the directory where the NiFi binary package was downloaded.
* To extract the NiFi binary package into a directory of your choice, use the tar command as follows:

tar nifi-.tar.gz -xvf

**2.3 Start NiFi:**

Once NiFi is extracted, you can start the NiFi service by following these steps:

* Navigate to the directory where NiFi was extracted.
* Open a terminal window and execute the following command to start NiFi:
* Copy code

./bin/nifi.sh start

* This command will initiate the NiFi service, and you will see the status of NiFi displayed in the terminal window.

**2.4 Access NiFi UI (User Interface):**

To access the NiFi User Interface (UI) and begin configuring data flow pipelines, follow these steps:

* Open a web browser and enter the following URL in the address bar:
* Copy code

<http://localhost:8080/nifi>

* This will launch the NiFi UI, where you can create, manage, and monitor data flow processes using a visual interface.

**3. Configuration of NiFi (Master HDFS):**

Configuring Apache NiFi to function as the master node on the Hadoop Distributed File System (HDFS) involves connecting NiFi to HDFS, setting up NiFi as the master node, and ensuring high availability and fault tolerance. Here is a detailed guide on how to configure NiFi for optimal performance and reliability:

**3.1 Connect to HDFS:**

Make sure Hadoop and HDFS are correctly installed and setup on your Linux laptop before setting NiFi to operate with HDFS. The steps to connect NiFi to HDFS are as follows:

* Locate the conf folder by navigating to the installation location of NiFi.

Modify the nifi.properties file inside the conf folder to include the HDFS directory path and required Hadoop configuration files.

* In the nifi.properties file, include the locations to core-site.xml, hdfs-site.xml, and any other necessary Hadoop setup files.

**3.2 Set Up NiFi as Master on HDFS:**

Make sure Hadoop and HDFS are correctly installed and setup on your Linux laptop before setting NiFi to operate with HDFS. The steps to connect NiFi to HDFS are as follows:

* Locate the conf folder by navigating to the installation location of NiFi.

Modify the nifi.properties file inside the conf folder to include the HDFS directory path and required Hadoop configuration files.

* In the nifi.properties file, include the locations to core-site.xml, hdfs-site.xml, and any other necessary Hadoop setup files.

**3.3 Ensure High Availability and Fault Tolerance:**

As NiFi is the master node on HDFS, you need set up NiFi clusters for load balancing and failover to provide high availability and fault tolerance. To guarantee fault tolerance and high availability, take the following actions:

* Assign different NiFi instances to a cluster to balance the processing load and offer redundancy in the event of an outage.
* To ensure consistency and dependability, configure cluster nodes to interact with one another and synchronize their states.

**4.Data Ingestion:**

A variety of processors and connectors are offered by Apache NiFi to make it easier to import data from a variety of sources, such as local file systems, databases, APIs, and more. Identifying data sources, configuring data ingestion processors, and testing the data ingestion process are all part of setting up data ingestion pipelines in NiFi. Here's a comprehensive tutorial on utilizing Apache NiFi for data ingestion:

**4.1 Determine the Data Sources**:

Determine the sources from which data will be imported into NiFi prior to establishing data ingestion pipelines. In our instance, it consists of:

File systems locally (csv)

**Data Ingestion and Preprocessing Flow**:

**1. ListFile Processor**:

Files from a directory can be listed using the ListFile processor. The following lists the modifications and their justifications:

* Change to the Input Directory: The directory from which files should be listed was specified by modifying the input directory attribute.
* File Filter Modification: To filter files according to standards, including file extensions or naming conventions, the file filter property was changed.
* Relationship Interpretation: The 'failure' relationship reveals any errors that occurred during the listing process, whereas the 'success' relationship indicates that files have been correctly listed.

**2. UpdateAttribute Processor (1):**

This processor is used to add or change FlowFiles' attributes. The modifications are as follows:

* Additional Properties: To extract metadata from the FlowFiles, such as the original filename and path, more properties were added.
* Explanation: The 'original.path' property obtains the file's absolute path, but the 'original.filename' field uses the 'getDelimitedField' function to extract the filename.

For the preprocessing and cleaning up our data, the following was changed:

<processor>

<name>UpdateAttribute</name>

<! -- Existing Properties -->

<property name="filename">original.filename:getDelimitedField(1, '/') </property>

<property name="path">original.path:absolutePath</property>

<! -- Additional Properties for Preprocessing -->

<property name="includeHeaders">true</property>

<property name="removeDuplicates">true</property>

</processor>

**3. FetchFile Processor**:

Files can be retrieved from a distant or local file system using the FetchFile processor. The modifications are as follows:

* To Fetch File: To specify the precise file to obtain, the property defining the file to be fetched was changed.
* Relationship Changes: To handle various eventualities when fetching files, relationships like "failure," "not found," and "permission denied" were introduced.

**4. UpdateAttribute Processor (2):**

Additional FlowFiles attribute modifications are made using this second UpdateAttribute processor. The modifications are as follows:

* Adding Extra Properties: '.json' was added to the original filename by modifying the 'filename' field.
* Justification: This change makes sure that the output files contain the '.json' extension, which can be necessary for storing or analysing the files later.

For the preprocessing and cleaning up our data, the following was changed, after fetching the file:

</processor>

<name>UpdateAttribute</name>

<! -- Existing Properties -->

<property name="filename">original.filename:append('.json') </property>

<!-- Additional Properties for Preprocessing -->

<property name="handleMissingValues">true</property>

<property name="handleEmptySpaces">true</property>

</processor>

**5.PutHDFS Processor:**

FlowFiles are written to the Hadoop Distributed File System (HDFS) using the PutHDFS processor. The modifications are as follows:

* Resources for Configuring Hadoop: To supply the required configuration files, the attribute defining Hadoop configuration resources was changed.
* Directory Change: To indicate the destination directory in HDFS, the directory property was modified.
* Relationship Termination: To manage both successful and unsuccessful writing operations, "failure" and "success" relationships were terminated.

**Validate Preprocessing**: Check that data is cleaned, converted, and enhanced in accordance with the preprocessing criteria by testing the preprocessing procedure. To confirm the efficacy of preprocessing and guarantee the quality of the processed data, keep an eye on the data flow and examine the processed data in NiFi UI.

These modifications to the NiFi flow for preprocessing and data input helped us to make sure our pipeline for processing data is set up to handle different situations and efficiently prepare data for additional analysis or storage.

**Data processing conclusion**

This Apache NiFi Installation and Configuration Guide concludes by providing a thorough road map for customers to use NiFi for HDFS integration, preprocessing, and data intake. It offers comprehensive instructions for a smooth installation, setting up as an HDFS master node, and building strong data pipelines.

The tutorial covers advanced configurations for fault tolerance and high availability in addition to basic installation. It also provides information on establishing processor configurations, preprocessing step validation, and data ingestion pipeline setup.

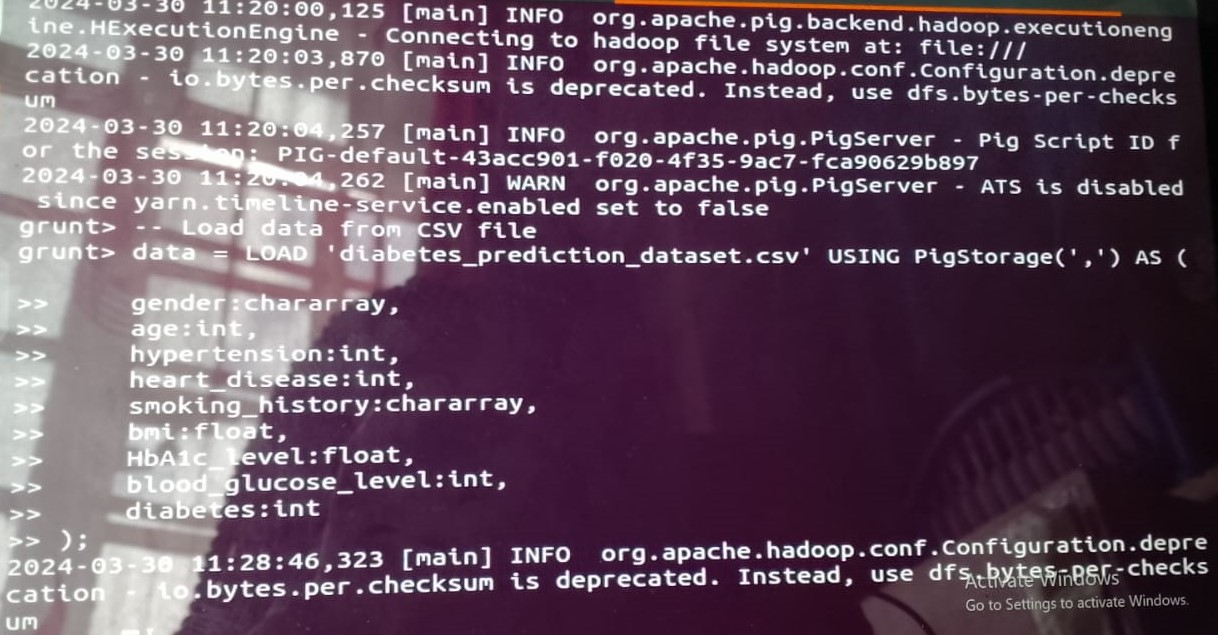
Through the utilization of NiFi processors and connectors, users may effectively collect and prepare data from various sources, incorporating it into their workflows with ease. Given the circumstances, this manual is an invaluable tool for businesses looking to maximize data integration and preparation using NiFi, allowing for more efficient workflows and insightful decisions.

**Data analysis**

Apache Hadoop and Apache Pig are utilized for efficient data processing and analysis. These tools excel in handling large datasets by levering distributed computing capabilities. In our case, we are analyzing diabetes related data to extract valuable insights that can inform decision making in healthcare.

To be able to achieve we followed through the following steps:

Download Apache Pig and python load the data into pig as indicated below:



Then using python, we created a python script that did the following:

1.Loaded the data using pandas.

2. Cleaned and preprocessed the data by removing records with missing values, removing duplicates and split the data into features (x) and target(y)

3. Split the data into training and testing data using “train\_test\_split”.

4. Apply one-hot encoding to categorical variables using “OneHotEncoder”.

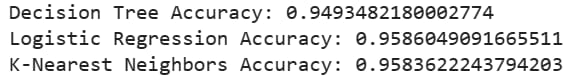
5. Scale numerical features using “StandardScaler”.

6. Concatenate encoded features with numerical features.

7. Train the ML models using Decision Tree, Logistic Regression. K-NN using the training datasets.

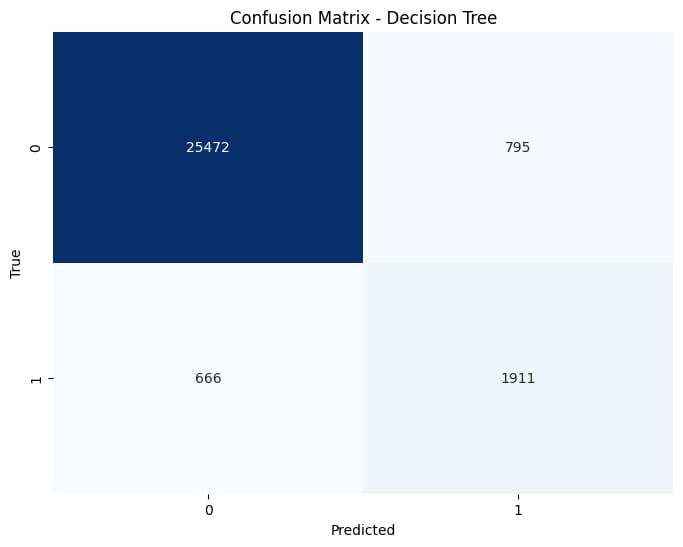
8. Make predictions on the testing data using trained models.

9. Calculate the accuracies of each model using “ accuracy\_score” and output the accuracies of each model.



10. Visualize the analysis for the Decision Tree model:

-> Plot confusion matrix using ‘sns.heatmap’

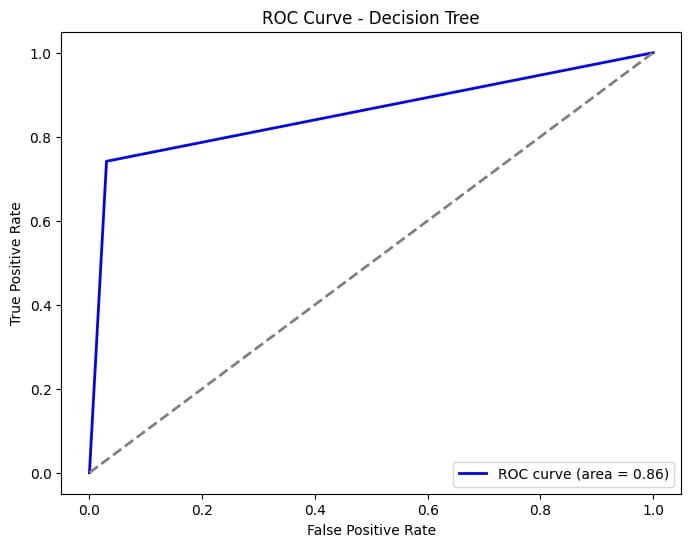


Explanation:

0 and 1 represent actual and predicted values. The confusion matrix is divided into four quadrants:

* Top left (True Positive): 25472
* Top right (False Positive): 795
* Bottom left (False Negative): 666
* Bottom right (True Negative): 19111

->Plot ROC curve using ‘plt.plot’



Explanation:

The ROC curve shows the trade-off between the true positive rate (TPR) and the false positive rate (FPR) at different classification thresholds.

The area under the ROC curve (AUC (Area Under the ROC Curve)) quantifies the model’s ability to distinguish between positive and negative classes.

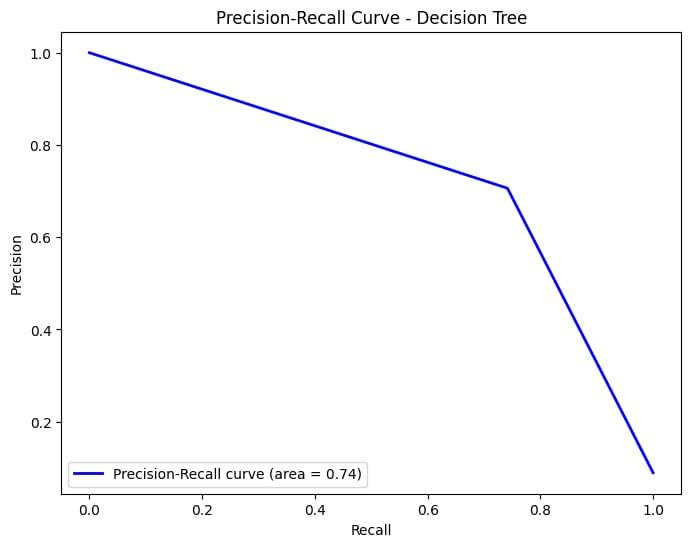
The ROC curve is drawn as a blue line.

The AUC value is displayed in the legend

A gray dashed line represents the line of no discrimination (random guessing)

The x-axis represents the false positive rate, and the y-axis represents the true positive rate.

-> Plot Precision-Recall curve using ‘plt.plot’



Explanation:  
The Precision-Recall curve shows the trade-off (the balance between two competing factors) between precision and recall at different classification thresholds.

Precision measures how many of the positive predictions were correct, while recall measures how many of them were correctly predicted.

11. Then using the ‘if\_ \_name\_ \_==” main\_ \_”:’ to block to ensure that the script can be run as standalone or imported as module., meaning it runs as the main program.

Then to facilitate the Python code into Pig data processing and to invoke python UDF (User Defined Function) the following were done:

-> Using ’REGISTER’ statement we registered python script (analysis.py) as a user –defined function (UDF). This means it tells Pug to treat the script as a resource that contains function or procedure that can be invoked from within the Pig.

-> Using ‘STORE’ statement we stored the data temporarily into a relation named ‘ temp\_data’ using Pig default storage format (‘PigStorage()’). This step is important to prepare the data for processing with the python UDF.

-> Using the ‘DEFINE’ statement we were able to bridge the gap between Pig’s native language and Python environment, allowing Pig to call Python function on the data.

-> Using ‘STREAM’ statement we processed the stored data.

**Data testing**

We used JUnit. We first installed JUnit and then maven. Then we configured the maven.xml and created JUnit as a dependency and started testing the data.

**Junit Installation and configuration**

**1.maven directory**

vboxuser@Ubunto:~$ mkdir my-project

vboxuser@Ubunto:~$ cd ~/my-project

vboxuser@Ubunto:~$ /my -project$ pwd #full path to your Maven project's root directory

/home/vboxuser/my-project

vboxuser@Ubunto:~$

**2.POM.XML FILE TEXT EDITOR**

Add JUnit in one of the dependencies using nano

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version> <! -- or the latest version available -->

<scope>test</scope>

</dependency>

**3.Install maven**

mvn clean install

**4.Test data using JUnit through src/java/test**

I used maven commands

**5.Using myclasstest.java. Now we are on to testing**.

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class MyClassTest {

@Test

public void testPredictDiabetesRisk\_NormalCase() {

// Prepare test data for a normal case

double glucoseLevel = 120;

double bmi = 25;

double bloodPressure = 0;

// Call the method being tested

DiabetesPredictor predictor = new DiabetesPredictor();

double risk = predictor.predictDiabetesRisk(glucoseLevel, bmi, bloodPressure);

// Assert the expected result

double expectedRisk = 0.2; // Example expected risk for normal case

assertEquals(expectedRisk, risk, 0.01); // Specify a delta for floating-point comparison

}

@Test

public void testPredictDiabetesRisk\_HighRiskCase() {

// Prepare test data for a high-risk case

double glucoseLevel = 200;

double bmi = 30;

double bloodPressure = 90;

// Call the method being tested

DiabetesPredictor predictor = new DiabetesPredictor();

double risk = predictor.predictDiabetesRisk(glucoseLevel, bmi, bloodPressure);

// Assert the expected result

double expectedRisk = 0.8; // Example expected risk for high-risk case

assertEquals(expectedRisk, risk, 0.01); // Specify a delta for floating-point comparison

}

}

**6. CREATE THE CLASS TEST IN THE ROOT DIRECTORY**

vboxuser@Ubunto:~/my-project/src/test/java$ nano home/vboxuser/my-project/pom.xml

vboxuser@Ubunto:~/my-project/src/test/java$ cd ~/my-project

vboxuser@Ubunto:~/my-project/src/test/java$ cd src/test/java

vboxuser@Ubunto:~/my-project/src/test/java$ echo'public class MyClassTest{' >

MyClassTest.java

vboxuser@Ubunto:~/my-project/src/test/java$nano MyClassTest.java

**7.TO CREATE A CLASS**

vboxuser@Ubunto:~/my-project/src/test/java$nano MyClass.Java

**8.TO START TESTING THE DATA**

Use the correct directory to run the class and the test. For example, I have used cd ~/my-project

Then confirm if pom.xml is in the directory using ls

**9.INPUT**

public class MyClass{

public String determinerisk(double risk) {

//double risk=0.1\*120+0.2\*25+0.2\*80

if (risk < 41) {

return "low risk";

}

else {

return "high risk";}

}

}

**10.MYCLASSTEST**

import org.junit.Test;

import static org.junit.Assert.\*;

public class MyClassTest {

@Test

public void testPredictDiabetesRisk\_NormalCase() {

MyClass predictor = new MyClass();

assertEquals("low risk",predictor.determinerisk(40)); // Specify a delta for floating-point comparison

}

@Test

public void testPredictDiabetesRisk\_HighRiskCase() {

MyClass predictor = new MyClass();

assertEquals("high risk",predictor.determinerisk(41));

}

}

**11.COMPILE THE TESTS**

Use cd ~/my-project directory to run the test

vboxuser@Ubunto:~my-project$ mvn package

The test will have 0 failures if correct

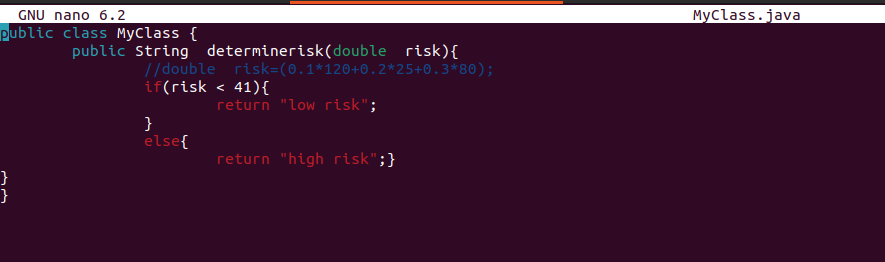
**Creating a java class**

So, we created a code that predicts whether one can develop a high risk of getting diabetes or a low risk.

We calculated the risk using the following formula:

RISK=0.1\*blood glucose level +0.2\*bmi+0.3\*hypertension

So, if the risk is above 41 you have a high chance of getting diabetes. When the risk is below 41 you have a low chance of getting diabetes.

  
**Creating a java test class**

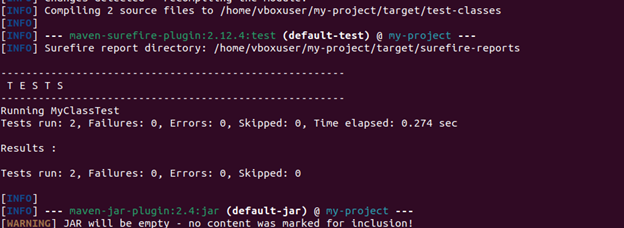
So, we test if the code is working, we predict the low risk, the risk to be less than 40 and the high risk to be more than or equal to 41.

A computer screen shot of a program code

Description automatically generated

**We tested using mvn packages**

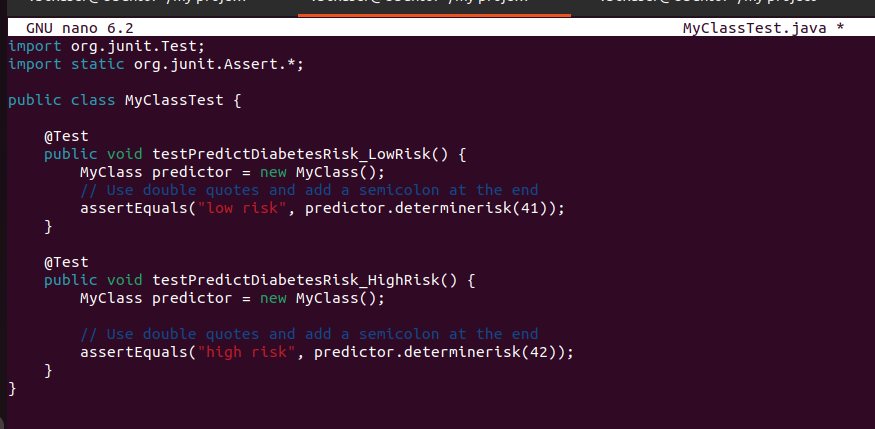
The test succeeded because the failures are zero



**To test if it works, it outputs the correct results’**

**Java class test**

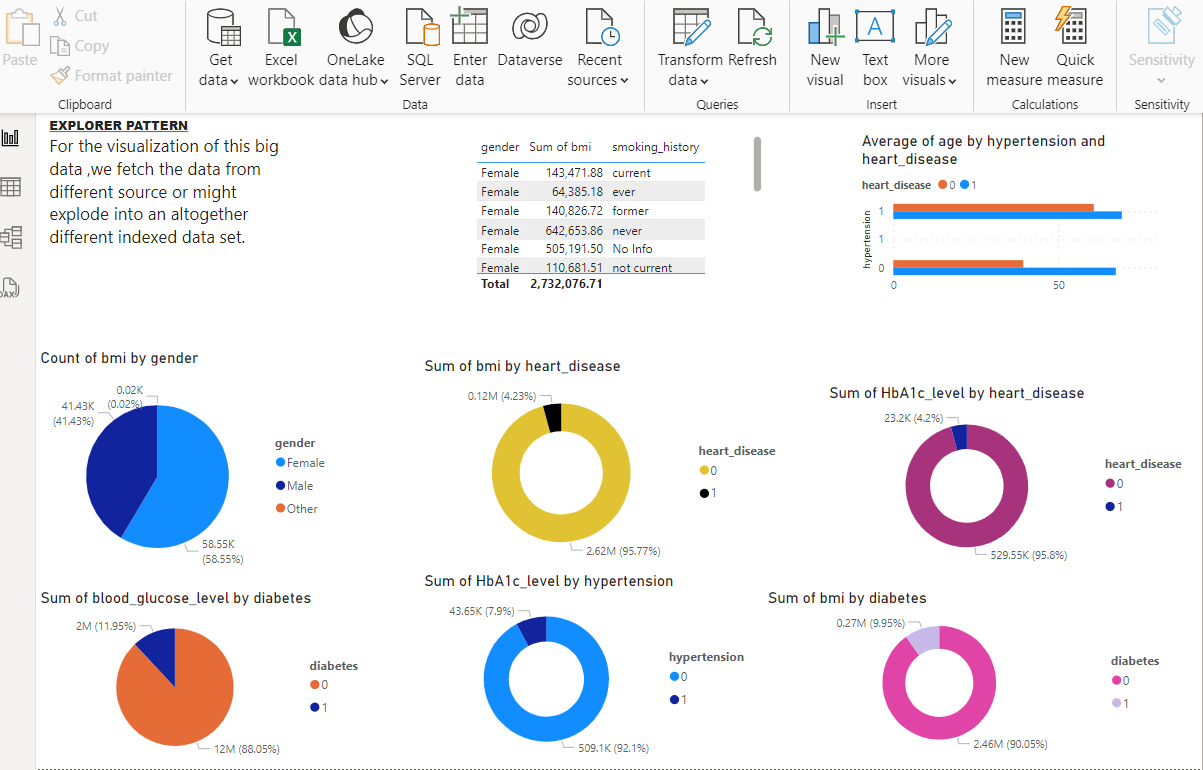
So, we input the risk as wrong to detect whether the code is working. We input low risk as 41 and high risk as 42.

  
**The test**

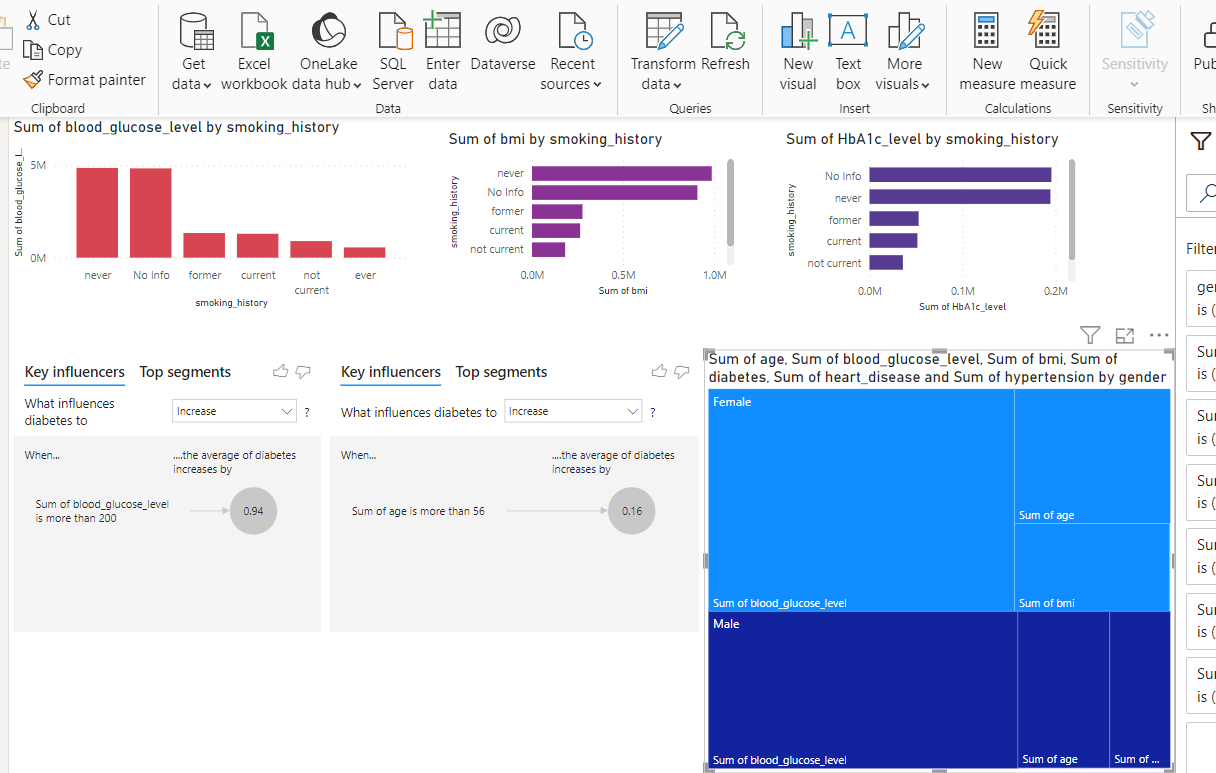
It outputs one failure.

A computer screen shot of a program

Description automatically generated  
**Data visualization**



1. A table that compares the sum of bmi to the gender and smoking history: A higher percentage of female and male, who have a high bmi are not smokers.
2. A chart about the average age of those who do not have heart disease and hypertension are in their forties whereas most people who are in their eighties have heart disease and hypertension.
3. Most females have a higher bmi(ratio to height and weight) than male and others.
4. 0 represents most people who have a higher sum of bmi tend to have heart disease and 1 represents the less.
5. (HbA1c is your average blood glucose (sugar) levels for the last two to three months), the sum of hba1c represents most people having a heart disease.
6. The sum of blood glucose level tends to have diabetes while 12 % of the population do not have diabetes.
7. The sum of hba1c levels tend to have hypertension, indicating it affects hypertension.
8. The sum of bmi increases indicating it is related to diabetes, majority who have a higher bmi have diabetes.



1. For the three column and bar charts, there is a trend where most people who have never smoked or there is no info about them have higher blood sugar level, bmi and hba1c level. Indicating that smoking history does not predict whether one can get diabetes or not.
2. The key influencers indicate the sum of blood glucose level over 200 increases the diabetes average by 0.94.
3. Diabetes increases by 0.16 when people are above 56 years of age, but it is not a must.
4. The last chart is the tree map that summarizes the sum of the records to the gender e.g. age, bmi, blood sugar level

**CONCLUSION**

The above information explains how we were able to ingest the data using Apache Flume, do predictive analysis using Apache Hadoop, Pig and Python, do testing with JUnit and do visualization with Power BI. Though our journey to be able to accomplish was not too easy we were able to learn a few things such as how to use the various tools that are there, that for a tool to work we need it to be compatible with your device, communication between members and finally the most important how a Hadoop ecosystem works.