SPARK-4588: ML Attributes

This document is publicly visible.

Authors: Xiangrui Meng, Sean Owen
Revision History:

- Feb 2015: initial version

Overview

We use DataFrames as ML datasets in the new ML pipeline API. Each dataset consists of typed columns, e.g., string, double, vector, etc. For ML purposes, knowing the column type is not sufficient to handle data properly. For example, a column containing country indices stored as integers 0, 1, 2, … cannot be used directly in a linear algorithm. Because the value 0, 1, 2, … are just indices, including them in some arithmetics doesn't make sense. However, algorithms cannot infer this information by just looking at the values. So we want to provide ML data types and their associated description along with the data, referred as ML attributes. This design doc discusses ML attributes and the proposed API.

Requirements

- Support common ML data types and metadata.
- Support attributes for vector-typed columns.
- Convert to/from SQL's metadata with efficient storage.
- Properly handle unknown attributes.
- Reference attributes.
- APIs are Java friendly.

Interfaces and design discussions

Attribute types

We start with ML attribute types. There are two basic types:

1. **numeric**, e.g., 1.24, -5, 1.0, where the numeric values themselves are meaningful. Numeric values are comparable and could be used in arithmetic. The values could be discrete or continuous.
2. **nominal/categorical**, e.g., "true"/"false", "small"/"medium"/"large", where the values fall into some predefined categories. The categories could be ordinal, e.g., "small" < "medium" < "large".

Under the two basic types, we can have sub-types:

1. **numeric and continuous** (e.g., length: 1.23, 2.52, 5.13, …).
2. **numeric and discrete** (e.g., hourOfDay: 0, 1, 2, 3, …, 23).
3. **nominal and ordinal** (e.g., clothingSize: "small", "medium", "large", "x-large").

4. **nominal but not ordinal** (e.g., zipCode: “94015”, “94305”, …).
5. **binary** (e.g., “true”/”false”). This is a special sub-type because if there are two categories they can be represented by 0/1 and 0/1 could be treated as numeric values.

We don’t make separate types for discrete and continuous, because most ML algorithms do not treat discrete values differently from continuous values.

ML columns

We call a column with ML attributes attached an ML column. **The data in ML columns are stored in as Doubles**. For a numeric data column, we store its values directly. For a nominal column, we store the encoded indices instead. For example, in an ML column containing cloth sizes, we save the category indices instead of the raw categories and store the raw categories in the metadata:

	size column	ML column
metadata		[small, medium, large, x-large]
	small	0.0
	medium	1.0
	small	0.0
	large	2.0

We chose using Double to store all values for simplicity. We could use boolean for binary data and integer for nominal data. But when we group data into vectors, they becomes Double.

For a nominal but not ordinal column, we recommend indexing values by their frequency to enhance sparsity. For a nominal and ordinal column, we require indexing based on the order.

Attribute group

For a vector column, we need to store ML attributes for its inner columns. We treat them as a group of attributes.

Attribute names

An attribute can be optionally associated with a name. For a scalar ML column, the attribute name is the column name. For a vector column, the attribute group name is the column name. In an attribute group, the attribute names are optional, though recommended.

JSON serialization

Attributes can be serialized into JSON and loaded back from JSON.

SQL data type conversion

An attribute can be converted into SQL's Metadata and stored in a StructField instance. We store attribute information as a subfield called "ml.attr" in the Metadata to avoid confliction with other metadata. Since StructField already has a name field, we don't store attribute or attribute group names in the metadata. The following is the DataType's JSON representation for an example scalar ML column:

```
{
  "name": "clothingSize",
  "type": "double",
  "nullable": false,
  "metadata": {
    "ml.attr": {
      "type": 1, // nominal
      "values": ["small", "medium", "large", "x-large"],
      "isOrdinal": true
    }
  }
}
```

and for a vector ML column:

```
{
  "name": "user",
  "type": "udt",
  "class": "o.a.s.m.l.VectorUDT",
  "nullable": false,
  "metadata": {
    "ml.attr": {
      "attributes": [
        {},
        {"name": "age"},
        {"name": "gender", "type": 2, "values": ["male", "female"]}
      ]
    }
  }
}
```

Unknown attributes

When we check the pipeline without touch any data, it may be hard to figure out the exact ML attributes. In this case, we should allow unknown attributes and components should be optimistic about unknown attributes. If an attribute info is not given, it is treated as numeric.

Unknown values, e.g. names, could be represented by Scala's Option, but it is not very Java friendly. So we provide a builder pattern to create attributes from the default attribute for each attribute type.

```
val clickedAttr = BinaryAttribute.defaultAttr()
  .withName("clicked")
  .withValues("true", "false")

val clothingSizeAttr = NominalAttribute.defaultAttr()
  .withName("clothingSize")
  .withValues("small", "medium", "large")
  .withOrdinal()
```

Attribute references

An attribute could be referenced by name, or by index if it is in an attribute group. For simplicity, we don't provide APIs to support referencing an attribute inside a group. Users can use a transformer VectorSlicer to slice a vector column.

```
val slicer = new VectorSlicer()
  .setInputCol("user")
  .setSelectedNames("country", "gender")
  .setSelectedIndexes(2, 5)
  .setOutputCol("slicedUser")

val assember = new VectorAssembler()
  .setInputCols("time", slicer.getOutputCol)
  .setOutputCol("features")
```

Storage

We should care about storage to properly handle millions of features. We don't output a key to JSON/Metadata if its value is the default value. For example, `{}` indicates a numeric attribute without a name, `{"name": "age"}` indicates a numeric attribute with name "age", and `{"name": "gender", "type": 2}` means a binary attribute with name "gender" but the raw categories are not given.

Summary statistics

It would be useful to save some summary statistics of the column in ML attributes, e.g., min, max, std, and nnz for a numeric attribute, and histogram for a nominal attribute.

Basic transformers as examples

We list a few basic transformers to demonstrate how ML attributes are generated.

LabelIndexer

`LabelIndexer` turns a column of string labels into a nominal ML column.

	clothingSize	encodedClothingSize

metadata		{
		“type”: 1,
		“values”: [“small”, “medium”, “large”, “x-large”]
		}

	small	0.0
	medium	1.0
	small	0.0
	large	2.0

OneHotEncoder

OneHotEncoder reads a nominal ML column and outputs a vector column with binary features.

	encodedClothingSize	clothingSizes

metadata	{
	“type”: “nominal”,
	“values”: [“small”, “medium”, “large”, “x-large”]
	}
		“attributes”: [
		{“name”: “is_small”, “type”: “binary”},
		{“name”: “is_medium”, “type”: “binary”},
		{“name”: “is_large”, “type”: “binary”}
		]

	0.0	[1, 0, 0]
	1.0	[0, 1, 0]
	0.0	[1, 0, 0]
	2.0	[0, 0, 1]

Binarizer
Binarizer reads a numeric vector column and turns nonzero values into 1.0.

	user	binUser

metadata	“attributes”: [{“name”: “a”}, {“name”: “b”}]	“attributes”: [{“name”: “a”, “type”: “binary”}, {“name”: “b”, “type”: “binary”}]
	[0.0, 1.5]	[0, 1]
	[0.0, 2.0]	[0, 1]
	[3.5, 2.0]	[1, 1]

VectorAssembler

VectorAssembler merges multiple columns into a single vector column.

	length	user	output
metadata	{}	“attributes”: [{“name”: “a”}, {“name”: “b”}]	“attributes”: [{“name”: “length”}, {“name”: “user_a”}, {“name”: “user_b”}]
	2.0	[0.0, 1.5]	[2.0, 0.0, 1.5]
	3.0	[0.0, 2.0]	[3.0, 0.0, 2.0]
	0.0	[3.5, 2.0]	[0.0, 3.5, 2.0]

The attribute naming convention is not discussed in this doc.

Interfaces

```
sealed trait Attribute extends Serializable {

  /** Attribute type: {0: numeric, 1: nominal, 2: binary}. */
  def attrType: Int

  /** Name of the attribute. None if it is not set. */
  def name: Option[String] = None

  /** Copies this attribute with a new name. */
  def withName(name: String): Attribute = copyWithName(Some(name))
  def withoutName: Attribute = copyWithName(None)
  private[ml] def copyWithName(name: Option[String]): Attribute

  /** Index of the attribute. None if it is not set. */
```

```
  def index: Option[Int] = None

  /** Copies this attribute with a new index. */
  def withIndex(index: Int): Attribute
  def withoutIndex: Attribute
  ...

  def isNumeric: Boolean

  def isNominal: Boolean

  def isOrdinal: Boolean

  /** Get the JSON representation of this attribute. */
  def toJson: String = compact(render(jsonValue))

  private[ml] def jsonValue: JValue

  /** Converts this attribute to metadata. */
  def toMetadata: Metadata
}

/** Trait containing methods to create attributes from JSON/Metadata. */
private[ml] trait AttributeFactory {

 private[ml] def fromJsonValue(json: JValue): Attribute

 def fromJson(json: String): Attribute = fromJsonValue(parse(json))

 def fromMetadata(metadata: Metadata): Attribute
}

class NumericAttribute private[ml] (
   override val name: Option[String] = None,
   override val index: Option[Int] = None,
   val min: Option[Double] = None,
   val max: Option[Double] = None,
   val std: Option[Double] = None,
   val sparsity: Option[Double] = None)
 extends Attribute {

 override def attrType: Int = Attribute.NUMERIC

 ...

 private[ml]
 def copyWithName(name: Option[String]): Attribute = copyWith(name = name)
```

```
 ...

 private[ml] def copyWith(
     name: Option[String] = name,
     index: Option[Int] = index,
     min: Option[Double] = min,
     max: Option[Double] = max,
     std: Option[Double] = std,
     support: Option[Double] = support) = {
   new NumericAttribute(name, index, min, max, std, support)
 }
}

object NumericAttribute extends AttributeFactory {

 /** A default numeric attribute. */
 val defaultAttr: NumericAttribute = new NumericAttribute

 ...
}

class NominalAttribute private[ml] (
   override val name: Option[String] = None,
   override val index: Option[Int] = None,
   val numValues: Option[Int] = None,
   val values: Option[Array[String]] = None,
   val isOrdinal: Option[Boolean] = None)
 extends Attribute {

 override def attrType: Int = Attribute.NOMINAL

 ...

 /** Index of a specific value. */
 def indexOf(value: String): Int

  ...
}

object NominalAttribute extends AttributeFactory {

 val defaultAttr: NominalAttribute = new NominalAttribute()

 ...
}

class BinaryAttribute private[ml] (
    override val name: Option[String],
```

```
    override val index: Option[Int],
    val values: Option[(String, String)])
  extends Attribute {
  ...
}

object BinaryAttribute extends AttributeFactory {

  val defaultAttr: BinaryAttribute

  ...
}

class AttributeGroup(val name: String, val attributes: Array[Attribute]) {

 /** Number of attributes. */
 def size: Int

 /** Index of an attribute specified by name. */
 def indexOf(attrName: String): Int

 /** Gets an attribute by name. */
 def apply(attrName: String): Attribute

 private[ml] def jsonValue: JValue

 def toJson: String

  def toMetadata: Metadata
}

object AttributeGroup {

  def fromJsonValue(json: JValue): AttributeGroup

  def fromJson(json: String): AttributeGroup

  def fromMetadata(metadata: Metadata): AttributeGroup
}
```

Updatability

Upon any JSON/Metadata schema change, we should guarantee that previously saved data can be correctly loaded back.