PySpark MLlib Pipelines

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 `Transformers` and `Estimators` together to specify an ML workflow.
- **`Parameter`**: All `Transformers` and `Estimators` now share a common API for specifying parameters.

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.

Each instance of a `Transformer` or `Estimator` has a unique ID, which is useful in specifying parameters

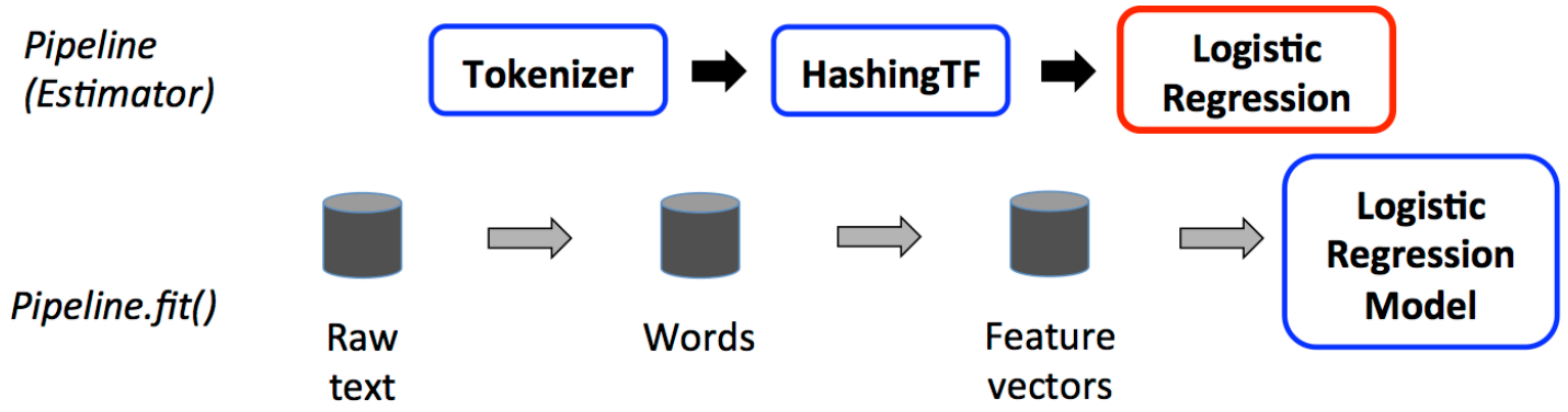
Estimators

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`.

Pipeline

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`.

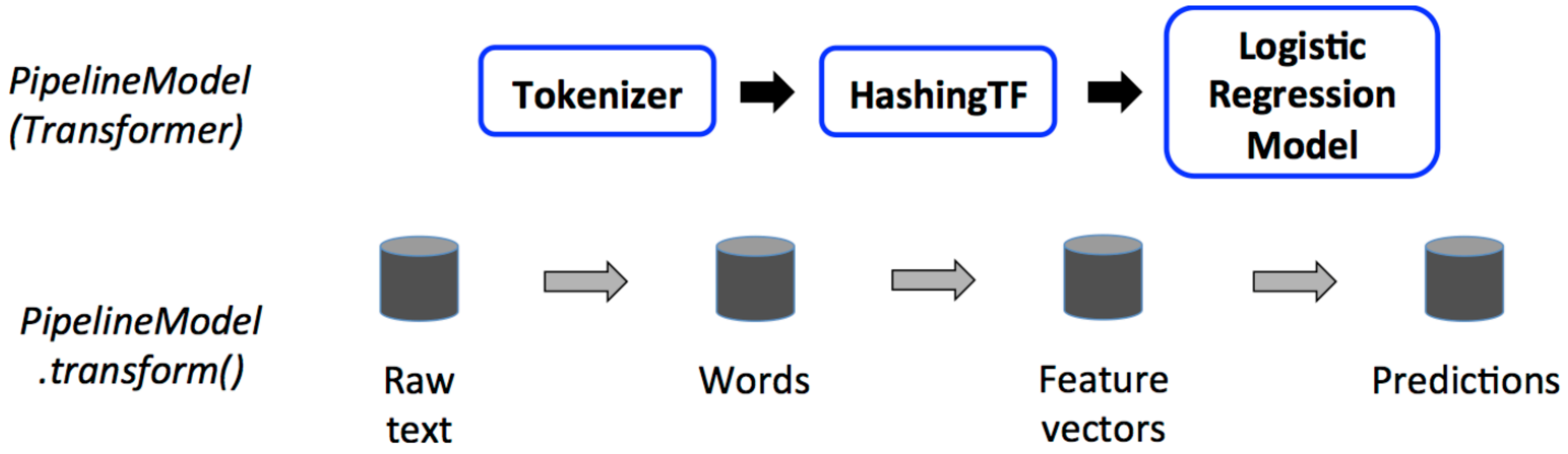
Example: Pipeline



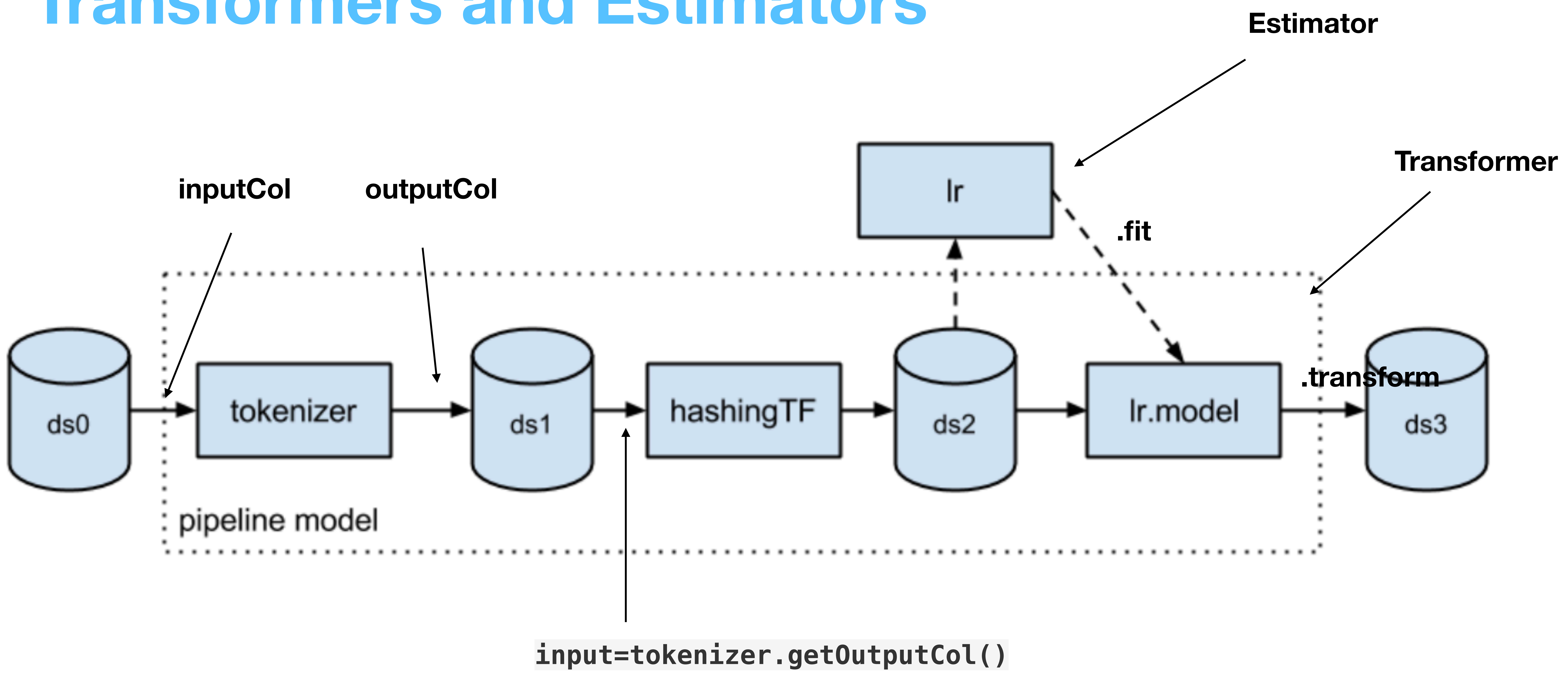
`Tokenizer` and `HashingTF` are `Transformers`
`LogisticRegression` is an `Estimator`

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*

Example: Test


PipelineModel (Transformer)
Tokenizer
HashingTF
Logistic Regression Model
PipelineModel .transform()
Raw text
Words
Feature vectors
Predictions

Transformers and Estimators



```
input=tokenizer.getOutputCol()
```

Stages and Pipeline

```
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)

#model is now a transformer
```

Pipeline Fit and Test

```
# 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)
```

StopWordsRemover and ngram Transformer

```
from pyspark.ml.feature import StopWordsRemover

sentenceData = spark.createDataFrame([
    (0, ["I", "saw", "the", "red", "balloon"]),
    (1, ["Mary", "had", "a", "little", "lamb"])
], ["id", "raw"])

remover = StopWordsRemover(inputCol="raw", outputCol="filtered")
remover.transform(sentenceData).show(truncate=False)
```

```
from pyspark.ml.feature import NGram

wordDataFrame = spark.createDataFrame([
    (0, ["Hi", "I", "heard", "about", "Spark"]),
    (1, ["I", "wish", "Java", "could", "use", "case", 
"classes"]),
    (2, ["Logistic", "regression", "models", "are", "neat"])
], ["id", "words"])

ngram = NGram(n=2, inputCol="words", outputCol="ngrams")

ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.select("ngrams").show(truncate=False)
```

Parameter Tuning and Cross validation

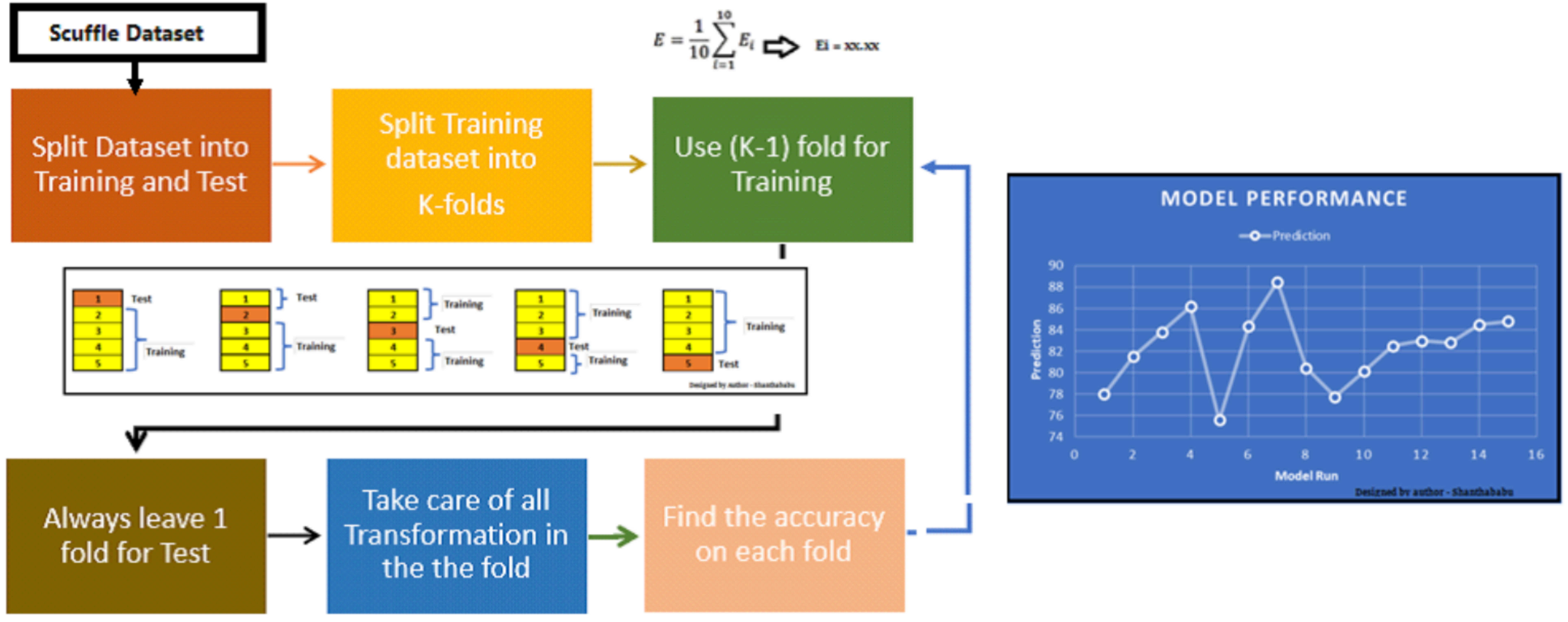
```python
paramGrid = ParamGridBuilder() \
    .addGrid(hashingTF.numFeatures, [10, 100, 1000]) \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .build()

crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(),
                          numFolds=2)  # use 3+ folds in practice

# Run cross-validation, and choose the best set of parameters.
cvModel = crossval.fit(training)

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

# Make predictions on test documents. cvModel uses the best model found (lrModel).
prediction = cvModel.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    print(row)
```


Scuffle Dataset
Split Dataset into Training and Test
Split Training dataset into K-folds
Use (K-1) fold for Training
$E = \frac{1}{10}\sum_{i=1}^{10} E_i$
Ei = xx.xx
Test
Training
Designed by author - Shanthababu
Always leave 1 fold for Test
Take care of all Transformation in the the fold
Find the accuracy on each fold
MODEL PERFORMANCE
Prediction
Model Run
Designed by author - Shanthababu

K Fold Cross Validation

Digned by author -Shanthababu

Train Validation Split

```python
# We use a ParamGridBuilder to construct a grid of parameters to search over.
# TrainValidationSplit will try all combinations of values and determine best model using
# the evaluator.
paramGrid = ParamGridBuilder()\
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.fitIntercept, [False, True])\
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
    .build()

# In this case the estimator is simply the linear regression.
# A TrainValidationSplit requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.
tvs = TrainValidationSplit(estimator=lr,
                           estimatorParamMaps=paramGrid,
                           evaluator=RegressionEvaluator(),
                           # 80% of the data will be used for training, 20% for validation.
                           trainRatio=0.8)

# Run TrainValidationSplit, and choose the best set of parameters.
model = tvs.fit(train)

# Make predictions on test data. model is the model with combination of parameters
# that performed best.
model.transform(test)\
    .select("features", "label", "prediction")\
    .show()
```

Collaborative Filtering (Alt. Least Squares)

```python
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row

lines = spark.read.text("data/mllib/als/sample_movielens_ratings.txt").rdd
parts = lines.map(lambda row: row.value.split("::"))
ratingsRDD = parts.map(lambda p: Row(userId=int(p[0]), movieId=int(p[1]),
                                     rating=float(p[2]), timestamp=int(p[3])))
ratings = spark.createDataFrame(ratingsRDD)
(training, test) = ratings.randomSplit([0.8, 0.2])

# Build the recommendation model using ALS on the training data
# Note we set cold start strategy to 'drop' to ensure we don't get NaN evaluation metrics
als = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating",
          coldStartStrategy="drop")
model = als.fit(training)

# Evaluate the model by computing the RMSE on the test data
predictions = model.transform(test)
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
                                predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print("Root-mean-square error = " + str(rmse))

# Generate top 10 movie recommendations for each user
userRecs = model.recommendForAllUsers(10)
# Generate top 10 user recommendations for each movie
movieRecs = model.recommendForAllItems(10)

# Generate top 10 movie recommendations for a specified set of users
users = ratings.select(als.getUserCol()).distinct().limit(3)
userSubsetRecs = model.recommendForUserSubset(users, 10)
# Generate top 10 user recommendations for a specified set of movies
movies = ratings.select(als.getItemCol()).distinct().limit(3)
movieSubSetRecs = model.recommendForItemSubset(movies, 10)
```

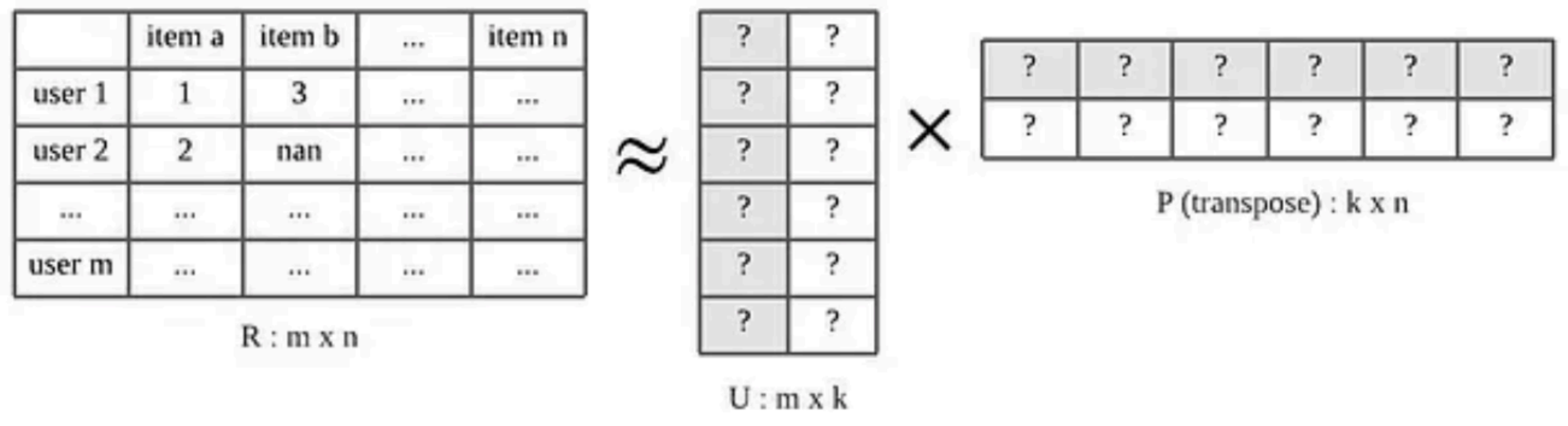


Figure 2. Matrix Factorization

DAG Pipelines

A Pipeline’s stages are specified as an ordered array.

It is possible to create non-linear Pipelines 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.

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.