Software Design Description for DAKA

Version 1.2

February 18, 2014

Prepared by:

Sergey Matskevich, Christopher Harvey, Ernesto Cejas Padilla,
ByeongGil Jeon and Jirakit Songprasit

Stakeholder: iPipeline

Advisors: Mr. William M. Mongan, Dr. Jeffrey Popyack

Drexel University

Revision History

Name	Date	Reason for changes	Version
Sergey Matskevich, Christopher Harvey, Ernesto Cejas Padilla, ByeongGil Jeon and Jirakit Songprasit	January 13, 2014	Initial version	1.0
Sergey Matskevich, Christopher Harvey, Ernesto Cejas Padilla, ByeongGil Jeon and Jirakit Songprasit	February 13, 2014	Included architecture and context diagrams. Description of the algorithms used.	1.1
Sergey Matskevich, Christopher Harvey, Ernesto Cejas Padilla, ByeongGil Jeon and Jirakit Songprasit	February 18, 2014	Included UML class diagrams for algorithms. Final annotations. Completed glossary	1.2

Table of Contents

1 Introduction

1.1 Purpose

This document specifies the entire software architecture for DAKA. The design decisions outlined below comply with the software constraints and functionality requirement outlined in the DAKA Software Requirements Specification document.

iPipeline leads its industry in providing the next-generation suite of sales distribution software to the insurance and financial services markets through its on-demand service. iPipeline is processing about 100 TB of data on a regular basis. The processing time it takes to produce results is too long for the business and a solution is required to improve processing time, as well as to account for future increased amounts of data to be processed.

DAKA will utilize horizontally scalable distributed system Hadoop and implement algorithms for it in order to achieve required performance. For this DAKA will be deployed on a cluster where number of nodes can grow as needed to meet new processing requirements.

1.2 Scope

This document describes the software architecture for the initial release of DAKA, version 1.0. The intended audience of this document exclusively includes the designers, the developers, and the testers of the DAKA software system.

1.3 Glossary

Attribute

An attribute is a data field representing a characteristic or feature of a data object.

Class

A classifier is constructed to predict class (categorical) labels. If we want to know if potential customers of life insurance will buy a type of life insurance then the classes would be “yes” and “no”.

Class conditional independent assumption

Two events, R and B, are conditionally independent given a third event, Y, precisely if the occurrence or non-occurrence of R and the occurrence or non-occurrence of B are independent events in their conditional probability distribution given Y.

Data classification

A two-step process, consisting of a learning step (where a classification model is constructed) and a classification step (where the model is used to predict class labels for given data).

Gaussian distribution

A very commonly occurring continuous probability distribution—a function that tells the probability that an observation in some context will fall between any two real numbers.

Hadoop

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. Hadoop scales up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high availability, the library detects and handles failures at the application layer, so delivering a highly available service on top of a cluster of computers, each of which may be prone to failures.

Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems; however, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and designed to deploy on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets.

Map

A Map function is a higher-order function that applies to each element of a list, returning a list of results.

Mean

The most common and effective numeric measure of the "center" of a set of data is the (arithmetic) mean. Let $x_1, x_2 \ldots x_N$ be a set of N values or observations, such as for some numeric attribute X, like salary. The mean of this set of values is

$$\text{Mean} = \frac{\sum_{i=1}^{N} x_i}{N} = \frac{x1+x2+\cdots+xN}{N}.$$

Reduce

A Reduce function is a higher-order function that recombines elements of a list, returning

a single value.

Standard deviation

In statistics and probability theory, the standard deviation (SD) (represented by the Greek letter sigma, σ) shows how much variation or dispersion from the average exists. A low standard deviation indicates that the data points tend to be very close to the mean (also called expected value). A high standard deviation indicates that the data points are spread out over a large range of values.

Statistical classifiers

In machine learning and statistics, classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known.

Training set

A set made up of a database tuples and their associated class labels.

Training tuples

The individual tuples making up the training set are referred to as training tuples.

Tuple

A tuple is an ordered list of elements. In set theory, an (ordered) *n*-tuple is a sequence (or ordered list) of *n* elements, where *n* is a non-negative integer.

Venn diagram

A diagram that shows all possible logical relations between a finite collection of sets. It is used to teach elementary set theory, as well as illustrate simple set relationships in probability, logic, statistics, linguistics, and computer science.

1.4 Context Diagram

The context diagram shown in Figure 1: Context Diagramshows how the main components of the DAKA system process interact. An external system produces data. DAKA's IO components load that data into the Hadoop framework. DAKA's Compute components process the data. Then DAKA's IO components export that data which an external system consumes.

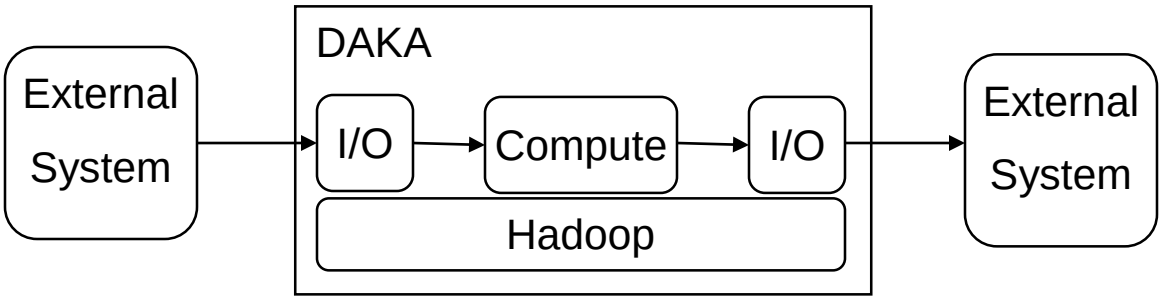

Figure 1: Context Diagram

2 Architecture

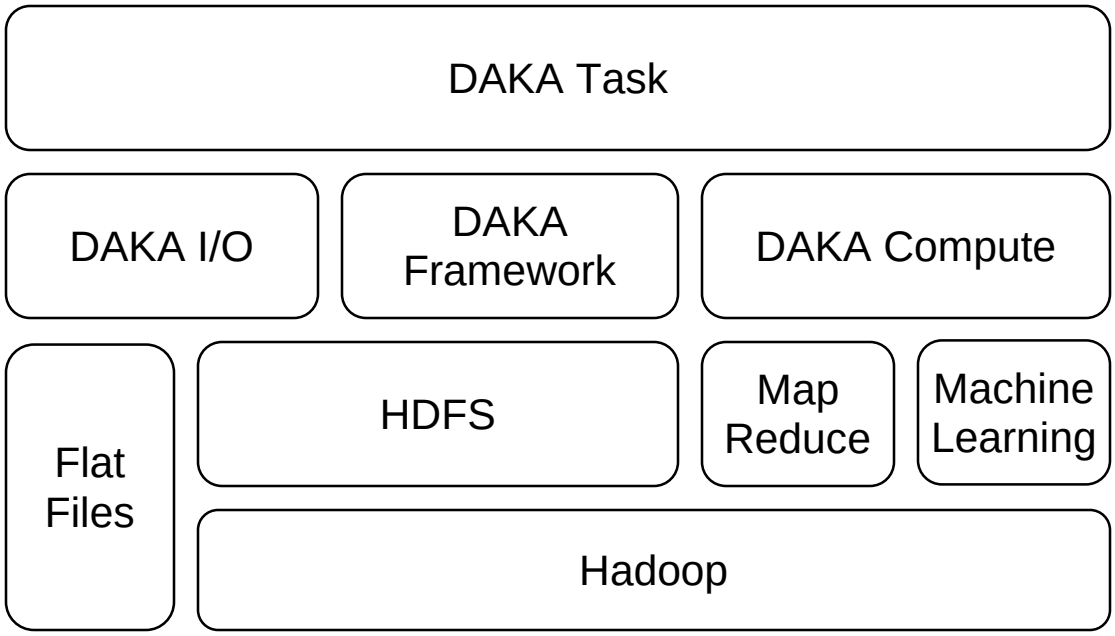

Figure 2: Architecture Diagram

2.1 Overview

DAKA's architecture is component based to maximize the separation of concerns for each module. Figure 2: Architecture Diagram shows the architecture diagram which lays out the components and how they interact.

2.2 Technologies Used

DAKA uses HDFS to store large amounts of data, distributed across several machines. DAKA uses Apache Hadoop for sorting and analyzing the data by implementing the MapReduce computation model.

2.2.1 HDFS

HDFS is a distributed and scalable file system written for the Hadoop framework. Figure 3: HDFS Architecture shows the architecture of an HDFS instance. A single instance of HDFS consists of a single namenode and multiple datanodes. The namenode tracks the metadata of the files being stored in the file system. This includes the locations of each piece of data. Each of the datanodes stores a portion of the data.

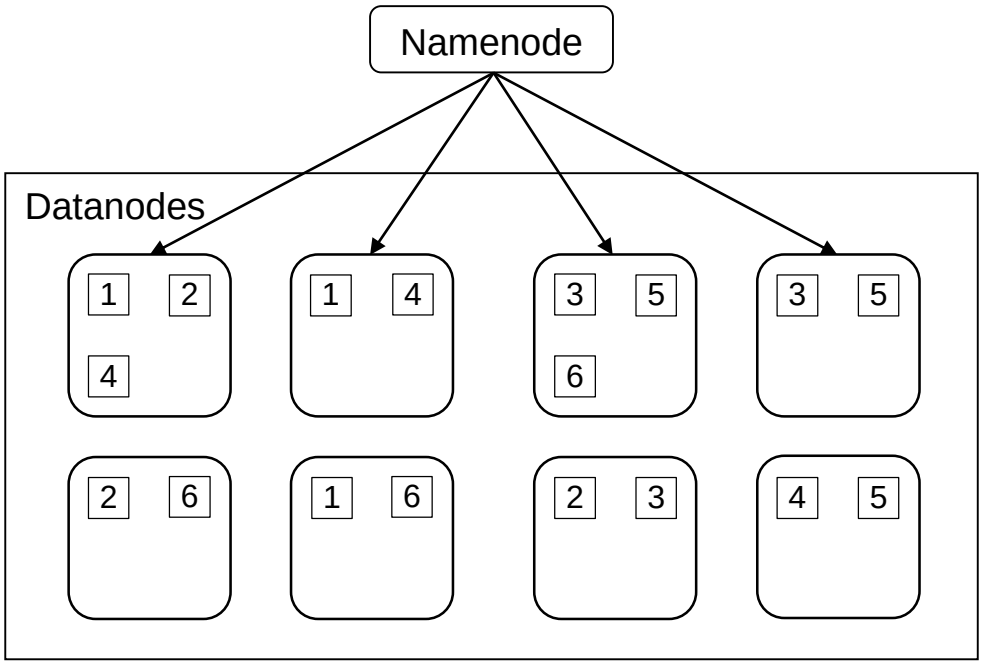

Figure 3: HDFS Architecture

HDFS provides reliability by replicating data across the datanodes. By default, there are three replications of each piece of data. HDFS also provides scalability by allowing for the addition of new datanodes. Each additional datanode adds additional storage space for the HDFS cluster.

2.2.2 Hadoop

Hadoop is a software framework that allows for processing large-scale data. It uses HDFS to store input data across a cluster of machines. Hadoop then manages the execution of programs written using the MapReduce paradigm. Finally, the results of the execution remain stored in HDFS. Figure 4: Hadoop execution flow shows the flow of Hadoop's execution.

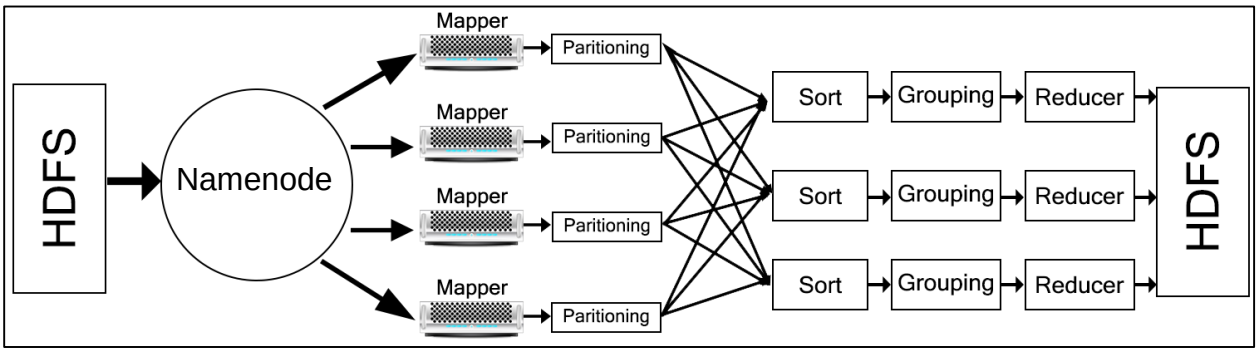

Figure 4: Hadoop execution flow

Because Hadoop uses HDFS, all data to be processed exists across multiple datanodes. Hadoop handles the execution of processing by distributing the execution across the nodes where the data exists. By moving the execution to the data, Hadoop reduces traffic between nodes by eliminating the need to transfer input data. In addition, data replication allows Hadoop to decide which copy of data to execute on to reduce load on individual datanodes.

The MapReduce paradigm allows for parallel execution of data. Hadoop manages this parallel execution by first executing the Map step on the input data where it exists in HDFS. Hadoop then transfers the intermediate results from the Map step to datanodes for the Reduce step. It then stores the result of the Reduce step in HDFS by replicating the data across nodes. By managing the execution of a program written with the MapReduce paradigm, Hadoop simplifies the work necessary to take advantage of distributed processing.

2.3 Framework Components

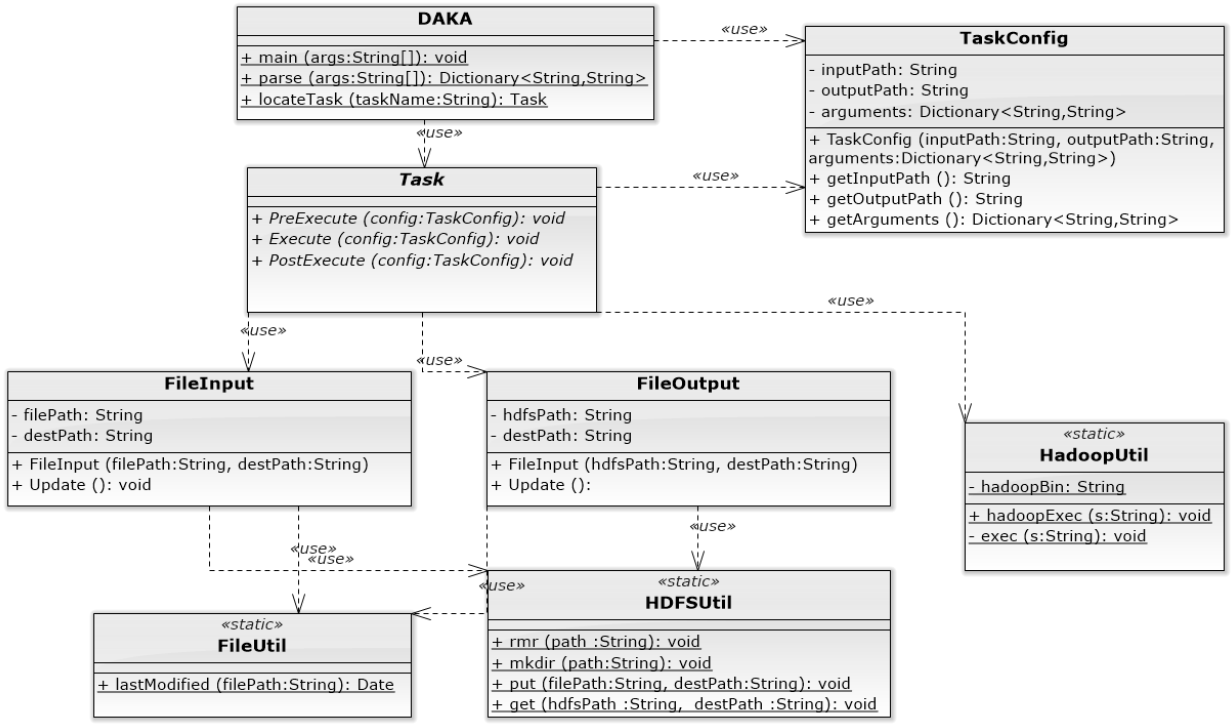

Figure 5: Overview of framework components

2.3.1 Overview

The DAKA framework consists of components that allow for the execution of DAKA software. This includes the interface that the user interacts with, the task model used to designate specific scenarios, as well as general utilities used through DAKA.

2.3.2 User Interface

2.3.2.1 DAKA

The DAKA component serves as the main entry point of the framework.

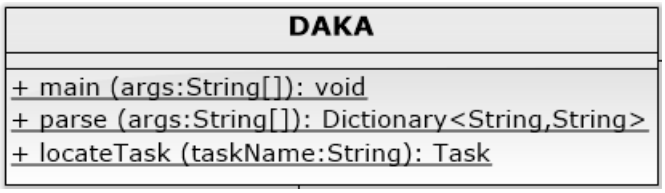

Figure 6: DAKA Class

- main – the entry point of the DAKA framework
- parse – parses the command line arguments to form a dictionary of arguments
- locateTask – locates and instantiates a task with the given name from a directory of task jars

2.3.3 Task Model

2.3.3.1 Task

The Task model is an abstraction that is implemented by user-defined tasks. The DAKA framework calls the three steps in the Task definition. Each step has an intended use but it is up to the user to define the purpose of each step.

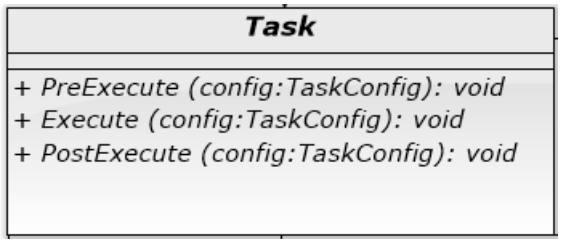

Figure 7: Task Class

- PreExecute – first method called intended for preparing input data
- Execute – second method called intended for running the bulk of the data processing
- PostExecute – third method called intended for handling output data

2.3.3.2 Task Configuration

The Task Configuration contains the collection of arguments passed to a Task at runtime.

These arguments include the location of input data and the destination of output data. The collection also includes any task specific arguments.

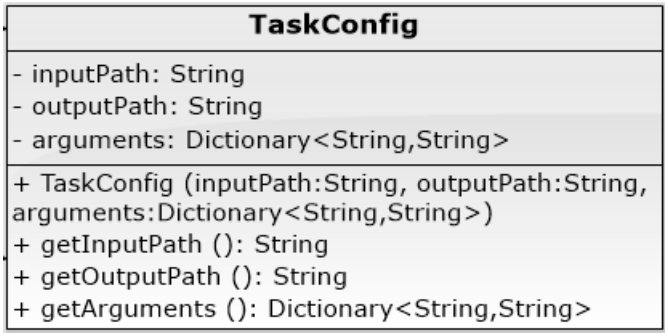

Figure 8: TaskConfig Class

- inputPath – the path entered at the command line for input data
- outputPath – the path entered at the command line for output data
- arguments – all arguments passed to DAKA at the command line
- TaskConfig – constructor which accepts values for the inputPath, outputPath, and arguments fields
- getInputPath – accessor for inputPath field
- getOutputPath – accessor for outputPath field
- getArguments = accessor for arguments field

2.3.4 General Utilities

2.3.4.1 File Utility

The File Utility handles interactions with local files.

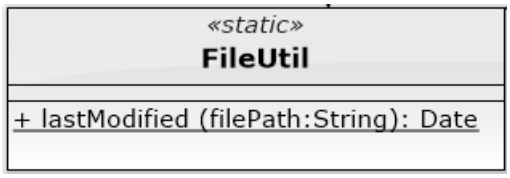

Figure 9: FileUtil Class

- lastModified – returns the last modified date of the file at the given path

2.3.4.2 Hadoop Utility

The Hadoop Utility component handles interactions with the Hadoop framework.

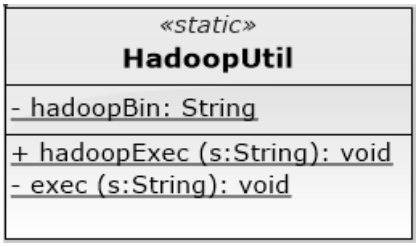

Figure 10: HadoopUtil Class

- hadoopBin – the path to the Hadoop binary
- hadoopExec – executes Hadoop commands using the Hadoop binary
- exec – executes command line commands

2.3.4.3 HDFS Utility

The HDFS Utility component handles interactions with files stored in HDFS.

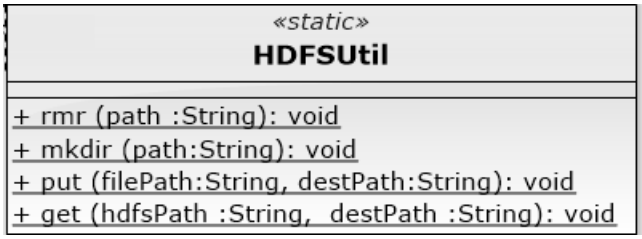

Figure 11: HDFSUtil Class

- rmr – recursively deletes a path in HDFS
- mkdir - creates a directory in HDFS
- put – transfers a file from the local filesystem at filePath into HDFS at destPath
- get – transfers a file from HDFS at hdfsPath into the local filesystem at destPath

2.4 Input and Output Components

2.4.1 Overview

DAKA's input and output components will help manage the data used as input for a DAKA task and the data resulting from the execution of the task.

2.4.2 Input

2.4.2.1 File Input

The File Input component facilitates using local files as inputs to a DAKA task. The component will transfer a local file into HDFS when necessary.

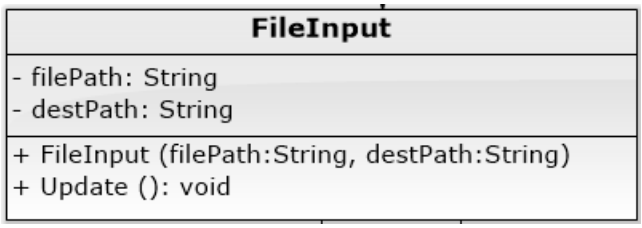

Figure 12: FileInput Class

- filePath – the path in the local filesystem where the input exists
- destPath – the path in HDFS where the input will be put
- FileInput – constructor which accepts values for the filePath and destPath fields
- Update – if local file is newer than destination file, replaces destination file with local file

2.4.3 Output

2.4.3.1 File Output

The File Output component facilitates storing the results of a DAKA task into a local file. The component will transfer the results from HDFS into a local file.

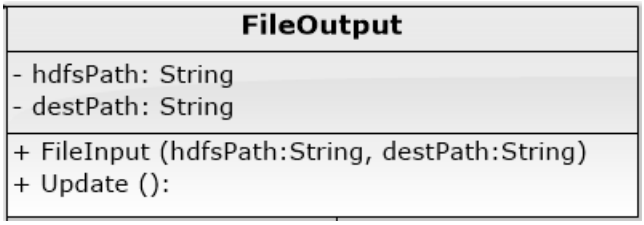

Figure 13: FileOutPut Class

- hdfsPath – the path in HDFS where the output exists
- destPath – the path in the local filesystem where the output will be put
- FileOutput – constructor which accepts values for the hdfsPath and destPath fields
- Update – if HDFS file is newer than destination file, replaces destination file with HDFS file

2.5 Compute Components

2.5.1 Overview

DAKA includes implementations of algorithms written to take advantage of the Hadoop framework. These implementations allow Hadoop to distribute execution by using the

MapReduce paradigm when possible.

2.5.2 FP-growth

2.5.2.1 FP-growth

The FP-growth object performs the FP-growth algorithm.

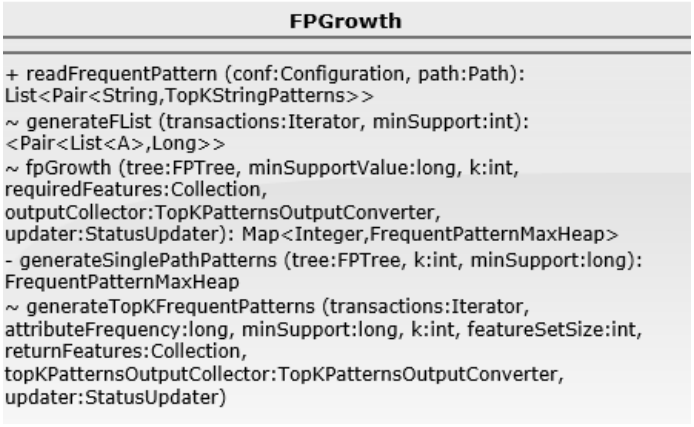

Figure 14: FPGrowth Class

- readFrequentPattern – reads the stored frequent patterns from the given location
- generateFList – generates the feature frequency list from the given transaction whose frequency is greater than minSupport
- fpGrowth – runs the FP-growth algorithm to generate the FP-tree with minimum support value
- generateSinglePathPatterns – generates a single path pattern with the given minimum support
- generateTopKFrequentPatterns – generates top *K* frequent patterns for every feature in returnable features given a stream of transactions and the minimum support

2.5.2.2 FP-tree

An FP-tree is a representation of the tree used to store the attributes analyzed by the FP-growth algorithm.

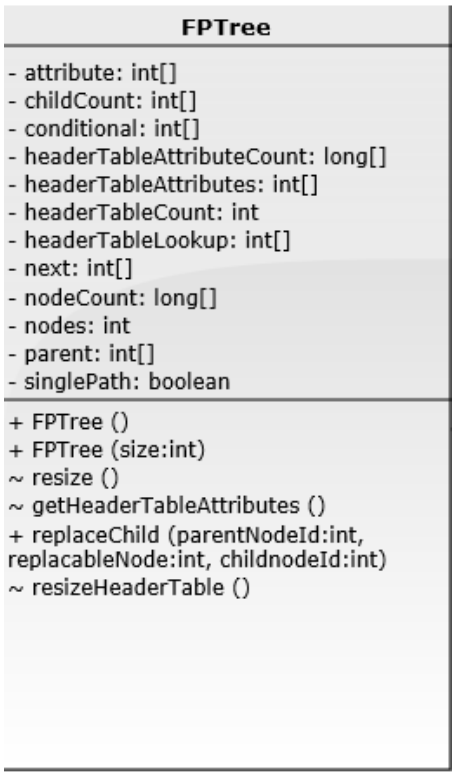

Figure 15: FPTree Class

- attribute – a collection of attribute identifiers
- childCount – the count of children of a node
- conditional – a collection of conditions
- headerTableAttributeCount – the count of header table attributes
- headerTableCount – the count of header tables
- headerTableLookup – the lookup table for headers
- next – a collections of pointers to the next nodes
- nodeCount – the collection of counts of nodes for each node
- nodes – the total count of all nodes
- parent – a collection of pointers to parent nodes
- singlePath – indicates whether the tree is a single path
- FPTree – constructs an empty FP-tree
- FPTree – constructs an FP-tree of given size
- resize – resizes the tree to include more elements

- getHeaderTableAttributes – returns attributes from the header table
- replaceChild – replaces the child of the specified node
- resizeHeaderTable – resizes table used to store headers

2.5.2.3 Pattern

A Pattern is a list of items and their support, which is the number of times the Pattern appears in the data set.

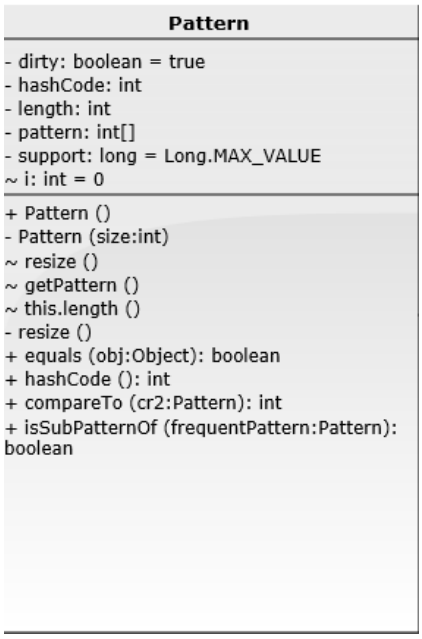

Figure 16: Pattern Class

- dirty – a Boolean that indicates the hash has not been computed
- hashCode – the hash code of the Pattern for comparisons
- length – the number of elements in the Pattern
- pattern – the elements of the Pattern
- support – the number of occurrences in the database
- Pattern – constructs an empty Pattern
- Pattern – constructs a Pattern with the given size
- resize – increases capacity if the Pattern
- equals – compares the Pattern to another object for equality
- hashCode – returns the hash code of the Pattern

- isSubPatternOf – checks if current pattern is a subpattern of another pattern
-

2.5.2.4 Frequent Pattern Max Heap

Frequent Pattern Max Heap is a priority queue that keeps top *k* attributes in a tree set.

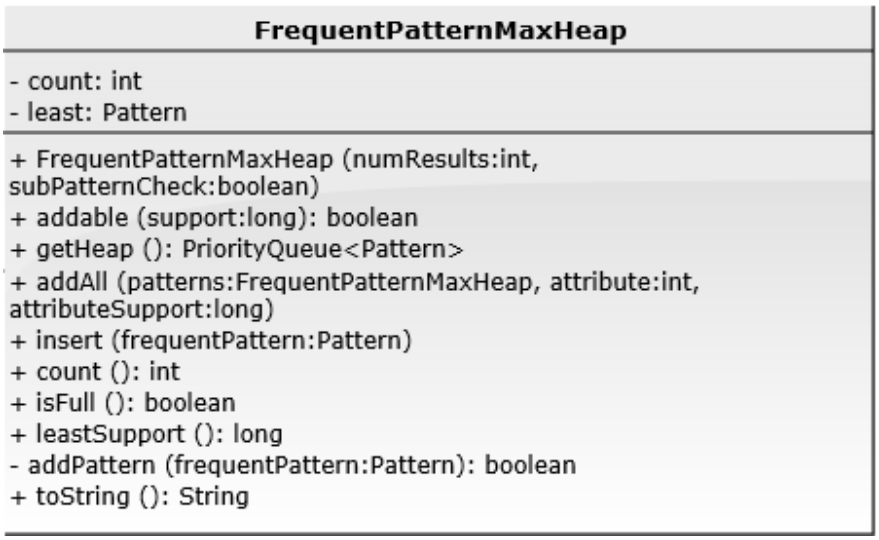

Figure 17: FrequentPatternMaxHeap Class

- count – stores the number of Patterns stored in heap
- least – stores the Pattern with minimum support
- FrequentPatterMaxHeap – constructs the heap with room for the given number of Patterns
- addable – checks if this pattern can be added to the list
- getHeap – returns the priority queue containing the Patterns
- addAll – adds all of the Patterns in a separate Frequent Pattern Max Heap
- insert – inserts a Pattern into the collection
- count – accessor for number of Paterns stored in heap
- isFull – returns a Boolean indicating if the heap is full
- leastSupport – returns minimum support parameter
- addPattern – adds a pattern to the internal Priority Queue
- toString – converts the Frequent Pattern Max Heap to a string

2.5.2.5 Top K Patterns Output Converter

Top K Patterns Output Converter is an output converter that converts the output patterns and collects them in a Frequent Pattern Max Heap.

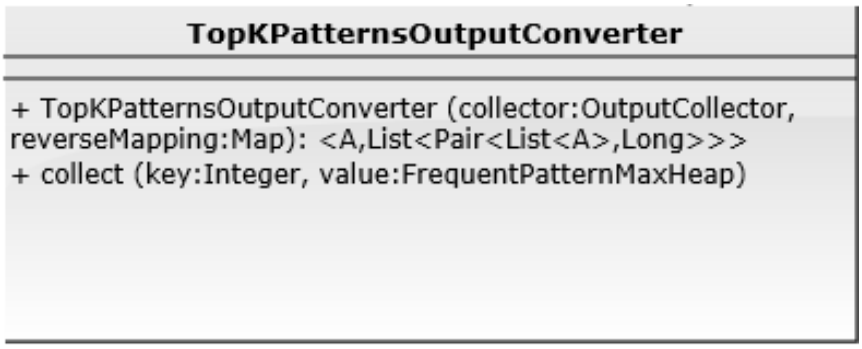

Figure 18: TopKPatternsOutputConverter Class

- TopKPatternsOutputConvert - constructs the Top K Patterns Output Converter with the given Output Collector and the Map function
- collect - gets collection of frequent patterns and converts it to appropriate output structure

2.5.2.6 FP-tree Depth Cache

The FP-tree Depth Cache caches a large FP-tree for each level of the recursive algorithm to reduce allocation overhead.

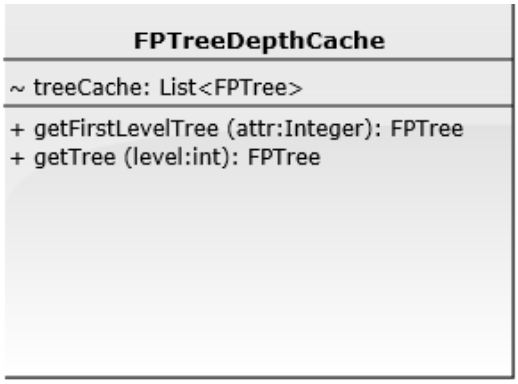

Figure 19: FPTreeDepthCache Class

- treeCache – a list of conditional FP-trees for recursive mining
- getFirstLevelTree – creates and returns conditional FP-tree for mining
- getTree - returns FP-tree of appropriate level

2.5.3 Naïve Bayesian Classification

2.5.3.1 Standard Naïve Bayes Classifier

Standard Naïve Bayes Classifier is the class implementing the Naive Bayes Classifier Algorithm.

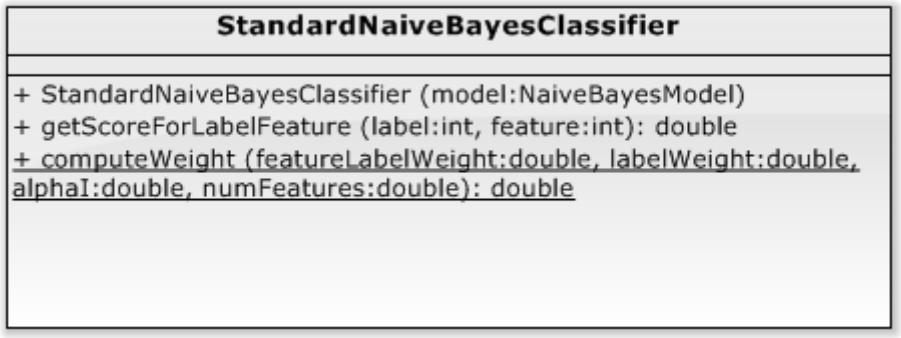

Figure 20: StandardNaiveBayesClassifier Class

- getScoreForLabelFeature – this function uses *computeweight()*method passing the arguments given to it.
- computeWeight – classifies the *labelWeight* with with a feature of *featureLabelWeight*
- Label – a property of the record
- Feature – the measurement of the property
- FeatureLabelWeight – the measurement of the weight of the property
- LabelWeight – how much weight should be given to a property
- alphaI – is the number we should add in case we need to apply Laplacian correction

2.5.3.2 Naïve Bayes Model

Naïve Bayes Model holds the weight Matrix, the feature and label sums and the weight normalizer vector.

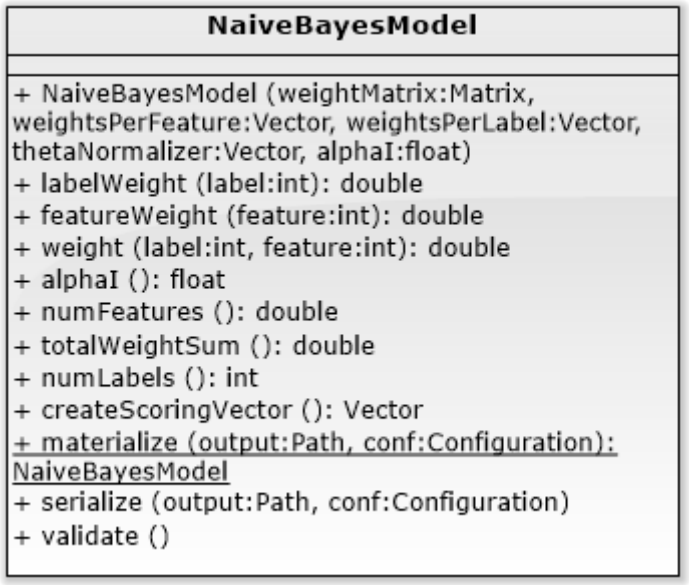

Figure 21: NaiveBayesModel Class

- weightMatrix – a matrix that inputs all the weights for the labels.
- weightsPerFeature – the weight for the occurrence of a feature.
- thetaNormalizer – used for the normalization of the labels.
- alphaI – the number we should add in case we need to apply Laplacian correction
- labelWeight – how much weight should be given to a property
- featureWeight – how much weight should be given to a value of a property
- weight – probability of the pair label-feature
- numFeatures – total number of features for this label
- totalWeightSum – total calculation of the probability with the training data
- numLabels – the amount of properties for the model
- createScoringVector – stores the intermediate probabilities for each feature
- materialize – pull the data from the path
- serialize – outputs the result of the algorithm
- validate – validates the input data from path
- conf – configuration for this algorithm

3 Algorithms

3.1 Overview

DAKA includes several algorithms as compute components. These algorithms are generic so that they can be applied to different data sets. This allows the user to solve different problems without needing to rewrite the algorithms. However, using the algorithms requires an understanding of their purpose, strengths, and weaknesses. By understanding both the MapReduce paradigm as well as the details of the included algorithms, the user can make better decisions about using them efficiently.

3.2 MapReduce Paradigm

3.2.1 Overview

The MapReduce paradigm is designed specifically for processing large data sets. A program in the MapReduce paradigm contains two steps, Map and Reduce. The Map step is applied to each record in the input. The results of the mapping are then sorted into groups. Finally, the Reduce step aggregates each group into the final output. Figure 22: The steps of MapReduce.

The strength of the MapReduce model does not come from the steps themselves but rather from the frameworks that manage the execution. Because the Mapper runs on each input and the Reducer runs on individual groups, they can each execute in parallel. This opportunity for parallelization is the reason the MapReduce paradigm is useful. When MapReduce programs are run within a MapReduce framework, the framework handles this parallelization creating time efficiency.

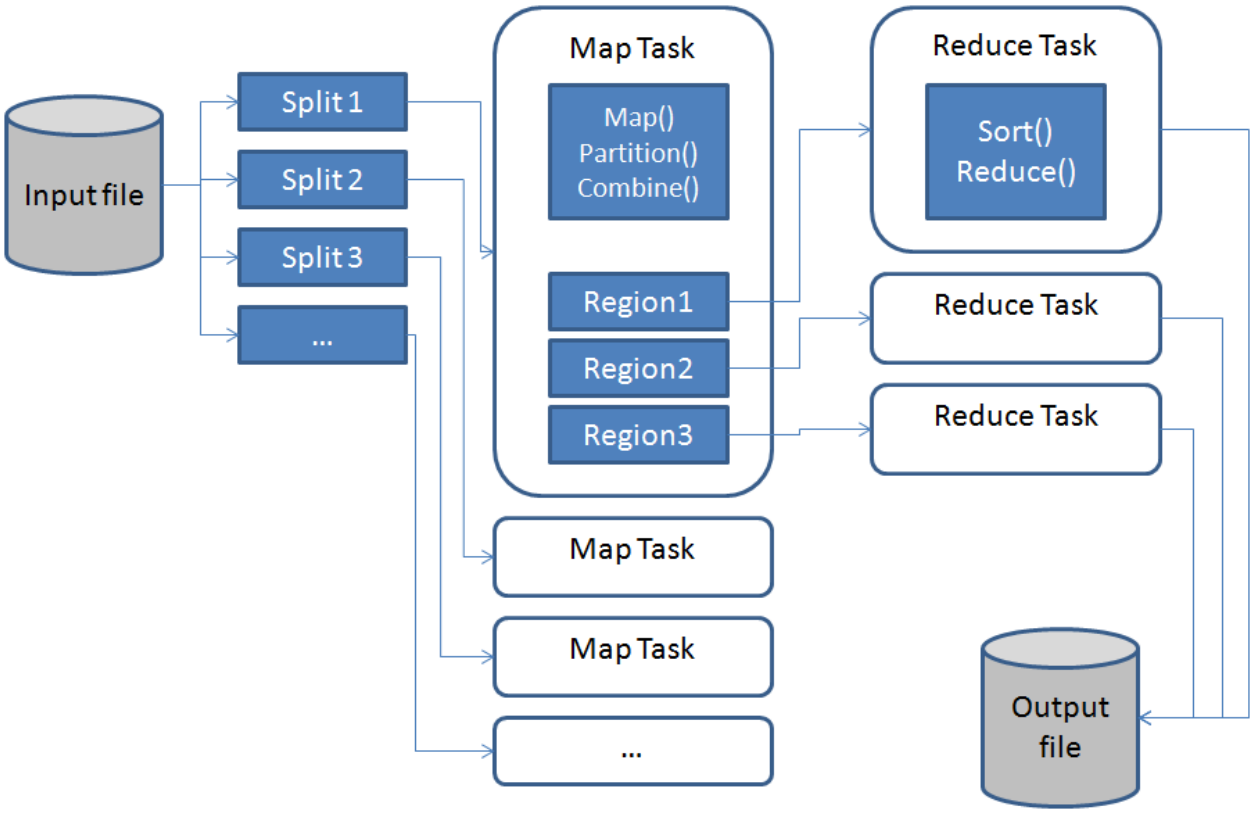

Figure 22: The steps of MapReduce

3.2.2 Map

The Map step is the initial step in a MapReduce program. The Map step consists of a function that takes a single instance of data as input. The function then generates key-value pairs that act as intermediate data. The Map function may emit any number of key-value pairs for any given input. The types of the keys and values do not matter as long as two instances of the key type are comparable.

3.2.3 Sorting

As the intermediate key-value pairs are generated, they are sorted by key. The framework that is managing the execution handles this process. Pairs with the same key are sorted together in preparation for the Reduce step. In a distributed system, the framework also manages the transfer of the values in the pairs to a machine that will handle the Reducing of a particular key.

3.2.4 Combine

Combining is an optional step in a MapReduce program. The Combine step is similar to the Reduce step in that it takes a key and a set of values as input. The difference is that the

Combine step is not guaranteed to have all values for a given key. This allows the Combine step to begin before the Map step produces all of the intermediate data. By combining the intermediate data before the Reducer is able to being, the Combiner speeds up execution and reduces the time required for the final Reducing. The use of the Combine step depends on the particular program. In some cases, it can be identical to the Reducer; in others, it is not possible to do any Combining.

3.2.5 Reduce

The Reduce step takes a set of values with a given key and produces the final outputs. The Reduce step assumes that all values with that key are present in the input and it is up to the framework to guarantee this. This step usually performs some sort of aggregation of all of the values. The Reduce function may produce multiple outputs and all of these outputs are members of the final output of the program.

3.3 FP-growth

3.3.1 Overview

The FP-growth Algorithm is an algorithm used to find frequent itemsets. It is based on a prefix tree representation of the given database of transactions (called an FP-tree), which can serve considerable amounts of memory for sorting the transactions.

The FP-growth algorithm is a recursive elimination scheme. First, in a preprocessing step, it deletes all items from the transactions that are not frequent individually. Then the algorithm selects all transactions that contain the least frequent item and deletes this item from them. It then recurses to process the reduced database, and the item sets found in the recursion share the deleted item as a prefix. When the algorithm returns, it removes the processed item from the database of all transactions and starts over. In these processing steps the prefix tree, which is enhanced by links between the branches, is exploited to quickly find the transaction containing a given item and to remove this item from the transaction after it has been processed.

3.3.2 FP-tree

A FP-tree is a compact data structure that represents the data set in tree form. The construction algorithm reads and maps each transaction onto a path in the FP-tree. This process repeats until all transactions have been read. Different transactions that have common subsets allow the tree to remain compact because their paths overlap.

The construction of a FP-tree occurs in two main steps

1. Determine the frequent items by scanning the data set and discarding the infrequent ones.
2. Scan the data and select one transaction at a time to create the FP-tree.
    a. If it is a unique transaction form a new path and set the counter for each node to one.
    b. If it shares a common prefix itemset, then increment the common itemset node counters and create new nodes if needed.
3. Continue this until each transaction has been mapped onto the tree.

3.4 Naïve Bayesian Classification

3.4.1 Overview

Bayesian classifiers are statistical classifiers. They can predict the probability that a given tuple belongs to a particular class. This classification is based on Bayes' theorem. Bayesian classifiers have exhibited high accuracy and speed when applied to large databases.

Naïve Bayesian Classification is a type of Bayesian classification that works under the *class-conditional independent* assumption. It assumes that the effect of an attribute value on a given class is independent of the values of the other attributes. This assumption simplifies the computations involved in the algorithm. It is also the reason that this algorithm is considered "naïve".

3.4.2 Bayes' Theorem

Bayes' Theorem is best understood in mathematical terms. Let ***A*** be a data tuple. ***A*** is considered to be "evidence". ***A*** is many measurements made to a set of *n* attributes. Let H be a hypothesis that says: ***A*** belongs to a specified class C. We want to determine P(H|***A***), the probability that the hypothesis H is true given the observed data tuple ***A***. Another way to say it is that we want to know the probability that tuple A belongs to class C, given that we know the attribute description of ***A***.

P(H|**A**) is the *posterior probability* of H conditioned on **A**. For example, data tuples processed by DAKA may be constrained by the attributes *age* and *income*. Let's say **A** is a 30 years old customer with an income of $50,000 and the hypothesis H is that the customer will buy a whole life policy. Then P(H|**A**) gives the probability that the customer will buy a whole life insurance given that we know the age and the income of the customer.

In contrast, *P*(H) is the *prior probability* of H. Using our example, is the probability that

any given customer will buy a whole life insurance regardless his income or age. $P(\mathbf{A})$ is the prior probability of **A**. The probability that a customer from our data set is 30 years old and earns $50,000.

Bayes' theorem use this concepts to provide a way of calculating the posterior probability, $P(\mathrm{H}|\mathrm{A})$, from $P(\mathrm{H})$, $P(\mathbf{A}|\mathrm{H})$ and $P(\mathbf{A})$. Equation.1: Bayes' Theorem.

$$P(H|\boldsymbol{A}) = \frac{P(\boldsymbol{A}|H)P(H)}{P(\boldsymbol{A})}$$

Equation.1: Bayes' Theorem

Another way to understand the relationship demonstrated by Bayes' theorem is through a Venn diagram. The Venn diagram in Figure 23: Probabilities of events A and Bshows the probabilities of events A and B occurring. The area of the left circle represents the probability that event A occurs. Similarly, the area of the right circle represents the probability that event B occurs. The overlapping area shows the probability that both events occur.

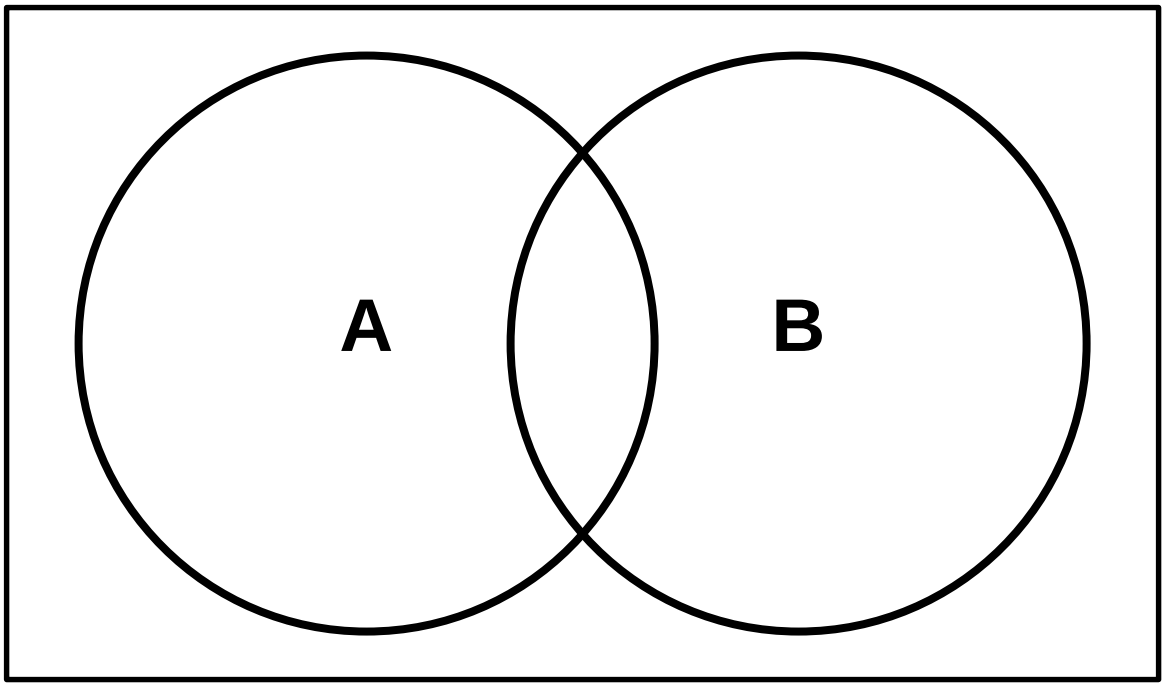

Figure 23: Probabilities of events A and B

Typically, the surrounding rectangle represents the entire realm of possibility. However, since it is given that event B has occurred, the probability of A occurring is equal to the portion of A and B occurring over the total area of B. Another way to formulate the probability of A and B occurring is the probability of B given A multiplied by the probability of A. Therefore

dividing this product by the probability gives the probability of A given B.

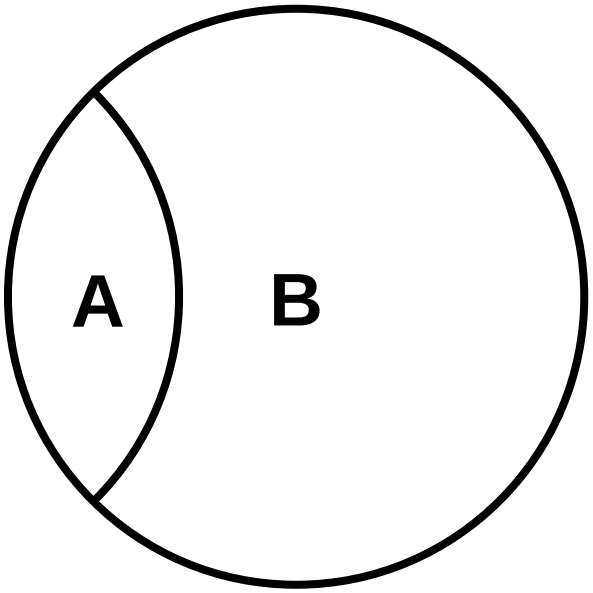

Figure24: Probabilities of events A and B given B

3.4.3 Classification

The naïve Bayesian classifier works as follows:

1. Let X be a training set of tuples and their associated class labels. Each tuple is represented by a vector, $\boldsymbol{A} = (\boldsymbol{a_1}, \boldsymbol{a_2}, \boldsymbol{a_3}, \ldots, \boldsymbol{a_n})$ depicting n measurements made on the tuple from n attributes, respectively, $X_1, X_2, \ldots, X_n$.

2. Suppose that there are m classes, $C_1, C_2, \ldots, C_m$. Given a tuple, $\boldsymbol{A}$, the classifier will predict that $\boldsymbol{A}$ belongs to the class having the highest posterior probability.
   The naïve Bayesian classifier predicts that the tuple $\boldsymbol{A}$ belongs to the class C_i, only if

$$P(C_i|\boldsymbol{A}) > P(C_j|\boldsymbol{A}) \text{ for } 1 \leq j \leq m, j \neq i.$$

   Thus, we maximize $P(C_i|\boldsymbol{A})$. The class C_i for which $P(C_i|\boldsymbol{A})$ is maximized is called the *maximum posteriori hypothesis*. Using Bayes' theorem:

$$P(Ci|\boldsymbol{A}) = \frac{P(\boldsymbol{A}|Ci)P(Ci)}{P(\boldsymbol{A})}$$

3. As $P(\boldsymbol{A})$ is constant for all classes, it is only necessary to maximize $P(\boldsymbol{A}|Ci)P(Ci)$. If the class' prior probabilities are not known, then we assume that the classes are equally likely, and we would maximize $P(\boldsymbol{A}|C_i)$. Otherwise, we maximize$P(\boldsymbol{A}|Ci)P(Ci)$.

The classes' prior probabilities could be calculated by $P(Ci) = \frac{|Ci,D|}{|D|}$ where $|C_{i,D}|$ is the number of training tuples of class C_i in D, where D is our dataset.

4. We estimate the probabilities $P(\boldsymbol{a}_1|C_i)$, $P(\boldsymbol{a}_2|C_i)$, …, $P(\boldsymbol{a}_n|C_i)$ from the training tuples. For each attribute we calculate in different way depending if it is continuous-valued or not. To calculate $P(\boldsymbol{A}|C_i)$, we consider:

a. If the attribute is categorical, then $P(x_k|C_i)$ is the number of tuples of class C_i in D.

b. If the attribute is continuous-valued, then we need to calculate its mean and standard deviation. It is assumed that the attribute it is going to have a Gaussian distribution.

$$g(a, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(a-\mu)^2}{2\sigma^2}}$$

Therefore,

$$P(x_k|C_i) = g(a_k, \mu_{Ci}, \sigma_{Ci}).$$

μ_{Ci} is the mean and σ_{Ci} is the standard deviation for the attribute A_k for training tuples of class C_i. With this two quantities we can plug to the equation to estimate our $P(x_k|C_i)$.

5. To predict the class label of $\boldsymbol{A}$, $P(\boldsymbol{A}|Ci)P(Ci)$ is evaluated for each class C_i.The classifier predicts that the class label of tuple $\boldsymbol{A}$ is the class C_i if and only if

$$P(\boldsymbol{A}|Ci)P(Ci) > P(\boldsymbol{A}|Cj)P(Cj) \qquad \text{for } 1 \leq j \leq m,\ j \neq i$$

In theory, Bayesian classifiers have the minimum error rate in comparison to all other classifiers; however, in practice this not always the case because we assume the class conditional independence.

We estimate $P(x_k|C_i)$ as the product of the probabilities $P(x_1|C_i)$, $P(x_2|C_i)$, …, $P(x_n|C_i)$,using the assumption of the class conditional independence. But, what happens if the

probability of one of the $P(x_k|C_i)$ is zero? This could happen if we don't have training tuples that represent a class. A cancel probability would cancel the effects of the other probabilities. To solve this we apply the *Laplacian correction*.

4 Summary

4.1 Advantages of Design

The component-based architecture chosen for DAKA is ideal because it allows for maximum separation of concerns. This allows each component to work with little or no dependencies on other components. Therefore, users of the framework can selectively use components to fit their needs.

By making use of the Hadoop framework, DAKA gains the ability to scale. Since Hadoop is an established open source software project, resources are readily available. This includes the software itself, documentation on use, as well as community support. The strengths of Hadoop, reliability and scalability, are incorporated into DAKA automatically.

Since DAKA is component-oriented and based on Hadoop, users can easily add their own components. These additional components could be integrations with databases or implementations of other algorithms. This extensibility allows the users to use DAKA to fit their individual purposes.

4.2 Disadvantages of Design

Since DAKA relies heavily on Hadoop, it is vulnerable to any issues with Hadoop. These issues could be defects or difficulties with deploying Hadoop. Because Hadoop is an actively maintained project, these issues should be limited. Another disadvantage is the dependence on multiple machines. Because DAKA and Hadoop scale by distributing work across machines, a cluster of machines is necessary to handle large amounts of data. Maintaining this cluster adds overhead that would not be present in a single machine system.

4.3 Design Rationale

DAKA's design allows for extensibility and reliability. Building on Hadoop allows DAKA to scale reliably for problems with large data sets. The component-based architecture allows the user to add or use functionality as needed.

5 Requirements Traceability Matrix

5.1 Traceability by Requirement

Non-functional requirements are labeled “NFR” to denote that there are no explicit software components designed to specifically satisfy them. These requirements have cross-cutting concerns in which many different components and environmental factors address.

Requirement	Design Component
3.1.1	2.5.2.1, 2.5.2.2, 2.5.2.3, 2.5.2.4, 2.5.2.5, 2.5.2.6, 2.5.3.1, 2.5.3.2, 2.5.3.3, 2.5.3.4 , 2.5.3.5
3.1.2	2.5.2.1, 2.5.2.2, 2.5.2.3, 2.5.2.4, 2.5.2.5, 2.5.2.6, 2.5.3.1, 2.5.3.2, 2.5.3.3, 2.5.3.4 , 2.5.3.5
3.1.3	2.5.2.1, 2.5.2.2, 2.5.2.3, 2.5.2.4, 2.5.2.5, 2.5.2.6, 2.5.3.1, 2.5.3.2, 2.5.3.3, 2.5.3.4 , 2.5.3.5
3.1.4	NFR
3.1.5	NFR
3.2.1	2.4.2.1
3.2.2	2.4.2.1
3.2.3	2.4.2.1
3.2.4	2.4.2.1
3.3.1.1	2.4.3.1
3.3.2.1	2.4.3.1

3.3.2.2	2.4.3.1
3.3.2.3	2.4.3.1
3.3.2.4	2.4.3.1
3.3.2.5	2.4.3.1
3.3.3.1	2.4.3.1
3.3.3.2	2.4.3.1
3.3.3.3	2.4.3.1
3.4.1	2.5.2, 2.5.3
3.4.2	2.5.2, 2.5.3
3.4.3	2.5.2, 2.5.3
3.5.1	2.3.2.1
3.5.2	2.3.2.1
3.5.3	2.3.2.1
3.5.4	2.3.2.1
3.5.5	2.3.2.1
3.5.6	2.3.2.1

5.2 Traceability by Design Component

Design Component	Requirement
2.3.2.1	3.5.1, 3.5.2, 3.5.3, 3.5.4, 3.5.6

2.4.2.1	3.2.1, 3.2.2, 3.2.3, 3.2.4
2.4.3.1	3.3.1.1, 3.3.2.1, 3.3.2.2, 3.3.2.3, 3.3.2.4, 3.2.2.5, 3.3.3.1, 3.3.3.2, 3.3.3.3
2.5.2	3.4.1, 3.4.2, 3.4.3
2.5.2.1	3.1.1, 3.1.2, 3.1.3
2.5.2.2	3.1.1, 3.1.2, 3.1.3
2.5.2.3	3.1.1, 3.1.2, 3.1.3
2.5.2.4	3.1.1, 3.1.2, 3.1.3
2.5.2.5	3.1.1, 3.1.2, 3.1.3
2.5.2.6	3.1.1, 3.1.2, 3.1.3
2.5.3	3.4.1, 3.4.2, 3.4.3
2.5.3.1	3.1.1, 3.1.2, 3.1.3
2.5.3.2	3.1.1, 3.1.2, 3.1.3
2.5.3.3	3.1.1, 3.1.2, 3.1.3
2.5.3.4	3.1.1, 3.1.2, 3.1.3
2.5.3.5	3.1.1, 3.1.2, 3.1.3

6 References

- An Implementation of the FP-growth Algorithm
  http://www.borgelt.net/papers/fpgrowth.pdf
- Mapper
  http://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/mapred/Mapper.html
- Reducer
  http://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/mapred/Reducer.html
- Han, Jiawei, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. 3rd ed. Amsterdam: Elsevier/Morgan Kaufmann, 2012. Print.