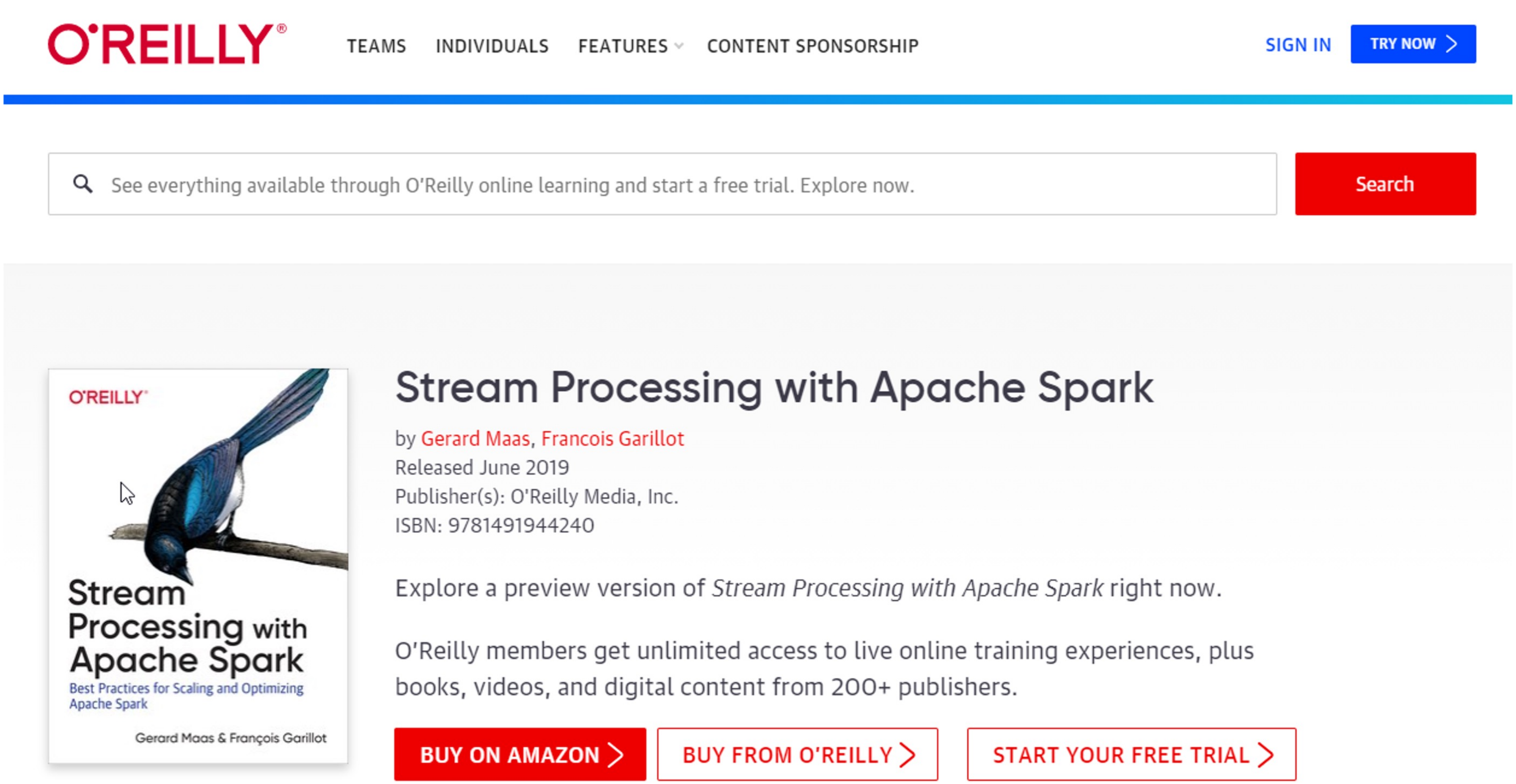
Data 603 – Big Data Platforms

UMBC

Lecture 9

Apache Spark MLlib - Part 1

More on Structured Streaming



Machine Learning

- Process for extracting patterns from the data **using statistics, linear algebra, and numerical optimization**.
- Types of machine learning:
  - Supervised
  - Semi-supervised
  - Unsupervised
  - Reinforcement Learning

Supervised Learning

- Data consists of a set of input records each with **associated labels.**
  - The goal is to predict the output label(s) given a new unlabeled input
  - The output labels can either be discrete or continuous.
- Classification
  - Separate the inputs into a discrete set of classes (labels).
  - Binary classes: two discrete values
  - Multiclass/Multinomial classification: three or more discrete labels.
- Regression
  - The value to predict is a continuous number

Supervised Learning Supported by Spark MLlib

Table 10-1. Popular classification and regression algorithms

Algorithm	Typical usage
Linear regression	Regression
Logistic regression	Classification (we know, it has regression in the name!)
Decision trees	Both
Gradient boosted trees	Both
Random forests	Both
Naive Bayes	Classification
Support vector machines (SVMs)	Classification

Unsupervised Learning

- Obtaining the labeled data required by supervised machine learning can be very expensive and/or infeasible.
  - https://www.mturk.com/
- Instead of predicting a label, **unsupervised ML helps to understand the structure of the data.**
- Can be used for outlier detection or as a preprocessing step for supervised machine learning
  - Reducing the dimensionality of the data set
  - K-means
  - Latent Dirichlet Allocation (LDA)
  - Gaussian mixture models

Spark Machine Learning

- Provides an ecosystem for data ingestion, feature engineering, model training and deployment
- Traditionally, developers needed to use different tools for each of these tasks.
- Spark has two machine learning packages
  - spark.mllib
    - Original machine learning API based on RDD API (in maintenance mode since Spark 2.0)
  - spark.ml
    - Newer API based on DataFrames.
- “Mllib” is used as an umbrella term to refer to both machine learning packages.
- spark.ml focuses on O(n) scale-out

ML Pipelines

https://spark.apache.org/docs/latest/ml-pipeline.html

ML Pipelines

- Pipeline = Workflow
- MLlib standardizes APIs for machine learning algorithms into a single pipeline.
- The concept of the pipeline is inspired by scikit-learn.

ML Pipelines

- DataFrame: 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: 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 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.

Machine Learning Pipelines

MLlib Terminology

- **DataFrame**
  - ML can be applied to a wide variety of data types including vectors, text, images, and structured data.
  - DataFrame can use ML Vector types
  - For Python, MLlib recognizes the following types as dense vectors:
    - NumPy's array <- More efficient
    - Python's list, e.g., [1, 2, 3]
  - and the following as sparse vectors:
    - MLlib's SparseVector.
    - SciPy's csc_matrix with a single column

- https://spark.apache.org/docs/latest/mllib-data-types.html#local-vector
- https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.mllib.linalg.Vectors.html

Machine Learning Pipelines

MLlib Terminology

- **Transformer**
  - Accepts a DataFrame as input, returns a new DataFrame with one or more columns appended to it.
  - An abstraction that includes feature transformers and learned models.
    - A feature transformer reads a column, 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.
  - Do not learn any parameters from the data.
    - Simply applies rule-based transformations to either prepare data for model training or generate predictions using a trained MLlib model.
  - .transform() method

Machine Learning Pipelines

MLlib Terminology

- **Estimator**
  - Abstracts the concept of a learning algorithm.
  - Learns (or "fits") parameters from the DataFrame
  - Returns a Model, which is a transformer
  - .fit() method
    - Accepts a DataFrame and produces a Model.
    - e.g. LogisticRegression is an Estimator, and calling fit() trains a LogisticRegressionModel, which is a Model and hence a Transformer

Machine Learning Pipelines

MLlib Terminology

- **Pipeline**
  - Created to support the common pattern in ML to run a sequence of algorithms to process and learn from the data
    - E.g. Split each document's text into words -> convert each document's words into a numerical feature vector -> learn a prediction model using feature vectors and labels
  - Provides a high-level API built on top of DataFrames to organize the machine learning workflow.
  - Composed of a series of transformers and estimators
  - A Pipeline consists of a sequence of PipelineStages (Transformers and Estimators) to be run in a specific order.
  - Pipelines are estimators.
  - Pipeline.fit() returns a PipelineModel, a transformer.

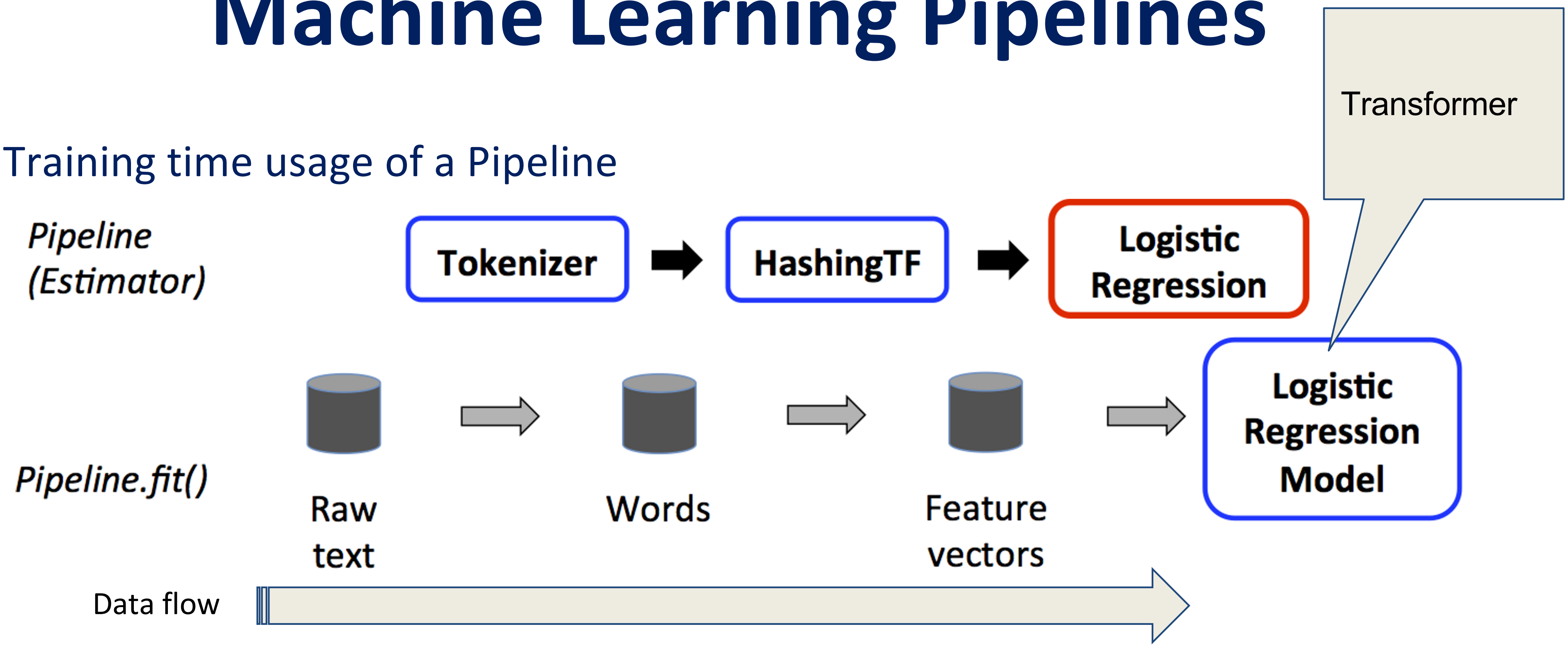
Machine Learning Pipelines

MLlib Terminology

- **Pipeline (Cont.)**
  - A Pipeline is a sequence of stages
    - Each stage is either a Transformer or an Estimator
    - Stages are run in order
    - An input DataFrame is transformed as it passes through each stage

Machine Learning Pipelines

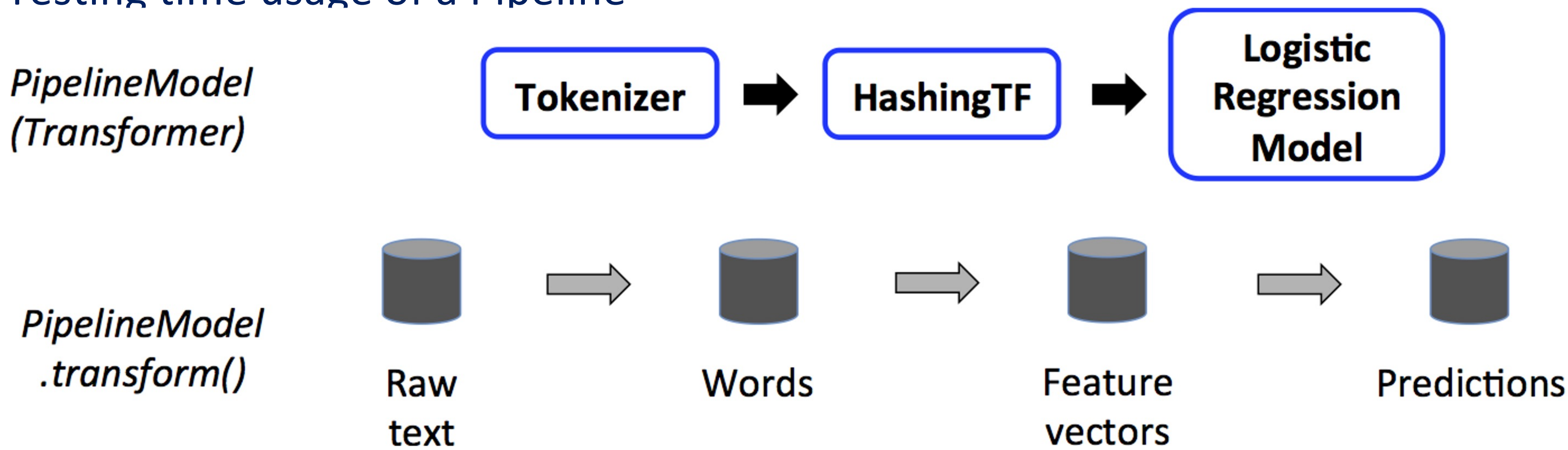
Training time usage of a Pipeline



- Three stage pipeline
- Tokenizer and HasingTF are Transformers
- LogisticRegression is an Estimator
- After a Pipeline's fit() method runs, a PipelineModel (a Transformer) is produced

Machine Learning Pipelines

Testing time usage of a Pipeline



- All Estimators in the original Pipeline have become Transformers
- PipelineModel.transform() triggers data to be passed through the fitted pipeline in order
- Each stage’s transform() updates the dataset and passes it to the next stage

Training and Test Data Sets

https://spark.apache.org/docs/latest/ml-pipeline.html

Training and Test Data Sets

Dividing the data

- Standard convention 80/20 (train/test) split
- Reason for not using the whole data set:
  - It is possible for the model to "memorize", or "overfit", the training data.
  - We need generalized model.
  - The model's performance on the test set is a proxy for how well it will perform on unseen data with similar distributions.
- Define a seed value for reproducibility.
  - Rerunning the code will get the same data points going to the train and testing sets. Only guaranteed with same numbers of executors.
  - Split data once and write it to its own train/test folder to reduce the reproducibility issues.

Training and Test Data Sets

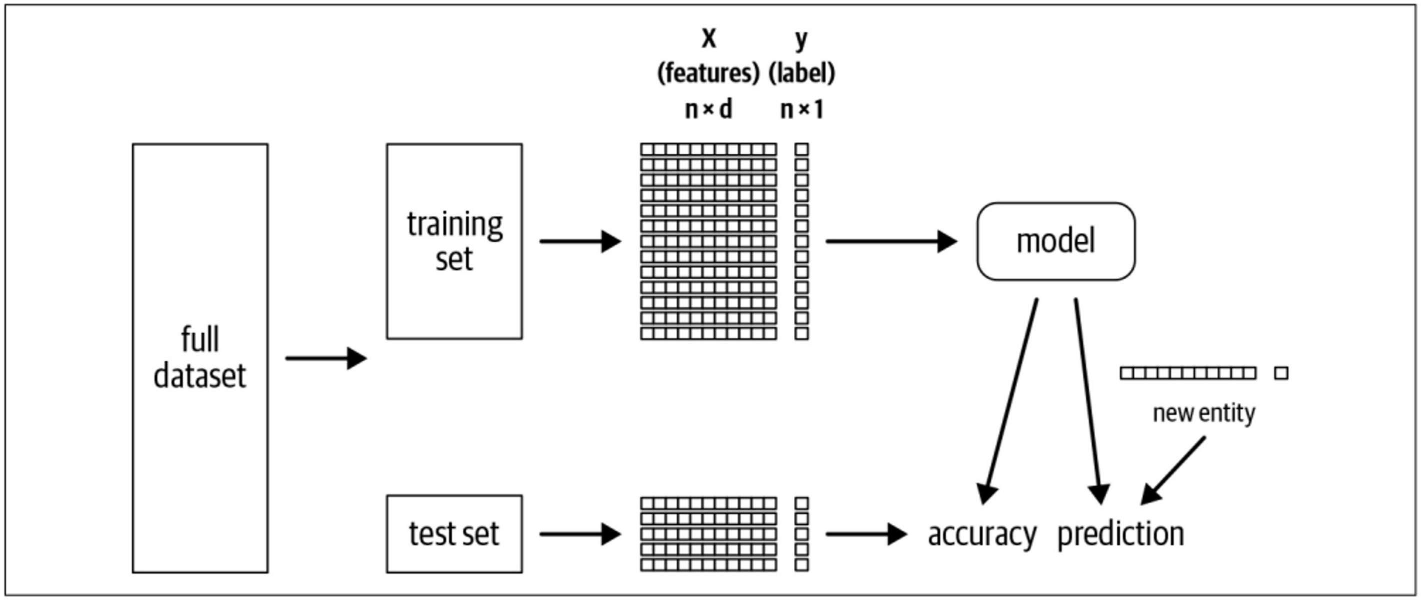


Figure 10-5. Train/test split

Training Set:

- A set of features, *X*, and a label, *y*
- *X* is a *n x d* matrix, *n* = number of data points (rows, examples), *d* is the number of features (fields, columns)
- *y*, denotes a *n x 1* vector. For every example (row), there is one label

Preparing Features with Transformers

Linear regression requires all the input features are contained within a single vector in a DataFrame.

- The data needs to be transformed.

Spark Transformers

- Accept a DataFrame as input and return a new DataFrame with one or more columns appended.
- They do not learn from the data
- They apply rule-based transformations using the transform() method.

Preparing Features with Transformers

VectorAssembler transformer

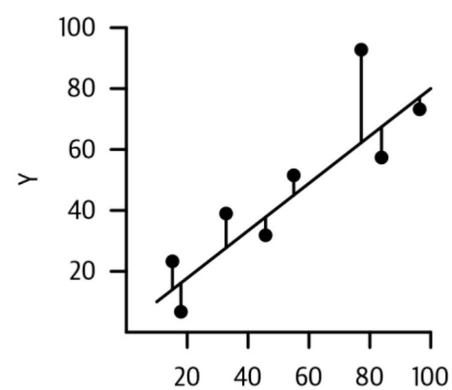
- Used to put all features into a single vector
- Takes a list of input columns and creates a new DataFrame with an additional column (features).
- Combines the values of the input columns into a single vector.

```
from pyspark.ml.feature import VectorAssembler
vecAssembler = VectorAssembler(inputCols=["bedrooms"],
outputCol="features")
vecTrainDF = vecAssembler.transform(trainDF)
vecTrainDF.select("bedrooms", "features", "price").show(10)
```

Linear Regression

Linear regression models a linear relationship between the dependent variable (label) and one or more independent variables (features).

- Linear regression seeks to fit an equation for a line to x and y.
  - y= mx +b, m is the slope, b is the offset (intercept).
  - y_hat = b_0 + b_1*x_1 + … + b_n*x_n + error
  - … + b_1_2 * x_1*x_2
- The goal of linear regression is to find a line that minimizes the square of the residuals (errors between the model predictions and the true values).



Using Estimators to Build Models 1/2

Spark's LinearRegression is a type of estimator

- It takes a DataFrame and returns a Model.
- Estimators learn parameters from the data.
- It has a fit() method. The output is a transformer.
- Estimators are eagerly evaluated (kick off Spark jobs). Transformers are lazily evaluated.
- Other examples of estimators
  - Imputer, DecisionTreeClassifier, RandomForestRegressor, etc.

```
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(featuresCol="features",
labelCol="price")
lrModel = lr.fit(vecTrainDF) # returns a
LinearRegressionModel
```

Using Estimators to Build Models 2/2

- Once the estimator has learned the parameters, the transformer can apply the parameters to new data points to generate prediction
- Inspecting the parameters LinearRegression estimator learned:

```
m = round(lrModel.coefficients[0], 2)
b = round(lrModel.intercept, 2)
print(f"""The formula for the linear regression line is
price = {m}*bedrooms + {b}""")
```

Creating a Pipeline 1/2

- Running a sequence of algorithms to process and learn from data includes several stages.
- Spark ML represents such a workflow as a Pipeline
- A Pipeline is specified as a sequence of stages
- 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.
- Pipeline API provides better code reusability and organization.
- In Spark, *Pipelines* are estimators, whereas *PipelineModels*—fitted Pipelines—are transformers.

```
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[vecAssembler, lr])
pipelineModel = pipeline.fit(trainDF)
```

Creating a Pipeline 2/2

- Pipeline API determines which stages are estimators or transformers eliminating the need to explicitly call fit() vs. transform() for each of the stages.
- Using *pipelineModel*, it is possible to apply it to the test data set.

```
predDF = pipelineModel.transform(testDF)
predDF.select("bedrooms", "features", "price", "prediction").show(10)
```

One-hot Encoding 1/4

- Most machine learning models in MLlib expect numerical values as input, represented as vectors
- To convert categorical values into numeric values, a technique called one-hot encoding (OHE) is used.
- The numeric values used should not introduce any relationships to the data set.
  - Instead, a separate column for each distinct value is desired.

```
"Dog" = [ 1, 0, 0]
"Cat" = [ 0, 1, 0]
"Fish" = [0, 0, 1]
```

  - This does the same thing as pandas.get_dummies()

One-hot Encoding 2/4

- The concern with memory consumption for data sets with potentially a large number of categories.
  - Spark internally uses a SparseVector when the majority of the entries are 0 (case with OHE).
  - It does not waste space storing 0 values.
- DenseVector vs SparseVector

  DenseVector(0, 0, 0, 7, 0, 2, 0, 0, 0, 0)

  SparseVector(10, [3, 5], [7, 2])
- SparseVector keeps track of the size of the vector (10), the indices of the nonzero elements ([3,5]), and the corresponding values at those indices ([7,2]).

One-hot Encoding 3/4

Ways to create one-hot encode

- Use the StringIndexer and OneHotEncoder
  - Apply the StringIndexer estimator to convert categorical values into category indices
    - The category indices are ordered by label frequencies – most frequent label gets index 0.
  - Once the categories indices are created, they are passed as input to the OneHotEncoder
    - OneHotEncoder maps a column of category indices to a column of binary vectors.

One-hot Encoding 4/4

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer
categoricalCols = [field for (field, dataType) in trainDF.dtypes
                   if dataType == "string"]

indexOutputCols = [x + "Index" for x in categoricalCols]
oheOutputCols = [x + "OHE" for x in categoricalCols]
stringIndexer = StringIndexer(inputCols=categoricalCols,
                              outputCols=indexOutputCols,
                              handleInvalid="skip")

oheEncoder = OneHotEncoder(inputCols=indexOutputCols,
                           outputCols=oheOutputCols)

numericCols = [field for (field, dataType) in trainDF.dtypes
               if ((dataType == "double") & (field != "price"))]

assemblerInputs = oheOutputCols + numericCols
vecAssembler = VectorAssembler(inputCols=assemblerInputs,
                               outputCol="features")
```

Feature Selection - RFormula

Another way to select features - using RFormula

- Syntax is inspired by the R programming language.
- The label and the features to be included are provided
- It supports a limited subset of the R operators including (~, ., :, +,and -).
- Examples:
  - formula = "y ~ bedrooms + bathrooms", predict y given just bedrooms and bathrooms,
  - formula = "y ~ .", use all of the available features (and automatically excludes y from the features).
  - y ~ bedrooms:bathrooms + bedrooms + bathrooms

Feature Selection - RFormula

Good

- RFormula will automatically StringIndex and OHE all string columns, convert numeric columns to double type, and combine all of these into a single vector using VectorAssembler under the hood.

Look out!

- OHE is not required or recommended for all algorithms.
- Tree-based methods do not need OHE categorical features.
  - It will often make the tree-based models worse.

Evaluating the Model

Evaluating Models - RMSE

In spark.ml, there are evaluators for classification, regression, clustering and ranking.

- Root Mean-Square Error (RMSE) – commonly used regression metric
  - Ranges from zero to infinity

Important!
Compare the RMSE against a baseline

$$\text{Root Mean Squared Error (RMSE)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

```
from pyspark.ml.evaluation import RegressionEvaluator
regressionEvaluator = RegressionEvaluator(
    predictionCol="prediction",
    labelCol="price",
    metricName="rmse")
rmse = regressionEvaluator.evaluate(predDF)
print(f"RMSE is {rmse:.1f}")
```

Evaluating Models – Coefficient of Determination (R^2)

- R^2 values range from negative infinity to 1.

$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$	R^2
$SS_{tot} = \sum_{i=1}^{n} (y_i - \bar{y})^2$	SS_{tot} is the total sum of squares when predicting y bar (average value)
$SS_{res} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$	Sum of Squared Errors. 0 when the model perfectly predicts every data points

- When $SS_{res} = 0$, then $R^2 = 1$
- If $SS_{res} = Ss_{tot}$, R^2 is 0, the model performs the same as always predicting the average value, y bar.
- If SS_{res} is large (it performs worse than predicting the average, y bar), then R^2 is going to be negative.

```
r2 = regressionEvaluator.setMetricName("r2").evaluate(predDF)
```

Additional Reading

- Learning Spark V2 Chapter 10
- https://spark.apache.org/docs/latest/ml-guide.html

Lab

https://docs.databricks.com/applications/machine-learning/train-model/mllib/index.html#apache-spark-mllib-pipelines-and-structured-streaming-example

Questions