

# **DAYANANDA SAGAR COLLEGE OF ENGINEERING**

(An Autonomous Institute affiliated to VTU, Belagavi, Approved by AICTE & ISO 9001:2008 Certified)

Accredited by National Assessment & Accreditation Council (NAAC) with 'A' grade, Shavige

Malleshwara Hills, Kumaraswamy Layout, Bengaluru-560078.



## **Mini Project Report**

on

## **“Document Type Classification using PySpark”**

Submitted By

**P Charith [1DS18CS086]**

**Tharunesh [1DS18CS088]**

**Prajwal Kumar B R[1DS18CS092]**

**5<sup>th</sup> Semester B.E (CSE)**

in

## **Cloud and Big Data Lab**

Under the guidance of

**Prof Anitha M**

**Assistant Professor**

**Department of CSE**

**DSCE, Bangalore**

**Department of Computer Science and Engineering**

**Dayananda Sagar College of Engineering**

**Bangalore-78**

### ABSTRACT

Document Classification analogous to general classification of instances, deals with assigning labels to documents. The documents can be of different length, structure (can be written by different authors using a variety of writing styles) and source.

Here, multi-class classification is used to classify the document types into the respective categories they belong to. Different types of models/classifiers are defined to predict the categories of the articles and their accuracy are studied.

Handling a large number of documents with various types of information is a very challenging task. When it is done manually, it is very time consuming and labor intensive as the user has to look into the data of each file to classify it. Even if this task is automated locally by the user, it is still time consuming and compute intensive as handling this huge data requires more computing power.

Developing a ML model locally to classify documents is the most challenging problem as the data required for training the model is huge, hence requires more computing power to get results fastly.

One of the main goals of this project is to optimize the handling and processing of huge data and to observe it using the Spark environment.

Finally, the considered problem statement is “Developing a machine learning model and handling/ processing Big Data using PySpark, for classifying different document(text file) types.”

## TABLE OF CONTENT

Sl.no	Particular	Page No.
1	Introduction	1-1
2	Design and Implementation	2-7
3	Testing and Analysis	8-14
4	Conclusion and Future Enhancements	15-15
5	References	16-16

# INTRODUCTION

The Project is based on Distributed Big Data Technology. Big Data Analytics(BDA) is the use of advanced analytic techniques against very large, diverse big data sets that include structured, semi-structured and unstructured data, from different sources, and in different sizes from terabytes to zettabytes. Big Data concerned with 5 v's volume, value, velocity, veracity, variety. In this project we make use of Apache spark.

Apache Spark is an open-source unified analytics engine for large-scale data processing. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.

This project could be applied in cases where the data is very huge and requires to be classified. Custom models can be created based on the training data and the training process can be optimally performed via PySpark. For example, Consider a college scenario where the documents are of different types such as college financial data, student's data, teacher's data etc, here the proposed solution can help to classify the data efficiently.

## TECHNOLOGIES USED:

**Apache Spark:** Apache Spark is a Big Data framework which operates on distributed data collections. It furnishes in-memory computations for improved and quicker data processing over MapReduce. It is a cluster-computing framework which is designed for faster data computations.

**Jupyter Notebook:** The Jupyter Notebook is a web-based interactive computing platform. The notebook combines live code, equations, narrative text, visualizations.

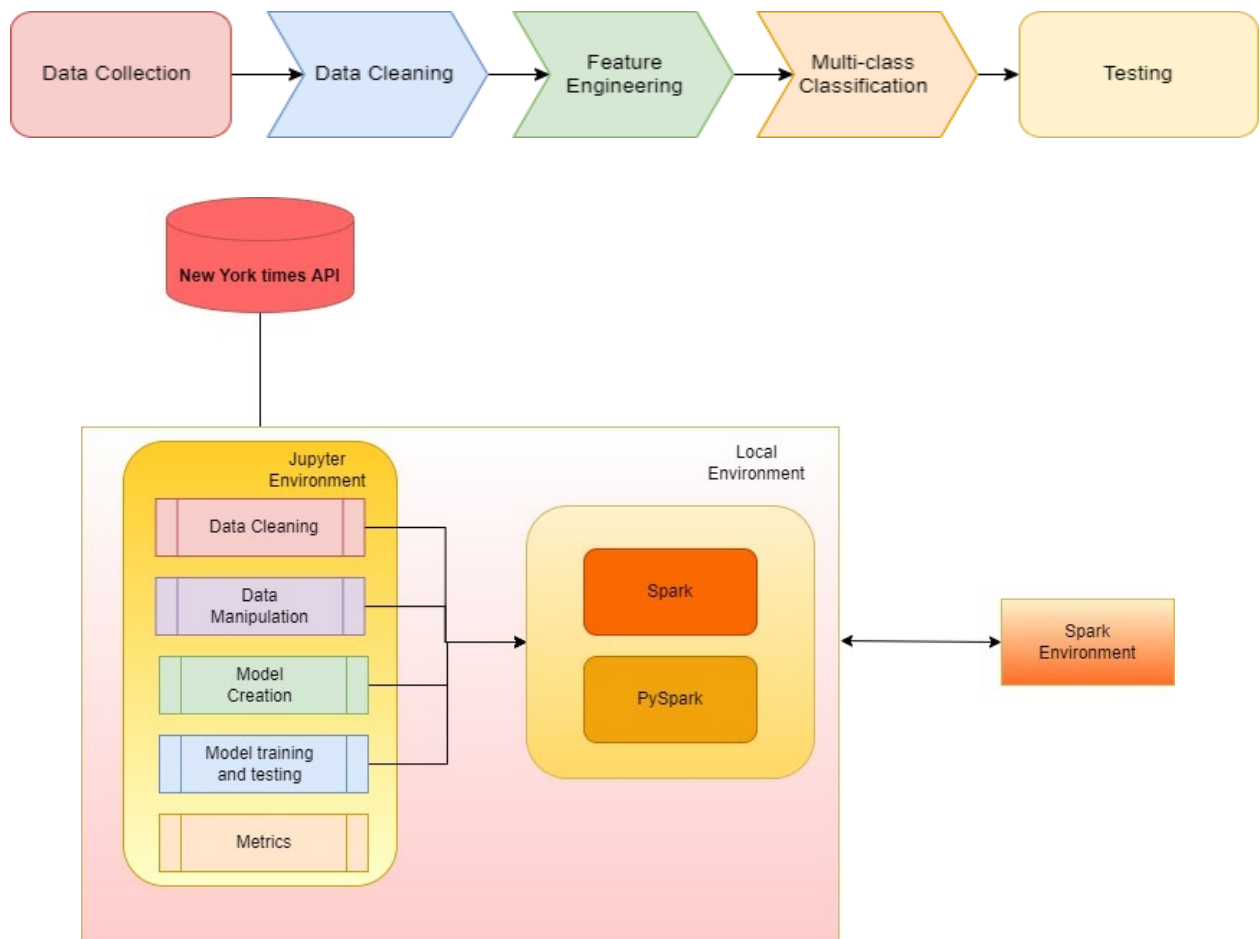
**PySpark:** PySpark is an interface for Apache Spark in Python. It not only allows you to write Spark applications using Python APIs, but also provides the PySpark shell for interactively analyzing your data in a distributed environment.

**NewYorkTimes API:** The Article Search API returns a max of 10 results at a time. The meta node in the response contains the total number of matches ("hits") and the current offset. Use the page query parameter to paginate thru results (page=0 for results 1-10, page=1 for 11-20). You can paginate through up to 100 pages (1,000 results). If you get too many results try filtering by date range.

## DESIGN AND IMPLEMENTATION

### This project contains 5 Stages :

- ❑ Data Collection, where data is collected from NewYorkTimes API.
- ❑ Data cleaning where Tokenization is performed and stop words are removed.
- ❑ Feature Engineering where the features are extracted using the functions HashingTF() and IDF().
- ❑ Multi-class classification using Logistic Regression and NaiveBayes classifiers and Testing , where classifiers are tested on unknown data.
- ❑ Throughout the computing process, data handling is observed in the spark environment.



**Complete Flowchart of the Project**

### Stages

#### 1. Data Collection:

New York Times articles are used as the input data. Data is collected with the NewYorkTimes API using the nytimesarticle package in python. Data is collected for the following categories: Fashion, Technology, Science and Movie. The collected data are stored in separate directories. The data is then processed in the next step. Data main folder has multiple sub-folders which contain the related text files. Data consists of text files based on following class:

1) Fashion 2) Technology 3) Science 4) Movie

```
import csv
import nytimesarticle
from nytimesarticle import articleAPI
api = articleAPI('WdG8nqXj0Sskbm33dkGsBOXPgIf5u7i0o')
```

#### Necessary libraries

```
all_articles = []
for i in range(0,100):
    articles = api.search(q = "fashion",page = i)
    articles = parse_articles(articles)
    all_articles = all_articles + articles
```

#### Using API to obtain articles.

```
all_articles
```

```
[{'url': 'https://www.nytimes.com/1963/04/07/archives/formula-more-movies-opposition-popularity-attraction.html'},  
{'url': 'https://www.nytimes.com/1980/04/13/archives/theater-merton-of-movies-returns-the-cast.html'},  
{'url': 'https://www.nytimes.com/1969/01/26/archives/hollywoods-haunted-movies-the-movies-that-still-haunt-hollywood.html'},  
{'url': 'https://www.nytimes.com/1969/01/05/archives/the-movies-are-now-high-art-the-movies-are-now-high-art.html'},  
{'url': 'https://www.nytimes.com/1966/01/05/archives/cbstv-to-offer-movies-a-2d-night.html'},  
{'url': 'https://www.nytimes.com/1980/03/28/archives/at-the-movies.html'},  
{'url': 'https://www.nytimes.com/1969/10/05/archives/whats-at-the-movies.html'},  
{'url': 'https://www.nytimes.com/1965/01/03/archives/fadeout-fadein-movies-the-history-of-an-art-and-an-institution-by.html'},  
{'url': 'https://www.nytimes.com/1967/02/07/archives/airline-plans-shift-to-movies-in-color.html'},  
{'url': 'https://www.nytimes.com/1963/06/03/archives/producers-favor-lowcost-movies-journal-urges-quality-film-as-remedy.html'},  
{'url': 'https://www.nytimes.com/1967/12/02/archives/books-of-the-times-off-to-the-picture-show-superlatives-abound.html'},  
... ..]
```

### Output after the API call

```
fas_df = spark.read.text('Data/Fashion/*')  
fas_df = fas_df.withColumn("category",lit("Fashion"))  
  
tech_df = spark.read.text('Data/Technology/*')  
tech_df = tech_df.withColumn("category",lit("Technology"))  
  
sci_df = spark.read.text('Data/Science/*')  
sci_df = sci_df.withColumn("category",lit("science"))  
  
mov_df = spark.read.text('Data/Movie/*')  
mov_df = mov_df.withColumn("category",lit("Movie"))  
  
merge_df1 = fas_df.union(tech_df)  
merge_df2 = merge_df1.union(sci_df)  
merge_df3 = merge_df2.union(mov_df)
```

## 2. Data Cleaning:

### RegexTokenizer

A regex based tokenizer that extracts tokens either by using the provided regex pattern (in Java dialect) to split the text (default) or repeatedly matching the regex (if gaps is false). Optional parameters also allow filtering tokens using a minimal length. It returns an array of strings that can be empty.

```
regexTokenizer = RegexTokenizer(inputCol="value", outputCol="words", pattern="\\W")
```

### StopWordsRemover

A feature transformer that filters out stop words from input. Since 3.0.0, StopWordsRemover can filter out multiple columns at once by setting the inputCols parameter. Note that when both the inputCol and inputCols parameters are set, an Exception will be thrown.

```
add_stopwords=nlk.corpus.stopwords.words('english')
add_stopwords_1 = ["nytimes","com","sense","day","common","business","todays","said","food","review","sunday","letters"]
stopwordsRemover = StopWordsRemover(inputCol="words", outputCol="filtered1").setStopWords(add_stopwords)
stopwordsRemover1 = StopWordsRemover(inputCol="filtered1", outputCol="filtered").setStopWords(add_stopwords_1)
```

The data that has been collected is cleaned in the following steps: Tokenization: Data is initially split into words(tokenized) using the regexTokenizer() which tokenizes the data with regular expressions. Removing stop words: After the tokenization of the data, the stop words are removed using the StopWordsRemover() which removes all the stop words in the data.

### 3. Feature Engineering:

**StringIndexer** : A label indexer that maps a string column of labels to an ML column of label indices. If the input column is numeric, we cast it to string and index the string values.

```
label_stringIdx = StringIndexer(inputCol = "category", outputCol = "label")
```

**Hashingtf**: Maps a sequence of terms to their term frequencies using the hashing trick. Currently we use Austin Appleby's MurmurHash 3 algorithm (MurmurHash3\_x86\_32) to calculate the hash code value for the term object. Since a simple modulo is used to transform the hash function to a column index, it is advisable to use a power of two as the numFeatures parameter; otherwise the features will not be mapped evenly to the columns.

```
hashingTF = HashingTF(inputCol="filtered", outputCol="rawFeatures", numFeatures=1000)
```

**IDF**: The standard formulation is used:  $\text{idf} = \log((m + 1) / (d(t) + 1))$ , where  $m$  is the total number of documents and  $d(t)$  is the number of documents that contain term  $t$ . This implementation supports filtering out terms which do not appear in a minimum number of documents (controlled by the variable minDocFreq). For terms that are not in at least minDocFreq documents, the IDF is found as 0, resulting in TF-IDFs of 0.

```
idf = IDF(inputCol="rawFeatures", outputCol="features", minDocFreq=5)
```

**Pipeline**: A simple pipeline, which acts as an estimator. A Pipeline consists of a sequence of stages, each of which is either an Estimator or a Transformer. When Pipeline.fit() is called, the stages are executed in order. If a stage is an Estimator, its Estimator.fit() method will be called on the input dataset to fit a model. Then the model, which is a transformer, will be used to transform the dataset as the input to the next stage. If a stage is a Transformer, its Transformer.transform() method will be called to produce the dataset for the next stage. The fitted model from a Pipeline is a PipelineModel, which consists of fitted models and transformers, corresponding to the pipeline stages. If stages is an empty list, the pipeline acts as an identity transformer.



```
pipeline = Pipeline(stages=[regexTokenizer, stopwordsRemover,stopwordsRemover1, hashingTF, idf, label_stringIdx])
```

After the data has been cleaned, the features are extracted characterizing each category. The categories are converted into labels from 0 to 3 using the StringIndexer(). The features are extracted using the functions HashingTF() and IDF(). HashingTF: HashingTF is a Transformer which takes sets of terms and converts those sets into fixed-length feature vectors. IDF: IDF is an Estimator which fits on a dataset and produces an IDFModel. The IDFModel takes feature vectors and scales each column. Intuitively, it down-weights columns which appear frequently in a corpus. HashingTF is used to hash the sentence into a feature vector and IDF is used to rescale the feature vectors. A pipeline is constructed using the above functions which cleans the data and extracts the features. `pipeline = Pipeline(stages=[regexTokenizer, stopwordsRemover, hashingTF, idf, label_stringIdx])`

**4. Multi-class Classification:** The classification algorithms used in this project are Logistic Regression ,Naïve Bayes, Decision Tree, Random Forest and One vs Rest classifier. The dataset is split into training data (80%) and test data (20%). The package pyspark.ml is used for the multi-class classification. Logistic Regression: Logistic Regression is carried out using the function, `LogisticRegression(maxIter=20, regParam=0.3, elasticNetParam=0)`. Naïve Bayes classification: The Naïve Bayes classification is carried out using the function, `NaiveBayes(smoothing=1)`. Decision Tree is implemented using `dt = DecisionTreeClassifier(impurity="gini")`. Random Forest is implemented using `rf = RandomForestClassifier(numTrees=50)`. One vs Rest is implemented using `ovr = OneVsRest(classifier=lr)` where lr is Logistic Regression.

### Logistic Regression

Logistic regression is a popular method to predict a categorical response. It is a special case of Generalized Linear models that predicts the probability of the outcomes. In spark.ml logistic regression can be used to predict a binary outcome by using binomial logistic regression, or it can be used to predict a multiclass outcome by using multinomial logistic regression. Use the family parameter to select between these two algorithms, or leave it unset and Spark will infer the correct variant.

Multinomial logistic regression can be used for binary classification by setting the family param to “multinomial”. It will produce two sets of coefficients and two intercepts.

```
lr = LogisticRegression(maxIter=20, regParam=0.3, elasticNetParam=0)
lrModel = lr.fit(trainingData)
```

### Naive Bayes classifiers

Naive Bayes classifiers are a family of simple probabilistic, multiclass classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between every pair of features.

```
nb = NaiveBayes(smoothing=1)
model = nb.fit(trainingData)
```

Naive Bayes can be trained very efficiently. With a single pass over the training data, it computes the conditional probability distribution of each feature given each label. For prediction, it applies Bayes' theorem to compute the conditional probability distribution of each label given an observation.

### Decision tree classifier

Decision trees and their ensembles are popular methods for the machine learning tasks of classification and regression. Decision trees are widely used since they are easy to interpret, handle categorical features, extend to the multiclass classification setting, do not require feature scaling, and are able to capture non-linearities and feature interactions. Tree ensemble algorithms such as random forests and boosting are among the top performers for classification and regression tasks.

```
dt = DecisionTreeClassifier(impurity="gini")
dtModel = dt.fit(trainingData)
```

### Random forest classifier

Random forests are ensembles of decision trees. Random forests combine many decision trees in order to reduce the risk of overfitting. The spark.ml implementation supports random forests for binary and multiclass classification and for regression, using both continuous and categorical features.

```
rf = RandomForestClassifier(numTrees=50)
rfModel = rf.fit(trainingData)
```

### One-vs-Rest

OneVsRest is an example of a machine learning reduction for performing multiclass classification given a base classifier that can perform binary classification efficiently. It is also known as "One-vs-All"

```
ovr = OneVsRest(classifier=lr)
ovrModel = ovr.fit(trainingData)
```

**5. Testing:** Unknown set of articles are used as the test set and it is processed into the pipeline which cleans and extracts features for the test data. This is then used in the classification algorithm for predicting the categories.

## TESTING AND ANALYSIS

### Logistic Regression

```
evaluator = MulticlassClassificationEvaluator(predictionCol="prediction")
train_accuracy = evaluator.evaluate(predictions_train)*100
print("-----Accuracy of train data using logistic_regression-----: " + str(evaluator.evaluate(predictions_train)*100)+"%")
```

```
-----Accuracy of train data using logistic_regression-----: 98.76222962433778%
```

#### Similarly for test and unknown data

```
-----Accuracy of test data using logistic_regression-----: 64.6386141204794%
-----Accuracy of unknown data using logistic_regression-----: 13.2128740824393%
```

---

### Naive Bayes Classifier

## Document Type Classification using PySpark

```
evaluator = MulticlassClassificationEvaluator(predictionCol="prediction")
train_accuracy= evaluator.evaluate(predictions_train)*100
print("-----Accuracy of train data using Naive Bayes Classifier-----: " + str(evaluator.evaluate(predictions_train)*100)+"%")
```

```
-----Accuracy of train data using Naive Bayes Classifier-----: 93.01496019989493%
```

### Similarly for test and unknown data

```
-----Accuracy of test data using naive_bayes-----: 58.8554632688887%
-----Accuracy of unknown data using naive_bayes-----: 31.242077171133104%
```

---

## Decision Tree Classifier

```
evaluator = MulticlassClassificationEvaluator(predictionCol="prediction")
print("-----Accuracy of train data using Decision Tree-----: " + str(evaluator.evaluate(predictions_dt)*100)+"%")
train_accuracy= evaluator.evaluate(predictions_dt)*100
```

```
-----Accuracy of train data using Decision Tree-----: 68.25192028280541%
```

### Similarly for test and unknown data

```
-----Accuracy of test data using Decision Tree-----: 60.854299475524165%
-----Accuracy of unknown data using Decision Tree-----: 27.29935032483758%
```

---

## Random Forest Classifier

```
evaluator = MulticlassClassificationEvaluator(predictionCol="prediction")
print("-----Accuracy of train data using Random Forest-----: " + str(evaluator.evaluate(predictions_rf)*100)+"%")
train_accuracy= evaluator.evaluate(predictions_rf)*100
```

```
-----Accuracy of train data using Random Forest-----: 73.71011365147397%
```

### Similarly for test and unknown data

```
-----Accuracy of test data using Random Forest-----: 63.919555981872556%
```

```
-----Accuracy of unknown data using Random Forest-----: 37.06631532718489%
```

---

## One-vs-Rest

```
evaluator = MulticlassClassificationEvaluator(predictionCol="prediction")
train_accuracy= evaluator.evaluate(predictions_svc)*100
print("-----Accuracy of train data using One-vs-Rest-----: " + str(train_accuracy)+"%")
```

```
-----Accuracy of train data using One-vs-Rest-----: 98.3488377709006%
```

### Similarly for test and unknown data

```
-----Accuracy of test data using One-vs-Rest-----: 64.61930305407222%
```

```
-----Accuracy of unknown data using One-vs-Rest-----: 13.2128740824393%
```

## Final Accuracy Metrics

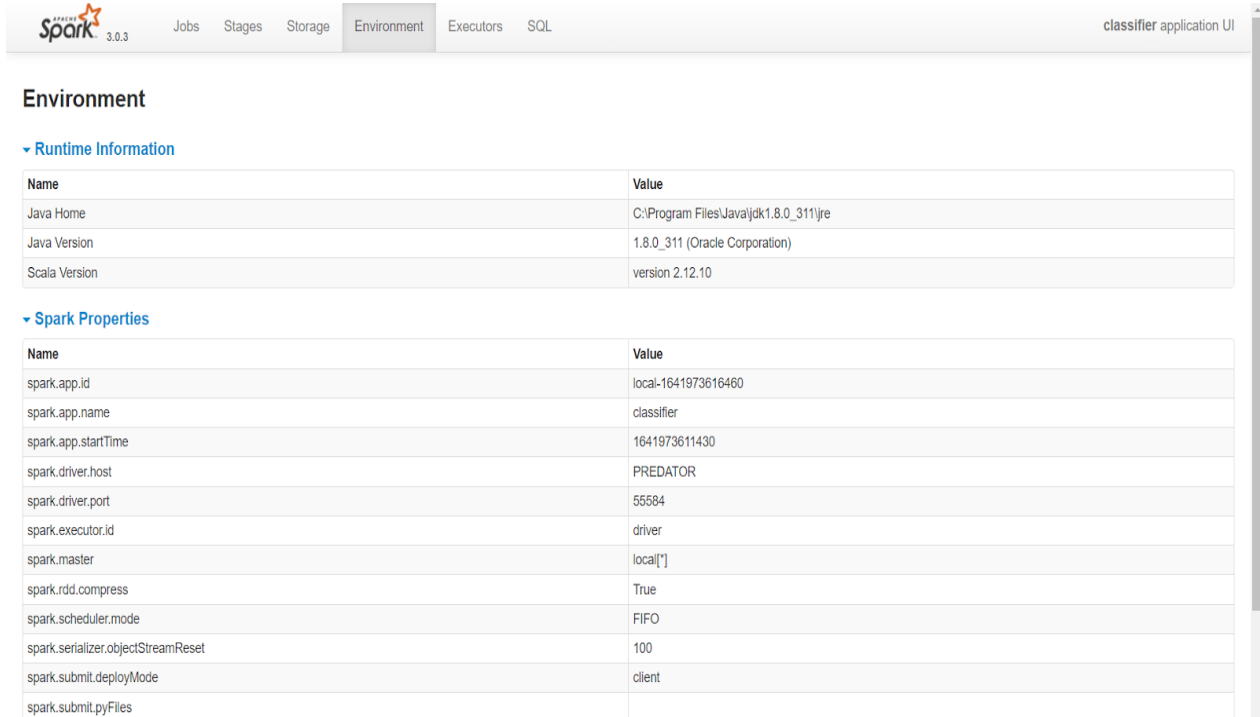
	<b>metric</b>	<b>Logistic Regression</b>	<b>Naïve Bayes</b>	<b>Decision Tree</b>	<b>Random Forest</b>	<b>One-vs-Rest</b>
<b>0</b>	Train_Accuracy	98.762230	93.014960	68.251920	73.710114	98.348838
<b>1</b>	Test_Accuracy	64.638614	58.855463	60.854299	63.919556	64.619303
<b>2</b>	Unknown_Accuracy	13.212874	31.242077	27.299350	37.066315	13.212874

## Spark Environment

One of the main reasons to provide this solution of distributed computing for handling the abundant text files, as per the mentioned problem, is to understand the working of spark using its environment.

With Spark environments, you can configure the size of the Spark driver and the size and number of the executors. Service Spark environments are not available by default. The Spark environment provides a detailed view and review of the tasks that are distributed. A visual display is also provided to understand better.

### Environment Tab



The screenshot shows the Spark Environment tab in the Databricks UI. The top navigation bar includes 'Jobs', 'Stages', 'Storage', 'Environment' (selected), 'Executors', and 'SQL'. The main content area is titled 'Environment' and contains two sections: 'Runtime Information' and 'Spark Properties'.

Name	Value
Java Home	C:\Program Files\Java\jdk1.8.0_311\re
Java Version	1.8.0_311 (Oracle Corporation)
Scala Version	version 2.12.10

Name	Value
spark.app.id	local-1641973616460
spark.app.name	classifier
spark.app.startTime	1641973611430
spark.driver.host	PREDATOR
spark.driver.port	55584
spark.executor.id	driver
spark.master	local[*]
spark.rdd.compress	True
spark.scheduler.mode	FIFO
spark.serializer.objectStreamReset	100
spark.submit.deployMode	client
spark.submit.pyFiles	

The above shows the environment page of spark. This is a useful place to check to make sure that your properties have been set correctly. Note that only values explicitly specified through spark-defaults.conf, SparkConf, or the command line will appear. For all other configuration properties, you can assume the default value is used. The Environment tab displays the values for the different environment and configuration variables, including JVM, Spark, and system properties.

## Document Type Classification using PySpark

This environment page has five parts. It is a useful place to check whether your properties have been set correctly. The first part 'Runtime Information' simply contains the [runtime properties](#) like versions of Java and Scala. The second part 'Spark Properties' lists the [application properties](#) like 'spark.app.name' and 'spark.driver.memory'.

Clicking the 'Hadoop Properties' link displays properties relative to Hadoop and YARN. Note that properties like 'spark.hadoop.\*' are shown not in this part but in 'Spark Properties'.

'System Properties' shows more details about the JVM. The last part 'Classpath Entries' lists the classes loaded from different sources, which is very useful to resolve class conflicts.

## Executor Tab

### Executors

[Show Additional Metrics](#)

#### Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(1)	34	4.7 MIB / 366.3 MIB	0.0 B	8	9	0	3842	3851	6.8 min (13 s)	58.4 MIB	29.2 MIB	31.5 MIB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1)	34	4.7 MIB / 366.3 MIB	0.0 B	8	9	0	3842	3851	6.8 min (13 s)	58.4 MIB	29.2 MIB	31.5 MIB	0

#### Executors

Show 20 entries

Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	PREDATOR:50236	Active	34	4.7 MIB / 366.3 MIB	0.0 B	8	9	0	3842	3851	6.8 min (13 s)	58.4 MIB	29.2 MIB	31.5 MIB	<a href="#">Thread Dump</a>

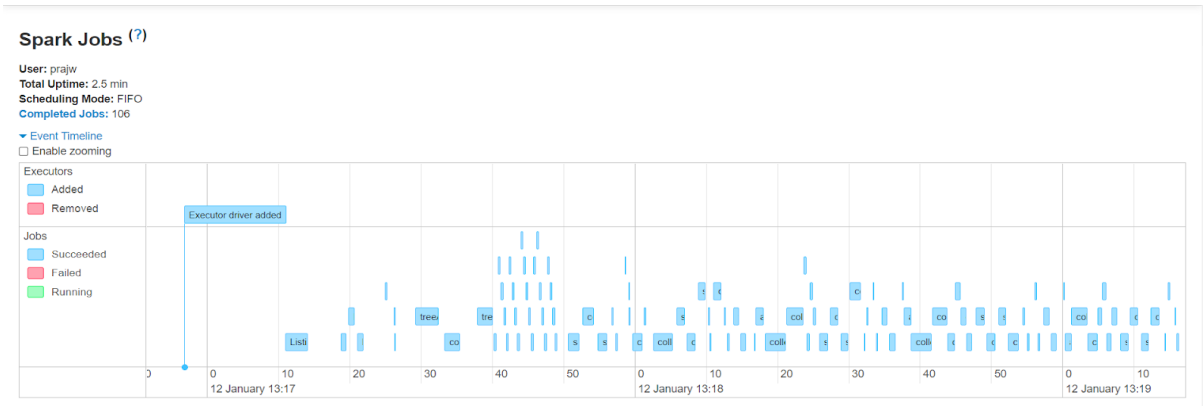
Showing 1 to 1 of 1 entries

[Previous](#) [1](#) [Next](#)

The Executors tab displays summary information about the executors that were created for the application, including memory and disk usage and task and shuffle information. The Storage Memory column shows the amount of memory used and reserved for caching data.

The Executors tab provides not only resource information (amount of memory, disk, and cores used by each executor) but also performance information (GC time and shuffle information).

## Job Tab



The Jobs tab displays a summary page of all jobs in the Spark application and a details page for each job. The summary page shows high-level information, such as the status, duration, and progress of all jobs and the overall event timeline. When you click on a job on the summary page, you see the details page for that job. The details page further shows the event timeline, DAG visualization, and all stages of the job.

The information that is displayed in this section is

- User: Current Spark user
- Total uptime: Time since Spark application started
- Scheduling mode: See [job scheduling](#)
- Number of jobs per status: Active, Completed, Failed
- Event timeline: Displays in chronological order the events related to the executors (added, removed) and the jobs (the above image shows the timeline)
- Details of jobs grouped by status: Displays detailed information of the jobs including Job ID, description (with a link to detailed job page), submitted time, duration, stages summary and tasks progress bar

When you click on a specific job, you can see the detailed information of this job.

### Job Detail

This page displays the details of a specific job identified by its job ID.

- Job Status: (running, succeeded, failed)
- Number of stages per status (active, pending, completed, skipped, failed)
- Associated SQL Query: Link to the sql tab for this job
- Event timeline: Displays in chronological order the events related to the executors (added, removed) and the stages of the job



## Stages Tab

### Stages for All Jobs

Completed Stages: 180

▼ Completed Stages (180)

Page: 1 2 > 2 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
179	collectAsMap at MulticlassMetrics.scala:61	+details 2022/01/12 13:19:16	17 ms	23/23			7.0 KIB	
178	map at MulticlassMetrics.scala:52	+details 2022/01/12 13:19:15	0.2 s	23/23	63.7 KIB			7.0 KIB
177	showString at <unknown>:0	+details 2022/01/12 13:19:15	0.1 s	23/23	63.7 KIB			
176	collect at StringIndexer.scala:204	+details 2022/01/12 13:19:14	59 ms	1/1			10.9 KIB	
175	collect at StringIndexer.scala:204	+details 2022/01/12 13:19:14	0.2 s	23/23	63.7 KIB			10.9 KIB
174	treeAggregate at IDF.scala:55	+details 2022/01/12 13:19:14	28 ms	4/4			20.0 KIB	
173	treeAggregate at IDF.scala:55	+details 2022/01/12 13:19:14	0.2 s	23/23	63.7 KIB			20.0 KIB
172	collectAsMap at MulticlassMetrics.scala:61	+details 2022/01/12 13:19:13	43 ms	34/34			12.8 KIB	
171	map at MulticlassMetrics.scala:52	+details 2022/01/12 13:19:12	1 s	34/34	1274.0 KIB			12.8 KIB
170	showString at <unknown>:0	+details 2022/01/12 13:19:10	1 s	34/34	1274.0 KIB			
169	collectAsMap at MulticlassMetrics.scala:61	+details 2022/01/12 13:19:10	39 ms	34/34			14.1 KIB	
168	map at MulticlassMetrics.scala:52	+details 2022/01/12 13:19:09	1 s	34/34	1274.0 KIB			14.1 KIB
167	showString at <unknown>:0	+details 2022/01/12 13:19:08	1 s	34/34	1274.0 KIB			
166	collectAsMap at RandomForest.scala:663	+details 2022/01/12 13:19:07	0.3 s	34/34			7.4 MIB	
165	mapPartitions at RandomForest.scala:644	+details 2022/01/12 13:19:06	0.4 s	34/34	3.0 MIB			7.4 MIB

The Stages tab displays a summary page that shows the current state of all stages of all jobs in the Spark application.

At the beginning of the page is the summary with the count of all stages by status (active, pending, completed, skipped, and failed)

After that are the details of stages per status (active, pending, completed, skipped, failed). In active stages, it's possible to kill the stage with the kill link. Only in failed stages, failure reason is shown. Task detail can be accessed by clicking on the description.

### In the proposed solution:

All the objects created using pyspark libraries, such as regextokenizer, decision tree, etc, when used to perform a task of data handling use distributed techniques, where the data is distributed amongst the many jobs and handled efficiently, conserving space and time.

## CONCLUSIONS AND FUTURE ENHANCEMENTS

In the current era of advanced technology, data plays a very important role. Hence, its storage and computing is of significant importance. Data is the most important part of all Data Analytics, Machine Learning, Artificial Intelligence. Without data, we can't train any model and all modern research and automation will go in vain. Data is critical for characterization, calibration, verification, validation, and assessment of models for predicting the long-term structural durability and performance of materials in extreme environments. Without adequate data to verify and assess them, many models would have no purpose.

In the considered problem statement, handling a large number of documents with various types of information is a very challenging task. When done manually, it is very time consuming and labor intensive as the user has to look into the data of each file to classify it. Even if this task is automated locally by the user, it is still time consuming and compute intensive as handling this huge data requires more computing power. Even if a machine learning model is created for classification, data feature engineering and model training is time intensive.

In order to process large amounts of Big Data, Distributed Computing is used. One such distributed computing Tool is Spark and its python counterpart is PySpark.

The proposed solution uses the distributed computing technologies for efficient data handling and for classification operation.

### **Future enhancements:**

- Train the considered model with more data.
- Considering parameter tuning.
- Advanced Feature engineering.
- Considering other available ML models.
- Considering other categories of data for classification.

## REFERENCES

- <https://cprosenjit.medium.com/9-classification-methods-from-spark-mllib-we-should-know-c41f555c0425>
- <https://towardsdatascience.com/multi-class-text-classification-with-pyspark-7d78d022ed35>
- <https://spark.apache.org/docs/latest/>
- <https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql.html>
- <https://spark.apache.org/docs/2.4.0/api/python/index.html>
- [https://www.youtube.com/watch?v=\\_C8kWso4ne4](https://www.youtube.com/watch?v=_C8kWso4ne4)