Latent Dirichlet Allocation (LDA) in the Pipelines API

Public Design Doc

Table of Contents

Document history

- Authors: Joseph Bradley
- October 2015: Initial draft.

Links

- Main JIRA: SPARK-5565

Development plan

- ML Sprint 6 (October)
  - Get initial Scala API into Spark
  - Get Python API into Spark
- Monitor for feature requests since it is unclear what features are needed.

Changes from this doc in initial implementation

- I am eliminating the doc ID. This is not very necessary with DataFrames since users can include a doc ID as another column if needed. The algorithm does not need to enforce its presence.

- I am modifying the spec for topDocumentsPerTopic since documents no longer have IDs. We can make use of DataFrame groupByKey, etc. instead.

Requirements for API

- Provide all LDA functionality which the spark.mllib API provides.
- Fit the Pipelines API, in which most outputs are provided via DataFrame columns.
- Resemble the spark.mllib API when possible to ease the transition for users.

Design considerations

Use cases

What are the main use cases? Relatedly, how can we categorize queries?

- Feature generation
  - Return datum per Row. Part of a Pipeline.
  - `topicDistributions, topTopicsPerDocument, topicAssignments`
- Model inspection
  - Return data of arbitrary shape. Not part of a Pipeline.
  - `describeTopics, topicsMatrix, topDocumentsPerTopic`
- Evaluation
  - Return scalar, or a scalar per Row.
  - `logLikelihood, perplexity`

DistributedLDAModel: Storing training data in the model

Issue:

- Certain LDA optimizers store info about the entire training dataset, and that info is hard to reproduce later (costly, or impossible without retraining due to randomness).
- It can be useful to access that data later for:
  - Model inspection
  - Making predictions on training data (without recomputing)

In spark.mllib

This is solved by providing 2 types of LDAModel subclasses, one for each optimizer:

- LocalLDAModel for Online optimizer
- DistributedLDAModel for EM optimizer

In spark.ml

Q: Can we provide similar functionality?

- Model inspection: Yes. Use a similar class hierarchy (discussed below).
- Avoiding recomputation: Maybe via caching or fitAndTransform() (discussed below).

Q: How does this work with Pipeline?

- A PipelineModel needs to use a single Model type.
- Not a problem: The PipelineModel can use the abstract type, and it will be oblivious to the output columns.

Proposed API for spark.ml

Items to discuss:

- Class structure
- Input data type
- Model.transform() columns
- Model inspection return types
- Evaluation

Class structure

Q: Should we keep a class structure?

- Pros
  - Class structure reflects semantic differences between models produced by different optimizers, which support different queries (for model inspection).
- Cons
  - LDA.fit() will return an abstract LDAModel, which must then be cast to the appropriate type in order to make certain queries.

Proposal: Yes, but modify the class structure. There will be a concrete LDAModel class, which is extended by a concrete DistributedLDAModel class. This will be less confusing to users, while still allowing expert users to access training data info from a DistributedLDAModel.

Proposed Scala API

```
class LDA {
  def fit(data: DataFrame): LDAModel
}

class LDAModel {
  def logLikelihood(data: DataFrame): Double
  ...
  def isDistributed: Boolean
```

```
}

class DistributedLDAModel extends LDAModel {
  def trainingLogLikelihood(): Double
  ...
}
```

The Python API will follow the same structure. We will create a private[ml] LDAModelWrapper to implement this API.

Basic usage

```
val model = lda.fit(trainingData)
val logLikelihood = model.logLikelihood(data)
// transform & evaluate will be used for advanced usage too.
val predictions = model.transform(data)
evaluator.evaluate(model, data)
```

Advanced usage

```
val model = lda.fit(trainingData)
val ll = model match {
  case m: DistributedLDAModel => model.logLikelihood
  case m: LDAModel => model.logLikelihood(trainingData)
}
```

Alternatives which avoid the type issue:

- Provide multiple LDA Estimators, one for each Optimizer.
    - Con: This will force non-expert users to choose an algorithm.
- Provide a single LDAModel type which throws errors when methods are called if the Optimizer did not store the needed data.
    - Con: This API would be confusing and cause runtime errors.

Alternative: Rename to provide 1 model type per Optimizer. E.g., EMLDAModel, OnlineLDAModel, GibbsLDAModel instead of LocalLDAModel, DistributedLDAModel.

- This should not be necessary; local (no info on training data) vs. distributed (info on training data) should be the only distinction.

Input data type

Q: What DataFrame schema should LDA take?
Proposal: We will initially accept a Vector column of word counts + a Long column of doc IDs. This will be most similar to the spark.mllib API.

- Later on, we can add support for other input column types, such as Seq[String].

Avoiding recomputation

Issue: A DistributedLDAModel stores info about its training data which can be output as columns in the output DataFrame (e.g., `topicDistributions`).

Q: How can we avoid recomputing it when the user calls transform() on the training data?
Proposal: Ignore this issue for now since it goes beyond LDA. Eventually, do one of the following:

- Provide fitAndTransform(), as in sklearn. We want to do this for all Estimators.
- Store a transient reference to both the training and transformed DataFrames in the Model, and recognize when transform() is called on the same DataFrame.

Q: Should we guarantee that these 2 queries return the same result?

- `DistributedLDAModel.myResult`
  - This takes data from the stored training set.
- `DistributedLDAModel.transform(trainingData).select("myResult")`
  - This could re-run inference on the training data, returning a different result due to randomness. (The inference process is different than during training since the topics remain fixed.)

Proposal: No. Do not guarantee it. Do not specify anything in the doc so there is no contract.

Model.transform(DataFrame) columns

This section discusses per-document (per-Row) outputs from calling `transform()` on a new `DataFrame`.

Proposal: Share these columns for all model types, regardless of the optimizer.

- Pro: Simpler API, eventually.
- Con: Currently, not all models will support all output columns, so we will throw errors at runtime when the user specifies an unsupported output.

Proposal: Output topicDistribution column only by default.

Parameterized output columns: Several queries in spark.mllib include threshold parameters, which may affect the results in output columns. These will be specified as Model Params.

Proposed output columns: The initial API will only include the first output column. We can provide the 2nd two as methods if needed, or wait and provide them as output columns later, depending on user feedback.

Column name	spark.mllib method	Schema	Description
topicDistribution	`topicDistributions`	Vector	length # topics

topTopics	`topTopicsPerDocument`	Array(
		  Struct(
		  Field(“topic”, Int)
		  Field(“weight”, Double)
		  )
		)
			length <= # topics. Sorted & truncated to top-weighted topics.

termTopicAssignments	`topicAssignments`	Array(
		  Struct(
		  Field(“term”, Int)
		  Field(“topic”, Int)
		  )
		)
			length # terms. Done per-term because of input data being unordered.

Q: Should we flatten output column schemas?
E.g.: In spark.mllib, `topTopicsPerDocument` returns a pair `Array[Int], Array[Double]` for each document. In spark.ml, what should the output schema be?

- One column: `ArrayType` of `StructType` with fields `topic: Int, weight: Double`
- Two columns: `ArrayType` of `Int`, `ArrayType` of `Double` for topics, weights

Decision: No. Keep structured.

- We may use MapType for topicAssignments, but it depends on whether maps become easier to work with in Spark SQL. We can delay the decision until we add it post-MVP.

Model inspection return types

Q: What return types should we use for model inspection methods?
Proposal: Local DataFrames for any non-trivial types.

E.g., for **`describeTopics:`**

`topic`	`termIndices`	`termWeights`	`terms`
Integer	Array(Integer)	Array(Double)	Array(String)
topic index	*sorted term indices*	*sorted term weights*	*sorted term Strings, if available*

`topicsMatrix:` `Matrix` type (not a DataFrame)
`topDocumentsPerTopic:` `DataFrame`

Evaluation

Proposal:

- Provide `logLikelihood, perplexity` methods returning scalar metrics for a dataset, as in spark.mllib.

Future work (out of scope):

- Add per-Row output columns, in some cases.
  - This should come later since it may require discussion to decide on the best evaluation metrics. (There are multiple ways to calculate likelihood & perplexity.)
- We will add an Evaluator for clustering later on.
  - Complication: For evaluation, it can be important to consider the penalty from the model prior as well, so this output column may not be sufficient for an Evaluator.

Future

The main issues for the future are:

- Adding another optimizer (Gibbs sampling)
  - *This should fit within the DistributedLDAModel API.*
- Provide per-Row evaluation metrics via new output columns
  - *Requires some discussion*
- Providing an Evaluator for LDA model selection
  - *Requires design doc*
- Avoiding recomputation when transforming training data
  - *Requires design doc*

==========
 # Casual readers can ignore stuff below.
==========

Functionality in spark.mllib

We list LDA's current functionality as of Spark 1.5. This does not all have to be supported in Pipelines via output columns, but some are important, as highlighted by the use cases below.

```
abstract class LDAModel {
  // For each topic, return top-weighted terms as:
  //   (term indices, term weights), sorted
```

```
  def describeTopics(
    maxTermsPerTopic: Int): Array[(Array[Int], Array[Double])]

  // #terms x #topics matrix
  def topicsMatrix: Matrix
}

class LocalLDAModel {
  // evaluation metric (similar API for other metrics)
  def logLikelihood(documents: RDD[(Long, Vector)]): Double

  // RDD over documents: (doc ID, topic distribution)
  def topicDistributions(
    documents: RDD[(Long, Vector)]): RDD[(Long, Vector)]

}

class DistributedLDAModel {
  // evaluation metric which operates on dataset stored by model
  def logLikelihood: Double

  // For each topic, return top docs as:
  //   (doc ID, doc weight), sorted
  def topDocumentsPerTopic(
    maxDocumentsPerTopic: Int): Array[(Array[Long], Array[Double])]

  // For each doc, return top topics as:
  //   (doc ID, topic index, topic weight), with topic arrays sorted
  def topTopicsPerDocument(
    k: Int): RDD[(Long, Array[Int], Array[Double])]

  // For each doc, return (term, topic) matching for all terms in
doc:
  //   (doc ID, term indices, topic indices)
  def topicAssignments: RDD[(Long, Array[Int], Array[Int])]

  // RDD over documents: (doc ID, topic distribution)
  def topicDistributions: RDD[(Long, Vector)]
}

// not in Spark, but could be
(FUTURE) class GibbsSamplingLDAModel {
  // For each doc, return current topic assigned to each token:
```

```
  //   (doc ID, topic indices corresponding to tokens)
  // Note: A “token” is an instance of a term in a doc.
  def topicAssignments: RDD[(Long, Array[Int])]
}
```