Statistical Machine Learning for Big Data Analysis with Spark

By: Andrea Wroblewski and Olivia Gette

Github Link: https://github.com/AHWroblewski4/Stroke-Prediction-Pipeline

Title: Predicting Stroke Risk Using Apache Spark-Based Machine Learning

Introduction:
Strokes remain one of the leading global causes of death and disability. To support early intervention, this project developed a predictive classification pipeline that uses demographic and clinical features to assess individual stroke risk. The pipeline was implemented using a distributed Apache Spark framework to ensure scalability and computational efficiency when processing large volumes of healthcare data. Although the dataset is moderate in size, using Spark allowed us to simulate real-world big data workflows and evaluate scalability across distributed computing environments.

Problem Statement: The main objective of this project is to build a distributed machine learning pipeline using Apache Spark to predict whether a patient is at risk of having a stroke. Stroke prediction is important for early intervention and improving patient outcomes. We used a publicly available dataset titled "Healthcare Dataset Stroke Data" from Kaggle, which contains patient demographic information, health conditions, and lifestyle-related risk factors such as hypertension, BMI, and smoking status.

To manage and process this dataset efficiently, we used Apache Spark across four virtual machines (VMs) in a distributed cluster setup. The Spark cluster consisted of one master node and three worker nodes. Running Spark in this distributed mode allowed us to compare performance across different numbers of VMs (1, 2, 3, and 4) and evaluate how distributed computing impacts training time and scalability. We expect that increasing the number of VMs would reduce overall computation time for preprocessing and model training by using Spark's ability to parallelize tasks.

Dataset:
Kaggle Stroke Prediction Dataset
Records: 5,110
Soriano, F. (2021). *Stroke Prediction Dataset* [Data set]. Kaggle.
https://www.kaggle.com/datasets/fedesoriano/stroke-prediction-dataset

Uploading the dataset:
The dataset was uploaded to the Hadoop Distributed File System (HDFS) on the masternode using:

- hdfs dfs -put healthcare-dataset-stroke-data.csv /user/sat3812/stroke_project/data/

Code Explained:
Python Packages:

Package	Purpose
pyspark.sql.SparkSession	Initialize Spark.
pyspark.sql.DataFrame	DataFrame manager.
pyspark.sql.functions as F	SQL functions.
pyspark.ml.Pipeline	Creates a pipeline to help organize.
pyspark.ml.feature.Imputer	To handle missing data.
pyspark.ml.feature.StringIndexer	To convert categories to numbers.
pyspark.ml.feature.OneHotEncoder	OneHot to convert categorical features in binary.
pyspark.ml.feature.VectorAssembler	Create a vector to combine all features into one.
pyspark.ml.feature.StandardScaler	To standardize features.
pyspark.ml.classification	Machine Learning Models: LogisticRegression, LinearSVC, GBTClassifier, MultilayerPerceptronClassifier
pyspark.ml.evaluation.BinaryClassificationEvaluator	To evaluate model performance.
pyspark.ml.tuning.CrossValidator	Selects the best hyperparameters.
pyspark.ml.tuning.ParamGridBuilder	Creates different hyperparameters for fine tuning.
time	To Measure execution time.
logging	To control Spark output to make it easier to read.

- In the python program itself the dataset path was set using:

```
csv_input_path =
"hdfs://master:9000/user/sat3812/stroke_project/data/healthcare-dataset-st
roke-data.csv"
```

- It was then read in Spark using:

```
input_dataframe = spark_session.read.csv(csv_input_path, header=True,
inferSchema=True)
)
```

- To reduce the amount of log output messages and making it easier to read:

```
logger = logging.getLogger("py4j")
logger.setLevel(logging.ERROR)
```

Data Preprocessing:
Functions were created to preprocess the data to allow for efficiency and reproducibility in the pipeline.

The preprocessing steps:
Cleaning Functions:

- Make null tokens to data that can be recognized by Spark:
    - Define nulle tokens:

```
null_token_list = [t.lower() for t in ["na", "n/a", "null", "none", "n.a.", "na.", "", "N/A"]]
```

    - Use `null_token_list` in the `convert_to_null function` to replace values to ones that can be recognized by Spark.
- Standardizing column names (`clean_column_names function`):
    - Goes through all columns, converting their names to lowercase, trimming whitespace, and replacing spaces or hyphens with underscores .
- Clean BMI column and Checks format (`normalize_numeric`):
    - Using `convert_to_null function` to clean BMI column.
    - Ensures certain columns () are stored as `("double")` to allow PySpark to run calculations.

Handle Class imbalance with oversampling (since stroke events are rare):

- The `replicate_minority_class_with_oversampling` function handles class imbalance by using an oversampling technique through replication.
- The function first calculates the total count of records for both the majority and minority classes using PySpark's `groupBy` and `count` operations.
- Then it divides the majority count by the minority count to calculate how many times the minority needs to be copied to have a similar number to the majority creating a replication factor.

- The function separates the data into minority and majority groups, then copies the minority group multiple times (based on the replication factor) and uses `union` to combine the minority copies and the majority group, creating a balanced dataset.
- The balanced dataset is randomly shuffled uses `randomSplit` to help prevent bias

Preprocessing function (`preprocessing`):

- Define Columns:
- Input Numeric columns using mean(`numeric_imputer`):
    - Handles missing values in continuous number columns, by filling it using the median of all the other values in the column.
- StingIndexers and OneHotEncoding for categorical columns(`string_indexers` and `one_hot_encoders`):
    - StingIndexers: Assigns an index to each category.
    - OneHotEncoder: ……..
- Combine Features into a Single Vector(`feature_assembler`):
    - Takes all the processed columns and combines into one column(`unscaled_features`).
- Standardizing Features(`feature_scaler`):
    - Ensure features (`unscaled_features`) are in a similar range and saves to `features`.
- Calculate the total number of input features after all the preprocessing is done(`infer_feature_vector_length`):
    - Counts total number of items in the `features` vector to use for input in the machine learning models.

Evaluate Model Performance Function (`evaluate_models`):

- This function measures how well the machine learning model made its predictions.
- Gets Confusion Matrix values by:

```
true_positive = predictions.filter((F.col(label_column_name) == 1) &
(F.col("prediction") == 1)).count()
    false_positive = predictions.filter((F.col(label_column_name) == 0) &
(F.col("prediction") == 1)).count()
    false_negative = predictions.filter((F.col(label_column_name) == 1) &
(F.col("prediction") == 0)).count()
    true_negative = predictions.filter((F.col(label_column_name) == 0) &
(F.col("prediction") == 0)).count()
```

- Uses the information from the confusion matrix to get accuracy, AUC, precision, and recall:

```
precision = true_positive / (true_positive + false_positive) if
(true_positive + false_positive) else 0.0
    recall = true_positive / (true_positive + false_negative) if
(true_positive + false_negative) else 0.0
    specificity = true_negative / (true_negative + false_positive) if
(true_negative + false_positive) else 0.0
```

- Outputs as a dictionary

Combining everything and model training (`models`):

- This calls all the previous functions (cleaning, oversampling, preprocessing) to prepare the data for modeling, train the model, and evaluate it.
- First it connects to spark, makes the output easier to read, and confirm the program is working by:

```
 spark_session =
SparkSession.builder.appName("sat5165_stroke_pipeline").getOrCreate()
    spark_session.sparkContext.setLogLevel("ERROR")
    print(f"Loading: {csv_input_path}")
```

- The dataset is then read directly from HDFS, ensuring distributed access across all virtual machines.
- After loading, the data undergoes the previously defined cleaning, normalization, and oversampling steps to handle missing values, standardize columns, and address the imbalance between stroke and non-stroke cases.
- Next, the data is split into training (80%) and testing (20%) subsets using:

```
    training_raw_dataframe, testing_dataframe =
input_dataframe.randomSplit([0.8, 0.2], seed=42)
```

- The minority class is then oversampled.
- The preprocessing pipeline (created with Imputer, StringIndexer, OneHotEncoder, VectorAssembler, and StandardScaler) is then collected and fitted to the training data.
    - This ensures the preprocessed data is applied consistently during training and testing.
- Input feature vector size is then calculated to configure the neural network layer dimension.

Model Training:
Both of us used the same preprocessing code, but implemented different models. This allowed us to compare model performance and investigate which approach best predicted stroke risk.

Machine Learning Methods Used:

- Four models were implemented and evaluated:
  - Logistic Regression
  - Linear SVM
  - Gradient-Boosted Tree
  - Multilayer Perceptron
- Each model is inside the Spark ML Pipeline and tuned with cross-validation to test different hyperparameter settings and automatically select the best configuration:

```
cross_validator = CrossValidator(
    estimator=pipeline_with_estimator,
    estimatorParamMaps=param_grid,
    evaluator=auc_evaluator,
    numFolds=3,
    parallelism=2,
    seed=42
)
```

- 3-fold cross-validation divides the training data into three parts:
  - Training on two and validating on one
- Each model is then trained using:

```
best_model = cross_validator.fit(training_dataframe)
```

- Then evaluated:

```
predictions = best_model.transform(testing_dataframe)
metrics = evaluate_models(predictions, label_column)
```

- Results were then summarized in a comparison table:

Model	Acc	AUC	Prec	Recall	Spec	Time(s)
Logistic Regression	0.727	0.861	0.154	0.810	0.722	242.91
Linear SVM	0.719	0.861	0.155	0.845	0.712	1065.26
Gradient Boosted Trees	0.819	0.802	0.162	0.500	0.839	1545.94
Multilayer Perceptron	0.781	0.697	0.097	0.328	0.809	258.94

Machine Learning Models:

Logistic Regression:

The Logistic Regression model is used as the baseline model for this stroke prediction project.

Andrea:

Linear Support Vector Classifier

Linear Support Vector Classifier (LinearSVC) is a powerful and efficient model that finds the best straight line to separate two different classes of data. SVC is a specific type of Support Vector Machine (SVM). The LinearSVC works by looking at the data points closest to the line and maximizing the distance between these vectors and the line. This approach makes the model robust and helps it generalize well. In this stroke prediction project, the LinearSVC model demonstrated high sensitivity.

Gradient-Boosted Trees Classifier

Gradient-Boosted Trees Classifier (GBTClassifier) is a highly effective ensemble learning method that builds a sequence of weak prediction models, typically decision trees. Unlike methods that build trees independently (like Random Forest), GBTs train each new tree to correct the errors made by the previous trees in the sequence. This process of iteratively improving prediction accuracy by focusing on the remaining hard-to-classify samples results in a highly accurate and powerful final model. This has both the highest accuracy and precision.

Olivia:

The Multilayer Perceptron

Multilayer Perceptron (MLP) Classifier is a type of neural network that consists of multiple layers of interconnected nodes (neurons). It learns complex, non-linear relationships in the data by adjusting connection weights during training through backpropagation. Each layer transforms the input data in a way that allows the model to capture deeper patterns and interactions. In this stroke prediction project, the MLP classifier performed well in identifying subtle relationships among features, contributing to improved overall prediction accuracy.

Performance Comparison Across Models:
The following table summarizes the performance metrics for all models on the testing dataset:

Model	Accuracy	AUC	Precision	Recall	Specificity
Logistic Regression	0.727	0.861	0.154	0.810	0.722
Linear SVC	0.719	0.861	0.155	0.845	0.712
Gradient Boosting Trees	0.819	0.802	0.162	0.500	0.839
Multilayer Perceptron	0.779	0.698	0.096	0.328	0.8097

All four models achieved decent overall accuracy (0.719-0.819). However, precision was low across all models (0.097-0.162). This could be due to the severe class imbalance within the dataset and decision boundaries that favored recall over precision. As a result, the models were able to identify most true stroke cases (high recall) but generated many false positives (low precision).

Logistic Regression and LinearSVC models had the highest recall values (0.810 and 0.845), indicating stronger sensitivity in detecting true stroke cases. GBT demonstrates the best trade-off, achieving high overall accuracy and specificity while maintaining the highest Precision score.

The MLP was the poorest performer across all critical metrics, showing the lowest Recall (0.328), lowest Precision (0.097), and lowest AUC (0.697). This suggests that the default neural network architecture used or the limited tuning applied, did not capture the underlying data patterns effectively.

Beyond model accuracy, we also evaluated how distributed computing resources impacted training efficiency.

VM Performance Comparison:

In this project, we compared the performance of running the pipeline on 1, 2, 3, and 4 virtual machines using Apache Spark in distributed mode. As shown, distributing computation across more VMs led to a noticeable reduction in total runtime, which shows Spark's scalability for parallelized machine learning workflows.

Number of VMs	Cores Used (8 per VM)	Total Runtime (min)	Total Model Training and Evaluation Time (seconds)
1	8	54	3,113.05
2	16	37	2,140.14
3	24	36	1,999.54
4	32	27	1,536.06

1 VM:

Worker Id	Address	State	Cores	Memory	Resources
worker-20251109020828-192.168.13.200-34165	192.168.13.200:34165	ALIVE	8 (0 Used)	4.0 GiB (0.0 B Used)	

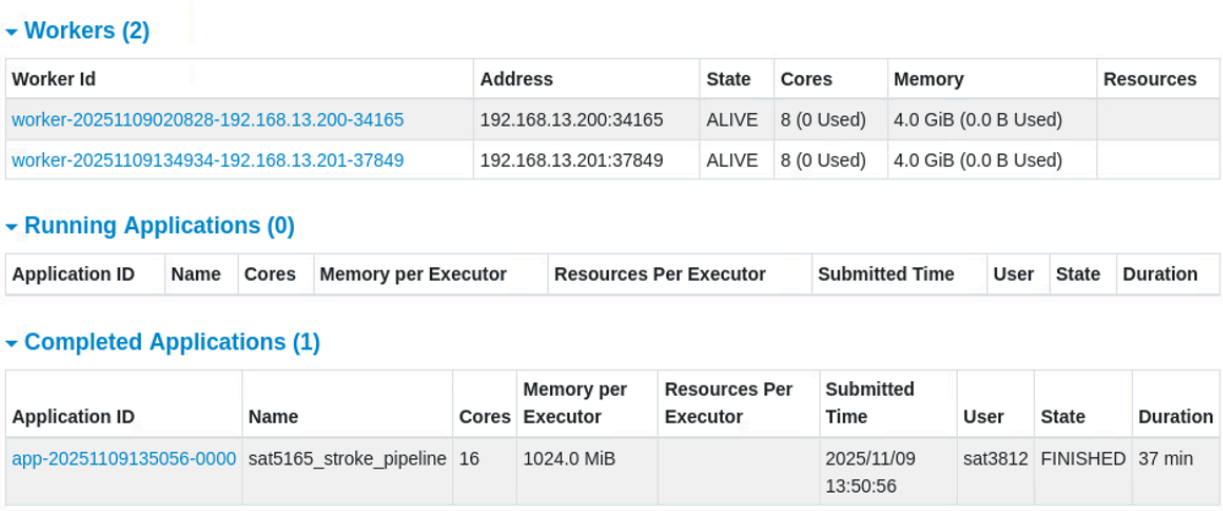
Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20251109122752-0000	sat5165_stroke_pipeline	8	1024.0 MiB		2025/11/09 12:27:52	sat3812	FINISHED	54 min

2 VMS:



Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20251109020828-192.168.13.200-34165	192.168.13.200:34165	ALIVE	8 (0 Used)	4.0 GiB (0.0 B Used)	
worker-20251109134934-192.168.13.201-37849	192.168.13.201:37849	ALIVE	8 (0 Used)	4.0 GiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20251109135056-0000	sat5165_stroke_pipeline	16	1024.0 MiB		2025/11/09 13:50:56	sat3812	FINISHED	37 min

3 VMS:

▾ Workers (3)

Worker Id	Address	State	Cores	Memory	Resources
worker-20251109161018-192.168.13.201-32833	192.168.13.201:32833	ALIVE	8 (0 Used)	4.0 GiB (0.0 B Used)	
worker-20251109163402-192.168.13.200-37339	192.168.13.200:37339	ALIVE	8 (0 Used)	4.0 GiB (0.0 B Used)	
worker-20251109164312-192.168.13.122-37287	192.168.13.122:37287	ALIVE	8 (0 Used)	4.0 GiB (0.0 B Used)	

▾ Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

▾ Completed Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20251109164405-0000	sat5165_stroke_pipeline	24	2.0 GiB		2025/11/09 16:44:05	sat3812	FINISHED	36 min

4 VMS:

▾ Workers (4)

Worker Id	Address	State	Cores	Memory	Resources
worker-20251109192507-192.168.13.201-33947	192.168.13.201:33947	ALIVE	8 (0 Used)	4.0 GiB (0.0 B Used)	
worker-20251109192515-192.168.13.122-43307	192.168.13.122:43307	ALIVE	8 (0 Used)	4.0 GiB (0.0 B Used)	
worker-20251109192515-192.168.13.123-42317	192.168.13.123:42317	ALIVE	8 (0 Used)	4.0 GiB (0.0 B Used)	
worker-20251109192516-192.168.13.200-46823	192.168.13.200:46823	ALIVE	8 (0 Used)	4.0 GiB (0.0 B Used)	

▾ Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

▾ Completed Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20251109193815-0000	sat5165_stroke_pipeline	32	3.0 GiB		2025/11/09 19:38:15	sat3812	FINISHED	27 min

Discussion:

This project demonstrates how these machine learning models have to align with clinical goals. While the GBT model achieved the best accuracy, precision, and specificity, its lower recall means that some stroke cases might go undetected. In a clinical context, missing a potential stroke case is far more harmful than incorrectly flagging a non-stroke patient. In clinical practice, a model with higher recall such as Logistic Regression or SVM could be preferred despite lower precision. This model reinforces how predictive analytics should be implemented as a support tool, such as alerting clinicians or prompting follow ups, rather than replacing clinical judgment.

Running the pipeline in distributed mode across multiple VMs significantly reduced the total training and evaluation time. As the number of VMs increased from one to four, total runtime

decreased by about 50%, demonstrating the scalability and efficiency of Spark for handling large health datasets.

Overall, this project successfully built a distributed machine learning pipeline for stroke prediction using Spark. It showcased how integrating scalable data processing with machine learning can help support early intervention and inform healthcare decision making. Future work could focus on more advanced hyperparameter tuning, more deep learning models, and using some more advanced patient data to improve the prediction reliability.