IMPUTAÇÃO EM LARGA ESCALA COM SPARK

Cristian Danner
Prática em Pesquisa Aplicada

Introdução

A tarefa de **imputação** tem como objetivo recuperar valores ausentes de maneira mais precisa, através de técnicas que variam desde [Little, 2015]:

- Média simples
- Regressão linear
- Modelos preditivos
- Algoritmos de aprendizado de máquina

Métodos de imputação são classificados em dois grupos:

- Métodos simples
- Métodos híbridos

Natureza dos dados:

- Numéricos: Contínuos ou discretos.
- **Categóricos: alfanuméricos.**

Motivação

- Abordagens convencionais para a complementação de dados ausentes tornam a análise tendenciosa.
- Soluções utilizando métodos estatísticos ou de aprendizado de máquina estão à disposição.
- Problema: aumento de inconsistências de dados.
- Consequência: prejudica a análise de dados e descoberta de conhecimento.
- Solução: imputação.

Objetivo

- Analisar os efeitos da aplicação de diversas estratégias de complementação de dados (***categóricos***) ausentes em larga escala.
- Permitir a composição dos experimentos de imputação utilizando várias tarefas de extração de conhecimento (***knowledge-discovery in databases - KDD***) e diferentes algoritmos.

Novos Desafios

As técnicas de imputação devem adequar-se aos novos desafios atuais:

- Grandes e variados volumes de dados (Big Data)
- Processamento em larga escala
- Proveniência
- Inteligência de Negócios (Business Intelligence)

Big Data

Processamento e armazenamento de imensos volumes de dados.
Caracterização pelo modelo dos cinco Vs [Chen et al., 2014]:

- Volume (grandes volumes)
- Variedade (diversas modalidades)
- Velocidade (geração rápida)
- Veracidade (dados verdadeiros)
- Valor (valor agregado)

Big Data demanda estruturas de processamento paralelo e distribuído.

Apache Spark

O Apache Spark é um sistema de processamento distribuído de código aberto utilizado para execução de tarefas de natureza big data.

Utiliza o armazenamento em cache na memória e a execução otimizada para obtenção de alto desempenho.

Apache Spark

- Spark provê uma interface para programação de clusters com paralelismo e tolerância a falhas.
- Foi desenvolvido nativamente na linguagem Scala, entretanto suporta também as linguagens Java, Python e R.
- Utiliza o modelo de programação MapReduce

Apache Spark - Arquitetura

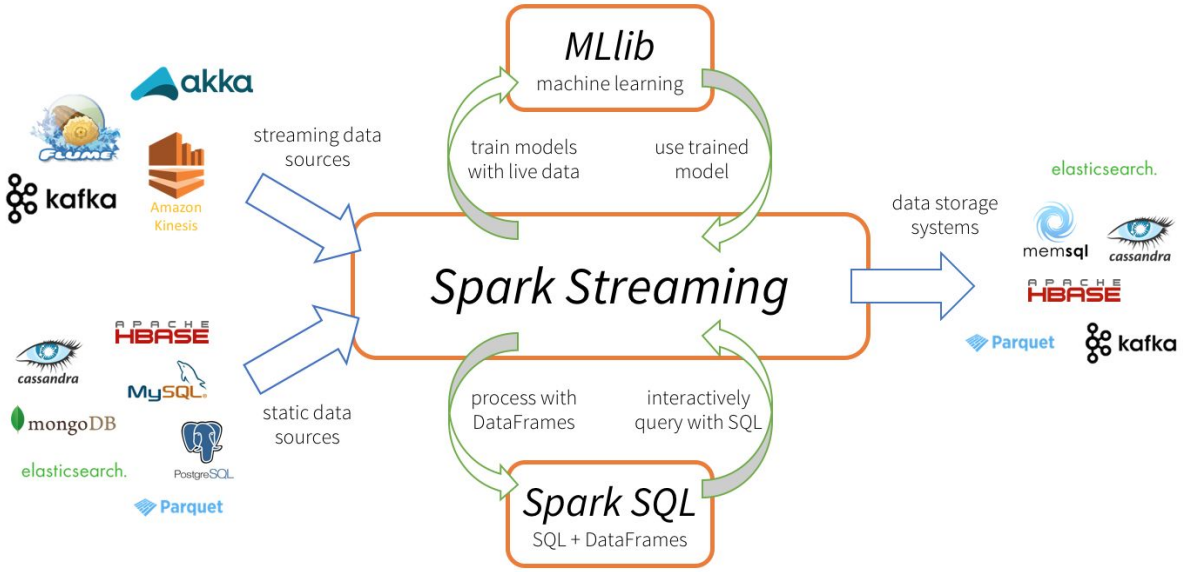


Figura 5: Spark. Fonte: Databricks - Apache Spark Streaming Concepts [2017]

Apache Spark - Arquitetura

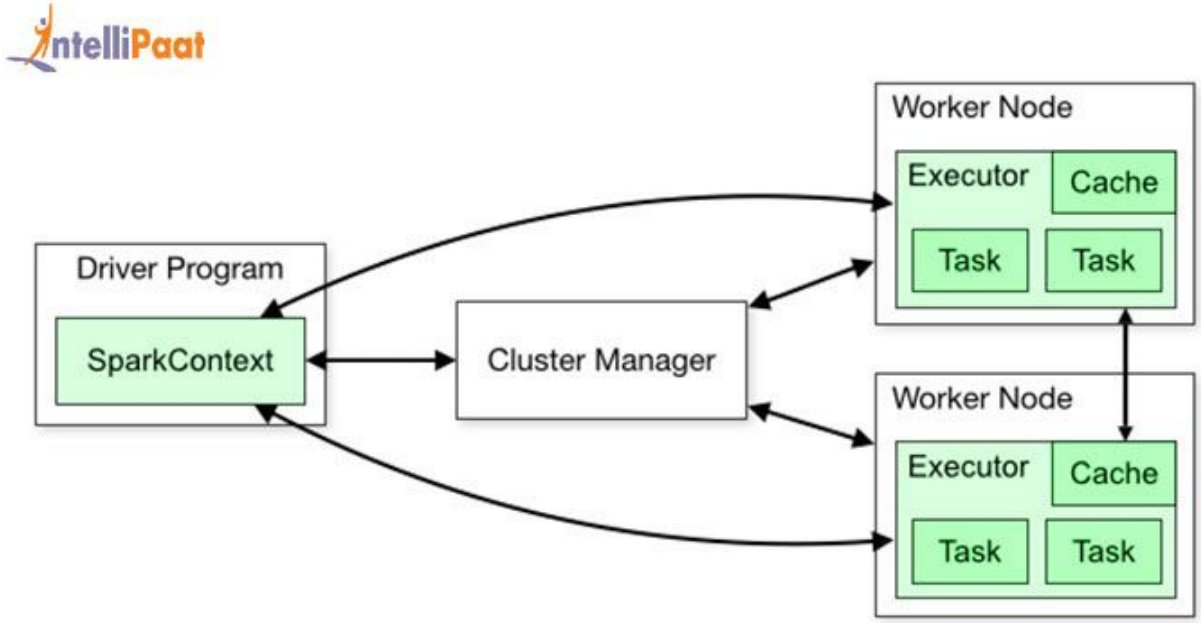


Figura 6: Arquitetura. Fonte: IntelliPaat – Spark: Arquitetura [2016]

Trabalhos Relacionados

- GOPALANI et al. [2015] ⇨ K-Means: implementação Spark

Tamanho dos dados	Número de *cores*	Tempo (s)
62MB	1	18
1240MB	1	149
1240MB	2	85

Tabela 1: Spark (MLib). Fonte: adaptado de Gopalani and Arora [2015].

Tamanho dos dados	Número de *cores*	Tempo (s)
62MB	1	44
1240MB	1	291
1240MB	2	163

Tabela 2: Apache Mahout. Fonte: adaptado de Gopalani and Arora [2015].

Appraisal-Spark

Objetivo Geral:

Desenvolvimento de uma ferramenta de imputação em bases de dados big data, garantindo a proveniência e reprodutibilidade de experimentos, com processamento de tarefas de forma distribuída e em larga escala.

Appraisal-Spark

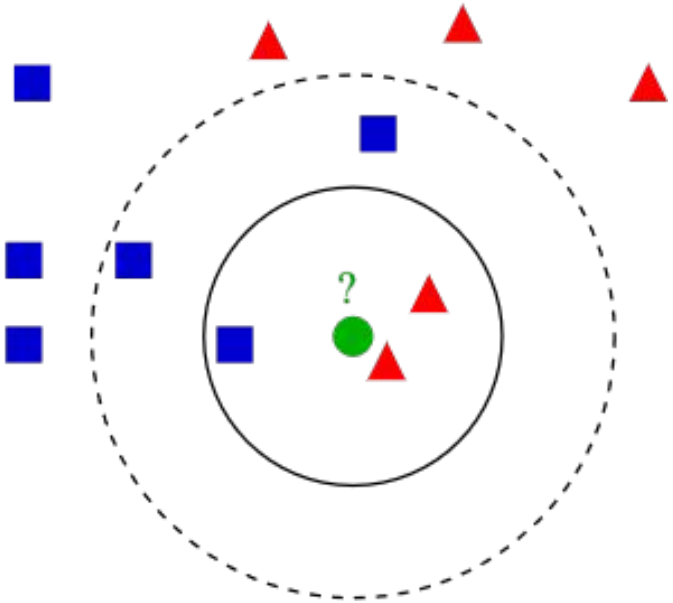
Objetivos Específicos:

- Permitir a construção de quaisquer planos de imputação, utilizando diversas técnicas de aprendizado de máquina, norteadas pelo modelo de imputação composta.
- Permitir a comparação de resultados de planos de imputação, através da utilização de diferentes parâmetros (execução de algoritmos) e amostras de dados.

k-Nearest Neighbor (k-NN)

O **k-NN** (***k vizinhos mais próximos***) é um dos algoritmos de classificação mais utilizados na área de aprendizagem de máquina.

A busca pela vizinhança é feita utilizando uma medida de distância nessa procura. No experimento desenvolvido, foi utilizado a distância Euclidiana.

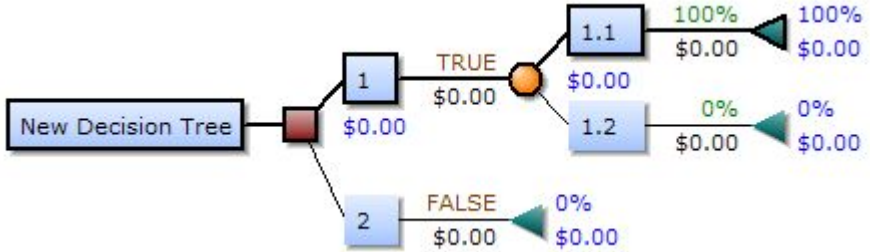


k-Nearest Neighbor (k-NN)

```
object Knn {

  def run(idf: DataFrame, k: Int, attribute: String, params: HashMap[String, Any] = null): entities.Entities.CategoricImputationResult = {

    val attributes: Array[String] = params("attributes").asInstanceOf[Array[String]]

    val removeCol = idf.columns.diff(attributes).filter(!_.equals("lineId"))
    val calcCol = attributes.filter(!_.equals(attribute))

    var fidf = utility.Utility.filterNullAndNonNumeric(idf.drop(removeCol: _*), calcCol)

    calcCol.foreach(att => fidf = fidf.withColumn(att, utility.Utility.toDouble(col(att))))

    fidf = fidf.withColumn("originalValue", col(attribute)).drop(attribute)

    val rdf = knn(fidf, k, calcCol).filter(_._2 == null)

    entities.Entities.CategoricImputationResult(rdf.map(r => entities.Entities.CategoricResult(r._1, r._2, r._3)))

  }

}
```

Decision Tree

O **Decision Tree** (***Árvore de Decisão***), é uma técnica de Mineração de Dados escolhida para identificar padrões de produção, por ser uma modelagem preditiva, exige a definição de um atributo *alvo* ou *preditor*. Esse atributo orienta o algoritmo a classificar os demais atributos, quando possível, para identificar padrões que se repitam no repositório de dados fornecido.



Decision Tree

```
object DecisionTree {

  def run(idf: DataFrame, attribute: String, params: HashMap[String, Any] = null, callback: String => Double): entities.Entities.NumericImputationResult = {

    val attributes: Array[String] = params("attributes").asInstanceOf[Array[String]]

    val removeCol = idf.columns.diff(attributes).filter(_ != "lineId")
    val remidf = idf.drop(removeCol: _*)

    val context = remidf.sparkSession.sparkContext

    val calcCol = attributes.filter(_ != attribute)

    var fidf = context.broadcast(utility.Utility.filterNullAndNonNumeric(remidf, calcCol))

    val columns = fidf.value.columns

    val lineIdIndex = columns.indexOf("lineId")
    val attributeIndex = columns.indexOf(attribute)

    val vectorsRdd = fidf.value.rdd.map(row => {

      val lineId = row.getLong(lineIdIndex)
      val attributeValue = row.getString(attributeIndex)
```

Random Forest

O **Random Forest** consiste em uma técnica de agregação de classificadores do tipo árvore, construídos de forma que a sua estrutura seja composta de maneira aleatória. A classificação final é dada pela classe que recebeu o maior número de votos entre todas as árvores da floresta.

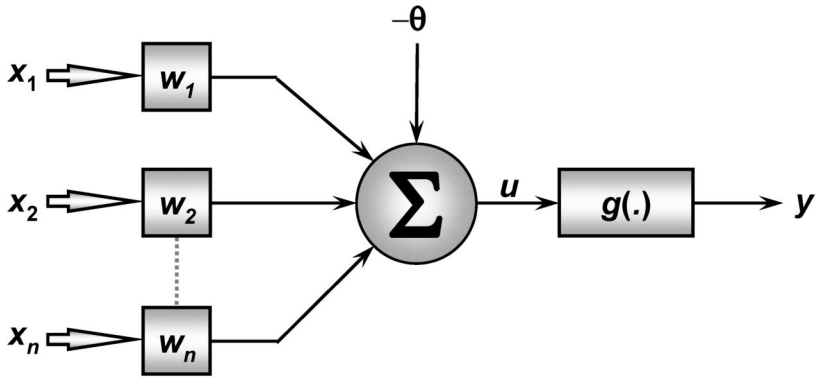
O classificador é baseado no método *Bagging*. Para cada árvore gerada é utilizado um conjunto de treinamento diferente, formado por *n* instâncias de treinamento escolhidas aleatoriamente.

Random Forest

```
object RandomForest {

  def run(idf: DataFrame, attribute: String, params: HashMap[String, Any] = null, callback: String => Double): entities.Entities.NumericImputationResult = {

    val attributes: Array[String] = params("attributes").asInstanceOf[Array[String]]

    val removeCol = idf.columns.diff(attributes).filter(_ != "lineId")
    val remidf = idf.drop(removeCol: _*)

    val context = remidf.sparkSession.sparkContext

    val calcCol = attributes.filter(_ != attribute)

    var fidf = context.broadcast(utility.Utility.filterNullAndNonNumeric(remidf, calcCol))

    val columns = fidf.value.columns

    val lineIdIndex = columns.indexOf("lineId")
    val attributeIndex = columns.indexOf(attribute)

    val vectorsRdd = fidf.value.rdd.map(row => {

      val lineId = row.getLong(lineIdIndex)
      val attributeValue = row.getString(attributeIndex)
```

Multilayer Perceptron

A **Multilayer Perceptron** (***Perceptron multicamadas***) é uma rede neural semelhante à perceptron, mas com mais de uma camada de neurônios em alimentação direta. Tal tipo de rede é composta por camadas de neurônios ligadas entre si por sinapses com pesos. O aprendizado nesse tipo de rede é geralmente feito através do algoritmo de Retropropagação (Backpropagation) do erro.



Multilayer Perceptron

```
object MultilayerPerceptron {

  def run(idf: DataFrame, attribute: String, params: HashMap[String, Any] = null, callback: String => Double): entities.Entities.NumericImputationResult = {

    val attributes: Array[String] = params("attributes").asInstanceOf[Array[String]]

    val removeCol = idf.columns.diff(attributes).filter(_ != "lineId")
    val remidf = idf.drop(removeCol: _*)

    val context = remidf.sparkSession.sparkContext

    val calcCol = attributes.filter(_ != attribute)

    var fidf = context.broadcast(utility.Utility.filterNullAndNonNumeric(remidf, calcCol))

    val columns = fidf.value.columns

    val lineIdIndex = columns.indexOf("lineId")
    val attributeIndex = columns.indexOf(attribute)

    val vectorsRdd = fidf.value.rdd.map(row => {

      val lineId = row.getLong(lineIdIndex)
      val attributeValue = row.getString(attributeIndex)

      var values = new Array[Double](calcCol.length)
```

Experimentação

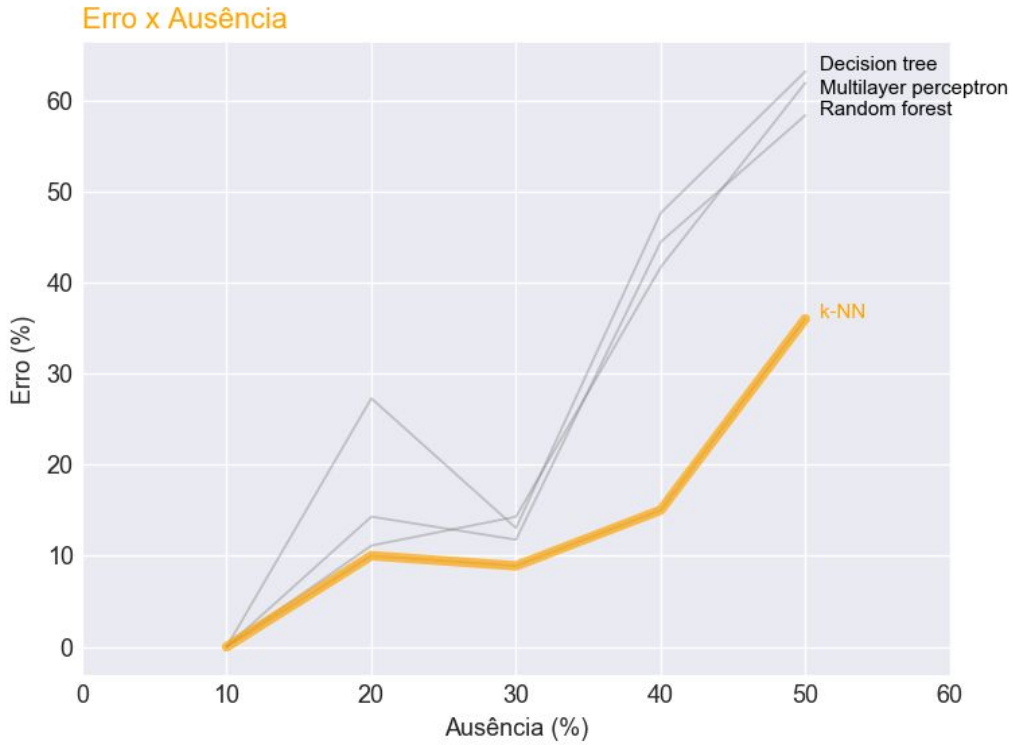
Resultados Obtidos com o Appraisal 2.11

- Foram reproduzidos os experimentos de imputação composta de Soares [2007] nas bases de dados: iris plants e treatment of migraine headaches, utilizando a nova versão do sistema.
- Foram simuladas ausências de dados de dez, vinte, trinta, quarenta e cinquenta por cento para cada um dos atributos das bases de dados.

Experimentação

Resultados Obtidos com o Appraisal 2.11

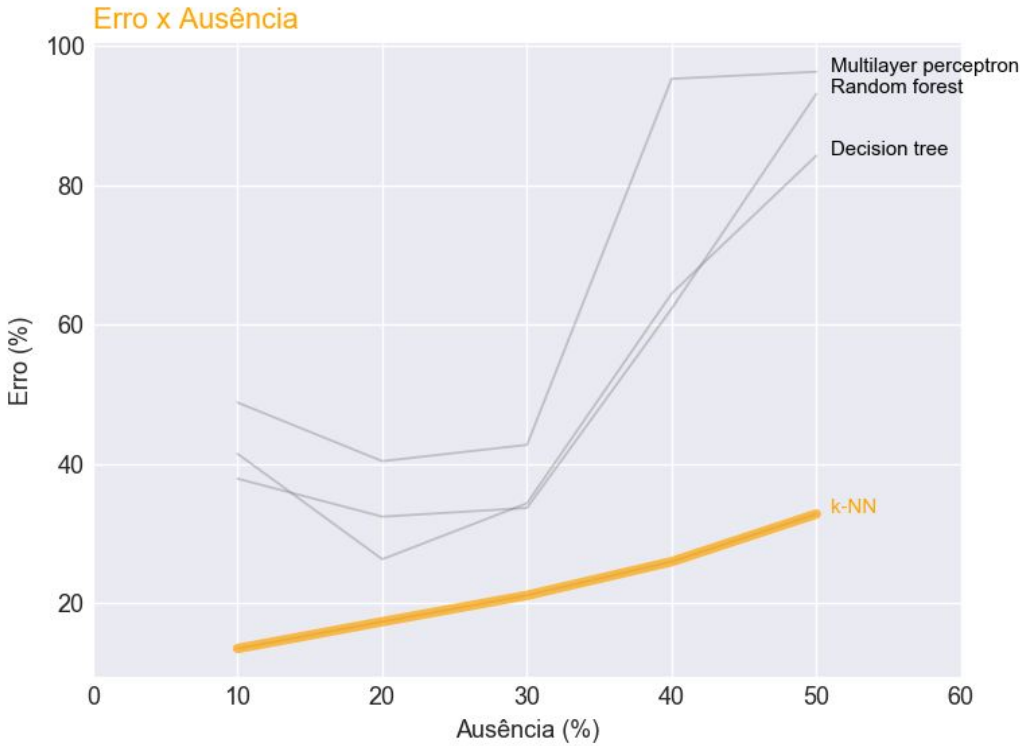
Iris dataset:



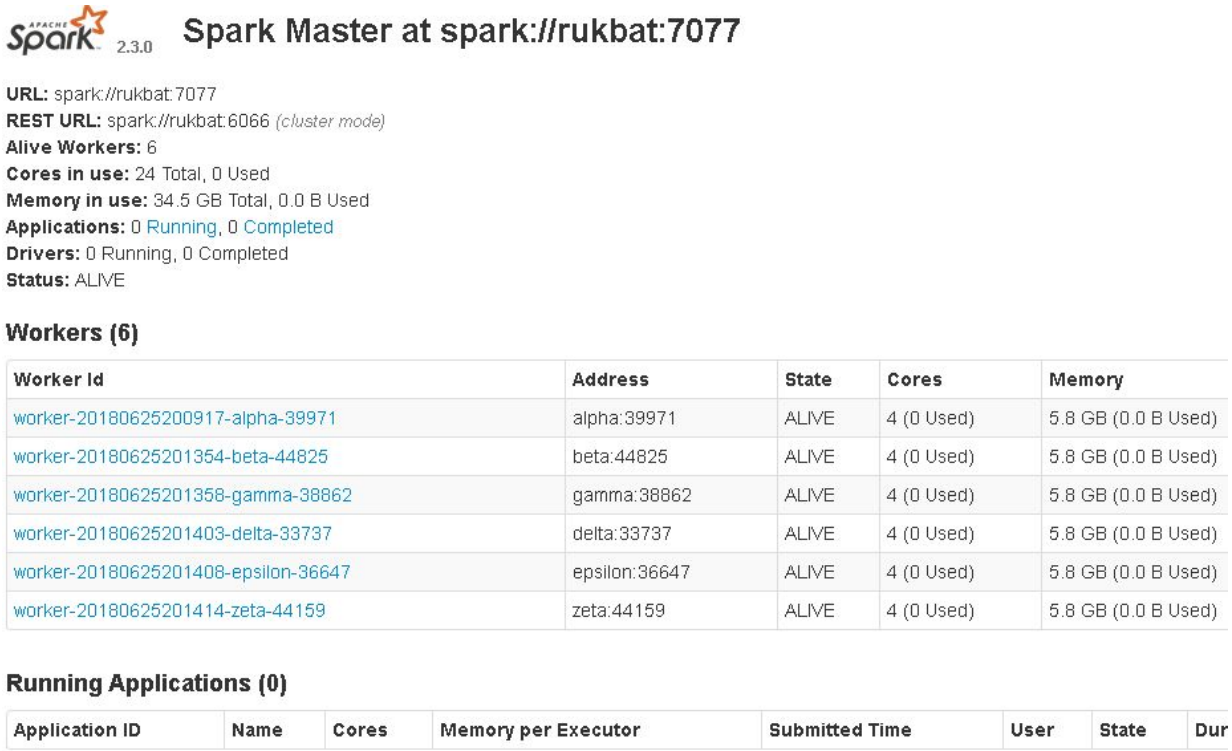
Experimentação

Resultados Obtidos com o Appraisal 2.11

Headache dataset:



Cluster



Spark Master at spark://rukbat:7077

URL: spark://rukbat:7077
REST URL: spark://rukbat:6066 *(cluster mode)*
Alive Workers: 6
Cores in use: 24 Total, 0 Used
Memory in use: 34.5 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (6)

Worker Id	Address	State	Cores	Memory
worker-20180625200917-alpha-39971	alpha:39971	ALIVE	4 (0 Used)	5.8 GB (0.0 B Used)
worker-20180625201354-beta-44825	beta:44825	ALIVE	4 (0 Used)	5.8 GB (0.0 B Used)
worker-20180625201358-gamma-38862	gamma:38862	ALIVE	4 (0 Used)	5.8 GB (0.0 B Used)
worker-20180625201403-delta-33737	delta:33737	ALIVE	4 (0 Used)	5.8 GB (0.0 B Used)
worker-20180625201408-epsilon-36647	epsilon:36647	ALIVE	4 (0 Used)	5.8 GB (0.0 B Used)
worker-20180625201414-zeta-44159	zeta:44159	ALIVE	4 (0 Used)	5.8 GB (0.0 B Used)

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Dur

Considerações Finais

- É possível imaginar que em ambientes Big Data, novos conhecimentos sejam descobertos acerca do tema imputação (grandes volumes de dados x flutuações individuais).
- A capacidade de utilizar diferentes planos de imputação, além dos cinco propostos por Soares [2007], pode permitir que novas técnicas de imputação sejam descobertas.
- É uma ferramenta capaz de impulsionar a pesquisa científica nas áreas de imputação, KDD e ML.

Perguntas

IMPUTAÇÃO EM LARGA ESCALA COM SPARK

Cristian Danner
Prática em Pesquisa Aplicada