ML Pipelines

In this section, we introduce the concept of ***ML Pipelines***. ML Pipelines provide a uniform set of high-level APIs built on top of DataFrames that help users create and tune practical machine learning pipelines.

Table of Contents

Main concepts in Pipelines

MLlib standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline, or workflow. This section covers the key concepts introduced by the Pipelines API, where the pipeline concept is mostly inspired by the scikit-learn project.

- **`DataFrame`**: This ML API uses `DataFrame` from Spark SQL as an ML dataset, which can hold a variety of data types. E.g., a `DataFrame` could have different columns storing text, feature vectors, true labels, and predictions.
- **`Transformer`**: A `Transformer` is an algorithm which can transform one `DataFrame` into another `DataFrame`. E.g., an ML model is a `Transformer` which transforms a `DataFrame` with features into a `DataFrame` with predictions.
- **`Estimator`**: An `Estimator` is an algorithm which can be fit on a `DataFrame` to produce a `Transformer`. E.g., a learning algorithm is an `Estimator` which trains on a `DataFrame` and produces a model.
- **`Pipeline`**: A `Pipeline` chains multiple `Transformer`s and `Estimator`s together to specify an ML workflow.
- **`Parameter`**: All `Transformer`s and `Estimator`s now share a common API for specifying parameters.

DataFrame

Machine learning can be applied to a wide variety of data types, such as vectors, text, images, and structured data. This API adopts the `DataFrame` from Spark SQL in order to support a variety of data types.

`DataFrame` supports many basic and structured types; see the Spark SQL datatype reference for a list of supported types. In addition to the types listed in the Spark SQL guide, `DataFrame` can use ML `Vector` types.

A `DataFrame` can be created either implicitly or explicitly from a regular `RDD`. See the code examples below and the Spark SQL programming guide for examples.

Columns in a `DataFrame` are named. The code examples below use names such as "text," "features," and "label."

Pipeline components

Transformers

A `Transformer` is an abstraction that includes feature transformers and learned models. Technically, a `Transformer` implements a method `transform()`, which converts one `DataFrame` into another, generally by appending one or more columns. For example:

- A feature transformer might take a `DataFrame`, read a column (e.g., text), map it into a new column (e.g., feature vectors), and output a new `DataFrame` with the mapped column appended.
- A learning model might take a `DataFrame`, read the column containing feature vectors, predict the label for each feature vector, and output a new `DataFrame` with predicted labels appended as a column.

Estimators

An `Estimator` abstracts the concept of a learning algorithm or any algorithm that fits or trains on data. Technically, an `Estimator` implements a method `fit()`, which accepts a `DataFrame` and produces a `Model`, which is a `Transformer`. For example, a learning algorithm such as `LogisticRegression` is an `Estimator`, and calling `fit()` trains a `LogisticRegressionModel`, which is a `Model` and hence a `Transformer`.

Properties of pipeline components

`Transformer.transform()`s and `Estimator.fit()`s are both stateless. In the future, stateful algorithms may be supported via alternative concepts.

Each instance of a `Transformer` or `Estimator` has a unique ID, which is useful in specifying parameters (discussed below).

Pipeline

In machine learning, it is common to run a sequence of algorithms to process and learn from data. E.g., a simple text document processing workflow might include several stages:

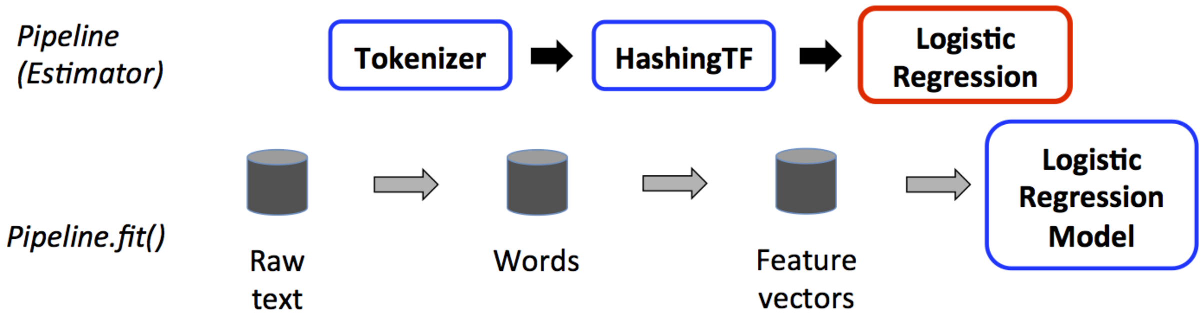
- Split each document's text into words.
- Convert each document's words into a numerical feature vector.
- Learn a prediction model using the feature vectors and labels.

MLlib represents such a workflow as a `Pipeline`, which consists of a sequence of `PipelineStage`s (`Transformer`s and `Estimator`s) to be run in a specific order. We will use this simple workflow as a running example in this section.

How it works

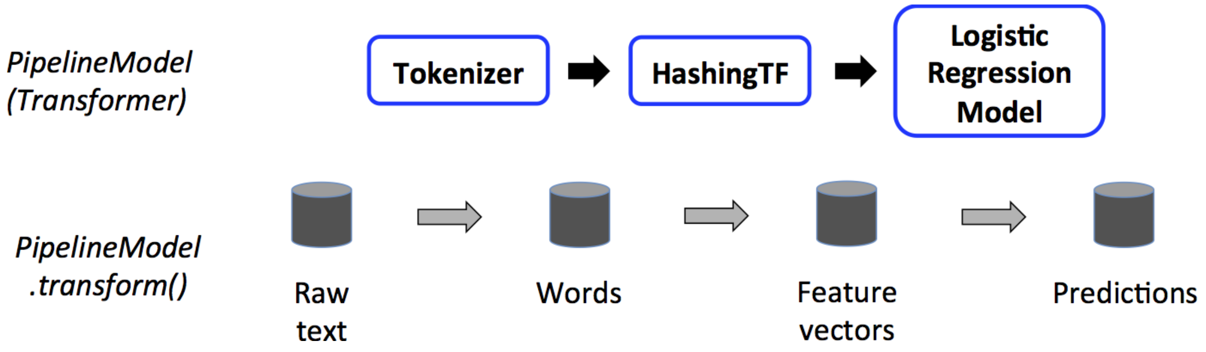
A `Pipeline` is specified as a sequence of stages, and each stage is either a `Transformer` or an `Estimator`. These stages are run in order, and the input `DataFrame` is transformed as it passes through each stage. For `Transformer` stages, the `transform()` method is called on the `DataFrame`. For `Estimator` stages, the `fit()` method is called to produce a `Transformer` (which becomes part of the `PipelineModel`, or fitted `Pipeline`), and that `Transformer`'s `transform()` method is called on the `DataFrame`.

We illustrate this for the simple text document workflow. The figure below is for the *training time* usage of a `Pipeline`.



Above, the top row represents a `Pipeline` with three stages. The first two (`Tokenizer` and `HashingTF`) are `Transformer`s (blue), and the third (`LogisticRegression`) is an `Estimator` (red). The bottom row represents data flowing through the pipeline, where cylinders indicate `DataFrame`s. The `Pipeline.fit()` method is called on the original `DataFrame`, which has raw text documents and labels. The `Tokenizer.transform()` method splits the raw text documents into words, adding a new column with words to the `DataFrame`. The `HashingTF.transform()` method converts the words column into feature vectors, adding a new column with those vectors to the `DataFrame`. Now, since `LogisticRegression` is an `Estimator`, the `Pipeline` first calls `LogisticRegression.fit()` to produce a `LogisticRegressionModel`. If the `Pipeline` had more `Estimator`s, it would call the `LogisticRegressionModel`'s `transform()` method on the `DataFrame` before passing the `DataFrame` to the next stage.

A `Pipeline` is an `Estimator`. Thus, after a `Pipeline`'s `fit()` method runs, it produces a `PipelineModel`, which is a `Transformer`. This `PipelineModel` is used at *test time*; the figure below illustrates this usage.



In the figure above, the `PipelineModel` has the same number of stages as the original `Pipeline`, but all `Estimator`s in the original `Pipeline` have become `Transformer`s. When the `PipelineModel`'s `transform()` method is called on a test dataset, the data are passed through the fitted pipeline in order. Each stage's `transform()` method updates the dataset and passes it to the next stage.

`Pipeline`s and `PipelineModel`s help to ensure that training and test data go through identical feature processing steps.

Details

DAG `Pipeline`s: A `Pipeline`'s stages are specified as an ordered array. The examples given here are all for linear `Pipeline`s, i.e., `Pipeline`s in which each stage uses data produced by the previous stage. It is possible to create non-linear `Pipeline`s as long as the data flow graph forms a Directed Acyclic Graph (DAG). This graph is currently specified implicitly based on the input and output column names of each stage (generally specified as parameters). If the `Pipeline` forms a DAG, then the stages must be specified in topological order.

Runtime checking: Since `Pipeline`s can operate on `DataFrame`s with varied types, they cannot use compile-time type checking. `Pipeline`s and `PipelineModel`s instead do runtime checking before actually running the `Pipeline`. This type checking is done using the `DataFrame` *schema*, a description of the data types of columns in the `DataFrame`.

Unique Pipeline stages: A `Pipeline`'s stages should be unique instances. E.g., the same instance `myHashingTF` should not be inserted into the `Pipeline` twice since `Pipeline` stages must have unique IDs. However, different instances `myHashingTF1` and `myHashingTF2` (both of type `HashingTF`) can be put into the same `Pipeline` since different instances will be created with different IDs.

Parameters

MLlib `Estimator`s and `Transformer`s use a uniform API for specifying parameters.

A `Param` is a named parameter with self-contained documentation. A `ParamMap` is a set of (parameter, value) pairs.

There are two main ways to pass parameters to an algorithm:

1. Set parameters for an instance. E.g., if `lr` is an instance of `LogisticRegression`, one could call `lr.setMaxIter(10)` to make `lr.fit()` use at most 10 iterations. This API resembles the API used in `spark.mllib` package.
2. Pass a `ParamMap` to `fit()` or `transform()`. Any parameters in the `ParamMap` will override parameters previously specified via setter methods.

Parameters belong to specific instances of `Estimator`s and `Transformer`s. For example, if we have two `LogisticRegression` instances `lr1` and `lr2`, then we can build a `ParamMap` with both `maxIter` parameters specified: `ParamMap(lr1.maxIter -> 10, lr2.maxIter -> 20)`. This is useful if there are two algorithms with the `maxIter` parameter in a `Pipeline`.

Saving and Loading Pipelines

Often times it is worth it to save a model or a pipeline to disk for later use. In Spark 1.6, a model import/export functionality was added to the Pipeline API. Most basic transformers are supported as well as some of the more basic ML models. Please refer to the algorithm's API documentation to see if saving and loading is supported.

Code examples

This section gives code examples illustrating the functionality discussed above. For more info, please refer to the API documentation (Scala, Java, and Python).

Example: Estimator, Transformer, and Param

This example covers the concepts of `Estimator`, `Transformer`, and `Param`.

Scala **Java** **Python**

Refer to the `Estimator` Python docs, the `Transformer` Python docs and the `Params` Python docs for more details on the API.

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.classification import LogisticRegression

# Prepare training data from a list of (label, features) tuples.
```

```
training = spark.createDataFrame([
    (1.0, Vectors.dense([0.0, 1.1, 0.1])),
    (0.0, Vectors.dense([2.0, 1.0, -1.0])),
    (0.0, Vectors.dense([2.0, 1.3, 1.0])),
    (1.0, Vectors.dense([0.0, 1.2, -0.5]))], ["label", "features"])

# Create a LogisticRegression instance. This instance is an Estimator.
lr = LogisticRegression(maxIter=10, regParam=0.01)
# Print out the parameters, documentation, and any default values.
print("LogisticRegression parameters:\n" + lr.explainParams() + "\n")

# Learn a LogisticRegression model. This uses the parameters stored in lr.
model1 = lr.fit(training)

# Since model1 is a Model (i.e., a transformer produced by an Estimator),
# we can view the parameters it used during fit().
# This prints the parameter (name: value) pairs, where names are unique IDs for this
# LogisticRegression instance.
print("Model 1 was fit using parameters: ")
print(model1.extractParamMap())

# We may alternatively specify parameters using a Python dictionary as a paramMap
paramMap = {lr.maxIter: 20}
paramMap[lr.maxIter] = 30  # Specify 1 Param, overwriting the original maxIter.
paramMap.update({lr.regParam: 0.1, lr.threshold: 0.55})  # Specify multiple Params.

# You can combine paramMaps, which are python dictionaries.
paramMap2 = {lr.probabilityCol: "myProbability"}  # Change output column name
paramMapCombined = paramMap.copy()
paramMapCombined.update(paramMap2)

# Now learn a new model using the paramMapCombined parameters.
# paramMapCombined overrides all parameters set earlier via lr.set* methods.
model2 = lr.fit(training, paramMapCombined)
print("Model 2 was fit using parameters: ")
print(model2.extractParamMap())

# Prepare test data
test = spark.createDataFrame([
    (1.0, Vectors.dense([-1.0, 1.5, 1.3])),
    (0.0, Vectors.dense([3.0, 2.0, -0.1])),
    (1.0, Vectors.dense([0.0, 2.2, -1.5]))], ["label", "features"])

# Make predictions on test data using the Transformer.transform() method.
# LogisticRegression.transform will only use the 'features' column.
# Note that model2.transform() outputs a "myProbability" column instead of the usual
# 'probability' column since we renamed the lr.probabilityCol parameter previously.
prediction = model2.transform(test)
result = prediction.select("features", "label", "myProbability", "prediction") \
    .collect()

for row in result:
    print("features=%s, label=%s -> prob=%s, prediction=%s"
          % (row.features, row.label, row.myProbability, row.prediction))
```

Find full example code at "examples/src/main/python/ml/estimator_transformer_param_example.py" in the Spark repo.

Example: Pipeline

This example follows the simple text document `Pipeline` illustrated in the figures above.

»

Scala **Java** **Python**

Refer to the `Pipeline` Python docs for more details on the API.

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)

# Prepare test documents, which are unlabeled (id, text) tuples.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    rid, text, prob, prediction = row
    print("(%d, %s) --> prob=%s, prediction=%f" % (rid, text, str(prob), prediction))
```

Find full example code at "examples/src/main/python/ml/pipeline_example.py" in the Spark repo.