

VIETNAM NATIONAL UNIVERSITY - HCM CITY
INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING



BIG DATA TECHNOLOGY
PROJECT REPORT

Topic: Disease Prediction Model Using SparkML

Instructors:

Dr. Ho Long Van

Members:

Nguyen Huy Bao - ITDSIU21076

Nguyen Ngoc Gia Linh - ITDSIU22173

1 Introduction

1.1 Motivation for the Study

Healthcare is an essential domain in which technological advancements can have a profound impact on human lives. Machine-powered disease prediction models have the potential to revolutionize the healthcare industry by enabling early diagnosis, improving patient outcomes, and optimizing treatment strategies. With the growing availability of large healthcare data sets, predictive modeling has become increasingly feasible and accurate.

1.2 Objectives and goals

This project focuses on building a scalable and efficient disease prediction model using Apache Spark's machine learning library (SparkML). The goal is to leverage distributed computing and big data frameworks to handle large-scale healthcare data and provide timely, accurate predictions of diseases based on patient symptoms. By integrating Hadoop Distributed File System (HDFS) for data storage and SparkML for model training, this project combines powerful tools to address the challenges of big data in healthcare. The broader objective is to demonstrate how distributed frameworks can overcome the computational and storage limitations of traditional machine learning models, making predictive analytics more accessible and scalable for healthcare applications. Additionally, the project's deployment on a user-friendly platform like Streamlit ensures practical usability and real-world applicability.

2 Techniques and Methodology

2.1 HDFS (Hadoop Distributed File System)

2.1.1 Overview of HDFS

HDFS, or the Hadoop Distributed File System, is a scalable and fault-tolerant file system designed to store and manage large datasets across multiple nodes in a distributed computing environment. It provides high throughput for data-intensive applications and ensures data reliability through replication.

2.1.2 Reasons for choosing HDFS

HDFS was chosen as the storage backend for this project because:

- **Scalability:** It efficiently handles datasets of varying sizes, from gigabytes to petabytes.
- **Fault Tolerance:** Data is replicated across multiple nodes, ensuring redundancy and availability in case of node failures.
- **Integration with Spark:** HDFS seamlessly integrates with Apache Spark, enabling distributed data processing for model training and predictions.
- **Cost-effectiveness:** HDFS can run on commodity hardware, reducing infrastructure costs.

2.1.3 Integration of HDFS to our Model

HDFS acts as the primary storage for the healthcare dataset, which contains information such as patient demographics, symptoms, and diagnoses. The data stored in HDFS is accessed by Spark for processing and training the machine learning model. HDFS ensures that the data is readily available across all Spark nodes, facilitating efficient distributed processing. For instance, when a user inputs symptoms for prediction, the model retrieves relevant data from HDFS to generate real-time insights.

2.2 Apache Spark

2.2.1 Overview of Apache Spark

Apache Spark is an open-source, distributed computing framework known for its speed and versatility. It supports a variety of workloads, including batch processing, real-time streaming, and machine learning. The SparkML library, a core component of Spark, provides tools for building, training, and deploying machine learning models at scale.

2.2.2 Reasons for choosing Spark

Spark was selected for model training due to its:

- High Speed: In-memory computation accelerates data processing compared to disk-based alternatives.
- Distributed Computing: Spark's ability to parallelize tasks across clusters makes it ideal for processing large datasets efficiently.
- Scalability: Spark scales from a single server to thousands of nodes, accommodating future data growth.
- Integrated Machine Learning (SparkML): SparkML provides pre-built algorithms and pipelines for streamlined model development.

2.2.3 Integration of Spark to our Workflow

- Data Processing: Spark reads and preprocesses raw healthcare data stored in HDFS, handling tasks like cleaning, encoding, and feature extraction.
- Data Processing: Spark reads and preprocesses raw healthcare data stored in HDFS, handling tasks like cleaning, encoding, and feature extraction.
- Prediction: After training, the model is used to predict diseases based on user inputs, with Spark handling real-time computation for seamless predictions.

3 Data preparation

3.1 Data Storage (HDFS)

3.1.1 Description of the dataset

Link to the dataset: [Disease Prediction Using Machine Learning](#) The dataset contains information on symptoms of patients, including their frequency. It is structured in tabular format with 132 relevant columns and one unnamed column labeled as "133". This dataset is stored in a structured format, suitable for big data processing.

summary	itching	skin_rash	nodal_skin_eruptions	continuous_sneezing	shivering
count	4920	4920	4920	4920	4920
mean	0.1378048780487805	0.1597560975609756	0.02195121951219512	0.045121951219512194	0.02195121951219512
stddev	0.34473010874032145	0.36641694249935314	0.14653916900608888	0.20759267768168482	0.1465391690060889
min	0	0	0	0	0
max	1	1	1	1	1

Figure 1: Description of dataset

3.1.2 Storing and managing the data using HDFS

The dataset was uploaded and stored in a Hadoop Distributed File System (HDFS) to ensure scalability and fault tolerance. HDFS provides efficient data storage and retrieval capabilities, which are critical for handling large datasets during model development.

```
1 df = spark.read.csv('hdfs://localhost:9000/usr/baonguyen/data.csv', header=True, inferSchema=True)
```

Figure 2: Store dataset on HDFS

3.2 Exploratory Data Analysis (EDA) and Preprocessing

3.2.1 EDA

Top 5 symptoms visualization with charts The top 5 symptoms were identified and displayed using a bar chart, representing the frequency distribution of these symptoms. This provides insights into the most common conditions.

```
1 for prognosis in result_df["prognosis"].unique():
2     prognosis_data = result_df[result_df["prognosis"] == prognosis]
3     sorted_data = prognosis_data.sort_values(by="frequency", ascending=False).head(5)
4
5     # Plot the horizontal bar chart
6     plt.figure(figsize=(10, 6))
7     plt.barh(sorted_data["Symptom"], sorted_data["frequency"], color="skyblue")
8     plt.gca().invert_yaxis() # Highest frequency at the top
9     plt.title(f"Top 5 Symptoms for Prognosis: {prognosis}")
10    plt.xlabel("Frequency")
11    plt.ylabel("Symptoms")
12    plt.tight_layout()
```

Figure 3: Code for Top 5 symptoms

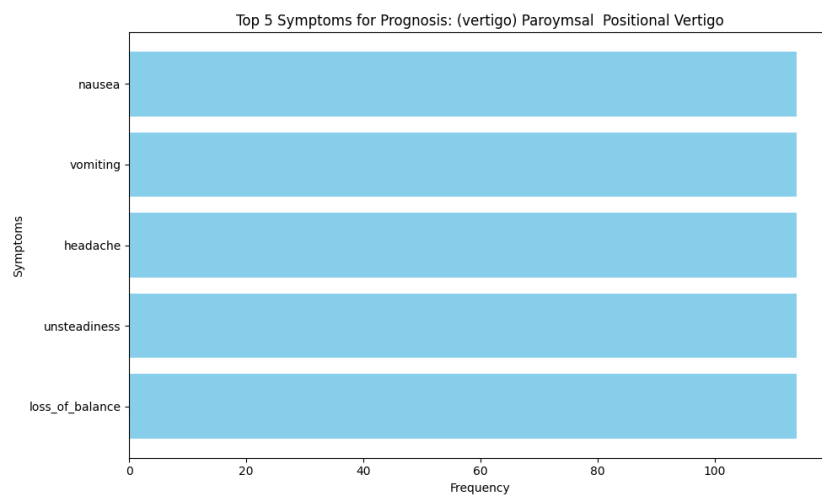


Figure 4: Plot of Top 5 symptoms for Payroymal

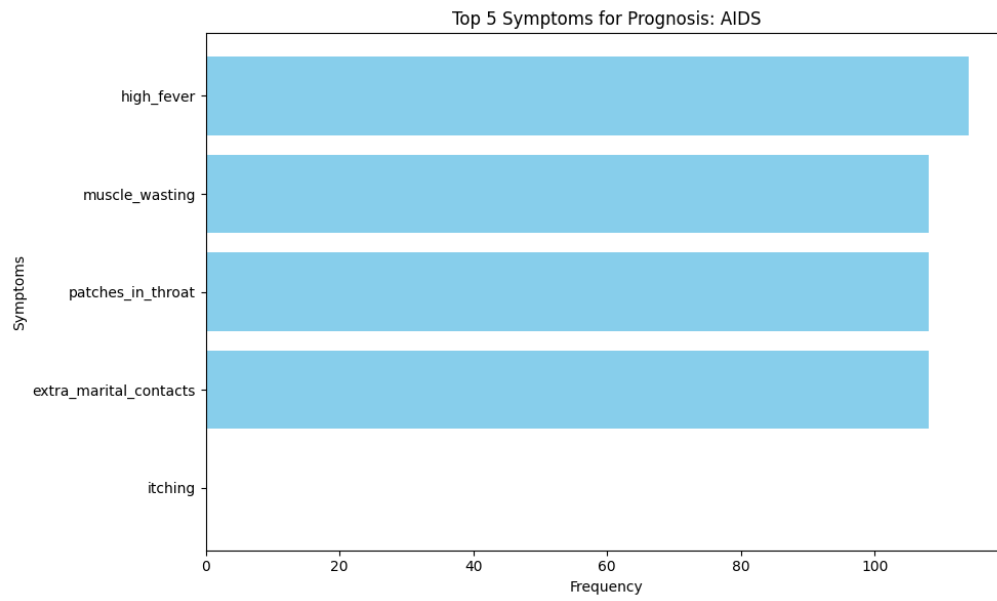


Figure 5: Plot of Top 5 symptoms for AIDS

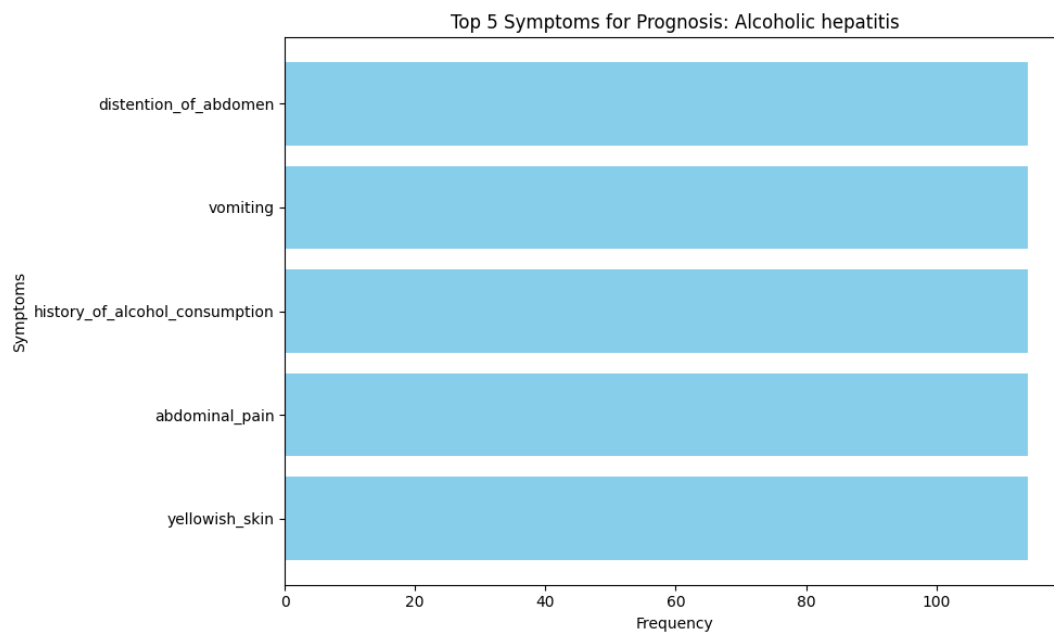


Figure 6: Plot of Top 5 symptoms for Alcoholic hepatitis

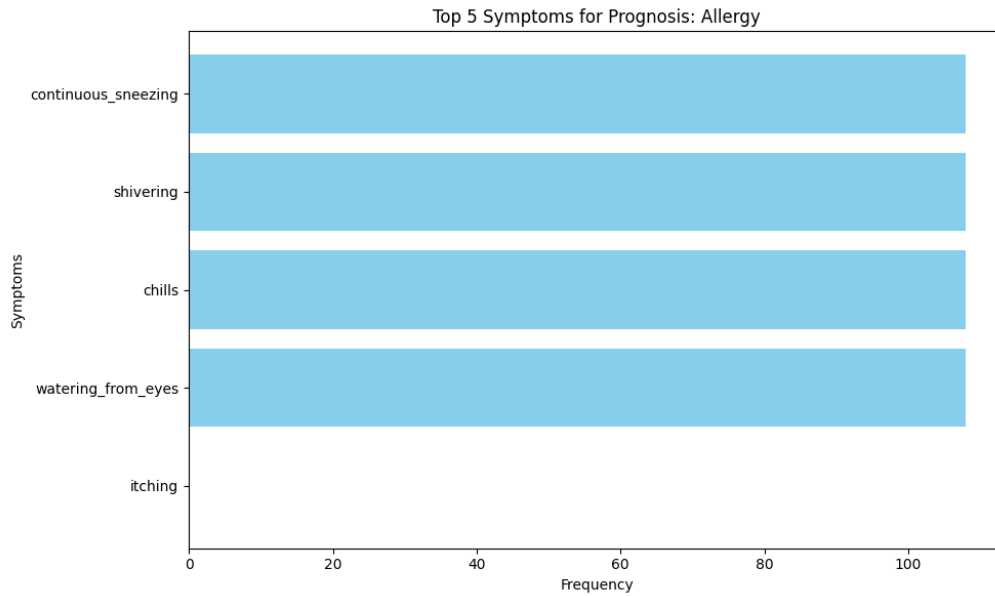


Figure 7: Plot of Top 5 symptoms for Allergy

Frequency of symptoms The frequency of all symptoms helps in understanding the distribution and significance of different symptoms.

```

1 frequency = {}
2
3 # Assuming 'features' is a list of feature column names
4 for cols in features:
5     frequency[cols] = df.select(sum(col(cols)).alias("frequency")).collect()[0]["frequency"]
6
7 # Sort the frequency dictionary in descending order
8 sorted_frequency = dict(sorted(frequency.items(), key=lambda item: item[1], reverse=True))
9
10 # Extract column names and their corresponding frequencies
11 columns = list(sorted_frequency.keys())
12 frequencies = list(sorted_frequency.values())
13
14 # Plot the frequencies as a horizontal bar chart
15 plt.figure(figsize=(10, 20))
16 plt.barh(columns, frequencies)
17 plt.gca().invert_yaxis() # Invert the y-axis to show the highest frequency at the top
18 plt.title("Frequency of Symptoms")
19 plt.xlabel("Frequency")
20 plt.ylabel("Symptoms")

```

Figure 8: Code for Frequency of symptoms

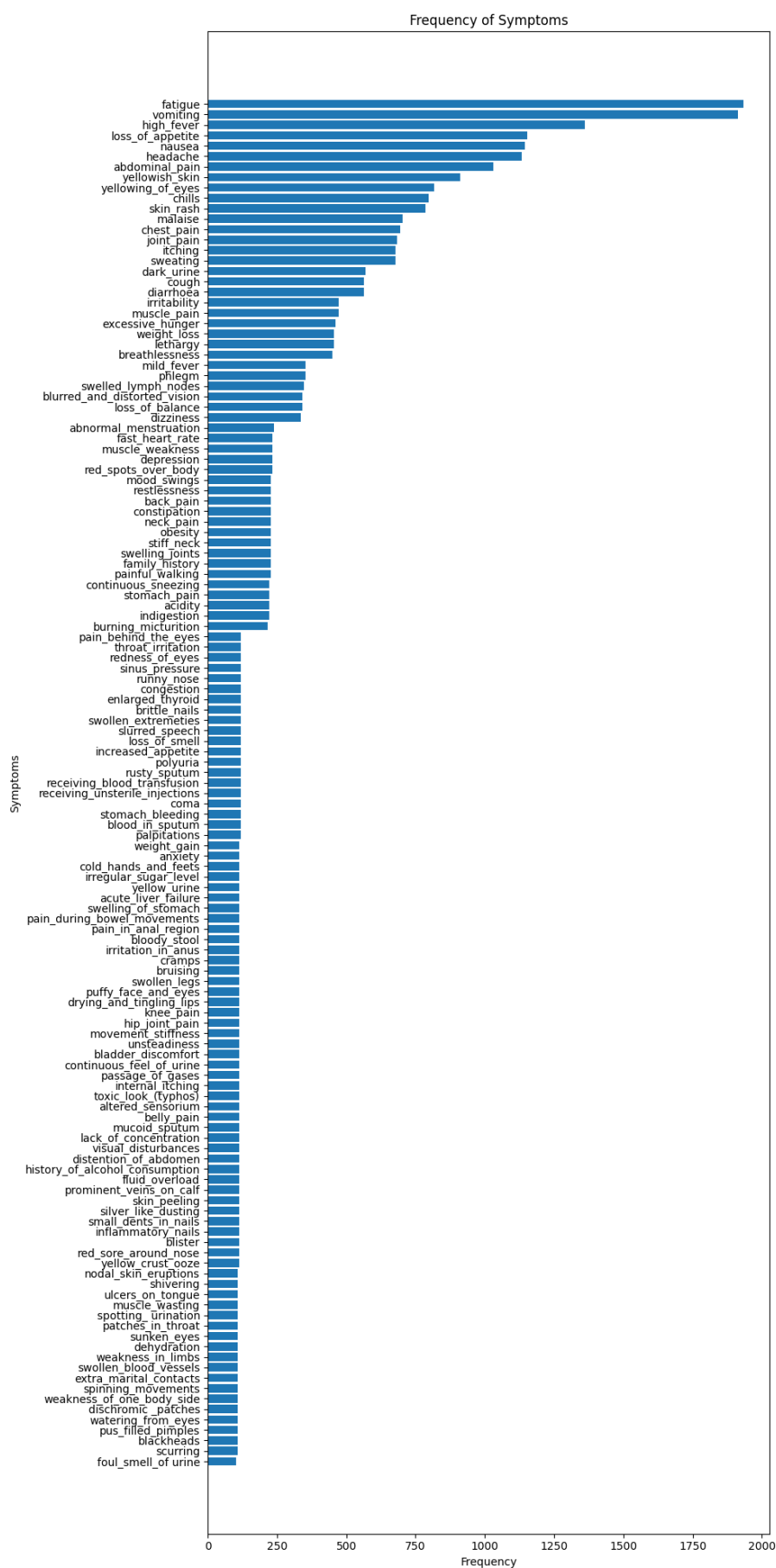


Figure 9: Frequency of symptoms

3.3 Preprocessing

Handling missing values Missing values were handled to ensure the dataset was complete and consistent for training.

```
1 # Count null values for each column
2 null_counts = df.select(
3     [sum(col(c).isNull().cast("int")).alias(c) for c in df.columns]
4 )
5
6 # Collect the null counts as a dictionary
7 null_columns = null_counts.first().asDict()
8
9 # Filter columns with non-zero null counts
10 null_columns_with_counts = {col_name: count for col_name, count in null_columns.items() if count > 0}
```

Figure 10: Code for handling null values

```
No columns with null values found.
```

Figure 11: Handling null values result

Missing value was detected to be extracted from the data in order to not cause bias.
There was no missing value found.

Handling data inconsistencies An unnamed column labeled “133” was identified as irrelevant and dropped from the dataset to avoid introducing noise during model training.

```
1 # Drop unwanted columns
2 df = df.drop(*columns_to_drop)
3
4 # Show the resulting DataFrame
5 print(f"Dropped columns: {columns_to_drop}")
6 df.show(5)
```

Figure 12: Drop unwanted column

4 Model Training (SparkML)

4.1 Description of models used

4.1.1 Decision Tree Classifier

A decision tree is a flowchart-like structure used to make decisions or predictions. It consists of nodes representing decisions or tests on attributes, branches representing the outcome of these decisions, and leaf nodes representing final outcomes or predictions. Each internal node corresponds to a test on an attribute, each branch corresponds to the result of the test, and each leaf node corresponds to a class label or a continuous value.

4.1.2 Structure of a Decision Tree

- **Root Node:** Represents the entire dataset and the initial decision to be made.
- **Internal Nodes:** Represent decisions or tests on attributes. Each internal node has one or more branches.
- **Branches:** Represent the outcome of a decision or test, leading to another node.
- **Leaf Nodes:** Represent the final decision or prediction. No further splits occur at these nodes.

4.1.3 How Decision Trees Work

The process of creating a decision tree involves:

1. **Selecting the Best Attribute:** Using a metric like Gini impurity, entropy, or information gain, the best attribute to split the data is selected.
2. **Splitting the Dataset:** The dataset is split into subsets based on the selected attribute.
3. **Repeating the Process:** The process is repeated recursively for each subset, creating a new internal node or leaf node until a stopping criterion is met (e.g., all instances in a node belong to the same class or a predefined depth is reached).

4.1.4 Metrics for Splitting

- **Gini Impurity:** Measures the likelihood of an incorrect classification of a new instance if it was randomly classified according to the distribution of classes in the dataset.

$$\text{Gini} = 1 - \sum_{i=1}^n p_i^2,$$

where p_i is the probability of an instance being classified into a particular class.

- **Entropy:** Measures the amount of uncertainty or impurity in the dataset.

$$\text{Entropy} = - \sum_{i=1}^n p_i \log_2(p_i),$$

where p_i is the probability of an instance being classified into a particular class.

- **Information Gain:** Measures the reduction in entropy or Gini impurity after a dataset is split on an attribute.

$$\text{Information Gain} = \text{Entropy}_{\text{parent}} - \sum_{i=1}^n \left(\frac{|D_i|}{|D|} \times \text{Entropy}(D_i) \right),$$

where D_i is the subset of D after splitting by an attribute.

4.2 Random Forest Classifier

4.2.1 Ensemble Methods

Ensemble learning methods are made up of a set of classifiers—e.g., decision trees—and their predictions are aggregated to identify the most popular result. The most well-known ensemble methods are bagging, also known as bootstrap aggregation, and boosting. In 1996, Leo Breiman introduced the bagging method; in this method, a random sample of data in a training set is selected with replacement—meaning that the individual data points can be chosen more than once. After several data samples are generated, these models are then trained independently, and depending on the type of task—i.e., regression or classification—the average or majority of those predictions yield a more accurate estimate. This approach is commonly used to reduce variance within a noisy dataset.

4.2.2 Random Forest Algorithm

The random forest algorithm is an extension of the bagging method as it utilizes both bagging and feature randomness to create an uncorrelated forest of decision trees. Feature randomness, also known as feature bagging or the “random subspace method,” generates a random subset of features, which ensures low correlation among decision trees. This is a key difference between decision trees and random forests. While decision trees consider all the possible feature splits, random forests only select a subset of those features.

4.2.3 How Random Forest Works

The Random Forest algorithm operates as an ensemble of decision trees and is commonly used for both regression and classification tasks. Before training, three key hyperparameters need to be set:

1. **Node size:** The minimum number of samples required to split a node.
2. **Number of trees:** The total number of trees in the forest.
3. **Number of features sampled:** The subset of features considered for splitting at each node.

Each tree in the forest is built from a bootstrap sample, which is a subset of the training data selected with replacement. To introduce additional randomness and reduce correlation among trees, feature bagging is applied by randomly selecting a subset of features for each split.

- **Out-of-Bag (OOB) Samples:** Approximately one-third of the training data, not included in the bootstrap sample, is set aside for validation. This is called the out-of-bag sample and is used later for cross-validation and performance evaluation.
- **For Predictions:**
 - **Regression Tasks:** The final prediction is determined by averaging the outputs of all individual trees.
 - **Classification Tasks:** The predicted class is determined by a majority vote, where the most frequent class across all trees is selected.

Finally, the OOB sample is used to validate and fine-tune the model, ensuring the predictions are accurate without requiring additional holdout datasets.

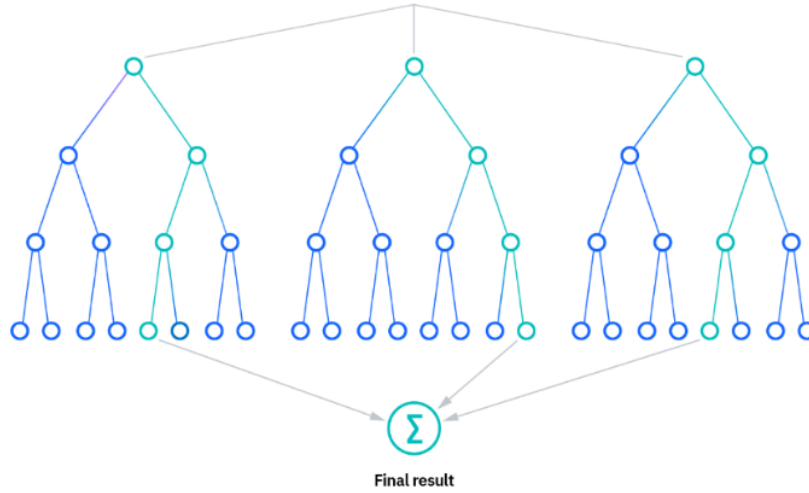


Figure 13: Random Forest Classifier Diagram

4.3 Naïve Bayes Classifier

4.3.1 What are Naïve Bayes classifiers?

The Naïve Bayes classifier is a supervised machine learning algorithm used for classification tasks, such as text classification. It employs principles of probability to perform classification tasks.

Naïve Bayes belongs to the family of generative learning algorithms, which aim to model the distribution of inputs for a given class or category. Unlike discriminative classifiers, such as logistic regression, it does not specifically learn which features are most important to differentiate between classes.

4.3.2 A Brief Review of Bayesian Statistics

Naïve Bayes is also known as a probabilistic classifier since it is based on Bayes' Theorem. This theorem, also called Bayes' Rule, allows us to "invert" conditional probabilities.

4.3.3 Bayes' Theorem:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \quad (1)$$

Where:

- $P(A|B)$: Posterior probability of event A given that event B has occurred.
- $P(B|A)$: Likelihood of event B given that event A is true.
- $P(A)$: Prior probability of event A .
- $P(B)$: Marginal probability of event B .

4.3.4 How Naïve Bayes Works:

- **Posterior:** This is the probability we aim to calculate, $P(A|B)$, based on the observed data.
- **Likelihood:** $P(B|A)$, which measures how likely the observed data is given the hypothesis.
- **Prediction:** The algorithm calculates the posterior probability for all classes and selects the class with the highest probability.

4.3.5 Types of Naïve Bayes Classifier:

Naïve Bayes classifiers are divided into the following types, depending on the distribution of data:

- **Gaussian Naïve Bayes:** Used for data that follows a normal distribution. It assumes the likelihood of features is Gaussian.
- **Multinomial Naïve Bayes:** Suitable for discrete data, such as text classification tasks where word frequencies are used as features.
- **Bernoulli Naïve Bayes:** Used for binary/boolean data, often applied in text classification problems, where the presence or absence of a word is considered.

4.3.6 Relevance to the Project:

For the disease prediction task in this project, Naïve Bayes classifiers are suitable for handling categorical features such as symptom presence or absence. The algorithm's ability to deal with high-dimensional data efficiently makes it a strong candidate for classifying diseases based on patient symptoms. The classifier can help predict the likelihood of a disease by learning patterns in the training dataset.

4.4 Training Process

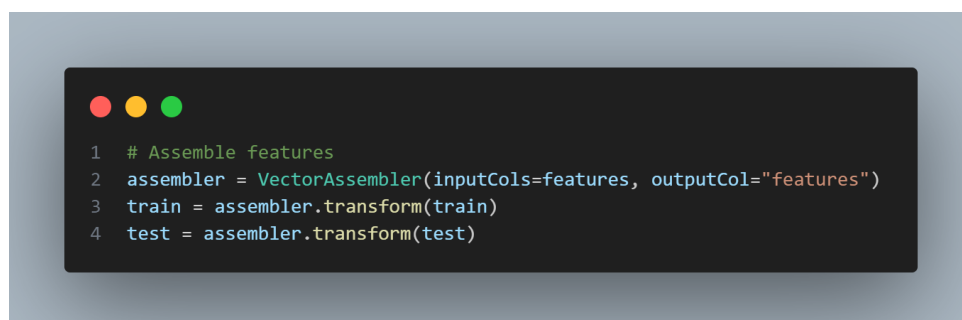
4.4.1 Data Splitting (Training and Testing)

The dataset is divided into two parts:

- **Training and Validation Set:** 80% of the data is used for training and validating the machine learning model.
- **Testing Set:** 20% of the data is reserved for evaluating the model's performance after training.

4.4.2 Feature Assembly using VectorAssembler

The `VectorAssembler` is a feature transformer in Spark ML that merges multiple feature columns into a single vector column. This is necessary because Spark ML models require input features in the form of a single vector. The `VectorAssembler` combines the specified columns into a vector, which is then used as input to the machine learning pipeline.



```
1 # Assemble features
2 assembler = VectorAssembler(inputCols=features, outputCol="features")
3 train = assembler.transform(train)
4 test = assembler.transform(test)
```

Figure 14: Code for using VectorAssembler

4.4.3 Parameter Tuning using ParamGridBuilder

Parameter tuning involves selecting the optimal combination of hyperparameters for the machine learning model. In Spark ML, the `ParamGridBuilder` is used to construct a grid of parameters for cross-validation. This process evaluates different combinations of hyperparameters to identify the best-performing configuration.

```
1 # Parameter Grid for Decision Tree
2 dt_param_grid = ParamGridBuilder() \
3     .addGrid(dt.maxDepth, [2, 5, 10]) \
4     .addGrid(dt.maxBins, [10, 20, 40]) \
5     .build()
6
7 # Parameter Grid for Random Forest
8 rf_param_grid = ParamGridBuilder() \
9     .addGrid(rf.numTrees, [50, 100]) \
10    .addGrid(rf.maxDepth, [5, 10, 20]) \
11    .build()
12
13 # Parameter Grid for Naive Bayes
14 nb_param_grid = ParamGridBuilder() \
15    .addGrid(nb.smoothing, [0.5, 1.0, 2.0]) \
16    .build()
17
```

Figure 15: Code for using ParamGridBuilder

The grid search is typically combined with cross-validation to ensure robust evaluation of the model. By systematically exploring the parameter space, `ParamGridBuilder` helps optimize the model's performance on unseen data.

4.4.4 Model training

```
1 for model_name, model, param_grid in models_with_params:
2     print(f"Training and evaluating {model_name}...")
3
4     # Define pipeline
5     pipeline = Pipeline(stages=[model])
6
7     # Define CrossValidator
8     cv = CrossValidator(
9         estimator=pipeline,
10        estimatorParamMaps=param_grid,
11        evaluator=evaluator,
12        numFolds=5 # Adjust number of folds as needed
13    )
14
15    # Fit model using CrossValidator
16    cv_model = cv.fit(train)
```

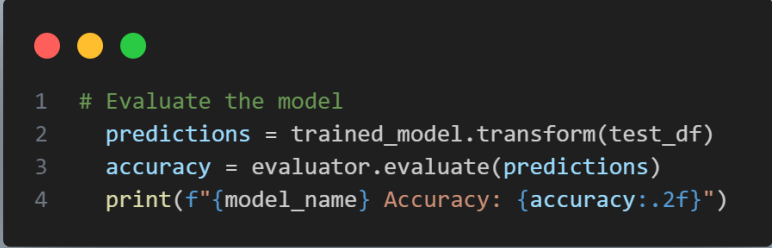
Figure 16: Code for Model training

4.5 Metrics Used for Evaluation

4.5.1 Accuracy

Accuracy measures the proportion of correctly classified instances out of the total instances in the dataset. It is calculated as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in Python and shows how to evaluate a model's accuracy using a trained model and an evaluator.

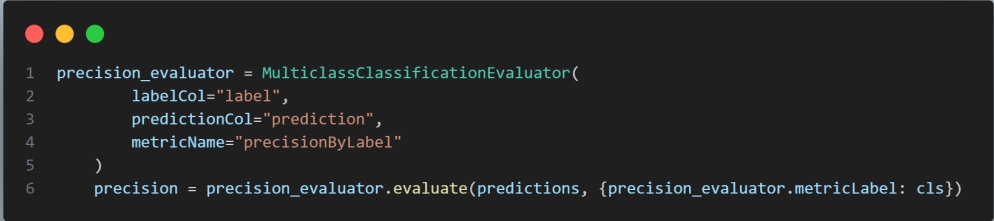
```
1 # Evaluate the model
2 predictions = trained_model.transform(test_df)
3 accuracy = evaluator.evaluate(predictions)
4 print(f"{model_name} Accuracy: {accuracy:.2f}")
```

Figure 17: Code for Accuracy calculation

4.5.2 Precision

Precision measures the proportion of true positive predictions among all positive predictions. It is calculated as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in Python and shows how to calculate precision using a MulticlassClassificationEvaluator.

```
1 precision_evaluator = MulticlassClassificationEvaluator(
2     labelCol="label",
3     predictionCol="prediction",
4     metricName="precisionByLabel"
5 )
6 precision = precision_evaluator.evaluate(predictions, {precision_evaluator.metricLabel: cls})
```

Figure 18: Code for Precision calculation

4.5.3 Recall

Recall (also known as Sensitivity or True Positive Rate) measures the proportion of actual positives that are correctly identified. It is calculated as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

```

1 recall_evaluator = MulticlassClassificationEvaluator(
2     labelCol="label",
3     predictionCol="prediction",
4     metricName="recallByLabel"
5 )
6 recall = recall_evaluator.evaluate(predictions, {recall_evaluator.metricLabel: cls})

```

Figure 19: Code for Recall calculation

4.6 Evaluation Process

4.6.1 Using MulticlassClassificationEvaluator

For model selection, we use `MulticlassClassificationEvaluator` from Spark ML, with **accuracy** as the evaluation metric.

```

1 # Define evaluator
2 evaluator = MulticlassClassificationEvaluator(
3     labelCol="label", predictionCol="prediction", metricName="accuracy"
4 )

```

Figure 20: Code for MulticlassClassificationEvaluation

The process is as follows:

- **Cross-Validation:** We perform cross-validation with $k = 5$ folds to tune the model parameters. This ensures that the model's performance is consistent across different subsets of the data and helps avoid overfitting.

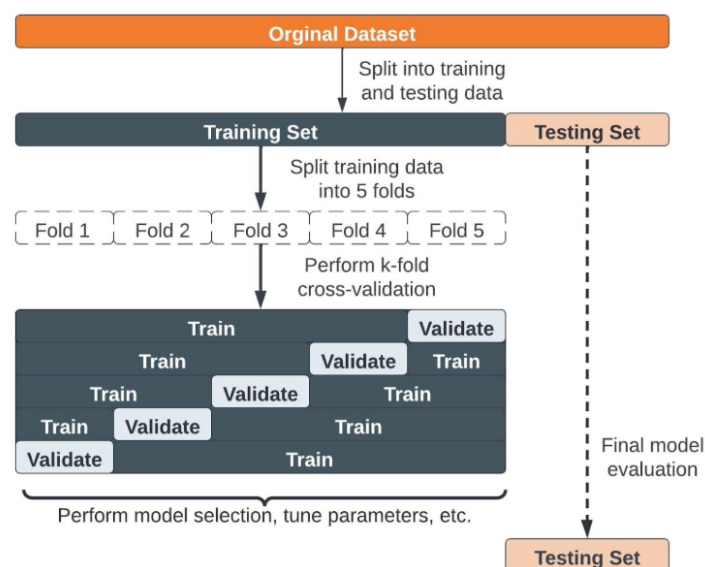


Figure 21: CrossValidation process

```

1  # Define CrossValidator
2      cv = CrossValidator(
3          estimator=pipeline,
4          estimatorParamMaps=param_grid,
5          evaluator=evaluator,
6          numFolds=5 # Adjust number of folds as needed
7      )

```

Figure 22: Code for CrossValidation

- **Final Evaluation:** Once the best model is selected based on cross-validation, it is evaluated on the testing set to measure its performance on unseen data.

Trains the model for each hyperparameter combination and selects the best-performing model based on the evaluator's metrics.

```

1  avgMetrics = cv_model.avgMetrics
2      # Print the average metrics
3      print("Average evaluator metrics for each parameter combination:")
4      for i, metric in enumerate(avgMetrics):
5          print(f"Parameter combination {i + 1}: {metric}")

```

Figure 23: Code for Average evaluation metrics (Accuracy)

```

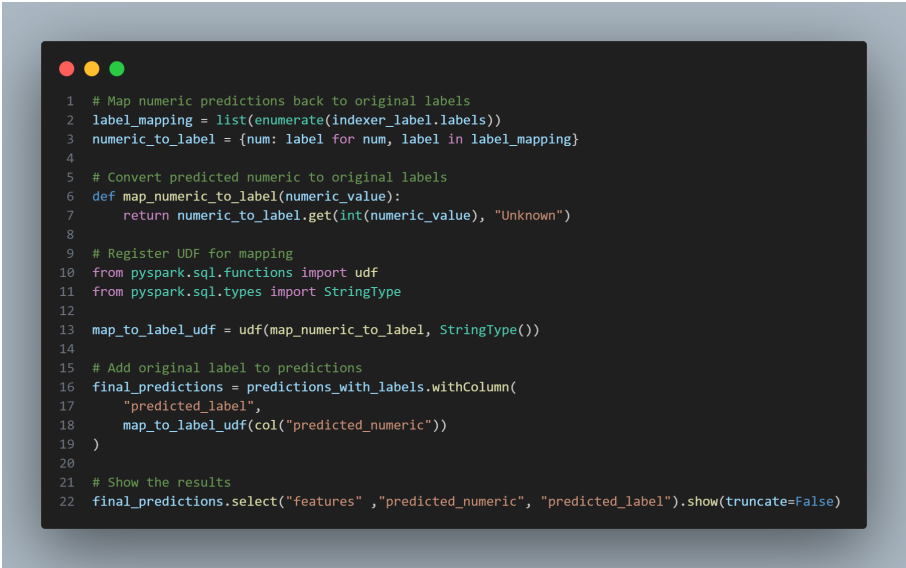
1  # Get best model
2      best_model = cv_model.bestModel
3      avgMetrics = cv_model.avgMetrics
4      # Print the average metrics
5      print("Average evaluator metrics for each parameter combination:")
6      for i, metric in enumerate(avgMetrics):
7          print(f"Parameter combination {i + 1}: {metric}")
8
9      # Specify the path to save the best model
10     output_path = f"hdfs://localhost:9000/usr/bigdatapoint/models/{model_name}"
11
12     # Save the best model
13     best_model.write().overwrite().save(output_path)
14     print(f"Best model saved to {output_path}")

```

Figure 24: Code for Best model selection

This two-step evaluation process ensures that the final model is both robust and generalizable to new data.

4.7 Prediction



```
1 # Map numeric predictions back to original labels
2 label_mapping = list(enumerate(indexer_label.labels))
3 numeric_to_label = {num: label for num, label in label_mapping}
4
5 # Convert predicted numeric to original labels
6 def map_numeric_to_label(numeric_value):
7     return numeric_to_label.get(int(numeric_value), "Unknown")
8
9 # Register UDF for mapping
10 from pyspark.sql.functions import udf
11 from pyspark.sql.types import StringType
12
13 map_to_label_udf = udf(map_numeric_to_label, StringType())
14
15 # Add original label to predictions
16 final_predictions = predictions_with_labels.withColumn(
17     "predicted_label",
18     map_to_label_udf(col("predicted_numeric"))
19 )
20
21 # Show the results
22 final_predictions.select("features", "predicted_numeric", "predicted_label").show(truncate=False)
```

Figure 25: Code for predicting model with test data

4.7.1 Steps in the Prediction Process

1. **Load the Model:** The trained model is loaded from the specified path in HDFS.
2. **Generate Predictions:** The `transform()` method of the pipeline model is used to generate predictions on the test dataset.
3. **Extract Relevant Information:** The columns for features, numeric predictions, and predicted probabilities are selected from the prediction results.
4. **Map Numeric Predictions to Original Labels:** A mapping is created between numeric predictions and their corresponding original labels. A user-defined function (UDF) is used to perform this mapping.
5. **Add Original Labels:** The original labels corresponding to the numeric predictions are added to the predictions dataset as a new column.
6. **Display Results:** The final dataset, containing features, numeric predictions, and original labels, is displayed for analysis.

4.7.2 Output

The final output contains the following columns:

- **Features:** The input features used for prediction.
- **Predicted Numeric:** The numeric representation of the predicted label.
- **Predicted Label:** The original label corresponding to the numeric prediction.

features	predicted_numeric	predicted_label
(131,[0,1,2,101],[1.0,1.0,1.0,1.0])	15.0	Fungal infection
(131,[3,4,5,102],[1.0,1.0,1.0,1.0])	4.0	Allergy
(131,[7,8,9,11,24,55],[1.0,1.0,1.0,1.0,1.0,1.0])	16.0	GERD
(131,[0,11,32,34,35,39,43],[1.0,1.0,1.0,1.0,1.0,1.0,1.0])	9.0	Chronic cholestasis
(131,[0,1,7,12,13],[1.0,1.0,1.0,1.0,1.0])	14.0	Drug Reaction
(131,[11,30,35,39,91,92],[1.0,1.0,1.0,1.0,1.0,1.0])	33.0	Peptic ulcer disease
(131,[10,22,25,74],[1.0,1.0,1.0,1.0])	1.0	AIDS
(131,[14,19,20,21,23,48,66,73,103,104],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])	12.0	Diabetes
(131,[11,26,29,40],[1.0,1.0,1.0,1.0])	17.0	Gastroenteritis
(131,[14,24,25,27,105,106],[1.0,1.0,1.0,1.0,1.0,1.0])	6.0	Bronchial Asthma
(131,[31,55,63,84,108],[1.0,1.0,1.0,1.0,1.0])	23.0	Hypertension
(131,[8,30,31,48,73,80,94,95,109],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])	30.0	Migraine
(131,[37,56,62,63,84],[1.0,1.0,1.0,1.0,1.0])	7.0	Cervical spondylosis
(131,[11,31,86,97],[1.0,1.0,1.0,1.0])	32.0	Paralysis (brain hemorrhage)
(131,[0,11,14,19,25,32,33,39],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])	28.0	Jaundice
(131,[5,11,25,28,31,34,40,96],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])	29.0	Malaria
(131,[0,1,14,21,25,31,35,41,46,47,98],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])	8.0	Chicken pox
(131,[1,5,6,11,14,25,31,34,35,36,37,47,96,98],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])	11.0	Dengue
(131,[5,11,14,25,31,34,38,39,40,93,99],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])	37.0	Typhoid
(131,[6,11,32,33,34,35,39,40,41,43,96],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])	40.0	Hepatitis A

only showing top 20 rows

Figure 26: Result of model prediction

5 Deployment

5.1 Overview

The deployment of the Disease Prognosis System was implemented using a combination of **Streamlit** for the web interface and **Apache Spark** for handling machine learning models and predictions. The system allows users to input symptoms and obtain disease predictions based on pre-trained models.

5.2 Tools and Frameworks

The following tools and frameworks were utilized for the deployment:

- **Streamlit**: A Python-based framework for creating interactive web applications.
- **Apache Spark**: A distributed computing framework used for model training and predictions.
- **PySpark**: Python API for Apache Spark.

5.3 Web Application Design

- The web application has a clean and user-friendly interface, designed to guide users through the process of selecting symptoms and obtaining predictions.

Figure 27: User interface of the Web

- The application includes the following features:
 - **Symptom Selection:** Users can select up to five symptoms from a predefined list.
 - **Prediction Results:** Displays the disease predictions based on Random Forest and Decision Tree models.
 - **Medical Disclaimer:** Provides a disclaimer encouraging users to consult medical professionals for accurate diagnoses.

5.4 System Components

- **Spark Session Initialization:** The system initializes a Spark session with a check to avoid duplicate Spark contexts:

```
if SparkContext._active_spark_context:
    SparkContext._active_spark_context.stop()
```

- **Model Loading:** Pre-trained machine learning models, including Random Forest and Decision Tree models, are loaded from the filesystem.
- **Symptom Mapping:** Selected symptoms are converted to a binary vector to match the feature schema used during model training.
- **Feature Assembly:** The `Spark VectorAssembler` is used to combine features into a vector for input into the prediction models.

5.5 Prediction Workflow

1. Users select symptoms through dropdown menus in the Streamlit interface.
2. The selected symptoms are mapped to a binary vector format using Pandas and converted to a Spark DataFrame.

3. Pre-trained machine learning models (Random Forest, Decision Tree) are applied to predict the most likely disease based on the input symptoms.
4. Results are displayed, including the predicted disease and a message encouraging professional medical consultation for severe cases.

5.6 Error Handling and Notifications

- Proper error handling is implemented to notify users of any issues, such as missing files or failed predictions.
- The application provides user-friendly messages for scenarios like:
 - Missing feature files.
 - Errors during model loading or prediction processes.

5.7 Deployment Instructions

1. Install the required Python libraries, including Streamlit and PySpark.
2. Start the Streamlit application using:

```
streamlit run app.py
```

3. Ensure the feature list (`features.json`) and pre-trained models are available in the specified paths.
4. Access the application via the provided URL (typically `http://localhost:8501`).

5.8 Future Development

- Integration of additional models, such as Naive Bayes, for enhanced predictions.
- Expanding the dataset to include a broader range of symptoms and diseases.
- Real-time processing using Spark Streaming for instantaneous predictions.

5.9 Disclaimer

The application is designed for light disease prognosis and is not a substitute for professional medical advice. Users are encouraged to consult healthcare professionals for accurate diagnoses.

6 Future Development

6.1 Expanding Dataset and Features

To improve the system's accuracy and usability, we plan to incorporate a broader range of symptoms, diseases, and patient demographics. This enhancement will ensure a more comprehensive dataset, enabling predictions that cater to diverse populations and health conditions.

6.2 Integration with Wearable Devices

The integration of real-time data from IoT and wearable devices, such as smartwatches and fitness trackers, will provide enhanced predictive capabilities. These devices can supply continuous updates on vital signs and activity levels, enriching the system's ability to offer timely and personalized health insights.

6.3 Advanced Models

We aim to explore and implement deep learning frameworks that can identify complex relationships and patterns in the data. Advanced models will enhance prediction accuracy, especially for multifaceted conditions and rare diseases.

6.4 Real-Time Processing

By incorporating Spark Streaming, the system will enable real-time data analysis and predictions. This capability is essential for scenarios requiring immediate insights, such as emergency health monitoring and critical care support.

References

1. [Spark Performance: Local FS vs. HDFS Replication Factor - Dip 215MC](#)
2. [Disease Prediction Using Machine Learning Dataset - Kaggle](#)
3. [Apache Spark 3.5.1: Memory Management Overview](#)
4. [Cross Validation in Machine Learning - GeeksforGeeks](#)
5. [Apache Spark Python API Documentation](#)