

U

O

W

Machine Learning with Apache Spark

– Handling Massive Data Set (Part II)

CSCI316: Big Data Mining Techniques and Implementation



UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Large-Scale Machine Learning

- We know MR and Spark model is powerful for distributed computation (e.g., SQL and matrix operation), but how about machine learning?



Machine Learning Setting

- Ingredients of a learning algorithm [Goodfellow et al. 2016]
 - Task (i.e., classification, regression, clustering, etc.)
 - Performance measure (e.g., accuracy, MSE, etc.)
 - Experience (i.e., dataset w/o target values)
- The ML landscape is large. We consider:
 - a generic learning technique called *gradient descent*
 - We use *linear regression* as an example, but most ML models can be trained via gradient descent.

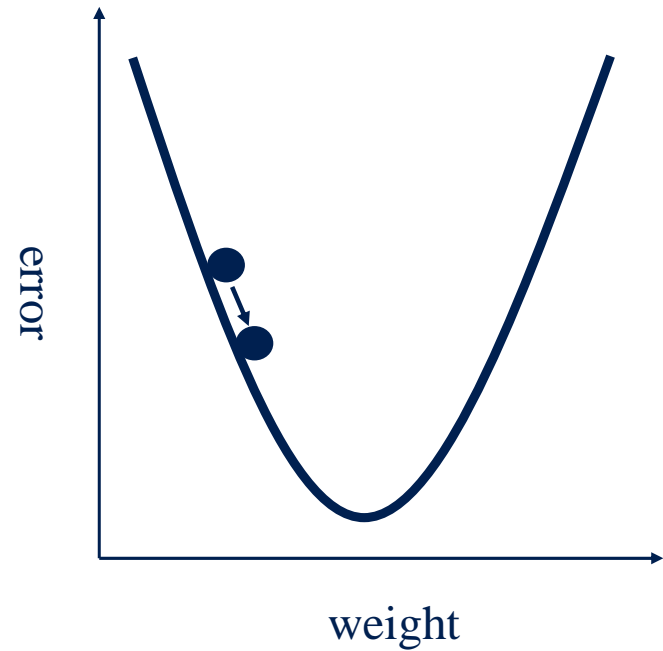
Linear Regression

- Linear regression builds a *linear function* that maps the data to the predicted values.
- We use vectors (called *feature vector*) to represent features of data.
- Given a d -dimensional vector \mathbf{x} , the *predicted value* is $\hat{y} = \mathbf{w}^T \mathbf{x}$ where \mathbf{w} is a vector of parameters. Let y denote the *true value*.
 - \mathbf{w}^T refers to a transpose of \mathbf{w} (i.e., column vector \rightarrow row vector)
 - If fixing the first element in \mathbf{x} as 1, then we obtain the form $\hat{y} = \mathbf{w}'^T \mathbf{x}' + b$ where \mathbf{w}' is the weights and b is the bias
- For a set of m samples, we have a $d \times m$ matrix \mathbf{X} (i.e., \mathbf{X} includes all feature vectors) and a vector $1 \times m$ vector \mathbf{y} (i.e. \mathbf{y} includes all true values).
- The mean squared error is $MSE = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 = \frac{1}{m} \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$.
- The goal of training a linear regressor is finding some \mathbf{w} that minimises the MSE.



Gradient Descent

- Start at a random point
- Repeat
 - Determine a descent direction
 - Choose a step size
 - Update
- Until stopping criterion is satisfied

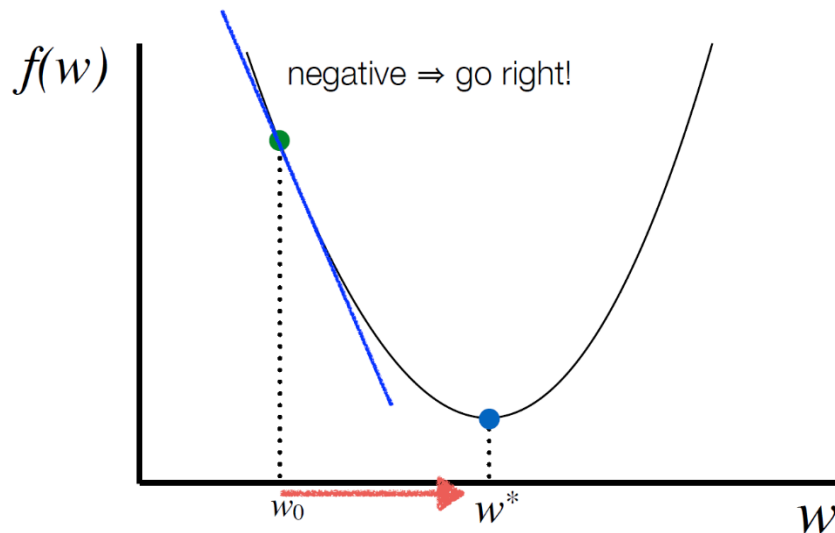
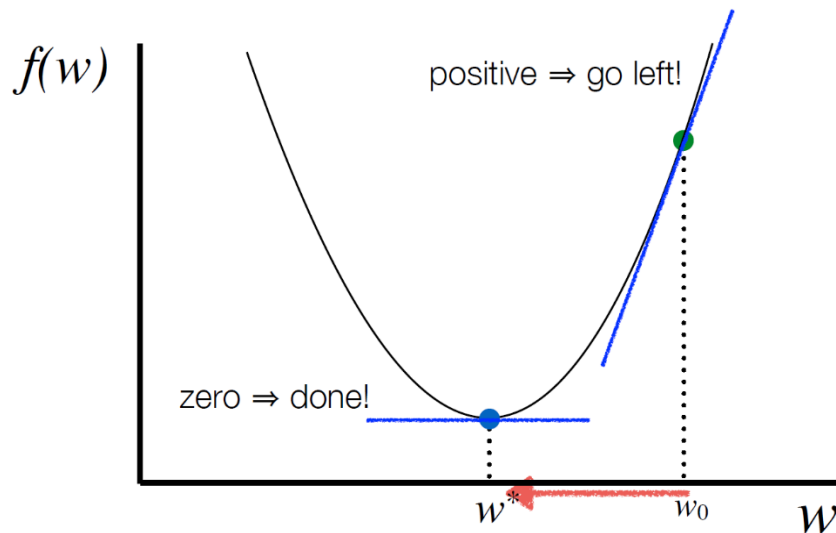


Direction of Descent - Slope

- Know the error function, for example, $f(w) = w^2$
- Then, $\frac{df(w)}{dw} = 2 \cdot w \Rightarrow -2$

Note. If f has one variable only, we write d (the standard derivative) rather than ∂ (the partial derivative).

- Move in the *opposite direction* of the gradient.



Descent Direction and Magnitude (1D)

- The opposition direction of the slope points in the direction of steepest error descent in the weight space

$$w_{i+1} = w_i - \alpha_i \frac{df}{dw} (w_i)$$

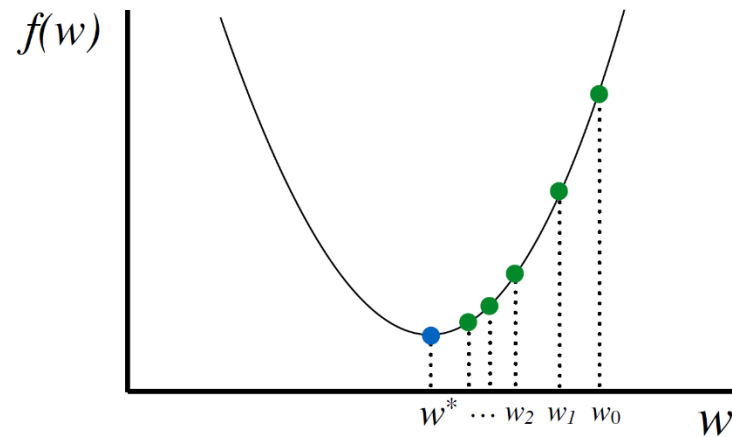
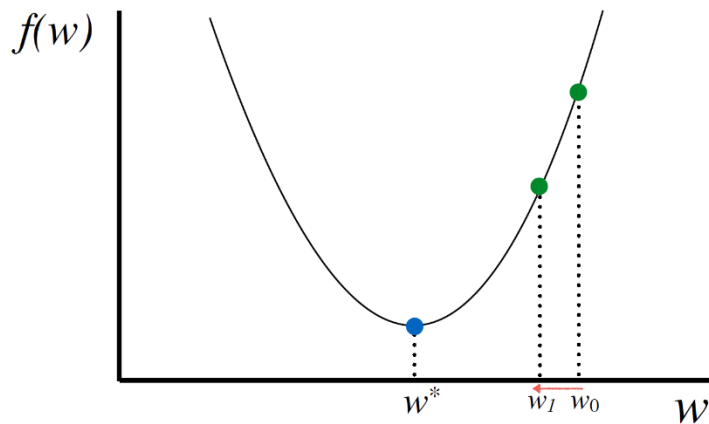
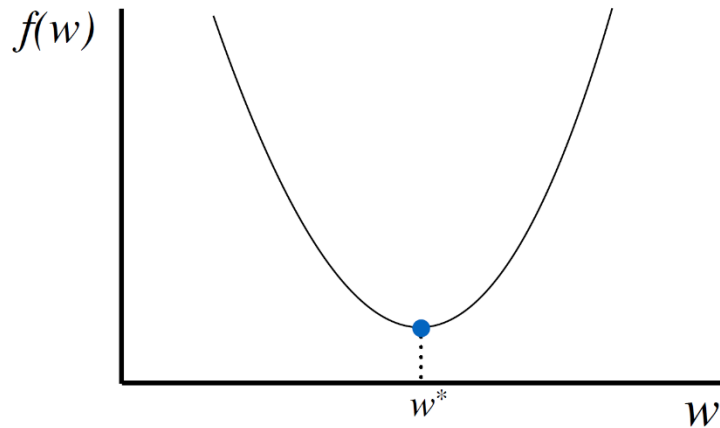
step size

negative slope

i refers to the i -th step (or epoch) in the gradient decent

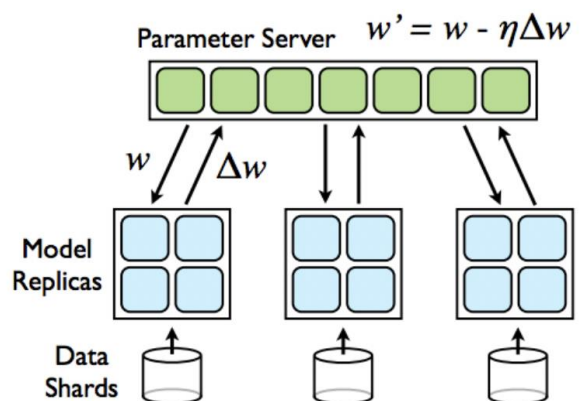
- Step size is a free parameter that has to be chosen carefully for each problem.
 - It can be (and usually is) updated dynamically during the iteration.

Descent Direction and Magnitude (1D)

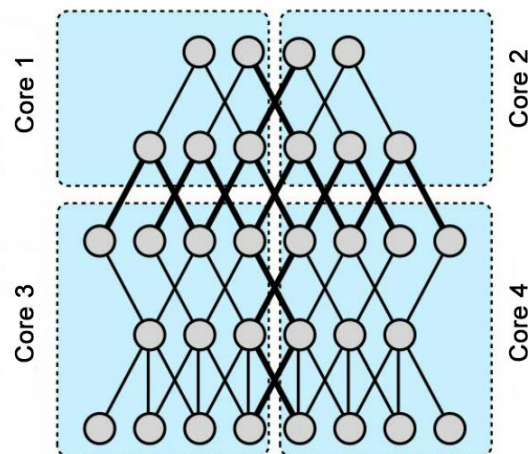


Large-Scale ML

- Recall that ML updates a model \mathcal{M} based on data \mathbf{X}
 - For linear regression (with MSE as the cost function, without regularisation), the model is expressed by the vector \mathbf{w} .
- Big data: the input data \mathbf{X} is too large to hold in the main memory
- Big model: the model \mathcal{M} is too large to hold in the main memory
- Data parallelism* and *model parallelism*



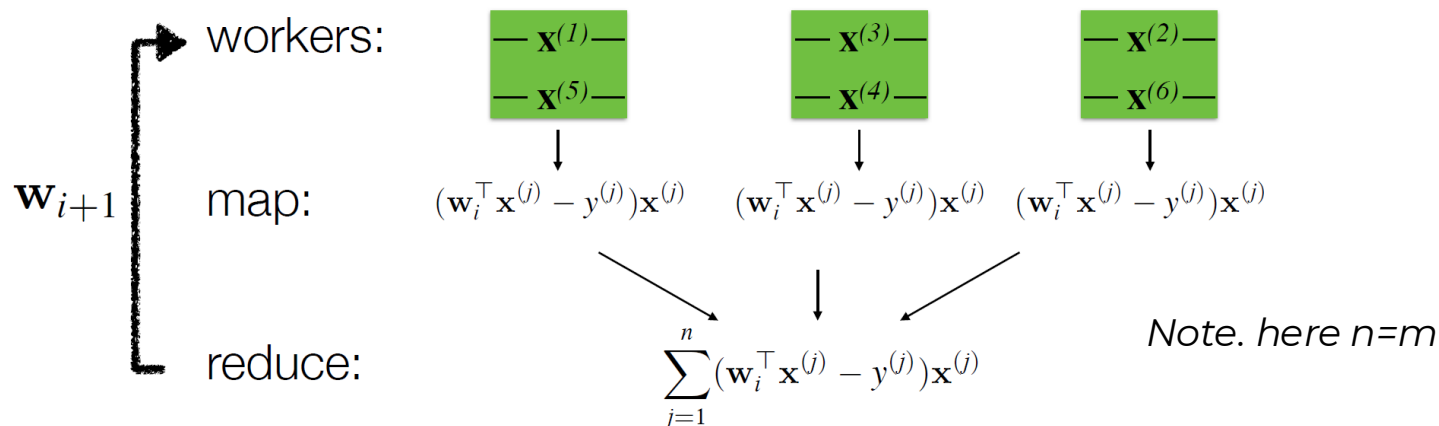
Data parallelism



Model parallelism

Gradient Descent: Big m, Big d

- Weight update in Linear Regression: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j=1}^m (\mathbf{w}_i^T \mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$
- By *data parallelism*, we compute summands in parallel on workers receiving all \mathbf{w}_i at every iteration.
- For example, let $m = 6$ and the number of works is 3:



- Bottleneck: transferring \mathbf{w}_i between the driver (or parameter server) and the workers.

Stochastic Gradient Descent (SGD)

- Recall gradient descent for linear regression (with MSE), every iteration processes all samples (m is the total number of samples):
- $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j=1}^m (\mathbf{w}_i^T \mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$
- The gradient is an *expectation* that can be approximated using a small set of samples drawn uniformly *at random* from the whole data set.
 - In particular, if the data set is highly redundant – the gradient in the first half will be very similar to that in the second half.
- In SGD, we update the model with only one sample instead of m .
- SGD can improve the training speed and mitigate the large data issue
- Practice also shows that SGD helps the algorithm jump out of local minima and find the global minima.

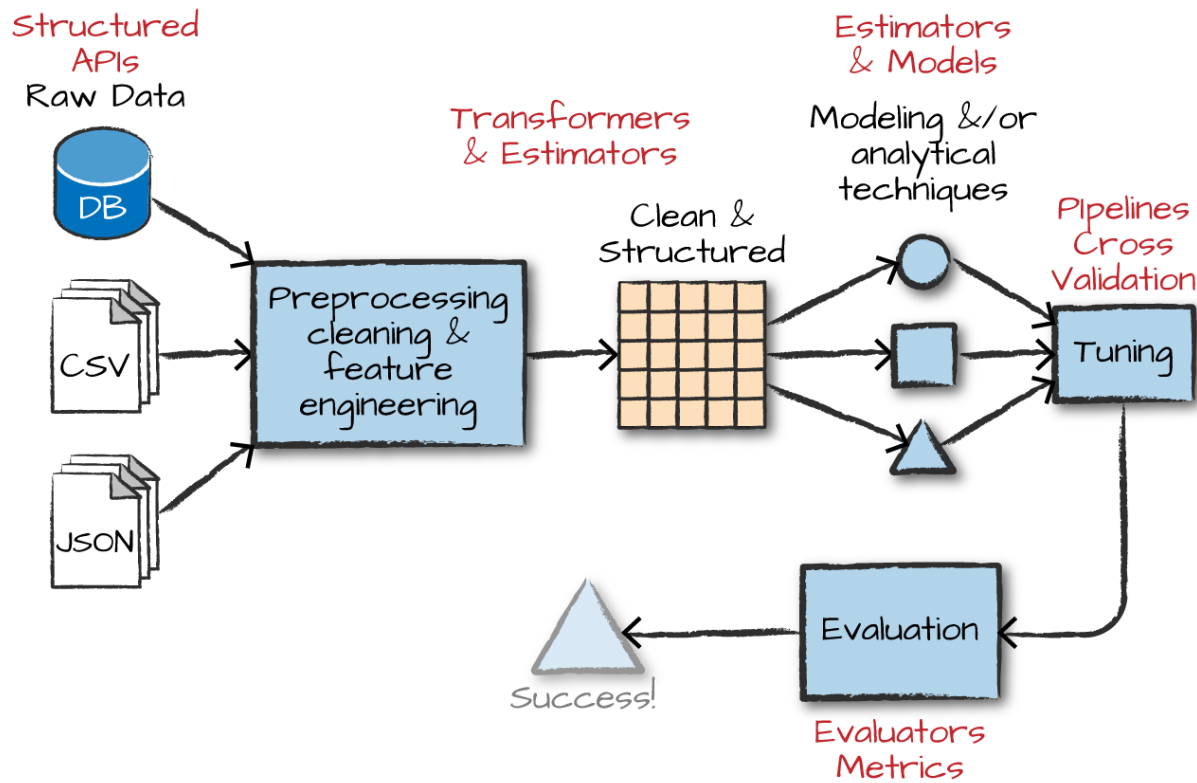
Minibatch Gradient Descent

- Increase the batch size from 1 to a smaller number than m .
- Divide the data set into small batches of examples, compute the gradient using a single batch, makes an update, then move to the next batch of examples.
- E.g., a small batch of 16: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j=1}^{16} (\mathbf{w}_i^T \mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$
- Computing the gradient simultaneously using the matrix-matrix multiplications which are efficient, especially on GPUs
- Benefit: more stable than SGD
- For classification, ideally mini-batches need to be balanced for classes (e.g., using stratified sampling)

Spark MLlib – The Machine Learning Library of Spark

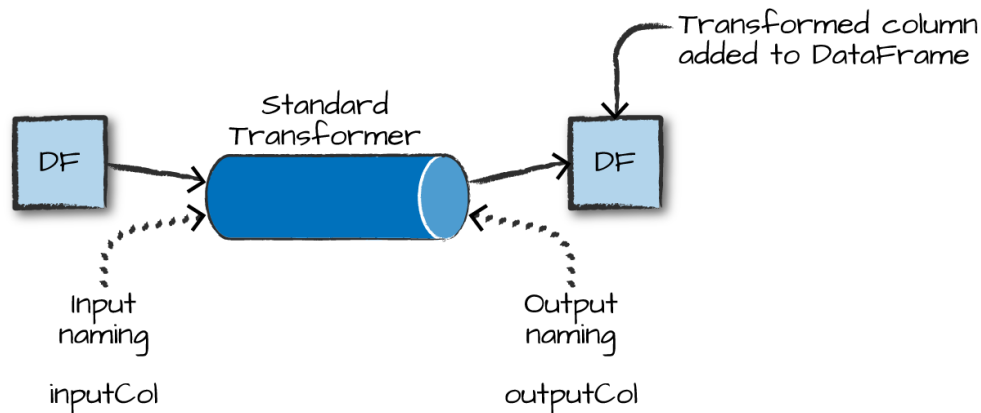
High-Level MLlib Concepts

- Spark's MLlib is conceptually similar to Scikit-Learn, but leverages Spark's powerful distributed computing engine.



High-Level MLlib Concepts

- Transformers
 - functions that convert raw data in some way.
 - E.g., to create a new interaction variable, to convert string categorical values into numerical values
 - primarily used in pre-processing and feature engineering
 - takes a DataFrame as input and produces a new DataFrame as output



High-Level MLlib Concepts

- Estimators
 - if provided with data, result in transformers
 - algorithms that are used to train models
- Evaluators
 - evaluate how a given model performs according to criteria (e.g., accuracy, ROC)
- Pipeline
 - MLlib's highest-level data type
 - Transformers, estimators and evaluators are all *stages* in a pipeline
 - Similar to Scikit-Learn's pipeline API

MLlib in Action

- **Example 1: a (synthetic) dataset**

```
df = spark.read.json(dataset)
```

```
df.orderBy("value2").show(5)
```

```
+-----+-----+-----+-----+
|color| lab|value1|          value2|
+-----+-----+-----+-----+
|  red|good|    35|14.386294994851129|
| blue|bad|    12|14.386294994851129|
|  red|bad|     2|14.386294994851129|
| blue|bad|     8|14.386294994851129|
|  red|bad|    16|14.386294994851129|
+-----+-----+-----+-----+
```

only showing top 5 rows

- a categorical label with two values (good or bad), a categorical variable (colour), and two numerical variables.

MLlib in Action

- Feature Engineering with Transformers
 - As mentioned, transformers manipulate existing columns in and add new columns to a DataFrame
 - In MLlib, all inputs to ML algorithms in Spark must consist of type `Double` (for a label) and `Vector[Double]` (for features).
 - Note. our synthetic dataset does not meet this requirement
 - RFormula: a declarative language for specifying ML transformers and is simple to use (supports a limited subset of the R operators):
 - ~ Separate target and terms
 - + Contact terms
 - Remove terms
 - : Interaction
 - . All columns except the target/dependent variable

MLlib in Action

- In our case, we use all variables and also add in the interactions between some columns.

```
from pyspark.ml.feature import RFormula
supervised = RFormula(formula="lab ~ . + color:value1 +
color:value2")
```

- The next step is to fit and apply the RFormula transformer to data
 - Just call the `fit` and `transform` methods of an RFormula instance:

```
fittedRF = supervised.fit(df)
preparedDF = fittedRF.transform(df)
preparedDF.show(4)
```

New columns, which are to
be fed into ML algorithms

color	lab	value1	value2	features	label
green	good	1	14.386294994851129	(10, [1, 2, 3, 5, 8], [...]	1.0
blue	bad	8	14.386294994851129	(10, [2, 3, 6, 9], [8....]	0.0

MLlib in Action

- Create a simple test set based on a sample split of the data
`train, test = preparedDF.randomSplit([0.7, 0.3])`
- Fit a model (we choose a **decision tree** classifier)
 - set the label columns and the feature columns; the column names—label and features—are actually the default labels.

```
from pyspark.ml.classification
    import DecisionTreeClassifier
dt = DecisionTreeClassifier(labelCol="label",
    featuresCol="features")
```

- Kick off a Spark job to train the model:

```
fittedLR = dt.fit(train)
```

MLlib in Action

- Apply the model to the training dataset and see the prediction:

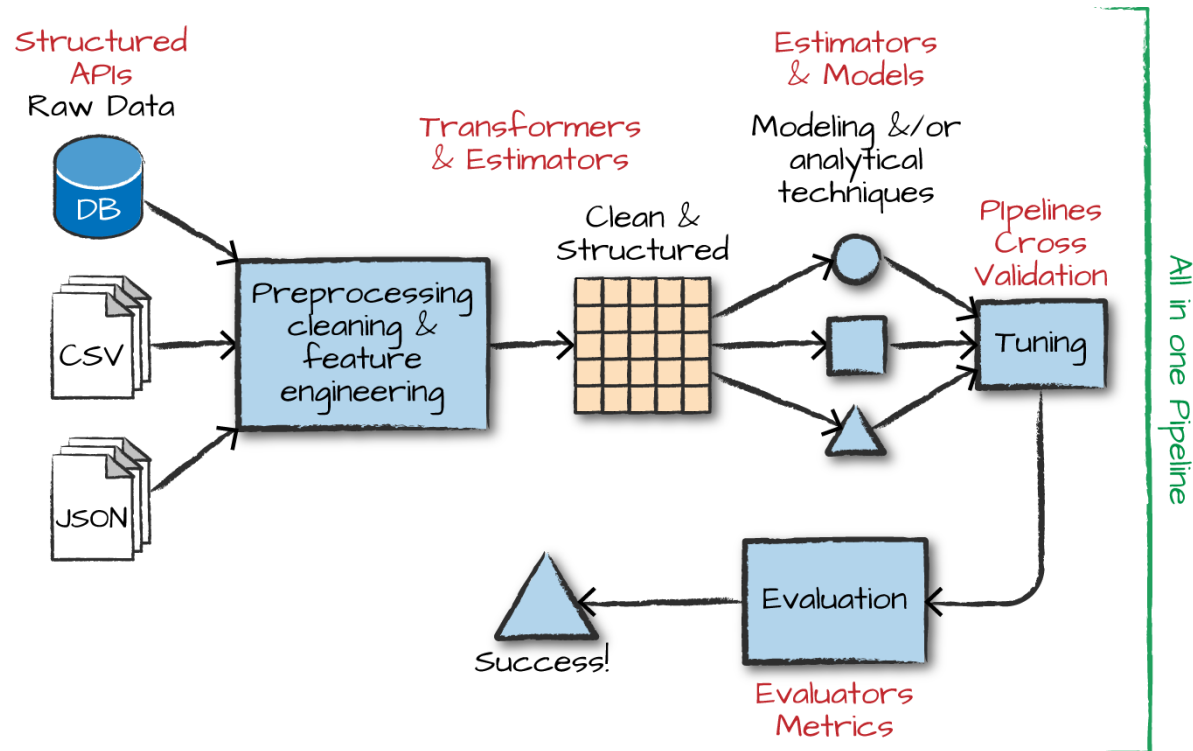
```
fittedLR.transform(train).select("label",  
    "prediction").show()
```

```
+-----+-----+  
|label|prediction|  
+-----+-----+  
|  0.0 |      0.0 |  
|   ... |      ... |  
|  1.0 |      1.0 |  
|   ... |      ... |
```

- Next, we can evaluate this model and calculate the performance metrics (e.g., TP rate and FN rate)
 - In practice, we also need to try out difference combinations of *model hyperparameters*
- Use the Pipeline interface to save the manual effort on model selection and hyperparameter tuning.

MLlib in Action

- Pipelining steps in an advanced analytics workflow
 - The Pipeline interface allows to set up a dataflow of a sequence of related operations that ends with an estimator.



MLlib in Action

- To build a pipeline, first split the data based on the *original* dataset `df` (not `preparedDF`)
`train, test = df.randomSplit([0.7, 0.3])`
- Recall that each stage in a Pipeline may be a transformer or an estimator.
- There are two estimators in our case: for `RFormula` and the logistic regression classifier.

```
rForm = RFormula()
dt = DecisionTreeClassifier() \
    .setLabelCol("label") \
    .setFeaturesCol("features")
from pyspark.ml import Pipeline
stages = [rForm, dt]
pipeline = Pipeline().setStages(stages)
○ A “logical” pipeline is built in the last step.
```

Can also use other ML APIs, e.g.,
`DecisionTreeClassifier`, `NaiveBayes`, see
<https://spark.apache.org/docs/latest/ml-classification-regression.html>

MLlib in Action

- Training and Evaluation

- Train several variations of the model by specifying different combinations of the hyperparameters
- MLlib provides a ParamGridBuilder class for this purpose:

```
from pyspark.ml.tuning import ParamGridBuilder
params = ParamGridBuilder()\
    .addGrid(rForm.formula, [
        "lab ~ . + color:value1",
        "lab ~ . + color:value1 + color:value2"])\
    .addGrid(dt.maxDepth, [2, 3, 4])\
    .addGrid(dt.maxBins, [4, 5]).build()
```

□ In the above example code, we have selected:

- 2 versions of RFormula
- 3 options for the maxDepth parameter
- 2 options for the maxBins parameters

More about the elastic net and regularization parameters in the next lecture

Can call `dt.explainParams()` to retrieve a list of hyperparameters

MLlib in Action

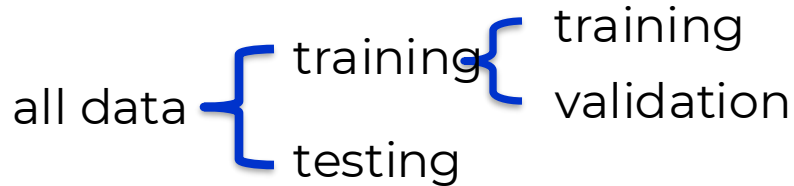
- Thus, we want to evaluate a total of 12 different combinations of parameters
- To determine which combination is optimal, use an *evaluator* and an *evaluation metric*.
 - We choose `BinaryClassificationEvaluator` and the `areaUnderROC` metric.

```
from pyspark.ml.evaluation import  
    BinaryClassificationEvaluator  
evaluator = BinaryClassificationEvaluator() \  
    .setMetricName("areaUnderROC") \  
    .setRawPredictionCol("prediction") \  
    .setLabelCol("label")
```

- To actually perform the evaluation that determines a best model that we train, some kind of *validation* is needed

MLlib in Action

- Dataset splits:



- The validation and testing usually can share common methods.
 - split the training dataset into two different groups (used below)
 - perform K-fold cross-validation, etc.

```
from pyspark.ml.tuning import TrainValidationSplit
tvsv = TrainValidationSplit() \
    .setTrainRatio(0.75) \ #how to split training set
    .setEstimatorParamMaps(params) \
    .setEstimator(pipeline) \
    .setEvaluator(evaluator)
```

MLlib in Action

- We are ready to train the model:

```
tvsfitted = tvs.fit(train)
```

```
type(tvsfitted)
```

```
Out: pyspark.ml.tuning.TrainValidationSplitModel
```

- Finally, we can test it with our holdout dataset

```
evaluator.evaluate(tvsfitted.transform(test))
```

```
Out: 1.0
```

Or:

```
evaluator.evaluate(tvsfitted.bestModel.transform(test))
```

```
Out: 1.0
```

MLlib in Action

- Usually, a single pipeline in Spark includes one ML algorithm (as an estimator).
 - If you want to imply multiple competing ML algorithms (e.g., a DT classifier and a LR classifier), you may need to specify multiple pipeline manually.
- Persisting and Applying Models
 - To save the model (to facilitate future usage),
`tvsvFitted.bestModel.write().save("path...")`
 - To load the model

```
from pyspark.ml.pipeline import PipelineModel  
myModel = PipelineModel.load("path...")
```

Summary

- Large-Scale ML
 - Gradient Descent in distributed computation
 - Linear regression as an example
- Spark MLlib