

## SKRIPSI

### STUDI DAN IMPLEMENTASI APACHE SPARK MLLIB UNTUK ANALISIS BIG DATA



Kresna Dwi Cahyo

NPM: 2014730048

PROGRAM STUDI TEKNIK INFORMATIKA  
FAKULTAS TEKNOLOGI INFORMASI DAN SAINS  
UNIVERSITAS KATOLIK PARAHYANGAN  
2018



**UNDERGRADUATE THESIS**

**STUDY AND IMPLEMENTATION OF APACHE SPARK  
MLLIB FOR BIG DATA ANALYSIS**



**Kresna Dwi Cahyo**

**NPM: 2014730048**

**DEPARTMENT OF INFORMATICS  
FACULTY OF INFORMATION TECHNOLOGY AND SCIENCES  
PARAHYANGAN CATHOLIC UNIVERSITY  
2018**



## ABSTRAK

Seiring dengan pemanfaatan teknologi informasi yang makin meluas, data dengan cepat terkumpul dan dalam waktu cepat dapat menjadi Big Data. Big data dapat dianalisis untuk mendapatkan informasi atau pengetahuan yang berharga. Namun, dibutuhkan komputer dengan kekuatan komputasi yang sangat tinggi untuk menganalisis data dengan ukuran yang sangat besar. Sistem terdistribusi adalah solusi dari masalah tersebut.

Salah satu jenis sistem terdistribusi adalah *Distributed Computing System*. *Distributed Computing System* merupakan sistem terdistribusi yang digunakan untuk komputasi dengan kebutuhan kinerja yang tinggi. Hadoop adalah salah satu *framework* yang banyak dipakai saat ini. Hadoop merupakan platform yang dapat menyediakan penyimpanan dan kemampuan komputasi terdistribusi. Seiring berjalannya waktu, *Framework* lainnya juga mulai bermunculan. Salah satunya adalah Apache Spark.

Apache Spark adalah sebuah *framework cluster computing* yang dapat dimanfaatkan untuk mengolah Big Data dengan cepat. Apache Spark dapat berjalan diatas infrastruktur Hadoop untuk meningkatkan fungsionalitas. MLlib merupakan *library* yang mengimplementasikan teknik analisis *machine learning*. Teknik *machine learning* dimiliki MLlib meliputi *statistic, classification, regression, collaborative filtering, clustering, Dimensionality reduction, and feature extraction*.

Perangkat lunak demo dikembangkan untuk dapat mengolah data dengan menggunakan beberapa fungsi dari *library* MLlib. Perangkat lunak demo berjalan pada lingkungan cluster hadoop. Karya tulis ini menjelaskan konsep dan cara kerja spark serta menunjukan cara pemanggilan fungsi-fungsi MLlib. Selain itu, modifikasi dilakukan pada library K-Means MLlib agar dapat menghasilkan suatu pola bagi pengguna. Pengujian fungsional dan uji performa dilakukan untuk menguji fitur-fitur pada perangkat lunak demo. Berdasarkan hasil pengujian yang sudah dilakukan, dapat disimpulkan bahwa kinerja dari fungsi-fungsi MLlib sangat baik untuk komputasi pada ukuran data yang besar.

**Kata-kata kunci:** Sistem Terdistribusi, Apache Spark, MLlib, Machine Learning, Hadoop, HDFS, K-Means, Scala



## ABSTRACT

Along with use of information technology that increasingly widespread, data quickly gathered. Data will become Big Data In a short time. Big Data can be analized to gain valuable information or a knowledge. However, it takes a computer with high power of computing to analized large data. Distributed system is the solution.

One type of distributed system is the Distributed Computing System. Distributed Computing System is a distributed system that used for high performance computing. Hadoop is one of the framework that commonly used. Hadoop is a platform that provide storage system and computational capability. As time goes by, another framework is developed. One of them is Apache Spark.

Apache Spark is a cluster computing platform that can be utilized to processing Big Data faster. Apache Spark can run on hadoop environment to gain functionality. MLlib is a Spark library which implements machine learning analysis technique. Such as statistic, classification, regression, collaborative filtering, clustering, dimensionality reduction, and feature extraction.

The demo software was developed to be able to process data by calling function from MLlib library. The demo software runs on hadoop cluster environment. This paper explain spark concepts, how spark works, and shows how to call MLlib function. In addition, modifications are made to the K-Means MLlib library in order to generate patterns for users. Functional tests and performance tests are performed to test features of the demo software. Based on the tests results that have been done, it can be concluded that the performance of MLlib functions is good for computing on large data size.

**Keywords:** Distributed System, Apache Spark, MLlib, Machine Learning, Hadoop, HDFS, K-Means, Scala



# DAFTAR ISI

<b>DAFTAR ISI</b>	<b>ix</b>
<b>DAFTAR GAMBAR</b>	<b>xi</b>
<b>DAFTAR TABEL</b>	<b>xv</b>
<b>1 PENDAHULUAN</b>	<b>1</b>
1.1 Latar Belakang . . . . .	1
1.2 Rumusan Masalah . . . . .	2
1.3 Tujuan . . . . .	2
1.4 Batasan Masalah . . . . .	2
1.5 Metodologi . . . . .	3
1.6 Sistematika Pembahasan . . . . .	3
<b>2 LANDASAN TEORI</b>	<b>5</b>
2.1 Sistem Terdistribusi . . . . .	5
2.1.1 Pengertian Sistem Terdistribusi . . . . .	5
2.1.2 Jenis Sistem Terdistribusi . . . . .	6
2.2 Hadoop . . . . .	6
2.2.1 MapReduce . . . . .	8
2.2.2 Hadoop Distributed File System (HDFS) . . . . .	9
2.2.3 Blok HDFS . . . . .	13
2.3 Apache Spark . . . . .	13
2.3.1 Komponen Apache Spark . . . . .	16
2.3.2 Arsitektur Apache Spark . . . . .	17
2.3.3 Resilient Distributed Datasets . . . . .	20
2.3.4 MLlib . . . . .	23
2.4 Scala . . . . .	31
2.4.1 Tipe Data . . . . .	32
2.4.2 Variabel . . . . .	33
2.4.3 Method dan Fungsi . . . . .	33
2.4.4 Iterasi dan Percabangan . . . . .	34
2.4.5 Objek . . . . .	35
2.4.6 Method Main . . . . .	36
<b>3 STUDI DAN EKSPLORASI APACHE SPARK MLlib</b>	<b>37</b>
3.1 Instalasi Apache Spark . . . . .	37
3.2 Eksplorasi menggunakan Spark Shell . . . . .	39
3.3 Pemanggilan Fungsi-Fungsi pada library MLlib . . . . .	40
3.3.1 Summary Statistics . . . . .	43
3.3.2 Naive Bayes . . . . .	44
3.3.3 Principal Component Analysis (PCA) . . . . .	46
3.3.4 Term Frequency - Inverse Document Frequency (TF-IDF) . . . . .	48

3.3.5	Alternating Least Squares (ALS) . . . . .	49
3.3.6	K-Means . . . . .	51
<b>4</b>	<b>PENGEMBANGAN K-MEANS PADA LINGKUNGAN SPARK</b>	<b>55</b>
4.1	Kebutuhan Pengembangan . . . . .	55
4.2	Analisis K-Means MLlib . . . . .	56
4.3	Modifikasi K-Means MLlib . . . . .	64
<b>5</b>	<b>PERANCANGAN, IMPLEMENTASI, DAN PENGUJIAN PERANGKAT LUNAK DEMO</b>	<b>69</b>
5.1	Diagram Use Case dan Skenario . . . . .	69
5.2	Diagram Kelas dan Pemanggilan Library MLlib . . . . .	73
5.3	Rancangan User Interface . . . . .	79
5.4	Implementasi Perangkat Lunak . . . . .	85
5.4.1	Lingkungan Perangkat Keras . . . . .	85
5.4.2	Lingkungan Perangkat Lunak . . . . .	85
5.4.3	User Interface . . . . .	86
5.4.4	Implementasi Pemanggilan Fungsi MLlib . . . . .	89
5.5	Pengujian Perangkat Lunak . . . . .	95
5.6	Eksperimen untuk Uji Performa Fungsi-fungsi MLlib . . . . .	101
5.6.1	Lingkungan Perangkat Keras . . . . .	101
5.6.2	Lingkungan Perangkat Lunak . . . . .	102
5.6.3	Hasil Eksperimen . . . . .	102
<b>6</b>	<b>KESIMPULAN DAN SARAN</b>	<b>109</b>
6.1	Kesimpulan . . . . .	109
6.2	Saran . . . . .	109
<b>DAFTAR REFERENSI</b>		<b>111</b>
<b>A</b>	<b>KODE PROGRAM</b>	<b>113</b>
<b>B</b>	<b>HASIL EKSPERIMEN</b>	<b>115</b>

## DAFTAR GAMBAR

2.1 Sistem terdistribusi yang diorganisir middleware . . . . .	5
2.2 Arsitektur Hadoop . . . . .	8
2.3 Arsitektur HDFS : Client berkomunikasi dengan NameNode(Master) dan DataNode(slave) . . . . .	9
2.4 Arsitektur HDFS : Client berkomunikasi dengan NameNode(Master) dan DataNode(slave) . . . . .	10
2.5 Diagram ekosistem data processing Spark . . . . .	14
2.6 Tahap pemrosesan data pada Spark . . . . .	14
2.7 Tahap pemrosesan data pada MapReduce Hadoop . . . . .	15
2.8 Tiga cara untuk melakukan deployment pada Spark . . . . .	15
2.9 Apache Spark stack . . . . .	16
2.10 Arsitektur Apache Spark . . . . .	18
2.11 RDD pada sebuah cluster . . . . .	20
2.12 Proses yang umum terjadi pada RDD . . . . .	22
2.13 Scatterplot contoh data dua dimensi . . . . .	29
2.14 Bentuk umum <i>function</i> Scala . . . . .	33
3.1 Tampilan Spark-Shell . . . . .	39
3.2 Membuat <i>project</i> SBT pada IntelliJ . . . . .	42
3.3 Mengatur konfigurasi project pada IntelliJ . . . . .	42
3.4 Membuat SparkContext . . . . .	43
3.5 Membuat data masukkan untuk Statistic . . . . .	43
3.6 Pemanggilan method colStat dan fungsi-fungsinya . . . . .	44
3.7 Hasil output percobaan Summary Statistic . . . . .	44
3.8 Membuat data input untuk Naive Bayes . . . . .	44
3.9 Pembagian data pelatihan dan pengujian . . . . .	44
3.10 Pemanggilan method train untuk pelatihan data . . . . .	45
3.11 Menyimpan model hasil pelatihan . . . . .	45
3.12 Hasil Naive Bayes dikeluarkan pada console . . . . .	45
3.13 Hasil Naive Bayes dikeluarkan pada console . . . . .	46
3.14 Membuat data input untuk eksperimen PCA . . . . .	46
3.15 Menjalankan komputasi perhitungan PC . . . . .	46
3.16 Proyeksi data dengan nilai PC . . . . .	47
3.17 Proyeksi data dengan nilai PC . . . . .	47
3.18 Komputasi PC menggunakan kelas PCA . . . . .	47
3.19 Proyeksi data dengan nilai PC yang dihasilkan kelas PCA . . . . .	47
3.20 Menampilkan hasil RowMatrix ke console . . . . .	47
3.21 Menampilkan hasil kelas PCA ke console . . . . .	47
3.22 Hasil keluaran eksperimen PCA pada console . . . . .	48
3.23 Memanggil method perhitungan TF . . . . .	48
3.24 Memanggil method perhitungan IDF . . . . .	48
3.25 Pemanggilan IDF dengan batas minimum document . . . . .	48

3.26 Menampilkan hasil ke console . . . . .	49
3.27 Hasil output pada console . . . . .	49
3.28 Hasil output dengan minimum document pada console . . . . .	49
3.29 Mengkonversi data menjadi Rating . . . . .	50
3.30 Pelatihan data menggunakan ALS . . . . .	50
3.31 Prediksi data menggunakan model . . . . .	50
3.32 Menghitung rata-rata kesalahan prediksi . . . . .	51
3.33 Menyimpan model ALS . . . . .	51
3.34 Hasil output prediksi . . . . .	51
3.35 Membuat RDD dengan menghubungkan HDFS . . . . .	52
3.36 Menyimpan model hasil pelatihan dan memuat model . . . . .	52
3.37 Hasil output program eksperimen KMeans . . . . .	53
4.1 Method Run() KMeans.scala . . . . .	57
4.2 Konversi data menjadi Norm Vektor pada method run() . . . . .	58
4.3 Inisialisasi centroid awal pada kelas KMeans . . . . .	59
4.4 Kode program implementasi algoritma pada method runAlgorithm() . . . . .	60
4.5 Diagram flow iterasi K-Means pada MLLib . . . . .	61
4.6 Fungsi di Method mapPartitions() pada KMeans . . . . .	62
4.7 Fungsi di Method reduceByKey() pada KMeans.scala . . . . .	63
4.8 Fungsi di Method reduceByKey() pada KMeans.scala . . . . .	63
4.9 Modifikasi untuk mendapatkan nilai minimum, maksimum, dan standar deviasi dalam iterasi objek(point) . . . . .	65
4.10 Nilai variabel modifikasi ditambahkan dalam kembalian . . . . .	66
4.11 Bagian kode untuk menghitung nilai akhir rata-rata, minimum, maksimum, dan standar deviasi . . . . .	66
4.12 Method untuk memeriksa status converged . . . . .	67
4.13 Pola disimpan ke variabel global . . . . .	67
4.14 Implementasi perhitungan formula Standar Deviasi . . . . .	68
4.15 Method untuk menyimpan kedalam file (.txt) . . . . .	68
5.1 Diagram <i>use case</i> perangkat lunak demo . . . . .	70
5.2 Diagram kelas perangkat lunak demo . . . . .	74
5.3 Tampilan pengolahan data menggunakan algoritma Naive Bayes . . . . .	80
5.4 Tampilan pengolahan data menggunakan algoritma K-Means . . . . .	81
5.5 Tampilan pengolahan data menggunakan metode Statistik . . . . .	82
5.6 Tampilan pengolahan data menggunakan algoritma PCA . . . . .	83
5.7 Tampilan pengolahan data menggunakan algoritma ALS . . . . .	84
5.8 Tampilan pengolahan data menggunakan algoritma TF-IDF . . . . .	85
5.9 Tampilan pengolahan data menggunakan algoritma Naive Bayes . . . . .	86
5.10 Tampilan pengolahan data menggunakan algoritma K-Means . . . . .	87
5.11 Tampilan pengolahan data menggunakan metode Statistik . . . . .	87
5.12 Tampilan pengolahan data menggunakan algoritma PCA . . . . .	88
5.13 Tampilan pengolahan data menggunakan algoritma ALS . . . . .	88
5.14 Tampilan pengolahan data menggunakan algoritma TF-IDF . . . . .	89
5.15 Data masukkan untuk fitur naive bayes . . . . .	96
5.16 Data masukkan untuk fitur k-means . . . . .	97
5.17 Data masukkan untuk fitur Statistik dan PCA . . . . .	97
5.18 Data masukkan untuk fitur ALS . . . . .	97
5.19 Data masukkan untuk fitur TF-IDF . . . . .	98
5.20 Hasil pengujian fungsional naive bayes model . . . . .	98
5.21 Hasil pengujian fungsional prediksi dengan model naive bayes . . . . .	98

5.22 Hasil pengujian fungsional K-Means . . . . .	99
5.23 Hasil pengujian fungsional statistik . . . . .	99
5.24 Hasil pengujian fungsional PCA . . . . .	100
5.25 Hasil pengujian fungsional model ALS . . . . .	100
5.26 Hasil pengujian fungsional prediksi dengan model ALS . . . . .	100
5.27 Hasil pengujian fungsional TF-IDF . . . . .	101
5.28 Arsitektur lingkungan cluster hadoop untuk eksperimen . . . . .	101
5.29 Grafik perhitungan waktu eksekusi pembuatan model Naive Bayes . . . . .	103
5.30 Grafik perhitungan waktu eksekusi prediksi data dengan model Naive Bayes . . . . .	103
5.31 Grafik perhitungan waktu eksekusi eksperimen K-Means . . . . .	104
5.32 Grafik perhitungan waktu eksekusi eksperimen Statistik . . . . .	105
5.33 Grafik perhitungan waktu eksekusi eksperimen PCA . . . . .	106
5.34 Grafik perhitungan waktu eksekusi pembuatan model Alternating Square Least . . . . .	107
5.35 Grafik perhitungan waktu eksekusi prediksi data dengan model Alternating Square Least . . . . .	107
5.36 Grafik perhitungan waktu eksekusi eksperimen TF-IDF . . . . .	108



## **DAFTAR TABEL**

2.1 Tipe data pada bahasa pemrograman Scala [1] . . . . .	32
5.1 Hasil Pengujian Fungsional Perangkat Lunak . . . . .	96



1

## BAB 1

2

### PENDAHULUAN

#### 3 1.1 Latar Belakang

4 Seiring dalam pemanfaatan teknologi informasi yang makin meluas, data dengan cepat terkumpul,  
5 sehingga dalam waktu yang singkat dapat mencapai ratusan gigabyte. Sebagai contoh, beberapa  
6 data yang dapat terkumpul dengan cepat adalah data log web(ditekan), data teks(email, facebook,  
7 dll), data waktu dan lokasi, data sensor, dan masih banyak lagi. Data-data tersebut dinamakan  
8 "Big Data". Big Data menurut Gartner(2012) didefinisikan sebagai data yang memiliki ukuran  
9 (volume), kecepatan (velocity), dan/atau ragam (variety) yang ekstrim, yang menuntut pemrosesan  
10 informasi cepat dan inovatif untuk mendukung pengambilan keputusan dan otomatisasi proses.  
11 Terdapat 3 jenis Big Data yaitu structured data, unstructured data, semi-structured data.

12

13 Big data dapat dianalisis untuk mendapatkan informasi atau pengetahuan yang berharga. Bebe-  
14 rapa teknik analisis Big Data diantaranya adalah peringkasan data yang berbasis Association Rule  
15 Learning, Classification tree analysis, Genetic algorithms, Machine learning, Regression analysis,  
16 Sentimental Analysis, dan Social network analysis.

17

18 Hadoop merupakan platform yang dapat menyediakan penyimpanan dan kemampuan komputasi  
19 terdistribusi. Dengan Hadoop, Pengguna dapat menyimpan Big Data dan melakukan komputasi  
20 analisis terhadap Big Data. Komponen utama dari hadoop yaitu HDFS dan MapReduce. HDFS  
21 merupakan suatu *file system* yang bertujuan untuk menyimpan data yang berukuran besar secara  
22 terdistribusi. Sedangkan MapReduce yang melakukan data processing pada Hadoop.

23

24 Apache Spark adalah sebuah platfrom cluster computing yang dapat dimanfaatkan untuk meng-  
25 olah Big Data dengan cepat. Spark merupakan pengembangan model MapReduce untuk melakukan  
26 komputasi agar lebih cepat dan efisien. Apache Spark dapat berjalan diatas infrastruktur Hadoop  
27 seperti HDFS (Hadoop Distributed File System) untuk meningkatkan fungsionalitas. Apache Spark  
28 dibuat menggunakan bahasa pemrograman Scala dan berjalan diatas Java Virtual Machine (JVM).  
29 Beberapa perpustakaan di dalam Spark adalah Spark SQL, Spark Streaming, MLlib, dan GraphX.

30

31 MLlib merupakan library yang mengimplementasikan teknik analisis Machine Learning pada Big  
32 Data. Beberapa jenis algoritma Machine Learning yang disediakan diantaranya adalah classification,  
33 regression, clustering, collaborative filtering, dll. Semua algoritma tersebut didesain untuk dapat  
34 dioperasikan diatas cluster hadoop. Pada algoritma clustering MLlib didapatkan bahwa K-Means

1 perlu dimodifikasi agar dapat menghasilkan pola.

2  
3 Pada skripsi ini akan dibuat sebuah perangkat lunak demo untuk menagkses fungsi-fungsi  
4 pada MLlib. agar MLlib lebih mudah diakses dan digunakan. Selain itu, penulis melakukan modi-  
5 fikasi pada algoritma K-Means MLlib agar dapat menghasilkan pola yang bermanfaat bagi pengguna.  
6

## 7 **1.2 Rumusan Masalah**

8 Rumusan masalah pada penelitian ilmiah ini adalah :

- 9 1. Bagaimana konsep Apache Spark dan Hadoop?
- 10 2. Bagaimana cara kerja fungsi-fungsi *library* MLlib pada Apache Spark?
- 11 3. Bagaimana modifikasi dari algoritma K-Means pararel pada Spark MLlib?
- 12 4. Bagaimana performa Apache Spark MLlib dengan menggunakan *Big Data*?
- 13 5. Bagaimana pengembangan perangkan lunak demo MLlib?

## 14 **1.3 Tujuan**

15 Berdasarkan rumusan masalah tersebut, tujuan dari penelitian ini adalah sebagai berikut:

- 16 1. Memahami konsep framework Apache Spark dan Hadoop
- 17 2. Memahami cara kerja *library* MLlib untuk analisis *Big Data* dan melakukan uji coba peman-  
18 faatan/pemanggilan fungsi-fungsi MLlib
- 19 3. Merancang modifikasi yang dilakukan pada source code K-Means MLlib
- 20 4. Mengembangkan perangkat lunak untuk mempraolah data yang memanggil fungsi-fungsi pada  
21 MLlib untuk analisis data dan menampilkan hasil analisis data tersebut.

## 22 **1.4 Batasan Masalah**

23 Batasan masalah pada Skripsi ini adalah:

- 24 1. Studi literatur konsep Hadoop hanya dilakukan pada dasar Hadoop dan konsep file system  
25 Hadoop yaitu HDFS.
- 26 2. Studi literatur Apache Spark hanya dilakukan pada komponen Apache Spark yaitu library  
27 MLlib.

## **1.5 Metodologi**

- 2 Penyusunan Skripsi ini menggunakan metodologi sebagai berikut:
- 3    1. Melakukan studi literatur tentang konsep Apache Spark.
- 4    2. Melakukan studi literatur penggunaan Hadoop terutama HDFS (Hadoop Distributed File System).
- 5    3. Melakukan instalasi dan konfigurasi Spark MLlib pada *cluster* Hadoop.
- 6    4. Melakukan studi literatur bahasa pemrograman Scala.
- 7    5. Melakukan studi literatur tentang algoritma-algoritma pada MLlib dan konsep MLlib.
- 8    6. Melakukan eksperimen cara pemanggilan fungsi-fungsi pada library MLlib.
- 9    7. Melakukan modifikasi *source code* algoritma K-Means pada *library* MLlib.
- 10    8. Mencari dan mengumpulkan data studi kasus.
- 11    9. Merancang dan mengimplementasikan perangkat lunak demo.
- 12    10. Melakukan pengujian dan eksperimen untuk menguji performance perangkat lunak untuk menganalisis big data.
- 13    11. Menulis dokumen skripsi.

## **1.6 Sistematika Pembahasan**

- 17 Bab 1 Pendahuluan  
18 Bab 1 berisi latar belakang, rumusan masalah, tujuan, batasan masalah, metodologi penelitian, dan sistematika pembahasan.
- 20  
21 Bab 2 Dasar Teori  
22 Bab 2 berisi teori-teori mengenai sistem terdistribusi, konsep dan cara kerja apache spark dan hadoop  
23 HDFS, perpustakaan Spark MLlib, Dasar-dasar sintaks penulisan dalam bahasa pemrograman scala.
- 24  
25 Bab 3 Studi dan Eksplorasi Hadoop  
26 Bab 3 berisi eksperimen-eksperimen dimulai dari langkah-langkah instalasi Apache Spark, eksperimen dengan Spark Shell dan eksperimen pemanggilan fungsi Mllib.
- 28  
29 Bab 4 Analisis  
30 Bab 4 berisi penjabaran kebutuhan pengembangan K-Means MLlib, analisis *source code* K-Means MLlib, dan pengembangan yang dilakukan pada K-Means MLlib sesuai kebutuhan.
- 32  
33 Bab 5 Perancangan, Implementasi, dan Pengujian  
34 Bab 5 berisi lingkungan implementasi, implementasi antarmuka perangkat lunak demo, pengujian

<sup>1</sup> perangkat lunak demo, dan eksperimen uji performa perangkat lunak demo.

<sup>2</sup>

<sup>3</sup> Bab 6 Kesimpulan dan Saran

<sup>4</sup> Bab 6 berisi kesimpulan dan saran.

1

## BAB 2

2

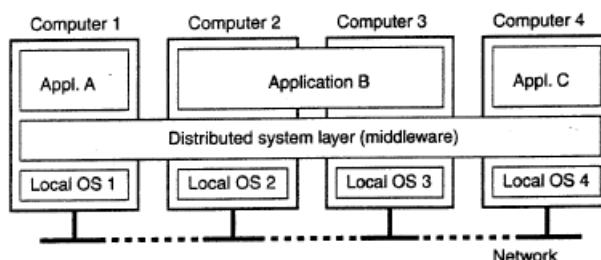
### LANDASAN TEORI

#### 3 2.1 Sistem Terdistribusi

##### 4 2.1.1 Pengertian Sistem Terdistribusi

5 Sistem terdistribusi atau sistem tersebar dapat didefinisikan sebagai kumpulan dari komputer-  
6 komputer yang independen namun tampak sebagai sistem koheren tunggal [2]. Sebuah sistem  
7 terdistribusi dapat terdiri dari beberapa komponen (contohnya komputer) yang otonom. Komponen-  
8 komponen tersebut harus berkolaborasi agar tampak oleh user sebagai sebuah system tunggal.  
9 Kolaborasi ini dilakukan dengan menghubungkan komponen-komponen tersebut melalui sebuah  
10 jaringan komputer. Sistem terdistribusi memiliki karakteristik utama yaitu komunikasi antara  
11 komponen-komponen tersebut tidak tampak oleh pengguna. Untuk dapat mencapai sebuah sistem  
12 terdistribusi, terdapat layer yang mengorganisir komponen-komponen sistem yang disebut sebagai  
13 middleware. Pada gambar 2.1 dapat dilihat bahwa layer sistem terdistribusi secara logis berada  
14 diantara layer tingkat atas (user dan aplikasi) dan Sistem Operasi.

15



Gambar 2.1: Sistem terdistribusi yang diorganisir middleware

16

Tujuan yang ingin dicapai pada sistem terdistribusi yaitu:

17  
18

- *Making Resources Accessible* : memudahkan pengguna (dan aplikasi) untuk dapat mengakses resource secara remote dan membagikan resource secara efisien dan terkontrol.

19  
20

- *Distribution Transparency* : menyembunyikan fakta bahwa proses dan *resource* secara fisik terdistribusi pada beberapa komputer.

21  
22

- *Openness* : Sistem terdistribusi yang terbuka artinya sistem tersebut menawarkan *service* menurut aturan yang standar. Contoh pada jaringan komputer terdapat aturan standar yang

1 mengatur format dan konten pesan yang terkirim dan yang diterima.

- 2 • *Scalability* : Sistem distribusi harus bersifat scalability artinya sistem dapat terukur melalui 2  
3 dimensi yang utama yaitu berdasarkan ukuran dan letak geografis.

4 **2.1.2 Jenis Sistem Terdistribusi**

5 Pada sistem terdistribusi terdapat beberapa jenis sistem terdistribusi. Diantaranya adalah:

6

- 7 a. Distributed Computing System  
8 b. Distributed information System  
9 c. Distributed embedded system

10 *Distributed Computing System* merupakan sistem terdistribusi yang digunakan untuk komputasi  
11 berkinerja tinggi. Terdapat kelompok pada *Distributed Computing System* yaitu *Cluster Computing*  
12 *System* dan *Grid Computing Sistem*. Sebuah *cluster computing*(Kluster komputasi) terdiri dari  
13 sekumpulan PC yang serupa dan terhubung oleh *local-area network* berkecepatan tinggi. Sedangkan  
14 *grid computing* terdiri dari sistem terdistribusi yang dibangun sebagai sebuah persekutuan dari  
15 sistem komputer. Perbedaan dari keduanya adalah *cluster computing* bersifat homogen sedangkan  
16 *grids computing* bersifat heterogen.

17

18 Seiring dengan perkembangan volume data dan informasi setiap tahunnya. Dibutuhkan suatu  
19 teknik untuk menyediakan penyimpanan dan sekaligus memproses data tersebut. Salah satunya  
20 solusinya adalah dengan menggunakan *distributed computing*. Pada bagian berikut ini akan dibahas  
21 beberapa *framework* yang mengimplementasikan *distributed computing*.

22

23 **2.2 Hadoop**

24 Pada era perkembangan teknologi informasi dan meluasnya pemanfaatan teknologi informasi saat  
25 ini, terdapat berbagai data yang semakin cepat terkumpul. Sehingga dalam waktu yang singkat  
26 dapat mencapai ratusan gigabyte. Sebagai contoh, beberapa data yang dapat terkumpul dengan  
27 cepat adalah data web, data teks(email, facebook, dll), data waktu dan lokasi, data sensor, data  
28 transaksi belanja online, dan masih banyak lagi. Data-data tersebut bisa saja disimpan dalam server  
29 atau data center. Namun dengan semakin banyak data yang tersimpan semakin banyak biaya yang  
30 dikeluarkan untuk memperbesar kapasitas penyimpanan. Terkumpulnya data yang semakin banyak  
31 memperburuk keadaan karena tidak memberi makna yang berarti bagi pemilik datanya. Dengan  
32 demikian perlu dilakukan pemrosesan pada data-data tersebut. Jumlah data yang besar artinya  
33 membutuhkan kemampuan komputasi yang besar pula. Distributed Computation Sistem dapat  
34 memberikan solusi untuk permasalahan ini.

35

36 MapReduce pertama kali diperkenalkan oleh Google. MapReduce merupakan model pemro-  
37 graman yang digunakan untuk melakukan komputasi data berukuran sangat besar. MapReduce

1 merupakan abstraksi sebuah komputasi paralel dengan data yang terdistribusi. MapReduce menjadi  
2 solusi dalam membaca dan memproses data dengan cepat. Salah satu framework Open Source yang  
3 menerapkan teknik ini adalah framework Hadoop.

4

5 Hadoop adalah sebuah *framework open source* untuk menyimpan dan menjalankan aplikasi  
6 terdistribusi yang memproses kumpulan data yang berukuran besar [3]. Secara sederhana, Hadoop  
7 merupakan sebuah platform yang menyediakan penyimpanan terdistribusi dan kemampuan kompu-  
8 tasi terdistribusi [4]. Distributed computing memiliki makna yang luas dan dapat sangat bervariasi.  
9 Namun, hadoop memiliki karakteristik yang berbeda dari distributed computing lainnya yaitu:

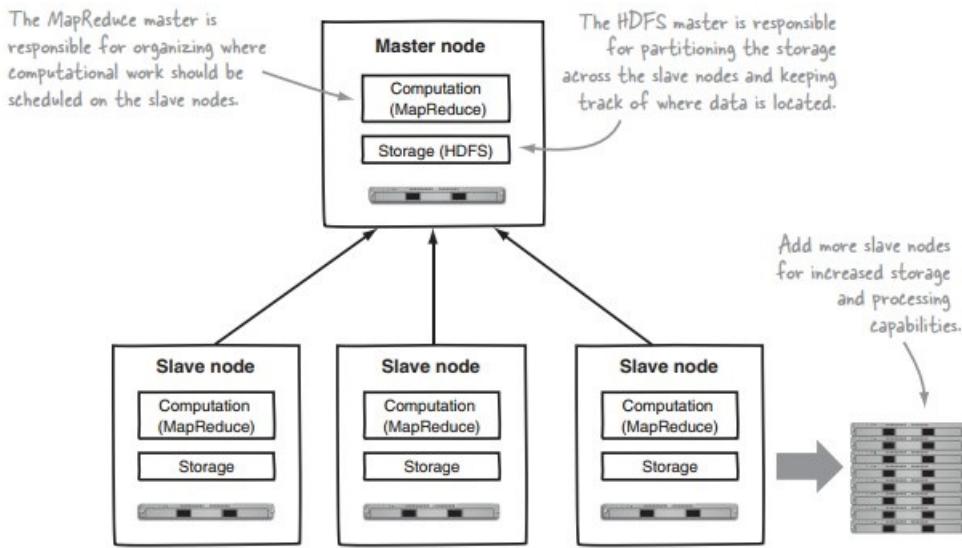
10

- 11 • Accessible. Hadoop dapat berjalan pada cluster berukuran besar yang komponennya berupa  
12     "commodity machine". *Commodity machine* adalah mesin atau komputer yang beredar  
13     dipasaran seperti *personal computer*(PC).
- 14 • Robust. Hadoop dapat menangani kesalahan yang sering terjadi pada hardware *commodity*  
15     *machine*.
- 16 • Scalable. Untuk menangani data yang lebih besar, hadoop dapat menambahkan node yang  
17     lebih banyak ke dalam cluster hadoop.
- 18 • Simple. Hadoop memberikan kemudahan untuk pengguna dalam menulis kode paralel yang  
19     efisien.

20 Terdapat modul-modul yang membentuk hadoop sebagai *framework* yang dapat memproses  
21 sekaligus menyimpan data yang sangat besar. Modul-modul itu diantaranya adalah :

22

- 23 • Hadoop Distributed File System (HDFS), Sebuah *file-system* terdistribusi yang menyimpan  
24     data pada *commodity machine* dan juga menyediakan *bandwidth* yang sangat tinggi ke seluruh  
25     cluster.
- 26 • Hadoop MapReduce, Model pemrograman untuk pemrosesan data berskala besar dan terdis-  
27     tribusi.
- 28 • Hadoop YARN, Sebuah platform *resource-management* yang bertanggung jawab untuk meng-  
29     elola sumber daya komputasi dalam kelompok dan menggunakan untuk penjadwalan  
30     aplikasi pengguna.
- 31 • Hadoop Common, Berisi *libraries* dan *utilities* yang dibutuhkan oleh modul Hadoop lainnya.



Gambar 2.2: Arsitektur Hadoop

1 Hadoop memiliki arsitektur *distributed master-slave*. Artinya struktur dari hadoop terdiri  
 2 dari node Master dan node slave. Node merupakan istilah untuk sebuah mesin dalam cluster.  
 3 Masing-masing node memiliki komponen yang paling utama yaitu HDFS (Hadoop Distributed File  
 4 System) dan MapReduce [4]. HDFS merupakan *filesystem* pada hadoop. Sedangkan MapReduce  
 5 merupakan model pemrograman untuk melakukan komputasi pada data. Dengan demikian hadoop  
 6 akan menyimpan data dengan memasukkan data ke dalam HDFS. Ketika data akan diproses dengan  
 7 melakukan komputasi maka MapReduce yang akan menangani hal tersebut.

8

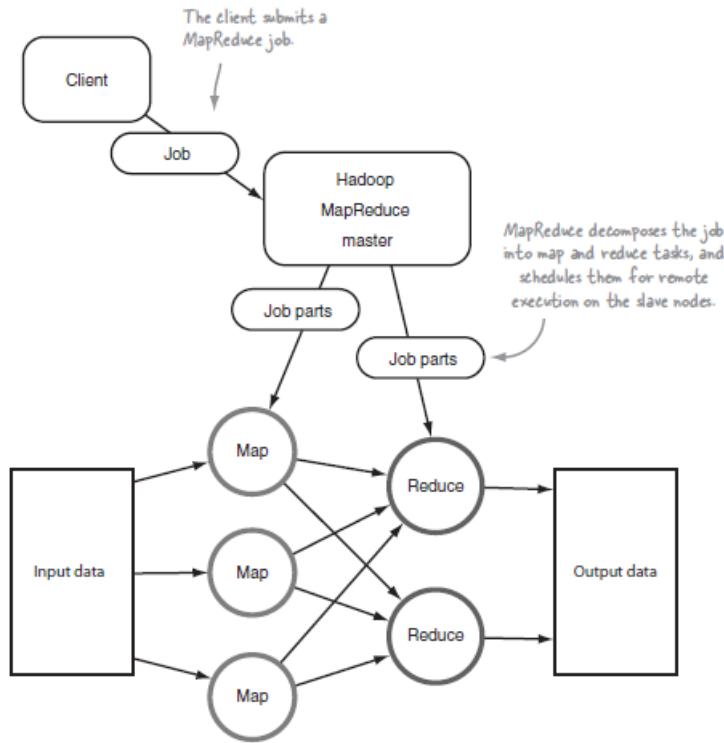
### 9 2.2.1 MapReduce

10 MapReduce merupakan sebuah programming model dan yang terkait implementasi untuk mempro-  
 11 ses dan menghasilkan sebuah dataset yang berukuran besar [5]. MapReduce awalnya diperkenalkan  
 12 oleh Google dalam upaya melakukan simplifikasi pemrosesan data pada *cluster* berukuran besar.  
 13 Pada programming model MapReduce, komputasinya membutuhkan sebuah set input berupa  
 14 pasangan *key/value* dan menghasilkan sebuah set output pasangan *key/value*. Di dalam model  
 15 MapReduce terdapat sebuah pemrosesan data primitif seperti *mappers* dan *reducers*. Mengurai  
 16 aplikasi pemrosesan data ke dalam *mappers* dan *reducer* sebenarnya adalah hal yang umum dan  
 17 wajar. Namun berbeda halnya ketika melakukan *scaling* aplikasi agar dapat berjalan pada ratusan  
 18 atau ribuan mesin pada sebuah *cluster* akan membutuhkan perubahan konfigurasi.

19

20 Pada Hadoop, model ini diterapkan sebagai model pemrosesan data. Keunggulannya adalah  
 21 memudahkan penskalaan(*scaling*) pada pemrosesan data di node komputasi yang sangat banyak.  
 22 MapReduce menyederhanakan pemrosesan paralel dengan mengabstraksikan kerumitan ketika  
 23 berhadapan dengan sistem terdistribusi seperti komputasi paralel, distribusi pekerjaan, dan mena-  
 24 ngani hardware dan software yang tidak bisa diandalkan. Dengan adanya abstraksi ini, pengguna  
 25 (*programmer*) tidak dipusingkan dengan kerumitan pada sistem terdistribusi.

1



Gambar 2.3: Arsitektur HDFS : Client berkomunikasi dengan NameNode(Master) dan DataNode(slave)

2 MapReduce memiliki cara kerja (seperti yang dapat dilihat pada Gambar 2.3) yaitu ketika  
 3 menerima pekerjaan dari *client* yaitu mendekomposisi pekerjaan ke dalam *worker* map dan reduce  
 4 berukuran kecil yang bekerja secara paralel. Map dan reduce pada MapReduce Hadoop tidak memi-  
 5 liki interdependensi ketika mengeksekusi secara pararel. *Programmer* hanya perlu mendefinisikan  
 6 fungsi map dan reduce. Fungsi map akan mengeluarkan output pasangan *key/value* yang kemudian  
 7 diproses oleh fungsi reduce untuk menghasilkan output akhir.

8

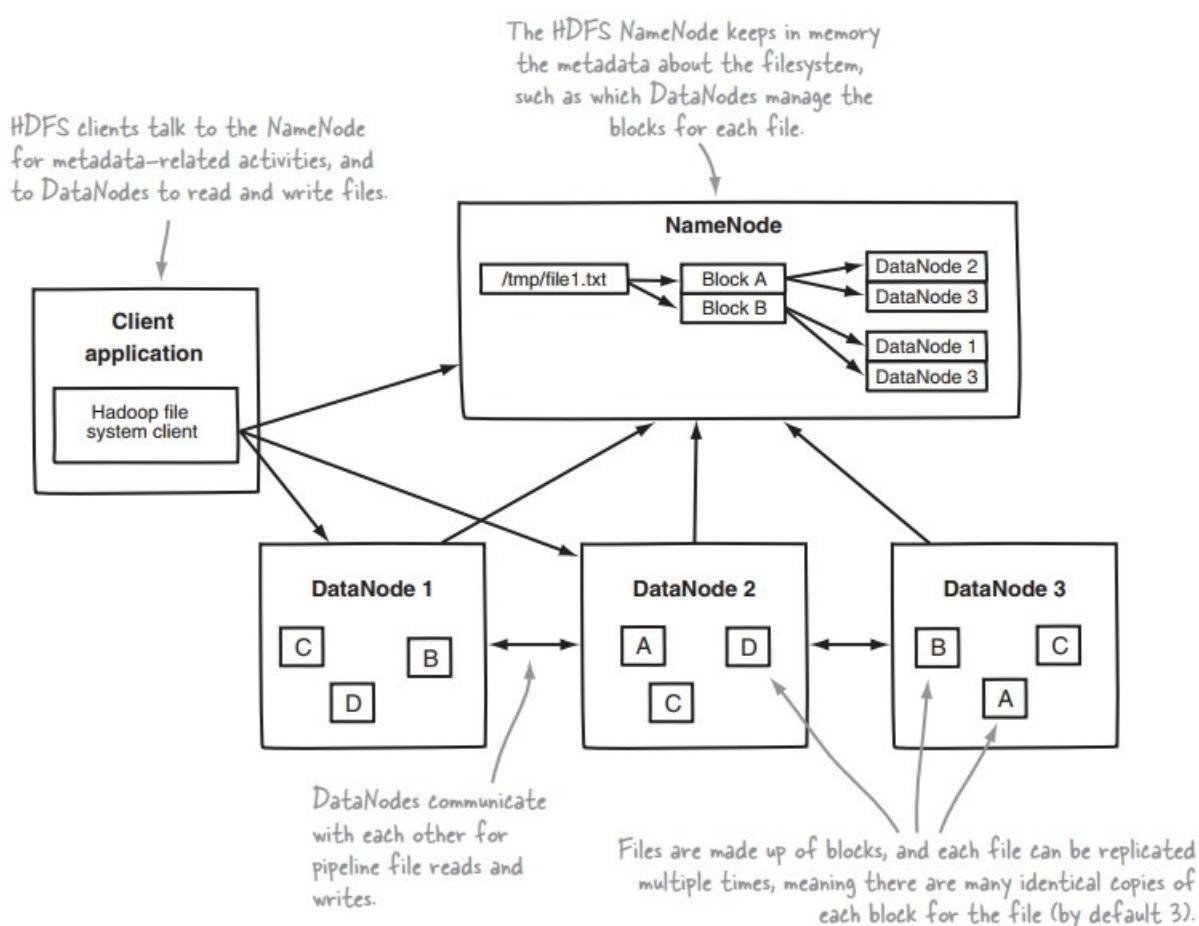
### 9 2.2.2 Hadoop Distributed File System (HDFS)

10 Hadoop Distributed File System(HDFS) adalah sebuah *file system* terdistribusi yang menyimpan  
 11 data pada mesin komoditas. HDFS merupakan komponen penyimpanan pada Hadoop. HDFS  
 12 telah optimasi sehingga memiliki *throughput* data yang besar dan dapat bekerja dengan baik ketika  
 13 membaca dan menulis file-file berukuran besar(gigabytes dan yang lebih besar lagi). Hal ini karena  
 14 HDFS memiliki ukuran block yang besar daripada file-system yang umum digunakan [3].

15

16 HDFS memiliki sifat scalability dan availability yang sangat baik. HDFS dapat bertahan jika ada  
 17 kerusakan perangkat lunak dan perangkat keras. Hal ini karena HDFS secara otomatis mereplikasi  
 18 ulang blok data pada node yang mengalami *hardware failure*.

19



Gambar 2.4: Arsitektur HDFS : Client berkomunikasi dengan NameNode(Master) dan DataNode(slave)

1        HDFS memiliki daemon-daemon(proses-proses) yang berjalan pada masing-masing mesin di  
 2        cluster yang menjalankan hadoop. Terdapat dua daemon yang dimiliki HDFS dalam sebuah cluster  
 3        yaitu NameNode dan DataNode [3]. Perbedaan dari kedua daemon tersebut adalah :

4

- 5        • NameNode : NameNode adalah master dari HDFS yang memerintahkan *slave* (DataNode)  
 6        untuk menjalankan pekerjaan *low-level* input/output(I/O). NameNode diibaratkan seperti  
 7        pemegang buku(bookkeeper). Tugasnya adalah mencatat proses file dipecah menjadi blok-  
 8        blok file. Selain itu, NameNode mencatat lokasi node yang menyimpan blok-blok tersebut.  
 9        Dan dapat memantau seberapa baik kondisi distributed filesystem. NameNode tidak akan  
 10      menyimpan menyimpan data dari pengguna. Hal ini bertujuan mengurangi beban pekerjaan  
 11      pada mesin yang memiliki NameNode. Dengan demikian Node yang menjadi NameNode tidak  
 12      dapat sekaligus menjadi DataNode. Dari sisi negatifnya, NameNode menjadi sebuah 'single  
 13      point of failure' dari cluster hadoop. Sehingga, jika NameNode mengalami kegagalan maka  
 14      Hadoop tidak dapat berjalan lagi.
- 15        • DataNode : Setiap node slave pada cluster akan memiliki daemon DataNode untuk melakukan

1 pekerjaan dari filesystem terdistribusi. Misalnya membaca dan menulis blok-blok HDFS  
2 menjadi file-file yang aktual pada filesystem lokal. Ketika akan membaca atau menulis suatu  
3 file HDFS, file tersebut akan dipecah menjadi blok dan NameNode akan memberi tahu  
4 kepada client pengguna dimana setiap blok DataNode berada. Client pengguna berkomunikasi  
5 langsung dengan daemon DataNode untuk memproses file lokal yang sesuai dengan blok yang  
6 dituju. Disamping itu, DataNode dapat berkomunikasi dengan DataNodes lainnya dengan  
7 tujuan mereplikasi blok-blok data.

8 Kelemahan NameNode diatas HDFS dengan cara membuat Secondary NameNode (SNN).  
9 Secondary NameNode adalah asisten untuk memantau keadaan cluster HDFS. Sama halnya dengan  
10 NameNode, Setiap cluster hanya memiliki satu Secondary NameNode. Secondary NameNode berbe-  
11 da dengan NameNode. Hal ini dikarenakan Secondary NameNode berkomunikasi dengan NameNode  
12 untuk mengambil snapshots dari metadata HDFS pada interval yang ditentukan oleh konfigurasi  
13 cluster. Fungsi dari snapshot ini adalah untuk membantu meminimalkan *downtime* dan kehilangan  
14 data. Jika kegagalan NameNode terjadi, maka dibutuhkan tindakan dari pengguna(manusia) untuk  
15 mengkonfigurasi ulang cluster agar dapat menggunakan SNN sebagai Primary NameNode [3].

16  
17 HDFS dirancang untuk menyimpan file data yang besar. HDFS telah menunjukkan skalabilitas  
18 produksi hingga 200 PB penyimpanan dan satu cluster dari 4500 server, mendukung hampir satu  
19 miliar file dan blok. HDFS adalah sistem penyimpanan terdistribusi yang *fault-tolerant* yang  
20 dikoordinasikan oleh YARN. YARN (Yet Another Resource Negotiator) adalah salah satu komponen  
21 pada hadoop yang berfungsi untuk manajemen *resource* pada cluster hadoop.

22  
23 Semua detail yang dilakukan HDFS pada cluster diabstraksi sehingga seolah-olah pengguna  
24 hanya menangani suatu file saja. Kalau pada Unix filesystem pada umumnya memiliki standar Unix  
25 file tools, HDFS tidak memiliki standar itu. Misalnya perintah pada Unix filesystem yaitu ls dan  
26 cp tidak akan dapat dijalankan pada HDFS. Begitu pula perintah-perintah standar Unix lainnya.  
27 Hadoop memiliki sebuah set command line utilities yang fungsinya sama seperti pada command file  
28 Linux. Hadoop memiliki bentuk file commands seperti berikut

29 `Hadoop fs -cmd <args>`

30 Dimana cmd merupakan file command spesifik dan `<args>` adalah macam-macam argumen.  
31 Perintah cmd umumnya dinamai seperti yang digunakan pada Unix. Contohnya, perintah listing  
32 file `ls` dan membuat directory `mkdir`. HDFS memiliki sebuah direktori default yaitu `/user/$USER`  
33 dibawah direktori `root`, dimana `$USER` adalah nama pengguna yang login. Direktori ini tidak  
34 terbentuk otomatis, maka perlu dibuat sendiri dengan perintah `mkdir`. Berikut adalah contoh-contoh  
35 perintah dasar dari hdfs.

36  
37 Membuat direktori:

38 `hadoop fs -mkdir /user/kresna`

39 Parent directory akan otomatis dibuat jika belum pernah dibuat atau tidak ada. Listing file  
40 atau direktori:

1    `hadoop fs -ls /`

2       Listing file akan menghasilkan response seperti berikut :

3    `Found 1 items`

4    `drwxr-xr-x - kresna supergroup 0 2018-01-14 10:00 /user`

5       Sedangkan untuk listing seluruh *subdirectory* gunakan perintah

6    `hadoop fs -lsr /`

7       Untuk meletakkan sebuah file kedalam direktori, misal sebuah file diberi nama contoh.txt pada  
8       filesystem lokal. Maka dapat digunakan perintah put pada Hadoop untuk meng-*copy* file dari  
9       filesystem lokal ke filesystem HDFS.

10      `hadoop fs -put contoh.txt .`

11       Tanda titik(.) pada argumen terakhir memiliki arti bahwa file akan diletakkan pada default  
12       directory. Atau dapat dikatakan perintah diatas sama seperti perintah berikut

13      `hadoop fs -put contoh.txt /user/kresna`

14       Berikut adalah tampilan output listing file ketika ada suatu file didalamnya

15      \$ `hadoop fs -ls`

16      `Found 1 items`

17      `-rw-r--r-- 1 kresna supergroup 264 2018-01-14 11:00`

18      `âđe/user/kresna/contoh.txt`

19       Properti dari tampilan output seperti *permission*, *owner*, *group*, ukuran file, dan *last modification date*, sama seperti pada Linux atau Unix. Perbedaan utama hanya pada kolom yang bertuliskan  
20       '1' yang menandakan jumlah replikasi pada file tersebut. Pada penggunaan umumnya replikasi  
21       biasanya bernilai 3. Sedangkan untuk mengembalikan file dari HDFS ke *filesystem* lokal digunakan  
22       perintah  
23       perintah

24      `hadoop fs -get contoh.txt .`

25       Untuk menghapus file pada Hadoop command digunakan perintah

26      `hadoop fs -rm contoh.txt`

27       Beberapa daftar hadoop command file lainnya dapat dicari melalui HDFS API atau menggunakan  
28       perintah

29      `Hadoop fs`

30       Untuk melihat deskripsi singkat dari salah satu command gunakan perintah berikut

31      `hadoop fs -help ls`

### 1    2.2.3 Blok HDFS

2    File dalam filesystem hdfs akan dibagi menjadi satu atau lebih segmen dan/atau disimpan dalam  
3    node data individual. Segmen file ini disebut blok. Dengan kata lain, jumlah minimum data yang  
4    dapat dibaca atau dibaca oleh HDFS disebut Blok. Ukuran blok default adalah 64MB, namun dapat  
5    ditingkatkan sesuai kebutuhan untuk mengubah konfigurasi HDFS. kebanyakan sistem berjalan  
6    dengan ukuran blok 128 megabyte atau lebih besar. Setiap blok file direplikasi secara independen  
7    di beberapa DataNodes.

8

9       Blok disimpan pada sistem file lokal pada DataNode. Namenode secara aktif memantau jumlah  
10      replika blok. Ketika replika blok hilang karena kegagalan DataNode atau kegagalan disk, NameNode  
11      menciptakan replika blok lainnya. NameNode memelihara pohon *namespace* dan pemetaan blok ke  
12      DataNode, serta memegang seluruh citra *namespace* di RAM. NameNode tidak secara langsung  
13      mengirim permintaan ke DataNode. Namenode mengirimkan instruksi ke DataNode dengan mem-  
14      balas sebuah 'hearbeats' yang dikirim oleh DataNode tersebut.

15

## 16    2.3 Apache Spark

17      Berbagai perusahaan telah menggunakan Hadoop untuk menganalisis data set mereka. Hal ini tentu  
18      saja karena programming model MapReduce Hadoop yang memberikan solusi untuk komputasi  
19      yang scalable, *fault-tolerant*, dan efektifitas biaya yang baik. Seiring dengan semakin banyaknya  
20      teknik analisis data yang menggunakan MapReduce. Mempertahankan kecepatan dalam melakukan  
21      komputasi pada data set yang besar menjadi hal yang penting. Kecepatan dalam melakukan  
22      komputasi yang dimaksud adalah dalam hal waktu tunggu diantara banyak query dan lamanya  
23      waktu dalam menjalankan program.

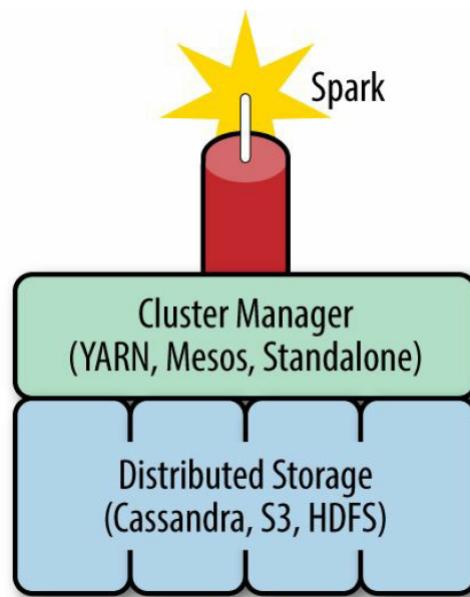
24

25       Apache Spark merupakan sebuah framework untuk melakukan analisis data pada sistem kom-  
26      putasi terdistribusi. Spark menyediakan fitur komputasi di memori sehingga dapat meningkatkan  
27      kecepatan dalam hal pemrosesan data dengan mapreduce. Spark dapat berjalan diatas Hadoop  
28      cluster dan dapat mengakses data yang ada pada Hadoop filesystem (HDFS). Hadoop hanya salah  
29      satu cara dalam mengimplementasikan Spark. Spark dapat digunakan pada sistem penyimpanan  
30      terdistribusi lainnya (Contoh: Cassandra atau S3) dan sebuah cluster manager (Standalone Cluster  
31      Manager, Apache Mesos, dan Hadoop YARN (Lihat Gambar 2.5)).

32

33       Spark pada awalnya merupakan sub projek dari Hadoop yang dikembangkan pada tahun 2009 di  
34      UC Berkeley's AMPLab oleh Matei Zaharia. Pada tahun 2010 Spark menjadi *open source* dibawah  
35      lisensi BSD. Pada tahun 2013, Spark didonasikan ke Apache Software Foundation. Saat ini Apache  
36      Spark menjadi top-level Apache projek [6].

37

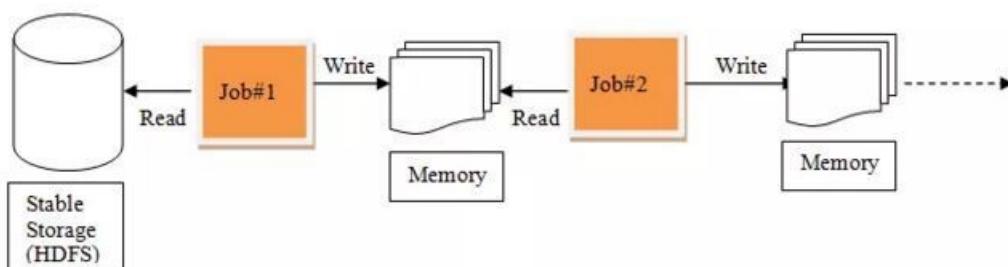


Gambar 2.5: Diagram ekosistem data processing Spark

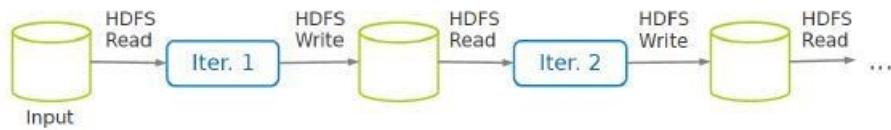
1 Apache Spark is a general-purpose cluster computing system to process big data workloads.  
 2 What sets Spark apart from its predecessors, such as Hadoop MapReduce, is its speed, ease of  
 3 use, and sophisticated analytics [6].

4 Apache Spark merupakan suatu ekosistem dari berbagai macam *library*, *package*, dan sistem.  
 5 Dengan demikian, Spark dapat mengerjakan berbagai macam teknik dalam memproses big data.  
 6 Apache Spark bukan untuk menggantikan framework Hadoop. Melakinkan sebuah alternatif dari  
 7 Hadoop MapReduce. Framework Hadoop melakukan pemrosesan data secara paralel menggunakan  
 8 pekerjaan map dan reduce. Hal ini membutuhkan waktu yang lama untuk menjalankan pekerjaan  
 9 hingga selesai. Spark didesain untuk berjalan diatas hadoop dan Spark menjadi sebuah alternatif  
 10 dari model *map/reduce*.

11



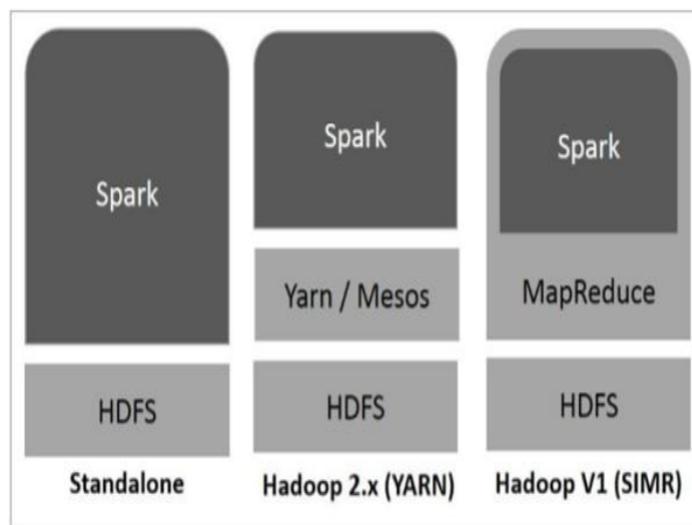
Gambar 2.6: Tahap pemrosesan data pada Spark



Gambar 2.7: Tahap pemrosesan data pada MapReduce Hadoop

1 MapReduce pada Hadoop cenderung menulis dan membaca data dari hard disk komputer setiap  
 2 tahap pemrosesan. Hal ini menimbulkan waktu tunggu pada saat pemrosesan berlangsung. Spark  
 3 dapat memproses data lebih cepat daripada model Hadoop MapReduce karena Spark telah diopti-  
 4 malkan untuk berjalan di memori. Lihat gambar 2.7 dan gambar 2.6 untuk ilustrasi pemrosesan  
 5 data pada masing-masing model. Jika pengolahan data membutuhkan perulangan atau iterasi pada  
 6 data maka Spark dapat memberikan kelebihan dalam hal kecepatan proses.

7



Gambar 2.8: Tiga cara untuk melakukan deployment pada Spark

8 Ada tiga cara untuk membangun Spark diatas komponen hadoop(Gambar 2.8) yaitu

9

- 10 • Standalone, Deployment Spark Standalone artinya Spark berjalan di atas HDFS. Pada kondisi  
 11 ini Spark berjalan berdampingan dengan MapReduce untuk mencakup semua pekerjaan Spark  
 12 pada cluster
- 13 • Hadoop Yarn, Deployment pada Hadoop Yarn artinya spark berjalan diatas Yarn tanpa  
 14 preparasi instalasi apapun atau meminta hak akses root. Hal ini membantu Spark untuk  
 15 berintegrasi dengan ekosistem pada Hadoop. Pada karya ilmiah ini, akan digunakan cara  
 16 deployment ini.
- 17 • Spark in Map Reduce (SIMR), Deployment ini digunakan untuk menjalankan pekerjaan spark  
 18 secara independen. Pengguna dapat menjalankan spark dan menggunakan spark shell tanpa

<sup>1</sup> perlu hak ases apapun.

<sup>2</sup> Spark terdiri dari beberapa komponen yang saling berintegrasi. Komponen-komponen ini sering  
<sup>3</sup> kali disebut dengan nama Spark Unified Stack. Pada intinya, Spark merupakan mesin komputasi  
<sup>4</sup> yang bertugas untuk melakukan penjadwalan, pendistribusian, dan pengawasan aplikasi yang  
<sup>5</sup> terdiri dari banyak tugas komputasi yang tersebar di banyak node. Sedangkan komponen pada  
<sup>6</sup> tingkat yang lebih tinggi merupakan komponen yang dapat mengerjakan tugas dengan berbagai  
<sup>7</sup> spesialisasi(Contohnya SQL atau Machine Learning). Komponen-komponen ini didesain beroperasi  
<sup>8</sup> secara berdekatan sehingga seperti sebuah *library* dalam projek perangkat lunak. Pada bagian  
<sup>9</sup> selanjutnya akan dijelaskan komponen yang ada pada Apache Spark.

<sup>10</sup>

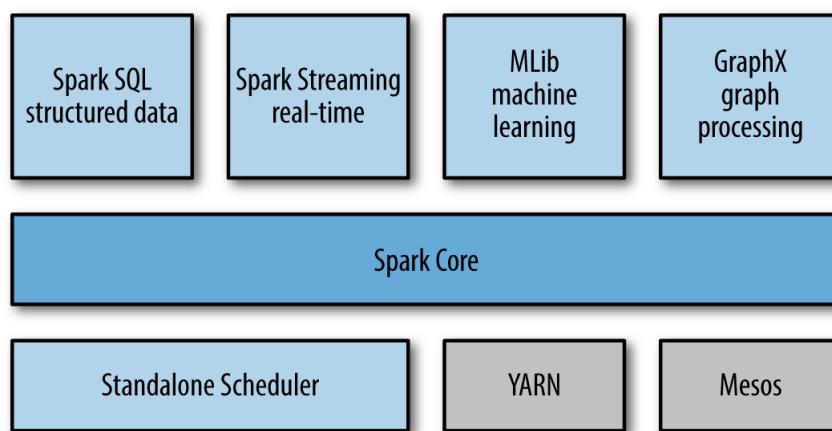
### <sup>11</sup> 2.3.1 Komponen Apache Spark

<sup>12</sup> Apache Spark memiliki komponen-komponen di dalamnya yang dinamakan Spark Stack(Gambar  
<sup>13</sup> 2.9). Di dalam Spark, terdapat sekumpulan API yaitu Streaming, SQL, Machine Learning, dan  
<sup>14</sup> GraphX. Sedangkan dibawahnya terdapat Spark Core yang menjadi pusat dari Spark. Saat ini,  
<sup>15</sup> API yang ada pada Spark tersedia dalam berbagai bahasa pemrograman diantaranya Scala, Java,  
<sup>16</sup> Python, dan R. Spark didesain untuk mencapai tujuan sebagai sebuah mesin komputasi yang  
<sup>17</sup> optimal. Untuk mendukung hal itu, Spark dapat menggunakan berbagai macam cluster manager  
<sup>18</sup> bahkan dapat berjalan independen(Standalone). Begitu juga dalam hal penyimpanan, Spark tidak  
<sup>19</sup> memiliki penyimpanan miliknya sendiri tetapi Spark dapat terhubung dengan berbagai macam  
<sup>20</sup> storage engine [7].

<sup>21</sup>

<sup>22</sup> Pada karya ilmiah ini, komponen yang akan dipelajari adalah Spark MLlib. Dengan demikian,  
<sup>23</sup> akan dijelaskan pemahaman tentang MLlib, Spark Core dan Cluster Manager. Berikut adalah  
<sup>24</sup> penjelasan masing-masing.

<sup>25</sup>



Gambar 2.9: Apache Spark stack

- <sup>26</sup> • Spark Core

<sup>27</sup> Pusat dari kegiatan atau inti dari Spark disebut dengan Spark Core. Spark Core dapat

memberikan layanan berupa mengatur memori, penjadwalan tugas pada *cluster*, melakukan *recovery* pada pekerjaan yang gagal, dan melakukan interaksi dengan sistem penyimpanan [7]. Spark Core memberikan abstraksi API bagi pengguna dari pekerjaan teknis tingkat rendah pada cluster. Spark-Core juga menyediakan API Resilient Distributed Datasets(RDD). RDD ini akan menjadi basis API dari API di level yang lebih tinggi. RDD akan dibahas pada bagian selanjutnya.

• Spark MLlib

MLlib merupakan kependekan dari Machine Learning Library yang ada pada Spark. Machine learning adalah salah satu jenis dari artificial intelligence(AI). Spark MLlib memberikan manfaat kepada programmer atau developer agar dapat menggunakan algoritma machine learning hanya dengan memanggil library ini. Sebagian besar dari algoritma machine learning melakukan processing secara iteratif. Dengan keunggulan spark yang dapat melakukan komputasi iteratif secara cepat, MLlib dapat berjalan secara optimal pada cluster computing. MLlib akan dijelaskan lebih detail pada bagian berikutnya.

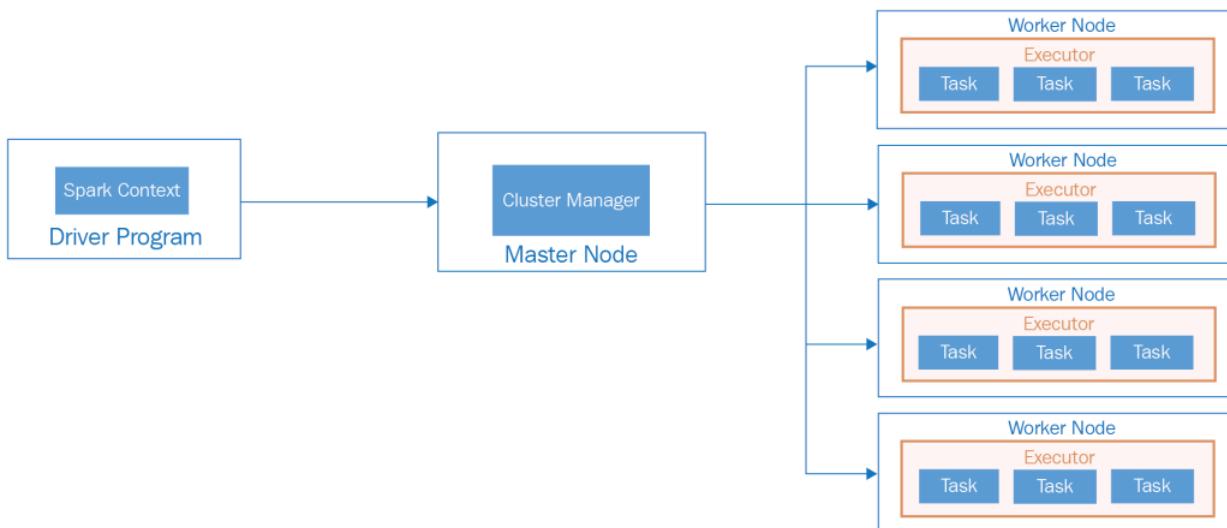
• Cluster Manager

Dibawah stack Spark Core terdapat Cluster Manager. *Cluster manager* adalah komponen yang mengatur *resource* node-node komputer pada sebuah cluster. Spark dirancang agar secara efisien ketika mengalami peningkatan jumlah node dari satu ke ribuan jumlah node komputasi. Dengan demikian, Spark mendukung untuk dapat berjalan pada berbagai cluster manager, seperti Hadoop YARN, Apache Mesos, dan cluster manager yang ada didalam Spark yang disebut Standalone Scheduler. Standalone Scheduler akan mengatur segala sesuatunya pada komputer ketika tidak ada cluster manager pada komputer.

Spark memiliki *tool* untuk menjalankan perintah-perintah sederhana yang dinamakan Spark Shell. Spark shell sama seperti scala REPL(Read-Evaluate-Print-Loop) dan dapat memanggil API yang ada pada Spark. Spark shell berguna untuk mempelajari Spark dan API yang ada didalamnya.

### 2.3.2 Arsitektur Apache Spark

Setelah mengetahui komponen apa saja yang ada didalam Spark. Selanjutnya dapat ditinjau lebih jauh arsitektur perangkat lunak Apache Spark. Pada bagian ini akan dijelaskan berdasarkan tinjauan pada tingkat tinggi. Selain itu akan dijelaskan pula beberapa deskripsi dari komponen penting dalam perangkat lunak. Diantaranya adalah Driver program, Master node, Worker node, Executor, Tasks, SparkContext, dan Spark session. Spark menggunakan arsitektur master/slave yaitu arsitektur yang memiliki satu koordinator central dan banyak *worker*(pekerja) yang terdistribusi. Pada gambar 2.10 menunjukan arsitektur Apache Spark dimana pada Node yang berbeda, ada komponen yang berbeda pula.



Gambar 2.10: Arsitektur Apache Spark

1 Berikut penjelasan masing-masing komponen perangkat lunak :

3 • **Driver Program**

4 *Driver program* merupakan program utama dari aplikasi Spark. Mesin(Komputer) tempat  
 5 proses Aplikasi Spark berjalan dinamakan *Driver node*. Sedangkan proses tersebut dina-  
 6 makannya dinamakan *Driver Process*. *Driver program* akan berkomunikasi dengan *Cluster Manager* untuk  
 7 mendistribusikan pekerjaan kepada *executor*. Pengertian *executor* akan dijelaskan pada poin  
 8 berikutnya. *Driver program* berjalan didalam JVM.

9 • **Cluster Manager**

10 Sesuai dengan namanya, *Cluster manager* berfungsi untuk mengatur sebuah *cluster*. Sebe-  
 11 lumnya telah dibahas bahwa Spark dapat berjalan dengan berbagai macam *cluster manager*.  
 12 Termasuk didalamnya adalah *Standalone Cluster Manager*. *Standalone Cluster Manager*  
 13 didominasi oleh dua *daemon*. Satu *daemon* berjalan pada *node master*, sedangkan satunya  
 14 lagi berada pada *node worker*.

15 • **Worker**

16 *Worker node* mirip seperti *slave node* pada Hadoop. Dimana mereka merupakan mesin-mesin  
 17 yang melakukan tugas *processing*. Mesin-mesin *worker* adalah tempat dimana pekerjaan dari  
 18 sebuah program yang dieksekusi. Setiap node pada spark(kecuali master node) menjalankan  
 19 *worker process*. Process ini akan memberikan informasi kepada master node tentang *resource*  
 20 yang tersedia. Normalnya, hanya satu *daemon worker* yang dapat berjalan pada setiap *worker*  
 21 *node*. Ketika *worker* dijalankan, *worker* akan memonitor *executor* dari aplikasi.

22 • **Executors**

23 *Master node* akan mengalokasikan resource dan menggunakan *worker* di cluster untuk mem-  
 24 buat *executor* dari *driver*. *Driver* dapat menggunakan banyak *executor* untuk menjalankan  
 25 pekerjaan. *Executor* dibuat ketika ada pekerjaan yang akan dieksekusi pada *node worker*.

1 Setiap aplikasi memiliki *executor proses*. *Executor* tersebut akan tetap ada selama aplikasi  
2 itu berjalan dan akan berjalan pada beberapa *thread*. Hal ini mengakibatkan data tidak akan  
3 dibagikan diantara aplikasi yang berjalan. *Executor* bertanggung jawab terhadap pekerjaan  
4 yang berjalan dan yang mengatur data untuk berada di memori atau berada pada *storage*.

5 • **Tasks**

6 Sebuah unit *task* adalah sebuah satuan kerja yang akan dikirimkan ke *executor*. Biasanya  
7 berupa sebuah perintah yang dikirimkan oleh *driver program* kepada *executor* dengan mense-  
8 rialisasi fungsi. *Executor* akan mendeserialisasi perintah tersebut dan mengeksekusinya pada  
9 sebuah partisi. Sebuah *partition* adalah sebuah potongan data yang distribusi ke seluruh  
10 cluster Spark. Pada umumnya ketika spark membaca data dari *distributed storage*, Spark akan  
11 membagi data kedalam beberapa *partition* dalam rangka untuk memproses data secara paralel  
12 di *cluster*. Spark akan menjalankan satu tugas pada setiap *partition*. Dengan demikian jumlah  
13 *partition* penting untuk diperhatikan. Secara default spark akan menentukan jumlah *partition*  
14 yang akan dibentuk. Namun pengguna juga dapat mengganti jumlah *partition* secara manual.

15 • **SparkContext**

16 SparkContext merupakan komponen yang menghubungkan pengguna dengan cluster Spark.  
17 SparkContext menjembatani pengguna dengan *cluster*. SparkContext dapat digunakan untuk  
18 membuat RDD, *accumulator*, dan *broadcast variable*. Sangat dianjurkan untuk hanya memiliki  
19 satu SparkContext yang aktif per JVM(Java Virtual Machine). Jika ada SparkContext  
20 yang aktif maka pengguna dapat memberhentikan SparkContext tersebut agar dapat dibuat  
21 SparkContext yang baru. Pada Spark Shell SparkContext otomatis dibuat dan *variable* yang  
22 merujuk SparkContext tersebut adalah variable sc.

23 • **Spark Session**

24 *Spark session* adalah titik awal untuk melakukan programming pada Spark dengan dataset.

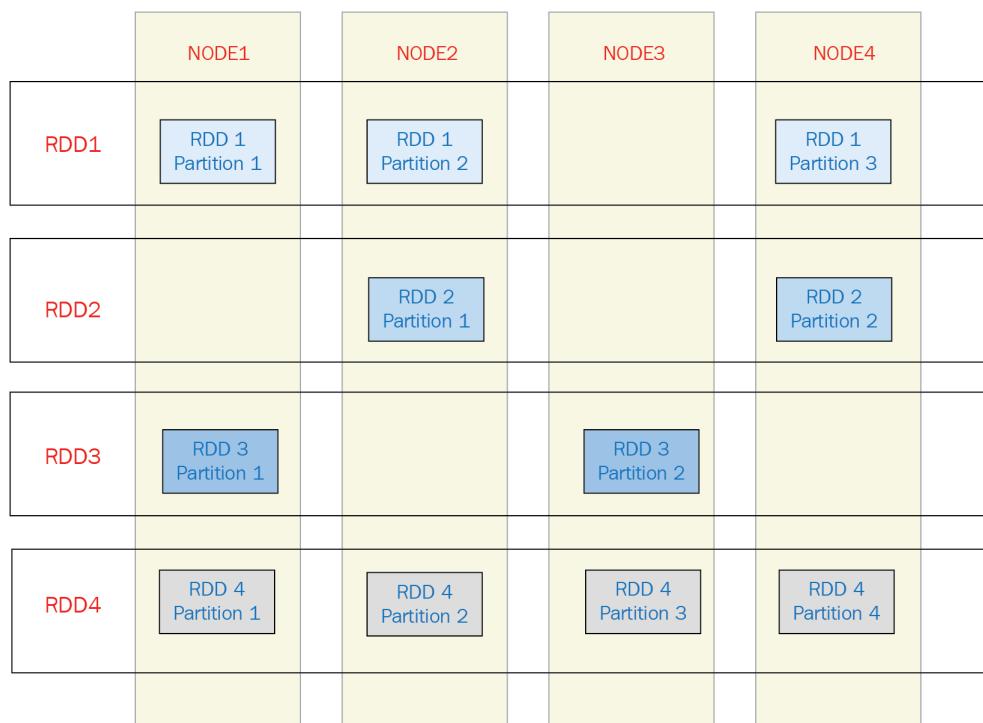
25 Ada beberapa poin penting yang perlu diperhatikan dalam arsitektur ini yaitu:

- 26
- 27 • Setiap aplikasi mendapatkan proses *executor* sendiri, yang tetap aktif selama aplikasi berlang-  
28 sung dan menjalankan tugas di banyak *thread*. Hal Ini memiliki manfaat untuk mengisolasi  
29 aplikasi satu sama lain, baik pada sisi penjadwalan (masing-masing jadwal driver pada setiap  
30 tugasnya sendiri) dan sisi *executor* (tugas dari berbagai aplikasi berjalan di JVM yang berbe-  
31 da). Namun, hal ini juga berarti bahwa data tidak dapat dibagi di berbagai aplikasi Spark  
32 yang berbeda (contoh SparkContext) tanpa menuliskannya ke sistem penyimpanan eksternal.
  - 33 • Spark tidak dapat mengetahui kondisi dari *cluster manager* yang mendasarinya. Selama  
34 Spark masih dapat mengakuisisi proses *executor* dan masih saling berkomunikasi, Spark relatif  
35 mudah untuk menjalankannya bahkan pada *cluster manager* yang juga mendukung aplikasi  
36 lain (misalnya Mesos / YARN).
  - 37 • Driver program harus mendengarkan dan menerima koneksi yang masuk dari *worker* sepanjang  
38 waktu (misalnya, dengan melihat spark.driver.port di bagian konfigurasi jaringan). Dengan  
39 demikian, *driver program* harus bisa dialamatkan dari *node worker*.

- Karena *driver* menjadwalkan tugas di *cluster*, maka *driver* harus dijalankan dekat dengan *worker node*, sebaiknya pada jaringan area lokal yang sama. Jika ada permintaan ke *cluster* dari jarak jauh, ada baiknya menggunakan RPC ke *driver* dan membiarkannya melakukan operasinya dari dekat daripada menjalankan *driver* yang jauh dari *worker node*.

### 2.3.3 Resilient Distributed Datasets

Pada bagian sebelumnya telah dibahas arsitektur dari spark dan komponen penting pada aplikasi. Salah satunya adalah SparkContext. Pada bagian SparkContext disebutkan bahwa SparkContext dapat membuat RDD. RDD merupakan salah satu konsep penting dalam Spark. Lalu apa itu RDD?



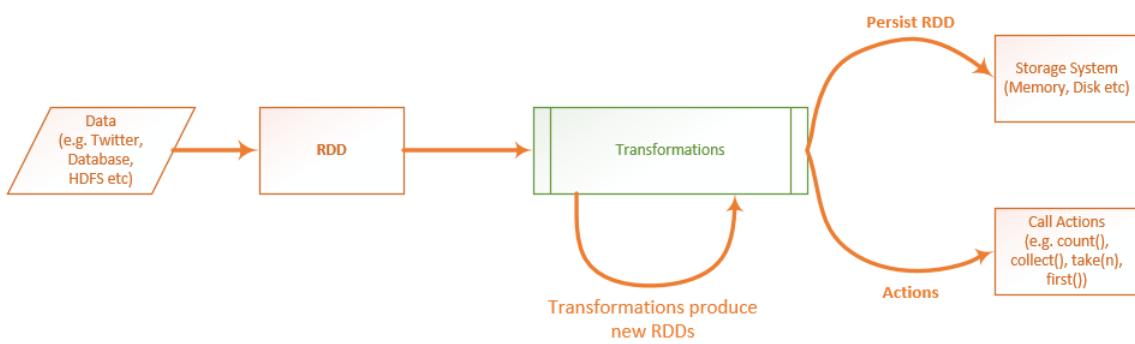
Gambar 2.11: RDD pada sebuah cluster

RDD merupakan singkatan dari Resilient Distributed Datasets. Pada dasarnya RDD merupakan sebuah dataset yang didistribusikan pada sebuah cluster (Gambar 2.11). Dengan demikian, RDD Spark dapat didefinisikan sebagai sebuah koleksi data terdistribusi yang bisa dioperasikan secara paralel. Setiap RDD dibagi menjadi beberapa partisi dan Spark akan menjalankan suatu pekerjaan untuk setiap partisi. RDD bukan berisi data aktual namun berbentuk objek yang berisi informasi data yang berada pada *cluster*.

RDD bersifat *resilient* atau *fault-tolerant*. Artinya kegagalan node dapat diatasi oleh RDD. RDD bersifat *resilient* karena RDD dapat dikomputasi kembali dari RDD lineage graph. RDD lineage graph merupakan sebuah graf dari seluruh *parent* RDD dari RDD itu sendiri [7].

Selain bersifat *resilient*, RDD memiliki beberapa pembeda lainnya yaitu :

- 1     ● *In Memory*, RDD merupakan sebuah kumpulan object yang berada pada memori. Meskipun  
2     RDD memiliki pilihan untuk disimpan di memori, hardisk, atau bahkan keduanya, kecepatan  
3     eksekusi Spark berdasarkan fakta bahwa data berada dalam memori. Bukan mengambil dari  
4     hard disk setiap kali menjalankan operasi.
- 5     ● *Partitioned*, Melakukan partisi adalah suatu syarat mutlak teknik untuk mengoptimalkan  
6     performa pada distributed system. Hal ini bertujuan meminimalisir network traffic dan  
7     sekaligus pemungkas high performance workloads. Dalam melakukan partisi, data yang  
8     berupa *key/value* ditempatkan sesuai dengan rentang key yang bernilai sama. Tujuannya  
9     untuk meminimalisir data yang berpindah-pindah.
- 10    ● *Typed*, Data didalam RDD akan selalu digolongkan berdasarkan type data.
- 11    ● *Lazy evaluation*, Transformation pada Spark bersifat "lazy" atau malas. Ini artinya data  
12     didalam RDD tidak akan tersedia sampai dilakukan sebuah action.
- 13    ● *Immutable*, Sebuah RDD yang telah dibuat tidak dapat berubah. Meskipun demikian, RDD  
14     dapat ditransformasi menjadi sebuah RDD baru dengan melakukan perintah transformation  
15     pada RDD.
- 16    ● *Parallel*, Sebuah RDD dapat dioperasikan secara pararel. Artinya partisi yang ada pada RDD  
17     dapat dioperasikan secara pararel di seluruh cluster.
- 18    ● *Cacheable*, karena RDD bersifat lazy evaluation, setiap action yang dilakukan pada RDD akan  
19     menyebabkan RDD mengevaluasi kembali transformasi yang menyebabkan pembuatan RDD.  
20     Karena hal ini merupakan sifat yang berdampak buruk pada dataset berukuran besar, maka  
21     Spark memberikan pilihan untuk melakukan *cache* data di *memory* ataupun pada hard disk.
- 22    Program Spark umumnya memiliki pola urutan proses yang terjadi pada RDD (Gambar 2.12) yaitu:
- 23    a. Pembuatan RDD dari sumber data
- 24    b. Sebuah perintah *transformation*
- 25    c. Mempertahankan RDD pada memori atau disk
- 26    d. Memanggil satu atau lebih action pada RDD untuk melakukan operasi pararel pada *cluster*.



Gambar 2.12: Proses yang umum terjadi pada RDD

1 Ada dua cara untuk membuat suatu RDD. Pertama, dengan cara melakukan *parallelizing* sebuah  
 2 *collection* di *driver program*. Parallelizing sebuah *collection* dilakukan dengan memanggil method  
 3 `parallelize()` pada `SparkContext` didalam *driver program*. Method akan `parallelize()` melalui  
 4 permintaan kepada Spark untuk membuat RDD baru berdasarkan dataset yang ada. Setelah  
 5 sebuah dataset/*collection* tersebut telah diubah menjadi sebuah RDD, maka dataset(RDD) dapat  
 6 dioperasikan secara pararel. `parallelize()` Biasanya hanya digunakan untuk *prototyping*. Hal ini  
 7 dikarenakan data set yang dibutuhkan untuk membuat RDD harus ada pada mesin(komputer) itu.  
 8 Contoh kode program yang melakukan *parallelizing* sebuah *collection* di `SparkShell`

9  
 10 Cara yang kedua adalah dengan cara memberikan referensi ke suatu sumber data eksternal.  
 11 Contoh: Filesystem HDFS, Hbase, Amazon S3, dsb. Cara ini merupakan metode yang digunakan  
 12 dalam tahap produksi. Ada beberapa method yang digunakan :

- 13
- 14 • `hadoopFile()`: Membuat RDD dari file Hadoop dengan format input yang beragam.
  - 15 • `objectFile()`: Memuat sebuah RDD yang disimpan sebagai `SequenceFile` yang berisi  
 16 serialized object, dengan keys bertipe `NullWritable`, dan values bertipe `BytesWritable` yang  
 17 berisi serialized partition
  - 18 • `sequenceFile()`: Membuat sebuah RDD dari sequence file Hadoop dengan tipe data *key* dan  
 19 *value* yang telah ditentukan.
  - 20 • `textFile()`: Membuat RDD bertipe String dengan membaca sebuah file teks dari HDFS atau  
 21 filesystem lokal.

22 RDD dapat melakukan operasi-operasi. Ada 2 operasi yang dapat dilakukan RDD yaitu *Transformation*  
 23 dan *Action*. *Transformation* adalah operasi yang membuat dataset baru. Operasi ini  
 24 tidak mengembalikan suatu nilai apapun ke driver program karena sifatnya yang "lazily evaluated".  
 25 Contoh, sebuah transformation yaitu fungsi `map` yang akan menelusuri setiap elemen RDD dan  
 26 mengembalikan RDD baru yang merepresentasikan hasil aplikasi dari fungsi tersebut terhadap

1 dataset yang asli.

2

3 Operasi lainnya bernama *Action*. *Action* merupakan operasi yang melakukan komputasi berda-  
4 sarkan data dari RDD. Hasil dari *Action* dapat dikembalikan pada driver program atau menyim-  
5 pannya pada external storage system(seperti HDFS). Seperti yang telah dijelaskan sebelumnya,  
6 seluruh transformation Spark memiliki sifat 'malas'. Hal ini memiliki arti bahwa Spark mengingat  
7 seluruh *transformation* yang dilakukan terhadap RDD. Kemudian menerapkannya secara optimal  
8 ketika sebuah action dipanggil.

9

10 Spark RDD akan kembali dibuat setiap kali operasi *action* dipanggil. Agar ketika *action*  
11 dipanggil berkali-kali RDD yang telah dibuat dapat digunakan kembali, Spark dapat membantu  
12 untuk menyimpan RDD dengan memanggil method **persist()**. Spark akan menyimpan konten  
13 RDD pada memori dan menggunakan kembali pada *action* berikutnya.

14

15 Setelah mengetahui tentang RDD, dapat disimpulkan bahwa RDD memiliki fitur yang berbeda  
16 jika dibandingkan dengan hadoop. Perbedaan itu diantaranya adalah

17

- 18 • Di hadoop, data disimpan menggunakan blok-blok dan disimpan didalam data node yang  
19 berbeda. Pada Spark, partisi dibuat dari RDD dan disimpan di datanode yang dikumputasi  
20 secara paralel di semua node.
- 21 • Di hadoop, data harus direplikasi agar dapat mengatasi kegagalan hardware atau software,  
22 namun pada Spark replikasi tidak diperlukan karena sudah dilakukan oleh RDD
- 23 • RDD memuat data untuk pengguna dan bersifat resilient, artinya data tersebut dapat  
24 digunakan kembali untuk komputasi
- 25 • RDD Melakukan dua jenis operasi yaitu *Transformation* dan *Action*
- 26 • RDD merupakan lazy evaluation, artinya RDD menyimpan jejak *Transformation* dan mengecek  
27 secara berkala. Jika ada node yang gagal, RDD dapat melakukan kembali partisi RDD yang  
28 hilang pada node lain secara paralel

29 **2.3.4 MLLib**

30 Pada berbagai macam kasus, pengetahuan yang ingin didapatkan dari data mentah tidak dapat  
31 terlihat jelas dengan melihat pada data. Maka dari itu dibutuhkan peran dari *Machine Learning*  
32 yaitu untuk mengubah data menjadi sebuah informasi atau pengetahuan yang berguna [8]. *Machine*  
33 *Learning* memiliki makna sama seperti automated learning. Belajar (Learning) adalah sebuah proses  
34 mengubah pengalaman menjadi sebuah pengetahuan. Dengan demikian *Machine Learning* dapat  
35 diibaratkan seperti komputer yang dapat belajar dari input yang diberikan. Input yang diberikan  
36 pada algoritma *Machine Learning* berupa data pelatihan yang menggambarkan suatu pengalaman  
37 dan outputnya adalah pengetahuan.(understanding machine learning)

38

1 Terdapat dua kategori algoritma machine learning yang akan dipelajari pada karya ilmiah ini  
2 yaitu

3 • **Supervised Learning:** pada *supervised learning*, data yang digunakan sudah terlebih dahulu  
4 dilabeli. Data tersebut dapat digunakan untuk pelatihan model yang nantinya dapat digunakan  
5 untuk memprediksi label pada data baru yang belum dilabeli. Algoritma yang umum digunakan  
6 adalah NaÃrve Bayes, Decision trees, regression, neural network, svm, dan lain-lain.

7 • **Unsupervised learning:** pada *unsupervised learning*, data yang digunakan tidak memiliki  
8 label. Dengan menggunakan algoritma machine learning, wilayah yang objeknya serupa pada  
9 ruang multidimensi dapat diketahui. Algoritma yang umum digunakan adalah Clustering,  
10 principal component analysis, dan lain-lain.

11 Apache Spark memiliki *library* kumpulan algoritma-algoritma *Machine Learning* yang dina-  
12 makan MLlib. MLlib dirancang untuk berjalan secara paralel di *cluster*. MLlib berisi berbagai  
13 algoritma Machine Learning dan dapat diakses dari semua bahasa pemrograman Spark seperti  
14 (Scala, Java, dan Phyton). Namun demikian, MLlib dibangun dengan bahas pemrograman Scala.  
15 Machine learning adalah topik yang cukup luas untuk dipelajari. Pada studi literatur ini menjelaskan  
16 bagaimana cara menggunakan beberapa *library* MLlib yang ada pada Spark.

17  
18 Modul Machine Learning menyediakan beberapa fungsionalitas dintaranya adalah :

- 19 • Statistics  
20 • Classification dan Regression  
21 • Collaborative Filtering  
22 • Clustering  
23 • Dimensionality Reduction  
24 • Feature Extraction

25 Algoritma *machine learning* membuat prediksi atau keputusan berdasarkan pelatihan data dan  
26 memaksimalkan tujuan tentang perilaku algoritma. Semua algoritma Machine Learning memerlukan  
27 set fitur untuk setiap objek, yang akan dimasukkan ke dalam fungsi Machine Learning. Misalnya  
28 untuk kasus penentu email spam. Beberapa fitur dapat seperti server email itu berasal, jumlah  
29 penyebutan kata (contohnya kata "FREE"), atau warna teksnya. Dalam banyak kasus, mendefi-  
30 nisikan fitur yang tepat adalah bagian yang paling menantang dalam menggunakan *machine learning*.

31  
32 Sebagian besar algoritma dibuat hanya untuk menangani fitur bernilai numerik (contoh, sebuah  
33 vektor angka yang mewakili nilai untuk setiap fitur). Karena itu, pengolahan terlebih dahulu seperti  
34 Feature Extraction dan Feature transformation untuk menghasilkan fitur tersebut.

35  
36 MLlib memiliki beberapa tipe data khusus, lokasinya berada di org.apache.spark.mllib (Java /  
37 Scala). Diantaranya adalah:

- 1 • Vector, adalah sebuah vektor matematika. MLlib mendukung kedua *dense vector*, di mana  
2 setiap entri disimpan, dan *sparse vectors*, di mana hanya entri yang tidak nol yang disimpan  
3 untuk menghemat ruang. Vektor dapat dibangun dengan kelas mllib.linalg.Vectors.
- 4 • LabeledPoint, digunakan untuk *supervised learning algorithm* (algoritma Machine Learning  
5 yang diawasi) seperti *classification* dan *regression*. LabeledPoint berisik vektor fitur dan label  
6 (yang merupakan nilai *floating-point*). Terletak di paket mllib.regression.
- 7 • Rating, Rating produk oleh pengguna, digunakan dalam paket **mllib.recommendation** untuk  
8 rekomendasi produk
- 9 • Berbagai model *class*, Setiap Model adalah hasil dari pelatihan algoritma, dan biasanya  
10 memiliki method **predict()** untuk menerapkan model ke suatu data baru atau ke RDD dari  
11 sebuah data baru.

12 Ada beberapa hal yang perlu diperhatikan untuk kelas Vector di MLlib, yang akan menjadi yang  
13 paling banyak digunakan. Pertama, vektor tersedia dalam dua bentuk: *Dense* dan *Sparse*. *Dense*  
14 *Vector* menyimpan semua entri mereka ke dalam sebuah array bilangan *floating-point*. Misalnya,  
15 vektor dengan ukuran 100 akan berisi 100 nilai double. Sebaliknya, *Sparse Vector* menyimpan hanya  
16 nilai-nilai yang tidak nol dan indeks mereka. *Sparse Vector* biasanya lebih disukai (baik dari segi  
17 penggunaan memori dan kecepatan) jika paling banyak 10% elemen tidak nol. Hal ini menjadikan  
18 kunci optimalisasi.

19  
20 Untuk membuat data bertipe *vector*, pengguna harus terlebih dahulu melakukan *import* terhadap  
21 *package library* MLlib yaitu

22 `import org.apache.spark.mllib.linalg.Vectors`

23 Selanjutnya, agar dapat menginisialisasi data Vector digunakan method **dense** dari object  
24 *Vectors*. Ada dua cara untuk membuat data dense vector berisi nilai 1.0, 2.0, dan 3.0.

25 `val denseVec1 = Vectors.dense(1.0, 2.0, 3.0)`

26 atau

27 `val denseVec2 = Vectors.dense(Array(1.0, 2.0, 3.0))`

28 Sedangkan untuk membuat data *sparse Vector*, pengguna dapat menggunakan method **sparse**  
29 dari object *Vectors*. Method **sparse** menerima parameter ukuran vector, array posisi nilai bukan  
30 nol, dan nilai dari datanya. Berikut cara membuat sparse Vector yang nilainya <1.0, 0.0, 2.0,  
31 0.0> .

32 `val sparseVec1 = Vectors.sparse(4, Array(0, 2), Array(1.0, 2.0))`

### 33 Clustering

34 *Clustering* adalah suatu teknik *machine learning* yang melakukan pengelompokan objek kedalam  
35 suatu *cluster* yang memiliki tingkat kemiripan yang tinggi. *Clustering* adalah termasuk kedalam ka-  
36 tegori unsupervised learning. Ini artinya, *clustering* menggunakan sebuah data yang tidak memiliki

1 label. *Clustering* pada umumnya digunakan untuk melakukan eksplorasi data dan biasanya dapat  
2 mendeteksi anomali pada data.

3  
4 Ada 7 kategori algoritma pada teknik *clustering*. Diantaranya adalah *Hierarchical clustering*  
5 *algorithm*, *Density-based clustering algorithm*, *Partitioning clustering algorithm*, *Graph-based al-*  
6 *gorithm*, *Grid-based algorithm*, *Model-based clustering algorithm*, dan *Combinational clustering*  
7 *algorithm*. Algoritma *clustering* ini memberikan hasil yang berbeda sesuai dengan tujuan spesifiknya.  
8 Sebagian teknik *clustering* lebih optimal dipakai jika menggunakan dataset yang berukuran besar.  
9 Sebagian lainnya memberikan hasil yang baik dalam menentukan *cluster* yang bervariasi [9]. Salah  
10 satu algoritma *clustering* yang digunakan pada skripsi ini adalah algoritma K-Means.

11  
12 K-Means memiliki tujuan yang ingin dicapai yaitu mengelompokan sebuah dataset kedalam  
13 sejumlah  $k$  kelompok(disebut juga *cluster*). Setiap *cluster*(kelompok) didefinisikan dengan sebu-  
14 ah titik yang dinamakan centroid. Centroid adalah titik tengah dari semua objek di dalam cluster [8].  
15

16 Cara kerja K-Means diuraikan sebagai berikut. Pertama, sejumlah  $k$  centroid diinisialisasi secara  
17 acak. Selanjutnya, setiap titik(atau disebut objek) pada dataset dimasukkan kedalam sebuah *cluster*.  
18 Penempatan tersebut dilakukan dengan cara mencari centroid terdekat dari objek dan memasukkan  
19 objek tadi ke *cluster* tersebut. Setelah seluruh objek ditempatkan ke masing-masing *cluster*, semua  
20 centroid kemudian diupdate dengan mencari nilai rata-rata dari objek didalam cluster. Iterasi ini  
21 dilakukan hingga tidak ada objek yang clusternya berubah. Berikut adalah *pseudocode*-nya [8].  
22

---

**Algorithm 1:** K-Means algorithm

---

**Result:** Centroid

Create  $k$  points for starting centroids (often randomly) **while** *any point has changed cluster  
assignment do*

**for** *every point in our dataset do*  
 | **for** *every centroid do*  
 | | calculate the distance between the centroid and point  
 | **end**  
 | assign the point to the cluster with the lowest distance  
**end**  
**for** *every cluster calculate the mean of the points in that cluster do*  
 | assign the centroid to the mean  
**end**  
**end**

24 Dalam mencari centroid terdekat, dilakukan perhitungan jarak. Perhitungan jarak dapat  
25 menggunakan metode Euclidean Distance. Jika  $p = (p_1, p_2, \dots, p_n)$  dan  $q = (q_1, q_2, \dots, q_n)$  adalah  
26 representasi centroid dan objek maka Euclidean Distance dirumuskan sebagai berikut :

$$d(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

27 MLlib memiliki modul algoritma K-Means didalamnya. K-Means yang dimiliki adalah varian K-

1 Means yang disebut K-means||. Algoritma K-means|| memiliki efektifitas yang baik pada komputasi  
2 paralel. K-means|| hampir sama dengan prosedur inisialisasi K-means++ yang sering digunakan  
3 dalam komputasi pada mesin tunggal. K-means memiliki parameter yang paling penting yaitu  
4 jumlah *cluster* yang akan dihasilkan( $k$ ).

5 Pada penggunaannya, pengguna jarang mengetahui jumlah kelompok yang ideal untuk dite-  
6 rapkan. Maka dari itu, cara terbaik adalah dengan melakukan percobaan pada beberapa nilai  $k$   
7 sampai jarak rata-rata antar *cluster* berhenti menurun secara derastis. Sayangnya, algoritma  
8 K-Means hanya dapat menerima perintah untuk mengkomputasi sebuah nilai  $kK$  sekali dijalankan.  
9 Pada algoritma K-Means MLlib, selain parameter  $K$ , K-means MLlib dapat menerima parameter  
10 maxIterations. maxIteration digunakan untuk mengatur jumlah iterasi maksimum untuk dijalankan.

11 Pemanggilan K-Means dilakukan dengan membuat objek dari package `mllib.clustering.KMeans`.  
12 Dibutuhkan RDD berisi Vectors sebagai input parameter. KMeans MLlib menghasilkan output  
13 berupa KMeansModel yang dapat digunakan untuk mendapatkan centroid bertipe array. Selain  
14 itu, dengan KMeansModel pengguna dapat memanggil method `predict()` pada suatu data baru  
15 untuk menampilkan prediksi kelompoknya. Method `predict()` akan mengembalikan hasil berupa  
16 *cluster*(kelompok) terdekat dengan objek, bahkan jika objek jauh dari semua kelompok.

17

18 Algoritma K-Means MLlib bersifat iteratif. Itu artinya, dalam melakukan komputasi, program  
19 akan menggunakan data RDD berulang kali. Maka dianjurkan untuk memanggil method `cache()`  
20 pada data RDD sebelum dilakukan pelatihan data ke *library* MLlib.

## 21 **Classification dan Regression**

22 Classification adalah suatu teknik dalam machine learning untuk mengatasi masalah dalam meng-  
23 identifikasi kategori dari sebuah data baru. Untuk dapat mengidentifikasi sebuah data baru,  
24 dibutuhkan sebuah data pelatihan yang kategori keanggotaanya diketahui [7]. Dengan demikian  
25 Classification termasuk dalam kategori *supervised learning*. Salah satu algoritma pada MLlib yang  
26 akan digunakan pada skripsi ini adalah algoritma Naive Bayes.

27 Regression adalah sebuah teknik untuk memprediksi sebuah nilai berdasarkan model pelatihan  
28 sebuah data [7]. Perbedaan dengan classification adalah regression menggunakan data continuous  
29 untuk memprediksi sebuah nilai. Regression membuat asumsi keterkaitan antara input dan output.

30 Classification dan regression menggunakan kelas khusus untuk tipe data di MLlib yaitu kelas  
31 `LabeledPoint`. Kelas ini berada dibawah package `mllib.regression`. LabeledPoint terdiri dari  
32 sebuah label(Bertipe Double) dan sebuah vector fitur.

33 Salah satu algoritma yang akan digunakan pada karya tulis ini adalah algoritma Classification  
34 yaitu Naive Bayes. Algoritma Naive bayes adalah sebuah teknik klasifikasi menggunakan metode  
35 probabilitas dan statistik yg dikemukakan oleh ilmuwan Inggris Thomas Bayes. Naive bayes meru-  
36 pakan sebuah model probabilistik yang membuat prediksi dengan melakukan komputasi probabilitas  
37 dari sebuah titik data yang dimiliki oleh suatu kelas. Naive bayes model mengasumsikan bahwa  
38 setiap fitur berkontribusi secara independen terhadap probabilitas yang ditetapkan ke suatu kelas.

39

40 Ada dua tahap pada Naive Bayes dalam machine learning, tahap pertama yaitu pelatihan  
41 (*training*) atau pembelajaran. Pada tahap pelatihan digunakan set data yang isi set datanya

1 memiliki label kelas. Pada tahap kedua yaitu pengujian atau pengklasifikasian sebuah set data  
 2 baru. Pada tahap ini set data belum memiliki label kelas dan akan dikelompokan menjadi sebuah  
 3 label kelas yang ada pada model.

4  
 5 Pada tahap training digunakan teorema Bayesian(Bayes). Untuk menjelaskan secara singkat  
 6 teorema bayes, dimisalkan ada dua kejadian probabilistik A dan B. Probabilitas terjadinya suatu  
 7 kejadian B bila diketahui bahwa kejadian A telah terjadi disebut probabilitas bersyarat dan dinota-  
 8 sikan dengan  $P(B | A)$ , Begitu pula sebaliknya dinotasikan dengan  $P(A | B)$ . Bayes menggambarkan  
 9 hubungan antara peluang bersyarat dari dua kejadian A dan B sebagai berikut:

10

11 
$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

12 Rumus di sebelah kanan digunakan untuk mendapatkan nilai di sebelah kiri rumus. Pada machine  
 13 learning perhitungan ini dilakukan pada setiap kelas dan membandingkan kedua kemungkinan.  
 14 Pada algoritma naive bayes, setiap objek dikategorikan menjadi i kelas( $c_i$ ) dan setiap atribut ke-n  
 15 objek dituliskan sebagai  $w_n$ . Berikut persamaan yang dipakai:

16

17 
$$P(c_i|w) = \frac{P(w|c_i) \times P(c_i)}{P(w)}$$

18 Untuk menghitung  $p(c_i)$  dilakukan penjumlahan setiap kelas i muncul dibagi dengan total seluruh  
 19 objek. Sedangkan untuk menghitung  $p(w|c_i)$  harus terlebih dahulu dijabarkan fitur per individu.  
 20 Sehingga dapat dituliskan sebagai  $p(w_0, w_1, w_2 \dots w_n | c_i)$ . Karena asumsinya setiap fitur independen  
 21 maka probabilitas dapat dihitung dengan  $p(w_0|c_i), p(w_1|c_i), p(w_2|c_i), \dots, p(w_n|c_i)$ . Implementasi  
 22 fungsi ini dijelaskan dengan pseudocode berikut ini [8]:

23

---

**Algorithm 2:** Naive Bayes algorithm
 

---

Count the number of documents in each class **for** *every training document do*  
**for** *each class do*  
 | **if** *a token appears in the document then*  
 | | increment the count for that token  
 | **end**  
 | increment the count for tokens  
**end**  
**for** *each class do*  
 | **for** *each token do*  
 | | divide the token count by the total token count to get conditional probabilities  
 | **end**  
**end**  
**return** *conditional probabilities for each class*  
**end**


---

<sup>1</sup> **Statistics**

<sup>2</sup> Statistik merupakan teknik paling dasar dalam melakukan analisis data. Di matematika teknik  
<sup>3</sup> ini sangat umum digunakan. Beberapa perhitungan statistik disediakan didalam MLlib. Seperti  
<sup>4</sup> diantaranya menghitung nilai statistik mean(rata-rata), minimum, maximum, dan varian (Disebut  
<sup>5</sup> sebagai *summary statistics*). Selain itu ada fungsi yang dapat mencari korelasi antara dua dan  
<sup>6</sup> masih ada beberapa fungsi lainnya.

<sup>7</sup>

<sup>8</sup> MLlib menyediakan suatu kelas untuk menjalankan perhitungan statistik pada RDD Spark. Ke-  
<sup>9</sup> las itu adalah `mllib.stat.Statistics`. Pengguna hanya perlu memanggil method yang diperlukan  
<sup>10</sup> untuk melakukan operasi kepada RDD. Method yang akan digunakan pada skripsi ini diantaranya  
<sup>11</sup> adalah `Statistics.colStats()`. Method ini digunakan untuk melakukan perhitungan statistik  
<sup>12</sup> seperti nilai rata-rata, minimum, maksimum, dan varian untuk setiap kolom pada sebuah data  
<sup>13</sup> vektor yang diberikan.

<sup>14</sup>

<sup>15</sup> Rumus untuk menghitung nilai rata-rata ( $\bar{x}$ ) adalah sebagai berikut :

<sup>16</sup>

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i = \frac{1}{n} (x_1 + \dots + x_n)$$

<sup>17</sup> Sedangkan untuk menghitung nilai variance ( $\sigma^2$ ) adalah sebagai berikut :

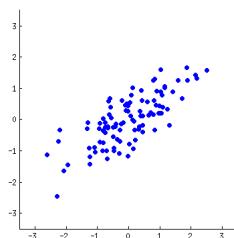
<sup>18</sup>

$$\sigma^2 = \frac{\sum_{i=1}^n (x_i - \mu)^2}{n}$$

<sup>19</sup> **Dimensionality Reduction**

<sup>20</sup> Ketika mengolah sebuah data yang dengan ukuran dimensional yang sangat besar akan lebih rumit  
<sup>21</sup> untuk dianalisis. Sehingga dibutuhkan penyederhanaan yaitu dengan mengurangi dimensi dari data.  
<sup>22</sup> Salah satu teknik yang akan digunakan dalam skripsi ini adalah principal component analysis(PCA).  
<sup>23</sup> Contoh, sebuah data dua dimensi diplot (x dan y) dengan scatterplot berikut :

<sup>24</sup>



Gambar 2.13: Scatterplot contoh data dua dimensi

<sup>25</sup> PCA berusaha membuat sebuah garis lurus yang bisa menjelaskan data tersebut. Setiap garis  
<sup>26</sup> lurus yang terbentuk merepresentasikan sebuah nilai Principal Component (PC) atau hubungan

1 antara variabel yang independen dan dependen. Semakin besar dimensi data, semakin banyak  
 2 *principal component*. PCA akan memprioritaskan nilai-nilai tersebut. *Principal component* pertama  
 3 menjelaskan data dengan nilai variansi terbanyak. Matrix covariance adalah sebuah deskripsi empiris  
 4 dari data yang diobservasi. Matrix covariance ini berisi nilai varian dan covarian pada variable da-  
 ta. *Eigenvector* dan *eigenvalue* pada sebuah matrix covariance ekuivalen dengan *principal component*.

6  
 7 MLLib menyediakan salah satu teknik dimensionality reduction PCA. Perhitungan *principal*  
 8 *component* dapat dilakukan dengan memanggil fungsi dari MLLib. Untuk melakukan perintah  
 9 perhitungan PC, pengguna perlu merepresentasikan data kedalam kelas RowMatrix yang terletak  
 10 pada package `mllib.linalg.distributed.RowMatrix`. Kemudian method computePrincipalCom-  
 11 ponents() dapat dipanggil melalui kelas tersebut. Method tersebut mengembalikan hasil dalam  
 12 sebuah object `mllib.linalg.Matrix` yang merepresentasikan sebuah dense matrix.

13

#### 14 Feature Extraction

15 Pada Machine Learning terdapat suatu istilah *Feature Engineering*. *Feature Engineering* adalah  
 16 sebuah proses transformasi data mentah menjadi sebuah representasi fitur-fitur sesuai dengan tujuan  
 17 model prediksi [7]. Hal ini bertujuan untuk meningkatkan nilai akurasi model. Salah satu caranya  
 18 adalah dengan *Feature extraction*. Salah satu algoritma *Feature extraction* yang dimiliki oleh MLLib  
 19 adalah TF-IDF.

20  
 21 Term frequency-inverse document frequency (TF-IDF) adalah metode vektorisasi fitur yang  
 22 sering digunakan pada *text mining*. TF-IDF bertujuan untuk merefleksikan pentingnya sebuah  
 23 term terhadap dokumen pada suatu corpus. Sebuah term dinotasikan dengan  $t$ , dokumen ( $d$ ), dan  
 24 corpus ( $D$ ). Term frequency  $TF(t, d)$  menunjukkan berapa kali term  $t$  itu muncul dalam dokumen  
 25  $d$ . Sedangkan document frequency  $DF(t, D)$  adalah jumlah dokumen yang mengandung term  $t$ .  
 26 Inverse document frequency (IDF) merupakan ukuran numerik dari seberapa banyak informasi yang  
 27 diberikan sebuah term. Berikut rumusnya:

28

$$IDF(t, D) = \log \frac{|D| + 1}{DF(t, D) + 1}$$

29 Notasi  $|D|$  menunjukkan jumlah dokumen pada corpus. Sedangkan untuk perhitungan TF-IDF  
 30 adalah dot product TF dan IDF.

$$TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D)$$

31 MLLib menyediakan fungsi TF-IDF terpisah menjadi TF dan IDF sendiri. Fungsi TF dan  
 32 IDF berada pada package `spark.mllib.feature`. TF menggunakan kelas `HashingTF` dan IDF  
 33 menggunakan kelas `IDF`. Ada hal yang sedikit berbeda pada implementasi TF di MLLib. Hal ini  
 34 karena TF di MLLib menggunakan fungsi hashing. Fitur pada data dipetakan kedalam sebuah  
 35 indeks term dengan menerapkan fungsi hash. Fungsi hash memungkinkan adanya *collision* pada  
 36 indeks term. Untuk mengurangi *collision*, pengguna dapat meningkatkan target dimensi fitur.

1

## 2 Collaborative Filtering dan Recommendation

3 Sering kali pada website belanja online dapat ditemukan barang-barang rekomendasi berdasarkan  
4 pencarian atau pembelian barang yang dilakukan. Contohnya ketika ada pengguna membeli produk  
5 smartphone di Tokopedia, tokopedia akan menampilkan rekomendasi barang seperti aksesoris-  
6 aksesoris hp sesuai yang dibeli pengguna. Hal ini adalah basis dari recommender system yang  
7 biasanya menggunakan teknik *collaborative filtering*

8 *Collaborative filtering* adalah suatu teknik agar sistem dapat merekomendasikan sesuatu dimana  
9 penilaian(*rating*) dan reaksi pengguna terhadap suatu produk digunakan untuk merekomendasikan  
10 produk yang baru. *Collaborative filtering* hanya menggunakan data berupa daftar interaksi produk  
11 dengan pengguna(contohnya rating suatu produk). Dengan ini, algoritma Collaborative filtering  
12 dapat mempelajari suatu produk yang kiranya serupa dengan produk lainnya dan sekaligus pengguna  
13 yang serupa dengan pengguna lainnya [9].

14

15 Salah satu algoritma yang dimiliki oleh MLlib adalah algoritma Alternating Least Squares (ALS).  
16 Implementasi Algoritma ALS pada MLlib dilakukan dengan menyatukan hubungan pengguna,  
17 produk, dan rating pada suatu kelas yaitu **Rating**. Dari data-data hubungan pengguna dan produk,  
18 maka suatu model dapat dihasilkan dengan pelatihan. Pelatihan dilakukan dengan menggunakan  
19 objek ALS yang disediakan MLlib pada package spark.mllib.recommendation. Dengan pelatihan  
20 data, maka didapatkan sebuah model **MatrixFactorizationModel** yang bisa digunakan untuk  
21 memprediksi pada data yang tidak memiliki rating produk.

## 22 2.4 Scala

23 Scala adalah bahasa pemrograman multi-paradigma modern yang dirancang untuk mengekspresikan  
24 pola pemrograman yang umum secara ringkas, elegan, dan aman dalam pengetikan. Scala pertama  
25 kali dikandung pada tahun 2001 di Ecole Polytechnique Federale de Lausanne oleh Martin Odersky,  
26 yang merupakan co-creator dari bahasa pemrograman Generik Java, javac, dan EPFL's Funnel.  
27 Peluncuran publik pertama Scala terjadi pada tahun 2004 yang diikuti oleh versi 2.0 pada bulan  
28 Maret 2006.

29

30 Scala dapat mengintegrasikan fitur berorientasi objek dan bahasa fungsional. Scala adalah  
31 bahasa berorientasi objek murni yang mempunyai arti bahwa setiap nilai adalah obyek. Jenis dan  
32 perilaku objek dijelaskan oleh class dan trait. Kelas dapat di-extend dengan subclassing. Scala juga  
33 merupakan bahasa fungsional yang memiliki arti bahwa setiap fungsi adalah sebuah nilai. Scala  
34 menyediakan sintaks yang ringan untuk mendefinisikan fungsi anonim, mendukung fungsi orde  
35 tinggi, memungkinkan fungsi bersarang, dan mendukung currying. Kelas kasus Scala dan dukungan  
36 bawaannya untuk model aljabar model pencocokan pola yang digunakan dalam banyak bahasa  
37 pemrograman fungsional. Objek Singleton memberikan cara mudah untuk mengelompokkan fungsi  
38 yang bukan anggota kelas.

39

1 Perintah 'scalac' digunakan untuk mengkompilasi program Scala dan akan menghasilkan beberapa  
 2 file kelas di direktori saat ini. Salah satunya akan disebut file **.class**. Ini adalah bytecode  
 3 yang akan berjalan di Java Virtual Machine (JVM) dengan menggunakan perintah 'scala'.

4  
 5 Berikut ini adalah konvensi dalam pemrograman Scala :

- 6
- 7 Case Sensitivity Scala peka terhadap huruf besar, yang berarti pengenal Halo dan halo akan  
 memiliki arti yang berbeda dalam Scala. Nama Class - Untuk semua nama kelas, huruf  
 pertama harus menggunakan huruf kapital. Jika beberapa kata digunakan untuk membentuk  
 nama kelas, masing-masing kata dalam huruf pertama harus menggunakan huruf kapital.
  - 8 Nama Method - Semua nama method harus dimulai dengan huruf kecil. Jika beberapa kata  
 digunakan untuk membentuk nama method, maka setiap huruf pertama kata dalam harus  
 menggunakan huruf kapital.
  - 9 Nama File - Nama file program harus sama persis dengan nama objek. Saat menyimpan file,  
 Pengguna harus menyimpannya dengan menggunakan nama objek (Ingat Scala adalah *case  
 sensitive*) dan tambahkan '.scala' ke akhir nama. (Jika nama file dan nama objek tidak sesuai  
 dengan program pengguna maka file tidak akan dikompilasi).
  - 10 def main (args: Array [String]) - Pengolahan program Scala dimulai dari method  
 main() yang merupakan bagian wajib setiap Program Scala.

11  
 12 **2.4.1 Tipe Data**

13 Tipe data numerik di Scala sama seperti pada java yaitu Byte, Char, Double, Float, Int, Long, and  
 14 Short. Perbedaanya, tipe data numerik di Scala adalah sebuah object. Pada tabel 2.1, ukuran bit  
 15 dan jarak tipe data hampir sama pada primitive java string.

16

Value Type	Range
Byte	8-bit signed two's complement integer (-2 <sup>7</sup> to 2 <sup>7</sup> - 1, inclusive)
Short	16-bit signed two's complement integer (-2 <sup>15</sup> to 2 <sup>15</sup> - 1, inclusive)
Int	32-bit signed two's complement integer (-2 <sup>31</sup> to 2 <sup>31</sup> - 1, inclusive)
Long	64-bit signed two's complement integer (-2 <sup>63</sup> to 2 <sup>63</sup> - 1 , inclusive)
Char	16-bit unsigned Unicode character (0 to 2 <sup>16</sup> - 1, inclusive)
String	a sequence of Chars
Float	32-bit IEEE 754 single-precision float
Double	64-bit IEEE 754 double-precision float
Boolean	true or false

Tabel 2.1: Tipe data pada bahasa pemrograman Scala [1]

24 Tipe data String Scala sama seperti Java String. Hal ini karena String scala berada pada  
 25 package Java.lang. Sedangkan tipe data lainnya berada pada package scala. Tipe data ini sudah di  
 26 import otomatis ke setiap source file scala. Sehingga pengguna dapat langsung memanggil nama  
 27 tipe data(Contoh Boolean, String, Double, dll).

### 2.4.2 Variabel

suatu expressions adalah pernyataan atau argumen yang dapat dikomputasi. Contoh :

1 + 1

Ekspressions dapat dikembalikan dengan println. Contoh :

```
5 println(1) // 1
6 println(1 + 1) // 2
7 println("Hello!") // Hello!
```

ekspressions atau pernyataan seperti diatas dapat disimpan dalam sebuah variable ada dua jenis variable di Scala yaitu val dan var. Setelah val diinisialisasi, val tidak dapat diisi kembali artinya isi dari val tidak dapat diubah.

```
11 val x = 1 + 1
12 val msg = "Hello, world!"
```

sedangkan var merupakan variabel yang nilainya bisa diubah setelah diinisialisasi dan ini disebut variabel yang bisa berubah(Mutable Variable). Contoh:

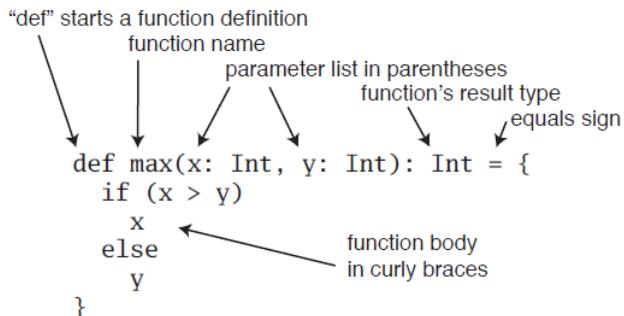
```
15 var x = 1 + 1
16 x = 3
17 println(x) // 3
```

atau secara eksplisit dinyatakan seperti berikut:

```
19 var x: Int = 1 + 1
```

### 2.4.3 Method dan Fungsi

*Function* (Fungsi) adalah sekumpulan argumen yang melakukan suatu tugas untuk menghasilkan suatu keluaran. *Function* adalah sebuah expressions yang menerima parameter dan dapat mengembalikan suatu nilai. Untuk mendeklarasikan function, diawali dengan def diikuti nama fungsi itu sendiri.



Gambar 2.14: Bentuk umum *function* Scala

Pada gambar 2.14 nama function nya adalah max. Setelah menulis nama, pengguna dapat menulis parameter. Parameter dapat dipisahkan dengan koma jika ada lebih dari satu parameter.

1 Setiap parameter harus didefinisikan tipenya. Setelah parameter adalah *result type* dari *function*.  
 2 Pada gambar 2.14 dituliskan : `Int` artinya kembalian *function* adalah bertipe Integer. Pada bagian  
 3 body *function* dapat diisi dengan satu atau beberapa *expression*. Untuk memanggil *function*,  
 4 pengguna dapat mengetik nama dan isi parameter seperti berikut:

5 `max(2,1)`

6 Function yang tidak memiliki parameter dan memiliki hasil yang tidak ada maknanya didefini-  
 7 sikan seperti berikut:

8 `def count() = println(1+1)`

9 Tanda kurung yang kosong mengartikan function ini tidak menerima parameter. Sedangkan  
 10 tipe kembalian Unit mengartikan kembalian yang tidak ada maknanya atau jika pada java adalah  
 11 kembalian tipe void. function dapat dituliskan dalam bentuk lain, salah satunya function literal.  
 12 function literal adalah sebuah fungsi tanpa nama di scala yang menggunakan syntax khusus. function  
 13 literal dapat dituliskan seperti contoh,

14 `(x: Int, y: Int) => x + y.`

15 Selain *function*, scala juga memiliki istilah *method*. *Method* merupakan suatu *function* yang  
 16 menjadi anggota dari suatu kelas, interface java, trait scala, dan lain-lain. Jadi, cara mendeklarasikan  
 17 method sama saja dengan *function*.

#### 18 2.4.4 Iterasi dan Percabangan

19 Iterasi atau *loop* dapat dideklarasikan dengan `while`. Penulisannya mirip pada Java. berikut  
 20 contohnya :

```
21 var i = 0
22 while (i < 5) {
23   println("Hello")
24   i += 1
25 }
```

26 expresi didalam tanda kurung kurawal '' pada while akan dijalankan berulang kali hingga kondisi  
 27 while terpenuhi. Kondisi while adalah yang berada pada tanda kurung '()' yaitu `i<5`. Di scala,  
 28 kondisi while harus didefinisikan dengan sebuah expression yang mengembalikan nilai Boolean.  
 29 Sama seperti pada percabangan (`if`), kondisi didalam kurung `if` harus mengembalikan nilai Boolean.  
 30 seperti pada contoh berikut :

```
31 if (i != 0)
32   print(" ")
```

33 Deklarasi `if` di atas tidak menggunakan kurung kurawal karena kurung kurawal ini *optional*  
 34 untuk pengguna. Dengan demikian `if` juga dapat didefinisikan sebagai berikut:

```
1 if (i != 0){  
2   print(" ")  
3 } else{  
4   print("else")  
5 }
```

6        Jika kondisi **if** bernilai **false** maka expression di dalam **else** akan dieksekusi. Iterasi juga  
7 dapat menggunakan **foreach** dan **for** pada Scala. **foreach** adalah suatu method pada array scala.  
8 Sebagai contoh, sebuah array scala dinamai **args** dan dilakukan iterasi pada array tersebut,  
9

```
10 args.foreach(arg => println(arg))
```

11        method **foreach** ini memberikan sebuah function yaitu function literal dengan sebuah parameter  
12 **arg**. Sedangkan isi dari function tersebut adalah **println(arg)**. Untuk menggunakan **for** di scala,  
13 digunakan perintah berikut:

```
14 for (arg <- args)  
15   println(arg)
```

16        Di dalam tanda kurung setelah **for** memiliki makna yaitu untuk setiap elemen dari array **args**,  
17 sebuah **val arg** diinisialisasikan dengan *value element*.

#### 18 2.4.5 Objek

19 Objek (Object) adalah instansi tunggal dari definisi kelas. Objek dapat dianggap sebagai sesuatu  
20 instansi yang tunggal pada class itu sendiri. Untuk mendefinisikan objek, digunakan kata kunci  
21 **Object**.

```
22 object IdFactory {  
23   private var counter = 0  
24   def create(): Int = {  
25     counter += 1  
26     counter  
27   }  
28 }
```

29        Untuk dapat mengakses sebuah objek, caranya cukup dengan mengacu pada nama objek tersebut.

```
30 val newId: Int = IdFactory.create()  
31 println(newId) // 1  
32 val newerId: Int = IdFactory.create()  
33 println(newerId) // 2
```

### 1    2.4.6   Method Main

2    *Main Method* adalah *entry point* sebuah program. Java Virtual Machine(JVM) membutuhkan *Main*  
3    *Method* untuk diberi nama main dan mengambil satu argumen, sebuah array dari string. Dengan  
4    menggunakan sebuah *Object*, *Main Method* sebagai berikut:

```
5 object Main {  
6     def main(args: Array[String]): Unit =  
7         println("Hello, Scala developer!")  
8 }
```

9       Method ini akan dijalankan pertama kali ketika file atau program di-*run*.

1

## BAB 3

2

### STUDI DAN EKSPLORASI APACHE SPARK MLLIB

3

#### 3.1 Instalasi Apache Spark

4 Pada bagian ini, akan dijelaskan tahap-tahap untuk melakukan instalasi Apache Spark. Apache  
5 Spark yang akan digunakan adalah Apache Spark versi 2.2.0. Spark dapat berjalan diatas sistem  
6 operasi Windows dan UNIX systems (Contoh Linux, Mac OS). Penulis akan menjelaskan langkah-  
7 langkah instalasi Spark pada kedua sistem operasi. Sebelum melakukan instalasi Apache Spark,  
8 ada beberapa persyaratan minimum yang harus dipenuhi diantaranya adalah :

- 9     • Untuk menjalankan secara lokal pada satu mesin(komputer), pengguna perlu menginstal java  
10    di system PATH, atau JAVA\_HOME environment variable yang menunjuk ke instalasi Java.
- 11    • Spark berjalan pada versi Java 8+, Python 2.7+\3.4+ dan R 3.1+.
- 12    • Untuk API Scala, Spark 2.2.0 menggunakan versi Scala 2.11.
- 13    • Diharuskan untuk menggunakan versi Scala yang compatible (2.11.x).
- 14    • Perhatikan bahwa dukungan untuk Java 7, Python 2.6 dan versi Hadoop lama sebelum 2.6.5  
15    dihapus seperti Spark 2.2.0.
- 16    • Perhatikan bahwa dukungan untuk Scala 2.10 sudah tidak berlaku lagi seperti Spark 2.1.0,  
17    dan dapat dihapus di Spark 2.2.0.

18    Jika seluruh persyaratan di atas telah terpenuhi maka instalasi dapat dilakukan. Berikut  
19    langkah-langkah untuk instalasi Apache Spark pada sistem operasi Linux (Ubuntu).

- 20    • Menginstal Java Update index package pada terminal dengan mengetik :

21        sudo apt-get update

22        Install Java JDK dengan perintah berikut:

23        sudo apt-get install default-jdk

- 24    • Menginstall Scala

25        Untuk menginstall Scala, gunakan perintah berikut:

1        sudo apt-get install scala

- 2        • Menginstall Spark

3        Download source file Apache Spark pada halaman website berikut :

4        <https://spark.apache.org/downloads.html>

5        Ekstrak file yang telah di-download dengan perintah:

6        tar xvf spark-2.0.2-bin-hadoop2.7.tgz

7        Mendefinisikan environment variables dibawah ini pada file .bashrc :

8        export YARN\_CONF\_DIR=\$HADOOP\_HOME/etc/hadoop  
9        export SPARK\_HOME=<extracted\_spark\_package>  
10      export PATH=\$PATH:\$SPARK\_HOME/bin

11      Selain instalasi pada sistem operasi Linux, penulis juga melakukan instalasi pada sistem operasi  
12     Windows untuk membuat lingkungan Spark yang *standalone*. Instalasi pada sistem operasi Windows  
13     dilakukan dengan langkah-langkah berikut :

- 14      1. Download dan install Java versi 8  
15      2. Buat environment variables Java dengan nama

16      JAVA\_HOME

17      dan isi variable value dengan path direktori java

18      'C:\Program Files\Java\jdk1.8.0\_141\'

- 19      3. Download source file Apache Spark di website Apache Spark <https://spark.apache.org/downloads.html>.  
20      File yang telah di-download akan bertipe .tgz. Contoh: spark-2.2.0-bin-hadoop2.7.tgz  
21      4. Ekstrak source file Apache Spark pada local disk  
22      5. Buatlah environment variables Apache Spark dengan nama

23      SPARK\_HOME

24      dan isi variable value dengan path direktori Spark, contoh :

25      'C:\SPARK\BIN'

## 1    3.2    Eksplorasi menggunakan Spark Shell

2    Dalam melakukan eksplorasi, penulis menggunakan Spark Shell untuk mempelajari sintaks sederhana dan mempelajari fungsi-fungsi dan API yang tersedia pada Spark. Spark Shell menyediakan media yang simpel untuk mempelajari API Spark sekaligus sebagai alat yang dapat menganalisa data secara interaktif. Spark Shell tersedia dengan bahasa Scala (yang berjalan di JVM(Java Virtual Machine)) atau Python. Berikut ini akan dijelaskan cara menjalankan Spark Shell dan beberapa contoh penggunaanya.

8

9    Pengguna dapat menjalankan Spark shell dengan CLI seperti *command prompt* pada Windows  
10 atau menggunakan terminal pada Ubuntu. Spark shell berada di dalam direktori `/bin` Spark

11    `./bin/spark-shell`

12    Tampilan CLI akan berubah menjadi tampilan User Interface Spark Shell(Gambar 3.1)

13

```
Welcome to
    _/\_ _\ \_/\_ /_/\_ 
  / \ \_ .\_/\_,/_/\_/\_ \_ version 2.2.0
  \_/
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_141)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Gambar 3.1: Tampilan Spark-Shell

14    Pemanggilan fungsi-fungsi dasar seperti membuat RDD dapat dilakukan dengan menggunakan  
15 SparkSession atau SparkContext. Di Spark shell, SparkSession disimpan dengan nama variable  
16 `spark` sedangkan nama variable SparkContext adalah `sc`. Berikut adalah perintah untuk membuat  
17 RDD baru dari sebuah file teks `README.md` di direktori source Spark:

18    `scala> val textFile = spark.read.textFile("README.md")`  
19    `textFile: org.apache.spark.sql.Dataset[String] = [value: string]`

20    Setelah membuat RDD, pengguna dapat memanggil beberapa Action atau melakukan Transformations pada RDD untuk mendapatkan RDD yang baru. Berikut adalah contoh pemanggilan  
21 Action:  
22

23    `scala> textFile.count()`  
24    `res0: Long = 126`  
25  
26    `scala> textFile.first()`  
27    `res1: String = # Apache Spark`

1 Untuk memanggil perintah Transformation (Contoh: `Filter()`) dilakukan dengan cara berikut:

```
2 scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
3 linesWithSpark: org.apache.spark.sql.Dataset[String] = [value: string]
```

4 Transformation dan Action dapat dipanggil dengan cara digabungkan bersama. Berikut adalah  
5 caranya:

```
6 scala> textFile.filter(line => line.contains("Spark")).count()
7 res3: Long = 15
```

### 8 3.3 Pemanggilan Fungsi-Fungsi pada library MLLib

9 MLlib memiliki berbagai macam fungsi dan disediakan dalam bentuk API. Pengguna dapat  
10 memanggil fungsi dengan cara tertentu sesuai dengan dokumentasi Spark yang tersedia di ha-  
11 laman website <https://spark.apache.org/docs/2.2.0/api.html>. Package MLlib berada pada  
12 `org.apache.spark.mllib`.

13  
14 Penulis menggunakan metode *Self-Contained Application* untuk melakukan studi eksplorasi  
15 pemanggilan fungsi pada *library* MLlib. Spark versi 2.2.0 dirancang untuk dapat berjalan dengan  
16 Scala versi 2.11, maka versi Scala yang digunakan adalah 2.11.11. Untuk membuat Self-Contained  
17 Application Spark dengan menggunakan Scala, digunakan 2 cara yaitu:

- 18 • Dengan menggunakan Scala Built Tool(SBT).
- 19 • Dengan menggunakan Maven Project.

20 Cara membuat aplikasi Spark menggunakan Maven Project yaitu dengan menambahkan de-  
21 pendency Maven yang dibutuhkan untuk memanggil fungsi Spark pada folder proyek. *Dependency*  
22 proyek maven berada pada file `pom.xml` di dalam direktori proyek pengguna. Dependency yang  
23 diperlukan adalah dependency spark-core:

```
24 <dependency>
25   <groupId>org.apache.spark</groupId>
26   <artifactId>spark-core_2.11</artifactId>
27   <version>2.2.0</version>
28   <scope>provided</scope>
29 </dependency>
```

30 dan untuk menggunakan *library* Mllib, pengguna perlu menambahkan dependency Maven MLlib:

```
31 <dependency>
32   <groupId>org.apache.spark</groupId>
33   <artifactId>spark-mllib_2.11</artifactId>
34   <version>2.2.0</version>
35   <scope>provided</scope>
36 </dependency>
```

1 Membuat Self-Contained Applications menggunakan Scala Built Tool(SBT) dapat dilakukan  
2 pengguna dengan menginstall terlebih dahulu SBT pada perangkat pengguna. SBT dapat diunduh  
3 di halaman website <https://www.scala-sbt.org/download.html>. Kemudian buat environment  
4 variable pada sistem operasi Windows untuk SBT. Sistem operasi linux dapat menggunakan perintah  
5 berikut:

```
6 echo "deb https://dl.bintray.com/sbt/debian /"  
7   | sudo tee -a /etc/apt/sources.list.d/sbt.list  
8 sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80  
9   --recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823  
10 sudo apt-get update  
11 sudo apt-get install sbt
```

12 Langkah-langkah dalam membuat *project* SBT yaitu buat terlebih dahulu folder *project* SBT dengan  
13 perintah berikut:

```
14 mkdir myProject  
15 kemudian masuk ke folder tersebut dan jalankan perintah berikut pada command line:
```

```
16 sbt
```

17 SBT akan otomatis menginisialisasi file-file configurasi SBT. Setelah selesai, sbt shell akan langsung  
18 berjalan pada CLI. Pengguna dapat mengatur konfigurasi sendiri(Nama *project*, Versi program, dan  
19 lain-lain) seperti berikut:

```
20 >set name := "namaProject"  
21 >set version := "1.0"  
22 >set scalaVersion := "2.11.8"
```

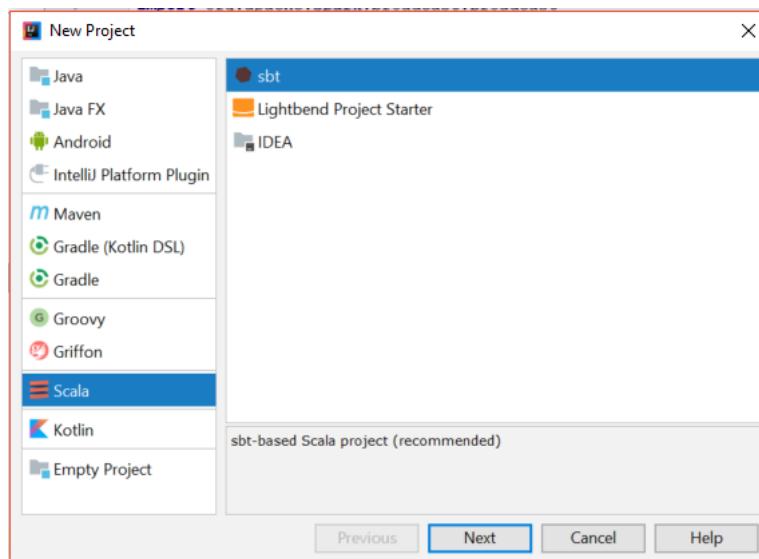
23 untuk menambahkan *dependency* spark-core dan spark-mllib pada project SBT, gunakan perin-  
24 tah berikut:

```
25 >set libraryDependencies += "org.apache.spark" % "spark-core_2.11" % "2.2.0"  
26 >set libraryDependencies += "org.apache.spark" % "spark-mllib_2.11" % "2.2.0"
```

27 Untuk menjalankan project Spark, digunakan perintah:

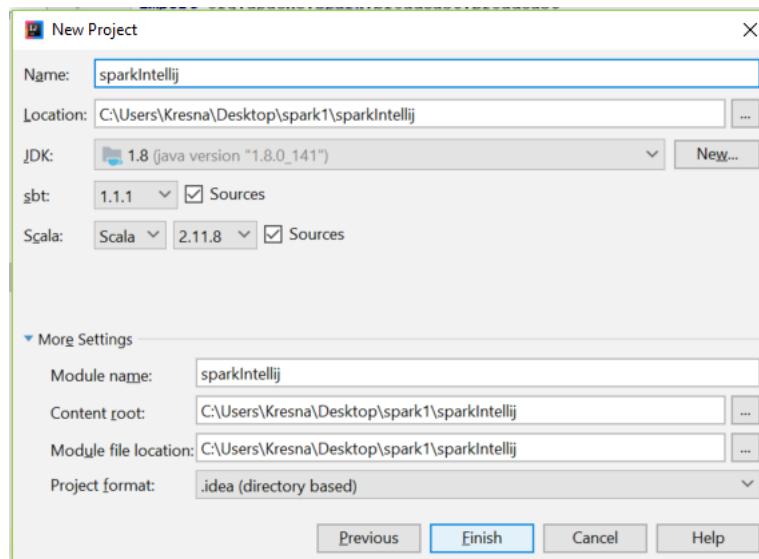
```
28 >run
```

29 Penulis menggunakan sebuah IDE yaitu IntelliJ dalam membuat program. Untuk membuat  
30 project pada IntelliJ pertama-tama install terlebih dahulu IntelliJ pada perangkat komputer. Buka  
31 IDE IntelliJ dan buat *project* baru. Saat membuat *project* baru, pilih sbt-based Scala Project seperti  
32 pada Gambar 3.2



Gambar 3.2: Membuat *project SBT* pada IntelliJ

1 Kemudian atur konfigurasi *project* meliputi nama project, versi scala, dan sebagainya (Gambar  
2 3.3).



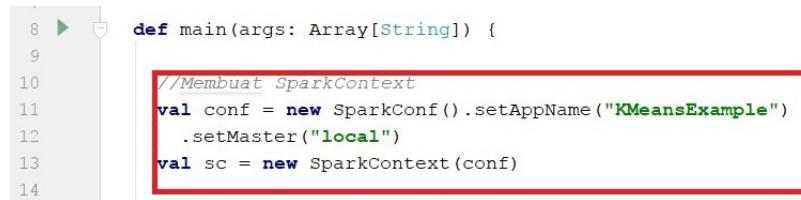
Gambar 3.3: Mengatur konfigurasi pada IntelliJ

3 Spark menyediakan dua library yang berbeda pada MLlib. Library pertama memberikan API  
4 berupa RDD-based sedangkan library kedua menggunakan DataFrame-based API. Sehingga package  
5 yang digunakan masing-masing pun berbeda. Untuk DataFrame-based API diletakan pada package  
6 **spark.ml**. Sedangkan untuk RDD-based API berada pada package **spark.mllib**, package inilah  
7 yang akan digunakan penulis dalam melakukan eksperimen.

8  
9 Eksperimen ini dilakukan dengan menulis kode program sebuah aplikasi Spark yang memanggil  
10 library MLlib. Hal pertama yang harus dilakukan untuk membuat program Spark adalah membuat  
11 objek `SparkContext`(Gambar 3.4). Untuk membuat `SparkContext` pengguna harus melakukan  
12 import pada package `Spark` terlebih dahulu. Inisialisasi `SparkConf` terlebih dahulu untuk mengatur

- 1 konfigurasi SparkContext. SparkConf memiliki parameter yang harus dikonfigurasi seperti master  
 2 dari cluster dan nama aplikasi. Eksperimen menggunakan master lokal untuk menjalankan pada  
 3 komputer tunggal.

4



```

8  def main(args: Array[String]) {
9
10 //Membuat SparkContext
11 val conf = new SparkConf().setAppName("KMeansExample")
12   .setMaster("local")
13 val sc = new SparkContext(conf)
14

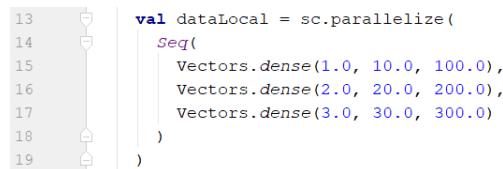
```

Gambar 3.4: Membuat SparkContext

### 5 3.3.1 Summary Statistics

- 6 Fungsi Statistik pada MLlib dapat digunakan untuk perhitungan statistika seperti menghitung  
 7 rata-rata, mencari nilai maksimum dan minimum, menghitung standar deviasi dari setiap fitur data,  
 8 dan lain-lain. MLlib menyediakan kelas untuk melakukan perhitungan statistik dibawah package  
 9 spark.mllib.stat. Kelas (atau objek) yang digunakan adalah Statistics dan MultivariateStatistical-  
 10 Summary. Sehingga perlu meng-import terlebih dulu kelas atau objek tersebut.

11



```

13 val dataLocal = sc.parallelize(
14   Seq(
15     Vectors.dense(1.0, 10.0, 100.0),
16     Vectors.dense(2.0, 20.0, 200.0),
17     Vectors.dense(3.0, 30.0, 300.0)
18   )
19 )

```

Gambar 3.5: Membuat data masukkan untuk Statistic

- 12 Data yang digunakan adalah data dummy yang dibuat pada kode program(Gambar 3.5). Untuk  
 13 melakukan perhitungan Summary Statistics, MLlib menyediakan method colstat pada objek  
 14 Statistic (Lihat Gambar 3.6). Method tersebut akan mengembalikan sebuah objek MultivariateS-  
 15 tatisticalSummary. MultivariateStatisticalSummary dapat memberikan nilai perhitungan statistik  
 16 pada data yang diberikan, pengguna dapat memanggil method yang terkait yaitu meliputi:

17

- 18 1. *mean*, method ini akan mengembalikan nilai rata-rata pada setiap kolom atau fitur pada data.
- 19 2. *variance*, method ini akan mengembalikan nilai variansi atau standar deviasi pada setiap  
 20 kolom atau fitur pada data.
- 21 3. *min*, method ini akan mengembalikan nilai paling kecil(minimum) pada setiap kolom atau  
 22 fitur pada data.
- 23 4. *max*, method ini akan mengembalikan nilai paling besar(maximum) pada setiap kolom atau  
 24 fitur pada data.
- 25 5. *numNonZero*, method ini akan mengembalikan jumlah element yang nilainya tidak nol pada  
 26 setiap kolom atau fitur pada data.

6. *Count*, method ini mengembalikan jumlah objek atau baris data.

```

23   val summary: MultivariateStatisticalSummary = Statistics.colStats(dataLocal)
24     println("Mean :" +summary.mean)
25     println("Variance :" +summary.variance)
26     println("Max :" +summary.max)
27     println("Min :" +summary.min)
28     println("numNonZero :" +summary.numNonzeros)
29     println("Count :" +summary.count)

```

Gambar 3.6: Pemanggilan method colStat dan fungsi-fungsinya

- Hasil keluaran kode program berupa vector yang dapat ditampilkan ke console (Lihat Gambar 3.7).

4

```

Run  SummaryStatistic
18/04/20 08:45:40  INFO DAGScheduler:
  Mean :[2.0,20.0,200.0]
  Variance :[1.0,100.0,10000.0]
  Max :[3.0,30.0,300.0]
  Min :[1.0,10.0,100.0]
  numNonZero :[3.0,3.0,3.0]
  Count :3

```

Gambar 3.7: Hasil output percobaan Summary Statistic

### 3.3.2 Naive Bayes

- Algoritma Naive bayes adalah salah satu algoritma Classification. Pelatihan Naive Bayes pada MLLib menggunakan kelas pada package `spark.mllib.classification` yaitu kelas `NaiveBayes` dan `NaiveBayesModel`. Penulis hanya perlu menggunakan objek dari kelas tersebut untuk melakukan pelatihan data. Gunakan method `train()` milik `NaiveBayes`. Method `train` menerima parameter `RDD LabeledPoint`. Data yang akan digunakan pada pelatihan diinisialisasi didalam kode program untuk eksperimen. Sehingga perlu melakukan *import* tipe data `LabeledPoint`. Data disimpan dalam variable `rawData`(Gambar 3.8).

13

```

11   // Generate data
12   val rawData = Array(
13     LabeledPoint(1.0, Vectors.dense(0.0, 6.0, 4.0)),
14     LabeledPoint(1.0, Vectors.dense(1.0, 1.0, 6.0)),
15     LabeledPoint(2.0, Vectors.dense(2.0, 2.0, 5.0)),
16     LabeledPoint(3.0, Vectors.dense(0.0, 2.0, 5.0)),
17     LabeledPoint(2.0, Vectors.dense(3.0, 8.0, 4.0)),
18     LabeledPoint(1.0, Vectors.dense(0.0, 5.0, 0.0))
19   )

```

Gambar 3.8: Membuat data input untuk Naive Bayes

- Dari data masukkan, akan dibagi menjadi 60% untuk pelatihan dan 40% untuk pengujian(Gambar 3.9).

16

```

23   // Split data training (60%) and test (40%).
24   val Array(training, test) = data.randomSplit(Array(0.6, 0.4))

```

Gambar 3.9: Pembagian data pelatihan dan pengujian

- 1     Method train dipanggil dengan parameter data pelatihan, lambda, jenis algoritma naive bayes  
 2     yaitu *multinomial*. Method ini mengembalikan sebuah model dalam sebuah objek NaiveBayesModel,  
 3     yang dapat digunakan untuk melakukan prediksi kelas sesuai model hasil pelatihan.

4

```

val model = NaiveBayes.train(training, lambda = 1.0, modelType = "multinomial")

//Testing
val predictionAndLabel = test.map(p => (model.predict(p.features), p.label))

val accuracy = 1.0 * predictionAndLabel.filter(x => x._1 == x._2).count() / test.count()
println(s"Nilai akurasi pelatihan: $accuracy")

```

Gambar 3.10: Pemanggilan method train untuk pelatihan data

- 5     Dalam melakukan testing, method `predict()` akan dipanggil untuk memprediksi fitur pada data.  
 6     Hal ini dilakukan dengan menggunakan memanggil method map pada RDD. Fungsi yang dijalankan  
 7     pada operasi map itu adalah memanggil method predict pada setiap fitur dan memasangkannya  
 8     dengan label asli data. Sehingga hasilnya data RDD menjadi sebuah pasangan kelas hasil prediksi  
 9     dan kelas aslinya.

10

- 11    Menghitung akurasi pada pengujian dilakukan dengan menghitung terlebih dulu jumlah data  
 12    yang kelas hasil prediksi nya sama dengan data aslinya. Kemudian dibagi dengan total jumlah data  
 13    yang dipakai pada pengujian.

14

```

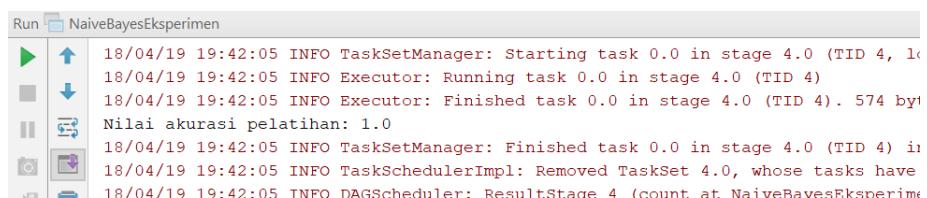
34 // Save and load model
35 model.save(sc, "E:/Output/NaiveBayes")
36 val sameModel = NaiveBayesModel.load(sc, "E:/Output/NaiveBayes")

```

Gambar 3.11: Menyimpan model hasil pelatihan

- 15    Model dapat disimpan pada *filesystem* menggunakan method `save()` pada model. Hal ini  
 16    dilakukan agar model dapat digunakan kembali. Model naive bayes ini juga dapat menampilkan  
 17    daftar kelas-kelas yang diprediksi berupa array.

18



```

Run NaiveBayesEksperimen
18/04/19 19:42:05 INFO TaskSetManager: Starting task 0.0 in stage 4.0 (TID 4, local
18/04/19 19:42:05 INFO Executor: Running task 0.0 in stage 4.0 (TID 4)
18/04/19 19:42:05 INFO Executor: Finished task 0.0 in stage 4.0 (TID 4). 574 bytes
Nilai akurasi pelatihan: 1.0
18/04/19 19:42:05 INFO TaskSetManager: Finished task 0.0 in stage 4.0 (TID 4)
18/04/19 19:42:05 INFO TaskSchedulerImpl: Removed TaskSet 4.0, whose tasks have
18/04/19 19:42:05 INFO DAGScheduler: ResultStage 4 (count at NaiveBayesEksperimen)

```

Gambar 3.12: Hasil Naive Bayes dikeluarkan pada console

```

Run [ ] NaiveBayesEksperimen
  18/04/19 19:42:10 INFO TaskSchedulerImpl: Removed TaskSet 11.0,
  18/04/19 19:42:10 INFO DAGScheduler: ResultStage 11 (foreach at
  18/04/19 19:42:10 INFO BlockManagerInfo: Removed broadcast_6_pi
  18/04/19 19:42:10 INFO DAGScheduler: Job 9 finished: foreach at
  1.0
  3.0
  2.0
  3.0
  2.0
  1.0
  1.0
  2.0
  2.0
  18/04/19 19:42:11 INFO BlockManagerInfo: Removed broadcast_7_pi
  18/04/19 19:42:11 INFO ContextCleaner: Cleaned accumulator 175

```

Gambar 3.13: Hasil Naive Bayes dikeluarkan pada console

- 1 Gambar 3.13 dan Gambar 3.12 menunjukan hasil keluaran nilai akurasi dan hasil prediksi kelas
- 2 pada suatu contoh data tanpa label.

### 3 3.3.3 Principal Component Analysis (PCA)

- 4 Principal Component Analysis(PCA) merupakan salah satu teknik untuk mengurangi dimensionalitas pada data yang sering digunakan. Perhitungan PC disediakan MLLib pada kelas RowMatrix pada package `spark.mllib.linalg.distributed.RowMatrix`. RowMatrix merupakan salah satu jenis dalam tipe data RDD yaitu Distributed matrix. RowMatrix dapat dibuat dengan sekumpulan vektor. Ukuran data matrix harus diketahui sehingga tidak ada data yang melebihi ukuran matrix.
- 5 Untuk melakukan perhitungan PC, pengguna dapat memanggil `computePrincipalComponents()` pada RowMatrix dan mengisi parameter yaitu nilai teratas dari PC (Lihat gambar 3.15).

- 11 Hal pertama yang dilakukan untuk melakukan eksperimen ini adalah menyiapkan data dalam bentuk RDD. Hal ini dilakukan dengan menginisialisasi vektor dan mengubahnya dalam bentuk RDD dengan method `parallelize` (Lihat gambar 3.14).

```

19
20
21
22
23
24
25
26
27
val data = Array(
  Vectors.sparse(5, Seq((1, 1.0), (3, 7.0))),
  Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),
  Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0),
  Vectors.dense(5.0, 2.0, 0.0, 5.0, 8.0),
  Vectors.dense(5.0, 0.0, 1.0, 7.0, 9.0),
  Vectors.dense(3.0, 1.0, 0.0, 5.0, 7.0))
val rows = sc.parallelize(data)

```

Gambar 3.14: Membuat data input untuk eksperimen PCA

```

39 // top 4 principal components.
40 val mat: RowMatrix = new RowMatrix(rows)
41 val pc: Matrix = mat.computePrincipalComponents(4)

```

Gambar 3.15: Menjalankan komputasi perhitungan PC

```
// Project the rows to the linear space spanned by the top 4 principal components.
val projected: RowMatrix = mat.multiply(pc)
```

Gambar 3.16: Proyeksi data dengan nilai PC

- 1 Selain menggunakan RowMatrix, pengguna juga dapat menggunakan kelas PCA yang disediakan  
 2 MLlib pada package `spark.mllib.feature.PCA`. Kelas ini perlu diinisialisasi terlebih dulu dengan  
 3 parameter nilai PC. Hal ini dikarenakan objek nya belum didefinisikan oleh library MLlib. Kelas  
 4 ini memiliki satu method yaitu `fit()` dengan parameter yaitu RDD vector(Lihat gambar 3.18).  
 5 Method fit akan mengembalikan sebuah objek PCAModel yang kemudian dapat mentransform fitur  
 6 data sebelumnya dengan hasil PC yang dilakukan. Pengguna mencoba menggunakan sebuah data  
 7 LabeledPoint untuk melakukan perhitungan PC pada fitur data(Gambar 3.17).

8



```
51 val data2: RDD[LabeledPoint] = sc.parallelize(Seq(
52   new LabeledPoint(0, Vectors.dense(1, 0, 0, 0, 1)),
53   new LabeledPoint(1, Vectors.dense(1, 1, 0, 1, 0)),
54   new LabeledPoint(1, Vectors.dense(1, 1, 0, 0, 0)),
55   new LabeledPoint(0, Vectors.dense(1, 0, 0, 0, 0)),
56   new LabeledPoint(1, Vectors.dense(1, 1, 0, 0, 0)))
```

Gambar 3.17: Proyeksi data dengan nilai PC

```
// Compute the top 5 principal components.
val pca = new PCA(5).fit(data2.map(_.features))
```

Gambar 3.18: Komputasi PC menggunakan kelas PCA

```
val projected2 = data2.map(p => p.copy(features = pca.transform(p.features)))
```

Gambar 3.19: Proyeksi data dengan nilai PC yang dihasilkan kelas PCA

- 9 Agar hasil dapat ditampilkan, lakukan pemanggilan method `collect` pada RDD (Lihat gambar  
 10 3.20).

11

```
val collect = projected.rows.collect()
println("Projected Row Matrix principal component:")
collect.foreach { vector => println(vector) }
```

Gambar 3.20: Menampilkan hasil RowMatrix ke console

```
val collect2 = projected2.collect()
println("Projected vector of principal component:")
collect2.foreach { vector => println(vector) }
```

Gambar 3.21: Menampilkan hasil kelas PCA ke console

- 12 Hasil output dapat dilihat pada gambar 3.22 berikut:

```

Projected Row Matrix principal component:
[0.42728978006230384,-4.575530681088792,-3.924563878285694,-2.384917365628506]
[-5.046205956059748,-0.32969122246627414,-3.9656759827826313,-2.3299572180627894]
[-7.688325015615871,-3.9910480619510174,-4.221579511060134,-1.7210800077552924]
[-9.095609731714921,-3.8799684588507466,-2.301808744220712,-2.8069318719973104]
[-9.845757439333296,-3.9310658971344616,-5.339275866697909,-2.49087649077203]
[-7.232540018920385,-3.3755441938894597,-2.9164315186553162,-1.5003879173294512]
Projected vector of principal component:
(0.0,[0.5206573684395938,-0.4271322870657463,-0.7392387395392245,-9.272192293572736E-17,1.0])
(1.0,[-1.1529018904707842,-0.7155024798795668,-0.39858930271029713,1.9520404828574198E-17,1.0])
(1.0,[-0.7557893406837773,0.1721478589408803,-0.6317812811178027,-7.93016446160826E-17,1.0])
(0.0,[0.0,0.0,0.0,0.0,1.0])
(1.0,[-0.7557893406837773,0.1721478589408803,-0.6317812811178027,-7.93016446160826E-17,1.0])

```

Gambar 3.22: Hasil keluaran eksperimen PCA pada console

### 3.3.4 Term Frequency - Inverse Document Frequency (TF-IDF)

Salah satu teknik feature extraction yang disediakan MLlib adalah TF-IDF. TF-IDF merupakan kepanjangan dari Term Frequency dan Inverse Document Frequency. Teknik ini digunakan untuk mendapatkan informasi dari suatu dokumen. MLlib menyediakan perhitungan TF dalam kelas HashingTF. Sedangkan untuk perhitungan IDF dapat menggunakan kelas IDF. Method transform disediakan pada kedua kelas ini untuk mengubah fitur data masukkan menjadi sebuah vector TF dan vector DF.

8

```

val hashingTF = new HashingTF()
val tf: RDD[Vector] = hashingTF.transform(documents)

```

Gambar 3.23: Memanggil method perhitungan TF

Implementasi perhitungan TF pada MLlib dapat dilakukan dengan menginisialisasi kelas hashingTF dan kemudian memanggil method transform.

11

```

tf.cache()
val idf = new IDF().fit(tf)
val tfidf: RDD[Vector] = idf.transform(tf)

```

Gambar 3.24: Memanggil method perhitungan IDF

Sedangkan untuk implementasi TF-IDF membutuhkan dua tahap yaitu melakukan operasi IDF dan dilanjutkan dengan mengalikan IDF dengan TF.

14

```

val idfIgnore = new IDF(minDocFreq = 2).fit(tf)
val tfidfIgnore: RDD[Vector] = idfIgnore.transform(tf)

```

Gambar 3.25: Pemanggilan IDF dengan batas minimum document

Operasi IDF pada MLlib menyediakan sebuah pilihan untuk dapat mengabaikan sebuah term yang muncul kurang dari sejumlah  $n$  document. Untuk melakukan itu pada MLlib, pengguna dapat mengisi parameter kelas IDF dengan jumlah minimum document.

18

```

    println("tfidf: ")
    tfidf.foreach(x => println(x))

    println("tfidfIgnore: ")
    tfidfIgnore.foreach(x => println(x))

```

Gambar 3.26: Menampilkan hasil ke console

- Untuk melihat hasilnya, dilakukan foreach pada hasil TFIDF. Hasil keluaran eksperimen TF-IDF dapat dilihat pada gambar 3.27 dan 3.28.

```

18/04/20 08:59:31 INFO BlockManagerMasterEndpoint: Registering block manager 192.168.100.10:61050 with 1445.7 MB RAM, I
18/04/20 08:59:31 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver, 192.168.100.10, 61050, None)
18/04/20 08:59:31 INFO BlockManager: Initialized BlockManager: BlockManagerId(driver, 192.168.100.10, 61050, None)

tfidf:
(1048576, [30465, 617253, 756426, 914452], [1.7047480922384253, 0.4519851237430572, 1.2992829841302609, 0.7884573603642703])
(1048576, [351218, 617253, 809213, 914452], [1.2992829841302609, 0.4519851237430572, 1.7047480922384253, 0.7884573603642703])
(1048576, [206183, 329900, 552513, 617253], [1.7047480922384253, 1.7047480922384253, 0.4519851237430572])
(1048576, [167741, 300721, 617253, 1004364], [1.2992829841302609, 1.2992829841302609, 0.4519851237430572, 1.7047480922384253])
(1048576, [479587, 617253, 876121, 914452], [1.7047480922384253, 0.4519851237430572, 1.2992829841302609, 0.7884573603642703])
(1048576, [223635, 326189, 548860, 1038045], [1.7047480922384253, 1.7047480922384253, 1.2992829841302609, 1.7047480922384253])
(1048576, [300721, 639608, 909704, 914452], [1.2992829841302609, 1.0116009116784799, 1.7047480922384253, 0.7884573603642703])
(1048576, [167741, 617253, 639608, 876121], [1.2992829841302609, 0.4519851237430572, 1.0116009116784799, 1.2992829841302609])
(1048576, [17372, 548860, 639608, 674826], [1.7047480922384253, 1.2992829841302609, 1.0116009116784799, 1.7047480922384253])
(1048576, [351218, 517023, 595906, 756426], [1.2992829841302609, 1.7047480922384253, 1.2992829841302609])

tfidfIgnore:
(1048576, [30465, 617253, 756426, 914452], [0.0, 0.4519851237430572, 1.2992829841302609, 0.7884573603642703])
(1048576, [351218, 617253, 809213, 914452], [0.0, 0.4519851237430572, 0.0, 0.7884573603642703])
(1048576, [206183, 329900, 552513, 617253], [0.0, 0.0, 0.0, 0.4519851237430572])
(1048576, [167741, 300721, 617253, 1004364], [1.2992829841302609, 1.2992829841302609, 0.4519851237430572, 0.0])
(1048576, [479587, 617253, 876121, 914452], [0.0, 0.4519851237430572, 1.2992829841302609, 0.7884573603642703])
(1048576, [223635, 326189, 548860, 1038045], [0.0, 0.0, 1.2992829841302609, 0.0])
(1048576, [300721, 639608, 909704, 914452], [1.2992829841302609, 1.0116009116784799, 0.0, 0.7884573603642703])
(1048576, [167741, 617253, 639608, 876121], [1.2992829841302609, 0.4519851237430572, 1.0116009116784799, 1.2992829841302609])
(1048576, [17372, 548860, 639608, 674826], [0.0, 1.2992829841302609, 1.0116009116784799, 0.0])
(1048576, [351218, 517023, 595906, 756426], [1.2992829841302609, 0.0, 0.0, 1.2992829841302609])

```

Gambar 3.27: Hasil output pada console

```

tfidfIgnore:
(1048576, [30465, 617253, 756426, 914452], [0.0, 0.4519851237430572, 1.2992829841302609, 0.7884573603642703])
(1048576, [351218, 617253, 809213, 914452], [0.0, 0.4519851237430572, 0.0, 0.7884573603642703])
(1048576, [206183, 329900, 552513, 617253], [0.0, 0.0, 0.0, 0.4519851237430572])
(1048576, [167741, 300721, 617253, 1004364], [1.2992829841302609, 1.2992829841302609, 0.4519851237430572, 0.0])
(1048576, [479587, 617253, 876121, 914452], [0.0, 0.4519851237430572, 1.2992829841302609, 0.7884573603642703])
(1048576, [223635, 326189, 548860, 1038045], [0.0, 0.0, 1.2992829841302609, 0.0])
(1048576, [300721, 639608, 909704, 914452], [1.2992829841302609, 1.0116009116784799, 0.0, 0.7884573603642703])
(1048576, [167741, 617253, 639608, 876121], [1.2992829841302609, 0.4519851237430572, 1.0116009116784799, 1.2992829841302609])
(1048576, [17372, 548860, 639608, 674826], [0.0, 1.2992829841302609, 1.0116009116784799, 0.0])
(1048576, [351218, 517023, 595906, 756426], [1.2992829841302609, 0.0, 0.0, 1.2992829841302609])

```

Gambar 3.28: Hasil output dengan minimum document pada console

### 3.3.5 Alternating Least Squares (ALS)

- Collaborative filtering adalah suatu teknik agar sistem dapat merekomendasikan sesuatu dimana penilaian(*rating*) dan reaksi pengguna terhadap suatu produk digunakan untuk merekomendasikan produk yang baru. Fokus utama teknik ini adalah memprediksi suatu nilai entry pada hubungan pengguna dan produk pada matrix asosiasi yang belum diketahui nilainya. MLLib menggunakan algoritma Alternating Least Squares(ALS) untuk mengobservasi sebuah set data. Set data tersebut mendeskripsikan hubungan pengguna dan produk. Beberapa parameter yang perlu diketahui yaitu :

- *numBlocks*, adalah jumlah block yang digunakan untuk melakukan komputasi pararel (nilai -1 agar dikonfigurasi otomatis).
- *rank*, sebuah angka yang menunjukan jumlah fitur yang digunakan (Jumlah latent factor).
- *Iterations*, adalah jumlah iterasi pada ALS. Umumnya hasil yang optimal dapat dicapai ALS dalam iterasi kurang atau sama dengan dari 20.

- *lambda*, adalah nilai regularisasi pada ALS. Nilai lambda ini digunakan dalam rangka penggunaan algoritma Alternating Least Squares with Weighted  $\lambda$  Regularization (ALS-WR).
- *implicitPrefs*, diberikan dua pilihan pada ALS yaitu implicit feedback atau explicit feedback data.

Pelatihan data pada algoritma ALS di MLlib menggunakan kelas ALS pada package spark.mllib.recommendation.ALS. Sedangkan data masukkan algoritma ALS menggunakan kelas Rating pada package spark.mllib.recommendation.Rating. Rating ini mendefinisikan hubungan pengguna dengan produk. Penulis mengubah setiap baris data input menjadi Rating.

```
// Load and parse the data
val data = sc.textFile("E:/InputTest/ALS_data.txt")
val ratings = data.map(_.split(',') match { case Array(user, item, rate) =>
  | Rating(user.toInt, item.toInt, rate.toDouble)
})
```

Gambar 3.29: Mengkonversi data menjadi Rating

Setiap baris rating ini yang akan dilatih pada ALS. Pelatihan memanggil method **ALS.train()** (Lihat gambar 3.30). Method ini dapat berisikan beberapa variasi parameter (*numBlocks*, *Rank*, *Iteration*, dan *Lambda*). *implicit feedback* dilakukan dengan memanggil method **trainImplicit()**.

```
// Build model ALS
val rank = 10
val numIterations = 10
val model = ALS.train(ratings, rank, numIterations, 0.01)
```

Gambar 3.30: Pelatihan data menggunakan ALS

Pelatihan ini menghasilkan sebuah model MatrixFactorizationModel. Kelas ini masih berada di package spark.mllib.recommendation. Model ini digunakan untuk melakukan prediksi *rating* pada suatu data yang tidak memiliki *rating* produk. Pada eksperimen *rating* produk dari data masukkan dihilangkan lalu dijadikan data input untuk prediksi (Lihat gambar 3.31).

```
// Evaluate the model on rating data
val usersProducts = ratings.map { case Rating(user, product, rate) =>
  (user, product)
}
println("predictions : ")
val predictions =
  model.predict(usersProducts).map { case Rating(user, product, rate) =>
    println(user + " " + product + " " + rate)
    ((user, product), rate)
}
```

Gambar 3.31: Prediksi data menggunakan model

Untuk menghitung rata-rata kesalahan prediksi, digunakan perbandingan dengan rating pada data awal. Kemudian dapat dihitung nilai rata-rata kesalahan prediksi (Lihat gambar 3.32).

```

val ratesAndPreds = ratings.map { case Rating(user, product, rate) =>
  ((user, product), rate)
}.join(predictions)

val MSE = ratesAndPreds.map { case ((user, product), (r1, r2)) =>
  val err = (r1 - r2)
  err * err
}.mean()
println(s"Mean Squared Error = $MSE")

```

Gambar 3.32: Menghitung rata-rata kesalahan prediksi

- 1 Hasil yang dapat ditampilkan adalah hasil prediksi data. Sedangkan model dapat disimpan
- 2 pada *filesystem* (Lihat gambar 3.33). Hasil keluaran dari eksperimen ini dapat dilihat pada gambar
- 3 3.34

```

// Save and load model
model.save(sc, "E:/Output/ALS")
val sameModel = MatrixFactorizationModel.load(sc, "E:/Output/ALS")

```

Gambar 3.33: Menyimpan model ALS

```

Run CollaborativeFilteringALS
18/04/20 09:06:20 INFO BlockManagerMasterEndpoint: Registering BlockManager...
18/04/20 09:06:20 INFO BlockManagerMaster: Registered BlockManager: Initialize
predictions :
4 4 4.995256587097616
4 1 1.000245201054117
4 3 1.000245201054117
4 2 4.995256587097616
1 4 0.9999533764540887
1 1 4.996715891880623
1 3 4.996715891880623
1 2 0.9999533764540887
3 4 4.995256587097616
3 1 1.000245201054117
3 3 1.000245201054117
3 2 4.995256587097616
2 4 0.9999533764540887
2 1 4.996715891880623
2 3 4.996715891880623
2 2 0.9999533764540887
Mean Squared Error = 8.336907353558553E-6

```

Gambar 3.34: Hasil output prediksi

#### 3.3.6 K-Means

- 5 Algoritma K-Means merupakan algoritma clustering. Artinya algoritma termasuk kategori *Unsupervised learning* dan masukkan data yang digunakan adalah data tidak berlabel. Penulis menggunakan dataset bunga iris tanpa memiliki label. Dataset tersebut diambil dari sebuah halaman website [www.archive.ics.uci.edu/ml/datasets/iris](http://www.archive.ics.uci.edu/ml/datasets/iris). Pada eksperimen ini dilakukan percobaan memasukkan file pada hadoop filesystem yaitu HDFS. Menyimpan file kedalam HDFS dilakukan dengan
- 10 memanggil perintah berikut:

```
11 hadoop dfs -put /kresna/dataset/Iris.csv /user/hadoop/iris-dataset
```

```

1 import org.apache.spark.{SparkConf, SparkContext}
2 import org.apache.spark.mllib.clustering.KMeans
3 import org.apache.spark.mllib.clustering.KMeansModel
4 import org.apache.spark.mllib.linalg.Vectors
5
6 object kmeans {
7
8   def main(args: Array[String]) {
9
10     //Membuat SparkContext
11     val conf = new SparkConf().setAppName("KMeansExample")
12       .setMaster("local")
13     val sc = new SparkContext(conf)
14
15     // Me-load data dengan membuat RDD
16     val data = sc.textFile("hdfs://localhost:9001/user/hadoop/iris-dataset");
17     val parsedData = data.map(s => Vectors.dense(s.split(' ')
18       .map(_.toDouble))).cache()
19
20     // Melakukan pelatihan data dengan algoritma K-Means
21     val numClusters = 3
22     val numIterations = 20
23     val clusters = KMeans.train(parsedData, numClusters
24       , numIterations)
25

```

Gambar 3.35: Membuat RDD dengan menghubungkan HDFS

Setelah dataset tersebut telah tersimpan dalam HDFS, pada kode program hanya perlu dituliskan lokasi direktori HDFS dataset tersebut(Gambar 3.35). Kemudian pelatihan data dengan menggunakan algoritma K-Means gunakan objek KMeans yang disediakan API dan panggil method `train()`.

```

20 // Melakukan pelatihan data dengan algoritma K-Means
21 val numClusters = 3
22 val numIterations = 20
23 val clusters = KMeans.train(parsedData, numClusters
24   , numIterations)
25
26 // Hasil evaluasi komputasi clustering
27 val WSSSE = clusters.computeCost(parsedData)
28 println("Within Set Sum of Squared Errors = " + WSSSE)
29
30 // Menyimpan model yang dihasilkan
31 clusters.save(sc, "E:/TestKmeans")
32 val sameModel = KMeansModel.load(sc, "E:/TestKmeans")
33
34 // Menampilkan Clusters Centers
35 println("Cluster Centers: ")
36 sameModel.clusterCenters.foreach(println)
37
38 //Stop Spark Context
39 sc.stop()
40 }
41

```

Gambar 3.36: Menyimpan model hasil pelatihan dan memuat model

Hasil pelatihan data pada algoritma K-Means berupa sebuah `KMeansModel`. Model ini dapat disimpan ke penyimpanan eksternal. Jika hasil dari model dibutuhkan, maka kita tinggal memanggil method `load()` (Gambar 3.36)untuk mengambil kembali model dan data yang diperlukan seperti centroid(clusterCenters). Karena hanya satu `SparkContext` yang dapat aktif per JVM, perlu menghentikan `SparkContext` untuk membuat `SparkContext` yang baru. Cara menghentikan

- 1 SparkContext adalah dengan memanggil method `stop()` pada `SparkContext` yang sedang aktif.

2

```
<terminated> Testkmeans$ [Scala Application] C:\Java\jre1.8.0_141\bin\javaw.exe (Nov 24, 2017, 2:56:33 PM)
17/11/24 14:56:46 INFO CodeGenerator: Code generated in 66.826158 ms
17/11/24 14:56:46 INFO CodeGenerator: Code generated in 39.476139 ms
17/11/24 14:56:46 INFO InternalParquetRecordReader: RecordReader initialized will read a total of 3 records.
17/11/24 14:56:46 INFO InternalParquetRecordReader: at row 0. reading next block
17/11/24 14:56:46 INFO CodecPool: Got brand-new decompressor [.snappy]
17/11/24 14:56:46 INFO InternalParquetRecordReader: block read in memory in 17 ms. row count = 3
17/11/24 14:56:46 INFO Executor: Finished task 0.0 in stage 21.0 (TID 21). 1805 bytes result sent to driver
17/11/24 14:56:46 INFO TaskSetManager: Finished task 0.0 in stage 21.0 (TID 21) in 529 ms on localhost (executor driver)
17/11/24 14:56:46 INFO TaskSchedulerImpl: Removed TaskSet 21.0, whose tasks have all completed, from pool
17/11/24 14:56:46 INFO DAGScheduler: ResultStage 21 (collect at KMeansModel.scala:144) finished in 0.530 s
17/11/24 14:56:46 INFO DAGScheduler: Job 17 finished: collect at KMeansModel.scala:144, took 0.555433 s
Cluster Centers:
[5.901612983225807, 2.748387096774194, 4.393548387096774, 1.4338709677419355]
[5.005999999999999, 3.4180000000000006, 1.4640000000000002, 0.2439999999999999]
[6.85, 3.0736842105263147, 5.742105263157893, 2.071052631578947]
17/11/24 14:56:46 INFO SparkUI: Stopped Spark web UI at http://192.168.0.107:4040
17/11/24 14:56:46 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
17/11/24 14:56:46 INFO MemoryStore: MemoryStore cleared
17/11/24 14:56:46 INFO BlockManager: BlockManager stopped
17/11/24 14:56:46 INFO BlockManagerMaster: BlockManagerMaster stopped
17/11/24 14:56:46 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
17/11/24 14:56:46 INFO SparkContext: Successfully stopped SparkContext
17/11/24 14:56:46 INFO ShutdownHookManager: Shutdown hook called
17/11/24 14:56:46 INFO ShutdownHookManager: Deleting directory C:\Users\Kresna\AppData\Local\Temp\spark-b58cda41-6b92-4
```

Gambar 3.37: Hasil output program eksperimen KMeans

- 3 Hasil output library K-Means MLlib yaitu centroid setiap Cluster pada iterasi terakhir(Gambar  
4 3.37). Oleh karena itu, K-Means MLlib tidak bisa mendapatkan pola yang ada. Penulis akan  
5 melakukan pengembangan pada K-Means MLlib dalam rangka mendapatkan pola dari set data.



1

## BAB 4

### 2 PENGEMBANGAN K-MEANS PADA LINGKUNGAN SPARK

3 Pada bab ini, penulis akan menjelaskan apa saja yang dilakukan dalam pengembangan K-Means  
4 MLlib. Pengembangan dilakukan untuk mencapai tujuan yaitu mendapatkan pola dari dataset yang  
5 diolah. Pola yang ingin didapatkan meliputi perhitungan rata-rata, nilai maksimum, nilai minimum  
6 dan nilai standar deviasi dari setiap fitur yang ada pada data. Selain itu, perlu didapatkan juga  
7 jumlah anggota pada setiap cluster yang dihasilkan dari algoritma K-Means pada fungsi clustering.

#### 8 4.1 Kebutuhan Pengembangan

9 Dalam melakukan pengembangan perlu diketahui terlebih dulu kebutuhan pengembangan. Pe-  
10 ngembangan dilakukan untuk mendapatkan pola dari dataset yang diolah. Pola yang dibutuhkan  
11 diantaranya :

- 12 1. Centroid setiap cluster
- 13 2. Jumlah anggota setiap cluster
- 14 3. Nilai rata-rata setiap atribut/fitur objek
- 15 4. Nilai minimum setiap atribut/fitur objek
- 16 5. Nilai maximum setiap atribut/fitur objek
- 17 6. Nilai standar deviasi atribut/fitur atribut objek

18 Dapat dibandingkan dengan output yang dihasilkan kelas KMeans pada MLlib yang dilakukan  
19 sebelumnya. Output yang dihasilkan KMeans hanya centroid dari setiap cluster yang dihasilkan  
20 dari pelatihan. Sehingga dibutuhkan pengembangan yang meliputi:

- 21 1. Menghitung Jumlah anggota pada setiap cluster
- 22 2. Menghitung nilai rata-rata setiap atribut/fitur objek
- 23 3. Menghitung nilai minimum setiap atribut/fitur objek
- 24 4. Menghitung nilai maximum setiap atribut/fitur objek
- 25 5. Menghitung nilai standar deviasi setiap atribut/fitur objek

26 Perlu diketahui bahwa pola-pola tersebut harus dapat diatur sesuai keinginan pengguna. Contoh,  
27 jika pengguna tidak menginginkan untuk perhitungan minimum fitur maka pengguna dapat mengatur  
28 konfigurasinya.

## 1 4.2 Analisis K-Means MLlib

2 Setelah mengetahui kebutuhan untuk pengembangan K-Means, penulis perlu melakukan analisis  
3 pada K-Means MLlib terlebih dulu sebelum melakukan perubahan/modifikasi pada source co-  
4 de. Analisis dilakukan dengan mempelajari *source code* K-Means yang ada pada library MLlib.  
5 Apache Spark memiliki repository berupa mirror dari source code Apache Spark. Repository ini  
6 merupakan repositori yang digunakan untuk pengembangan Apache Spark. Repositori berlokasi  
7 di <https://github.com/apache/spark>. Sedangkan untuk source code dari library MLlib berada di  
8 <https://github.com/apache/spark/tree/master/mllib/src/main/scala/org/apache/spark/mllib>.

9  
10 Sebelum masuk pada bagian source code, akan dijelaskan struktur dari *library* k-means di  
11 MLlib berdasarkan dokumentasi Spark versi 2.2.0. Algoritma inisialisasi centeroid K-Means pada  
12 kelas KMeans menggunakan algoritma KMeans|| yang dikenalkan oleh Bahman Bahmani et al.  
13 Sedangkan untuk algoritma iterasi mengimplementasikan algoritma K-Means yang standar.

14  
15 Dalam melakukan komputasi Algoritma K-Means, MLlib menggunakan dua buah file yaitu  
16 KMeans.scala dan KMeansModel.scala. File KMeans.scala berisi implementasi algoritma  
17 K-Means. Sedangkan KMeansModel.scala berisi deklarasi untuk Model keluaran KMeans. Masing-  
18 masing file ini mendeklarasikan *class* dan *object*. Kelas dari KMeans adalah kelas yang meng-  
19 implementasi algoritma K-Means. Sedangkan objek K-Means merupakan objek *singleton* yang  
20 disediakan agar dapat melakukan pemanggilan fungsi dengan cepat. Kelas KMeans memiliki  
21 parameter diantaranya :

- 22 1. *k*, merupakan jumlah kluster yang akan dihasilkan.
- 23 2. *maxIterations*, merupakan jumlah maksimum iterasi yang dapat dilakukan.
- 24 3. *initializationMode*, merupakan mode inisialisasi nilai default-nya adalah "k-means||"
- 25 4. *initializationSteps*, merupakan jumlah steps untuk mode inisialisasi KMeans||
- 26 5. *epsilon*, merupakan nilai ambang batas untuk menentukan centroid telah converge. nilai  
27 defaultnya adalah *1e-4*.
- 28 6. *seed*, merupakan nilai acak untuk inisialisasi cluster. Nilai default-nya adalah 'random'.

29 Sedangkan kode program implementasi algoritma K-Means terdapat pada method **run()**. Para-  
30 meter method **run()** bernama data yaitu bertipe **RDD[vector]**. Berdasarkan dokumentasi Spark,  
31 data yang akan diteruskan ke parameter disarankan untuk di *cache* terlebih dulu untuk peningkatan  
32 performa komputasi. Output dari method ini berupa **KMeansModel**.

33  
34 KMeansModel sesuai yang sudah dijelaskan pada bagian [Pemanggilan fungsi K-Means](#) memiliki  
35 deklarasi kelas dan object. Object KMeansModel adalah sebagai penyimpan model yang dihasilkan  
36 dari pelatihan data pada KMeans. KMeansModel beberapa *method* yang dapat digunakan pengguna  
37 diantaranya :

1. Menampilkan cluster center atau centroid dari setiap cluster.
2. computeCost(data), yaitu untuk menghitung nilai cost K-Means dari model ini untuk data yang diberikan melalui parameter data.
3. predict(), yaitu untuk melakukan prediksi index cluster pada setiap object yang diberikan

5        Setelah mempelajari struktur KMeans MLlib, penulis kemudian menelusuri source code dari  
 6 K-Means. Ini dilakukan untuk menganalisis implementasi dari algoritma KMeans. Kelas yang akan  
 7 menjadi fokus utama analisis adalah kelas KMeans karena kelas ini method run() dari KMeans  
 8 didefinisikan.

```

203  /**
204   * Train a K-means model on the given set of points; `data` should be cached for high
205   * performance, because this is an iterative algorithm.
206   */
207  @Since("0.8.0")
208  def run(data: RDD[Vector]): KMeansModel = {
209    run(data, None)
210  }
211
212  private[spark] def run(
213    data: RDD[Vector],
214    instr: Option[Instrumentation[NewKMeans]]): KMeansModel = {
215
216    if (data.getStorageLevel == StorageLevel.NONE) {
217      logWarning("The input data is not directly cached, which may hurt performance if its"
218        + " parent RDDs are also uncached.")
219    }
220
221    // Compute squared norms and cache them.
222    val norms = data.map(Vectors.norm(_, 2.0))
223    norms.persist()
224    val zippedData = data.zip(norms).map { case (v, norm) =>
225      new VectorWithNorm(v, norm)
226    }
227    val model = runAlgorithm(zippedData, instr)
228    norms.unpersist()
229
230    // Warn at the end of the run as well, for increased visibility.
231    if (data.getStorageLevel == StorageLevel.NONE) {
232      logWarning("The input data was not directly cached, which may hurt performance if its"
233        + " parent RDDs are also uncached.")
234    }
235    model
236  }

```

Gambar 4.1: Method Run() KMeans.scala

9        Jika mengacu pada Docs yang disediakan oleh Apache Spark, algoritma Kmeans berada dibawah  
 10 method `run()` (Gambar 4.2). Method ini memanggil method `run()` yang memiliki modifier private  
 11 dengan memberikan nilai parameter sebuah data bertipe RDD. Perlu diketahui bahwa seluruh data  
 12 yang akan diolah sudah dalam bentuk RDD pada tahap ini. Setiap baris data disebut dengan objek.  
 13

```
221      // Compute squared norms and cache them.  
222      val norms = data.map(Vectors.norm(_, 2.0))  
223      norms.persist()  
224      val zippedData = data.zip(norms).map { case (v, norm) =>  
225          new VectorWithNorm(v, norm)  
226      }  
227      val model = runAlgorithm(zippedData, instr)  
228      norms.unpersist()
```

Gambar 4.2: Konversi data menjadi Norm Vektor pada method `run()`

1 Di dalam method `private run()` seluruh data akan diubah menjadi bentuk norm vektor. Hal  
2 ini dilakukan dengan cara memanggil `map()`. Seperti yang sudah diketahui, RDD memiliki operasi  
3 *transformation* dan *action*. Method `map()` merupakan salah satu operasi *transformation*. Method  
4 `map()` adalah operasi yang mengembalikan sebuah RDD baru yang dibentuk dengan melakukan  
5 iterasi setiap element(object) pada RDD dengan menerapkan sebuah argumen melalui sebuah fungsi  
6 di parameternya. Pada gambar 4.3 fungsi didalam parameter `map()` mengubah vector menjadi norm  
7 Vektor(`VectorWithNorm`). Perlu diingat juga, method ini tidak akan dikerjakan sampai operasi  
8 *action* dipanggil. Sebelumnya data di-zip dengan memanggil `zip()`.

9

```

238     /**
239      * Implementation of K-Means algorithm.
240      */
241     private def runAlgorithm(
242       data: RDD[VectorWithNorm],
243       instr: Option[Instrumentation[NewKMeans]]): KMeansModel = {
244
245     val sc = data.sparkContext
246
247     val initStartTime = System.nanoTime()
248
249     val centers = initialModel match {
250       case Some(kMeansCenters) =>
251         kMeansCenters.clusterCenters.map(new VectorWithNorm(_))
252       case None =>
253         if (initializationMode == KMeans.RANDOM) {
254           initRandom(data)
255         } else {
256           initKMeansParallel(data)
257         }
258     }
259     val initTimeInSeconds = (System.nanoTime() - initStartTime) / 1e9
260     logInfo(f"Initialization with $initializationMode took ${initTimeInSeconds%.3f} seconds.")
261
262     var converged = false
263     var cost = 0.0
264     var iteration = 0
265
266     val iterationStartTime = System.nanoTime()
267
268     instr.foreach(_.logNumFeatures(centers.head.vector.size))
269
270     // Execute iterations of Lloyd's algorithm until converged
271     while (iteration < maxIterations && !converged) {

```

Gambar 4.3: Inisialisasi centroid awal pada kelas KMeans

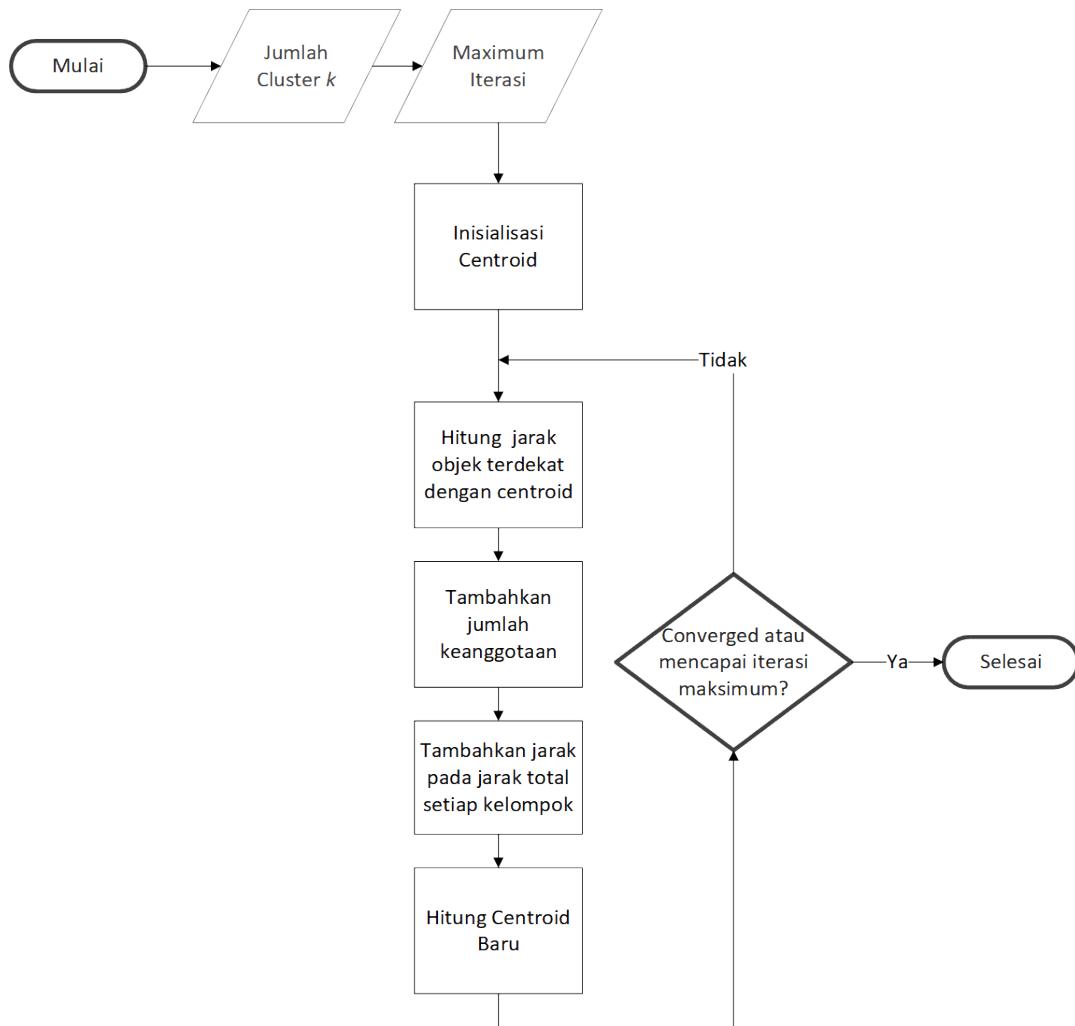
- 1 Setelah data telah diubah, method `runAlgorithm()` dipanggil dengan parameter RDD. `runAlgorithm()`  
 2 merupakan method yang implementasi algoritma K-Means. Pada method inilah iterasi algoritma  
 3 K-Means akan dilakukan. Pertama-tama, method ini menyimpan SparkContext kedalam variabel  
 4 lokal dan menyimpan waktu saat itu sebagai waktu mulai. Centroid awal akan dibuat berdasarkan  
 5 `initialModel` yang digunakan dan disimpan dalam variable `centers`. Terdapat dua variabel yang  
 6 perlu diperhatikan setelahnya adalah `converged` dan `iteration`. `Converged` adalah nilai Boolean  
 7 yang menandakan status centroid telah *converged* atau tidak. Sedangkan `iteration` adalah sebuah  
 8 nilai *integer* untuk menyimpan jumlah iterasi yang sudah berjalan.
- 9

```

270     // Execute iterations of Lloyd's algorithm until converged
271     while (iteration < maxIterations && !converged) {
272         val costAccum = sc.doubleAccumulator
273         val bcCenters = sc.broadcast(centers)
274
275         // Find the sum and count of points mapping to each center
276         val totalContribs = data.mapPartitions { points =>
277             val thisCenters = bcCenters.value
278             val dims = thisCenters.head.vector.size
279
280             val sums = Array.fill(thisCenters.length)(Vectors.zeros(dims))
281             val counts = Array.fill(thisCenters.length)(0L)
282
283             points.foreach { point =>
284                 val (bestCenter, cost) = KMeans.findClosest(thisCenters, point)
285                 costAccum.add(cost)
286                 val sum = sums(bestCenter)
287                 axpy(1.0, point.vector, sum)
288                 counts(bestCenter) += 1
289             }
290
291             counts.indices.filter(counts(_) > 0).map(j => (j, (sums(j), counts(j)))).iterator
292         }.reduceByKey { case ((sum1, count1), (sum2, count2)) =>
293             axpy(1.0, sum2, sum1)
294             (sum1, count1 + count2)
295         }.collectAsMap()
296
297         bcCenters.destroy(blocking = false)
298
299         // Update the cluster centers and costs
300         converged = true
301         totalContribs.foreach { case (j, (sum, count)) =>
302             scal(1.0 / count, sum)
303             val newCenter = new VectorWithNorm(sum)
304             if (converged && KMeans.fastSquaredDistance(newCenter, centers(j)) > epsilon * epsilon) {
305                 converged = false
306             }
307             centers(j) = newCenter
308         }
309
310         cost = costAccum.value
311         iteration += 1
312     }

```

Gambar 4.4: Kode program implementasi algoritma pada method `runAlgorithm()`



Gambar 4.5: Diagram flow iterasi K-Means pada MLlib

1      Kode program diilustrasikan dengan *flow chart*(Gambar 4.5). Input yang dibutuhkan adalah  
 2      jumlah cluster  $k$  yang diinginkan dan jumlah iterasi maksimum. Centroid awal diinisialisasi terlebih  
 3      dulu sebelum menjalankan iterasi. Iterasi akan berhenti jika status *converged* bernilai `true` dan  
 4      jumlah iterasi sudah mencapai nilai iterasi maksimum. Iterasi ini dilakukan untuk mencari nilai  
 5      centroid baru sampai nilai centroid stabil. Nilai centroid terakhir akan dikembalikan ketika iterasi  
 6      berhenti.

7      Langkah pertama yang dilakukan pada iterasi ini adalah dengan melakukan broadcast nilai  
 8      `centers`. Broadcast adalah fungsi untuk menyimpan suatu variabel yang akan bersifat *read-only* dan  
 9      *di-cached* di setiap *worker*. Hal ini dilakukan agar sebuah nilai variabel tidak dikirimkan ke setiap  
 10     mesin berulang kali setiap suatu tugas dikerjakan. Seluruh operasi perhitungan untuk mendapatkan  
 1      nilai centroid baru disimpan dalam `newCenters`. Jumlah centroid sama dengan jumlah cluster  $k$   
 2      yang ditentukan. Operasi *map* dan *reduce* akan digunakan untuk melakukan operasi iterasi RDD.  
 3

```

276     val totalContribs = data.mapPartitions { points =>
277         val thisCenters = bcCenters.value
278         val dims = thisCenters.head.vector.size
279
280         val sums = Array.fill(thisCenters.length)(Vectors.zeros(dims))
281         val counts = Array.fill(thisCenters.length)(0L)
282
283         points.foreach { point =>
284             val (bestCenter, cost) = KMeans.findClosest(thisCenters, point)
285             costAccum.add(cost)
286             val sum = sums(bestCenter)
287             axpy(1.0, point.vector, sum)
288             counts(bestCenter) += 1
289         }
290
291         counts.indices.filter(counts(_) > 0).map(j => (j, (sums(j), counts(j)))).iterator
292     }.reduceByKey { case ((sum1, count1), (sum2, count2)) =>

```

Gambar 4.6: Fungsi di Method `mapPartitions()` pada KMeans

4     Method `map()` yang digunakan adalah `mapPartitions(func)`. Method ini sama seperti `map()`  
5     namun dengan peningkatan kecepatan komputasi. Parameter `mapPartition` pada hal ini adalah  
6     menjalankan operasi yang menghitung jarak dari setiap objek ke center dan memilih cluster dengan  
7     jarak terdekat. Untuk menghitung jarak terdekat digunakan method `findClosest()`.

8  
9     Ada dua variabel bertipe `Array` yang dipakai pada bagian ini yaitu `sums` dan `counts`. `Sums`  
10   menyimpan jumlah nilai-nilai atribut object. Sedangkan `counts` menyimpan jumlah object yang  
11   terdekat dengan suatu *cluster*. `Sums` berukuran sama dengan jumlah *cluster* dan seluruhnya diinis-  
12   alisasi dengan nilai 0. Nilai `sums` diakumulasi pada baris kode ke-287 (Gambar 4.6) menggunakan  
13   method `axpy`. Nilai yang diakumulasi adalah nilai setiap atribut object. Nilai tersebut kemudian  
14   disimpan kedalam array `sums` dengan index sesuai dengan nomor urut *cluster* yang paling dekat  
15   dengan objek tersebut.

16  
17   Nilai return `mapPartition` ini berada pada baris ke-291 (Gambar 4.6). `Counts.indices` adalah  
18   untuk mendapatkan nilai *Range*. *Range* digunakan untuk jumlah iterasi. Filter adalah fungsi  
19   menyaring objek dengan *predicate* tertentu. *Predicate* merupakan sebuah argumen yang mengha-  
20   silkan nilai boolean. Argumen pada *source code* memiliki arti jika nilai count pada suatu index  
21   bernilai lebih besar dari 0 maka akan bernilai true dan sebaliknya. Kemudian dipanggil `map()`  
22   yang isi fungsinya adalah mengubah setiap nilai J menjadi sebuah pasangan *key* dan *value*. *Key*  
23   yang digunakan adalah nilai index nomor cluster. Sedangkan *value*-nya berupa sebuah *tupple* nilai  
24   variabel `sums` dan `counts`. Nilai yang diambil adalah nilai pada index J, dimana J adalah nomor  
1   *cluster*. Dan yang terakhir adalah memanggil `iterator` untuk membuat semua object menjadi  
2   *iterable* untuk dapat dilakukan suatu operasi selanjutnya.

3

```

292     ).reduceByKey { case ((sum1, count1), (sum2, count2)) =>
293       axpy(1.0, sum2, sum1)
294       (sum1, count1 + count2)
295     }.collectAsMap()

```

Gambar 4.7: Fungsi di Method `reduceByKey()` pada KMeans.scala

Method reduce yang digunakan adalah `reduceByKey(func)`. Method ini digunakan dalam rangka menggabungkan dua buah pasangan *key* dan *value* yang nilai *key*-nya sama. Dalam proses menggabungkannya, *value* akan menjadi 1 nilai saja. *Value* yang ada pada kasus ini adalah variabel `count` dan `sum`. Kedua variabel ini masing-masing akan dijumlahkan nilainya dengan variabel pasangan lainnya yang nilai *key*-nya sama. Nilai `sum` dijumlahkan menggunakan method `Blas.axpy` karena bertipe data vector sedangkan nilai `count` dijumlahkan biasa di baris 294(Gambar 4.7). Setelah dilakukan `reduceByKey`, hasilnya yang masih berupa pasangan *key* dan *value* ([K, V]) kemudian dilakukan fungsi `collectAsMap` agar berubah menjadi sebuah tipe data `map`.

Pada baris kode berikutnya dilakukan penghapusan variabel `bcCenter` yang berisi sebuah *broadcast variable*. Pada bagian akhir iterasi, centroid yang berada pada variabel `centers` akan diperbarui. Hal ini dilakukan dengan cara melakukan iterasi pada setiap center yang telah dihasilkan. Untuk mendapatkan nilai centroid yang baru, dilakukan menghitung rata-rata setiap fitur objek (nilai `sum` dibagi dengan nilai `count`). Nilai tersebut disimpan dalam bentuk Vektor Norm(`VectorWithNorm`) di variabel `newCenter`. Pada kode program operasi ini dilakukan dengan memanggil `Blas.scal` karena fungsi ini adalah fungsi perkalian untuk tipe data vektor. Nilai variabel `converged` dan nilai `cost` juga diupdate pada bagian ini. Nilai `converged` akan bernilai `false` jika nilai jarak euclidean antara center yang baru (`newCenter`) dan nilai centroid pada saat itu(`centers`) masih bernilai lebih besar dari kuadrat nilai `epsilon`.

```

299     // Update the cluster centers and costs
300     converged = true
301     totalContribs.foreach { case (j, (sum, count)) =>
302       scal(1.0 / count, sum)
303       val newCenter = new VectorWithNorm(sum)
304       if (converged && KMeans.fastSquaredDistance(newCenter, centers(j)) > epsilon * epsilon) {
305         converged = false
306       }
307       centers(j) = newCenter
308     }
309     cost = costAccum.value
310     iteration += 1
311   }
312 }
```

Gambar 4.8: Fungsi di Method `reduceByKey()` pada KMeans.scala

Setelah iterasi algoritma berakhir, kelas ini mencatat waktu iterasi. Kemudian nilai centroid akhir dikembalikan dalam bentuk `KmeansModel`.

#### 4.3 Modifikasi K-Means MLlib

Dengan mengetahui source code dari pada K-Means MLlib, penulis dapat memulai pengembangan dengan melakukan modifikasi pada source code K-Means. Pada bagian ini, penulis melakukan modifikasi pada source code KMeans MLlib agar dapat menghasilkan pola yang telah disebutkan pada kebutuhan pengembangan. Modifikasi yang akan dilakukan pada source code adalah dengan menambahkan bagian kode yang dapat melakukan operasi-operasi berikut:

1. Menghitung Jumlah anggota untuk setiap cluster
2. Menghitung nilai rata-rata setiap atribut/fitur objek untuk masing-masing cluster
3. Menghitung nilai minimum setiap atribut/fitur objek untuk masing-masing cluster
4. Menghitung nilai maximum setiap atribut/fitur objek untuk masing-masing cluster
5. Menghitung nilai standar deviasi setiap atribut/fitur objek untuk masing-masing cluster

Pola-pola tersebut dapat diatur sesuai keinginan pengguna. Sehingga dibutuhkan masukkan berupa konfigurasi pola yang diinginkan pengguna. Input konfigurasi pola dapat dilewatkan melalui parameter constructor kelas KMeans. Pada scala, parameter constructor dapat sekaligus mendefinisikan variabel global. Penulis memberi nama resultConf. Tipe data yang digunakan adalah `Array[boolean]`. Jika pola ingin ditampilkan maka nilainya true. jika tidak ingin menampilkan pola, ubah menjadi false. Urutan kofigurasi pola yaitu rata-rata fitur, nilai minimum fitur, nilai maximum fitur, dan standar deviasi fitur objek. Selain itu ada tambahan 2 parameter kelas yaitu `outputText` untuk menyimpan pola dalam bentuk teks dan `arrResult` untuk menyimpan pola dalam bentuk Array.

Untuk menghitung nilai rata-rata setiap fitur objek untuk masing-masing cluster tidak perlu tambahan kode program. Hal ini karena nilai centroid merupakan nilai rata-rata tersebut. Untuk nilai minimum dan nilai maksimum setiap fitur objek diperlukan variabel yang menyimpan nilai sementara atau nilai akhir. Sedangkan untuk nilai standar deviasi(<sup>3</sup>) dihitung menggunakan rumus berikut:

$$s = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}}$$

Rumus standar deviasi ini cukup rumit untuk diimplementasikan kedalam program. Maka dari itu rumus tersebut akan diturunkan menjadi berikut:

$$s = \sqrt{\frac{n(\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2)}{n(n - 1)}}$$

Rumus tersebut akan dibagi menjadi beberapa bagian perhitungan yaitu:

<sup>3</sup>

$$\sum_{i=1}^n x_i^2$$

$$\sum_{i=1}^n x_i$$

$$(\sum_{i=1}^n x_i)^2$$

- 4 Sebut saja bagian *a*, *b*, dan *c* (Atas ke bawah). Bagian *c* dapat dikatakan kuadrat dari nilai  
 5 pada bagian *b*. Sedangkan perhitungan bagian *b* adalah sama halnya dengan variabel sum. Sehingga  
 6 tidak perlu tambahan kode untuk perhitungan *b* dan *c* didalam iterasi. Hanya bagian *a* perlu di  
 7 implementasikan ke dalam kode program.

8

```

265   points.foreach { point =>
266     val (bestCenter, cost) = CustomKMeans.findClosest(thisCenters, point)
267     costAccum.add(cost)
268     val sum = sums(bestCenter)
269     BLAS.axpy(1.0, point.vector, sum) |
270     counts(bestCenter) += 1
271
272     // Minimum
273     var arrMin = new Array[Double](dims)
274     for( i <- 0 to dims-1){
275       arrMin(i) = math.min(point.vector.apply(i), mins(bestCenter).apply(i))
276     }
277     mins(bestCenter) = Vectors.dense(arrMin)
278
279     // Maximum
280     var arrMax = new Array[Double](dims)
281     for( i <- 0 to dims-1){
282       arrMax(i) = math.max(point.vector.apply(i), maxs(bestCenter).apply(i))
283     }
284     maxs(bestCenter) = Vectors.dense(arrMax)
285
286     // Deviasi
287     val dev = devs(bestCenter)
288     var arrDev = new Array[Double](dims)
289     for( i <- 0 to dims-1){
290       arrDev(i) = point.vector.apply(i) * point.vector.apply(i)
291     }
292     BLAS.axpy(1.0, Vectors.dense(arrDev), dev)
293   }
294   counts.indices.filter(counts(_) > 0).map(j => (j, (sums(j), counts(j), mins(j), maxs(j), devs(j)))).iterator
295 }.reduceByKey { case ((sum1, count1, min1, max1, dev1),

```

Gambar 4.9: Modifikasi untuk mendapatkan nilai minimum, maksimum, dan standar deviasi dalam iterasi objek(point)

- 9 Semua operasi perhitungan pola diletakkan pada method `mapPartitions`. Untuk menampung  
 10 nilai masing-masing operasi maka dibuat variabel *array* bertipe `vector`. `Vector` yang berada dalam  
 11 variabel *array* merepresentasikan nilai pada masing-masing atribut. Masing-masing variabel diberi  
 12 nama, `mins` untuk variabel minimum, `maxs` untuk variabel maksimum, dan `devs` untuk variabel  
 13 standar deviasi. Khusus variabel `mins` diinisialisasi dengan nilai tak terhingga. Hal ini bertujuan  
 14 untuk perbandingan nilai sehingga didapatkan nilai yang terkecil atau minimum.

2

- 3 Didalam fungsi `mapPartitions` terdapat iterasi yang mencakup seluruh objek didalam RDD ya-

itu `points.foreach(f: (A) => Unit)`. Lihat gambar 4.9. `foreach` merupakan method milik `iterator`. `Iterator` merupakan salah satu `data structure` yang berguna untuk mengaplikasikan iterasi pada element yang ada didalamnya. Parameternya akan mengaplikasikan sebuah fungsi pada setiap nilai yang diiterasi. `point` merepresentasikan element yang ada pada `points`. Fungsi didefinisikan setelah simbol '`=>`'.

Pada kode program bawaan KMeans sudah terdapat nilai penjumlahan setiap atribut yang disimpan dalam variabel `sum`. Selain itu, terdapat variabel `count` yang menyimpan jumlah anggota cluster. Sehingga nilai `sum` dan `count` ini dapat digunakan untuk menghitung nilai rata-rata. Untuk mendapat nilai minimum dan maksimum digunakan algoritma sederhana untuk membandingkan nilai atribut setiap objek dengan nilai yang ada pada variabel `mins` dan `maxs`. Mengimplementasikan rumus standar deviasi (Sebelumnya disebut bagian *a*) adalah dengan menghitung kuadrat dari nilai setiap fitur terlebih dulu. Kemudian nilai tersebut dijumlahkan dengan nilai `devs` (yang disimpan dalam variabel `dev`) pada baris 280 (Lihat gambar 4.9). Untuk melakukan itu digunakan method `axpy()`.

```

286 // Deviasi
287 val dev = devs(bestCenter)
288 var arrDev = new Array[Double](dims)
289 for( i <- 0 to dims-1){
290   arrDev(i) = point.vector.apply(i) * point.vector.apply(i)
291 }
292 BLAS.axpy(1.0, Vectors.dense(arrDev), dev)
293 }
294 counts.indices.filter(counts(_ > 0).map(j => (j, (sums(j), counts(j), mins(j), maxs(j), devs(j)))).iterator
295 ).reduceByKey { case ((sum1, count1, min1, max1, dev1),
296 (sum2, count2, min2, max2, dev2)) =>

```

Gambar 4.10: Nilai variabel modifikasi ditambahkan dalam kembalian

```

295   (sum1, count1 + count2, minRes, maxRes, dev1),
296   (sum2, count2, min2, max2, dev2)) =>
297   BLAS.axpy(1.0, sum2, sum1)
298   // Minimum dan Maximum atribut
299   var arrMin = new Array[Double](sum1.size)
300   var arrMax = new Array[Double](sum1.size)
301   for( i <- 0 to sum1.size-1){
302     arrMin(i) = math.min(min1.apply(i), min2.apply(i))
303     arrMax(i) = math.max(max1.apply(i), max2.apply(i))
304   }
305   var minRes: Vector = Vectors.dense(arrMin)
306   var maxRes: Vector = Vectors.dense(arrMax)
307   // Deviasi atribut
308   BLAS.axpy(1.0, dev2, dev1)
309
310   (sum1, count1 + count2, minRes, maxRes, dev1)
311 }.collectAsMap()

```

Gambar 4.11: Bagian kode untuk menghitung nilai akhir rata-rata, minimum, maksimum, dan standar deviasi

Isi fungsi dari parameter method `map` ini melakukan operasi pada setiap objek. Dengan demikian, seluruh variabel dikembalikan(`mins`, `maxs`, dan `devs`) melalui baris 282(Lihat gambar 4.10) sebagai pasangan *key* dan *value*. Setelah dilakukan method `mapPartition` kemudian hasilnya akan diteruskan kedalam fungsi `reduceByKey`(Gambar 4.11). Pada fungsi ini, dilakukan penggabungan

- 4 an nilai variabel dari objek-objek yang memiliki nilai *key* yang sama. Nilai **sum** dan **count** akan  
 5 dijumlahkan. Sedangkan nilai **min** dan **max** perlu dibandingkan kembali nilai terbesar dan terkecilnya.

6

```

316   //Update the cluster centers and costs
317   converged = true
318   totalContribs.foreach { case (j, (sum, count, min, max, dev)) =>
319     BLAS.scal(1.0 / count, sum)
320     val newCenter = new VectorWithNorm(sum)
321     if (converged && CustomKMeans.fastSquaredDistance(newCenter, centers(j)) > epsilon * epsilon) {
322       converged = false
323     }
324     centers(j) = newCenter
325     if (iteration == maxIterations || !converged) {...}
326   }
327   cost = costAccum.value
328   iteration += 1
  
```

Gambar 4.12: Method untuk memeriksa status converged

7 Setelah centroid baru telah didapatkan, **centers** kemudian diperbarui dengan menggunakan  
 8 method **foreach**(Gambar 4.12). Ini skaligus memeriksa status *converged* dan memperbaruiinya.  
 9 penulis juga menambahkan operasi untuk menyimpan pola yang telah dikumpulkan. Karena operasi  
 10 tersebut dilakukan pada akhir iterasi, maka dideklarasikan argumen percabangan pada baris ke-325  
 11 (Lihat gambar 4.12).

1 Isi dari baris kode tersebut dapat dilihat pada gambar 4.13. Pada bagian ini perhitungan  
 2 nilai rata-rata dan nilai standar deviasi diimplementasikan. Standar deviasi diimplementasikan  
 3 berdasarkan rumus turunan yang telah dijelaskan sebelumnya.

```

318   totalContribs.foreach { case (j, (sum, count, min, max, dev)) =>
319     BLAS.scal(1.0 / count, sum)
320     val newCenter = new VectorWithNorm(sum)
321     if (converged && CustomKMeans.fastSquaredDistance(newCenter, centers(j)) > epsilon * epsilon) {
322       converged = false
323     }
324     centers(j) = newCenter
325     if (iteration == maxIterations || !converged) {
326       // Print Cluster dan Jumlah Object
327       //println("\nKlaster Ke-: " + j)
328       //println("Jumlah Objek Klaster ke-" + j + " : " + count)
329
330       var dims = sum.size
331       var mins : Array[Double] = null
332       var maxs : Array[Double] = null
333       var mean : Array[Double] = null
334       var deviasi : Array[Double] = null
335
336       //If conf not set, Set all true
337       if(resultConf == null){
338         resultConf = Array.fill[Boolean](4)(true)
339       }
340       if(resultConf(0)){...}
341       if(resultConf(1)){...}
342       if(resultConf(2)){...}
343       if(resultConf(3)){...}
344       arrResult(j) = Array(Array(count), mean, mins, maxs, deviasi)
345     }
346   }
347   cost = costAccum.value
  
```

Gambar 4.13: Pola disimpan ke variabel global

```

340      if(resultConf(0)){...}
349      if(resultConf(1)){...}
357      if(resultConf(2)){...}
365      if(resultConf(3)){
366          deviasi = Array.fill[Double](dims)(0)
367          for (i <- 0 to dims - 1) {
368              deviasi(i) = scala.math.sqrt((count * dev(i)) - (sum.apply(i) * sum.apply(i))) / (count * (count - 1)))
369              println("Deviasi atribut "+ (i+1) + " : " + deviasi(i))
370              //this.outputText += "Deviasi atribut "+ (i+1) + " : " + deviasi(i)+"\n"
371          }
372      }
373      arrResult(j) = Array(Array(count), mean, mins, maxs, deviasi)

```

Gambar 4.14: Implementasi perhitungan formula Standar Deviasi

- 4 Pada akhirnya seluruh nilai disimpan dalam bentuk text berupa data String pada variabel global.
- 5 Data text tersebut akan disimpan dalam bentuk sebuah file teks (.txt). Operasi untuk menyimpan
- 6 dalam bentuk file berada pada sebuah method `printToFileTxt()`(Gambar 4.15). Sehingga hanya
- 7 method ini dipanggil pada baris terakhir.

8

```

37      /**
38      * Method untuk print out ke File
39      * @param inputText text that will printed out to file
40      * @return param
41      */
42      def printToFileTxt(inputText: String)={
43          // Menggunakan Java io
44          val pw = new PrintWriter(new File("OutputTextKMEANS.txt" ))
45          pw.write(inputText)
46          pw.close
47          (inputText)
48      }

```

Gambar 4.15: Method untuk menyimpan kedalam file (.txt)

- 9 Pada tahap ini, pengembangan yang dilakukan sudah dapat menghasilkan keluaran yang sesuai
- 10 dengan kebutuhan pengembangan. Pola keluaran dari KMeans ini hanya dapat diakses dari file teks
- 1 yang dihasilkan oleh method `printFileTxt()` yang dikembangkan. File ini berukuran sangat kecil
- 2 sehingga file ini akan disimpan didalam filesystem lokal. Oleh karena itu, untuk memuat kembali
- 3 hasil pola keluaran, pengguna harus mengambil dan membaca file tersebut.

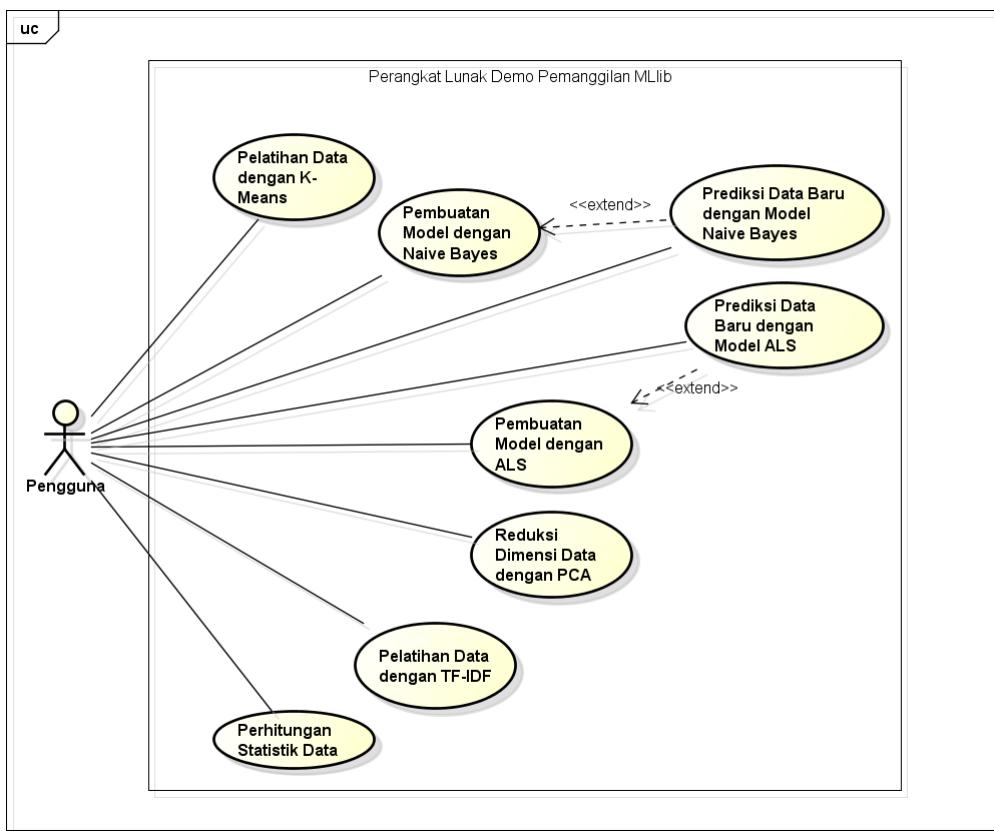
## BAB 5

### PERANCANGAN, IMPLEMENTASI, DAN PENGUJIAN PERANGKAT LUNAK DEMO

- 7 Pada bab ini, akan dijelaskan mengenai perancangan program dan implementasi pemanggilan  
8 Apache Spark MLlib dalam menganalisis Big Data. Perancangan meliputi implementasi 6 metode  
9 yang dapat dipakai pada library Apache Spark MLlib yaitu Naive Bayes, K-Means, Statistic,  
10 Principle Component Analysis, ALS, dan TF-IDF.

#### 11 5.1 Diagram Use Case dan Skenario

- 12 Diagram *Use Case* merupakan sebuah pemodelan untuk perilaku dari perangkat lunak yang akan  
13 dibuat. *Use Case* ini digunakan untuk mengetahui fungsi apa saja yang ada dalam perangkat lunak.  
14 Fungsi-fungsi dari Perangkat Lunak akan dioperasikan oleh satu pengguna. Pengguna memung-  
15 kinkan menggunakan beberapa fungsi yang disediakan Spark MLlib. Cara kerja dan perilaku dari  
1 Perangkat Lunak Demo akan dijelaskan dalam bentuk diagram *Use Case*. Diagram Use Case dapat  
2 dilihat pada Gambar 5.1.



Gambar 5.1: Diagram *use case* perangkat lunak demo

Berikut adalah skenario berdasarkan diagram Use Case (Gambar 5.1) :

1. Nama use case : Pelatihan Data dengan K-Means

- Aktor : Pengguna
- Pre-kondisi : set data yang akan diolah diletakkan pada cluster hadoop
- Pra-kondisi : pola disimpan pada komputer pengguna dan model hasil clustering disimpan di filesystem cluster hadoop
- Deskripsi : Fitur untuk menjalankan pelatihan set data dengan algoritma k-mean
- Langkah-langkah :
  - (a) Pengguna mengisi path input data dan output model
  - (b) Pengguna memilih jumlah cluster dan jumlah iterasi
  - (c) Pengguna memilih pola yang diinginkan
  - (d) Pengguna menekan tombol 'Generate Model'
  - (e) System melakukan pengolahan data dengan algoritma K-Means MLlib yang telah dimodifikasi pada cluster hadoop
  - (f) System menampilkan hasil berupa pola dari iterasi algoritma K-Means MLlib
  - (g) System menyimpan model berdasarkan path yang dimasukkan oleh pengguna

2. Nama use case : Pembuatan Model dengan Naive Bayes

- 4     ● Aktor : Pengguna
- 5     ● Pre-kondisi : set data yang akan diolah diletakkan pada cluster hadoop
- 6     ● Pra-kondisi : model hasil klasifikasi disimpan di filesystem cluster hadoop
- 7     ● Deskripsi : Fitur untuk menghasilkan model berdasarkan pelatihan set data dengan
- 8       algoritma naive bayes
- 9     ● Langkah-langkah :
  - 10       (a) Pengguna mengisi path input data dan output model
  - 11       (b) Pengguna memilih persentase dari data
  - 12       (c) Pengguna memilih pola yang diinginkan
  - 13       (d) Pengguna menekan tombol 'Generate Model'
  - 14       (e) System melakukan pengolahan data dengan algoritma Naive Bayes MLlib pada
  - 15       cluster hadoop
  - 16       (f) System menampilkan informasi hasil pengujian dan pembuatan model hasil pelatihan
  - 17       Naive Bayes MLlib
  - 18       (g) System menyimpan model pada path yang input pengguna

19     3. Nama use case : Pembuatan Model dengan ALS

- 20     ● Aktor : Pengguna
- 21     ● Pre-kondisi : set data yang akan diolah diletakkan pada cluster hadoop
- 22     ● Pra-kondisi : model hasil collaborative filtering disimpan di filesystem cluster hadoop
- 23     ● Deskripsi : Fitur untuk menghasilkan model berdasarkan pelatihan set data dengan
- 24       algoritma ALS
- 25     ● Langkah-langkah :
  - 26       (a) Pengguna mengisi path input data dan output model
  - 27       (b) Pengguna memilih jumlah rank dan iterasi
  - 28       (c) Pengguna menekan tombol 'Generate Model'
  - 29       (d) System melakukan pelatihan data dengan algoritma Alternating Least Square pada
  - 30       cluster hadoop
  - 31       (e) System menyimpan hasil pelatihan algoritma ALS pada path yang diinput pengguna
  - 32       (f) System menampilkan informasi terkait waktu pengolahan

33     4. Nama use case : Reduksi Dimensi Data dengan PCA

- 34     ● Aktor : Pengguna
- 35     ● Pre-kondisi : set data yang akan diolah diletakkan pada cluster hadoop
- 1     ● Pra-kondisi : data hasil reduksi algoritma PCA disimpan pada cluster hadoop
- 2     ● Deskripsi : Fitur digunakan untuk mereduksi dimensi pada data.
- 3     ● Langkah-langkah :

- 4                   (a) Pengguna mengisi path input data dan output model
- 5                   (b) Pengguna menentukan jumlah PC terbaik yang akan digunakan
- 6                   (c) Pengguna menekan tombol 'Compute PCA'
- 7                   (d) System melakukan perhitungan PC berdasarkan data masukkan dan memproyeksikan  
8                    data asli dengan PC pada cluster hadoop
- 9                   (e) System menyimpan hasil proyeksi data berdasarkan PC pada path yang diinput  
10                  pengguna
- 11                  (f) System menampilkan waktu eksekusi dan sampel hasil eksekusi

12       5. Nama use case : Pelatihan Data dengan TF-IDF

- 13                  • Aktor : Pengguna
- 14                  • Pre-kondisi : set data yang akan diolah diletakkan pada cluster hadoop
- 15                  • Pra-kondisi : data hasil perhitungan TF-IDF disimpan pada cluster hadoop
- 16                  • Deskripsi : Fitur ini digunakan untuk menghitung seberapa penting sebuah term atau  
17                  kata pada suatu dokumen
- 18                  • Langkah-langkah :
  - 19                   (a) Pengguna mengisi path input data dan output model
  - 20                   (b) Pengguna menentukan jumlah variasi term atau fitur pada document
  - 21                   (c) Pengguna menekan tombol 'Compute TFIDF'
  - 22                   (d) System melakukan pengolahan data dengan algoritma TFIDF pada cluster hadoop
  - 23                   (e) System menyimpan hasil pengolahan algoritma TFIDF pada path yang diinput  
24                  pengguna
  - 25                   (f) System menampilkan waktu eksekusi dan sampel hasil eksekusi

26       6. Nama use case : Perhitungan Statistik Data

- 27                  • Aktor : Pengguna
- 28                  • Pre-kondisi : set data yang akan diolah diletakkan pada cluster hadoop
- 29                  • Pra-kondisi : data hasil perhitungan statistik disimpan pada cluster hadoop
- 30                  • Deskripsi : Fitur ini untuk perhitungan statistik setiap fitur pada data.
- 31                  • Langkah-langkah :
  - 32                   (a) Pengguna memasukan path data masukkan dan path untuk menyimpan hasil perhi-  
33                   tungan
  - 34                   (b) Pengguna memilih perhitungan yang ingin dicari
  - 35                   (c) Pengguna menekan tombol 'Compute Statistic'
  - 1                   (d) System melakukan perhitungan statistik pada data di cluster
  - 2                   (e) System menampilkan hasil perhitungan statistic MLlib
  - 3                   (f) System menyimpan hasil perhitungan pada path yang dimasukkan oleh pengguna

4        7. Nama use case : Prediksi Data Baru dengan Model Naive Bayes

- 5              • Aktor : Pengguna
- 6              • Pre-kondisi : membuat model Naive Bayes dengan pelatihan data
- 7              • Pra-kondisi : mendapatkan prediksi kelas pada data baru
- 8              • Deskripsi : Fitur ini untuk memprediksi kelas pada data baru.
- 9              • Langkah-langkah :
- 10             (a) Pengguna memasukan path data baru sebagai masukkan dan path untuk menyimpan hasil prediksi
- 11             (b) Pengguna menekan tombol 'Predict'
- 12             (c) System melakukan prediksi kelas pada data baru
- 13             (d) System menyimpan hasil prediksi pada path yang ditentukan pengguna
- 14             (e) System menampilkan sample hasil prediksi dan waktu eksekusi

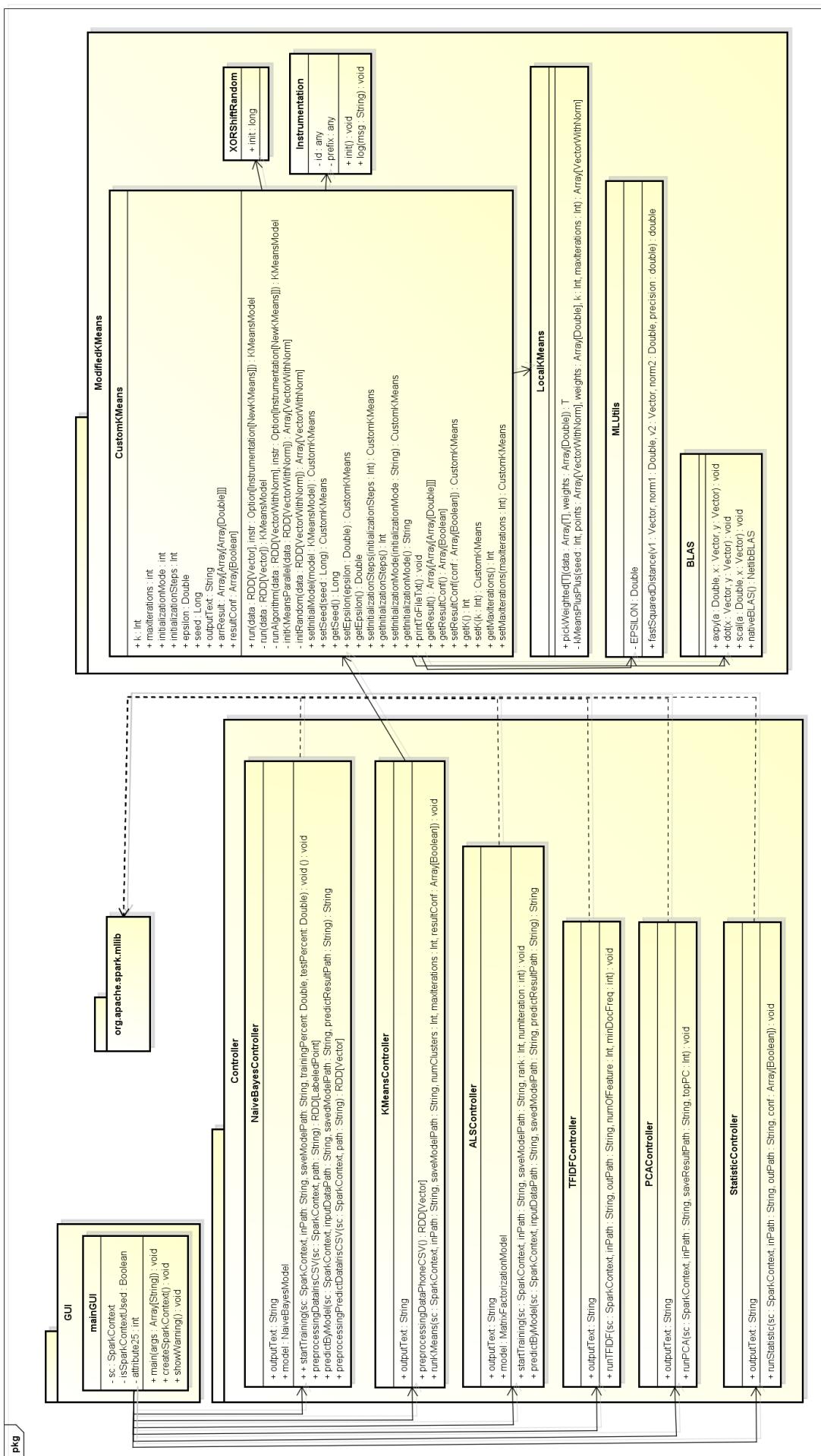
16        8. Nama use case : Prediksi Data Baru dengan Model ALS Statistik Data

- 17              • Aktor : Pengguna
- 18              • Pre-kondisi : set data yang akan diprediksi diletakkan pada cluster hadoop
- 19              • Pra-kondisi : mendapatkan prediksi produk pada data baru
- 20              • Deskripsi : Fitur ini untuk memprediksi produk pada suatu user pada data baru.
- 21              • Langkah-langkah :
- 22             (a) Pengguna memasukan path data baru sebagai masukkan dan path untuk menyimpan hasil prediksi
- 23             (b) Pengguna menekan tombol 'Predict'
- 24             (c) System melakukan prediksi product pada data baru
- 25             (d) System menyimpan hasil prediksi pada path yang ditentukan pengguna
- 26             (e) System menampilkan sample hasil prediksi dan waktu eksekusi

## 28        5.2 Diagram Kelas dan Pemanggilan Library MLlib

29        Pada bagian ini akan dijelaskan diagram kelas dari perangkat lunak. Dapat dilihat diagram kelas pada gambar (5.2) memiliki 3 package yang berbeda. Package tersebut meliputi package GUI, package controller, dan package ModifiedKMeans.

32  
33        Package GUI berisi kelas main yang digunakan untuk menjalankan tampilan perangkat lunak.  
34        Package controller digunakan untuk pemanggilan fungsi-fungsi algoritma MLlib. Sedangkan  
35        package ModifiedKMeans berisi kelas-kelas yang telah dimodifikasi untuk menjalankan fungsi K-  
36        Means dari library MLlib. Sebagian besar kelas-kelas pada perangkat lunak ini diimplementasikan  
1        kedalam sebuah object yang disebut singleton object pada scala. Hal ini memungkinkan untuk  
2        menggunakan suatu instansi dari kelas tanpa perlu menginisialisasi.



Gambar 5.2: Diagram kelas perangkat lunak demo

4 mainGUI adalah pusat dari perangkat lunak ini. GUI akan dijalankan menggunakan library  
5 Scala.swing di kelas ini. Setiap pengguna yang ingin menjalankan pengolahan data dengan teknik  
6 tertentu, kelas ini akan memanggil object controller yang dibutuhkan dan menjalankan fungsinya.  
7 Package GUI akan menggunakan package controller sebagai pengatur dalam pemanggilan fungsi-  
8 fungsi MLLib. Setiap kategori fungsi yang dipanggil memiliki controller masing-masing.

9

10 Package ModifiedKMeans merupakan sekelompok kelas yang dibutuhkan oleh KMeans ML-  
11 lib. Pada bab sebelumnya telah dibahas tentang modifikasi yang dilakukan pada kelas KMeans.  
12 Modifikasi dilakukan dengan memasukkan KMeans kedalam struktur perangkat lunak. Hal ini  
13 menyebabkan beberapa kelas yang dipanggil dari dalam KMeans juga harus diikutkan kedalam  
14 Perangkat Lunak. Namun tidak seluruhnya, hanya beberapa kelas yang tidak bisa diakses dari luar  
15 package library. Pada akhirnya, kelas KMeans diubah namanya menjadi CustomKMeans. Kelas  
16 inilah yang akan dipanggil oleh sebuah controller untuk menjalankan pelatihan data.

17

18 Package Controller merupakan kumpulan kelas-kelas yang melakukan pengolahan data dengan  
19 teknik tertentu. Kelas-kelas di dalam controller didefinisikan dengan object singleton sehingga dapat  
20 dipanggil tanpa perlu menginisialisasi. Fungsi-fungsi `machine learning` yang digunakan pada per-  
21 angkat lunak ini meliputi fungsi Naive Bayes, K-Means, Statistic, PCA, ALS, dan TF-IDF. Dengan  
22 demikian, penulis membagi kedalam beberapa object-object pada package controller. Berikut ini  
23 akan dijelaskan bagaimana cara kerja dari masing-masing object yang memanggil library MLLib.

24

## 25 Naive Bayes (NaiveBayesController)

26 Pemanggilan fungsi naive bayes MLLib dilakukan pada NaiveBayesController. Pada metode ini, ada  
27 dua fitur yang dapat dilakukan pengguna. Pertama, pengguna dapat membuat model naive bayes  
28 dari pelatihan data. Pembuatan model dilakukan pada method startTraining. Method ini menerima  
29 beberapa parameter yaitu sparkContext, path data masukkan, path untuk menyimpan model, per-  
30 sentase pelatihan, dan persentase pengujian. Path data yang digunakan adalah path pada filesystem  
31 hadoop HDFS. Data masukkan harus memiliki label kelas. Berikut adalah contoh isi dari data input:

32

33 Sebelum melakukan pelatihan pada data, dilakukan pembagian data kedalam dua buah data ya-  
34 itu data pelatihan dan pengujian. Pengujian dilakukan untuk mengukur tingkat akurasi dari model  
35 yang dihasilkan dari pelatihan data. Pembagian data dapat menggunakan method randomSplit  
36 yang disediakan oleh RDD. Method ini akan membagi data dengan komposisi yang diberikan pada  
37 parameter method tersebut.

38

39 `data.randomSplit(Array(trainingPercentage, testPercentage))` Setelah data dibagi menjadi dua  
40 bagian, data diteruskan kedalam pelatihan. Pelatihan dilakukan dengan menggunakan object  
1 NaiveBayes dari library MLLib. NaiveBayes memiliki method train untuk membuat model. Method  
2 ini mengembalikan sebuah object NaiveBayesModel.

3 `NaiveBayes.train(dataTraining, lambda = 1.0, modelType = "multinomial")`

4        Setelah model dibentuk, model akan disimpan kedalam hdfs sesuai dengan path yang dima-  
 5        sukkan pengguna. Penyimpanan model dilakukan dengan memanggil method save yang dimiliki  
 6        NaiveBayesModel. Method ini menerima parameter path lokasi penyimpanan model.

7  
 8        Fitur kedua yaitu, pengguna dapat melakukan prediksi kelas pada data yang belum diketahui  
 9        kelasnya. Prediksi dilakukan dengan menggunakan model NaiveBayesModel yang dihasilkan pada  
 10      fitur sebelumnya. Sehingga untuk menjalankan fitur ini, pengguna perlu membuat model terlebih  
 11      dahulu. Prediksi pada suatu data dapat memanggil method predict pada NaiveBayesModel.

12      `NaiveBayesModel.predict(data)`

13      Method ini mengembalikan data hasil prediksi dalam bentuk RDD. RDD memiliki method  
 14      `saveAsTextFile` untuk menyimpan data dalam bentuk file text. Method ini menerima parameter  
 15      berupa path lokasi tujuan penyimpanan yang dimasukkan pengguna.

16

## 17      K-Means (KMeansController)

18      Pemanggilan fungsi algoritma K-Means MLlib yang telah dimodifikasi dilakukan pada `KMeansController`.  
 19      Pada perangkat lunak ini, fungsi K-Means dikhkususkan untuk data aktivitas telepon genggam di  
 20      suatu kota.

21

22      Fungsi ini akan melakukan pelatihan data untuk mengelompokkan data. K-Means MLlib hanya  
 23      dapat melakukan pelatihan dengan data dengan kelas berupa data numerik, maka akan dilakukan  
 24      pra-pengolahan data terlebih dahulu. Pra-pengolahan adalah dengan mengambil fitur data yang  
 25      numerik. Fitur itu diantaranya adalah smsin, smsout, callin, callout, dan internet. Pra-pengolahan  
 26      ini dikerjakan didalam method `preprocessingDataPhoneCSV` pada `KMeansController`. Setelah itu  
 27      barulah data bisa dilatih.

28

29      Pelatihan data dilakukan didalam method `runKMeans`. Parameter yang diterima adalah  
 30      `SparkContext`, `inPath`, `saveModelPath`, `numClusters`, `maxIteration`, dan `resultConf`. Parameter  
 31      `numClusters`, `maxIteration`, dan `resultConf` diteruskan ke method pelatihan dengan algoritma  
 32      K-Means. Pelatihan dilakukan dengan memanggil method `train` pada object `CustomKMeans` dari  
 33      package `ModifiedKMeans`. Berikut syntaxnya:

34      `CustomKMeans.train(parsedData, numClusters, maxIterations, resultConf)`

35      Parameternya diantaranya adalah data yang sudah dilakukan pra-pengolahan, jumlah cluster  
 36      yang diinginkan, iterasi maksimum, dan pengaturan pola yang ingin dihasilkan. Pada bab sebe-  
 37      lumnya telah dibahas beberapa pola yang dapat dihasilkan oleh kelas `CustomKMeans`. Pola akan  
 38      disimpan dalam bentuk file .txt pada direktori aplikasi berjalan. Dan akan dibaca kembali oleh  
 1        perangkat lunak untuk ditampilkan ke tampilan antarmuka. Nilai Centroid juga ditampilkan di  
 2        GUI perangkat lunak.

3

4 **Statistic(StatisticController)**

5 Perhitungan statistic untuk fitur-fitur pada suatu data dengan memanggil library MLlib dilakukan  
6 pada StatisticController. Data masukkan untuk fitur ini adalah untuk fitur-fitur data yang bernilai  
7 numerik. Fitur ini menyediakan 5 perhitungan statistik yaitu perhitungan rata-rata, pencarian  
8 nilai minimum, pencarian nilai maksimum, dan perhitungan varian pada setiap fitur data. Statisti-  
9 cController menjalankan pemanggilan MLlib di dalam method runStatistic. runStatistic menerima  
10 parameter SparkContext, inPath, outPath, dan conf. Conf digunakan untuk memilih perhitungan  
11 statistic yang ingin ditampilkan. Di dalamnya runStatistic memanggil method colstat() milik object  
12 **Statistics** MLlib. Method ini menerima parameter RDD data yang akan dilakukan perhitungan  
13 statistik.

14 **Statistics.colStats(dataRDD)**

15 Serta mengembalikan suatu object MultivariateStatisticalSummary. MultivariateStatisticalSum-  
16 mary adalah object yang menyimpan nilai hasil perhitungan statistik berdasarkan data masukkan.

17 val summary: MultivariateStatisticalSummary = Statistics.colStats(dataRDD)  
18 summary.mean  
19 summary.min  
20 summary.max  
21 summary.variance

22 **PCA (PCAController)**

23 PCA adalah suatu teknik untuk mengurangi dimensi dari suatu data. Cara yang digunakan adalah  
24 dengan menghitung nilai Principal Component (PC). PCAController adalah yang bertugas mengatur  
25 pemanggilan fungsi reduksi dimensi data dengan teknik PCA. Pemanggilan fungsi tersebut berada  
26 pada method runPCA. Parameter runPCA yaitu SparkContext, inPath, saveResultPath, dan topPC.  
27 Perhitungan PC dilakukan dengan menggunakan method computePrincipalComponents yang ada  
28 pada RowMatrix. Maka dari itu perlu conversi data berbentuk RDD menjadi RowMatrix.

29 val mat: RowMatrix = new RowMatrix(dataRDD)  
30 mat.computePrincipalComponents(topPC)

31 computePrincipalComponents menerima parameter berupa jumlah PC terbaik yang diambil.  
32 Parameter ini dapat ditentukan oleh pengguna. Method ini mengembalikan nilai PC berupa Matrix.  
33 Untuk memproyeksikan data dengan nilai PC yang didapatkan, digunakan method multiply pada  
1 RowMatrix data masukkan. Hasil proyeksi data akan disimpan sesuai path yang dimasukkan oleh  
2 pengguna.

3 val projected: RowMatrix = mat.multiply(pc)

---

#### 4 ALS (ALSController)

5 Pemanggilan fungsi alternating least square MLlib dilakukan pada ALSController. Pada metode  
6 ini, ada dua fitur yang dapat dilakukan pengguna. Pertama, pengguna dapat membuat model  
7 alternatng dari pelatihan data. Pembuatan model dilakukan pada method startTraining. Method  
8 ini menerima beberapa parameter yaitu sparkContext, path data masukkan, path untuk menyimpan  
9 model, jumlah rank, dan jumlah iterasi. Data memiliki format konten yaitu memiliki 3 fitur  
10 diantaranya user, product, dan rate. Semua fitur merupakan data numerik.

11  
12 Sebelum melakukan pelatihan pada data, dilakukan konversi data bentuk Rating. Setelah  
13 data sudah dalam bentuk Rating, data diteruskan kedalam pelatihan. Pelatihan dilakukan dengan  
14 menggunakan object ALS dari `mlib.recommendation`. NaiveBayes memiliki method `train` untuk  
15 membuat model. Method ini mengembalikan sebuah object MatrixFactorizationModel.

16 `val model = ALS.train(ratings, rank, numIteration, 0.01)`

17 Setelah model dibentuk, model akan disimpan kedalam hdfs sesuai dengan path yang dima-  
18 sukan pengguna. Penyimpanan model dilakukan dengan memanggil method `save` yang dimiliki  
19 MatrixFactorizationModel. Method ini menerima parameter path lokasi penyimpanan model.

20  
21 Fitur kedua yaitu, pengguna dapat melakukan prediksi pada data yang belum diketahui rate-nya.  
22 Prediksi dilakukan dengan menggunakan model `MatrixFactorizationModel` yang dihasilkan pada  
23 fitur sebelumnya. Sehingga untuk menjalankan fitur ini, pengguna perlu membuat model terlebih  
24 dahulu. Prediksi pada suatu data dapat memanggil method `predict` pada `MatrixFactorizationModel`.

25 `MatrixFactorizationModel.predict(data)`

26 Method ini mengembalikan data hasil prediksi dalam bentuk RDD. RDD memiliki method  
27 `saveAsTextFile` untuk menyimpan data dalam bentuk file text. Method ini menerima parameter  
28 berupa path lokasi tujuan penyimpanan yang dimasukkan pengguna.

29

#### 30 TF-IDF(TFIDFController)

31 Perhitungan frequency sebuah term pada suatu data dilakukan dengan memanggil fungsi TF-IDF  
32 MLlib. Pemanggilan ini dilakukan pada TFIDFController. Data masukkan untuk fitur ini adalah  
33 untuk fitur-fitur data yang bernilai numerik. MLlib memiliki fungsi yang terpisah untuk melakukan  
34 perhitungan TF-IDF. Sehingga, perlu dilakukan perhitungan TF dilanjutkan perhitungan IDF.  
35 TFIDFController melakukan pemanggilan fungsi MLlib di dalam method `runTFIDF`. Pemanggilan  
36 fungsi menggunakan method `transform()` milik kelas HashingTF dan IDF, MLlib. Perhitungan  
37 TF dilakukan pada HashingTF dengan parameter data yang ingin diolah dalam bentuk RDD.  
1 Sedangkan perhitungan IDF dilakukan pada object IDF dengan parameter hasil perhitungan TF.

2 `hashingTF = new HashingTF()`  
3 `hashingTF.transform(documents)`

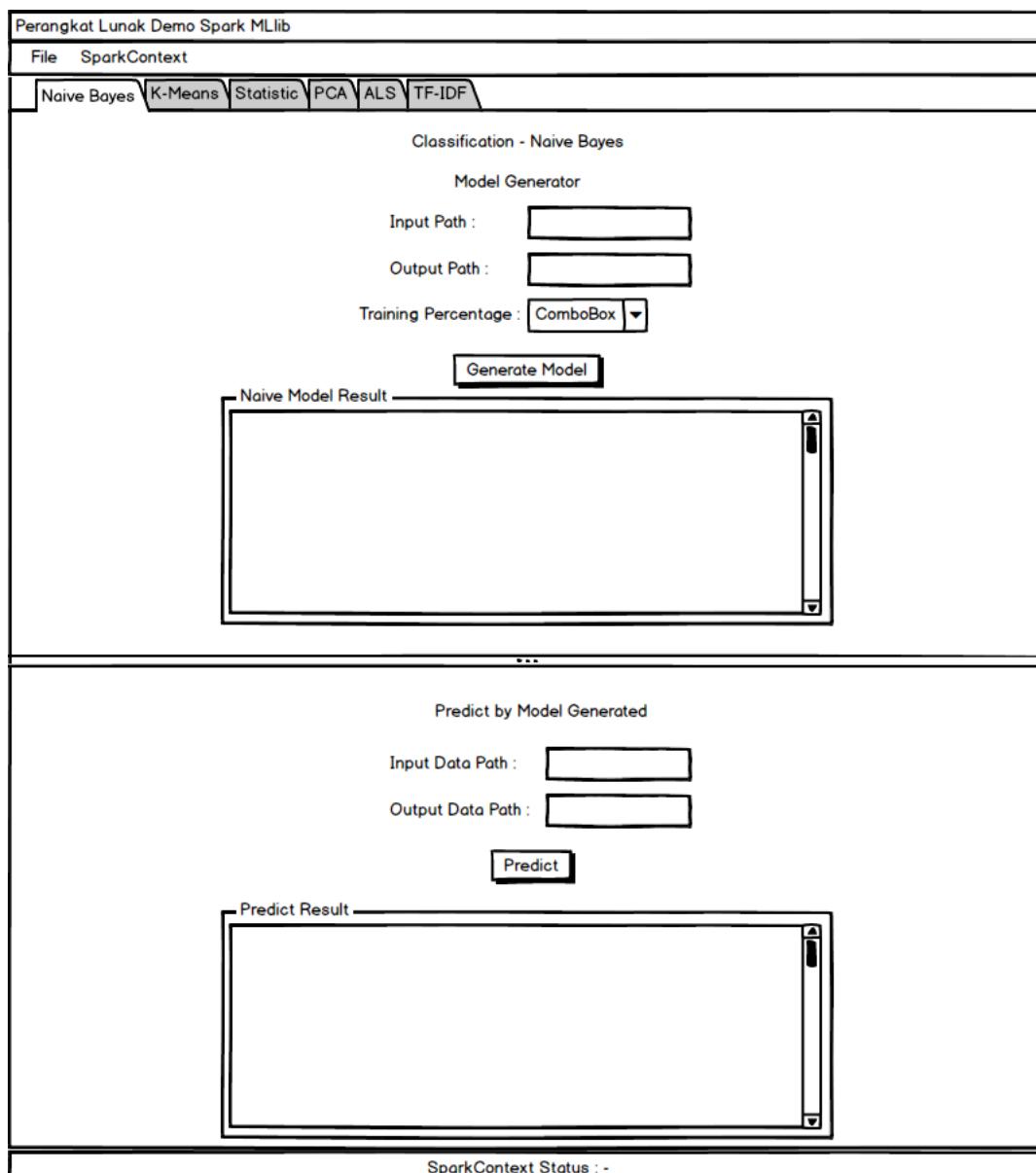
```
4  
5 idf = new IDF(minDocFreq = 2).fit(tf)  
6 val tfidf : RDD[Vector] = idf.transform(tf)
```

7 Parameter constructor minDocFreq adalah jumlah minimum document sebuah term ditemukan.  
8 Parameter tersebut tidak wajib ada. Hasil perhitungan berupa RDD vector yang disimpan pada  
9 path sesuai input pengguna.

## 10 5.3 Rancangan User Interface

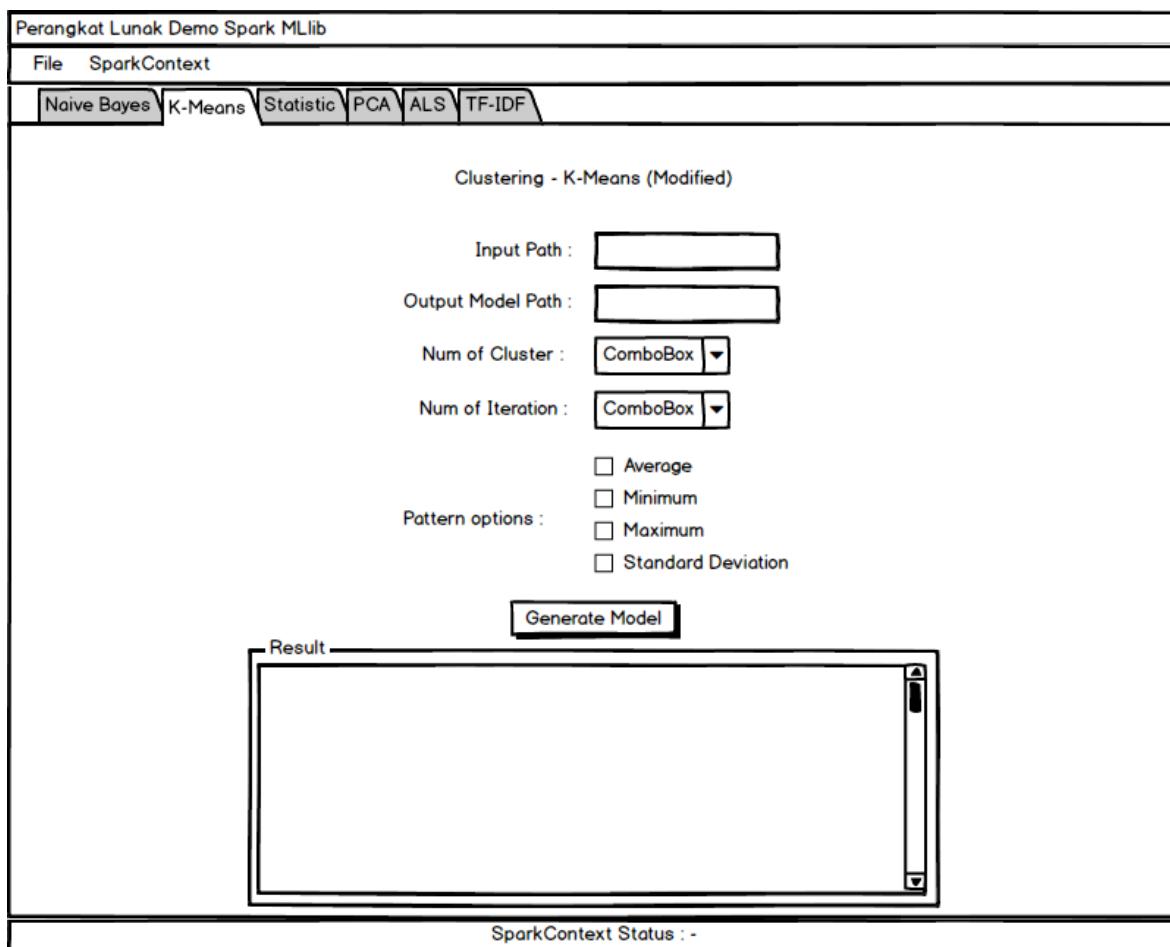
11 Berdasarkan kebutuhan fungsi-fungsi pada MLlib, perlu dirancang suatu antarmuka perangkat  
12 lunak demo. Pada halaman utama pada perangkat lunak ini yaitu berupa tampilan dari tab menu  
13 yang aktif. Tab menu bertujuan untuk memudahkan penggolongan fungsi-fungsi yang disediakan  
14 oleh MLlib. Halaman dari tab menu meliputi Naive Bayes, K-Means, Statistic, PCA, ALS, dan  
15 TF-IDF. Berikut akan dijelaskan rancangan setiap halaman tampilan:

16 1. Perancangan Antarmuka Fungsi Naive Bayes : Gambar 5.3 menunjukkan tampilan untuk  
17 fungsi naive bayes. Tampilan dibagi kedalam dua frame. Frame pertama menyediakan fitur  
18 pembuatan model naive bayes. Frame ini memiliki beberapa komponen textfield diantaranya  
19 adalah input path dan output model path. Text field input path berfungsi untuk menerima  
20 masukkan dari pengguna berupa path data yang ingin diolah. Output model path menerima  
21 masukkan berupa path untuk menyimpan model hasil pelatihan. Dibawahnya ada sebuah menu  
22 dropdown untuk menentukan persentase bobot pelatihan. persentase bobot untuk pengujian  
23 diambil dari bobot sisanya. Tombol Generate Model adalah tombol untuk menjalankan  
24 pelatihan. Informasi terkait hasil eksekusi ditampilkan dalam textarea di bawah tombol.  
25 Sedangkan frame satunya berada dibawah frame pertama. Frame kedua menyediakan fitur  
1 prediksi model. Pada bagian ini hanya ada 2 textfield untuk menerima masukkan berupa  
2 path data yang akan diprediksi dan path untuk menyimpan hasil rpediksi. Tombol predict  
3 untuk mengeksekusi prediksi pada data.



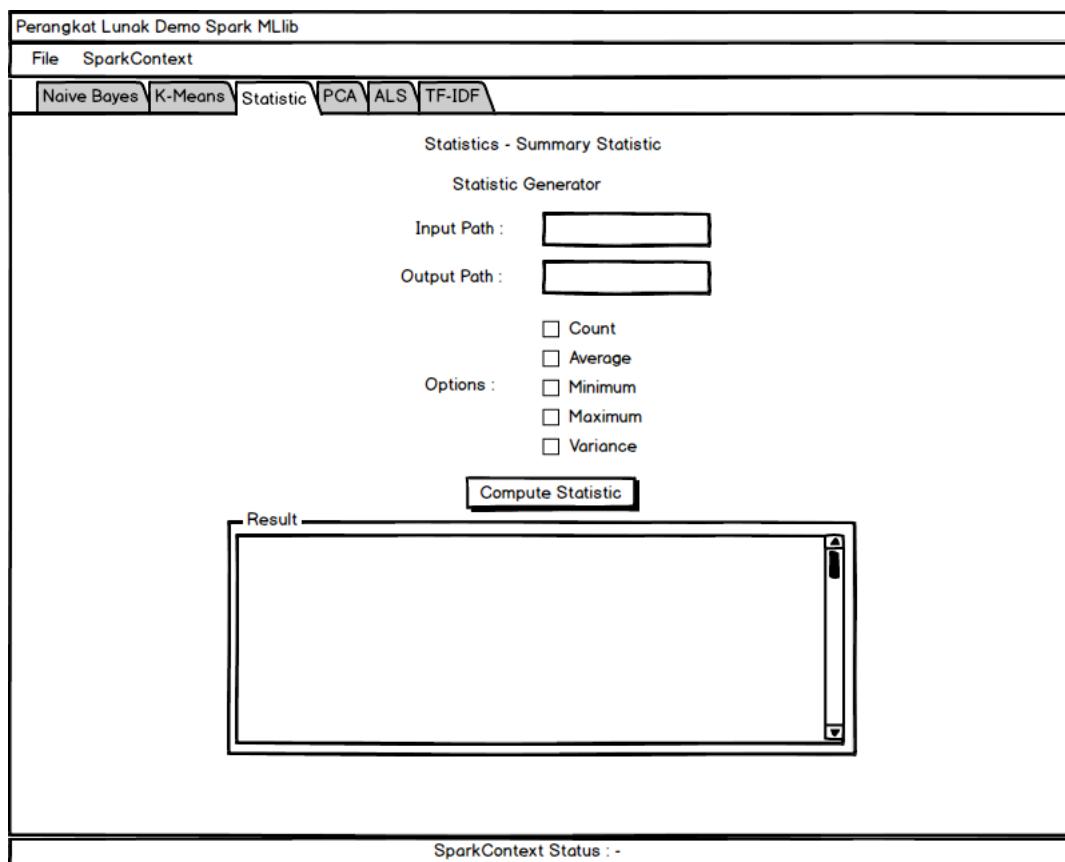
Gambar 5.3: Tampilan pengolahan data menggunakan algoritma Naive Bayes

- 4      2. Perancangan Antarmuka Fungsi K-Means : Gambar 5.4 menunjukan tampilan untuk fungsi  
 5      K-Means. Tampilan untuk fungsi K-Means juga dirancang untuk menerima parameter yang  
 1      dibutuhkan. Diantaranya adalah path input dan output, jumlah cluster, jumlah iterasi, dan  
 2      pola yang ingin ditampilkan. Informasi hasil eksekusi dan pola yang dihasilkan ditampilkan  
 3      di textarea *result*.



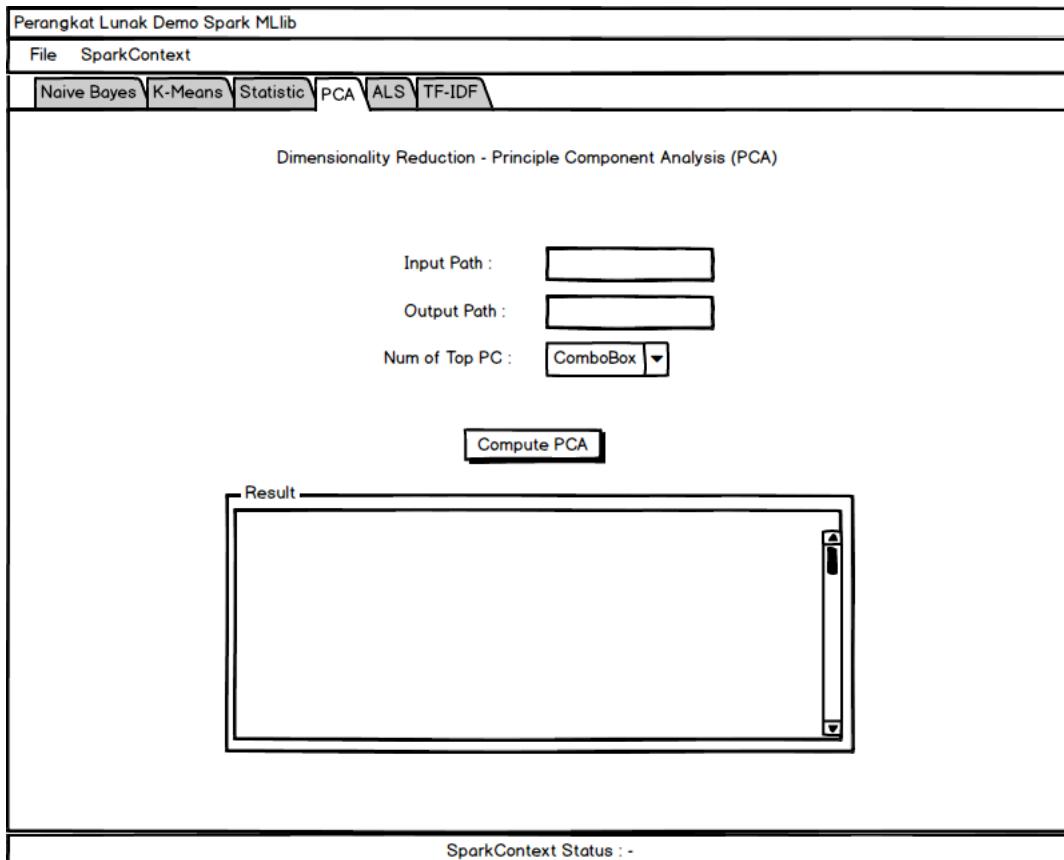
Gambar 5.4: Tampilan pengolahan data menggunakan K-Means

- 4     3. Perancangan Antarmuka Fungsi Statistik : Gambar ?? menunjukan tampilan untuk fungsi  
1       Statistik. Tampilan fungsi statistik menyediakan *field* untuk path data masukkan dan hasil  
2       perhitungan. Selain itu ada menu pilihan untuk menampilkan hasil perhitungan statistik yang  
3       dibutuhkan pengguna.



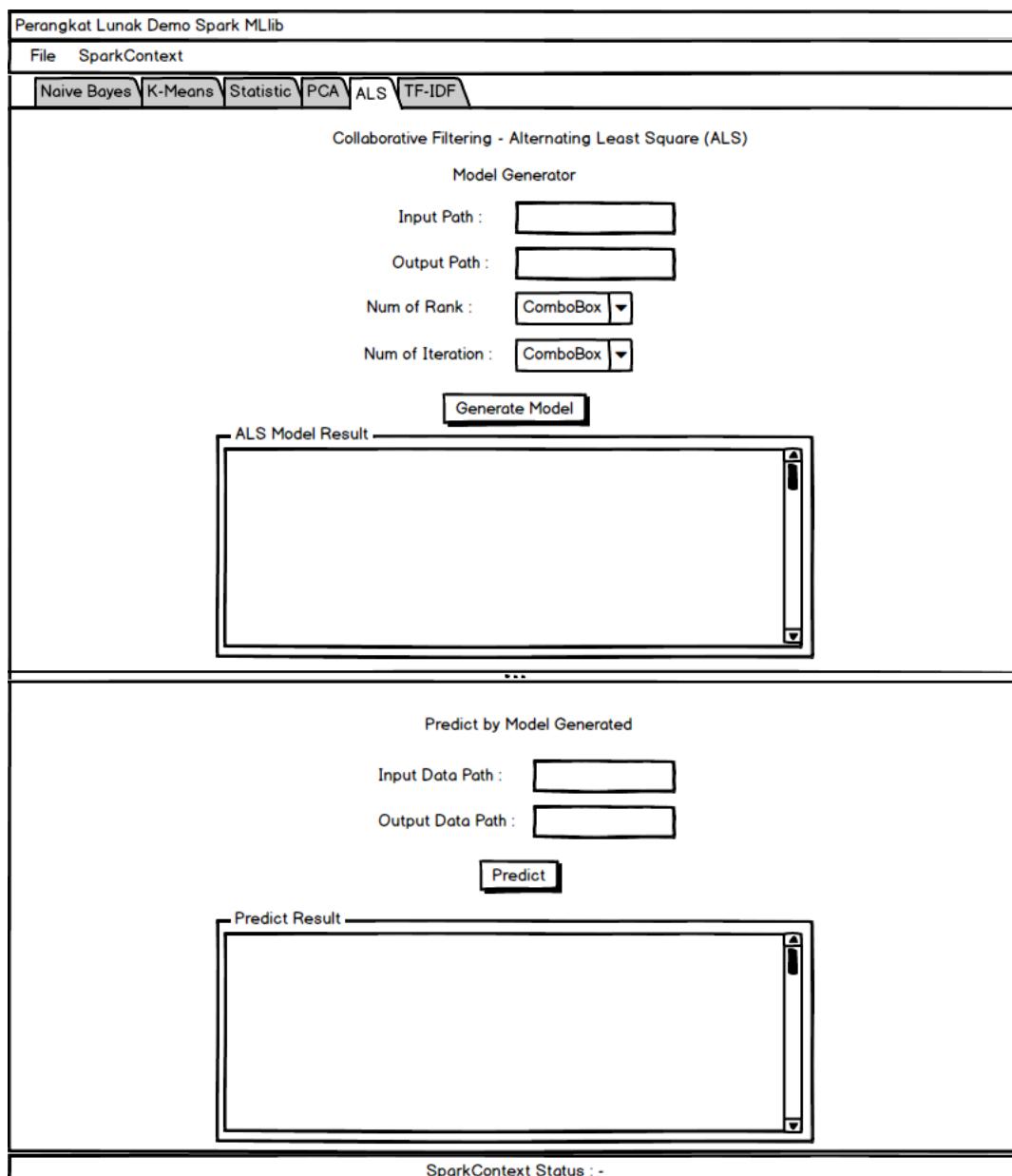
Gambar 5.5: Tampilan pengolahan data menggunakan metode Statistik

- 1 4. Perancangan Antarmuka Fungsi PCA : Gambar 5.6 menunjukkan tampilan untuk fungsi PCA.
- 2 Tampilan fungsi PCA menyediakan field untuk path data masukkan dan hasil komputasi.
- 3 Menu dropdown merupakan menu pilihan dropdown untuk jumlah PC yang digunakan.



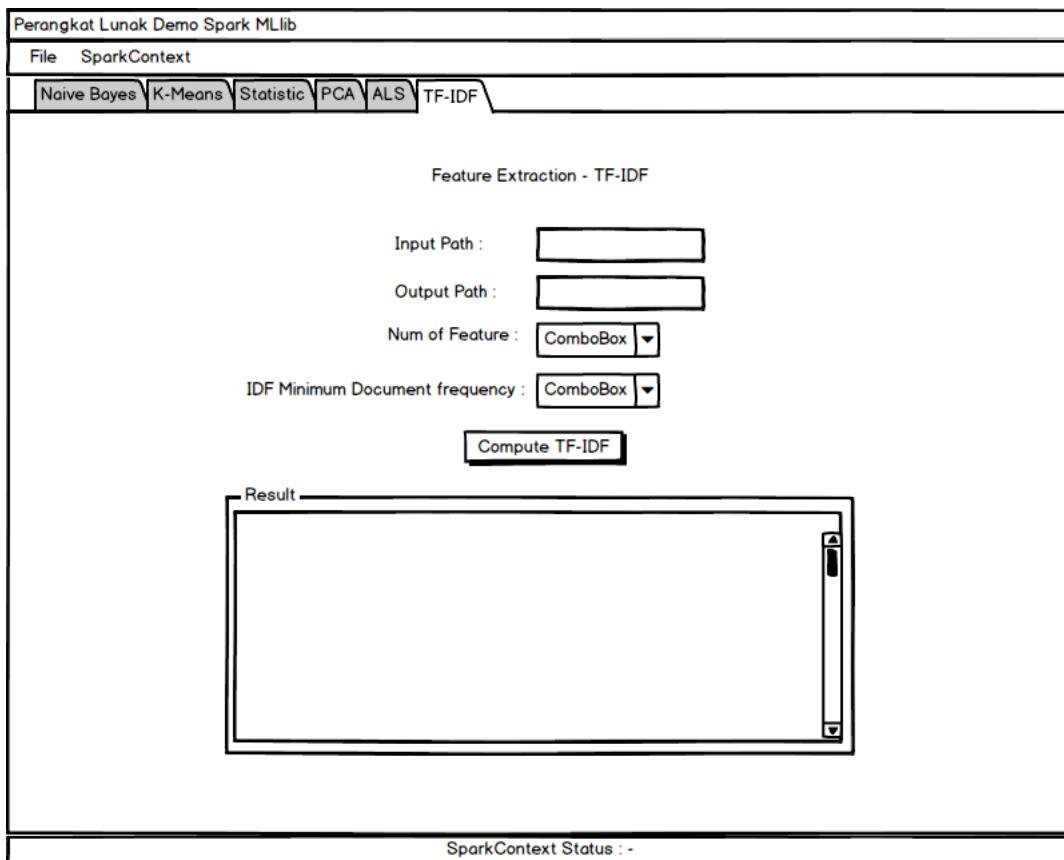
Gambar 5.6: Tampilan pengolahan data menggunakan algoritma PCA

- 1 5. Perancangan Antarmuka Fungsi ALS : Gambar 5.7 menunjukan tampilan untuk fungsi
- 2 Alternating Least Square. Tampilan pada fungsi ALS hampir sama seperti pada Naive Bayes.
- 3 Perbedaannya pada menu dropdown yang meliputi jumlah rank dan jumlah iterasi.



Gambar 5.7: Tampilan pengolahan data menggunakan algoritma ALS

- 4     6. Perancangan Antarmuka Fungsi TF-IDF : Gambar 5.8 menunjukkan tampilan untuk fungsi  
 1       Term Frequency - Inverse Document Frequency. menyediakan field untuk path data masukkan  
 2       dan hasil komputasi. Jumlah fitur dapat ditentukan menggunakan menu dropdown. Begitu  
 3       pula frequency minimum dokumen.



Gambar 5.8: Tampilan pengolahan data menggunakan algoritma TF-IDF

## **4 5.4 Implementasi Perangkat Lunak**

### **5 5.4.1 Lingkungan Perangkat Keras**

- 6 Perangkat keras yang digunakan dalam membangun perangkat lunak demo adalah sebuah notebook
- 7 Asus K401LB dengan spesifikasi :
- 8     • Processor : Intel i5 5200U @2.2GHz
- 9     • RAM : 12 GB DDR3
- 10    • VGA : NVIDIA GeForce 940M 2GB
- 11    • Harddisk : 1TB + 128GB SSD

### **12 5.4.2 Lingkungan Perangkat Lunak**

- 13 Perangkat lunak yang digunakan untuk membangun perangkat lunak demo antara lain sebagai berikut :
- 2     • Sistem Operasi : Windows 10 Pro 64-bit
- 3     • Bahasa Pemrograman : Scala

- IDE : IntelliJ IDEA 2017.3.2 (Community Edition)

- Versi Java : JDK 1.8.0\_141

- Versi SBT : SBT 1.0.2

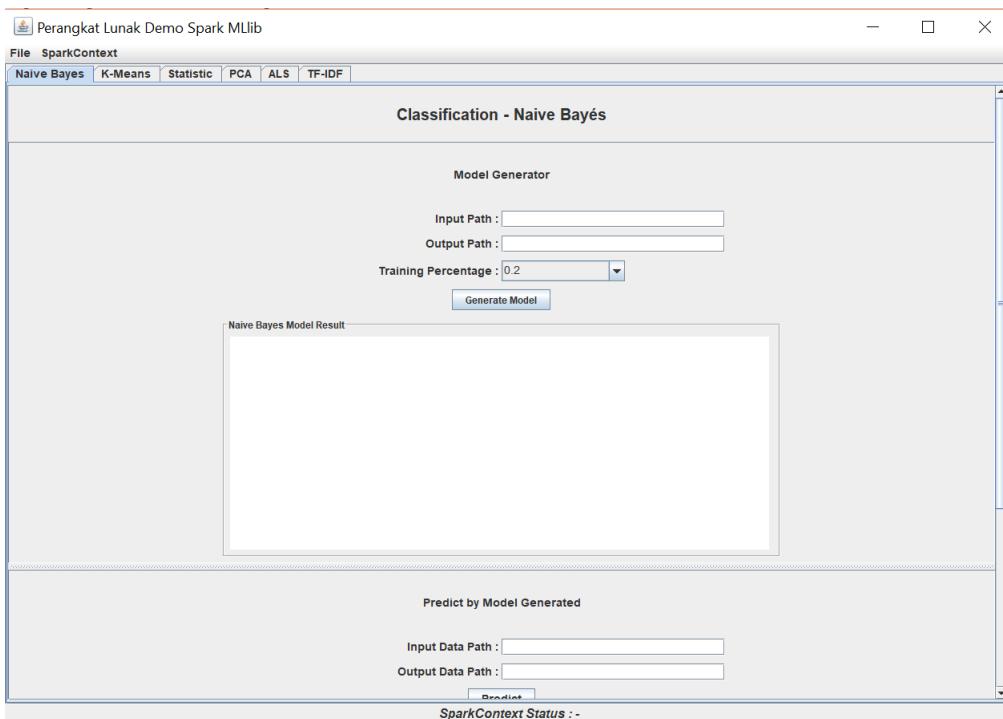
- Library Dependency :

- org.scala-lang:scala-library:2.11.8
- org.apache.spark:spark-core\_2.11:2.2.0
- org.apache.spark:spark-mllib\_2.11:2.2.0
- org.scala-lang:scala-swing:2.11.0-M7

### 5.4.3 User Interface

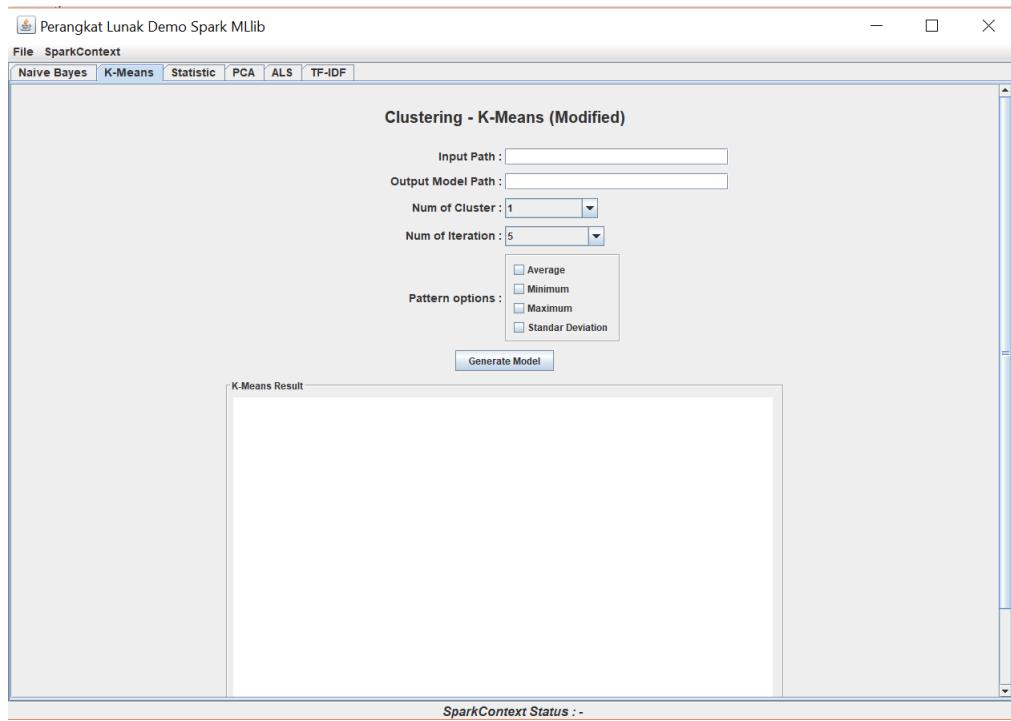
Implementasi rancangan tampilan antarmuka pada perangkat lunak ini menggunakan library dari Scala.swing. Berikut adalah tampilan masing-masing halaman fungsi:

1. Implementasi Antarmuka Fungsi Naive Bayes : Gambar 5.9 menunjukkan implementasi antarmuka untuk fungsi naive bayes.



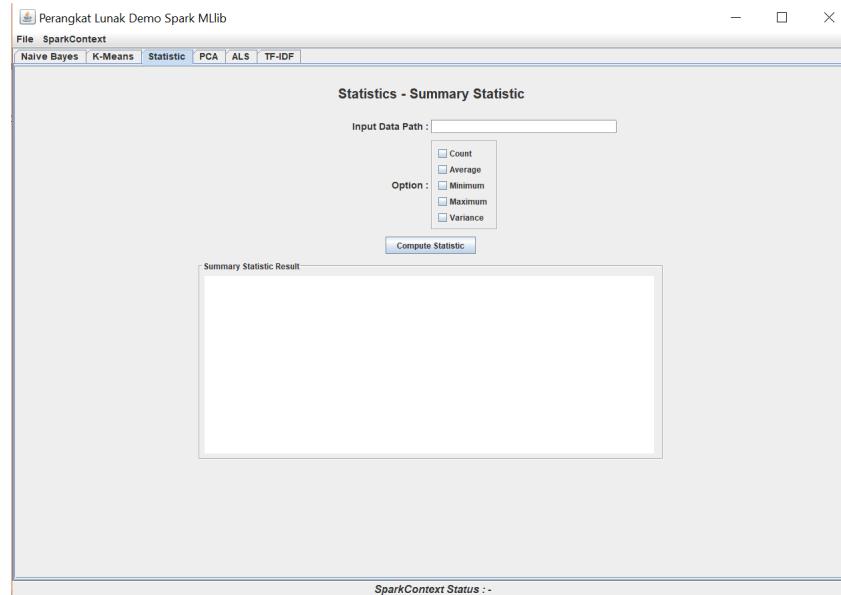
Gambar 5.9: Tampilan pengolahan data menggunakan algoritma Naive Bayes

2. Implementasi Antarmuka Fungsi K-Means : Gambar 5.10 menunjukkan implementasi antarmuka untuk fungsi K-Means.



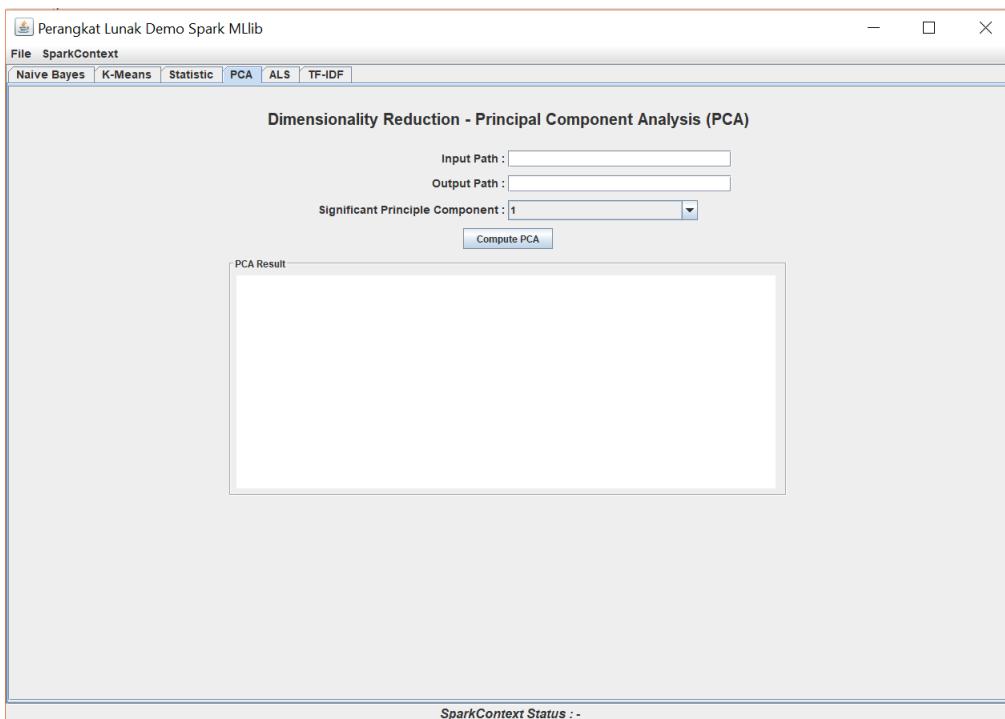
Gambar 5.10: Tampilan pengolahan data menggunakan K-Means

- 4 3. Implementasi Antarmuka Fungsi Statistik : Gambar 5.11 menunjukan implementasi antarmuka
- 1 untuk fungsi Statistik.



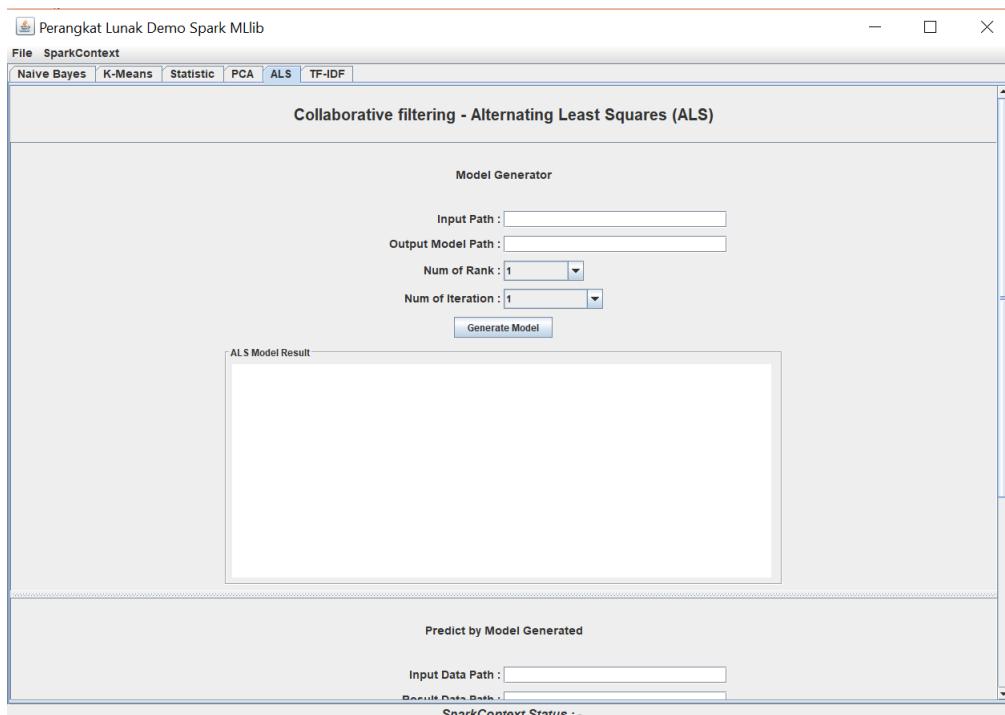
Gambar 5.11: Tampilan pengolahan data menggunakan metode Statistik

- 2 4. Implementasi Antarmuka Fungsi PCA : Gambar 5.12 menunjukan implementasi antarmuka
- 3 untuk fungsi PCA.



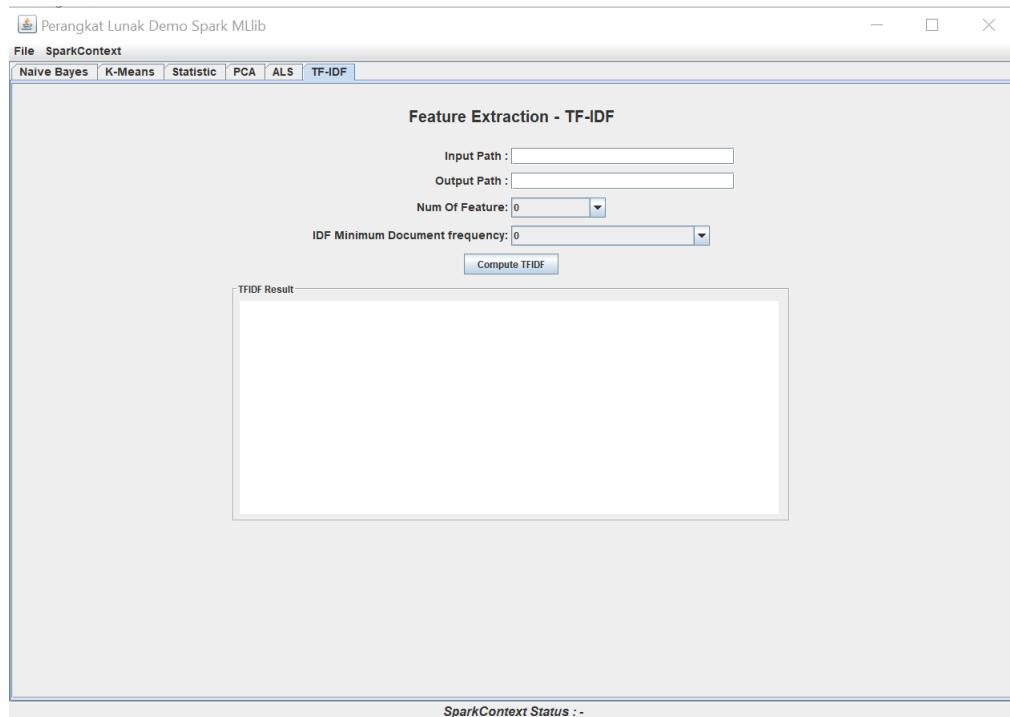
Gambar 5.12: Tampilan pengolahan data menggunakan algoritma PCA

- 4        5. Implementasi Antarmuka Fungsi ALS : Gambar 5.13 menunjukkan implementasi antarmuka  
1        untuk fungsi ALS.



Gambar 5.13: Tampilan pengolahan data menggunakan algoritma ALS

- 2        3        6. Implementasi Antarmuka Fungsi TF-IDF : Gambar 5.14 menunjukkan implementasi antarmuka  
 untuk fungsi TF-IDF.



Gambar 5.14: Tampilan pengolahan data menggunakan algoritma TF-IDF

#### 5.4.4 Implementasi Pemanggilan Fungsi MLlib

- 5 Berdasarkan perancangan, perangkat lunak ini memiliki fitur untuk menggunakan fungsi-fungsi MLlib. Pada diagram kelas telah dijelaskan method-method yang akan memanggil fungsi MLlib.
- 6 Berikut ini adalah potongan kode implementasi method yang memanggil fungsi-fungsi MLlib :

Listing 5.1: Pembuatan model dengan Naive Bayes

```

1 def startTraining(sc: SparkContext ,
2   inPath: String ,
3   saveModelPath: String ,
4   trainingPercent: Double ,
5   testPercent: Double): Unit = {
6
7
8   val data = preprocessingDataIrisCSV(sc , inPath)
9
10
11   val Array(training , test) = data .randomSplit(Array(trainingPercent , testPercent))
12
13   val initStartTrainingTime = System .nanoTime()
14   val modelNaive = NaiveBayes .train(training , lambda = 1.0 , modelType = "multinomial")
15   val trainingTimeInSecond = (System .nanoTime() - initStartTrainingTime)
16
17   val initStartTestTime = System .nanoTime()
18   val predictionAndLabel = test .map(p =>
19     (modelNaive .predict(p .features) , p .label)
20   )
21   val testTimeInSecond = (System .nanoTime() - initStartTestTime)
22   val accuracy = 1.0 * predictionAndLabel .filter(x => x ._1 == x ._2) .count() / test .count()
23   outputText += "Test Accuracy : " + accuracy + "\n"
24 }
```

```

23 modelNaive.save(sc, saveModelPath)
24 this.model = modelNaive
25 outputText = "Trained Model Saved! Location at "+ saveModelPath + '\n'
26 outputText += "Data Labels:\n"
27 this.model.labels.foreach(outputText += _ + '\n')
28
29 outputText += "\nSplit data into training (" + (trainingPercent * 100).toInt +
30   "%) and test (" + (testPercent * 100).toInt + "%)\n"
31 outputText += "Training Time: " + (trainingTimeInSecond / 1000000000.0) + "(Second)\n"
32 outputText += "Test Time: " + (testTimeInSecond / 1000000000.0) + "(Second)\n"
33 }

```

Listing 5.2: Prediksi data baru dengan model Naive Bayes

```

1 def predictByModel(sc: SparkContext,
2   inputDataPath: String,
3   savedModelPath: String,
4   predictResultPath: String): String ={
5
6   val data = preprocessingPredictDataIrisCSV(sc, inputDataPath)
7
8   if(this.model == null){
9     this.model = NaiveBayesModel.load(sc, savedModelPath)
10   }
11
12   val initStartPredictTime = System.nanoTime()
13   var result = this.model.predict(data)
14   val predictTimeInSecond = (System.nanoTime() - initStartPredictTime)
15
16   result.saveAsTextFile(predictResultPath)
17
18   var predictionResult: String = "Result saved at "+ predictResultPath +
19     "\nPrediction result:\n"
20   result.foreach(kelas => predictionResult += kelas + "\n")
21   predictionResult += "Prediction Time: " + (predictTimeInSecond / 1000000000.0) + "(Second)\n"
22   predictionResult
23 }

```

Listing 5.3: Pra-pengolahan data iris untuk pelatihan Naive Bayes

```

3 def preprocessingDataIrisCSV(sc: SparkContext,
4   path: String) : RDD[LabeledPoint] = {
5   val csv = sc.textFile(path)
6
7   val header = csv.first();
8
9   val data = csv.filter(_(0) != header(0));
10
11   data.map { line =>
12     val parts = line.split(',')
13     LabeledPoint(defineClassIris(parts(5)), Vectors.dense(
14       parts(1).toDouble,
15       parts(2).toDouble,
16       parts(3).toDouble,
17       parts(4).toDouble))
18   }
19 }

```

```

15     parts(4).toDouble)
16   }.cache()
17 }
```

Listing 5.4: Konversi nilai kelas menjadi numerik

```

1 def defineClassIris(label : String): Double = {
2   if(label == "Iris-setosa"){
3     1.0
4   }else if(label == "Iris-versicolor"){
5     2.0
6   }else {
7     3.0
8   }
9 }
```

Listing 5.5: Pelatihan data dengan K-Means

```

16 def runKMeans(sc: SparkContext ,
17   inPath: String ,
18   saveModelPath: String ,
19   numClusters: Int ,
20   maxIterations: Int ,
21   resultConf : Array[Boolean]): Unit ={ 
22   val parsedData = preprocessingDataPhoneCSV(sc , inPath)
23
24   val clusters = CustomKMeans.train(parsedData , numClusters , maxIterations , resultConf)
25
26   clusters.save(sc , saveModelPath)
27   val modelKMeans = KMeansModel.load(sc , saveModelPath)
28
29   outputText = Source.fromFile("OutputTextKMEANS.txt").mkString
30
31   outputText += "Cluster Centers:\n"
32   modelKMeans.clusterCenters.foreach(outputText += _ + "\n")
33 }
```

Listing 5.6: Pembuatan model dengan ALS

```

34 def preprocessingDataPhoneCSV(sc: SparkContext , path: String): RDD[Vector] ={ 
35   val csv = sc.textFile(path)
36
37   val header = csv.first();
38
39   val data = csv.filter(_(0) != header(0));
40
41   data.map { line =>
42     val parts = line.split(",",-1)
43     Vectors.dense(
44       parseDouble(parts(3)),
45       parseDouble(parts(4)),
46       parseDouble(parts(5)),
47       parseDouble(parts(6))
48     )
49   }
50 }
```

```
16     }.cache()
17 }
```

Listing 5.7: Perhitungan statistik pada data

```
1 def runStatistic(sc: SparkContext,
2     inPath: String,
3     outPath: String,
4     conf: Array[Boolean]): Unit = {
5     val data = sc.textFile(inPath)
6     val dataRDD = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble))).cache()
7
8     val initStartTrainingTime = System.nanoTime()
9     val summary: MultivariateStatisticalSummary = Statistics.colStats(dataRDD)
10    val trainingTimeInSecond = (System.nanoTime() - initStartTrainingTime)
11
12    var result = "";
13    if(conf(0)){
14        result = "Count: " + summary.count + "\n"
15    }
16    if(conf(1)){
17        result += "Mean: " + summary.mean + "\n"
18    }
19    if(conf(2)){
20        result += "Min: " + summary.min + "\n"
21    }
22    if(conf(3)){
23        result += "Max: " + summary.max + "\n"
24    }
25    if(conf(4)){
26        result += "Variance: " + summary.variance + "\n"
27    }
28    result += s"\nExecution Time: ${trainingTimeInSecond / 1000000000.0} seconds"
29    outputText = result
30    TextToFile.saveResultToTextFile("SummaryStatisticResult.txt", outputText)
31 }
```

Listing 5.8: Reduksi dimensi data dengan PCA

```
3 def runPCA(sc: SparkContext,
4     inPath: String,
5     saveResultPath: String,
6     topPC: Int): Unit = {
7     val rows = sc.textFile(inPath)
8     val dataRDD = rows.map(s => Vectors.dense(s.split(' ')
9         .map(_.toDouble)))
10    .cache()
11
12    val mat: RowMatrix = new RowMatrix(dataRDD)
13
14    val initStartTrainingTime = System.nanoTime()
15    val pc: Matrix = mat.computePrincipalComponents(topPC)
16    val trainingTimeInSecond = (System.nanoTime() - initStartTrainingTime)
```

```

16  val initProjectTime = System.nanoTime()
17  val projected: RowMatrix = mat.multiply(pc)
18  val finishProjectInSecond = (System.nanoTime() - initProjectTime)
19
20  projected.rows.saveAsTextFile(saveResultPath)
21
22  val collect = projected.rows.take(100)
23  var resultSample = ""
24  collect.foreach { vector => resultSample += vector + "\n" }
25
26  this.outputText = "Sample\u00d7Projected\u00d7Row\u00d7Matrix\u00d7principal\u00d7component:\u00d7\n"
27  this.outputText += resultSample
28  this.outputText += s"\n\u00d7Compute\u00d7PC\u00d7Time\u00d7:\u00d7${trainingTimeInSecond/\u00d71000000000.0}\u00d7seconds"
29  this.outputText +=
30    s"\n\u00d7Projecting\u00d7Data\u00d7Time\u00d7:\u00d7${finishProjectInSecond/\u00d7100000000.0}\u00d7seconds"
31  TextToFile.saveResultToTextFile("PCAResult.txt", outputText)
32 }
```

Listing 5.9: Pembuatan model dengan ALS

```

21 def startTraining(sc: SparkContext,
22   inPath: String,
23   saveModelPath: String,
24   rank: Int,
25   numIteration: Int): Unit = {
26   val data = sc.textFile(inPath)
27   val ratings = data.map(_.split(',') match { case Array(user, product, rate) =>
28     Rating(user.toInt, product.toInt, rate.toDouble)
29   })
30
31   ratings.persist()
32
33   val initStartTrainingTime = System.nanoTime()
34   val model = ALS.train(ratings, rank, numIteration, 0.01)
35   val trainingTimeInSecond = (System.nanoTime() - initStartTrainingTime)
36
37   model.save(sc, saveModelPath)
38   this.model = model
39   this.outputText += "Model\u00d7generated\u00d7at'" + saveModelPath + "'.\n"
40   this.outputText += s"\n\u00d7Execution\u00d7Time\u00d7:\u00d7${trainingTimeInSecond/\u00d71000000000.0}\u00d7seconds"
41 }
```

Listing 5.10: Prediksi data baru dengan model ALS

```

42 def predictByModel(sc: SparkContext,
43   inputDataPath: String,
44   savedModelPath: String,
45   predictResultPath: String): String ={
46   if(this.model == null){
47     this.model = MatrixFactorizationModel.load(sc, savedModelPath)
48   }
49
50   val data = sc.textFile(inputDataPath)
51
52   val predictions = data.map { line =>
53     val parts = line.split(',')
54     val user = parts(0).toInt
55     val product = parts(1).toInt
56     val rating = parts(2).toDouble
57
58     val prediction = this.model.predict(user, product)
59
60     s"$user,$product,$rating,$prediction"
61   }
62
63   predictions.saveAsTextFile(predictResultPath)
64 }
```

```

11  val usersProducts = data.map(_.split(',') match { case Array(user, product) =>
12    Rating(user.toInt, product.toInt)
13  })
14
15  val initStartPredictTime = System.nanoTime()
16  val predictions = model.predict(usersProducts).map { case Rating(user, product, rate) =>
17    (user, product, rate)
18  }
19  val predictTimeInSecond = (System.nanoTime() - initStartPredictTime)
20
21  predictions.saveAsTextFile(predictResultPath)
22
23  var predictionResult: String = "Result saved at " + predictResultPath +
24    "ALSResult/" + "\nSample hasil prediksi (top 100):\n"
25  predictions.takeOrdered(100).foreach { case (user, product, rate) =>
26    predictionResult += user + "\n"
27  }
28  predictionResult += s"\nPrediction Time: ${predictTimeInSecond / 1000000000.0} seconds"
29  TextToFile.saveResultToTextFile("ALSResult.txt", predictionResult)
30  predictionResult
31 }

```

Listing 5.11: Menghitung TF-IDF pada data

```

2 def runTFIDF(sc: SparkContext,
3   inPath: String,
4   outPath: String,
5   numOffeature: Int,
6   minDocFreq: Int): Unit = {
7   val documents: RDD[Seq[String]] = sc.textFile(inPath)
8     .map(_.split(" ")).toSeq
9
10  var hashingTF: HashingTF = null
11  if (numOffeature == 0){
12    hashingTF = new HashingTF()
13  } else{
14    hashingTF = new HashingTF(numOffeature)
15  }
16
17  val initStartTFTime = System.nanoTime()
18  val tf: RDD[Vector] = hashingTF.transform(documents)
19  val tfTimeInSecond = (System.nanoTime() - initStartTFTime)
20
21  tf.cache()
22
23  var idf : IDFModel = null;
24  if(minDocFreq != 0){
25    new IDF(minDocFreq = 2).fit(tf)
26  } else{
27    idf = new IDF().fit(tf)
28  }
29  val initStartTFIDFTime = System.nanoTime()

```

```
30  val tfidf: RDD[Vector] = idf.transform(tf)
31  val tfidfTimeInSecond = (System.nanoTime() - initStartTFIDFTime)
32
33  tfidf.saveAsTextFile(outPath)
34
35  outputText = s"TF-IDF result save in '$outPath'\n"
36  outputText += "Sample TF-IDF (top 100):\n"
37  tfidf.take(100).foreach(x => outputText += x + "\n")
38  outputText += s"TF Execution Time: ${tfTimeInSecond / 1000000000.0} seconds.\n"
39  outputText += s"TF-IDF Execution Time: ${tfidfTimeInSecond / 1000000000.0} seconds.\n"
40  TextToFile.saveResultToTextFile("TFIDFResult.txt", outputText)
41 }
```

## 16 5.5 Pengujian Perangkat Lunak

17 Pengujian yang dilakukan adalah pengujian fungsional dari perangkat lunak. Pengujian fungsional  
18 adalah pengujian yang dilakukan dengan mengeksekusi program dan kemudian diamati apakah  
19 hasil sesuai dengan yang diinginkan. Hal ini bertujuan untuk memastikan bahwa setiap fungsi pada  
20 perangkat lunak sudah berjalan sesuai dengan use case. Setiap fitur dari perangkat lunak diuji  
21 dengan menggunakan data dengan format tertentu. Data yang digunakan hanya memiliki ukuran  
1 kecil dengan beberapa kilobyte saja. Hal ini dilakukan hanya pada pengujian ini. Berikut ini adalah  
2 hasil pengujian fungsional dari perangkat lunak demo:

3

Tabel 5.1: Hasil Pengujian Fungsional Perangkat Lunak

No	Langkah Pengujian	Hasil yang Diharapkan	Hasil Pengujian	Status
1	Membuat model Naive Bayes	Menyimpan model pada path yang ditentukan	Menyimpan model pada path yang ditentukan	OK
2	Memprediksi data baru dengan sebuah model clustering Naive Bayes	Menyimpan data hasil prediksi pada path yang ditentukan dan menampilkan sampel hasil prediksi	Menyimpan data hasil prediksi pada path yang ditentukan dan menampilkan sampel hasil prediksi	OK
3	Membuat model dan pola klasifikasi K-means	Model tersimpan pada path yang ditentukan dan menampilkan pola	Model tersimpan pada path yang ditentukan dan menampilkan pola	OK
4	Menghitung nilai statistik	Menyimpan data hasil perhitungan pada path yang ditentukan dan menampilkan sampel hasil perhitungan	Menyimpan data hasil perhitungan pada path yang ditentukan dan menampilkan sampel hasil perhitungan	OK
5	Melakukan reduksi data dengan PCA	Menyimpan data hasil komputasi pada path yang ditentukan dan menampilkan sampel hasil komputasi	Menyimpan data hasil komputasi pada path yang ditentukan dan menampilkan sampel hasil komputasi	OK
6	Membuat model ALS	Menyimpan model pada path yang ditentukan	Menyimpan model pada path yang ditentukan	OK
7	Memprediksi data baru dengan sebuah model ALS	Menyimpan data hasil prediksi pada path yang ditentukan dan menampilkan sampel hasil prediksi	Menyimpan data hasil prediksi pada path yang ditentukan dan menampilkan sampel hasil prediksi	OK
8	Melakukan perhitungan TF-IDF	Menyimpan data hasil perhitungan pada path yang ditentukan dan menampilkan sampel hasil perhitungan	Menyimpan data hasil perhitungan pada path yang ditentukan dan menampilkan sampel hasil perhitungan	OK

Pengujian dilakukan menggunakan format data masukkan yang berbeda-beda untuk setiap fitur perangkat lunak. Berikut akan dijelaskan data masukkan untuk setiap fiturnya:

1. Data masukkan untuk fitur Naive Bayes adalah dataset bunga iris. Gambar 5.15 menunjukkan format penulisan dataset. Setiap baris adalah representasi setiap objek. Fitur setiap objek meliputi **SepalLengthCm**, **SepalWidthCm**, **PetalLengthCm**, **PetalWidthCm**, dan **Species** sebagai kelasnya. Kelas yang dimiliki ada 3 jenis yaitu Iris-setosa, Iris-versicolor, dan Iris-virginica.

```
Id, SepalLengthCm, SepalWidthCm, PetalLengthCm, PetalWidthCm, Species
1, 5.1, 3.5, 1.4, 0.2, Iris-setosa
2, 4.9, 3.0, 1.4, 0.2, Iris-setosa
3, 4.7, 3.2, 1.3, 0.2, Iris-setosa
4, 4.6, 3.1, 1.5, 0.2, Iris-setosa
5, 5.0, 3.6, 1.4, 0.2, Iris-setosa
6, 5.4, 3.9, 1.7, 0.4, Iris-setosa
7, 4.6, 3.4, 1.4, 0.3, Iris-setosa
```

Gambar 5.15: Data masukkan untuk fitur naive bayes

- 4       2. Data masukkan untuk fitur K-Means adalah dataset aktivitas telepon genggam. Gambar  
 5       5.16 menunjukkan format penulisan dataset. Setiap baris adalah representasi setiap waktu  
 6       dan identitas tertentu. Fitur setiap objek meliputi `datetime`, `CellID`, `countrycode`, `smsin`,  
 7       `smsout`, `callin`, `callout`, dan `internet`.

```
datetime,CellID,countrycode,smsin,smsout,callin,callout,internet
2013-11-07 00:00:00,1,0,0.2463,,,
2013-11-07 00:00:00,1,39,1.4627,1.661,0.2999,0.1644,55.8591
2013-11-07 00:00:00,2,0,0.2467,,,
2013-11-07 00:00:00,2,39,1.4768,1.6736,0.3033,0.1648,56.0243
2013-11-07 00:00:00,3,0,0.2471,,,
2013-11-07 00:00:00,3,39,1.4918,1.687,0.3071,0.1652,56.2002
2013-11-07 00:00:00,4,0,0.2453,,,
2013-11-07 00:00:00,4,39,1.4218,1.6246,0.2897,0.1634,55.3806
2013-11-07 00:00:00,5,0,0.2189,,,
2013-11-07 00:00:00,5,39,1.2681,1.4635,0.2683,0.1449,50.3968
2013-11-07 00:00:00,6,0,0.2471,,,
```

Gambar 5.16: Data masukkan untuk fitur k-means

- 8       3. Data masukkan untuk fitur Statistik adalah dataset berupa data numerik acak. Gambar 5.17  
 9       menunjukkan format penulisan dataset. Setiap baris adalah representasi objek yang memiliki  
 10      4 fitur bernilai numerik.
- 11      4. Data masukkan untuk fitur PCA sama seperti fitur statistik yaitu dataset berupa data numerik  
 12      acak. Gambar 5.17 menunjukkan format penulisan dataset. Setiap baris adalah representasi  
 13      objek yang memiliki 4 fitur bernilai numerik.

```
5.1 3.5 1.4 0.2
4.9 3 1.4 0.2
4.7 3.2 1.3 0.2
4.6 3.1 1.5 0.2
5 3.6 1.4 0.2
5.4 3.9 1.7 0.4
4.6 3.4 1.4 0.3
5 3.4 1.5 0.2
4.4 2.9 1.4 0.2
4.9 3.1 1.5 0.1
5.4 3.7 1.5 0.2
4.8 3.4 1.6 0.2
4.8 3 1.4 0.1
4.3 3 1.1 0.1
5.8 4 1.2 0.2
```

Gambar 5.17: Data masukkan untuk fitur Statistik dan PCA

- 14      5. Data masukkan untuk fitur ALS adalah dataset berupa data *rating* pelanggan terhadap suatu  
 1       produk. Gambar 5.18 menunjukkan format penulisan dataset. Setiap baris adalah representasi  
 2       pengguna yang memiliki *rating* terhadap suatu produk. Fitur setiap objek meliputi pelanggan,  
 3       produk, dan *rating*.

```
1,1,5.0
1,2,1.0
1,3,5.0
1,4,1.0
2,1,5.0
2,2,1.0
2,3,5.0
2,4,1.0
3,1,1.0
```

Gambar 5.18: Data masukkan untuk fitur ALS

- 4      6. Data masukkan untuk fitur TF-IDF adalah dataset berupa data numerik. Gambar 5.19  
 5      menunjukkan format penulisan dataset. Setiap baris adalah representasi dokumen yang  
 6      memiliki beberapa term.

```

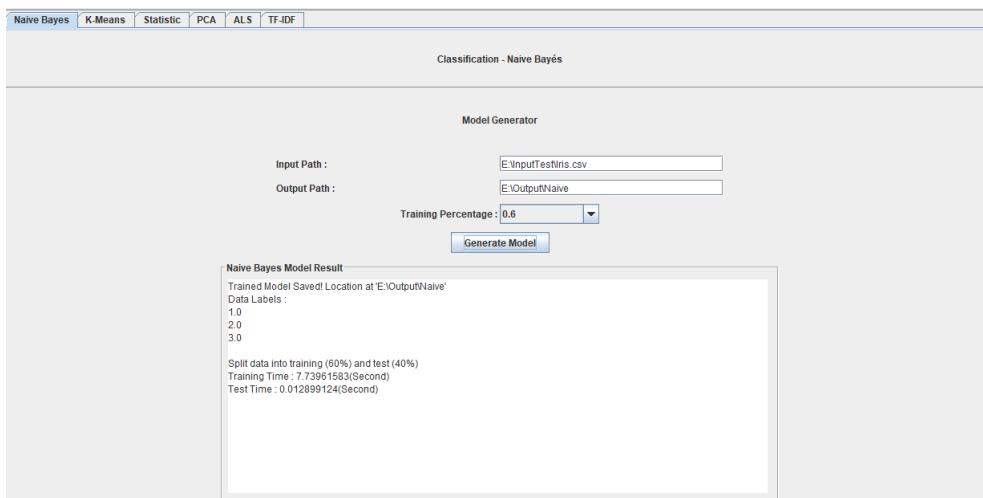
1 2 6 0 2 3 1 1 0 0 3
1 3 0 1 3 0 0 2 0 0 1
1 4 1 0 0 4 9 0 1 2 0
2 1 0 3 0 0 5 0 2 3 9
3 1 1 9 3 0 2 0 0 1 3
4 2 0 3 4 5 1 1 1 4 0
2 1 0 3 0 0 5 0 2 2 9
1 1 1 9 2 1 2 0 0 1 3
4 4 0 3 4 2 1 3 0 0 0
2 8 2 0 3 0 2 0 2 7 2
1 1 1 9 0 2 2 0 0 3 3

```

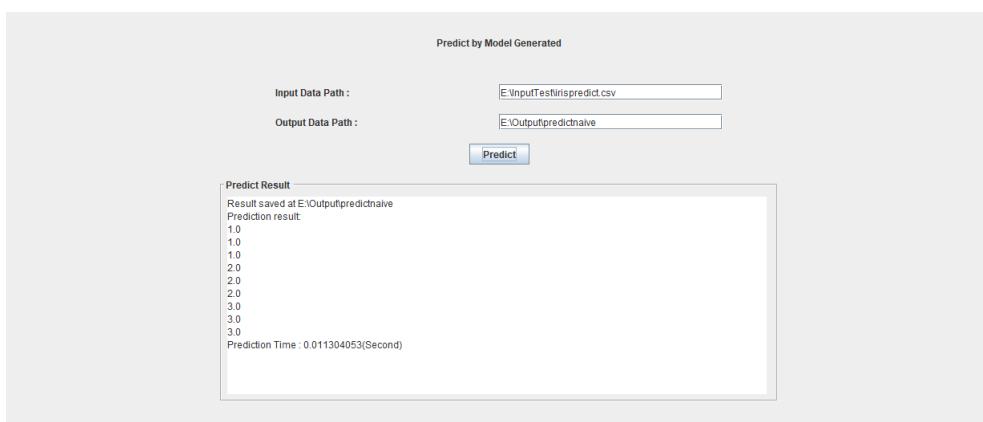
Gambar 5.19: Data masukkan untuk fitur TF-IDF

7      Berikut ini adalah hasil keluaran masing-masing fitur yang ditampilkan pada *user interface*  
 1      perangkat lunak:

- 2      1. Hasil pengujian fungsional Naive Bayes : Gambar 5.20 dan gambar 5.21 menunjukkan keluaran  
 3      untuk fungsi naive bayes.

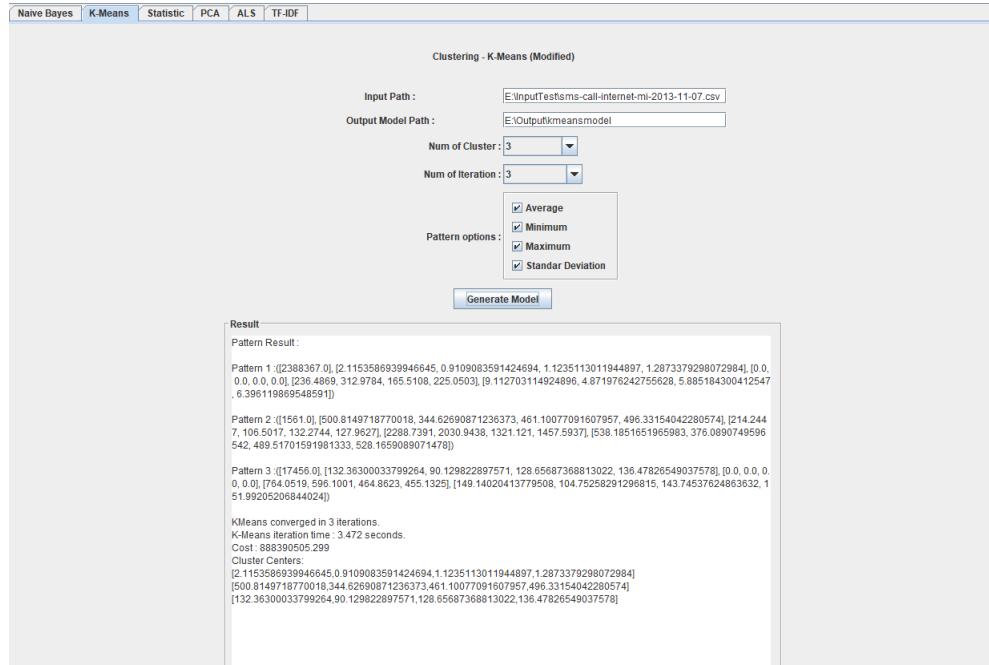


Gambar 5.20: Hasil pengujian fungsional naive bayes model



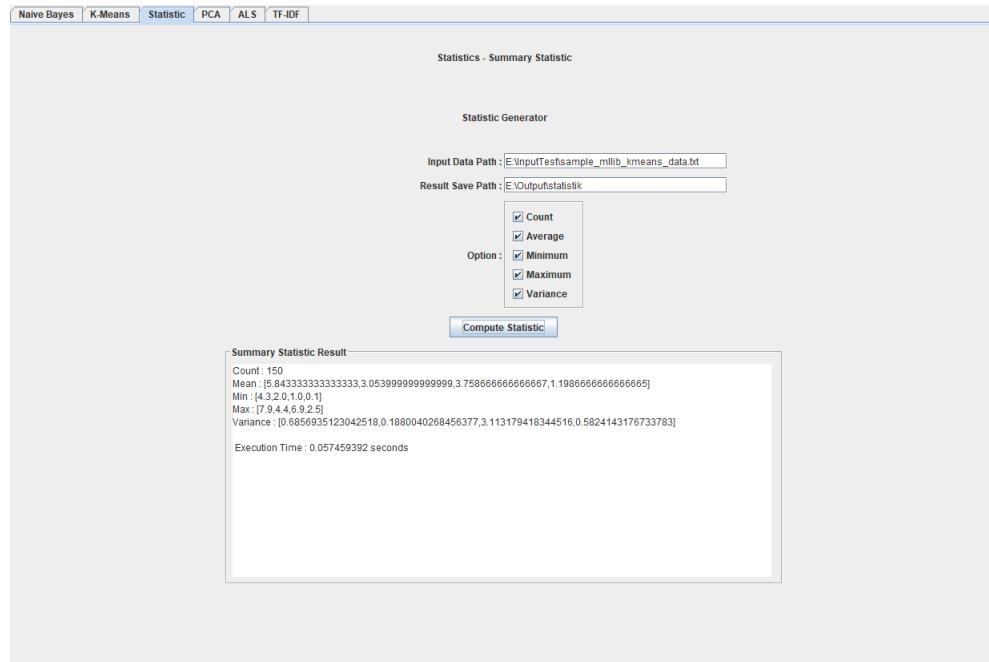
Gambar 5.21: Hasil pengujian fungsional prediksi dengan model naive bayes

- 4      2. Hasil pengujian fungsional K-Means : Gambar 5.22 menunjukan keluaran untuk fungsi  
 1      K-Means.



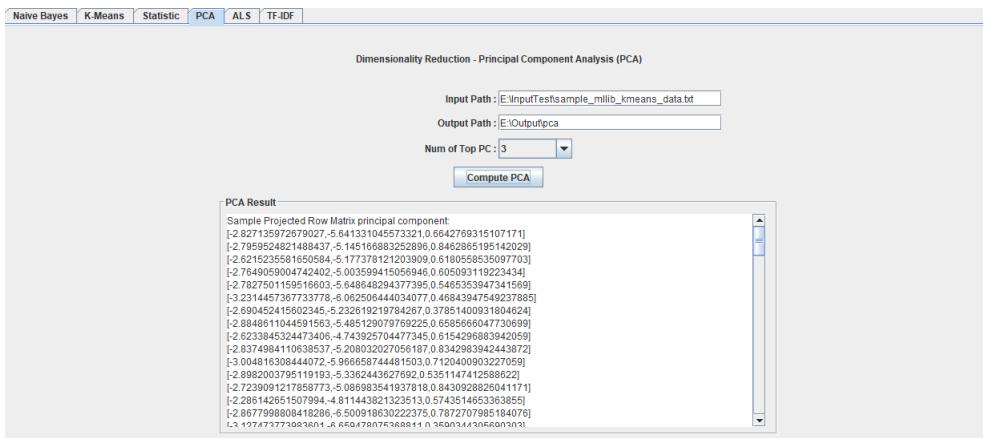
Gambar 5.22: Hasil pengujian fungsional K-Means

- 2      3. Hasil pengujian fungsional statistik : Gambar 5.23 menunjukan keluaran untuk fungsi statistik.



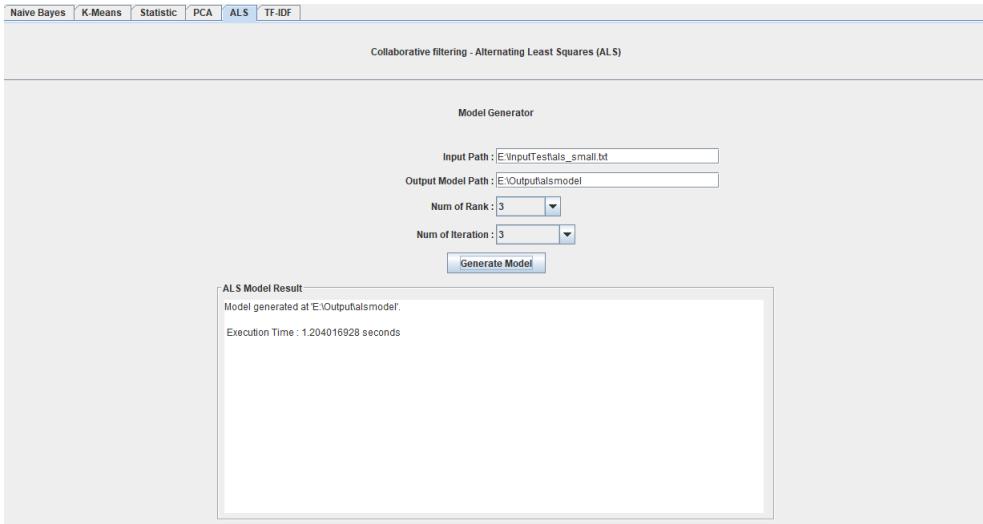
Gambar 5.23: Hasil pengujian fungsional statistik

- 3      4. Hasil pengujian fungsional PCA : Gambar 5.24 menunjukan keluaran untuk fungsi PCA.

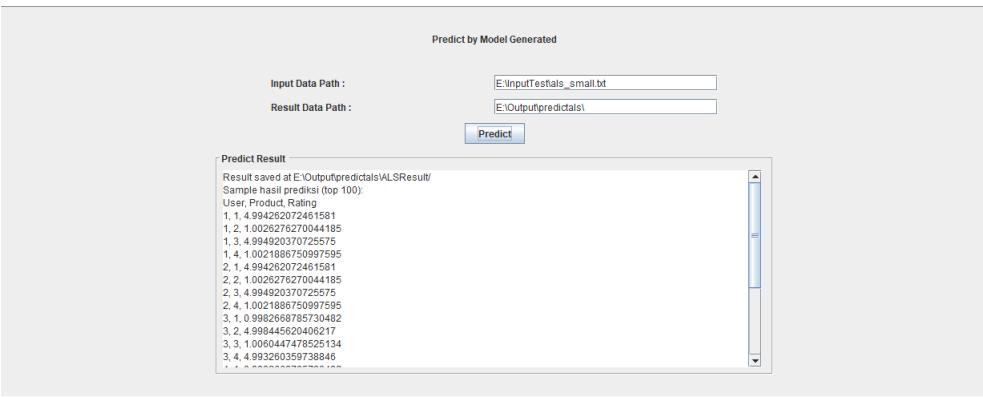


Gambar 5.24: Hasil pengujian fungsional PCA

- 1 5. Hasil pengujian fungsional ALS: Gambar 5.25 dan gambar 5.26 menunjukkan keluaran untuk fungsi ALS.
- 2

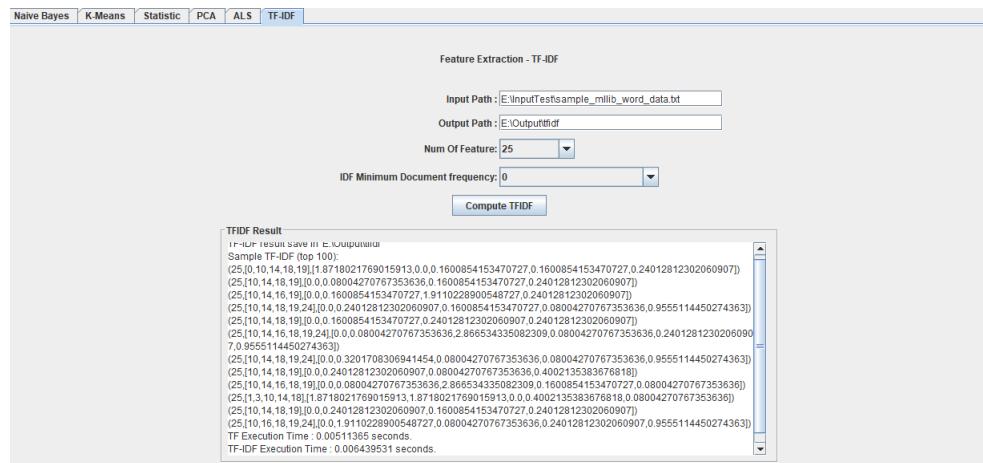


Gambar 5.25: Hasil pengujian fungsional model ALS



Gambar 5.26: Hasil pengujian fungsional prediksi dengan model ALS

- 3 6. Hasil pengujian fungsional TF-IDF: Gambar 5.27 menunjukkan keluaran untuk fungsi TF-IDF.



Gambar 5.27: Hasil pengujian fungsional TF-IDF

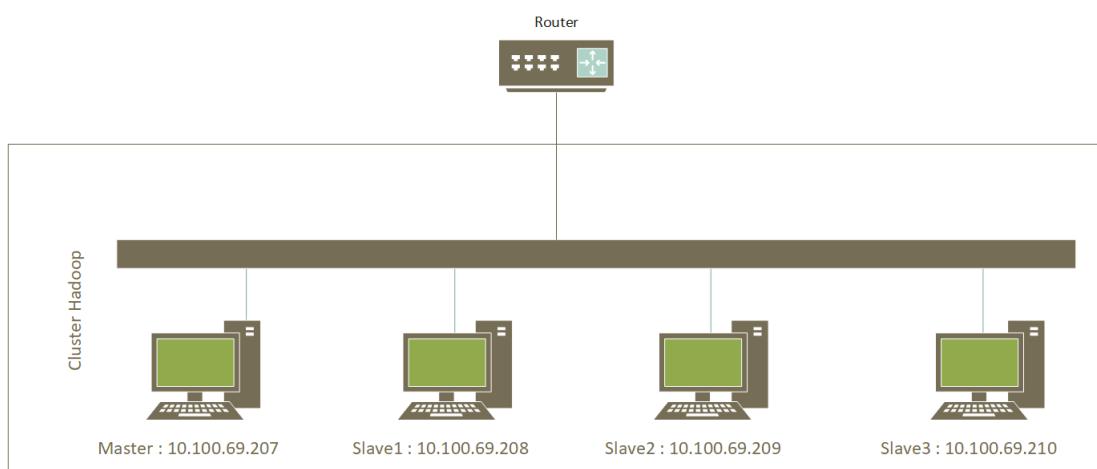
Pengujian fungsional perangkat lunak dilakukan dengan menguji setiap fitur yang ada. Pengujian tersebut menghasilkan hasil yang sesuai harapan. Sehingga dapat disimpulkan bahwa setiap fitur pada telah berjalan dengan baik dan benar.

## 5.6 Eksperimen untuk Uji Performa Fungsi-fungsi MLib

### 5.6.1 Lingkungan Perangkat Keras

Perangkat lunak ini dirancang untuk berjalan diatas sebuah cluster hadoop. Sehingga eksperimen dilakukan dengan menggunakan cluster hadoop yang terdiri dari 4 komputer. Perangkat keras yang digunakan dalam eksperimen perangkat lunak diantaranya meliputi 4 buah komputer yang masing-masing memiliki spesifikasi yang sama. Berikut spesifikasinya :

- Processor : Intel Core i3 550 3.2Ghz
- RAM : 8GB DDR3
- Harddisk : 270 GB



Gambar 5.28: Arsitektur lingkungan cluster hadoop untuk eksperimen

4 Gambar 5.28 memperlihatkan kondisi lingkungan dan konfigurasi dari cluster hadoop. Cluster  
5 memiliki 4 buah komputer terdiri dari satu komputer sebagai master dan tiga komputer lainnya  
6 sebagai slave. Semua komputer terletak dalam ruangan yang sama dan dalam satu jaringan lokal.

7 **5.6.2 Lingkungan Perangkat Lunak**

8 Penulis melakukan eksperimen perangkat lunak demo pada lingkungan cluster hadoop. Agar  
9 perangkat lunak ini dapat berjalan tanpa konflik, maka dibutuhkan perangkat lunak lain yang  
10 mendukung. Berikut merupakan lingkungan perangkat lunak yang digunakan pada eksperimen :

- 11 • Sistem Operasi Ubuntu 14.04 LTS 64bit  
12 • Java 1.8.0\_161  
13 • Apache Spark 2.2.0  
14 • Hadoop 2.7.1

15 **5.6.3 Hasil Eksperimen**

16 Eksperimen performa dilakukan dengan membandingkan waktu setiap ukuran data masukkan  
17 yang berbeda. Penulis melakukan eksperimen yang akan mencatat waktu eksekusi pemanggilan.  
18 eksperimen perangkat lunak dilakukan dengan menggunakan data yang berukuran besar. Ukuran  
19 data divariasi menjadi 3 yaitu ukuran data 1 GB, 3 GB, dan 5 GB. eksperimen dimaksudkan untuk  
20 mencatat waktu eksekusi. Waktu eksekusi yang dicatat hanya pada pemanggilan fungsi MLlib.  
21 Berikut adalah hasil eksperimen uji performa masing-masing fitur perangkat lunak.

22

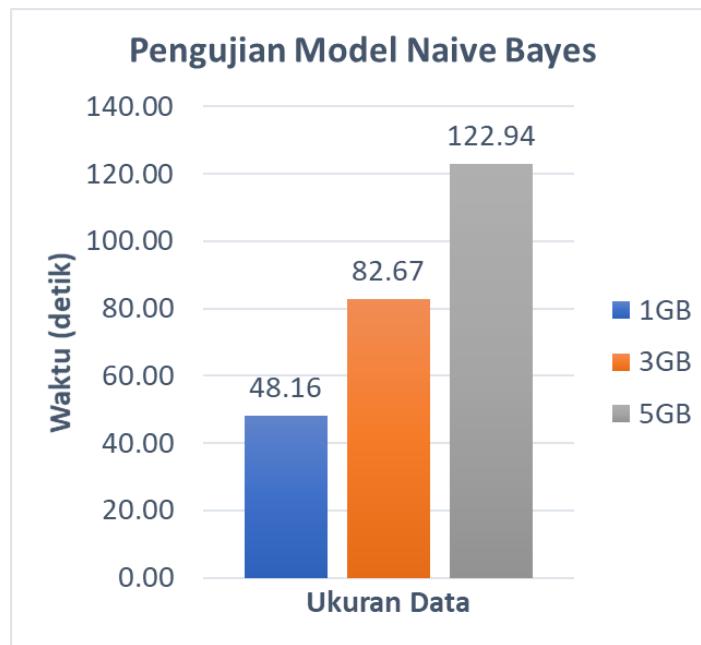
23 **Eksperimen Naive Bayes**

24 Eksperimen pada pengolahan data dengan Naive Bayes dilakukan dua tahap yaitu pembuatan  
25 model dan prediksi menggunakan model. Pada pembuatan model, data masukkan disimpan pada  
26 filesystem hdfs.

27 File data masukkan memiliki tipe file .csv dan memiliki header. Kelas dari masing-masing objek  
28 *Iris-setosa*, *Iris-versicolor*, dan *Iris-virginica*. Karena Naive Bayes MLlib hanya dapat melakukan  
29 pelatihan dengan data dengan kelas berupa data numerik, maka akan dilakukan pra-pengolahan data  
30 terlebih dahulu. Pra-pengolahan adalah dengan melakukan konversi kelas yang berupa data string  
31 menjadi numerik. Kelas Iris-setosa menjadi nilai 1.0, Iris-versicolor menjadi 2.0, dan Iris-virginica  
32 menjadi 3.0.

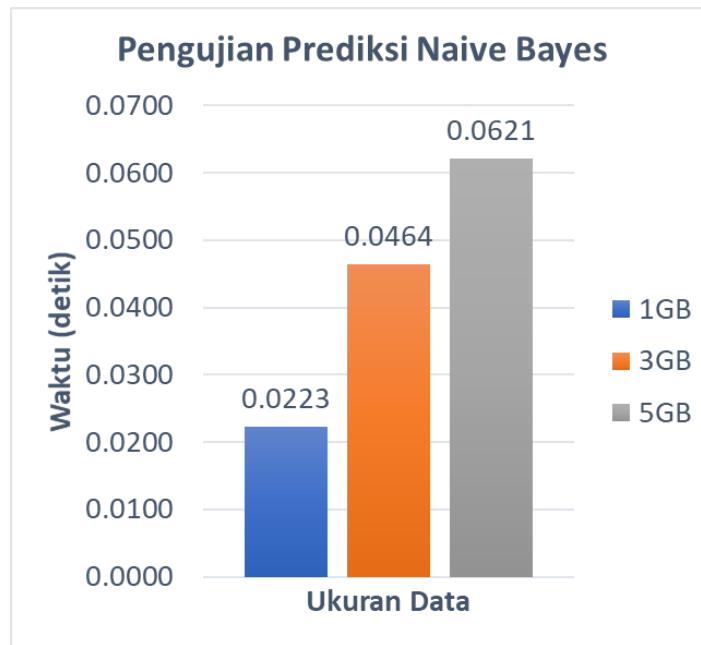
33

1 Eksperimen dilakukan dengan mengeksekusi pengolahan data dengan variasi ukuran pada data.  
2 Kemudian mencatat waktu pelatihan data pada algoritma Naive Bayes. Catatan waktu hasil  
3 eksperimen divisualisasikan dalam grafik pada gambar 5.29



Gambar 5.29: Grafik perhitungan waktu eksekusi pembuatan model Naive Bayes

4 Tahap selanjutnya adalah melakukan eksperimen prediksi pada suatu set data baru yang tidak  
 5 diketahui kelasnya. Waktu yang dicatat adalah ketika memanggil method `predict` pada data  
 6 dengan model yang telah dihasilkan pada tahap sebelumnya. Hasil waktu eksekusi dapat dilihat  
 7 pada gambar 5.30.



Gambar 5.30: Grafik perhitungan waktu eksekusi prediksi data dengan model Naive Bayes

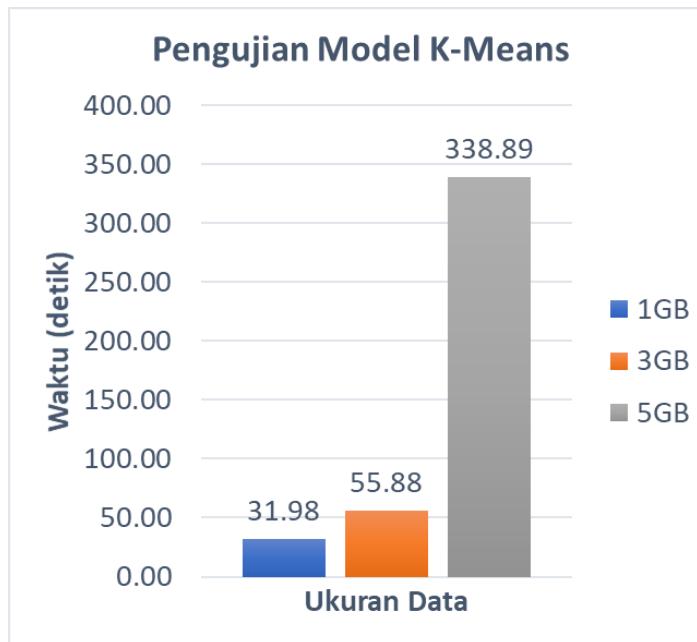
8 Berdasarkan hasil eksperimen diatas, fungsi pembuatan model dengan algoritma naive bayes  
 1 MLlib memiliki waktu yang cenderung meningkat seiring dengan ukuran data yang semakin besar  
 2 juga. Namun kelipatan waktu tidak melebihi kelipatan data. Dengan demikian, waktu eksekusi  
 3 naive bayes MLlib pada ukuran data 3 GB hanya 1,7 kali waktu eksekusi dari ukuran data 1 GB.

- 4 Sedangkan waktu eksekusi ukuran data 5 GB adalah 2,5 kali dari ukuran data 1 GB. Sehingga  
 5 dapat dikatakan bahwa performa naive bayes sangat baik untuk ukuran data yang besar.

## 6 Eksperimen K-Means

- 7 Eksperimen eksekusi fungsi K-Means dilakukan dengan pembuatan model yang sekaligus akan  
 8 menghitung pola. Pembuatan model data input akan disimpan pada cluster.

9 Data yang digunakan untuk pengolahan data dengan K-Means adalah data aktifitas telepon  
 10 genggam di suatu negara. Pada bagian **KMeansController** dijelaskan penanganan prapengolahan  
 11 data tersebut. Berikut waktu eksekusi yang dicatat dalam eksperimen:

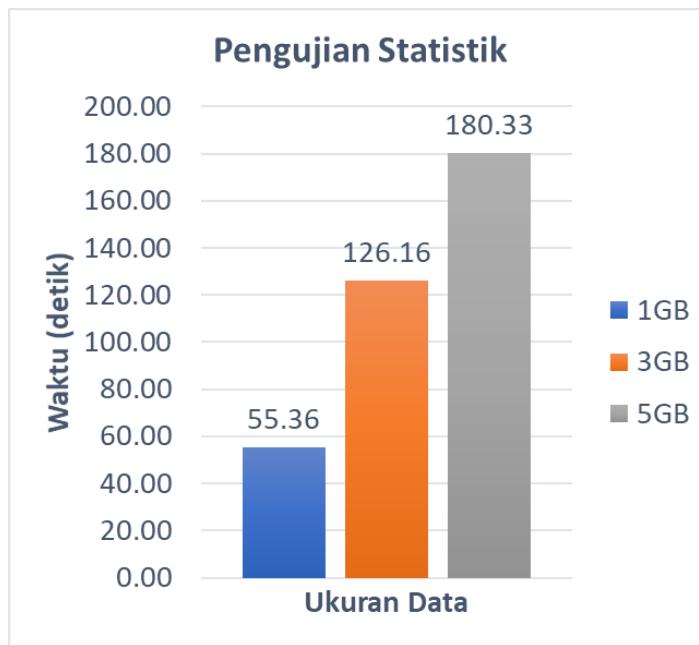


Gambar 5.31: Grafik perhitungan waktu eksekusi eksperimen K-Means

12 Berdasarkan hasil eksperimen diatas, fungsi clustering dengan algoritma K-Means MLlib memiliki  
 13 waktu yang cenderung meningkat seiring dengan data yang semakin besar juga. Pada eksperimen  
 14 dengan menggunakan data 3 GB, waktu eksekusi hanya 1,8 kali dari waktu eksekusi eksperimen  
 15 dengan data 1 GB. Meskipun demikian, pada hasil eksperimen dengan data berukuran 5 GB  
 16 menunjukkan waktu yang meningkat tajam. Hal ini disebabkan oleh *executor* yang mengalami  
 17 kegagalan. Spark akan melakukan kembali eksekusi yang telah gagal. Kegagalan sering kali  
 18 disebabkan oleh memori *executor* yang tidak dapat mencukupi. Ukuran memori pada lingkungan  
 19 perangkat keras tidak mencukupi kebutuhan eksekusi.

## 20 Eksperimen Statistik

- 21 Eksperimen pada fungsi statistik menggunakan data numerik yang memiliki 3 fitur. Pengujian  
 1 dilakukan dengan mengeksekusi fungsi statistik pada input data yang ukurannya berbeda-beda.  
 2 Dapat dilihat pada gambar 5.32 merupakan hasil eksekusi statistik pada masing-masing ukuran  
 3 data.



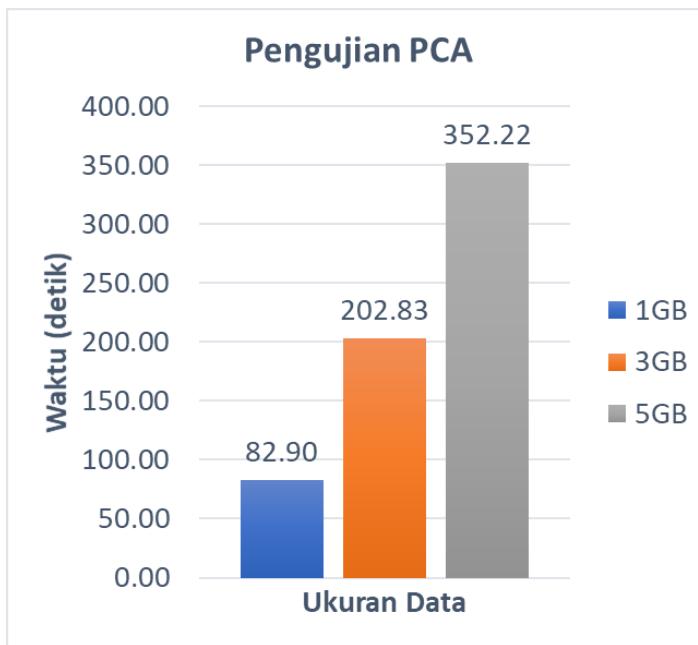
Gambar 5.32: Grafik perhitungan waktu eksekusi eksperimen Statistik

Dapat dilihat bahwa dari ketiga percobaan tersebut, semakin besar ukuran data, maka semakin besar pula waktu eksekusi. Waktu eksekusi fungsi statistik MLlib memiliki pengaruh terhadap ukuran data yang masukkan. Eksperimen pada ukuran data 3 GB membutuhkan waktu eksekusi 2,2 kali dibandingkan waktu eksekusi dengan data 1 GB. Sedangkan untuk eksperimen pada ukuran data 5 GB, waktu eksekusi hanya 3,2 kali dari ukuran data 1 GB. Dengan demikian, fungsi statistik MLlib berjalan dengan baik pada ukuran data besar.

#### Eksperimen PCA

Eksperimen pada PCA menggunakan data yang memiliki empat dimensi. Sesuai dengan ketentuan eksperimen, ukuran data divariasi menjadi 1 GB, 3 GB, dan 5 GB. Berikut adalah hasil eksperimen berupa waktu eksekusi perhitungan *principal component* pada fungsi PCA MLlib:

3

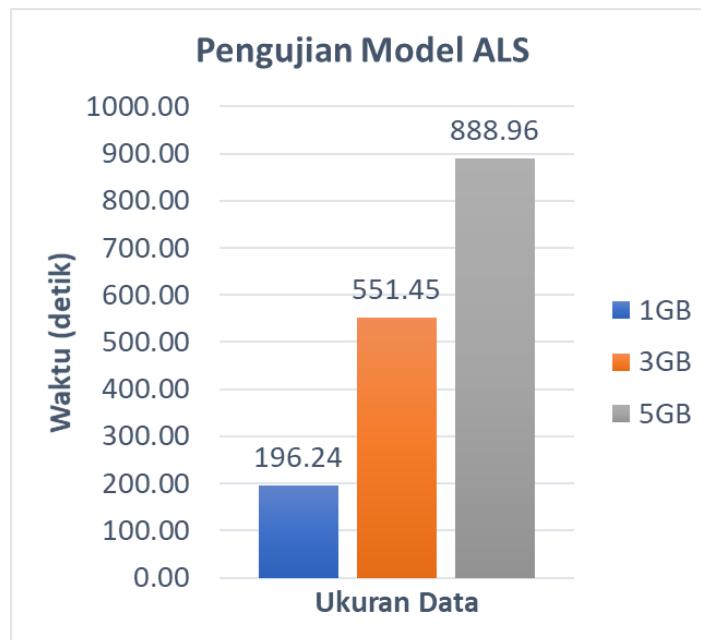


Gambar 5.33: Grafik perhitungan waktu eksekusi eksperimen PCA

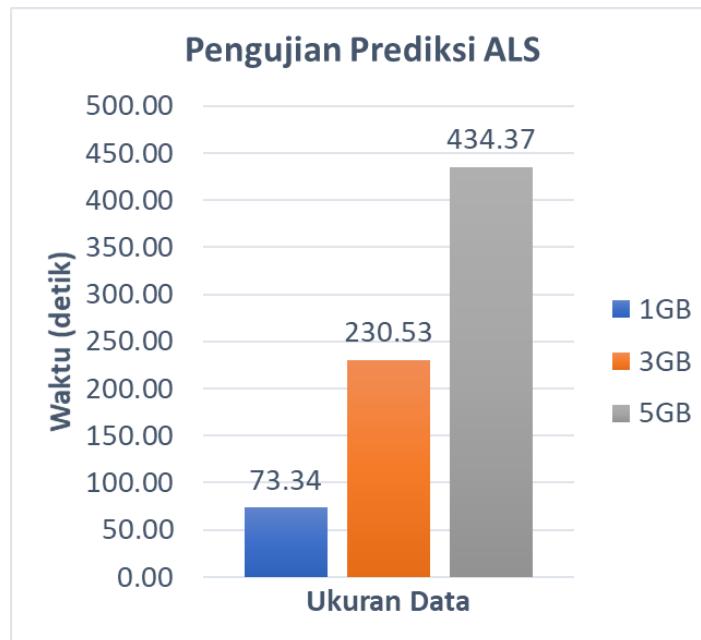
Dapat dilihat bahwa dari ketiga percobaan pada gambar 5.33 bahwa semakin besar ukuran data, maka semakin besar pula waktu eksekusi. Hal ini memiliki arti bahwa ukuran data masukkan berpengaruh terhadap waktu eksekusi perhitungan PC. Eksperimen pada ukuran data 3 GB membutuhkan waktu eksekusi 2,4 kali dibandingkan waktu eksekusi dengan data 1 GB. Sedangkan untuk eksperimen pada ukuran data 5 GB, waktu eksekusi hanya 4,2 kali dari ukuran data 1 GB. Dengan demikian, fungsi PCA MLlib berjalan cukup baik pada ukuran data besar.

#### 10 Eksperimen ALS

Eksperimen pada fungsi pengolahan ALS menggunakan sebuah file yang setiap barisnya merepresentasikan rate setiap pengguna terhadap suatu produk. Fungsi ALS menggunakan input data yang ukurannya berbeda-beda yaitu 1 GB, 3 GB, dan 5 GB. Pengolahan data dibagi menjadi dua tahap yaitu pembuatan model ALS dan prediksi data baru yang belum memiliki rate pengguna terhadap produk. Dapat dilihat pada gambar 5.34 merupakan hasil eksekusi pembuatan model (Gambar 5.34) dan waktu prediksi data baru (Gambar 5.35) pada masing-masing ukuran data:



Gambar 5.34: Grafik perhitungan waktu eksekusi pembuatan model Alternating Square Least

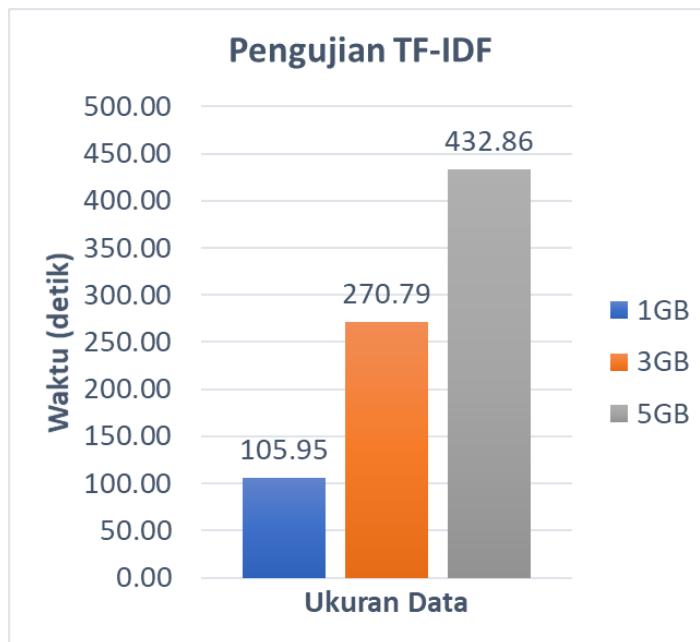


Gambar 5.35: Grafik perhitungan waktu eksekusi prediksi data dengan model Alternating Square Least

- 4 Dapat dilihat bahwa dari ketiga percobaan bahwa semakin besar ukuran data, maka semakin  
 5 besar pula waktu eksekusi. Dengan kata lain, ukuran data masukkan berpengaruh pada waktu  
 6 eksekusi fungsi ALS. Eksperimen pembuatan model ALS pada ukuran data 3 GB membutuhkan  
 1 waktu eksekusi 2,8 kali dibandingkan waktu eksekusi dengan data 1 GB. Sedangkan untuk eksperimen  
 2 pada ukuran data 5 GB, waktu eksekusi hanya 4,5 kali dari ukuran data 1 GB. Dengan demikian,  
 3 fungsi ALS MLlib berjalan cukup baik pada ukuran data besar.

#### 4 Eksperimen TF-IDF

5 Eksperimen pada fungsi perhitungan TF-IDF menggunakan sebuah file yang setiap barisnya merepre-  
6 sentasikan masing-masing dokumen. Fungsi TF-IDF menggunakan input data yang ukurannya  
7 berbeda-beda yaitu 1 GB, 3 GB, dan 5 GB. Dapat dilihat pada gambar 5.36 merupakan hasil eksekusi  
8 statistik pada masing-masing ukuran data.



Gambar 5.36: Grafik perhitungan waktu eksekusi eksperimen TF-IDF

9 Dapat dilihat bahwa dari ketiga percobaan bahwa semakin besar ukuran data, maka semakin  
10 besar pula waktu eksekusi. Dengan kata lain, ukuran data masukkan berpengaruh pada waktu  
11 eksekusi fungsi TF-IDF. Eksperimen fungsi TF-IDF pada ukuran data 3 GB membutuhkan waktu  
1 eksekusi 2,5 kali dibandingkan waktu eksekusi dengan data 1 GB. Sedangkan untuk eksperimen  
2 pada ukuran data 5 GB, waktu eksekusi hanya 4,1 kali dari ukuran data 1 GB. Dengan demikian,  
3 fungsi TF-IDF MLlib berjalan cukup baik pada ukuran data besar.

## BAB 6

### KESIMPULAN DAN SARAN

#### 6.1 Kesimpulan

7 Setelah melakukan proses studi dan eksplorasi Apache Spark MLlib, pengembangan K-Means,  
 8 perancangan, implementasi, dan eksperimen perangkat lunak demo pada karya ilmiah ini, penulis  
 9 mengambil beberapa kesimpulan, diantaranya adalah:

- 10 • Dengan melakukan studi literatur Apache Spark dan Hadoop terutama HDFS, maka dapat  
 11 dipahami bahwa Apache Spark tidak menggantikan hadoop sebagai framework sistem ter-  
 12 distribusi. Apache Spark dapat berjalan sendirian dan juga dapat berjalan pada lingkungan  
 13 hadoop. Hadoop menyediakan filesystem dan resource bagi Apache Spark. Apache Spark  
 14 mampu menjalankan berbagai komputasi pararel di memori. Namun tetap mempertahankan  
 15 model Map and Reduce yang sudah ada pada Hadoop.
- 16 • Apache spark memiliki *library* MLlib yang menyimpan berbagai macam implementasi algoritma  
 17 pengolahan data. Dengan demikian, MLlib mampu melakukan analisis Big Data dengan  
 18 menyediakan kelas-kelas yang siap dipanggil untuk teknik pengolahan data tertentu.
- 19 • Salah satu fungsi MLlib yaitu K-Means memiliki kebutuhan yang belum terpenuhi, maka dilakuk-  
 20 kan modifikasi pada source code MLlib. Hal ini dilakukan dengan menganalisis source code  
 21 Apache Spark MLlib, merancang modifikasi yang akan dilakukan, dan mengimplementasikan  
 22 pada perangkat lunak demo.
- 23 • Pengembangan perangkat lunak untuk mempraolah data diimplementasikan dengan meng-  
 24 gunakan API Spark meliputi Spark Core dan Spark MLlib dan berjalan pada platform Java  
 25 dengan bahasa pemrograman Scala. Fungsi-fungsi pada MLlib dipanggil untuk menganalisis  
 26 data dan menampilkan hasil analisis data pada *graphical user interface* (GUI) yang disediakan  
 27 perpustakaan Swing Scala.

#### 6.2 Saran

1 Setelah melakukan studi dan eksplorasi Apache Spark MLlib, pengembangan K-Means, perancangan,  
 2 implementasi, dan eksperimen perangkat lunak demo pada karya ilmiah ini, penulis mengambil  
 3 beberapa saran, diantaranya adalah:

- 4     ● Berdasarkan hasil eksperimen yang telah dilakukan, penulis menyarankan perlunya dilakukan  
5       pengujian dengan variasi jumlah node cluster yang besar. Sehubungan dengan keterbatasan  
2415      fasilitas, maka eksperimen pada karya tulis ini tidak dilakukan.
- 2416     ● Penulis menyarankan agar Apache Spark MLlib ini dijalankan pada cluster hadoop agar dapat  
2417       memanfaatkan kelebihan dari cluster computing.

## DAFTAR REFERENSI

- [1] Odersky, M., Spoon, L., dan Venners, B. (2016) *Programming in Scala: Updated for Scala 2.12*, 3rd edition. Artima Incorporation, USA.
- [2] Tanenbaum, A. S. dan Steen, M. v. (2006) *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- [3] Lam, C. (2010) *Hadoop in Action*, 1st edition. Manning Publications Co., Greenwich, CT, USA.
- [4] Holmes, A. (2012) *Hadoop in Practice*. Manning Publications Co., Greenwich, CT, USA.
- [5] Dean, J. dan Ghemawat, S. (2004) Mapreduce: Simplified data processing on large clusters. *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, Berkeley, CA, USA OSDI'04, pp. 10–10. USENIX Association.
- [6] Yadav, R. (2017) *Apache Spark 2.X Cookbook: Cloud-ready Recipes for Analytics and Data Science*, 2nd edition. Packt Publishing.
- [7] Abbasi, M. A. (2017) *Learning Apache Spark 2.0*. Packt Publishing.
- [8] Harrington, P. (2012) *Machine Learning in Action*. Manning Publications Co., Greenwich, CT, USA.
- [9] Karau, H., Konwinski, A., Wendell, P., dan Zaharia, M. (2015) *Learning Spark: Lightning-Fast Big Data Analytics*, 1st edition. O'Reilly Media, Inc.



**LAMPIRAN A**  
**KODE PROGRAM**



**LAMPIRAN B**  
**HASIL EKSPERIMEN**